# Streaming-based CNN Architecture for Physical Layer Wireless Receivers: Deployment-Oriented Dataset Generation and Quantisation Methods

**Andrew Georgios Maclellan**

Supervisors: Dr. Louise Crockett
Prof. Bob Stewart

StrathSDR Research Laboratory
Department of Electronic and Engineering
University of Strathclyde

This thesis is submitted for the degree of
*Doctor of Philosophy*

September 2025

Signed: Andrew Georgios Maclellan
Date: September 2025

# Abstract

With the rising demand for efficient wireless connectivity, Artificial Intelligence (AI) has become a key enabler for enhancing wireless radio system performance. Intelligent models support cognitive decision-making and real-time processing, enabling low-latency, edge-capable systems for Physical Layer (PHY) wireless communications tasks. While AI models perform well on Graphic Processing Unit (GPU)-based systems, edge deployment can significantly reduce latency and increase throughput, enabling operation in constrained environments.

This thesis presents the development and evaluation of a streaming-based Convolutional Neural Network (CNN) accelerator for Software Defined Radio (SDR) receivers operating on live, real-time signals. Designed from first principles using a synchronous dataflow model, the architecture is purpose-built to align with the continuous dataflow of FPGA-based SDR pipelines, enabling per-sample processing without data loss. The accelerator is implemented on the AMD Zynq UltraScale+ Radio Frequency System-on-Chip (RFSoC) platform, demonstrating low-latency operation (29.6 $\mu s$) and high-throughput (34k classifications per second) real-time operation in PHY layer tasks.

To support real-time deployment of CNN models on live signals captured by the ADC, this thesis introduces the DeepRFSoC dataset generation methodology, which enables training on realistic loopback data affected by simulated channel impairments and hardware-specific distortions. This methodology enables the accelerator to operate in real-time on the SDR platform processing live signals as they are captured.

A quantisation investigation is presented, comparing Post-Training Quantisation (PTQ) and Quantisation-Aware Training (QAT) under real-time live signal reception conditions on the AMD RFSoC. Results show that 8-bit QAT models can outperform their floating-point counterparts by 3%, while 4-bit and 2-bit models maintain competitive accuracy with only a 2% reduction, demonstrating the viability of quantised CNN models for real-time PHY-layer inference on edge FPGA-based SDR platforms, contributing to the development of intelligent, low-latency SDR systems.

# Acknowledgements

Firstly, I would like to thank my supervisors, Dr. Louise Crockett and Prof. Bob Stewart, for giving me the opportunity to pursue a PhD and the many opportunities, unwavering support, and direction they provided throughout the completion of this project. I feel very fortunate to have such dedicated and supportive supervisors.

I would like to express my gratitude to my friends and colleagues at the StrathSDR research lab at the University. You are all very dedicated, talented individuals and it has been a great privilege to work alongside you. The experience both in and out of the lab made this experience all the more rewarding. I'd like to give special thanks to Marius, Lewis, and Blair for their friendship and support throughout the years.

Thanks to Garrey Rice for the multiple internship opportunities at the MathWorks Glasgow office. I am grateful to have worked with such a passionate and skilled team, and for the valuable experience that helped shape my research and career.

Thanks to Graham Schelle and Patrick Lysaght for the internship opportunity at AMD. It was a pleasure to work with the PYNQ and AUP teams and the experience provided important skills that supported my PhD work.

I would like to thank my parents, my siblings, and my friends for their unwavering support and encouragement over the years. This has been a long and sometimes challenging PhD journey and I am incredibly lucky to be surrounded by such kind-hearted and supportive people.

Finally, I'd like to thank my partner, Annamae, for her patience, love, constant support, and encouragement throughout the years. Completing our PhDs together has been an incredible and great experience and I am truly grateful we were able to share this journey side by side.

This work was supported by the Engineering and Physical Sciences Research Council (EPSRC) [EP/R513349/1].

# Table of Contents

# List of Figures

# List of Tables

# Acronyms

**8PSK** 8 Phase Shift Keying

**Adam** Adaptive Moment Estimation

**AI** Artificial Intelligence

**ALRT** Average Likelihood Ratio Test

**AM** Analogue Modulation

**AM-DSB** Amplitude Modulation Double Sideband

**AM-SSB** Amplitude Modulation Single Sideband

**AMC** Automatic Modulation Classification

**ANN** Artificial Neural Network

**API** Application Programming Interface

**APU** Application Processing Unit

**ASIC** Application-Specific Integrated Circuit

**AUP** AMD University Program

**AWGN** Additive White Gaussian Noise

**AXI4** Advanced Extensible Interface

**BLAS** Basic Linear Algebra Subprograms

**BPSK** Binary Phase Shift Keying

**BRAM** BlockRAM

**CNN** Convolutional Neural Network

**CPFSK** Continuous Phase Shift Keying

**CPU** Central Processing Unit

**DDC** Digital Down-Converter

**DL** Deep Learning

**DMA** Direct Memory Access

**DNN** Deep Neural Network

**DSA** Dynamic Spectrum Access

**DSP** Digital Signal Processing

**DT** Decision Tree

**DUC** Digital Up-Converter

**FB** Feature-Based

**FC** Fully-Connected

**FF** Flip-Flop register

**FIR** Finite Impulse Response

**FM** Frequency Modulation

**FPGA** Field Programmable Gate-Array

**FSM** Finite State Machine

**GeMM** General Matrix Multiplication

**GFSK** Gaussian Frequency Shift Keying

**GLRT** Generalised Likelihood Ratio Test

**GPIO** General Purpose Input Output

**GPU** Graphics Processing Unit

**Gsps** Giga samples per second

**GUI** Graphical User Interface

**HDL** Hardware Description Language

**HLRT** Hybrid Likelihood Ratio Test

**HLS** High Level Synthesis

**ILA** Integrated Logic Analyser

**IoT** Internet of Things

**IP** Intellectual Property

**IPI** IP Integrator

**ISI** Inter-Symbol Interference

**KNN** k-Nearest Neighbours

**LB** Likelihood-Based

**LNA** Low Noise Amplifier

**LUT** Look-Up Table

**M-FSK** M-ary Frequency Shift Keying

**M-PAM** M-ary Pulse Amplitude Modulation

**M-PSK** M-ary Phase Shift Keying

**MAC** Multiply-Accumulate

**ML** Machine Learning

**MLE** Maximum Likelihood Estimator

**MLP** Multi-Layer Perceptron

**MMIO** Memory Mapped Input Output

**Msps** Mega samples per second

**MVM** Matrix-to-Vector Multiplication

**NPU** Neural Processing Unit

**OS** Operating System

**PAM4** 4-level Pulse Amplitude Modulation

**PDF** Probability Density Function

**PE** Processing Element

**PHY** Physical Layer

**PL** Programmable Logic

**PLL** Phase-Locked Loop

**PMU** Platform Management Unit

**PS** Processing System

**PTQ** Post-Training Quantisation

**QAM** Quadrature Amplitude Modulation

**QAT** Quantised-Aware Training

**QNN** Quantised Neural Network

**QPSK** Quadrature Phase Shift Keying

**RC** Raised Cosine

**ReLU** Rectified Linear Unit

**RF** Radio Frequency

**RF-ADC** RF Analogue to Digital Converter

**RF-DAC** RF Digital to Analogue Converter

**RFDC** RF Data Converter

**RFSoC** Radio Frequency System-on-Chip

**RPU** Real-time Processing Unit

**SDF** Synchronous Dataflow

**SDR** Software-Defined Radio

**SMA** SubMinature version A

**SNR** Signal-to-Noise Ratio

**SoC** System-on-Chip

**SPS** Samples-Per-Symbol

**STE** Straight-Through Estimator

**SVM** Support Vector Machine

**SWG** Sliding Window Generator

**TPU** Tensor Processing Unit

**UML** Unified Modelling Language

**URAM** UltraRAM

**WBFM** Wideband Frequency Modulation

**WoF** Window of Focus

# Chapter 1

# Introduction

Over the last decade, the number of wirelessly connected devices has grown drastically [1]–[3]. With the continuous advancements of wireless standards and significant improvements in data capacity, a wide range of new applications and sectors have adopted wireless technology. Applications include connected homes, smart cities, civilian drones, and the expanding network of satellites. The benefits of a more connected society are widely felt, from instant access to live news, streaming, music, and communications, to the enablement of autonomous vehicles and Internet of Things (IoT) systems. As global connectivity becomes increasingly essential, the demand for more efficient data transfer protocols, spectrum management techniques, and data bandwidth is rising in parallel.

To address the growing demand for more efficient wireless communications to manage the connectivity demand, current radio systems must evolve. This evolution points to radio systems that are capable of adapting dynamically to their environment, managing spectrum more efficiently, and making real-time decisions without human intervention. These radios are called 'intelligent radios'. Intelligent radios are enabled by the power of Artificial Intelligence (AI), which offers ways for machines to 'learn' how to perform some historically challenging tasks well. While previously a task like symbol-to-bit demapping had to be hand crafted by an engineer, a machine can instead learn to decode symbols and even out-perform state-of-the-art techniques under more extreme channel conditions [4].

AI is the theory and development of computer systems able to perform tasks without the need of human intelligence. A subset of AI is Machine Learning (ML), where computers learn patterns and make decisions from data without being explicitly programmed. Traditional ML methods often rely on hand-crafted features, manually selected characteristics of the input data that

help guide learning [5]. Deep Learning (DL) is a type of ML algorithm that uses Artificial Neural Networks (ANNs) and raw input samples, instead of features, to learn from the data. This distinction between learning a task through features and learning a task through the raw samples has led to DL emerging as a powerful tool for solving previously difficult tasks [6].

In computer vision and natural language processing, the abundance of data available has led to huge success in both fields, in tasks like object recognition, image segmentation, language translation, and language models [7], [8]. The leaps in algorithmic advancements in those fields can also be transferred over to other domains, like Physical Layer (PHY) wireless communications [9]. As 6G technologies emerge, and with an ever-increasing number of wirelessly connected devices, DL can be a tool to solve many of the challenging tasks prevalent in wireless communications. Examples include: channel estimation, signal identification, decoding, and synchronisation.

Current and emerging Software-Defined Radio (SDR) receivers, like the AMD Zynq UltraScale+ Radio Frequency System-on-Chip (RFSoC) [10], are improving in their capabilities with higher sampling rates, wider instantaneous bandwidths, and more powerful processing, enabling signal processing algorithms implemented on the devices to be processed at even higher data and sampling rates. Current AI acceleration occurs on dedicated processors, either on the Graphics Processing Units (GPUs), or on Neural Processing Units (NPUs) [11]–[14]. While GPUs and NPUs are excellent for training and accelerating a variety of neural network topologies, respectively, without changing the underlying architecture, they suffer from memory bottlenecking, latency, and throughput limitations depending on the dimensions of the topology, making them difficult to include into a radio receiver pipeline that requires deterministic latencies and fast throughputs [11]. The focus of this thesis is to develop a custom Convolutional Neural Network (CNN) accelerator architecture on a Field Programmable Gate-Array (FPGA) that can be used in line with, or as a replacement for, a traditional signal processing algorithm in a radio receiver to support real-time streaming from the RF Analogue to Digital Converter (RF-ADC).

This chapter will outline the main motivations for the work presented in this thesis. This includes why an application specific CNN accelerator architecture is needed for real-time PHY radio receiver applications, and how this can complement current radio receiver signal processing pipelines without requiring a structural overhaul to facilitate AI model support. In this context, *training*

refers to the offline process of teaching a model using labelled data, while *inference* is the real-time execution of that trained model on new, unseen data typically performed on embedded hardware.

## 1.1 Convolutional Neural Network Accelerators for Wireless Receivers

A typical wireless SDR receiver performs a sequence of signal processing operations to recover transmitted information from received radio signals. These include filtering, mixing to baseband, decimation, synchronisation, demodulation, and decoding, which are usually implemented using deterministic, hand-crafted algorithms optimised to handle adverse channel conditions [15]. However, as wireless environments grow more complex and congested, due to the increasing number and variety of wireless devices sharing the spectrum, traditional signal processing techniques begin to struggle with generalisation, often requiring manual reconfiguration.

As an alternative, there is growing interest in using AI, particularly CNNs, to tackle PHY tasks [16]. CNNs can learn relevant features directly from raw I/Q samples, reducing the need for expert feature engineering and improving adaptability across varying conditions. This shift also enhances productivity by allowing engineers to focus on model design rather than manual tuning.

In other domains that rely on AI, model size varies significantly depending on the task. For example, computer vision applications like object recognition may use models such as Tiny-YOLOv4, which has around 6 million parameters and runs comfortably on consumer-grade GPUs [17]. In natural language processing, models can be substantially larger; for instance, OpenAI's GPT-4 has an estimated 1.76 trillion parameters and requires tens of thousands of high-end GPUs just for inference [18]. These models are trained to generalise across many tasks, including understanding natural language instructions.

By contrast, AI models used for PHY communications do not require this scale. Most tasks can be handled with much smaller models. In this work, a candidate CNN architecture from the RadioML benchmark is used, containing approximately 260,000 parameters [19].

Although DL underpins all of these applications, the scale and nature of models vary widely across domains. Consequently, the deployment strategy should be tailored to the specific application. A general-purpose AI inference architecture designed for large models may not be optimal for smaller, domain-

specific models, particularly in scenarios where throughput and deterministic latency are critical, as is often the case in wireless communications. The combination of smaller model size and strict performance requirements makes CNNs for PHY tasks particularly well-suited for implementation directly on edge devices. Edge devices are computing devices located at the periphery of a network, typically close to the source of data or the end user. They often operate under constrained compute resources and are commonly used in low-power or battery-powered environments [20].

Many radio transceiver systems are deployed on System-on-Chip (SoC) devices and FPGAs. To integrate DL into these systems without disrupting the existing architecture, CNN models must follow a dataflow structure that mirrors the radio pipeline. In such a structure, signal processing operations are represented as nodes in a graph, with output samples from one stage streaming directly into the next [21]. This supports efficient parallel processing and resource utilisation. Dataflow architectures are particularly well-matched to FPGA-based SoCs such as the AMD RFSoC [22], which can be reprogrammed to deploy new neural network structures as needed.

Deploying a CNN model onto the Programmable Logic (PL) of an FPGA or RFSoC/SoC introduces several challenges. These include model compression (to fit weights into on-chip memory), quantisation of weights and feature maps, and architectural optimisations like resource sharing.

This thesis addresses the lack of deterministic, streaming-friendly CNN architectures that operate at radio line-rate and complement the structure of existing radio receiver pipelines. To demonstrate the approach, modulation classification is used as the application case study, as it represents a common AI task in modern radio systems that benefits from real-time, embedded processing [9]. The work also investigates the training and deployment requirements of such models on the AMD RFSoC, including the effects of weight quantisation on real-world performance.

While CNN models are promising for PHY tasks, integrating them into wireless receivers requires overcoming significant deployment challenges, as discussed in the following section.

## 1.2 Challenges of Deploying Deep Learning in Wireless Receivers

DL methods, as shown by [19], have emerged as powerful tools in the wireless communications domain due to their ability to model complex, non-linear systems by training on raw samples of example data, rather than relying on hand-crafted feature extraction techniques. Tasks such as modulation classification, signal identification, decoding, synchronisation, and channel estimation are some of the tasks where DL can help improve radio performance, particularly in challenging channel conditions where traditional Digital Signal Processing (DSP) algorithms struggle, such as low SNR, highly dynamic environments. By providing raw data samples and careful labelling, an AI model can be developed to answer almost any problem, if designed well.

Wireless communications presents a unique opportunity for DL applications because the environment can be simulated and synthesised. Tools such as MATLAB communications toolboxes [23] and GNU Radio [24] can simulate channel models and generate a vast array of different signal types, while simultaneously labelling datasets, overcoming a common bottleneck in many DL domains: lack of data [25]. Therefore wireless communications datasets for various tasks can be created without the need to gather real-world data, significantly accelerating development.

To fully leverage DL in a deployed radio system, several practical challenges must be addressed, particularly when targetting real-time, edge-deployed inference, like implementing a DL algorithm into the signal processing pipeline of the RFSoC.

Despite the strong potential of DL in wireless communications, integrating these models into a real-time radio comes with several practical challenges. In traditional radio pipelines on FPGAs, like Figure 2.13, samples are received by the radio and digitised at the RF-ADC before being streamed into the PL at a set sampling rate. For as long as the radio operates, this stream of data is constantly received. In some systems, the radio captures data and then pauses while the data is transferred to an external AI processor for inference. The system waits for a response before continuing operation, which may not be acceptable in applications that require uninterrupted, real-time processing. As a result, a DL accelerator that complements this continuous stream of samples is necessary, i.e. an architecture that operates within the streaming context of a traditional signal processing pipeline, ideally operating directly within the PL

alongside standard DSP blocks. This motivates the design of a streaming-based CNN accelerator: an architecture capable of consuming I/Q samples at runtime and performing inference in a filter-like manner.

Such an accelerator should address these deployment criteria:

- **Latency:** Many radio tasks, such as signal identification and modulation classification in spectrum monitoring, must produce timely outputs to be actionable. Offloading inference to software, external, or processor-based accelerators introduces delays incompatible with real-time constraints. A streaming hardware implementation minimises latency by keeping inference within the signal path.

- **Throughput:** Wireless signals, especially at high samples rates, demand high-throughput processing to avoid dropping samples. A streaming CNN can process a stream of samples, one or more each clock cycle, maintaining the line-rate required by the signal processing pipeline.

- **Resource and Power Efficiency:** While platforms like the RFSoC provide powerful resources for signal processing, they are still limited in terms of memory and logic elements. Complex and large designs can quickly exhaust these resources. To address this, AI accelerator architectures must be optimised for low resource usage. By implementing models in fixed-point on the PL improves hardware efficiency and enables low-power, edge-based inference without relying on external processors.

Overall, the case for a streaming-based AI accelerator is not just performance-driven, it is architectural. DL, when adapted to the streaming-based nature of radio hardware, can behave like a learned signal processing block, embedded within the same AXI4-Stream infrastructure that already supports filters, demodulators, and symbol to bit decoders.

## 1.3 Background and Prior Work

This thesis intersects three active fields of engineering research: DL, wireless communications and signal processing, and hardware accelerated FPGA development, illustrated by Figure 1.1.

While each area has seen significant progress independently, relatively few works attempt to bridge all three. Achieving a combination of the three domains requires a background in DL acceleration, FPGA development with SDRs, and

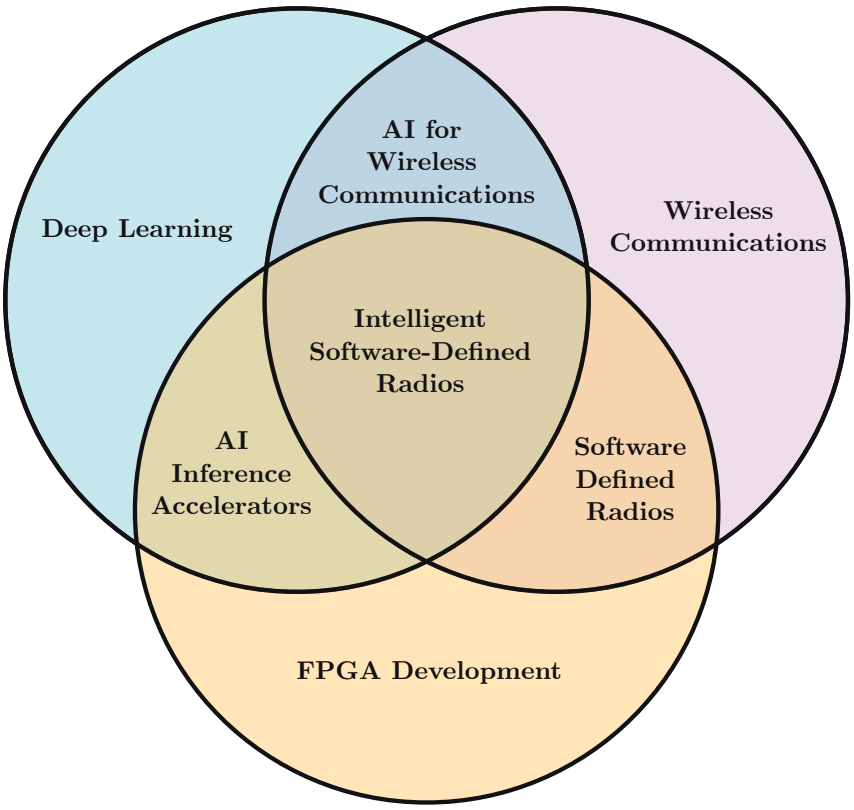



Figure 1.1 Venn diagram showing the intersection of three domains of engineering research.

a solid understanding of wireless communications and its applications with AI. This section provides an overview of key developments in each area to situate the present work within the broader research landscape.

### 1.3.1 CNNs in Wireless Communications

The use of AI in PHY wireless communications has grown substantially in recent years, driven by the need for more adaptable and data-driven solutions to address the increasingly harsh channel environments and congested spectrum [1]. Traditional PHY algorithms rely heavily on expert-designed signal processing functionality, which are often tailored to a specific environment and require manual tuning. AI models, particularly DL models, have the ability to learn directly from raw samples, and to perform tasks that are otherwise difficult to hand-craft.

CNNs have been widely adopted in PHY wireless communications due to their ability to extract local features from time-series or spectrogram representations of the complex data. These models have demonstrated strong performance in many communications tasks such as:

- Channel estimation, where CNNs are used to improve performance in harsh channel conditions, or to match the performance of traditional techniques using fewer pilots symbols [26]–[28].
- RF fingerprinting, where CNNs are used to uniquely identify devices by learning subtle, hardware-specific imperfections in their transmitted signals, know as RF fingerprints. This allows for reliable identification of devices in IoT environments [29], [30], across various channel conditions [31], and even when channel state information is unavailable [32].
- Signal decoding, where CNNs are used to recover bit sequences from received signals, showing promising results in comparison to traditional decoding methods [4], [33], [34].
- Signal identifications, where CNNs have been widely used for tasks such as Automatic Modulation Classification (AMC). This was popularised by [19], where CNN models were applied directly to raw I/Q data. Subsequent research focused on refining CNN architectures for improved performance [35]–[37]. More recent work has begin to explore transformer-based models for AMC, while still using CNN inspired structures [38].

CNNs are a popular model variant for addressing many PHY wireless communications tasks, where the acceleration of these model structures can assist with the realisation of state-of-the-art DL models operating on edge devices. AMC is a suitable candidate for demonstrating the effectiveness of a real-time system that can be reused for other wireless communication tasks.

### 1.3.2 CNN Dataflow Accelerators on FPGAs

While there are many NPUs and Tensor Processing Unit (TPU) structures out in the field that are specifically built to make AI model acceleration faster and more power efficient, they are built with the intention of being architecture agnostic, meaning that they will process the DL model calculations regardless of the model topology. One downside to these processors is the memory bottleneck associated with reading and writing data on and off-chip (more on this in Chapter 3). Synchronous Dataflow (SDF) models [21] instead implement a fixed topology that is optimised towards the model dimensions, resulting in a faster throughput accelerator. These models are suited for devices with PL, taking advantage of its inherent reprogrammability and scope to implement custom architectures.

SDF-based works that accelerate CNN models include: FINN [39], a design tool that generates custom accelerators for FPGA development by implementing each layer of a model using streaming architectures that are optimised for FPGAs; fpgaConvNet [40], a toolflow that works similarly to [39] by mapping CNN models to FPGAs; and hls4ml [41], a Python package for mapping AI model inference to FPGAs by creating firmware using High Level Synthesis (HLS) [41].

While tools like FINN, fpgaConvNet, and hls4ml focus on general-purpose acceleration, they do not specifically target applications where continuous data reception is a priority, such as in wireless communications. These methods prioritise computational efficiency without guaranteeing a pipeline with no sample loss. The approach taken in this thesis focuses on designing a streaming pipeline architecture based around the input data rate itself, ensuring that all the incoming samples into the model are processes without loss. This application-driven approach motivates the need for a custom architecture built for real-time radio receivers.

### 1.3.3 Real-time CNN Inference for Modulation Classification

Real-time inference of a CNN model for the task of modulation classification is a step towards the realisation of CNN models being deployed on FPGA-enabled SDR systems. Prior studies, such as [42] and [43], have explored Short Time Fourier Transforms (STFTs) and CNN architectures for modulation classification on FPGAs with pre-recorded data. These investigations primarily focused on simulated data and did not encompass real-time received signals. Another work demonstrated classification of radio signals in real-time using the Line Hough Transform with spectrograms on an AMD RFSoC device, a real-time deployment without the use of CNNs [44]. Preliminary results demonstrating how FINN [39] could be used to deploy a model for the RadioML dataset [45] were presented in a proof of concept by [46]. A low-precision CNN accelerator to perform modulation classification on the AMD RFSoC on the RadioML dataset was implemented by [47]. The work was continued by implementing a four class modulation classifier operating live with the RF-ADCs. However, the majority of these studies, expect for [47], rely on pre-saved datasets. While [47] does demonstrate live classification using RF loopback, the received signals are clean and unaffected by multipath channel effects. As a result, there remains

a gap in the literature concerning live classification of signals that have been subjected to adverse channel conditions.

To the best of author's knowledge, a single solution for a real-time CNN modulation classifier operating on live signals with a SDR has not been fully explored in any previous work.

## 1.4 Research Aim & Objectives

The research aim of this work is to develop a custom CNN accelerator architecture that can integrate into the streaming pipeline of FPGA-based radio receivers, particularly the AMD RFSoC, to support real-world PHY wireless communications applications. To accomplish this, the main objectives are as follows:

- To design and develop a streaming-based CNN accelerator that operates at the input data rate into the accelerator without dropping samples.
- To investigate the training requirements for deploying a CNN model in real-world, real-time radio environments, including the creation of a custom dataset to reflect deployment conditions.
- To evaluate the effects of fixed-point quantisation on the performance of the deployed model.

The following subsections expand on each of these research objectives and their importance in the context of wireless communications radio receivers deployed in real-world settings.

### 1.4.1 Streaming-based CNN Accelerator for Real-time Radio Receivers

A radio receiver pipeline is typically composed of DSP blocks that stream data from one stage to the next in a pipeline, with each block responsible for a specific signal processing task, as depicted in Figure 1.2. These blocks might include Digital Down-Converters (DDCs), Finite Impulse Response (FIR) filters, channelisers, demodulators, and other custom logic. In FPGA-based radio receivers like the AMD RFSoC, these DSP functions are usually deployed concurrently on the PL and operate in a streaming fashion. Each stage processes the data stream and passes it along, forming a dataflow architecture. This

parallelism enables the receiver to keep up with high sampling rates and meet the real-time demands of wireless communications.

DL-based models, particularly CNNs, are increasingly being explored to tackle some of the more challenging tasks in radio receivers. In simulation, CNNs can show strong performance improvements over traditional methods. However, deploying these models in a real-time radio system is far from straightforward. When using platforms like GPUs for inference, the receiver typically needs to buffer samples, transfer them to an external device, process them, and wait for the results before continuing. This adds significant latency and disrupts the streaming nature of the pipeline. Worse still, only chunks of data can be processed in this way, meaning that not every sample is acted on in real time, something that is unacceptable in many RF applications.

Instead, as shown in 1.2, a CNN accelerator that integrates directly into the streaming pipeline, treating the input as a continuous flow of samples, is far more suitable for deployment on FPGA-based receivers. Such an architecture must be designed to avoid dropping samples from the RF-ADC, and instead operate at the input data rate of the system. This thesis explores the design choices, challenges, and optimisations needed to create a CNN accelerator that fits naturally into the dataflow structure of real-time radio receivers.

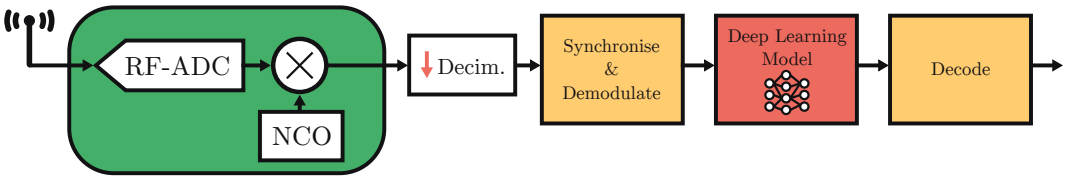



Figure 1.2 A DL model as a part of a streaming dataflow radio receiver pipeline.

### 1.4.2 Training Requirements for Deploying CNN Models in Real-world Scenarios

DL models for PHY tasks such as modulation classification [19] are often trained using large, high-quality datasets generated in simulation environments. These datasets are typically idealised, with controlled noise levels, well-defined modulation formats, and standardised channel models that simulate multi-path effects in wireless environments. Models are usually trained and evaluated using floating-point precision on high-performance computing platforms. While this setup is useful for initial development and benchmarking, it does not reflect the

constraints and imperfections faced during real-world deployment, especially on embedded platforms like the AMD RFSoC.

This work looks at how to build more representative training datasets that capture the real-world imperfections introduced by the RF Data Converters (RFDCs). The RFDC is a hardened components on the RFSoC that hosts the RF-ADCs and RF Digital to Analogue Converters (RF-DACs). By using the RFSoC itself as part of the training loop, data can be collected directly from the RF-ADC, capturing hardware-induced artefacts and quantisation effects inherent to the SDR platform. The goal is to establish a training and validation process that narrows the gap between simulation and deployment, allowing the final model to generalise more effectively when running on hardware and improving real-world performance consistency.

### 1.4.3 Lower Precision Weights for Real-Time CNN Models

Deploying a trained CNN model onto a real-time, FPGA-based radio receiver introduces a number of challenges beyond just maintaining throughput. Due to hardware limitations, floating-point arithmetic is often not practical in the PL of the RFSoC. Instead, fixed-point arithmetic is used to reduce logic area, power consumption, and overall resource usage. However, simply converting a floating-point model to fixed-point can lead to a drop in inference accuracy. This thesis investigates that precision-performance trade-off by comparing two quantisation techniques: Post-Training Quantisation (PTQ) and Quantised-Aware Training (QAT). Both approaches are evaluated at different bit-widths to assess the impact on accuracy and resource efficiency within the proposed accelerator architecture.

An important consideration is the engineering cost of migrating a model from software to hardware. This includes evaluating what retraining may be required to enable a successful transition, and whether the model weights can be reduced to fixed-point with minimal overhead. This thesis aims to provide a practical ‘path of least resistance’ for engineers, keeping the model topology fixed and instead evaluating how different quantisation and deployment strategies affect accuracy and implementation effort.

## 1.5 List of Contributions

Each chapter solves a specific challenge in the process from architecture design to real-time deployment. Chapter 4 addresses the architectural challenge of processing high data-rate I/Q samples directly on the PL using a streaming dataflow CNN architecture. Chapter 5 focuses on the challenge of creating a dataset that realistically represents RF hardware distortions and enables integration of the dataflow CNN architecture with the RF-ADC's live inputs. Chapter 6 tackles the deployment of low-precision weights into the custom CNN architecture and investigates the effects of different weight quantisation methods on the resulting real-time accuracy.

The contributions arising from this work are the following, depicted in Figure 1.3 and the list below.

1. **Development and evaluation of a streaming-based CNN architecture for SDR receivers.** To the best of the author's knowledge this is the first radio receiver-specific CNN architecture designed to process every sample received. A custom SDF CNN architecture is proposed that directly complements the constant stream of I/Q samples originating from the RF-ADC of the AMD RFSoC development platform. This work explores the design considerations required to successfully process every sample produced by the RF-ADC receiver pipeline while allowing the integration of the architecture with other streaming IP cores, such as FIR filtering stages. This contribution introduces design considerations such as ping-pong buffers, channel-first streaming matrix transformations, and resource-sharing factors based on the available clock rates, to maintain the constant processing and delivery of data samples.

2. **A methodology for generating datasets for radio receiver implementations.** Generating datasets for RF-applications using simulation tools, such as MATLAB/Simulink or GNU Radio, is good for initial DL model development; however, transferring the trained models into real-world implementation in hardware does not yield similar performance due to the hardware characteristics of the radio's RF-ADCs not being captured. This contribution proposes a method for generating channel-distorted signals from simulation tools while also learning the real-world hardware-specific impurities of the SDR.

3. **Investigations into quantised training methods and the effects on resulting real-world accuracy performance.** This thesis presents one of the first in-depth comparisons of PTQ and QAT applied to real-time modulation classification using fixed-point CNN models deployed on a SDR platform. This contribution explores the use of fixed-point quantisation, at 16-bit, 8-bit, 4-bit, and 2-bit precision, for weights within the proposed custom CNN architecture. It presents an investigation into the quantisation techniques, with a focus on their application in live signal reception scenarios using the AMD RFSoC platform. Unlike prior work, which relies on pre-saved datasets or clean loopback signals, this thesis targets the gap in evaluating quantised neural network models under real-world wireless conditions.

4. **Deployment and demonstration of a real-world real-time modulation classification application on the AMD RFSoC.** To the best of the author's knowledge this is the first implementation of a real-time CNN model performing modulation classification on an SDR affected by a multi-path distorted channel while receiving a live signal. This work is available on a GitHub repository [48] from where the demonstrator can be installed and run on AMD RFSoC development platforms.

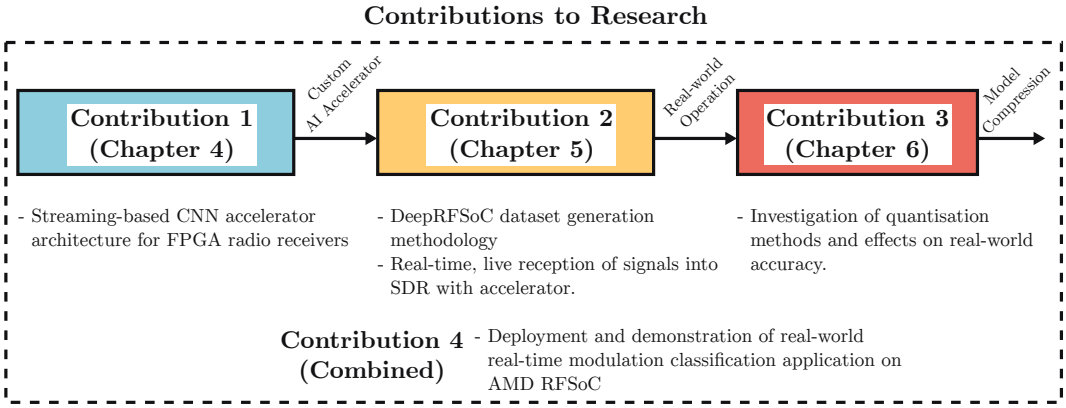



Figure 1.3 Connection between the contributions of this thesis.

## 1.6 Publications and Outputs

Various publications, invited talks, and demonstrators have been published as part of this PhD study, which are listed below.

(a) **A. Maclellan**, L. D. McLaughlin, L. H. Crockett, and R. W. Stewart, "FPGA Accelerated Deep Learning Radio Modulation Classification Using MATLAB System Objects & PYNQ," (conference paper), in *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, Barcelona, Spain, Sept. 2019, pp. 246-247
Available: https://doi.org/10.1109/FPL.2019.00045.

(b) **A. Maclellan**, L. H. Crockett, and R. W. Stewart, "Streaming-CNN FPGA Architecture for Communications-based Applications," (demonstrator), in *2022 30th IEEE International Symposium On Field-Programmable Custom Computing Machines (FCCM)*, Cornell University, New York, USA, May 2022.
Available: https://www.fccm.org/past/2022/demo-night-2022/

(c) **A. Maclellan**, L. H. Crockett, and R. W. Stewart, "Modulation classification for RFSoC showcasing streaming-CNN architectures," (poster), in *2022 IEEE SPS - EURASIP Summer School: Defining 6G: Theory, Application, and Enabling Technologies*, Linköping University, Linköping, Sweden, Sept. 2022.
Available: https://pureportal.strath.ac.uk/en/publications/modulation-classification-for-rfsoc-showcasing-streaming-cnn-arch

(d) T. Nyasulu, G. Fitzpatrick, **A. Maclellan**, E. Atimati, D. Crawford, "Dynamic Spectrum Access and Cognitive Radio," (book chapter), in *Software Defined Radio with Zynq UltraScale+ RFSoC*, Strathclyde Academic Media, 2023.
Available: https://www.rfsocbook.com/

(e) **A. Maclellan**, L. H. Crockett, and R. W. Stewart, "Streaming Convolutional Neural Network FPGA Architecture for RFSoC Data Converters," (conference paper), in *2023 21st IEEE Interregional NEWCAS Conference (NEWCAS)*, Edinburgh, UK, June. 2023, pp. 1-5.
Available: https://doi.org/10.1109/NEWCAS57931.2023.10198198

(f) **A. Maclellan**, "Deployable Deep Learning Inference on AMD RFSoC for Modulation Recognition," (invited talk), presented at University College London Radar Group Seminar, UK, Dec. 2023.
Available: https://youtu.be/_s2C6QPrIvc

(g) **A. Maclellan**, L. H. Crockett, and R. W. Stewart, "RFSoC Modulation Classification With Streaming CNN: Data Set Generation & Quantized-

Aware Training," (journal paper), in *IEEE Open Journal of Circuits and Systems (OJCAS)*, vol 6, Dec. 2024, pp. 38-49.
Available: https://doi.org/10.1109/OJCAS.2024.3509627

## 1.7 Thesis Organisation

The remainder of this thesis is organised as follows:

- **Chapter 2** covers the basics of PHY wireless communications and reviews the core digital modulation types, pulse-shaping, and channel impairments in the context of this thesis. The topic of AMC is reviewed, as well as the theory behind the transition from ML to DL-based approaches. The AMD RFSoC is introduced alongside PYNQ – these are the target device and software for all implementations in this thesis.

- **Chapter 3** reviews DL and AI accelerator fundamentals. Here, the core theory behind CNN models is covered, alongside training for both floating-point and quantised networks. The underlying concepts and motivations behind accelerating CNN models are discussed along with common implementation considerations.

- **Chapter 4** introduces a new streaming CNN architecture for SDR receivers. It also presents the design process and optimisation considerations involved in achieving a real-time streaming CNN model. The work discussed in this chapter was published and presented in (b), (c), (e), (f), and (g), as listed in Section 1.6.

- **Chapter 5** presents implementation considerations for deploying the architecture proposed in Chapter 4. A methodology for generating a dataset that can be used to train a CNN model intended for real-world deployment is presented. The work from this chapter was published and presented in (f) and (g).

- **Chapter 6** conducts an investigation into the different quantisation requirements when training a CNN model for modulation classification. This chapter compares the accuracy performance of deployed models quantised to 16-bit, 8-bit, 4-bit, and 2-bits trained with PTQ and QAT methods. Each model is evaluated on its dataset and live data capture accuracy performance. The work in this chapter was published and

presented in (a), (g), and (h), with accuracy performance improved since works were published.

- **Chapter 7** summarises the thesis, presents the key conclusions, limitations, and outlook from the research, and proposes future work.

# Chapter 2

# Physical Layer Communications and Modulation Classification

This chapter introduces the fundamental concepts in Physical Layer (PHY) wireless communications, modulation classification, and the SDR platform of choice for this thesis.

## 2.1 Primer on Physical Layer Communications

This section covers the fundamentals of physical layer communications required to understand the AMC application explored in this thesis.

### 2.1.1 Introduction

The PHY of a communication system is responsible for transmitting information from one device to another over a physical medium. In wireless systems, this medium is the electromagnetic spectrum. Unlike wired systems where the transmission channel is controlled and shielded, wireless communications must operate across a dynamic and unpredictable environment.

At a high level, a typical wireless link involves a transmitter, a channel the information passes through, and a receiver, as depicted in Figure 2.1. The transmitter takes binary information and maps it to symbols using a digital modulation scheme. These symbols are pulse-shaped to control bandwidth and inter-symbol interference, and then interpolated and modulated onto a carrier signal for Radio Frequency (RF) transmission through the channel. The role of the transmitter is to prepare the signal in such a way that it remains recoverable at the receiver despite channel impairments.

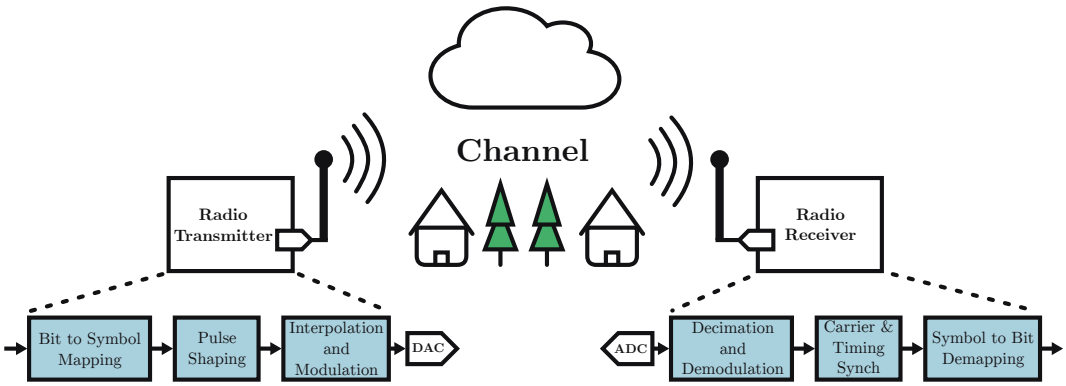



Figure 2.1 Overview of a typical wireless communications link between two radios [49].

At the receiver, the incoming RF signal is captured, demodulated, digitised, and decimated. From there, the receiver attempts to undo the effects of the channel by correcting frequency and timing offsets, and demapping the signal to extract the transmitted information from the symbols. In many scenarios, such as spectrum sharing or Dynamic Spectrum Access (DSA), the receiver may not have any prior coordination with the transmitter. Instead, it must monitor the spectrum for transmissions by identifying detected signal types before making any transmission decisions.

These challenges motivate the use of ML techniques such as modulation classification, which allow the receiver to automatically identify the modulation scheme used by an unknown radio. These data-driven approaches complement traditional signal processing and open the door to intelligent and adaptive wireless systems.

### 2.1.2 Digital Modulation

Digital modulation is the process of encoding a frame of bits onto a carrier signal for transmission over a wireless medium. Demodulation is the reverse process, where bit information is extracted from the received carrier signal. Corruption introduced by the wireless channel often results in bit errors after demodulation. To combat these impairments, digital messages are encoded using various modulation schemes designed to improve robustness against channel impairments.

To support more reliable communications, various types of modulation schemes are used to encode information into specific properties of the carrier wave. These include amplitude, frequency, and phase.

A general term for a modulated carrier signal is given by

$$s(t) = a(t)cos\left(2\pi f_c t + \phi(t) + \phi_0\right), \tag{2.1}$$

where $a(t)$ is the time-varying amplitude, $f_c$ is the carrier frequency, $\phi(t)$ is the phase modulation component, and $\phi_0$ is a fixed phase offset.

Eq. 2.1 can be rewritten in terms of in-phase and quadrature components using trigonometric identities

$$s(t) = s_I(t)\cos(2\pi f_c t) - s_Q(t)\sin(2\pi f_c t), \tag{2.2}$$

where the in-phase component of $s(t)$ is $s_I(t) = a(t)\cos\left(\phi(t) + \phi_0\right)$ and the quadrature component is $s_Q(t) = a(t)\sin\left(\phi(t) + \phi_0\right)$[15], [50].

$s(t)$ can be represented in terms of its complex low-pass representations as

$$s(t) = \mathrm{Re}\{u(t)\}\cos(2\pi f_c t) - \mathrm{Im}\{u(t)\}\sin(2\pi f_c t) \tag{2.3}$$

$$= \mathrm{Re}\{u(t)e^{j2\pi f_c t}\} \tag{2.4}$$

where $u(t) = s_I(t) + js_Q(t)$ is the complex baseband representation (or complex envelope) of the modulated signal $s(t)$. The in-phase and quadrature components can be recovered as $s_I(t) = \mathrm{Re}\{u(t)\}$ and $s_Q(t) = \mathrm{Im}\{u(t)\}$, respectively.

This formulation separates the signal into two orthogonal components, which align with vector representations like complex numbers and constellation points. This In-phase (I) and Quadrature (Q) representation forms the foundation of modern digital radio design.

The following section introduces digital modulation schemes used in this thesis. Each scheme encodes information bits differently using amplitude, frequency, phase, or a combination thereof. The design of each scheme reflects different priorities: some aim to maximise spectrum efficiency by packing more bits per symbol, while others prioritise robustness against noise and fading in challenging channel conditions.

### Common Modulation Schemes

A selection of common modulation schemes is described below. Amplitude, frequency, and phase modulation techniques encode the signal by varying the amplitude, frequency, or phase of the symbol duration $T_s$. By the end of each symbol interval, the transmitted signal has conveyed one or more information bits.

In amplitude and phase modulation schemes, the symbol transmitted during each interval $T_s$ is represented by a complex value $I + jQ$, mapped to a constellation point that corresponds to a group of $K$ bits. The duration $T_s$ is referred to as the 'symbol time', and each unique point in the constellation represents one 'symbol'.

**Amplitude Modulation** Amplitude modulation encodes information using the amplitude (or voltage level) of the signal over the symbol time $T_s$. The simplest form is M-ary Pulse Amplitude Modulation (M-PAM), a one-dimensional scheme with no quadrature component. In M-PAM, the signal's amplitude $A_s$, takes on one of $M$ discrete levels, each representing a unique symbol. The number of bits per symbol is $K = \log_2(M)$ [15], [50].

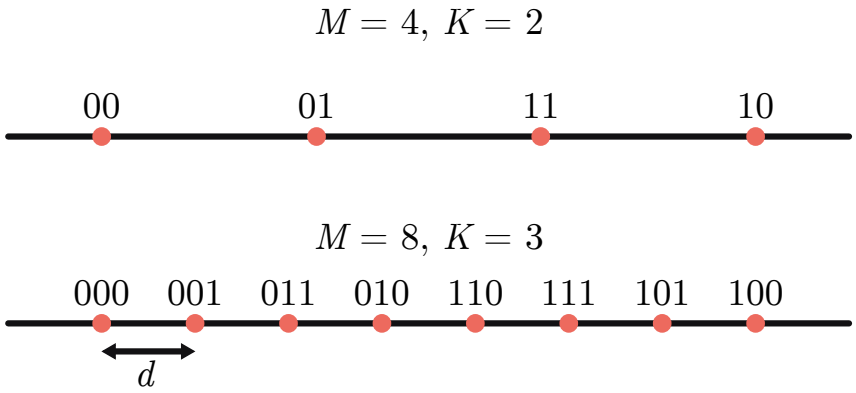



Figure 2.2 M-PAM encoding example [50].

Figure 2.2 shows examples of PAM encoding for $M = 4$ and $M = 8$. The amplitude of each symbol is chosen from $M$ distinct levels, allowing each symbol to represent $\log_2(M) = K$ bits. The spacing between amplitude levels is typically referred to as $d$, and it plays an important role in determining noise tolerance [50]. While PAM offers a simple implementation, it becomes increasingly sensitive to noise as the number of amplitude levels grow, since constellation points are spaced more closely.

**Phase Modulation** Phase modulation encodes information using the phase of the signal. A common type of phase modulation is M-ary Phase Shift Keying (M-PSK), with M discrete phases. The transmitted signal over a symbol time period $T_s$ is given by

$$s_i(t) = Re\{Ag(t)e^{j2\pi(i-1)/M}e^{j2\pi f_c t}\} \tag{2.5}$$

where $g(t)$ is the pulse shaping applied to the transmitted signal, and $A$ is the amplitude.

In M-PSK, both the amplitude and frequency stay constant while the information is carried entirely in the phase. The constellation points (or symbols) are evenly spaced in phase around the unit circle as described by the term $e^{j2\pi(i-1)/M}$. Figure 2.3 shows how constellation points are distributed for different $M$ values.

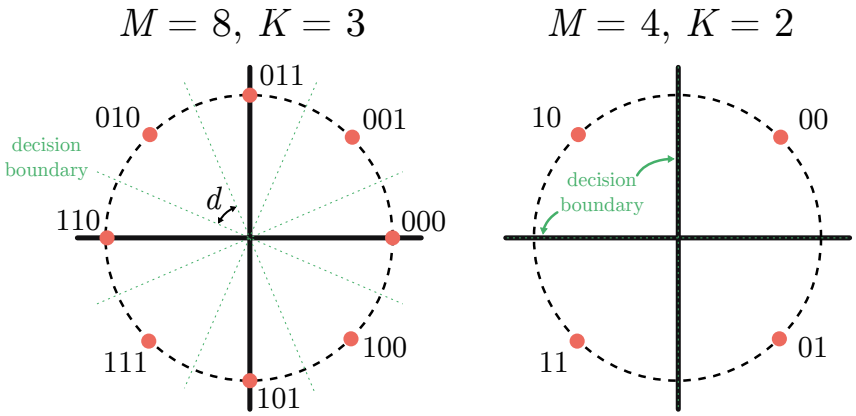



Figure 2.3 M-PSK encoding example [50].

The decision boundary defines which constellation point a received signal is closest to. Since the points are equally spaced, the angular spacing between them is $d = 2\pi/M$. All the symbols have the same transmit power, as they lie on the same amplitude around the circle [49], [50].

This thesis uses the M-PSK schemes BPSK, QPSK, and 8-PSK, which correspond to $M = 2$, 4, and 8, respectively. Choosing a higher $M$ value increases the bitrate, since more bits are encoded per symbol. However, this comes at the cost of the symbols being packed closer together in phase, which makes the signal more susceptible to phase noise and harder to decode reliably in noisy channels.

**Amplitude and Phase Modulation** Previously amplitude and phase modulation separately provided one degree of freedom in which to encode the information bits. By combining the amplitude and phase, two degrees of freedom allows for the constellation point to be placed anywhere in the unit circle, allowing for more symbols to be mapped while keeping the distance between points to a maximum. This scheme allows for the most bits per symbol for a given average energy [50]. The transmitted signal is given by

$$s_i(t) = Re\{A_i e^{j\theta_i} g(t) e^{j2\pi f_c t}\}. \tag{2.6}$$

Quadrature Amplitude Modulation (QAM) is a version of amplitude and phase modulation that builds constellations into square grids. An example of M-QAM (QAM with M constellation points) is shown in Figure 2.4.

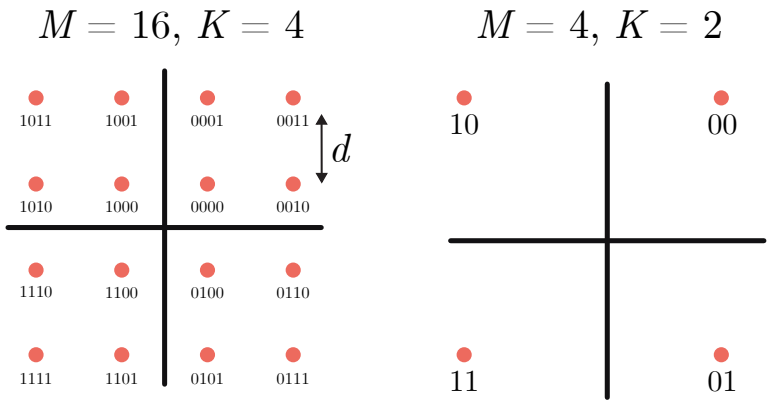



Figure 2.4 M-QAM encoding example [50].

In square constellation layouts, the distance between any pair of symbols is determined by $d_{ij} = ||s_i - s_j||$. The minimum distance between each symbol is $d$, the same as with M-PAM.

**Frequency Modulation** In digital communications, frequency modulation can refer to M-ary Frequency Shift Keying (M-FSK), where information is encoded by selecting one of $M$ discrete frequencies corresponding to each symbol. At each symbol interval, $K = \log_2 M$ bits determine the transmitted frequency through the index $i$, the digital frequency modulated signal can be expressed as:

$$s_i(t) = A\cos(2\pi f_i t + \phi_i), \quad 0 \leq t \leq T_s, \tag{2.7}$$

where $f_i$ is the frequency corresponding to the $i$th symbol with a constant phase offset $\phi_i$.

An equivalent formula uses the centre frequency $f_c$ and defines each symbol frequency as an offset from $f_c$. The frequency for the $i$th symbol is given by $f_i = f_c + \alpha_i \Delta f_c$, where $\alpha_i = 2i - 1 - M$ for $i = 1, 2, \ldots, M$, and $\Delta fc$ is the minimum frequency spacing between FSK carriers from $f_c$ [50].

The transmitted signal becomes:

$$s_i(t) = A\cos\left[2\pi f_c t + 2\pi\alpha_i \Delta f_c t + \phi_i\right], \quad 0 \leq t \leq T_s. \tag{2.8}$$

In M-FSK, each of the $M$ symbols modulates a distinct carrier frequency, similar to the $M$ levels in M-PAM, M-PSK, and M-QAM. Each frequency

represents a unique symbol, encoding $K$ bits. This results in a signal with abrupt frequency changes, which introduces phase discontinuity and can produce a spectrally inefficient signal due to bandwidth spreading [50]. Figure 2.5 shows an M-FSK signal in the time domain, alongside the sequence of bits it encodes.

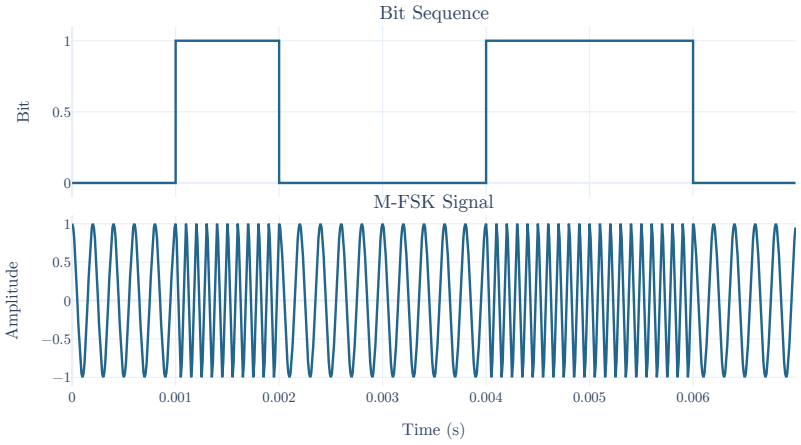



Figure 2.5 M-FSK encoding example for $M = 2$.

This thesis focuses on two modulation schemes that reduce phase discontinuity: Gaussian FSK (GFSK) and Continuous Phase FSK (CPFSK).

In GFSK, the raw binary bitstream is passed through a Gaussian filter prior to modulation. This smooths the transitions between bits (e.g. from 0 to 1), softening the frequency shifts and reducing the phase discontinuities present in standard M-FSK [50].

In CPFSK, when the signal changes frequency to encode new bits, the modulator maintains phase continuity by preserving the current phase and carrying it into the next symbol. This ensures that phase transitions remain smooth and continuous, preventing abrupt changes that would otherwise lead to spectral spreading. As a result, CPFSK maintains a constant envelope and remains band-limited [50].

### 2.1.3 Pulse Shaping

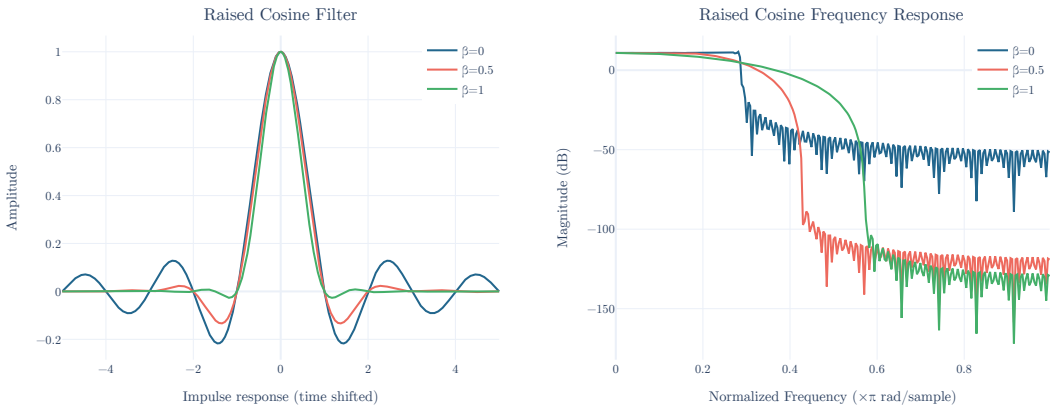
In a digital communication system, the transmission of symbols over a channel involves mapping each symbol to a time-domain pulse. Sudden changes in amplitude, such as those in M-PAM, can lead to poor spectral characteristics in the transmitted signal. While pulse shaping is used to smooth abrupt amplitude transitions, it also helps reduce spectral broadening caused by sudden phase shifts in phase modulation schemes. By smoothing the transitions between

symbols, pulse shaping improves both amplitude and phase continuity, resulting in a more spectrally efficient transmitted waveform.

Pulse shaping is a method of reducing the sidelobe energy relative to a rectangular pulse, but the shaping of the signal must be performed such that the received signal (after passing through a channel) has zero or minimal Inter-Symbol Interference (ISI).

One of the most popular pulse shapes for reducing ISI is the Raised Cosine (RC), shown in Figure 2.6. This pulse shaping filter has a frequency response with a flat top and sides that rolls off like a cosine function, hence the name 'raised cosine'. The filter is especially useful because it is parameterisable through the roll-off factor $\beta$. $\beta = 0$ yields a rectangular response, in the frequency domain, whereas $\beta = 1$ provides a wider-bandwidth and a smoother roll-off. The time-domain samples for the RC filter is defined as [50]

$$p(t) = \frac{\sin \pi t/T_s}{\pi t/T_s} \frac{\cos \beta \pi t/T_s}{1 - 4\beta^2 t^2/T_s^2}. \tag{2.9}$$




(a) Impulse response with changing roll-off $\beta$.

(b) Associated frequency response

Figure 2.6 Raised Cosine filter coefficients as $\beta$ changes and frequency response.

### 2.1.4 Channel Models and Effects

In wireless communications, signals that are sent over the air are subject to various impairments as they propagate through the channel. These impairments, caused by effects such as Doppler shifts, carrier frequency offsets, and additive noise, significantly distort the transmitted signal, altering its time, frequency, and phase characteristics. Understanding these distortions is critical for the design of any AI system that aims to classify received signals.

The transmitted signal $u(t)$ can be represented as a complex time-series of discrete bits modulated onto a complex sinusoid. After transmission through a multipath fading channel $c(t)$, the signal becomes:

$$v(t) = u(t) * c(t) \tag{2.10}$$

where $*$ denotes the convolution with the channel impulse response. The received signal $r(t)$ can be modelled as:

$$r(t) = v(t)e^{j2\pi f_c t} + w(t), \tag{2.11}$$

where $e^{j2\pi f_c t}$ represents the carrier at frequency $f_c$ and $w(t)$ denotes Additive White Gaussian Noise (AWGN) representing thermal background noise and other sources of non-deterministic noise.

Equation 2.11 provides a simplified system. In reality, more terms are present when modelling multipath propagation, leading to a more complete expression:

$$r(t) = u(t)e^{j2\pi f_c t}\left(\sum_{n=0}^{N(t)} a_n(t)e^{-j\phi_n(t)}\right) + w(t) \tag{2.12}$$

where the system experiences $N(t)$ multipath components, each with its own time-varying amplitude $a_n(t)$ and phase shift $\phi_n(t)$. The phase shift for each path is given by:

$$\phi_n(t) = 2\pi f_c \tau_n(t) - \phi_{D_n} \tag{2.13}$$

where $\tau_n(t)$ is the path time delay and $\phi_{D_n}$ represents the phase shifts produced by Doppler [50]. Figure 2.7 illustrates a communications channel between two radios. The radio receives a line-of-sight signal as well as reflected and delayed multipath components from the channel environment. The Doppler effect occurs when a signal is reflected off of a moving object, inducing a phase shift depending on the velocity of the object.

The delay spread $T_m$ of a wireless channel is the time difference between the arrival of the earliest and latest significant multipath components of the transmitted signal. It characterises the dispersion in time of a channel caused by multipath propagation. A narrowband fading model is typically assumed, where the delay spread of a channel is much smaller relative to the inverse of the signal bandwidth $B$ for the transmitted signal, i.e. $T_m \ll \frac{1}{B}$, meaning

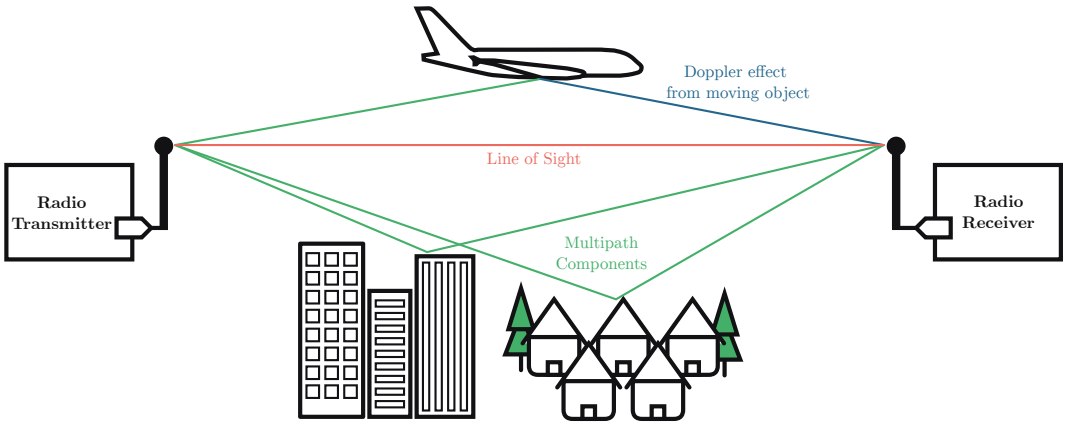



Figure 2.7 A communications channel between two radios with multipath and Doppler components.

that all multipath components arrive within a short duration and therefore the channel is assumed to have a flat frequency response.

These channel effects cause distortions that can disrupt core signal properties, making reliable interpretation and analysis of the received signals difficult. Addressing these challenges is essential for developing AI systems where the training data is as closely representative of a real-world scenario as possible and therefore defining the simulated channel models as accurately as possible is important.

### 2.1.5 The AMD RFSoC

A major outcome presented in this thesis is the integration of the proposed algorithms onto a SDR platform. It is important to demonstrate that the AI acceleration capabilities can run on a radio device that is suitable for real-world deployment.

The SDR platform used in this work is AMD's Zynq UltraScale+ RFSoC family. The RFSoC is a state-of-the-art solution with many advanced features for implementing flexible, high-performance radio systems [10]. It integrates wideband RF data converters with a general-purpose Processing System (PS) and PL into a single chip, enabling SDRs with both configurability and hardware acceleration capabilities.

The RFSoC includes multiple high-precision, high-speed RF-ADCs and RF-DACs, operating at sample rates of up to 5 Giga samples per second (Gsps) and 9.85 Gsps, respectively, with resolutions of up to 14-bits. The device consists of a PS, PL, and a set of hardened radio-specific Intellectual Property (IP) core blocks optimized for SDR applications.

The PS provides a range of processing resources, including the Application Processing Unit (APU), Real-time Processing Unit (RPU), and Platform Management Unit (PMU). It is responsible for running software associated with the SDR application.

The PL is a key part of the RFSoC architecture, enabling hardware acceleration and interfacing with the integrated RF-ADCs and RF-DACs. It hosts reconfigurable resources such as combinatorial logic, storage, and dedicated DSP blocks, including Look-Up Tables (LUTs), Flip-Flop registers (FFs), DSP48 slices, BlockRAMs (BRAMs), and UltraRAMs (URAMs) [10]. This fabric allows developers to implement custom DSP algorithms to hardware accelerate. Throughout the rest of the thesis, FPGA and PL are used interchangeably and refer to the usage of reconfigurable resources.

The RF-ADCs and RF-DACs are implemented in hardened RFDC IP cores within the RFSoC. These cores provide the RF front-end functionality, where each converter can be individually configured to tune to a desired spectrum band and sampling rate. In addition to data conversion, the RFDC includes mixers for up/down conversion between RF and baseband, as well as optional decimation and interpolation filters for bandwidth and sampling rate adjustment.

### AMD RFSoC 2x2

The RFSoC development board used in this thesis is the AMD University Program (AUP) RFSoC 2x2 Development Kit [22], shown in Figure 2.8. AUP, in collaboration with the board manufacturer, makes available subsidised academic boards for teaching and research purposes. The RFSoC 2x2 is a part of that initiative.

The RFSoC 2x2 development platform contains a 1st generation RFSoC chip, the XCZU28DR. This chip integrates two RF-ADCs and two RF-DACs (hence the '2x2' name). The RF-ADCs can achieve a maximum sampling rate of 4.096 Gsps and the RF-DACs can achieve a maximum sampling rate of 6.554 Gsps. The RFSoC 2x2's elevated sample rate capabilities and flexible reconfigurability open avenues for increasingly sophisticated applications across the frequency spectrum. It is an excellent candidate for evaluating custom AI acceleration architectures by building them into the PL building blocks. The RFDC in the PL digitises incoming analogue signals and passes the samples to the custom algorithm on the RFSoC for real-time signal evaluation. It is through this setup that this thesis demonstrates its operation on live signals [22].

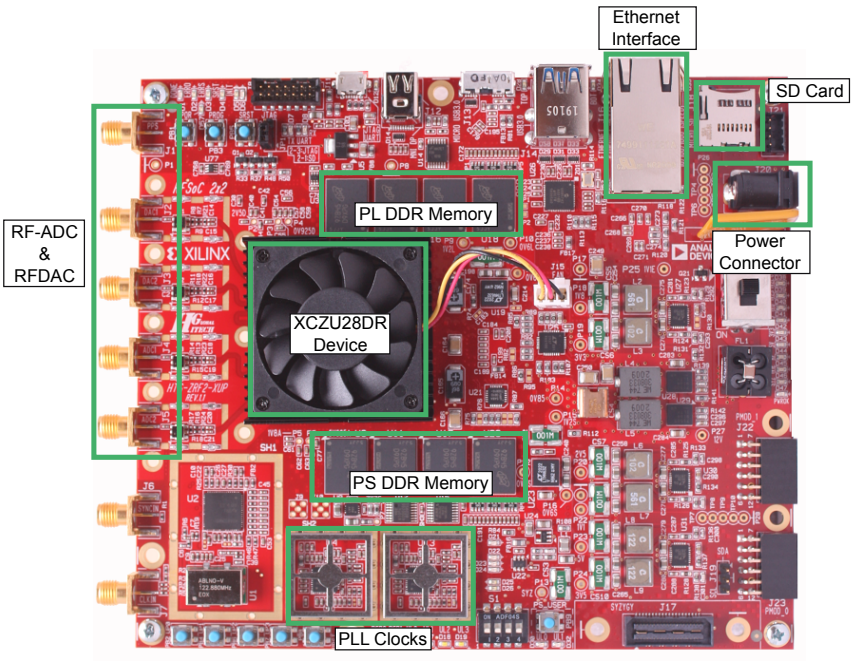



Figure 2.8 The AMD RFSoC 2x2 development board that is used throughout this thesis as the target platform for implementing and evaluating the custom AI accelerators.

The XCZU28DR part consists of a large FPGA which contains the resources detailed in Table 2.1. These are available for implementing custom logic.

Table 2.1 XCZU28DR available FPGA resources.

| **Resource** | LUTs | FF Registers | BRAM | Ultra RAM | DSP |
|---|---|---|---|---|---|
| **Total** | 425,280 | 850,560 | 1,080 | 80 | 4,272 |

The novel contributions described in this thesis target the RFSoC 2x2 development board and the AI accelerators are evaluated based on their total consumptions of the resources from Table 2.1.

### PYNQ Framework

PYNQ is a fully open source project, developed by AMD, that makes using adaptive compute platforms, such as Zynq-enabled FPGAs and RFSoCs, much easier. The name 'PYNQ' originates from the phrase 'Python productivity for Zynq', although it has since further developed into a framework that extends beyond interfacing with Zynq devices.

PYNQ uses the Python programming language, where designers can interact with functionality implemented on the PL while developing applications on the PS that benefit from the large ecosystem of Python libraries.

PYNQ ships with software and drivers that make it easy to communicate and interact with PL-accelerated functionality in conjunction with the applications operating on the PS. PYNQ enables the creation of high performance applications through: parallel hardware execution, high data-rate processing, hardware accelerated algorithms and real-time DSP [51], [52].

The PYNQ software stack, shown in Figure 2.9, builds on top of a typical embedded systems software stack. The hardware layer is at the bottom, which includes the FPGA PL with connections to the PS and hardened IP; followed by the Operating System (OS) layer, which hosts low-level drivers and Application Programming Interfaces (APIs) for interacting with the hardware; and finally the interface software, in this case Python and associated packages such as Jupyter [52]. The Jupyter session is hosted on the PS of the embedded device.

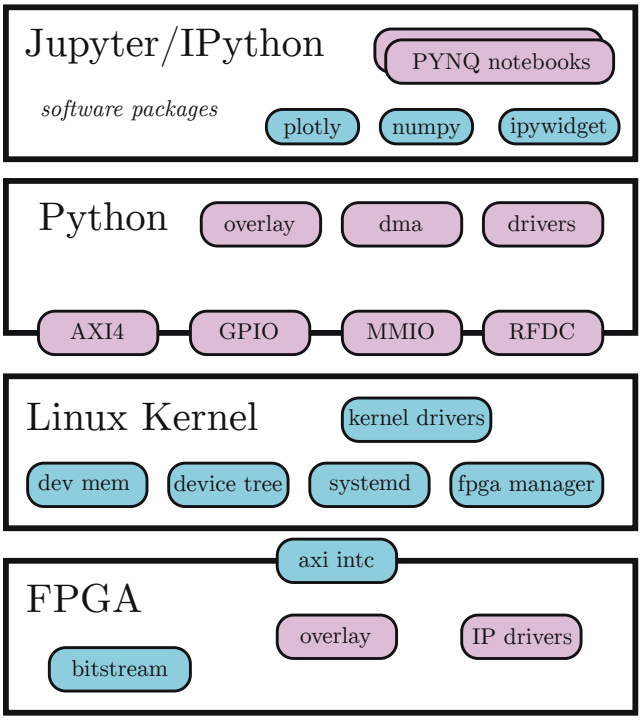



Figure 2.9 The PYNQ software stack layer breakdown. Pink cells represent PYNQ-enabled functionality.

The upper layers handle the user interaction with the PYNQ-enabled embedded device, facilitated through Jupyter notebooks where a browser-based IDE is used to create applications using the Python language and open-source libraries such as NumPy [53], Plotly [54], and ipywidgets [55]. PYNQ integrates additional PYNQ API functionality for interfacing with the PL and common IP cores. PYNQ also provides interfaces for hardened IP and external interfaces

such as the RFDC, General Purpose Input Output (GPIO), and Memory Mapped Input Output (MMIO) [52].

Jupyter [56] is an open-source web-based interface that allows users to write and execute code in an interactive, notebook-style environment. It combines code, visual output, and narrative text in a single document, making it a powerful tool for exploration debugging, and presentation. In the context of PYNQ, Jupyter is used to provide a platform for interacting with hardware-accelerated applications. Its notebook interface supports running code cells individually and in any order, enabling on-the-fly parameter tuning and experimentation with PL designs. Jupyter also supports `ipywidgets`, which allow developers to build interactive controls, such as sliders, buttons, and drop-down menus, enabling lightweight Graphical User Interfaces (GUIs) for real-time interaction with the PL and software components. Figure 2.10 shows a screenshot of a Jupyter notebook example for interacting with a RFSoC application via a web browser.

Python is used in PYNQ because it offers a high-level, user-friendly framework that simplifies the development of applications. Python's readability, large ecosystem of scientific and visualisation libraries, and interactive development style make it ideal for rapid prototyping and experimentation. By using Python, PYNQ lowers the barrier to entry for working with FPGA SoC devices, enabling developers, researchers, and students to focus on algorithm design rather than low-level hardware and software communication details [57].

The lower levels of the PYNQ software stack handle the interfaces with custom PL designs, board components, and the Ubuntu-based OS. PYNQ provides a high-level layer of abstraction to the Linux kernel connection to the target development platform through the device tree. This layer allows PYNQ to interface with hardened board components and PL bitstreams as if they were Python objects, simplifying the PS and PL interactions to an object-oriented approach.

The typical PYNQ workflow is described in Figure 2.11. Every PYNQ application begins with a complete bitstream with the desired hardware-accelerated algorithms implemented in the PL. To achieve a working bitstream, a block design is created, using the AMD Vivado development environment, and as long as the design successfully passes the synthesis and implementation stages, the bitstream can be generated.

PYNQ interactions begin when the bitstream is moved to the Jupyter session running on the target development platform. At this point, the developer can

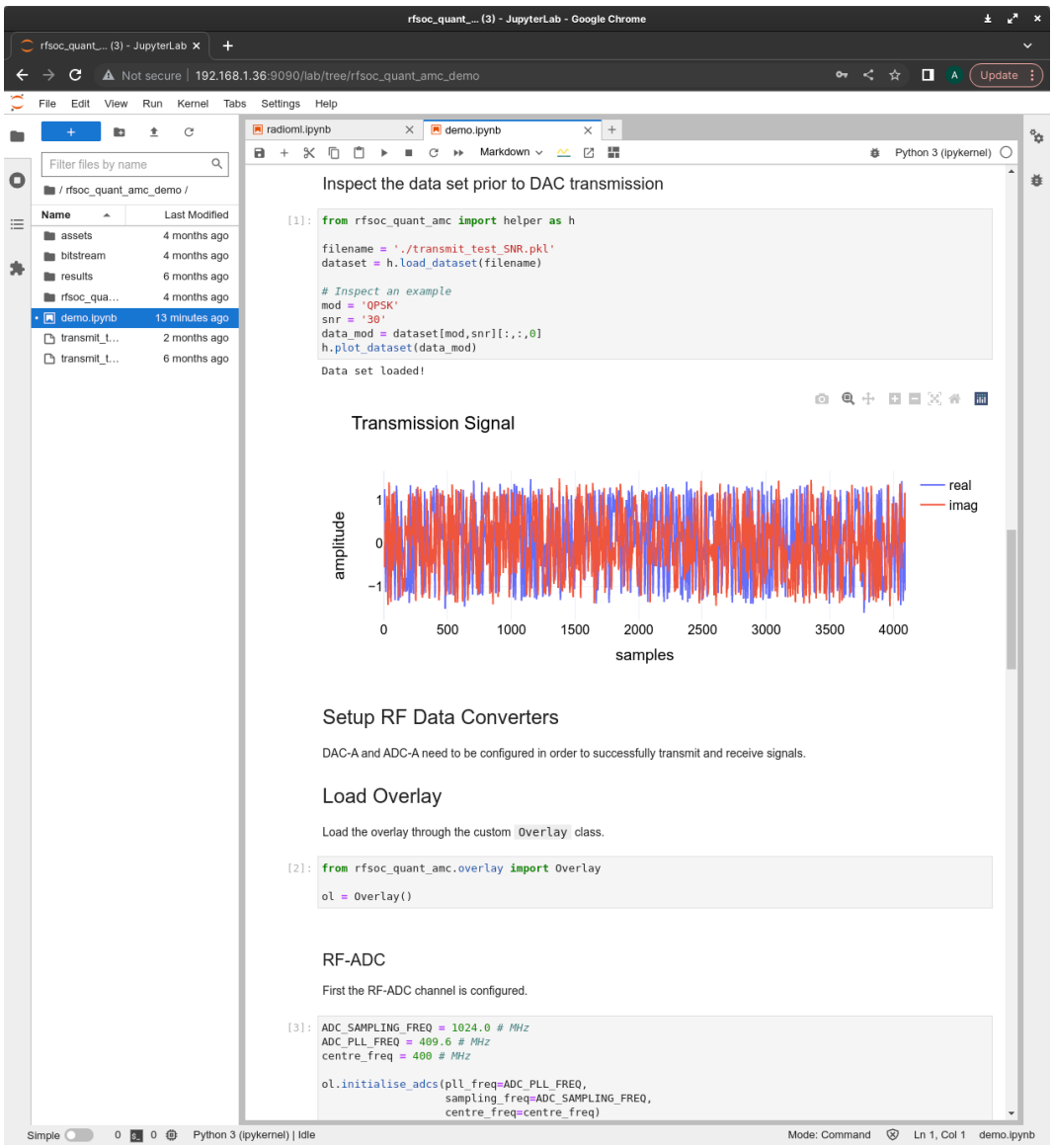



Figure 2.10 A Jupyter notebook running a PYNQ demo.

load the bitstream through the **`overlay`** class and import any associated PYNQ-enabled IP drivers. Through this initial set, the developer can interact with the IP cores in the loaded bitstream, and write custom drivers, functions, and interactive applications.

The power of PYNQ comes from its abstraction libraries, which make it easy to interact with the PL, and to develop and test algorithms on the PL. The typical development workflow involves testing the generated bitstream through PYNQ and adjusting the associated block design in Vivado to modify or correct any errors, which accelerates productivity when working with hardware/software co-design projects on the development platform.

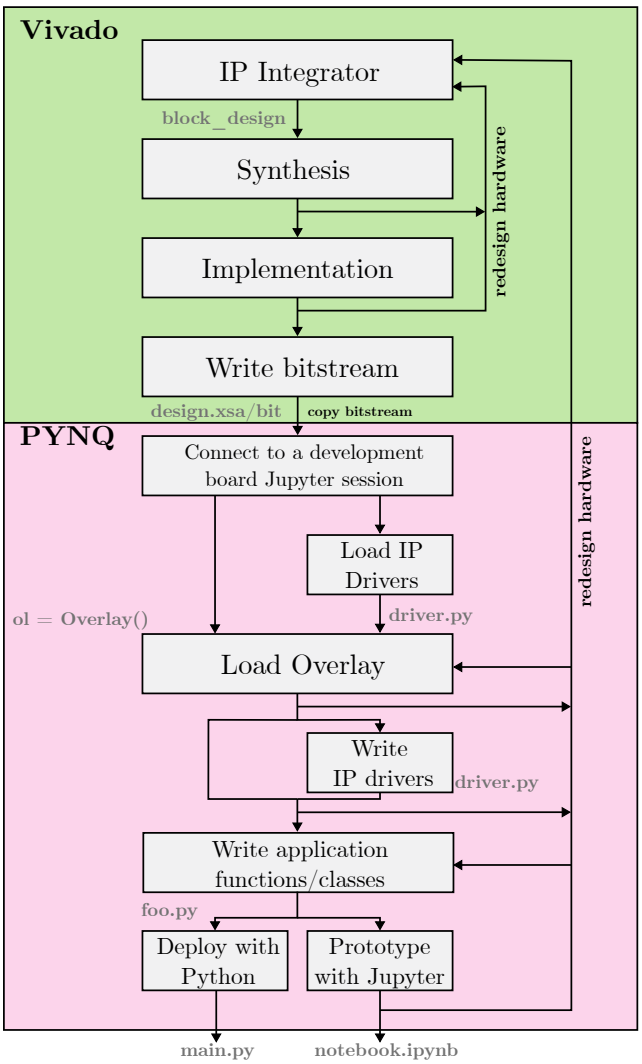



Figure 2.11 The typical PYNQ workflow.

### 2.1.6 Multirate Processing with AXI4-Stream

Modern RF systems often require signal processing pipelines that operate at different sampling rates across various processing stages. This is especially true when working with wideband signals that must be decimated or interpolated to match the capabilities of a downstream processing block, which is the case with the RFSoC due to its extremely high bandwidth RF-ADCs and RF-DACs. Multirate processing is a fundamental part of the RFSoC data pipeline, and understanding how it integrates with AXI4-Stream interfaces is critical for implementing efficient signal flows.

This section explains the streaming protocol used in AMD FPGAs, how data rates are represented through the AXI4-Stream protocol, and how decimation and interpolation filters enable flexible sampling rate conversion. The role of

the RFDC is also detailed, particularly in how it produces and accepts digitised signals.

### Overview of Multirate Processing

Multirate processing refers to the concept of handling data streams that operate at different sampling rates within the signal processing pipeline, by performing sampling rate transitions. In RF applications on the RFSoC, the incoming analogue signal is digitised at a high sampling rate to capture a wide bandwidth of data [58]. However, many downstream DSP components, such as filtering and symbol demapping, do not require the full bandwidth and can be performed at a reduced rate.

To manage the conversion of sampling rates, decimation and interpolation filters are used to adjust the sampling rates at different stages. These components are critical for optimising the usage of FPGA resources and ensuring that each module of the system receives or produces the appropriate data rate for the given task. The RFSoC platform supports multirate designs through the AXI4-Stream protocol which can transfer data between IP cores and the RFDC operating at different rates.

Performing functions at different sampling rates provides the capability for scalable and efficient signal processing pipelines, enabling real-time RF processing even when computational resources are limited.

### AXI4-Stream Protocol Fundamentals

IP Integrator (IPI), part of the AMD Vivado software, uses the Advanced Extensible Interface (AXI4) protocol to facilitate communications between PS and PL through AXI4 memory mapped and AXI4-Lite communications, as well as between IP cores with AXI4-Stream. AXI4-Stream is a lightweight data transfer protocol used to move data between IP cores in the PL for SoC devices with FPGAs.

There are three types of AXI4 protocols:

- **AXI4 Memory Mapped:** A communication interface that provides a standardised way to read and write data to one or more specific memory addresses.
- **AXI4-Lite:** Similar to AXI4 Memory Mapped, but only writes to one specific memory address at a time. Intended for light-weight control communications.

- **AXI4-Stream:** A protocol designed for high-throughput, continuous data streaming, making it suited for signal processing pipelines.

At the core of the AXI4-Stream protocol, simple handshake mechanisms between two IP cores, in a primary and secondary configuration, use the following signals:

- `TVALID`: which indicates which data samples are valid when sent.
- `TDATA`: carries the actual sample of data that is to be shared.
- `TREADY`: indicates to the upstream IP core that the secondary IP core is ready to receive data.
- `TLAST` (optional): is a flag signal that indicates when the last sample of a packet has been sent in the stream (when packet-based communication is being used).

The above signals are collected into a bus of signals that build the AXI4-Stream protocol.

Data from the primary IP core is transferred only when both its `TVALID` and the secondary IP core's `TREADY` are high on the same clock cycle. This ensures that a stream of samples can only flow through a signal processing pipeline while all blocks are ready to receive and send data. The diagram in Figure 2.12 illustrates the basic AXI4 handshake [59].

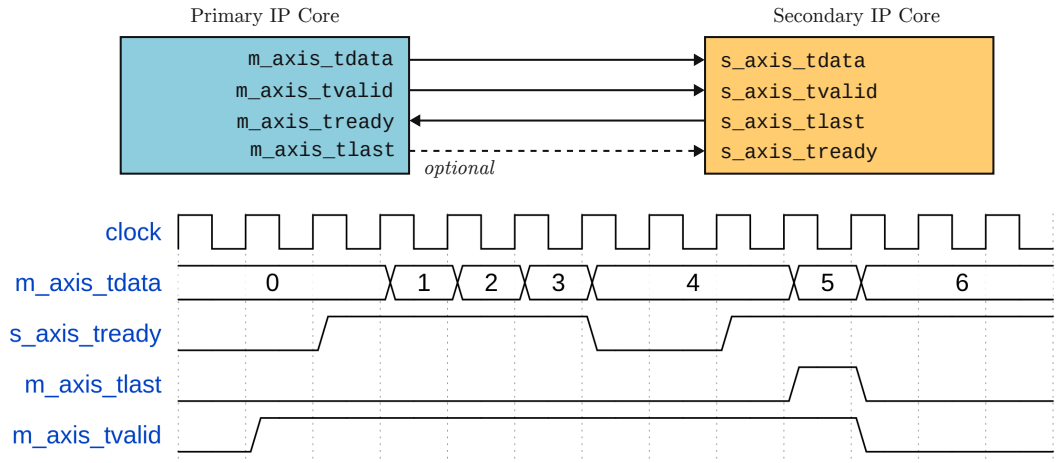



Figure 2.12 The basic AXI4-Stream protocol.

### Representing Lower Sampling Rates

When streaming data from high-speed ADCs (or to DACs), it is often necessary to represent lower effective sampling rates while operating at a fixed system clock. This enables real-time processing without changing the global clock rate.

This is normally achieved by applying decimation and interpolation filters (depending whether the samples are being received or transmitted), to change the ratio between the sampling rate and the clock rate. These operations therefore reduce or increase the sample rate without altering the streaming infrastructure through clock rate conversions. Further literature on decimation and interpolation filters can be found in [60].

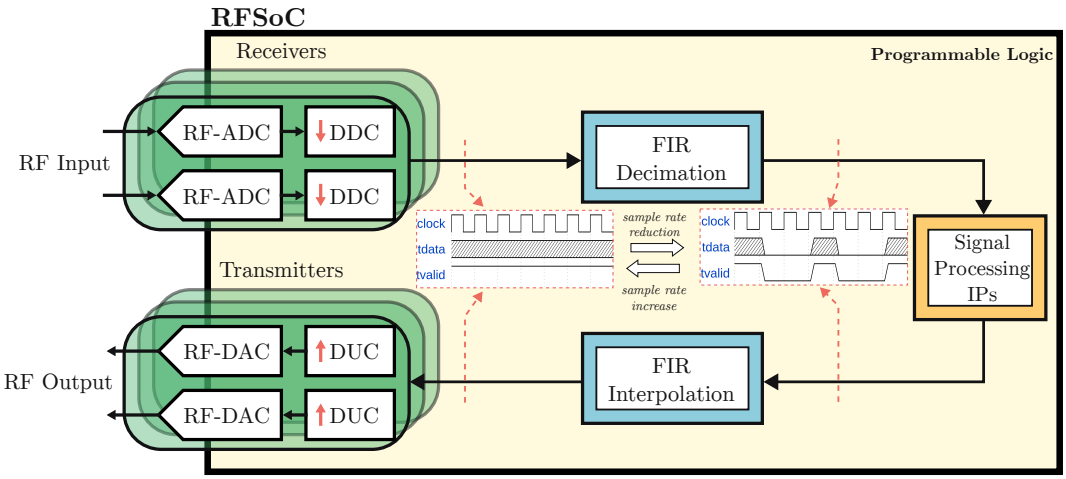



Figure 2.13 Sample rate reduction/increase through decimation/interpolation filter with a constant global clock.

Figure 2.13 illustrates samples being received from the RF-ADC and sent to the RF-DAC on the RFSoC development platform. When a signal is received by the RF-ADC, it is digitised and streamed into the PL in the AXI4-Stream protocol format. The signal is sampled at the clock rate used by the RF-ADC [10] and requires further rate reduction to be useable by the hardware IP cores, as the hardware accelerated DSP algorithm must operate at a lower, more manageable clock rate to meet timing constraints. This is achieved through a FIR decimation filter. The filter outputs another AXI4-Stream signal operating at the same clock rate as that used by the RF-ADC, but instead has a toggling 'valid' signal to simulate a lower overall sampling rate. Any subsequent signal processing stages can make use of the spare clock cycles to time-share resources (as presented by the architecture introduced in Chapter 4).

On the transmit side, the opposite occurs. The lower sample rate signal is increased by a FIR interpolation filter to reach the sample rate required by the RF-DAC before it is sent out of the radio.

## 2.2 Literature Review on Modulation Classification

One of the core tasks in DSA is spectrum sensing. Spectrum sensing provides awareness of nearby radio transmitters to inform spectrum allocation and avoid radio interference. Identifying and differentiating between nearby emitters, which have different behaviours and requirements, is important for developing a Cognitive Radio (CR) to dynamically use spare spectrum. Modulation recognition is the task of classifying the modulation type of a received radio signal as a first step towards understanding what type of signal and transmitter is nearby, also known as AMC. An overview of the topics covered in this section can be found in Figure 2.14.

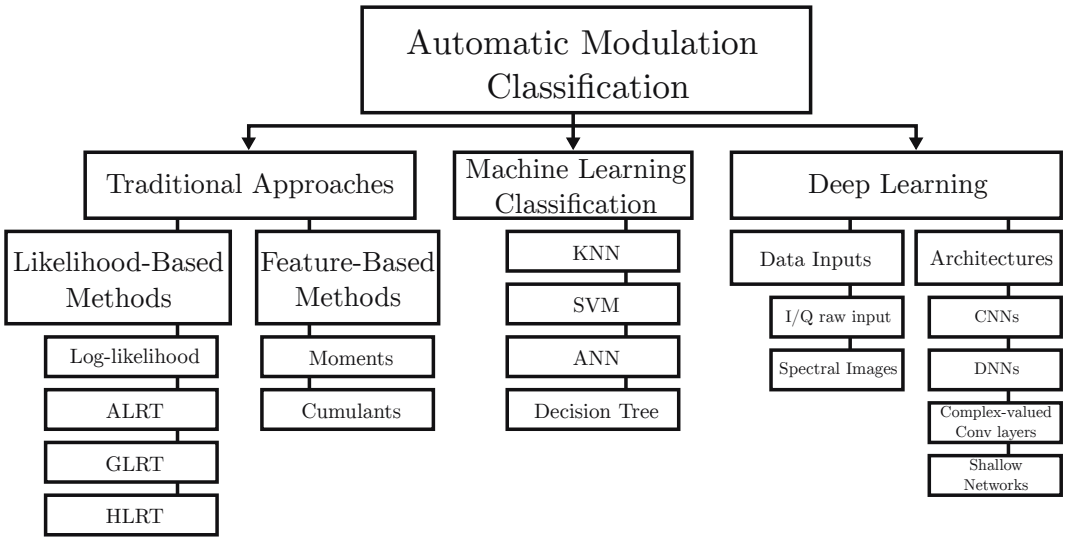



Figure 2.14 Overview of literature review topics for modulation classification.

### 2.2.1 Traditional Approaches

The task of modulation classification can be traced all the way back to 1969 by a paper by Weaver et al. [61], who investigated the use of pattern recognition techniques to automatically identify the modulation scheme between double-sideband amplitude modulation, single-sideband suppressed carrier, continuous

wave, FSK, and on-off keying (Morse code). Since then, the field of modulation classification has been divided into two general classes of AMC: Likelihood-Based (LB) and Feature-Based (FB) methods.

### Likelihood-Based Methods

A LB method models the probability of receiving a given signal under each possible modulation type and then selects the one that is most likely to have produced the received signal. In plain terms, if the receiver has knowledge of the modulation formats and understands how the noise and channel affect them, it can build a mathematical model of what the received signal should look like under each modulation type. The likelihood of the received signal is then calculated under each model, and the modulation type with the highest likelihood is chosen.

The basic likelihood function is defined as

$$\mathcal{L}(\theta|x_0,\ldots,x_{n-1}) = f(x_0,\ldots,x_{n-1}|\theta) = \prod_{i=0}^{n-1} f(x_i|\theta) \tag{2.14}$$

where this function expresses how likely a set of observed data points $x_0,\ldots,x_{n-1}$ is, given a parameter $\theta$. Here, $f(x_0,\ldots,x_{n-1}|\theta)$ is the joint Probability Density Function (PDF) of the observed data. If the data samples are assumed to be independent and identically distributed, the joint PDF can be written as a product of individual probability densities. To simplify the calculation, and because the natural logarithm is monotonically increasing, the log-likelihood function is often used:

$$\log(\mathcal{L}(\theta|x_0,\ldots,x_{n-1})) = \sum_{i=0}^{n-1} \log f(x_i|\theta) \tag{2.15}$$

Log LB methods aim to find the maximum value of Equation 2.15 by optimising parameters $\theta$ using the Maximum Likelihood Estimator (MLE):

$$\hat{\theta} = \arg\max_{\theta}(\log(\mathcal{L}(\theta|x_0,\ldots,x_{n-1}))) \tag{2.16}$$

Over the years, more advanced likelihood-based methods for AMC have been developed [5]. The Average Likelihood Ratio Test (ALRT) addresses one of the limitations of MLE-based classification: in real scenarios, channel parameters such as phase, frequency offset, or noise power may be unknown. Instead of estimating them, ALRT averages the likelihood over a range of possible values, averaging out the uncertainty. In contrast, the Generalised

Likelihood Ratio Test (GLRT) method estimates the unknown parameters first and then inserts them into the likelihood function before applying the MLE. The Hybrid Likelihood Ratio Test (HLRT) approach combines both techniques: it estimates some relevant parameters while averaging over others that are less critical [62].

### Feature-Based Methods

A FB method is a type of approach used in modulation classification that relies on extracting the signal characteristics or 'features' from the input data before performing classification. Features are typically chosen based on prior knowledge of the signal's characteristics [5].

FB methods are split into two categories: feature extraction methods and ML methods.

Feature extraction methods primarily focus on the specific spectral characteristics of the different modulation types. The three main features are: amplitude, phase, and frequency, because each modulation scheme produces different amplitude, phase, and frequency properties. One such technique group is statistical feature extraction, which is useful in low SNR environments, and disruptive channel conditions such as frequency offsets. The most common feature extraction methods are moment-based and cumulant-based techniques [63].

**Moments** describe the signal's PDF, where each $n$th-order moment of a signal $x(t)$ is the expected value of $x(t)^n$. The $n$th moment is defined by

$$m_n = \mathbb{E}\{x^n\} = \int_{-\infty}^{\infty} x^n f(x)\, dx \tag{2.17}$$

where the integral computes the expected value by integrating over $x^n$ weighted by the PDF $f(x)$. The central moment is a further extension of the moment where it is taken about the mean of the distribution and not about zero. A $n$th-order central moment is defined as [63], [64].

$$\mu'_n = \mathbb{E}\{(x-\mu)^n\} = \int_{-\infty}^{\infty} (x-\mu)^n f(x)\, dx \tag{2.18}$$

where $\mu = \mathbb{E}[x]$ is the mean of $x$ and $\mu'_n$ is the $n$th central moment. For AMC, central moments help describe important statistical properties of the received signal. The commonly used moments in AMC are: the mean $\mu$ (first moment) which defines the centering of the PDF; variance $\sigma^2$ (second moment), which

defines the spread of samples or the energy; skewness (third moment), which captures asymmetry of the data; and kurtosis (fourth moment), which describes the 'peakedness' of the distribution [63], [64].

Different modulation types (e.g. AM, FM, PSK, and QAM) produce different patterns of variance, skewness, and kurtosis because of their underlying symbol structure and how their amplitude and phase behave over time. By extracting these central moments from a received signal, a feature vector can be built that acts like the 'signature' of the modulation type.

**Cumulant-based** feature extraction is an extension to moment-based feature extraction. Moment-based feature extraction techniques rely on statistical moments to describe the shape and energy distribution of the signal (first and second order moments). Higher-order moments assist with discerning between modulation types. Cumulant-based feature extraction makes use of this by isolating independent statistical properties [63].

Cumulants can be obtained from moments with

$$K_n = \mu_n - \sum_{i=1}^{n-1} \binom{n-1}{i} K_{n-1}\mu_i \tag{2.19}$$

where $K_n$ is the $n$th order cumulant, $\mu_n$ is the $n$th order raw moment, and $\binom{n-1}{i}$ is the binomial coefficient. This recursive relationship allows cumulants to be systematically derived from moments up to any order [65].

In practice, cumulants capture similar statistical measures as in moments where higher-order cumulants capture increasingly fine details about the distribution.

For AMC, feature vectors composed of the second, fourth, and sixth-order cumulants are commonly used to distinguish between modulation schemes. Since higher-order cumulants are insensitive to Gaussian noise and scale linearly with signal power, they provide a compact and noise-robust statistical representation of the signal, enabling accurate classification even under challenging channel conditions [5], [63], [66].

### Machine Learning-based Classification

After feature extraction, the next stage in the AMC process is classification. The extracted features, such as moments and cumulants, form a feature vector that characterises the received signal. A machine learning classifier is then used to map the feature vector to one of the known modulation types [5], [63].

Several types of classifiers commonly are used in AMC [5]:

- **k-Nearest Neighbours (KNN)**: A simple, non-parametric algorithm that classifies a sample based on the majority class of its $k$ nearest neighbours in the feature space.
- **Support Vector Machine (SVM)**: A supervised learning method that finds the optimal hyperplane to separate different classes by maximizing the distance between them.
- **ANN**: A model composed of layers of interconnected neurons that can learn complex, non-linear relationships between input features and output classes.
- **Decision Tree (DT)**: A rule-based model that splits the feature space into regions using decision rules based on feature thresholds, forming a tree structure.

Each classifier type offers different trade-offs between computational complexity, interpretability, and classification performance. The choice of classifier depends on the system requirements of the task such as real-time constraints and accuracy needs.

A high-level overview of the traditional classification flow is shown in Figure 2.15, where a received signal $x(t)$ has its features extracted and then passed through a classifier to produce the predicted modulation label $\hat{y}$.

This approach to AMC relies on the careful extraction of features from the received signal before a machine learning model makes a prediction. As DL progressed, ML algorithms began to adopt more data-driven approaches, where models could learn to extract features automatically from raw samples. Prior knowledge of a signal's unique features was no longer required to build a successful ML classifier.

### 2.2.2 Deep Learning for Automatic Modulation Classification

The use of CNNs for AMC represents a major shift from traditional feature-based methods to data-driven learning methods. Rather than manually extracting features, such as moments or cumulants, CNN-based systems learn to automatically extract and optimise their own features directly from the raw

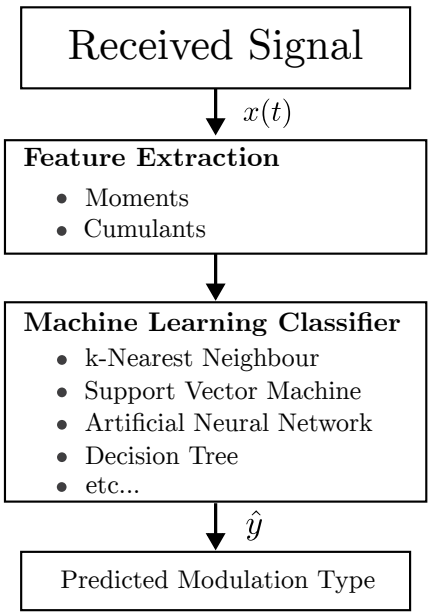



Figure 2.15 AMC ML classifier system.

input data. In the case of AMC this can be the raw in-phase and quadrature (I/Q) samples.

The transition from classical ML techniques to fully embracing the power of DL was popularised by the work of O'Shea et al. [19] through their paper on DL for radio signal classification. In this work, they demonstrated that relatively simple CNN architectures (compared to the larger models in the image processing field at the time) could achieve state-of-the-art modulation classification performance by learning directly from raw signal data that had undergone minimal processing. Their results showed that DL models could outperform traditional FB techniques, especially under harsh channel conditions such as low Signal-to-Noise Ratios (SNRs) and channel impairments.

A core contribution of this work was also the release of the RadioML dataset [45] (discussed in Section 2.2.3), a publicly available dataset of synthetically generated radio signals under various realistic channel conditions. The dataset includes a wide range of analogue and digital modulation schemes each labelled with the corresponding modulation type and the SNR that was added.

The dataset quickly became a benchmark for a variety of DL model improvements across the field of AI for RF, similar to famous benchmark datasets in the field of image processing such as the MNIST dataset [67], ImageNet [68], and CIFAR-10 [69].

A number of studies have since reinforced and refined this baseline. West et al. [70] explored deeper CNN architectures and found that deeper networks did

not lead to a significant improvement in classification accuracy. They proposed that future studies should focus on solving problems with synchronisation and channel equalisation. Similarly, Wang et al [37] compared CNNs to fully-connected DNNs and concluded that CNNs offer superior feature extraction capabilities due to their ability to exploit local spatial patterns in I/Q samples through convolutional layers. They also found that the representation of the input data affected the accuracy, citing improved accuracy when using constellation diagrams as inputs instead of raw I/Q samples.

The use of signal-to-image transformations forms a growing theme in the literature. Peng et al. [71] where they applied deep CNNs to constellation diagram images and demonstrated classification improvements. These works collectively underline the importance of how data is structured before entering the network.

In contrast, O'Shea et al.'s later work [72] reaffirmed the strength of CNNs operating directly on raw I/Q samples. Their study demonstrated excellent performance on signals captured over-the-air, validating the practical applicability of DL models in real-world RF environments. They also showed that networks trained on synthetic data can generalise to live RF signals, establishing confidence in the practical deployment of these models.

Other contributions have focused on architectural innovation. Krzyston et al. [73] proposed complex-valued convolutional layers that better preserve the relationships between I and Q components. In their approach, they found that this improved accuracy in low-SNR conditions.

Together, these developments have provided valuable insights into the types of network architectures best suited to modulation classification tasks. By examining how different architectural choices, data representations, and learning strategies affect performance under realistic conditions, the research community has established a solid foundation for future development. These insights not only clarify the types of networks that are most effective for AMC, but also help guide the design of hardware acceleration architectures, such as those explored in this thesis, that are compatible with the constraints of real-time embedded radio systems.

### 2.2.3 The RadioML Dataset

The dataset introduced by [19], named 'RadioML', consists of a collection of labelled frames of synthetically generated samples representing various modulation schemes transmitted over different communication channels. Figure 2.16

illustrates the RadioML generation process. It was created to address the need for standardised datasets in the field of radio signal processing and AI while also introducing how CNNs can be applied to PHY wireless communications tasks. The dataset includes samples from a variety of modulation schemes including: Amplitude Modulation Double Sideband (AM-DSB), Amplitude Modulation Single Sideband (AM-SSB), Wideband Frequency Modulation (WBFM), Binary Phase Shift Keying (BPSK), Quadrature Phase Shift Keying (QPSK), 8 Phase Shift Keying (8PSK), QAM 16 and 64, 4-level Pulse Amplitude Modulation (PAM4), Gaussian Frequency Shift Keying (GFSK), and Continuous Phase Shift Keying (CPFSK).

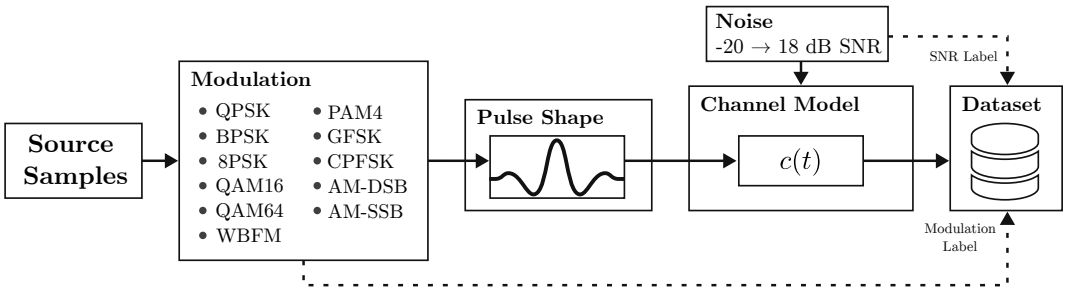



Figure 2.16 Generation process for the RadioML dataset.

Each frame of modulated I/Q samples is subjected to noise across a wide range of SNR levels and channel effects to simulate real-world communications scenarios.

By providing a standardised set of labelled I/Q data, the RadioML dataset facilitates the development and evaluation of machine learning models for radio signal processing tasks and serves as a foundation for developing networks for other wireless communication problems. It enables researchers to fairly compare the performance of different algorithms, verify hardware-implemented solutions, and ultimately push forward the state-of-the-art in wireless communication systems.

One of the biggest challenges in the field of AI and DL is the collection and organisation of training data. The quality of the dataset is as important, if not more, than the process of training the model itself. In a classification DL, the data must be carefully labelled while ensuring that it covers a wide range of scenarios and avoids inherent bias. ML in wireless communications benefits from the availability of simulation tools that can accurately replicate real world communication scenarios.

Radio communication signals are inherently synthetic and are generated deterministically through modulation, pulse shaping, coding, preambles, and

other configurable parameters. These deterministic rules are also applied when generating synthetic data where a channel model function is used to simulate the transmitted signal passing through a destructive and noisy environment.

The RadioML dataset generates a variety of digitally and analogue modulated signals using eleven different modulation schemes. The creators modulated real voice and text datasets onto the signals and used block randomisers to ensure that the bits were equiprobable. The generation of the synthetic data was undertaken using the GNU Radio [24] software toolkit in Python.

The radio signals are passed through a dynamic channel model that simulates a wide range of real-world effects, including:

- Time-varying multipath fading of the channel impulse response
- Random drift in the carrier frequency oscillator
- Random walk sampling time offsets
- AWGN
- Varying scaling, phase offsets, impulsive noise bursts, and timing dilation

After passing through the channel, a signal is segmented into frames of 128 complex-valued samples, where each sample consists of two channels representing the I/Q components of the signal.

The dataset is then labelled with the modulation scheme type and AWGN SNR before being stored in a Python pickle file using 32-bit floating-point precision. The total size of the dataset is approximately 500MB.

The resulting dataset provides a Python dictionary consisting of 11 modulation schemes: 8 digital and 3 analogue modulations, commonly used in many wireless communications applications. The modulation schemes used are: QPSK, BPSK, QAM16, QAM64, 8PSK, PAM4, GFSK, CPFSK for the digital modulations, and WB-FM, AM-SSB, and AM-DSB for the analogue modulations. Each data symbol is modulated with 8 Samples-Per-Symbol (SPS). Further information about the RadioML dataset and its generation can be found at [45].

## 2.3 Chapter Conclusion

This chapter has introduced digital modulation, the most common digital modulation schemes, and the fundamentals of pulse shaping. Channel effects

and channel models were reviewed, including multipath fading models. The concept of AMC was introduced, followed by an explanation of the different approaches to the task. Modern AMC techniques using CNNs were presented alongside the pivotal RadioML work. The RadioML dataset and its generation process was also covered. The AMD RFSoC and PYNQ was introduced, followed by multrate processing through AXI4-Stream on the RFSoC. Finally, the topic of DL challenges for wireless communications was discussed, highlighting the need for a CNN accelerator that supports the flow of samples in a radio signal processing pipeline.

This chapter outlined the essential connections between digital communication theory and modern DL-based approaches. It demonstrated how core concepts such as digital modulation schemes, channel effects, and signal representations form the basis for tasks like AMC. The limitations of traditional solutions in dynamic radio environments were contrasted with the performance advantages of CNNs. Additionally, the chapter introduced key hardware considerations, such as the RFSoC platform and the importance of effective data movement using AXI4 protocols, setting the stage for understanding the need for custom hardware accelerators capable of processing real-time radio signals efficiently.

The main motivation for this chapter was to introduce the concepts required to understand how a custom architecture can be applied to wireless communications tasks. The following chapter will cover the fundamentals of DL and DL hardware accelerators.

# Chapter 3

# Hardware Architectures for Deep Learning Inference

The material covered in this chapter underpins the application of DL and AI to PHY SDR FPGA receivers presented in the subsequent chapters of this thesis.

## 3.1 Introduction

DL is a field where problems are solved by training models on datasets of input-output examples. Using layers of connected neurons and sufficient training resources, these models can learn to perform a wide range of tasks: from object recognition in images, to signal classification in the frequency spectrum.

Getting a model to work is just the first step. The next challenge is deploying it at the edge, where resources are limited. Designing an accelerator that can perform inference with high throughput and low energy consumption is difficult, but when done correctly, it enables powerful DL applications to run directly on edge devices.

Mapping neural network models to FPGA hardware requires a solid understanding of the core building blocks of neural networks, which are introduced in the following sections.

## 3.2 Neural Networks

ANNs are a class of ML models inspired by the structure and function of the human brain. They are composed of interconnected processing units, or neurons, which collectively learn to model complex patterns in data. Over the last decade, neural networks (and particularly deep learning) have become a

cornerstone of modern AI, achieving remarkable performance across a range of tasks, from image and speech recognition to wireless signal classification [25], [74].

At its core, a neural network is a computational model that maps a set of input features to an output through a series of interconnected processing units, known as neurons. As illustrated in Figure 3.1, a typical use case involves a classification task, where the network takes input features and produces a predicted class label.

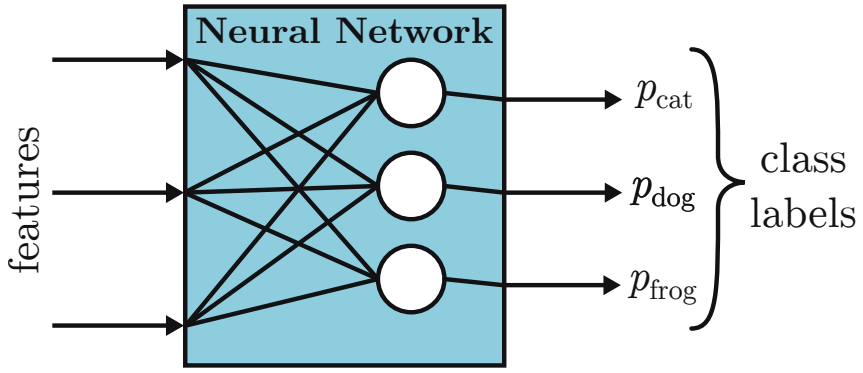



Figure 3.1 Example interface of a Neural Network.

To understand how this output is generated, it is helpful to examine the internal structure of the network. Figure 3.1 presents a conceptual view of the individual neurons. Each neuron processes the input features by computing a weighted sum followed by a non-linear activation function. These computations are passed forward through the network layers, producing a set of output probabilities corresponding to different class labels, such as 'cat' 'dog' or 'frog'. The class with the highest probability is selected as the final predicted label.

### 3.2.1 The Neuron

An artificial neuron serves as the fundamental building block in ANNs, drawing inspiration from biological neurons in the human brain. Mathematically, it computes an output by aggregating its weighted inputs, adding a bias term, and then applying a non-linear activation function. This process enables the modelling of complex, non-linear relationships within data. The output $y$ of a neuron is given by

$$y = f\left(\sum_{n=0}^{N-1} x_n w_n + b\right) \tag{3.1}$$

where a set of $N$ input features $x_n$ are each multiplied with an associated weight $w_n$. Each input and weight product is summed together, often with an additional bias term $b$ specific to the neuron. The bias acts as a constant additive offset, allowing the neuron to shift its activation threshold and adjust its decision boundary. The resulting sum is then passed through the activation function to introduce non-linearity. Figure 3.2 illustrates the building components of a neuron[10], [25], [75].

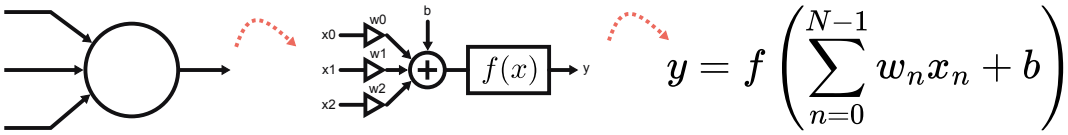



Figure 3.2 An Artificial Neuron.

### Activation Functions

Activation functions are critical components in ANNs, as they introduce non-linearity to the network. After a neuron computes the weighted sum of its inputs, the activation function $f$ is applied to determine the neuron's output [25]. Without activation functions, the network would only be able to model linear relationships, severely limiting its capacity to learn complex patterns. Therefore, activation functions allow neural networks to approximate intricate, non-linear mappings from input to output, enabling them to perform tasks such as classification and regression. Three of the most common activation functions are shown in Figure 3.3.

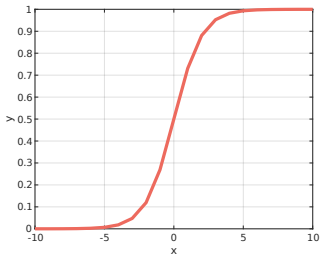
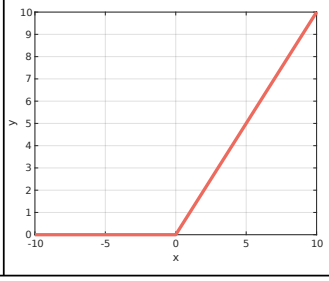
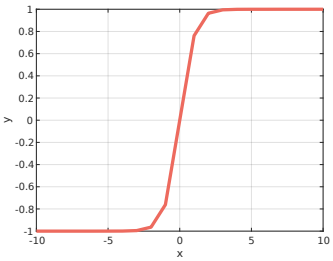
| **Sigmoid** | **ReLU** | **Tanh** |
|---|---|---|
| $\frac{1}{1+e^{-x}}$ | $max(0, x)$ | $\frac{2}{1+e^{-2t}} - 1$ |
|  |  |  |

Figure 3.3 Plots and equations of common activation functions.

The Sigmoid function maps any real-valued input to the range $(0, 1)$, making it suitable for models where the output can be interpreted as a probability. The

Hyperbolic Tangent (Tanh) function is a scaled version of the sigmoid function that maps inputs to the range $(-1, 1)$. The Rectified Linear Unit (ReLU) function is currently one of the most widely used activation functions due to its simplicity and efficiency for implementation in accelerators. It is defined as a simple *max* function where only positive valued numbers are retained, and negative inputs are set to zero. ReLU is computationally efficient to implement.

In summary, activation functions are a crucial component of neural networks, enabling them to model complex and non-linear relationships. While traditional functions like sigmoid, tanh, and ReLU have been widely adopted, research into alternative activation functions continues to evolve as DL architectural research progress. Constant advancements are introducing novel activation functions aimed at addressing limitations such as vanishing gradients or dead neurons. One such example is the Swish and Mish activation functions that outperform ReLU in some DL applications [76], [77]. The exploration of new activation functions remains an active area of research, as subtle changes in these non-linearities can significantly influence training dynamics and final model performance.

### 3.2.2 Deep Neural Networks

A Deep Neural Network (DNN) is an extension of the basic neural network concept, composed of multiple layers of neurons stacked one after another. These are also known as Multi-Layer Perceptrons (MLPs). Unlike shallow networks, which typically consist of one or two layers, DNNs incorporate many layers, allowing them to learn hierarchical and increasingly abstract representations of data. The term 'deep' refers to the depth of the network, which is the number of layers through which data is transformed. Each layer of a DNN can perform a large range of tasks, from extracting progressively higher order features from the input signal, reducing the dimensionality of the signal, or performing analysis on its spatial and temporal patterns [25]. For example, in image classification tasks, earlier layers may detect simple features such as edges and textures, while deeper layers can recognise complex shapes and objects.

Deep architectures can be constructed using various types of layers such as fully-connected layers, convolutional layers, recurrent layers, and many more, each contributing its own unique characteristic [25]. This thesis focuses on hardware implementations of CNNs, a type of DNN that performs many of its feature extractions through the use of convolutional layers. Therefore, the

following two sections will explore the two foundational layer types: fully-connected and convolutional. Following the overview of these two layers, the subsequent sections will describe how deep networks are trained using optimisation techniques such as backpropagation.

### 3.2.3 Fully-Connected Layers

A Fully-Connected (FC) layer, also referred to as a dense layer, is a core component of ANNs, particularly within architectures such as MLPs, CNNs, and transformer networks. The term 'fully-connected' arises from the fact that each neuron in the layer is connected to every neuron in both the preceding and succeeding layers. This dense connectivity ensures that all inputs are considered, but is also makes FC layers computationally expensive.

In an FC layer, each input from the previous layer is multiplied by a corresponding weight, summed across all inputs, and optionally offset by a bias term. The resulting value is then passed through a non-linear activation function, such as a ReLU, sigmoid, or tanh, to enable the network to capture complex relationships within the data. A diagram of a neural network with FC layers can be seen in Figure 3.4.

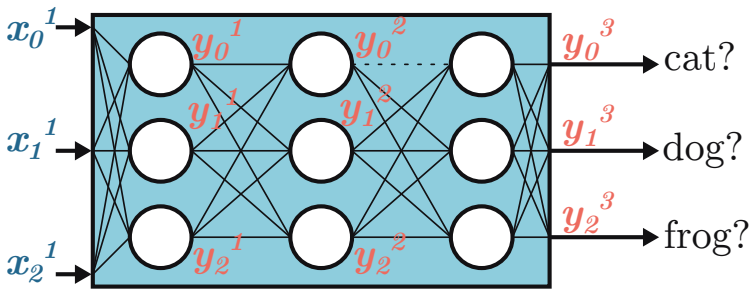



Figure 3.4 Graphical representation of FC layers in a neural network.

The output of a FC layer is a vector, with each element corresponding to the output of a neuron in that layer. Mathematically, this operation is expressed as

$$\boldsymbol{y}_n^l = f\left(\sum_{m=0}^{M-1} \boldsymbol{\Theta}_{mn}^l \boldsymbol{x}_m^l + \boldsymbol{b}_n^l\right), \quad \text{for } n = 0, \ldots, N-1 \tag{3.2}$$

where $\boldsymbol{y}_n^l$ represents the output of the $n$-th neuron in a layer $l$, computed by applying a non-linear activation function $f(\cdot)$ to the weighted sum of its inputs. The inputs to the layer are denoted by $\boldsymbol{x}_m^l$, where $m = 0, \ldots, M-1$ indexes the neurons in the previous layer. Each input $\boldsymbol{x}_m^l$ is multiplied by a corresponding weight $\boldsymbol{W}_{mn}^l$, where every possible connection between input size $M$ and output

size $N$ exists. The input and weight product is summed together to compute the total input to neuron $n$, and a bias term $\boldsymbol{b}_n^l$ is added [78].

Equation 3.2 can be simplified to the form

$$\boldsymbol{y}_{1\times N}^l = f(\boldsymbol{\Theta}_{M\times N}^l \boldsymbol{x}_{M\times 1}^l + \boldsymbol{b}_{1\times N}^l) \tag{3.3}$$

where a single FC layer, from an algebraic point of view, can be seen as a vector-matrix product with a per-neuron additive scalar bias $b$ after being passed to an activation function.

### 3.2.4 Convolutional Layers

Convolutional layers are a cornerstone of modern DNN models, particularly in tasks involving spatial or temporal data such as images, audio, or time series. Unlike FC layers, which connect every input neuron to every output neuron, convolutional layers introduce a concept of local connections and weight sharing. This not only reduces the number of learnable parameters, making the network more computationally efficient for larger inputs, but also allows the model to learn spatially local patterns. Additionally, parameter sharing improves the statistical efficiency of a neural network and increases its train-ability [78].

A convolutional layer transforms input feature maps $\boldsymbol{X}^{\text{conv}}$ into outputs feature map $\boldsymbol{Y}^{\text{conv}}$, each composed of multiple units (or neurons) arranged in a multi-dimensional grid-like topology. Unlike FC layers, which disregard spatial structure, convolutional layers preserve the local relationships in the input by applying learnable filters across the spatial dimensions on an input feature map. Each output feature map is the result of convolving a corresponding kernel with the input feature maps, followed by an optional bias term addition and an activation function [25], [78].

#### 2D Convolutional Layers

A simple and widely used variant of the convolution is the 2-dimensional (2D) convolutional layer.

Let the input to the convolutional layer be a 2D feature map $\boldsymbol{X}^{\text{conv}} \in \mathbb{R}^{H\times W}$, and let the kernel (filter) be $\boldsymbol{\Theta}^{\text{conv}} \in \mathbb{R}^{J\times K}$. The output feature map $\boldsymbol{Y} \in \mathbb{R}^{(H-J+1)\times(W-K+1)}$ is computed by sliding the kernel over the input spatially and performing element-wise multiplication and summation. The 2-dimensional convolution is defined as:

$$\boldsymbol{Y}^{\text{conv}}[a,b] = \sum_{J-1}^{j=0} \sum_{k=0}^{K-1} \boldsymbol{X}^{\text{conv}}[a-j, b-k] \cdot \boldsymbol{\Theta}^{\text{conv}}[j,k]. \tag{3.4}$$

However, most modern neural network frameworks implement a cross-correlation instead, where the kernel is not flipped during operation [79], [80]. This leads to a slightly altered equation.

$$\widetilde{\boldsymbol{Y}}^{\text{conv}}[a,b] = \sum_{j=0}^{J-1} \sum_{k=0}^{K-1} \boldsymbol{X}^{\text{conv}}[a+j, b+k] \cdot \boldsymbol{\Theta}^{\text{conv}}[j,k]. \tag{3.5}$$

The distinction between convolution and cross-correlation becomes irrelevant as the kernel weights are learned throughout the training process. For this reason, cross-correlation is more commonly used in deep learning libraries due to its simplicity. A diagram of the 2D convolution process can be seen in Figure 3.5.

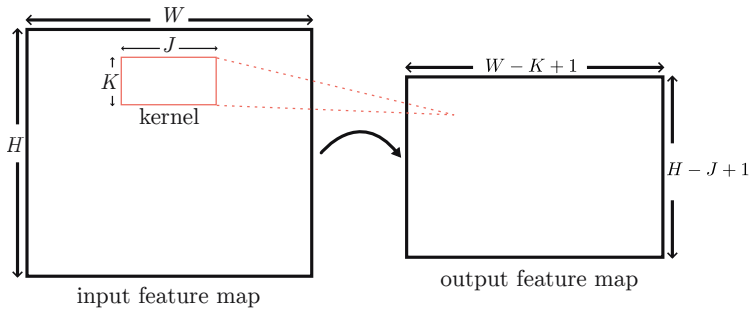



Figure 3.5 2D convolution of input feature map and kernel.

### 3D Convolutional Layers

In this thesis, 3-dimensional convolutions are primarily used.

Let the input to the convolutional layer be a 3-dimensional input feature map $\boldsymbol{X}^{\text{conv}} \in \mathbb{R}^{C \times H \times W}$, and let the kernel (filter) be $\boldsymbol{\Theta}^{\text{conv}} \in \mathbb{R}^{N \times C \times J \times K}$. The output feature map $\boldsymbol{Y} \in \mathbb{R}^{N \times (H-J+1) \times (W-K+1)}$ is computed by sliding the kernel over the input feature map.

Two common hyperparameters of convolution are padding and stride, which control the output spatial dimensions and the step size of kernel, respectively. The 3-dimensional convolution operation (without padding and a stride of 1) is defined as:

$$\boldsymbol{Y}^{\text{conv}}[n,a,b] = \sum_{c=0}^{C-1} \sum_{j=0}^{J-1} \sum_{k=0}^{K-1} \boldsymbol{X}^{\text{conv}}[c, m+j, n+k] \cdot \boldsymbol{\Theta}^{\text{conv}}[n,c,j,k] \tag{3.6}$$

Each kernel in the set $\boldsymbol{\Theta}^{\text{conv}}$ spans all input channels, allowing the model to combine information across feature dimensions. The output is produced by convolving each kernel with the full input (including the channels), resulting in $N$ distinct output feature maps – one for each kernel [25], [75], [78].

The same operation can be represented as a nested loop:

Listing 3.1 Nested For Loops for convolutions.

```
for n in range(N):  # Output channels
    for a in range(H):  # Input/Output height
        for b in range(W):  # Input/Output width
            for c in range(C):  # Input channels
                for j in range(J):  # Kernel height
                    for k in range(K):  # Kernel width
                        Y[n][a][b] += X[c][a + j][b + k] * Theta[n][c][j][k]
```

The convolutional layer performs a more structured and computationally complex operation compared to a FC layer. However, this complexity brings a significant advantage: it drastically reduces the number of trainable parameters by reusing a small set of weights/kernels across the input. Accelerating convolutional layers becomes a crucial objective focusing on optimising the core convolution operation to enhance throughput and reduce latency, especially in real-time or resource-constrained environments.

### 3.2.5 Neural Network Training

Training a neural network involves finding the optimal values for its parameters (weights and biases) so that it can perform a specific task successfully – such as classification – with high accuracy. This is achieved by presenting the network with input-output pairs, measuring how close its predictions are using a loss function, and adjusting its parameters accordingly, via gradient-based optimisation. In supervised learning, a neural network can be trained, through this process of presenting repeated examples and updating its parameters, to achieve a model that can perform a desired task with high accuracy.

This section will cover the core steps of training a neural network including calculating loss through loss functions, backpropagation, and parameter updates, as well as techniques that can assist in reducing phenomena like overfitting.

#### Loss Functions

In neural network training, the goal is to minimise the difference between the network's predictions and the actual target values, which is achieved using a loss function. The loss function guides the model to adjust its parameters

during training, improving its accuracy over time. In this thesis, only classification supervised learning applications are considered – so only loss functions associated with classifications-based models are reviewed.

In multi-class classification tasks, the output layer of a neural network typically produces raw values, known as logits, which represent the unnormalised prediction scores of each class.

The Softmax function is applied to the logits $z_i$ from the output layer of the network and converts them into probabilities $p_i$ [25]. The function is defined as:

$$p_i = \frac{e^{z_i}}{\sum_{j=0}^{C-1} e^{z^j}} \tag{3.7}$$

where $p_i$ is the predicted probability for class $i$ and $z_i$ is the logit for each class $i$ over $C$ total classes. The sum in the denominator ensures that the output values are normalised, meaning that the probabilities across all classes sum to one.

For classification tasks, Cross-Entropy Loss is the most commonly used loss function. It measures the difference between the predicted class probabilities and the actual class labels, encouraging the network to output higher probabilities for the correct class.

The Cross-Entropy Loss function is defined as [25]:

$$J(y_i, p_i) = -\sum_{i=0}^{C-1} y_i \log(p_i) \tag{3.8}$$

where:

- $C$ is the number of classes.
- $y_i$ is the actual class label (with values 0 or 1 for each class in a one-hot encoding).
- $p_i$ is the predicted probability for class $i$

This loss function penalises incorrect predictions more heavily when the model is confident about the wrong answer, thus driving the network to output more accurate class probabilities. Cross-Entropy Loss is particularly effective in multi-class classification tasks, making it a standard choice in many classification-based neural network applications.

### Backpropagation

Backpropagation is the key algorithm behind the effective training of neural networks. This algorithm is used to train neural networks by updating their parameters based on the computed loss, as discussed in Section 3.2.5. Once the forward pass computes predictions and an associated loss value, backpropagation calculates the gradients of the loss with respect to each weight and bias in the network by propagating the gradients back up the network, hence the name 'backpropagation' [81].

In a neural network's forward pass, an input feature map is propagated through each layer and neuron of a network, producing intermediate outputs at every stage. At the final layer, the network's prediction is compared to the true class using a loss function to compute a scalar loss value (e.g. Cross-Entropy Loss). The objective then becomes to adjust the model's weights so that the next time the same (or a similar) input is passed through, the resulting calculated loss is smaller.

Backpropagation is the terminology for minimising the loss function $J(\cdot)$. The ultimate goal is to compute

$$\min_{\theta} J(\Theta) \tag{3.9}$$

where an optimal set of parameters, $\Theta$, must be found that minimises $J(\cdot)$. This is performed by updating each of the parameters in the network to produce a model that minimises the loss.

Backpropagation uses the chain rule from calculus to calculate the derivative of a function that is composed of other functions. In feed-forward neural networks, this principle is essential, as it enables the derivative of the loss function to be propagated backward through the layers of the network. Figure 3.6 illustrates the process of backpropagation.

In the forward pass (green arrows of Figure 3.6), the output $y$ is calculated as a function of $f(\theta, b)$ using the weight or kernel parameter $\theta$ and the bias $b$. The backward pass (red arrows of the figure) begins by receiving the gradient of the loss function with respect to the output, denoted as $\frac{\partial J}{\partial y}$. The gradients of the loss with respect to the weight and bias parameters are then computed using the chain rule as follows [81]:

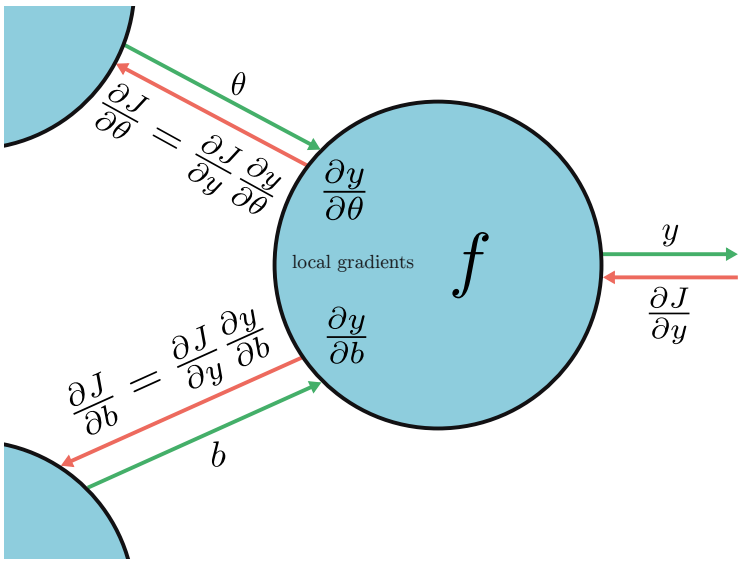



Figure 3.6 Neuron with forward pass variables and backward pass gradients.

$$\frac{\partial J}{\partial \theta} = \frac{\partial J}{\partial y}\frac{\partial y}{\partial \theta} \tag{3.10}$$

$$\frac{\partial J}{\partial b} = \frac{\partial J}{\partial y}\frac{\partial y}{\partial b} \tag{3.11}$$

By computing the partial derivatives of the loss with respect to each parameter, backpropagation determines how much each parameter contributed to the prediction error. This process is applied throughout the entire model, starting from the output layer and working backwards. Using the chain rule, all parameters in the model can be updated efficiently from a single forward and backward pass.

### Straight-Through Estimator (STE)

This thesis focuses on the implementation of Quantised Neural Networks (QNNs). As a preliminary, it is important to understand the effects of quantisation during training.

QNNs are the generalisation for when fixed-point arithmetic is used to represent the weights and activations of a neural network. QNNs with lower bit-widths than floating-point equivalent models tend to use less energy to perform multiply-accumulate functions [82]–[85], making QNNs an attractive candidate for accelerating neural network models by implementing weights and activations with binary, ternary, and arbitrary wordlengths.

Quantisation is applied in the forward pass during training. For a weight in a QNN that is quantised to Q bits in a fixed-point representation, the following deterministic quantisation function applies [78]:

$$q = \text{clip}(\frac{\text{round}(2^{Q-1} \times \theta)}{2^{Q-1}}, -1, 1 - 2^{-(Q-1)}) \tag{3.12}$$

where the clip function limits the resulting value of the quantisation function to a maximum and minimum value of $-1$ and $1 - 2^{-(Q-1)}$, respectively. However, this function is non-differentiable and breaks the backpropagation procedure. To successfully propagate gradients through discrete neurons, a Straight-Through Estimator (STE) function is used for the backpropagation. The STE pretends the quantisation operation is the identify function during backpropagation. In the forward pass, quantisation is applied, but in the backward pass, the quantisation step is skipped and the gradients are passed straight-through as if no quantisation occurred [86]. If the estimator $g_q$ of the gradient $\frac{\partial J}{\partial q}$ arrives from next layer, the gradient of $\frac{\partial q}{\partial \theta}$ is solved by the STE with:

$$\text{STE} = \text{clip}(\theta, -1, 1) \tag{3.13}$$

and the propagated STE of $\frac{\partial C}{\partial \theta}$ is

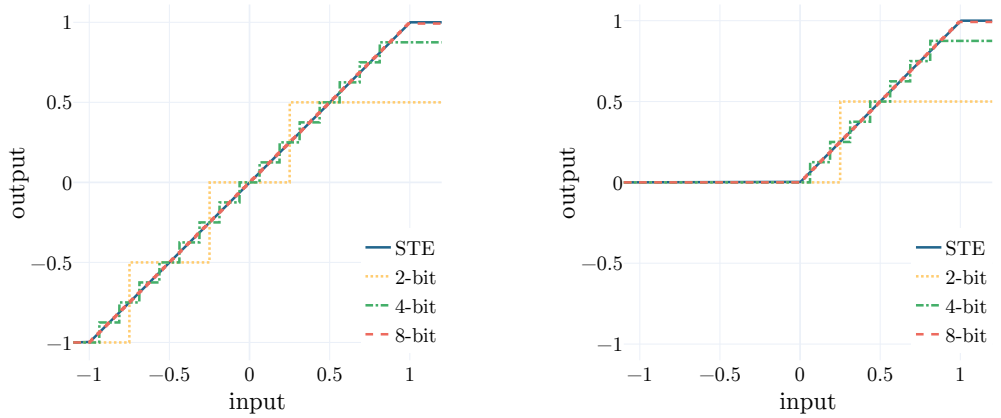
$$g_\theta = g_q \times \text{clip}(\theta, -1, 1) \tag{3.14}$$

If the fixed-point representation requires a larger number than -1 and 1 the clipping can be adjusted. The same STE techniques can also be applied to the activation quantisations. Figure 3.7 plots the STE against different weight and activation quantisation bit-widths.

### Optimisation Algorithms

To train a neural network, the loss $J(\cdot)$ computed from a forward pass must be minimised by updating the network's parameters. Optimisation algorithms assist with achieving a minimised loss by using the gradients during backpropagation and guiding their adjustment. In this thesis, the Adam optimiser is used for training all neural network modules.

Adaptive Moment Estimation (Adam) is a very popular optimiser that combines the benefits of two other techniques: momentum (which smoothens updates based on the past gradients) and adaptive learning rates (which scales updates based on the magnitude of recent gradients). This makes Adam effective for training neural networks and assists in quick convergence [87]. This




(a) Weight quantisation

(b) Quantised ReLU

Figure 3.7 Quantised weights and activations with the associated gradient estimation using the STE.

combination also reduces the need to tune hyperparameters, such as learning rate, momentum coefficients, and weight decay, which are parameters set before training and not learned from data. This is particularly useful when working with reduced precision models, where additional parameters like quantisation settings increase the number of hyperparameters to tune.

### Regularisation

Neural networks are excellent at representing the complex relationships between an input feature map and its associated output for a given task. This ability has enabled neural networks to outperform many traditional ML algorithms assigned similar tasks. When a model performs well on training data but poorly on new unseen inputs, the model has undergone 'overfitting'. When a neural network overfits on the training dataset, it learns an overly complex representation that models the distribution of the training data too well. As a result, it performs very well on the training dataset but generalises the overall task poorly, with sub-optimal results when given unseen data.

Regularisation is the term given to techniques that assist neural networks to generalise better to unseen data encountered after the training process, and reduce overfitting. Regularisation techniques minimise complexity and expose the network to a more generalised and/or diverse set of data. The regularisation techniques covered in this section are: early stopping, $L_p$ regularisation, and

the addition of noise. Many other techniques for regularisation exist that are not covered in this thesis [25].

**Early Stopping** is one of the simplest forms of regularisation techniques to help prevent overfitting. It involves monitoring the model's performance on unseen data throughout the training process, and stopping the training at an earlier epoch if the model stops improving when shown unseen data [88].

Early stopping is applied during the training process and requires that a portion of the training dataset is reserved as unseen data, known as the validation set. Unlike the training set, the validation set is not used to update the model's weights. Instead, it acts as a benchmark – similar to the test set – for measuring how well the model generalises unseen data during training. To support effective training and evaluation, a typical dataset is divided into three subsets: the training set, the validation set, and the test set – commonly split at a ratio of 80:10:10, respectively.

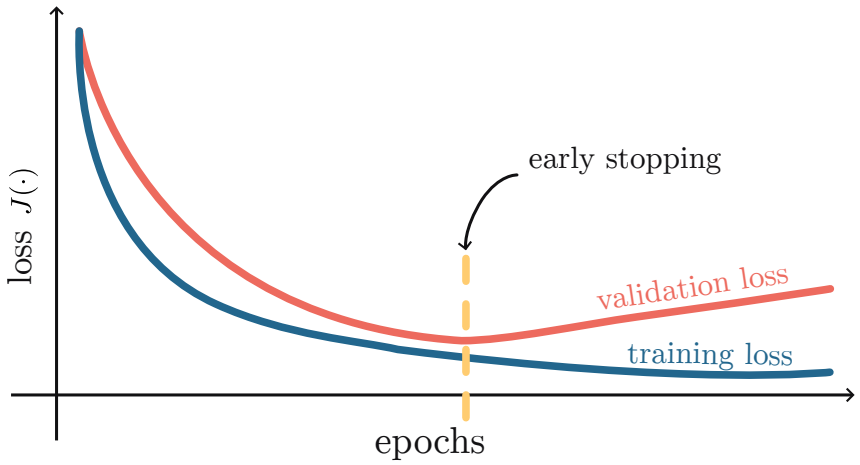



Figure 3.8 Example plot showing training and validation loss with early stopping.

Figure 3.8 illustrates an example plot for training and validation loss over time (epochs) during a training session. Early stopping is triggered when the validation loss stops improving, indicating that the model may begin to overfit if training continues. By halting training at this point, the model that performed best on unseen data throughout the training process is preserved.

**$L_p$ Regularisation** is a general term used for penalising the model's loss function based on the magnitude of its weights. They force the norm of the weight vector to stay sufficiently small, which encourages the model to learn more generalised solutions [25], [89]. The $p$ refers to the norm used, where

$p = 1$ and $p = 2$ are the two most common norms. The $L_p$ norm of a vector $x$ is given by:

$$L_p(x) = ||x||_p = (\sum_{i=0}^{N-1} |x_i|^p)^{1/p} \tag{3.15}$$

The $L_p$ regularisation is applied to the loss function for a set of model parameters by:

$$\tilde{J}(\theta) = J(\theta) + \alpha||\theta||_p \tag{3.16}$$

where $\alpha$ is the regularisation constant. When the parameters of the network become too large during training, $\alpha||\theta||_p$ increases and, since the goal of backpropagation is to minimise $\tilde{J}(\theta)$, the weights should decrease as a result of the increased overall loss.

**Addition of Noise** is another approach that is simple but a very effective form of regularisation. The addition of noise for the purposes of regularisation can be introduced to the input data, the output labels, the gradients of the backpropagation, or the weight values themselves [25]. In this thesis, noise is added primarily at the input data for the configurations considered as AWGN, to simulate noisy environments. This additive noise also assists with regularising the model to better generalise the signals it is classifying [90].

Additionally, the QAT technique is explored in Chapter 6 which applies fixed-point quantisation to the model weights prior to training. Quantisation of the weights inherently adds noise called 'quantisation noise', which arises from the use of low-precision values to represent the weights. Quantisation essentially discretises the range of values a parameter can take, which can be viewed as injecting a form of structured noise into the network weights.

As shown in Figure 3.9, an $N_b$-bit quantiser splits the input range into $2^{N_b}$ discrete levels, with each level separated by a fixed step size $\Delta$:

$$\Delta = \frac{2}{2^{N_b}} \tag{3.17}$$

Assuming a uniform quantiser the mean squared quantisation noise power is given by [60]:

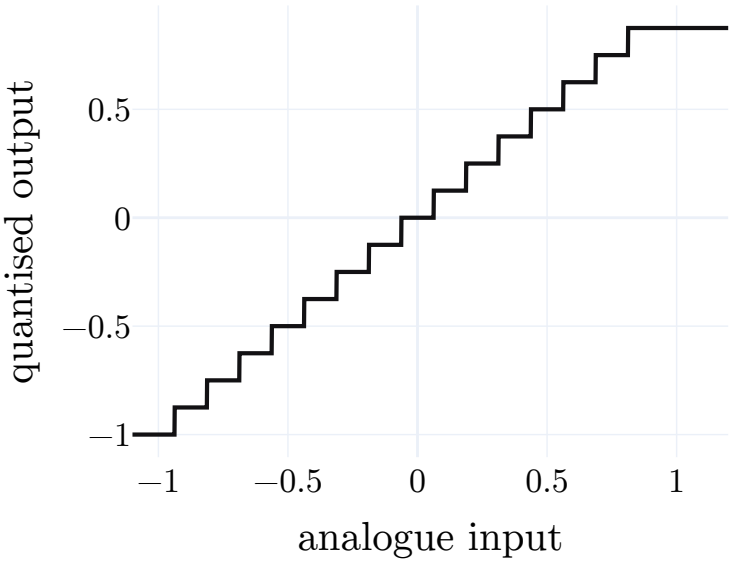



Figure 3.9 Quantisation of an analogue input.

$$\begin{aligned} Q_{N_b} &= 10\log(\frac{(2/2^{N_b})^2}{12}) \\ &= 10\log(2^{-2N_b}) + 10\log(\frac{4}{12}) \\ &= -6.02N_b - 4.77 \text{ dB} \end{aligned} \tag{3.18}$$

This shows that the quantisation noise power decreases with the number of bits. While quantisation introduces error, when properly integrated into training (as in QAT), it can act similarly to other regularisation techniques by improving the robustness and generalisation of the network under reduced precision constraints.

## 3.3 Accelerating Neural Networks

As DNNs become larger and more capable, the need for efficient, fast, and energy-conscious execution becomes increasingly important, particularly for real-time embedded applications. While high-end GPUs are widely used during the training phase, deploying neural networks in the real world often means operating on hardware with limited resources. Devices such as mobile phones,

IoT sensors, radios, and FPGA-enabled devices face constraints in power, memory, and compute power, requiring optimised accelerator architectures to successfully operate AI models. This thesis focuses specifically on accelerating neural networks for inference tasks, where speed and resource efficiency are key.

Neural network acceleration spans a range of platforms, from cloud servers with an abundance of compute power and memory, to mobile devices and edge systems where energy consumption and low latency are of utmost importance. Each platform brings its own set of trade-offs in terms of throughput, memory constraints, and power efficiency.

AI acceleration at the edge in wireless communications, particularly at the point where the radio operates and collects samples, offers substantial advantages for deploying AI models. As discussed in Chapter 2, PHY-layer AI applications can enhance signal decoding, improve channel corrections, and lead to better overall service. To make these benefits a reality, it is essential to establish a platform or architecture that not only supports these AI-driven functionalities, but also integrates well with the existing radio systems and their dataflow based signal processing approaches. This ensures that AI deployments complement and enhance the current workflows rather than disrupt them.

The remainder of this section explores the various hardware types used for AI acceleration, the typical deep learning workflow for embedded systems, the GeMM transform, and the role of quantisation in reducing computational load and memory usage.

### 3.3.1 Hardware Types for AI Acceleration

In the field of AI acceleration, various hardware platforms offer different advantages depending on the use case. The main compute platforms for AI include Central Processing Units (CPUs), GPUs, FPGAs, and Application-Specific Integrated Circuits (ASICs).

- **CPUs** are flexible and commonly used, but tend to offer lower parallel processing capabilities for computationally intensive AI tasks compared to GPUs. Recent advancements in CPU instruction sets better enables AI workloads on CPUs [91], however, they are still limited in terms of supported data types and they are power hungry.

- **GPUs** excel in parallel computation and are the preferred choice for training large neural networks due to their high throughput and parallel processing capabilities. However, while GPUs are highly efficient for

training, using them for inference of already trained models can be energy intensive. This increased energy consumption makes GPUs less ideal for continuous, real-time inference tasks, as they tend to be costly to maintain and operate over long periods of time.

- **FPGAs** provide customisability and energy efficiency, and offer a middle ground between flexibility and parallel computational capabilities. While FPGAs have great potential for successfully accelerating AI tasks while keeping energy consumption down, developing designs for FPGAs requires significant expertise.

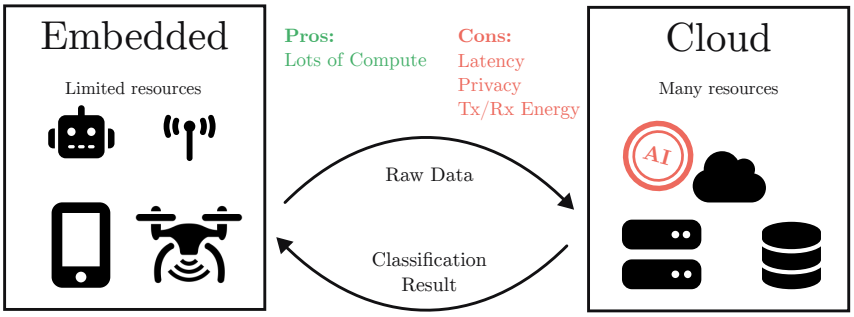
- **ASICs** are highly specialised chips designed for specific tasks, offering the best performance and energy efficiency for AI inference applications. However, once a design has fabricated as for an ASIC, it is permanent, making them a risky choice in the ever growing and fast changing field of AI acceleration.

DL techniques have become very popular for more and more tasks such as image recognition, audio processing, and wireless signal processing, for applications including autonomous vehicles, drone communications, cognitive radio, and robotics. Though excellent candidates for AI, these use cases require low latencies and fast throughputs from their associated AI models. For instance, in the case of autonomous vehicles, the speed at which an image can be processed through an AI model to identify the location of other cars or people is vital to the effectiveness and safety of the application.

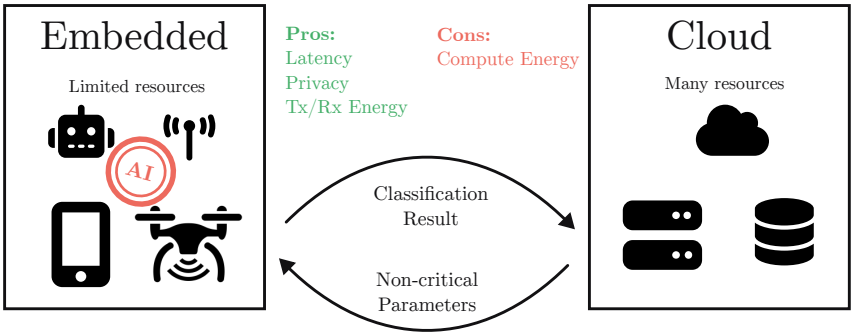
GPUs are fast at running AI models, but their high power consumption and hardware demands make them difficult to use directly at the edge, especially in power-constrained environments like vehicles, radios, or drones. A common workaround is to offload the AI processing to a remote server, or the cloud. In this setup, the edge device captures data, sends it to the remote GPU for processing, and then receives the results back. This approach comes with several drawbacks, as depicted in Figure 3.10a: transmitting and receiving large amounts of data can drain power on the edge device, the overall latency for the task increases due to the time taken to send and receive data, and there is a risk to data privacy. Encrypting the raw samples to protect privacy also adds to the device's power burden.

Alternatively, running an AI model directly on the edge addresses many of these challenges. Since the data no longer needs to be transmitted off-site, power consumption from the communication is reduced, latency is significantly

improved (as it operated upon on the device), and privacy is inherently better because the raw data stays local. With these major issues mitigated, the focus shifts to the remaining challenge of: how to run the model efficiently on resource-limited hardware. This requires minimising the energy used for computation, and optimising the model and hardware to fit within the constrained computing resources of the embedded system, as depicted in 3.10b.




(a) AI in the cloud.




(b) AI at the edge

Figure 3.10 Comparison of the pros and cons of computing AI in the cloud vs at the edge.

### 3.3.2 Embedded Deep Learning Flow

The process of deploying DL models on embedded devices typically begins with training the model on a GPU using high-level AI training frameworks. Here the model learns a classification task, as depicted in Figure 3.11, by receiving many data examples and predicting an output. Through optimising a loss function, the weights are updated in the model until a desirable accuracy and performance is met. This process typically happens on a local PC or on a GPU server.

To deploy the trained model on an edge device, the weights are extracted from the neural network in the training phase and transferred to the neural network for the inference phase. Here the deployment undergoes a series of

conversions and optimisation steps, including adjusting the model's precision, and converting the format of the weights to one that is compatible with the target edge platform. Finally, the edge device operates the embedded neural network model to classify real-life data.

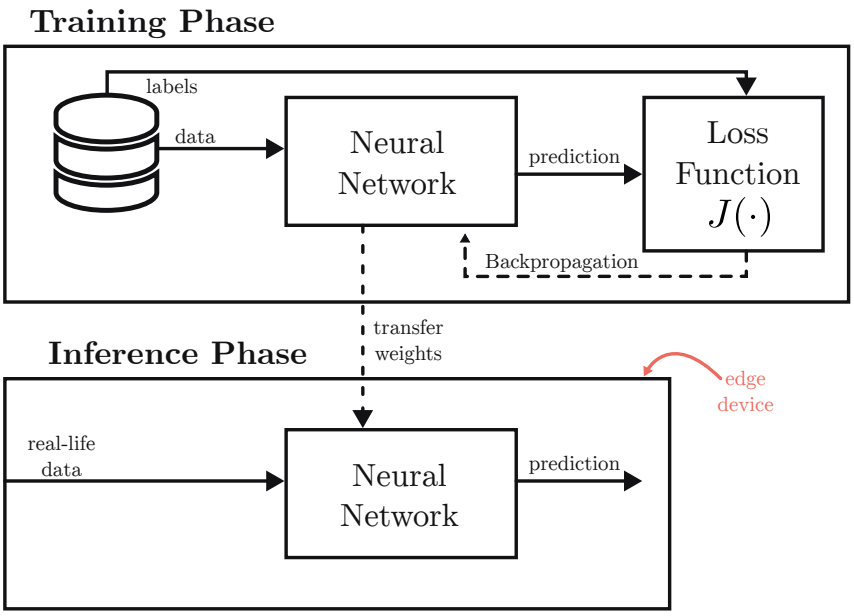



Figure 3.11 Design flow of embedded AI inference.

### 3.3.3 Matrix Multiplication in Neural Networks

In Sections 3.2.3 and 3.2.4 a theoretical overview of the FC and convolutional layers and their uses was covered. This section aims to detail the practical considerations of accelerating the FC and convolutional layers in the context of an AI accelerator. As stated in the previous section, the goal of optimisation for operating AI models on the edge is to minimise energy consumption and to optimise the computational capabilities of the model on the target device, so that it can operate in real-time and with minimal energy consumption.

#### Fully-Connected Layers

The equation for a FC layer was given by Equation 3.3 which highlighted that, when calculating the output of a FC layer the operation can be performed as a General Matrix Multiplication (GeMM), since every input is connected to every output neuron via a weighted connection..

GeMM is a fundamental operation in linear algebra and computational mathematics [92]. Matrix multiplication is a widely used mathematical opera-

tion that can be optimised by leveraging parallelism in computational hardware such as GPUs, CPUs, and FPGAs. Software libraries such as Basic Linear Algebra Subprograms (BLAS) and CUDA BLAS (cuBLAS) assist compilers in determining how to best parallelise matrix multiplication operation on selected hardware.

The nest loop for a FC layer is given in the Python code Listing 3.2. Here, each input is used $N$ times before an output for the FC layer is complete.

Listing 3.2 Nested For loops for FC layer.

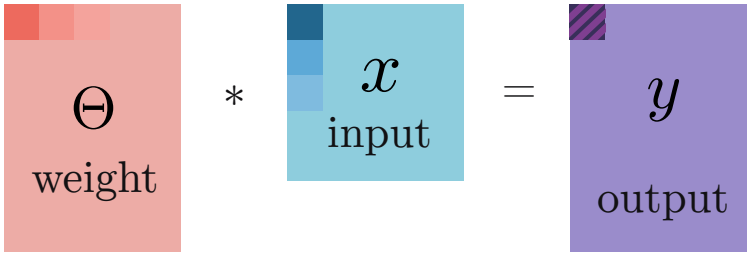
```
for n in range(N): # Output channels
    for m in range(M): # Input channels
        y[n] += Theta[m,n] * x[m]
```

Listing 3.3 shows an example of parallelising this process where the **parfor** operator represents a fully parallel computation where all elements of the loop are calculated at once.

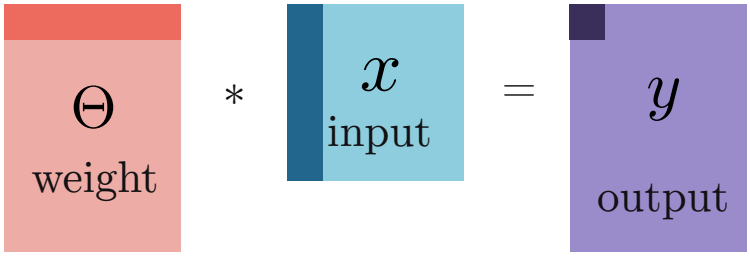
Listing 3.3 Nested For loops for FC layer with parallel process.

```
parfor n in range(N): # Output channels
    for m in range(M): # Input channels
        y[n] += Theta[m,n] * x[m]
```

Figure 3.12 illustrates the effects of parallelising a portion of the matrix multiplication operation. In Figure 3.12a, for each output sample for $y$, a summation of products between $\Theta$ and $x$ is performed over multiple clock cycles. In Figure 3.12b, parallelising this process results in an output sample of $y$ in one clock cycle by processing multiple samples from $\Theta$ and $x$ in parallel.




(a) Non-parallel matrix multiplication.




(b) Parallelised matrix multiplication.

Figure 3.12 Comparison of non-parallel and parallel matrix multiplication.

### Convolutional Layers

Convolutional layers work by applying a set of filters (also called kernels) to the input data to generate output feature maps. This involves sliding each filter across the input, performing element-wise multiplications and summations at each position (as illustrated in the nested loops in Listing 3.1). When dealing with large inputs, this process becomes computationally intensive and complex.

To speed up convolution operations, both the input data and filter kernels can be transformed into matrices, allowing the use of GeMM to perform the computation. Converting convolutions into a standard matrix multiplication brings several advantages for AI accelerators. It unifies the workload across convolutional and FC layers (where both become matrix multiplications), simplifies scheduling since the operation is consistent, and streamlines hardware design by focusing optimisation on a single core operation, i.e. matrix multiplication. Applying the GeMM transform to a convolutional layer comes at the cost of repeating input samples.

As shown in the FC layers, the GeMM transform makes it possible to leverage parallelism in FPGA hardware to accelerate the matrix multiplications for efficient computation. Hardware like GPUs use the GeMM transform to simplify the computational complexity of the convolutional layers. Optimising this is crucial for speeding up the training and inference process in DL models.

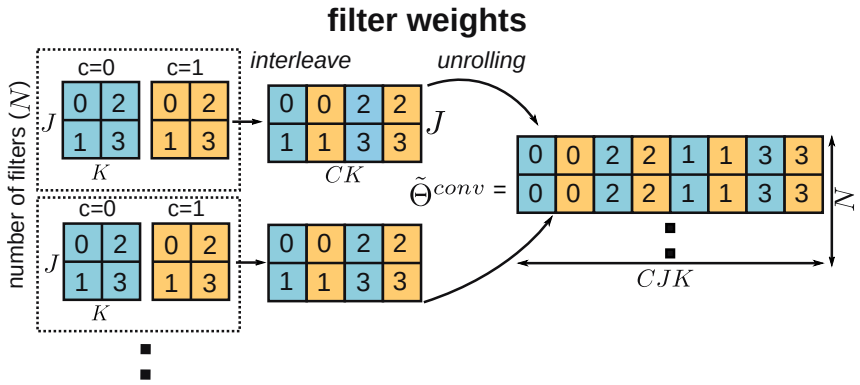
The GeMM transform for convolutional layers involves converting the input and layer kernels into their respective matrices. This is performed by applying the image to column (im2col) method [93]. The input and each set of layer kernels are converted in their respective matrices. This process is depicted in Figure 3.13b.

As discussed in Section 3.2.4, for a 3D convolution, the input feature map has 3-dimensions: channels $C$, width $W$, and height $H$. The filter kernels are a 4D tensor with number of kernels $N$, channel $C$, kernel height $J$, and kernel width $K$. For the filter kernel weights, the GeMM transform is applied by interleaving the filter channels $C$ for each filter $N$. Each interleaved filter is then unrolled and concatenated forming a single 2D matrix denoted as $\tilde{\Theta}^{\text{conv}}$ with dimensions $N \times CJK$. Effectively, the filter kernel weights are flattened from a 4D tensor into a 2D matrix, where the same number of total elements are used. To transform the input feature map, the following steps are required:

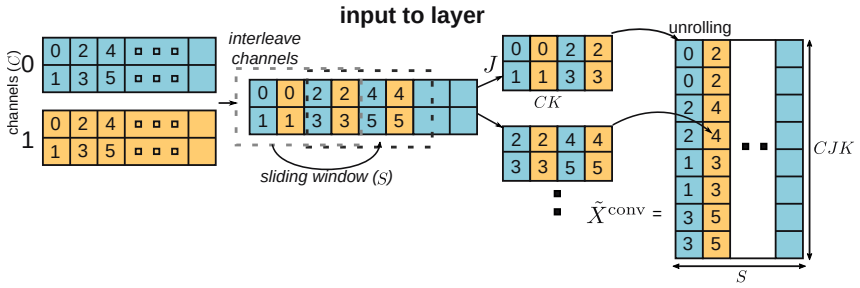
- Re-shaping the 3D input into a 2D matrix where the channels $C$ of the input are interleaved.

- Striding and replication functions are applied (see Figure 3.13b).
- Unrolling. For each stride step, the input samples under the filter window are unrolled and concatenated, resulting in the matrix $\tilde{\boldsymbol{X}}^{\text{conv}}$

Figure 3.13 illustrates the process of performing the GeMM transform for both the input feature map and the filter kernels for the convolutional layer.




(a) Filter kernel weights transform.




(b) Input transform

Figure 3.13 GeMM transform operations to 2D matrices of input and filter kernels.

When transforming the input, the columns $K$ of each input channel $C$ are interleaved to form a 2D matrix. The flattened sliding window is passed over the 2D input and split into smaller matrices based on how many strides $S$ are performed by the sliding window. Each of these split matrices are then unrolled and concatenated to form the resulting GeMM transformed input matrix $\tilde{\boldsymbol{X}}^{\text{conv}}$, with dimensions $CJK \times S$.

When the GeMM transform is applied to the input, the number of total samples in the feature map increases as a result of the reduction in complexity. This is because the sliding window operation is performed prior to the matrix multiplication calculation. Equation 3.19 shows how the convolution is calculated after applying the GeMM transform to the input $\tilde{\boldsymbol{X}}^{\text{conv}}$ and resulting

transformed filter kernel $\tilde{\Theta}^{\text{conv}}_{N\times CJK}$. The bias $b^{conv}_{N\times 1}$ is also added to the matrix multiplication output.

$$\tilde{Y}^{\text{conv}}_{N\times S} = \tilde{\Theta}^{\text{conv}}_{N\times CJK}\tilde{X}^{\text{conv}}_{CJK\times S} + b^{conv}_{N\times 1} \tag{3.19}$$

Equation 3.19 produces the convolutional layer output $\tilde{Y}^{\text{conv}}_{N\times S}$ with dimensions $N \times S$.

Simplifying the neural network layers down to matrix multiplications allows for the calculations to be accelerated more effectively due to increased scope for parallel operation. Additionally, the calculation for both FC and convolutional layers is a matrix multiplication, so the hardware required for both layers types can be shared.

## 3.4 Optimising Neural Network Computations

Many modern AI accelerators are built around the Neural Processing Unit (NPU) architecture [11]–[13], which is designed to execute the core operations of neural networks, such as the GeMM transformed convolutional and FC layers. NPUs typically rely on highly parallel processing units, memory hierarchies for tensor data, and low-precision arithmetic to reduce MAC operation power consumption and increase throughput.

While this thesis ultimately focuses on a custom AI accelerator based on a dataflow execution model, the NPU serves as a useful reference point for understanding how neural network computations are typically optimised in hardware. The concepts discussed in this section can be directly applied to AI acceleration architectures on programmable hardware devices, like FPGAs. This section builds on that foundation, introducing the core ideas and techniques used in AI hardware design such as parallelism, stationarity, and the introduction of dataflow models. The optimisation techniques for efficient AI task execution are shared between typical NPU structures and more custom architectures such as dataflow accelerator models.

### 3.4.1 A Typical Neural Processing Unit

An NPU is a specialised processor designed specifically to execute neural network computations efficiently. As the limitations of general purpose processors like CPUs and GPUs become more apparent in AI workloads, engineers have increasingly turned to custom hardware accelerators such as NPUs. These

architectures are designed to exploit the known dataflows of neural network calculations, enabling parallel execution of computations while minimising data movement between off-chip and on-chip memory, which can be costly in both energy and latency.

Figure 3.14 shows a simplified NPU architecture. It consists of on-chip memory, which stores the weights, inputs, and outputs of each layer, and a systolic array of Processing Elements (PEs). Each PE contains a Multiply-Accumulate (MAC) unit and local registers for temporary data storage. The systolic array performs matrix multiplication across layers, where multiple PEs operate in parallel to accelerate computation.

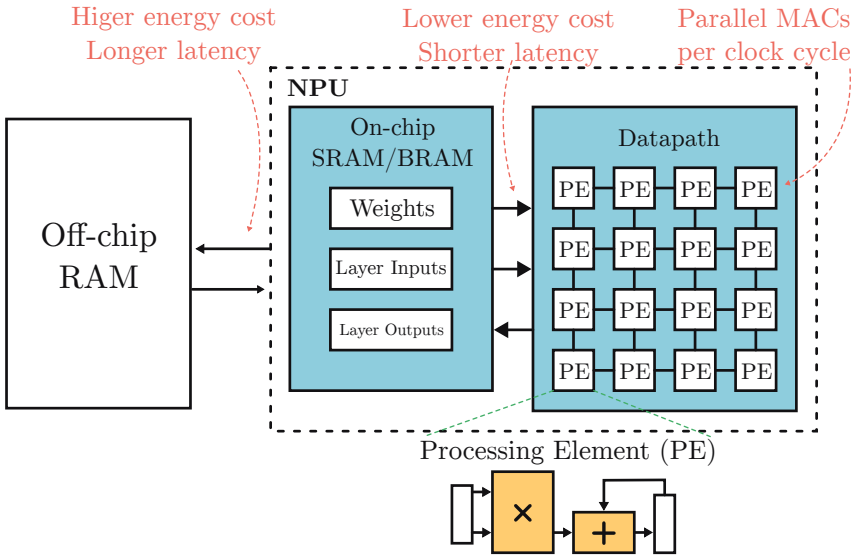



Figure 3.14 Basic structure of a typical NPU.

As shown in Horowitz's energy analysis [94], accessing off-chip memory (e.g. shared DDR memory) consumes significantly more energy than on-chip alternatives such as SRAM and BRAM. It also introduces higher latency, contributing to performance bottlenecks. For this reason, modern NPUs aim to reduce external memory traffic and keep data as close to the compute units as possible.

The regular structure of a systolic array enhances the performance of the accelerator by enabling multiple MAC operations per memory fetch. This not only improves data reuse, but also boosts throughput and energy efficiency, since multiple calculations are performed in each clock cycle.

This basic architecture is the foundation for many modern AI chips, including Google's TPU series (v1-v4) [11], [95], MIT's Eyeriss [12], and AMD's XDNA NPU in Ryzen AI processors [13]. Each of these AI accelerator ar-

chitectures contain hundreds of PEs that can perform many computations in parallel.

The following sections introduce core techniques for optimising neural network computations, focusing on how they apply to custom hardware architectures.

### 3.4.2 Parallelism and Stationarity

Facilitating data movement to and from an AI accelerator with hundreds of parallel PEs becomes a significant challenge unless spatial and temporal unrolling techniques are employed. Neural networks layers involving matrix multiplications regularly reuse their data (computations share the same inputs or weights). For example, in a GeMM operation $\Theta_{N,N} \times X_{M,M}$, each weight in $\Theta$ is used $M$ times to compute the output $Y$. Efficient accelerators take advantage of this reuse to minimise redundant data movement.

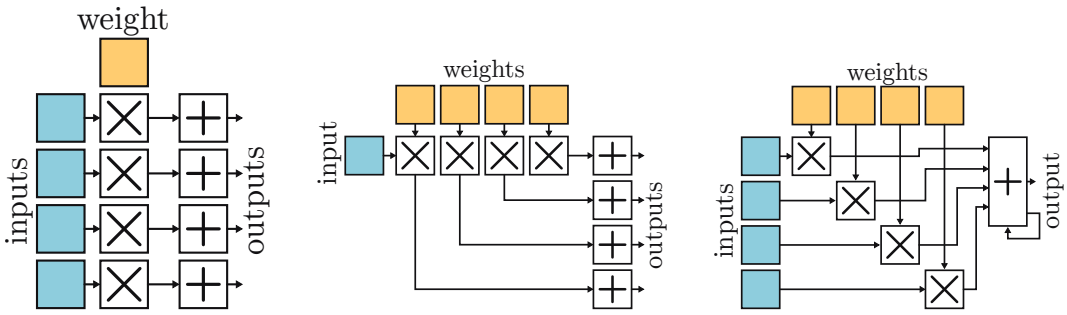
This is where the concepts of parallelism and stationarity come in. These terms describe how data reuse can be exploited:

- **Parallelism** refers to performing calculations in parallel across multiple PEs in the same clock cycle (spatial unrolling).
- **Stationarity** refers to reusing data across multiple cycles on the same PE (temporal unrolling).

In practice, parallelism is achieved by distributing computations across a PE array, allowing many operations to be performed simultaneously. Stationarity keeps data local to the PE for as long as possible to avoid costly memory fetches. By leveraging both, accelerators can significantly reduce memory bandwidth demands and improve throughput, latency, and energy efficiency.

Different strategies for unrolling or folding the computation lead to weight reuse, input reuse, or output reuse, each offering unique trade-offs depending on the application and hardware constraints.

Figure 3.15 and Table 3.1 illustrate three common data reuse strategies: weight stationary, input stationary, and output stationary. Each aim to reduce memory bandwidth (BW) requirements by minimising data movement while maximising parallel computation. Each strategy reuses data samples across clock cycles and exploits spatial parallelism using multiple PEs to increase the throughput [78], [96].




(a) Weight stationary (input parallel).

(b) Input stationary (weight parallel).

(c) Output stationary.

Figure 3.15 Spatial and Temporal Unrolling.

Table 3.1 Spatial and Temporal Unrolling.

| | **Weight stationary** (Fig. 3.15a) | **Input stationary** (Fig. 3.15b) | **Output stationary** (Fig. 3.15c) |
|---|---|---|---|
| **Weight** Memory BW | low | high | high |
| **Input** Memory BW | high | low | high |
| **Output** Memory BW | high | high | low |

- Weight stationary (input parallel), shown in Figure 3.15a, holds weights locally within the PEs while inputs stream through. This approach minimises the number of weight fetches (ideally loading each weight only once), but increases the demand on input bandwidth since new inputs are required with each operation. Additionally, because outputs are not accumulated across cycles, intermediate results are written to and read from memory, which raises output bandwidth requirements.

- Input stationary (weight parallel), shown in Figure 3.15b, does the opposite. Input data is held constant while different weights are streamed in across PEs. This significantly reduces input memory BW at the cost of higher weight BW, as weights must be frequently reloaded. As with the previous strategy, output accumulation occurs externally by sending partially accumulated values to store in memory, increasing the output BW.

- Output stationary, shown in Figure 3.15c, keeps the partial sums of the output activations locally within each PE. Inputs and weights are

reloaded each cycle, leading to higher input and weight BW, but outputs are accumulated in-place, significantly reducing output memory traffic.

These unrolling strategies are fundamental to optimising memory access patterns in AI accelerators. By reusing fetched inputs, weights, and/or outputs over multiple clock cycles, the overall memory traffic is reduced, leading to improvements in latency. Meanwhile, mapping multiple MAC operations across PEs increases the throughput, allowing many computations to be executed simultaneously.

In practice, AI accelerator designs typically implement hybrid schemes that combine elements from all three strategies to balance performance and resource constraints. For example, architectures in [97], [98] use weight stationary approaches to minimise redundant weight fetches, while [99], [100] adopt input stationary schemes that cache input data for reuse. In all of the mentioned implementations, output stationary techniques are also employed to accumulate partial sums locally, reducing memory stores and improving the overall efficiency.

### 3.4.3 Dataflow Models

In contrast to traditional AI accelerators like NPUs (as discussed in Section 3.4.1), which use a shared systolic array of PEs to time-multiplex computations across layers, SDF architectures take a fundamentally different approach. Neural network structures are typically viewed as a sequential chain of layers, each performing its computation before passing the results onto the next. In a SDF model, this abstract graph of layers is mapped directly into hardware, where each layer has its own dedicated computational logic on the FPGA fabric.

Instead of time-sharing a common systolic array of fixed PEs, SDF accelerators generate custom hardware for each layer of the model. These blocks are connected together in a streaming pipeline, allowing each layer to start processing and forwarding its outputs to the next as soon as data becomes available. This structure reflects the network's topology in physical hardware: each layer has its own compute logic where multiple layers can compute concurrently. Figure 3.16 illustrates a basic SDF for a four-layer neural network model.

Since the architecture preserves the model's topology and enables layers to pass data directly to one another, hardware pipelining can be applied between layers to increase throughput. Since there are limited memory fetches and

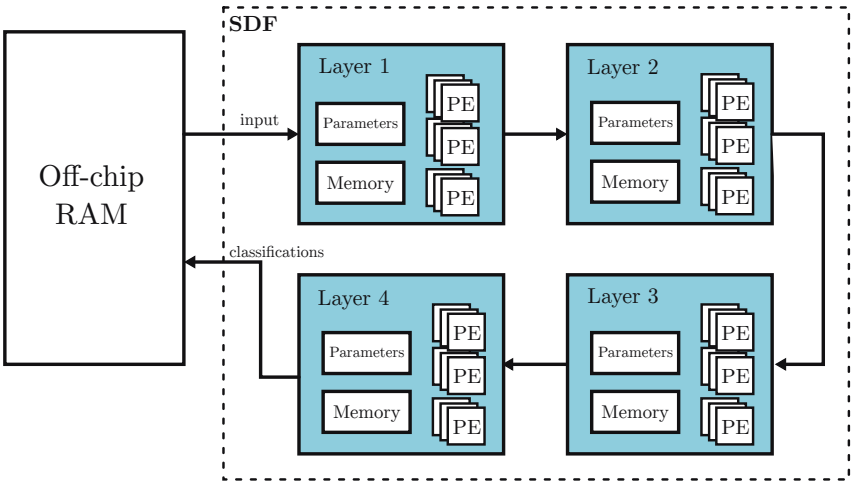



Figure 3.16 Basic structure of a SDF model for a custom neural network of four layers.

stores, the architecture's throughput bottleneck lies in the speed at which calculations can be processed.

Frameworks like FINN and fpgaConvNet are built around this dataflow concept, synthesising accelerators specifically tailored to a given neural network model [39], [40]. Since logic is not reused across layers, there is no need for centralised control or scheduling, reducing overhead and making this architecture especially efficient for compact, low-latency models deployed on edge FPGAs.

This approach plays to the strengths of FPGAs, which can implement highly parallelised customised hardware to complement the topology of the model and match desired throughput target. The core disadvantage to the SDF architecture is that each neural network model requires a specific hardware architecture as opposed to the NPU architectures that can process a range of model topologies. FPGAs are well suited for SDFs because of their ability to reprogram their PL fabric, allowing the hardware to be changed for each new model topology.

## 3.5 Chapter Conclusion

This chapter has covered the basic building blocks of neural networks. Fully-connected and convolutional layers were introduced, and DL techniques for training neural networks were covered, including training for quantised weights and activations. Training techniques for regularisation were also reviewed. The chapter then focused on summarising the core concepts behind accelerating neural networks on custom hardware AI accelerators and how AI inference can

be run on edge, low-resource devices. The concept of the GeMM transform was covered, as well as mapping of matrix multiplications to a typical NPU architecture, and the design considerations behind accelerating AI computations. Finally, dataflow models are introduced, which is the basis of the architecture proposed in this thesis. The concepts introduced in this chapter will appear in the following chapters when AI accelerator design choices are discussed.

This chapter has established the foundational concepts behind DL and the optimisations for accelerating DL inference models. It has also introduced an understanding of the core building blocks of a neural network and an algorithmic appreciation of the FC and convolutional layers, alongside how neural networks are trained. These concepts form the basis for understanding how algorithmic structures translate into hardware workloads and influence accelerator design. The material covered provides the groundwork for analysing trade-offs between computational efficiency, precision, and throughput in edge-based AI systems.

The next chapter will present a custom streaming-based CNN architecture for the FPGA-based SDRs using the data-flow model introduced in this chapter.

# Chapter 4

# Streaming-based CNN Architecture for FPGA Radio Receivers

This chapter will introduce a new custom streaming-based CNN dataflow architecture for FPGA-based radio receivers. The architecture is built using basic DSP building blocks with the purpose of achieving filter-like behaviour, so that it can be integrated with other DSP dataflow functionality. The architecture is evaluated with the RadioML dataset [45] on an embedded device while it operates in real-time.

Alongside the streaming CNN architecture, a methodology for building DL models from basic principles within MATLAB and Simulink is presented, as well as software drivers for interfacing with the deployed model once it has been transferred to the PL of the RFSoC.

## 4.1 Motivation

While modern NPUs and TPUs offer high-throughput and power-efficient inference for DL models, they are typically designed to be model topology-agnostic and target a general array of applications. This generally allows them to support a wide range of model topologies but introduces inefficiencies, particularly in edge environments where latency and bandwidth are critical constraints. One of the main limitations in these architectures is the memory bottleneck associated with transferring data between off-chip memory and compute cores, as discussed in Chapter 3.

In contrast, dataflow-oriented designs based on SDF models generate hardware that is tailored to a specific model structure. This approach removes the need for general purpose scheduling where the latency of a deployed model from input to classification is not immediately apparent. Instead, the SDF model allows for fully pipelined execution and optimised memory handling. Since the dataflow is statically defined, these accelerators can operate with deterministic latency and sustained throughput, making them well suited for real-time signal processing tasks.

The proposed streaming-based CNN dataflow architecture intends to be usable in-line with other signal processing and radio receiver pipeline tasks, such as filtering, channel estimation, and demodulation. Figure 4.1 illustrates this concept. The ultimate goal is to replace the existing traditional functions with DL models, without disrupting the dataflow structure of the receiver design.

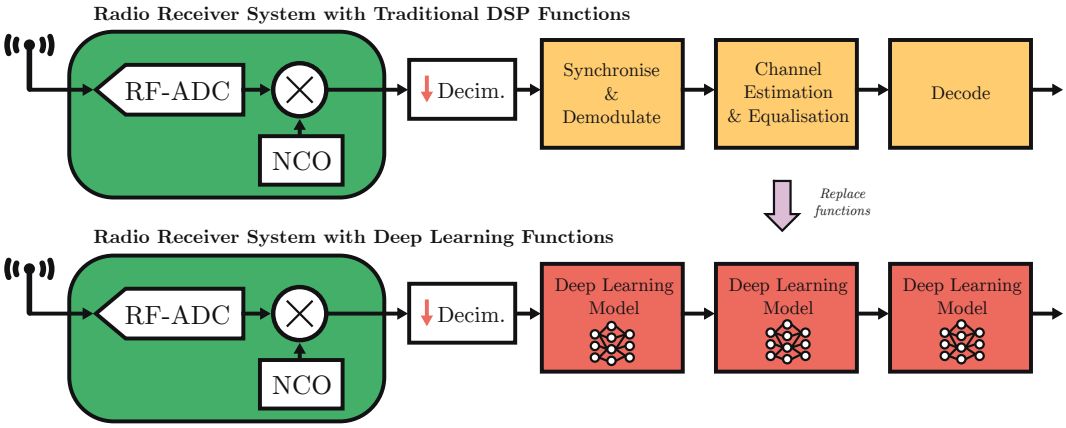



Figure 4.1 Motivation for replacing traditional radio pipeline functionality with deployed DL models.

## 4.2 Related Work

FPGA-based CNN accelerators have gained popularity for edge inference tasks due to their low-latency performance, energy efficiency, and the ability to customise parallelism for dataflow-style implementations. Unlike GPU-based or fixed-architecture NPUs, FPGAs offer fine-grained control over memory, compute, and parallelism, which can be customised to the needs of a specific model and application domain.

FINN [39] is a well-known framework for building dataflow accelerators targeting low-precision and quantised neural networks, typically in the 1–8-bit

range. It compiles trained models into custom hardware pipelines optimised for a fixed topology, achieving high throughput and efficiency. While FINN is mainly applied to vision tasks, it supports a range of high-throughput applications. However, its use in real-time PHY wireless communications tasks has not yet been demonstrated in the literature.

fpgaConvNet [40] is another framework that maps CNNs to FPGAs using performance modelling and hardware optimisation techniques. Like FINN, it automates the design process, exploring trade-offs through tiling and loop unrolling strategies to balance latency and resource usage. It has been used successfully in domains like image recognition and, to a lesser extent, natural language processing, but has not yet been applied to real-time wireless signal processing tasks.

hls4ml [41] offers a Python-based toolchain for converting trained ML models into HLS-compatible code for FPGA deployment. Originally developed for use in particle physics, it emphasises low-latency inference and interpretability in resource-constrained environments. The framework provides tuning options such as MAC reuse and parallelism levels to meet specific throughput and resource targets. While hls4ml shows promise for wireless applications, there are currently no known implementations targeting real-time SDR-based communications.

All three of the mentioned dataflow architecture compilers produce their IPs in a HLS language format. This makes simulating the resulting architecture in a Simulink-based environment difficult, disconnecting the CNN accelerator IP from the rest of the radio communications pipeline, if built using MathWorks tools.

In contrast to these frameworks, this work presents a custom streaming-based CNN architecture designed specifically for high-throughput, real-time RF data classification. The architecture supports direct interfacing with RF front-ends and is optimised for inference at the RF-ADC line rate on SDR platforms. Unlike existing systems, this approach introduces optimisations tailored to streaming RF data, including a modified GeMM transform that rearranges data into a channels-first format to enable faster sample propagation between layers. The architecture is built using MATLAB and Simulink toolkits such as HDL Coder [101] and Fixed-Point Designer [102], meaning the resulting architecture can be simulated with other radio tasks built in MATLAB/Simulink.

## 4.3 Streaming Convolutional Neural Network Architecture Design

In this work, the decision to create a bespoke neural network architecture for FPGA hardware originated from the positioning of the neural network model in a PHY wireless communications receiver application. To effectively use a DL model in a radio receiver to perform tasks such as decoding, channel estimation and correction, and spectrum sensing, the model should receive samples coming from the RF-ADC stage and process the samples as they enter the chip. In many receiver cases, processing every sample that enters the chip is imperative to the functionality of the aforementioned tasks. It is a strict requirement that no samples are lost in the process of performing a DL task, as radio communications samples are highly correlated with one another and dropping samples could result in the inability to successfully decode a signal. The CNN architecture described in this chapter was designed with this requirement at the heart of the implementation. The resulting design processes every sample that enters the model and has a deterministic latency, allowing it to be synchronised with other functionality in the receiver.

The fundamental principle guiding the architectural design revolves around the dataflow design paradigm, where incoming data samples are treated as a continuous infinite stream of samples, each undergoing real-time processing. A notable challenge encountered in this framework comes from the overproduction of output samples by the convolutional layers, limiting the throughput of the deployed DL model and leading to a requirement to balance between the implementation throughput and associated FPGA clock rate for the model. Mitigating this challenge involves adjusting the overall clock rate of the system, contingent on the configuration of the convolutional layer parameters, such as kernel size and number of filters, and the resource allocation scheme of the MAC units within the implemented layer. This concept is further explored in Chapter 5.

### 4.3.1 Input Data Pre-processing

The input samples to the neural network architecture arrive as a continuous stream of complex IQ values. These samples are assumed to originate from a decimation filter chain, which reduces the sample rate before entering the architecture. While the exact source and sampling rate are implementation-

dependent, we assume that the decimated samples are uniformly spaced in time and arrive synchronously with the system clock.

To align with the model's frame-based processing, a buffering mechanism is introduced to group streaming samples into fixed-length bursts. This is achieved using a Ping-Pong buffer, where two alternating buffers operate in parallel: while one buffer fills with incoming samples, the other is read out as a complete frame for CNN processing. Once a full frame (e.g. 256 samples) is collected, it is passed to the first convolutional layer.

This burst-style input flow ensures that each frame is processed in sequence without data loss, and that the model has sufficient time to complete its inference on one frame before the next is ready. A simplified diagram of the Ping-Pong buffering mechanism is shown in Figure 4.2.

Prior to entering the Ping-Pong buffer, the samples are interleaved so that the parallel I/Q samples are received as one stream of I and Q interleaved samples.

The Ping-Pong buffer is implemented with on-chip BRAM and URAM and is managed by a counter system that counts the size of burst frames required to make a classification.

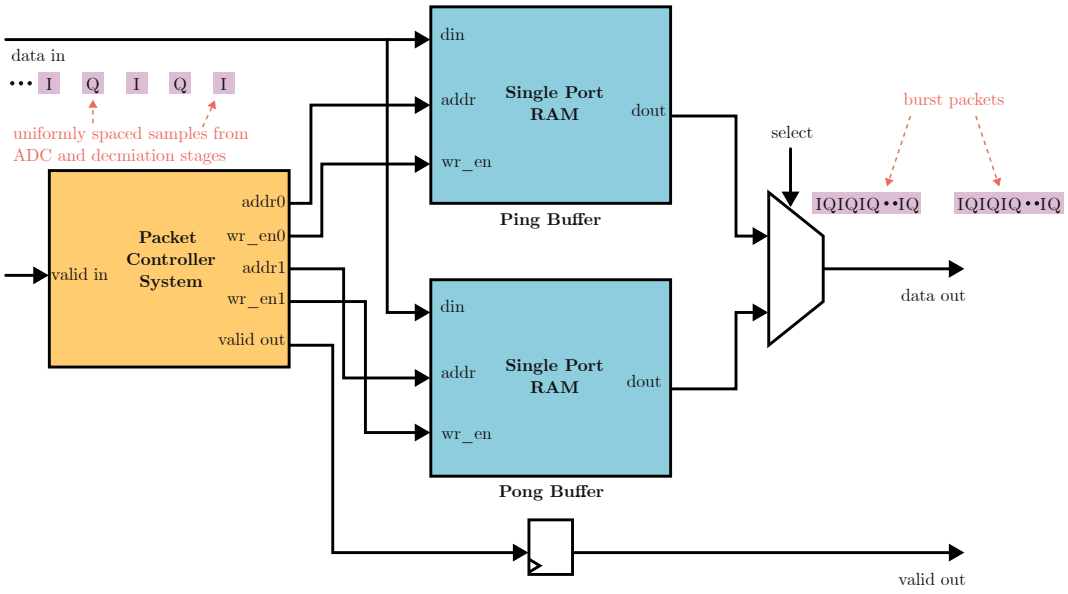



Figure 4.2 Ping-Pong buffer.

### 4.3.2 Convolutional Layers

Convolutional layers pose a computational challenge due to the inherent complexity of passing a kernel over the input feature map repeatedly. For a

streaming-based CNN architecture, repeatedly passing the kernel over an input can greatly slow down the throughput of the deployed model. To support a constantly streaming input of samples, the architecture must maximise throughput and make design choices to support this. The GeMM transform complexity optimisation, from Chapter 3, illustrated that the operational complexity of the convolutional layer can be simplified to a matrix multiplication at the sacrifice of repeated input sample calls.

In traditional convolution-to-matrix transformations such as those found in the BLAS library [103], kernel and input matrices are flattened using a 'channels last' ordering. This structure, discussed in Chapter 3, requires the full input feature map to be available before transformation can begin, which limits streaming efficiency in real-time systems.

### GeMM Transformation

In this thesis, an alternative GeMM transformation is proposed, referred to as the 'channels first' approach, which is specifically designed for streaming input data in radio receiver pipelines. Rather than interleaving the kernel window columns after channels, the transformation interleaves rows after channels. This adjustment allows the input data to begin transforming as soon as enough samples have been received to form the first valid kernel window, instead of waiting for the entire input frame.

The 'channels first' modification significantly improves throughput in scenarios where data arrives sample-by-sample, such as in wireless communication systems with continuous I/Q input streams. Figure 4.3 illustrates the conventional channels-last transformation and its limitations for streaming-based applications, while Figure 4.4 demonstrates the proposed channels-first method and how it enables low-latency data ingestion in a streaming pipeline.

In Figure 4.3, the feature maps coming from the previous layer (or the RF-ADC and decimation stages) enter the convolutional layer memory buffer either as interleaved 1D samples, or a frame of channels at a time. As the GeMM transform employs the channels last method here, a streaming GeMM transform function outputs the transformed samples one channel at a time, where the kernel has passed over the full first channel before proceeding to the next channel, and concatenates the resulting outputs into one matrix ready for multiplication with the GeMM transformed weights. While this description frames the operation in terms of building a full matrix, in practice

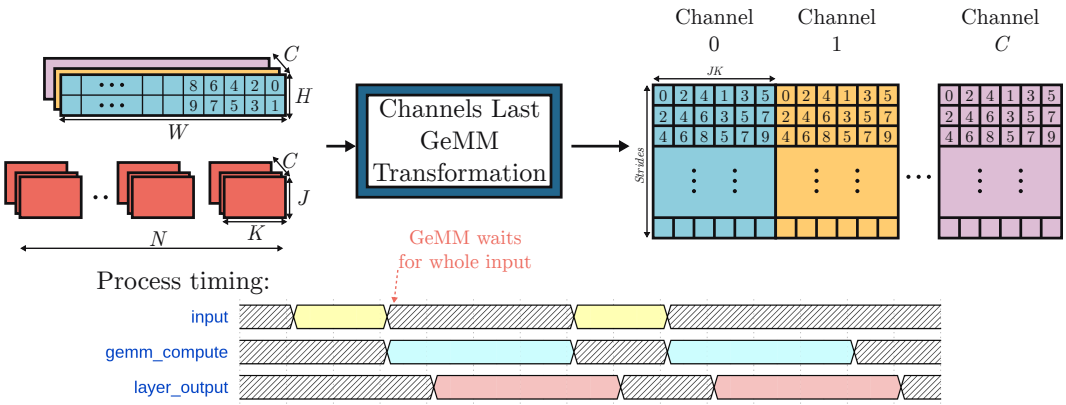



Figure 4.3 Channels last GeMM transform.

the computation is performed using streams of samples and on-chip memory, without requiring all samples to be grouped explicitly.

Since the framed data entering the convolutional layer comes in with a frame of channel data at a time, the convolutional layer needs to wait that enough frames of samples have been stored in the GeMM function in order to begin outputting the transformed feature map. The transform outputs a channel last configuration and therefore the layer would need to wait for almost the full frame to be received before it can begin transforming. This delay hinders the fluidity of the dataflow model and staggers the flow of data in the model.

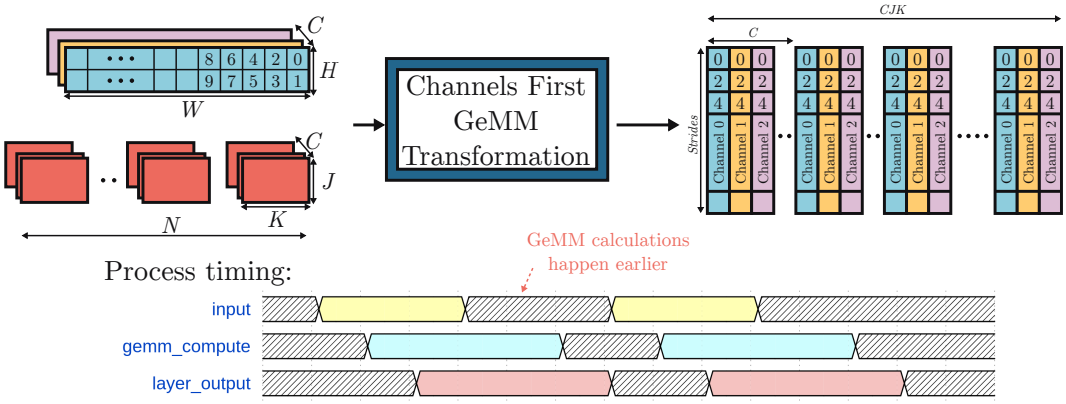



Figure 4.4 Channels first GeMM transform.

In contrast, since the data received in the layer comes in as frames of channels, then it makes sense to process the frames of channels as they enter the layer. The GeMM transform can be altered to support the data structure of the incoming samples by instructing the Sliding Window Generator (SWG) (covered in the next section) to process the channel $C$ frames first followed by the width $W$ and height $H$ of the input feature map. Figure 4.4 illustrates the better efficiency of the 'channels first' GeMM transform when receiving frames

of channels from the previous layer. Performing channels first results in the layer not needing to wait for the full feature map to enter the layer before the SWG can begin to process the transformation.

### Sliding Window Generator

The SWG core exists at the heart of the deployed convolutional layer for this architecture. The SWG receives input samples into the layer memory and transforms the samples in accordance to the GeMM transform formula detailed in Figure 3.13b, Chapter 3. The transformed samples are subsequently sent to a matrix-vector multiply stage where the layer output is calculated, before the optional bias and activation function is applied.

Figure 4.5 shows the SWG core in the convolutional layers. The SWG system consists of on-chip BRAM that is large enough to store the incoming feature map, and a state machine controller handling the RAM address read and write tasks. The incoming samples to the layer are first stored in an input buffer in the on-chip RAM in sequential address spaces through the `wr_addr` signal. The state machine performs the input feature map GeMM transform on the stored samples by reading out samples in accordance with the transformation formula with the `rd_addr` signal. These read samples are then streamed to the matrix-vector multiply stage, where they are multiplied and accumulated with pre-processed transformed layer kernels which are also stored in on-chip RAM. The SWG outputs the transformed feature map incrementally, either sample by sample or row by row, and conducts the multiplication with the GeMM-transformed filter weights to generate the layer output.

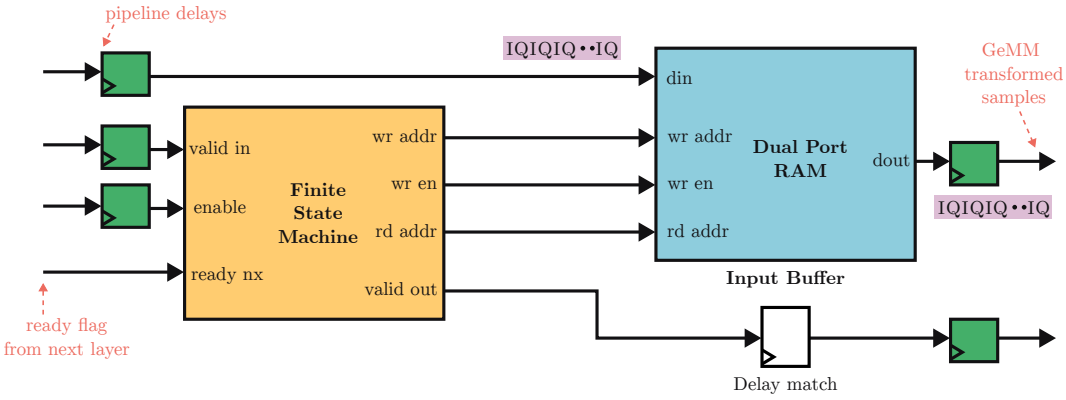



Figure 4.5 Diagram of the SWG core.

The modified GeMM transformation, described in Figure 4.4, enables the SWG core to begin transforming the input feature map as it is being received.

In the first convolutional layer of a CNN for radio applications, the input is received from the ADC and receiver IP stages where this data is a stream of complex I/Q samples forming a feature map of shape $C \times H \times W$, where $C = 1$, $H = 2$ (for I and Q), and $W$ is the frame length.

The transformation uses a 'channels-first' ordering, interleaving data by channels, rows, then columns, which allows the SWG core to start producing rows of the transformed matrix $\tilde{X}^{\text{conv}}$ as soon as the first kernel window is filled. This enables real-time operation without waiting for the entire feature map.

During live operation, the SWG core performs GeMM transformations only on the input feature map. Kernel weights are transformed offline and stored in BRAM for reuse during matrix-vector multiplication.

The SWG supports two modes of operation:

- **Streaming samples:** When receiving data directly from the front-end, samples stream in continuously and are transformed on the fly.
- **Streaming frames:** When processing intermediate feature maps from previous layers, the input arrives as framed tensors with multiple channels. The SWG reads these frames from parallel input buffers and performs the GeMM transform accordingly.

To manage this behaviour in real-time, the SWG operates as a Finite State Machine (FSM). It transitions through states for monitoring the availability of data, generating GeMM'd samples. Figure 4.6 shows the simplified internal control logic of the SWG core.

### Matrix-to-Vector Multiplication

In Section 4.3.2, the GeMM transform was used to reduce the complexity of convolution by restructuring the memory access pattern. Instead of sliding a kernel window across the input feature map in real-time, the transformation flattens both the input and the kernel into 2D matrices. This enables the convolution to be processed as a matrix multiplication, which can be parallelised for more efficient computation.

Recall, from Chapter 3, that the GeMM transformed convolutional layer can be calculated as a matrix multiplication:

$$\tilde{Y}^{\text{conv}}_{N\times S} = \tilde{\Theta}^{\text{conv}}_{N\times CJK}\tilde{X}^{\text{conv}}_{CJK\times S} + b^{conv}_{N\times 1} \tag{4.1}$$

In a dataflow model, the aim is to output results as soon as they are ready to be consumed by the next layer. The Matrix-to-Vector Multiplication

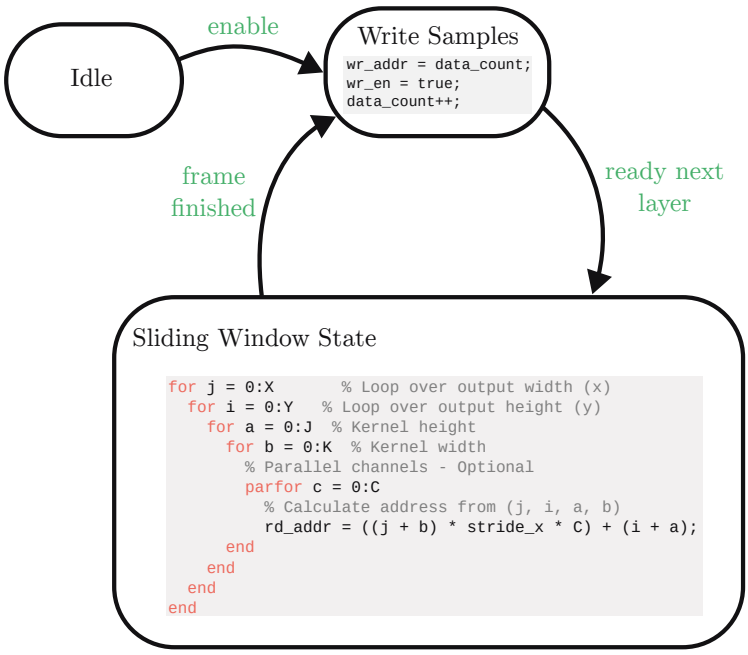



Figure 4.6 SWG state machine.

(MVM) stage takes the transformed inputs and performs a batched multiply, a multiplication on a frame of samples in parallel, to produce a stream of output frames. This sections explores how that operation works under two input conditions:

- **A single-sample stream** directly from the RF-ADC and decimation stages, where the input feature map has one channel.
- **A frame-based stream** from a previous layer, where the input has multiple channels.

Figure 4.7 shows the hardware implementation of a convolutional layer for the proposed accelerator architecture. The setup depicted is for the case where there are single sample inputs to the convolutional layer. The samples are stored in a memory buffer where the SWG reads the samples out in accordance with the GeMM transform. These samples are then sent to the MVM subsystem where the transformed samples are multiplied and accumulated with the on-chip stored filter weights in the MAC unit. Figure 4.8 illustrates the operation of the MAC unit, where a threshold controls when the accumulated data is output.

The MVM system, shown in Figure 4.7, receives one sample at a time from the feature map memory buffer and replicates it across a stack of MAC units, each paired with its own RAM. Each MAC unit also has a counter that tracks how many samples have entered, so that it can fetch the correct weight from

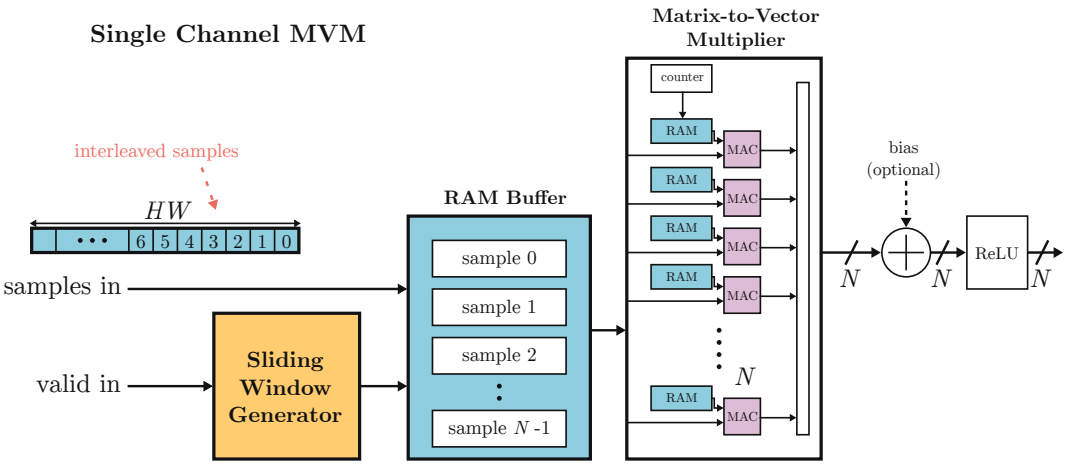



Figure 4.7 Convolutional layer with single sample input.

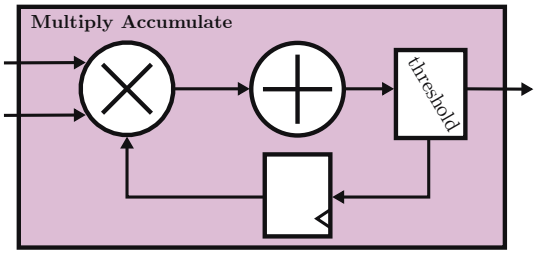



Figure 4.8 The multiply accumulate (MAC) unit.

RAM at each step. The MAC performs multiply-and-accumulate operations across $CJK$ input values before producing a result.

In this design, the convolutional layer is parallelised across the number of filters, $N$. The system only produces an output when a complete result is ready to be sent to the next layer. As such, this implementation can be described as input-parallel and output-stationary.

Figure 4.9 depicts a MVM that processes a single with multiple channels $C$. Here the received signal is received in frames of size $C$ that are stored in a parallel array of buffers. The SWG then performs the GeMM transform in framed batches and passes these frames batches to multiple MVMs and producing the layer output.

### 4.3.3 Fully-connected Layers

The core characteristics of a FC layer are dense connections, where every neuron of the layer is connected to every neuron from the previous. The FC layer is parameter-heavy because the layer tends to have a large number of trainable parameters which can make them computationally expensive. Since every neuron of a FC layer has connections and weights to every neuron of the

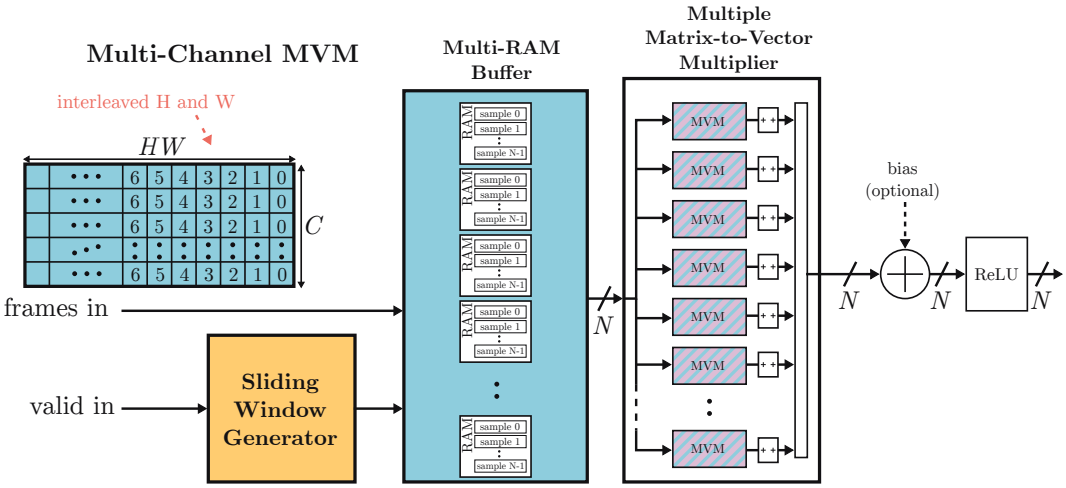



Figure 4.9 Matrix-vector multiplication system with framed input.

previous layer, the layer calculation can be expressed as a matrix multiplication where the input feature map $\boldsymbol{x}$ is multiplied with weight matrix $\boldsymbol{\Theta}$ to produce an output of $\boldsymbol{y}$, as discussed in Chapter 3.

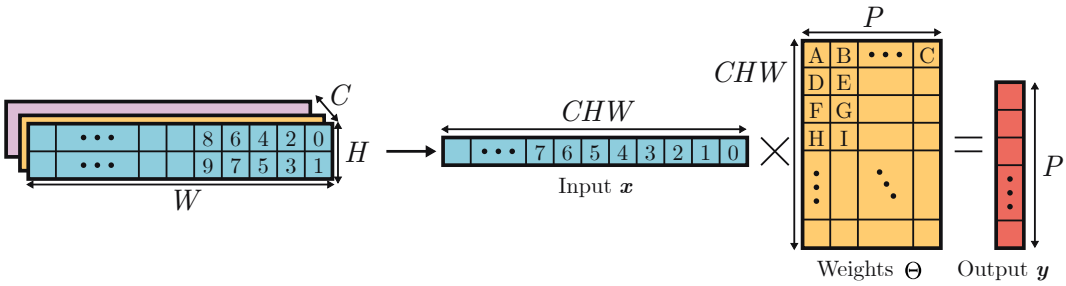



Figure 4.10 Fully-connected layer interpreted as a matrix multiplication.

In Figure 4.10, the feature map from the previous layer enters the FC layer with dimensions: channels $C$, height $H$, and width $W$. This 3D feature map is flattened into a one-dimensional input vector $\mathbf{x}$, where the total number of elements is $n = C \times H \times W$. The vector $\mathbf{x} \in \mathbb{R}^{1\times n}$ is then multiplied by a weight matrix $\boldsymbol{\Theta} \in \mathbb{R}^{p\times n}$, where $p$ is the number of neurons in the FC layer. The result is an output vector $\mathbf{y} \in \mathbb{R}^{1\times p}$, with each element representing the activation of a single neuron.

To realise a FC layer to hardware within the proposed streaming CNN architecture, a similar approach to the convolutional layer implementation has been implemented, as visualised in Figure 4.11. Firstly, the input feature map to the layer enters as a stream of samples or frames, which are stored in a buffer or series of buffers on-chip. The Buffer Controller then reads the samples from the buffer sample-by-sample to flatten the input. In this work, the input is flattened channel first before it is sent to the MVM. The MVM operates

exactly the same as in the convolutional layers, where a column of MAC units process samples in parallel to produce the output of the layer. The number of parallel MACs are equal to the number of neurons $p$ in the FC layer, and therefore a framed output equal to the number of neurons $p$ is produced. This is then followed by an optional serialisation stage and an activation function.

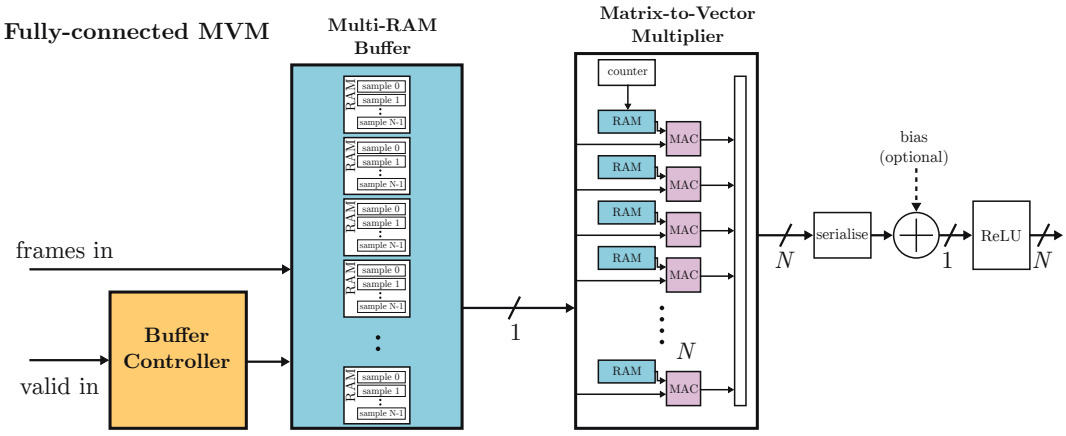



Figure 4.11 Fully-connected layer hardware implementation.

### 4.3.4 Activations and Bias

In neural networks, activations and bias terms play a crucial role in the performance and accuracy of the network. Activations determine the non-linear transformations applied to the output of each layer, while the bias terms shift the activation to allow the network to better fit the data.

Activation functions, such as Rectified Linear Unit (ReLU), Sigmoid, and Tanh, are vital in introducing non-linearity into the network. These non-linear functions enable the model to learn and represent complex patterns. For a hardware accelerator, ReLU has been widely chosen as a popular activation function due to its simplicity [25], as it can be efficiently implemented using a comparison operation. Sigmoid and Tanh are more resource-intensive since they require exponentiation and division operations, often implemented using LUTs or iterative approximation methods such as CORDIC [104].

All activation functions performed in the proposed architecture occur within the PL with the exception of the Softmax function, which due to its computationally complex equation, is performed in PS. This occurs after the final layer classification results are transmitted to the PS, turning them into a probability distribution.

In the proposed architecture, the ReLU activation is mapped using a threshold comparison where each sample or frame is passed into the IP and every negative value is set to zero through the operation $y = max(x)$. The ReLU function is depicted here as it is a common activation function and one that is simple to implement in PL hardware [105].

The bias terms are optional. They are added to the linear combinations of inputs in each layer, effectively shifting the activation function to better fit the data. The bias terms are applied to the output of a layer where the bias values are stored on-chip for the hardware accelerator to add the values quickly in real-time. In this work the bias terms were used initially, but later dropped due to there being no significant improvements observed in the accuracy performance.

## 4.4 Neural Network for Modulation Classification

To demonstrate the streaming CNN architecture's operation, an example neural network architecture for performing modulation classification was selected. The CNN chosen for this work is based on the architecture proposed in [19]. The objective is to demonstrate that the developed custom streaming-based architecture can process each sample received by a radio transceiver in real-time, while maintaining the classification performance of the original model.

The network comprises two convolutional layers followed by two FC layers, forming a 4-layer topology. All layers employ a ReLU activation function, except for the final output layer, which uses a Softmax activation to produce a one-hot encoded class prediction. Although relatively small compared to deeper AI models, this network offers a practical balance between computational efficiency and classification performance. Its compact structure makes it well-suited for hardware prototyping and demonstrates that shallow networks can be effective for tasks such as modulation classification, where signal characteristics can be captured without extensive hierarchical features. The architecture is illustrated in Figure 4.12.

The convolutional layers serve as feature extractors, identifying local patterns in both the real and imaginary components of the input signal. These layers exploit spatial relationships between adjacent samples to capture relevant modulation features [25]. The subsequent FC layers aggregate and compress the learned features, progressively reducing the dimensionality before reaching the

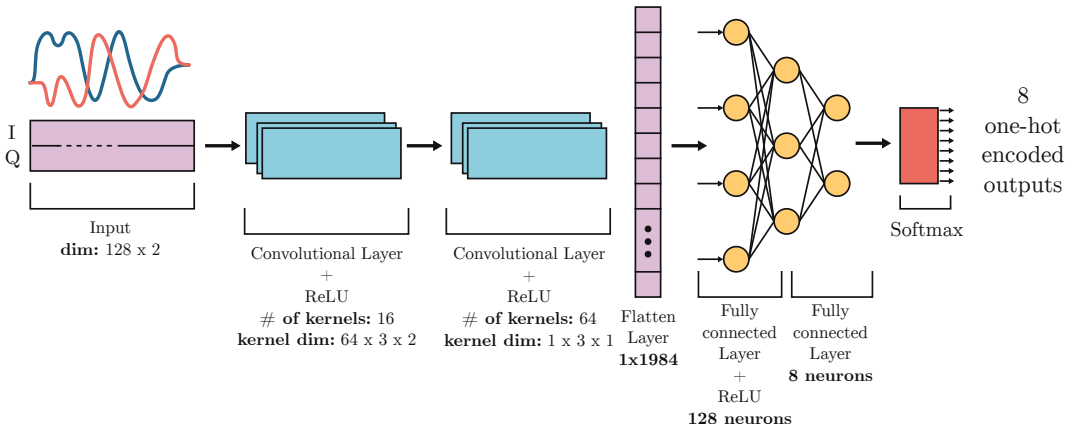



Figure 4.12 Neural Network Topology for RadioML-based Modulation Classification.

final classification output. Each neuron in a FC layer computes a weighted sum of its inputs, enabling the network to model complicated decision boundaries.

Non-linearity is introduced between layers via the ReLU activation, which improves the network's ability to approximate non-linear functions and detect subtle signal variations. The final Softmax activation normalizes the output of the last layer into a probability distribution over the eight modulation classes.

The network accepts a 3D input tensor with dimensions $C \times W \times H = 1 \times 128 \times 2$. The first convolutional layer applies 64 filters of size $1 \times 3 \times 1$, producing an output of shape $64 \times 126 \times 2$. The second convolutional layer uses 16 filters of size $64 \times 3 \times 2$, resulting in a compressed feature map. This output is flattened and passed through the two FC layers, ultimately yielding an 8-element output vector corresponding to the classification classes. ReLU activations are applied after each layer, and the final Softmax layer provides the class probabilities.

This work focuses on predicting only the digital modulation schemes, as similar hardware circuitry for further demodulation can be used for all eight.

## 4.5 Training

The CNN model is trained using the RadioML 2016.10a dataset, described in Section 2.2.3, with the PyTorch deep learning framework [79]. The dataset is first unpacked and reformatted into a structure compatible with PyTorch input pipelines. It is then split into training, validation, and test subsets.

Model layers are initialized according to the topology shown in Figure 4.12. Training is accelerated using an NVIDIA RTX 2060 GPU [106], enabling

efficient iteration over the large dataset and supporting parallel computation and gradient updates through backpropagation.

### 4.5.1 Dataset Preparation

The RadioML 2016.10a dataset contains signal samples for 11 modulation schemes, with SNR values ranging from -20dB to 18dB. For this implementation, only 8 digital modulation schemes are retained to maintain a fully digital classification pipeline.

Analogue modulation types, such as Analogue Modulation (AM) and Frequency Modulation (FM), are excluded from this work for both practical and architectural reasons. These schemes require fundamentally different pre-processing and demodulation techniques compared to digital modulation schemes, which are designed to recover discrete bits from a transmitted signal. In contrast, analogue modulations typically produce a continuous waveform at the output, like audio or baseband analogue signals, rather than bit representations. Supporting such formats would require entirely different circuitry and signal processing pipelines, making their inclusion beyond the scope of this work if the modulation classification application is taken further to demodulate the identified signal.

The selected digital modulation schemes are: QPSK, BPSK, 16-QAM, 64-QAM, 8PSK, PAM4, GFSK, and CPFSK. The dataset comprises 160,000 frames, evenly distributed across modulation types and SNR levels. Each frame consists of 128 complex-valued samples represented as two I/Q channels. Every modulation-SNR pair contains 1,000 examples, yielding 20,000 frames per modulation scheme.

Figure 4.13 shows time-domain plots of each modulation type at 18 dB SNR, illustrating the variation in waveform structure across the dataset. Each of the waveforms depicted was generated using the method described in Section 2.2.3 with time, frequency, and phase offsets applied to the signal along side multipath channel and AWGN.

Before the dataset can be used to train the DL model, it is prepared for training by splitting it into training, validation, and testing sets. The original RadioML 2016.10a dataset is first restructured from a dictionary format into a tensor of input samples and a corresponding list of labels, indicating the modulation scheme. Only the digital modulation classes are retained, as discussed earlier. The dataset is then randomly shuffled and divided with 70% allocated to training, with the remaining 30% reserved for validation and

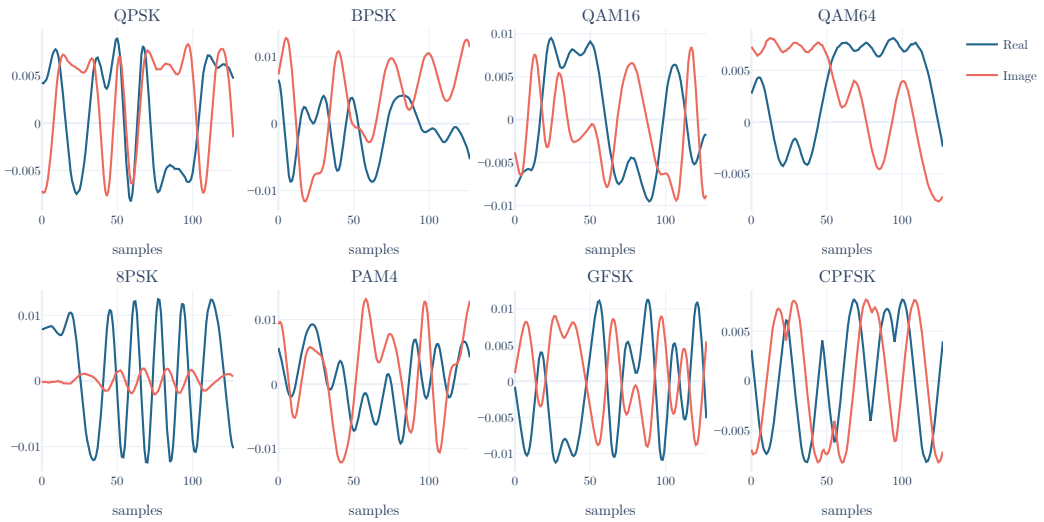



Figure 4.13 Time-domain plots of each modulation scheme at 18 dB SNR.

testing. From the validation and testing portion, a further 1:2 split is used to separate validation and testing data. Labels are converted to one-hot encoded vectors for compatibility with the classification model, which produces one class prediction at a time. A visual overview of the dataset preparation process is shown in Figure 4.14.

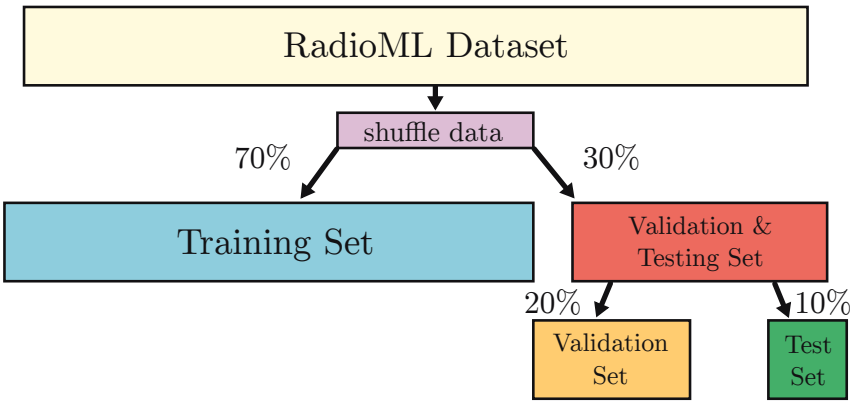



Figure 4.14 RadioML dataset split between training, validation and testing sets.

The training and validation sets are used during training to learn how to classify the different modulation schemes and to tune the model's hyperparameters. The validation set helps monitor how well the model generalises to the data during training and avoids overfitting. Early stopping is triggered if the validation loss has not improved over the course of training. The test set is kept completely separate and is only used at the end to evaluate the final model

performance on unseen data. The test set is also reserved for the evaluation of the architecture on the embedded device.

### 4.5.2 Network Training

With the dataset prepared and split into training, validation and testing sets, the CNN model can now be defined and trained. The network is designed to operate on 128-sample long I/Q frames, which are reshaped into a $(1 \times 2 \times 128)$ tensor to represent the real and imaginary components of the signal.

The model is trained using categorical cross-entropy loss and the Adam optimiser [87]. Key training parameters are summarised in Table 4.1.

Table 4.1 CNN training parameters.

| **Parameter** | **Value** |
|---|---|
| Epochs | 100 |
| Batch size | 128 |
| Early stopping (Patience) | 8 |
| Loss function | Cross Entropy Loss |
| Optimiser | Adam |
| Learning rate | $1e^{-4}$ |
| Weight decay (L2 Regularisation) | $1e^{-5}$ |

Figure 4.15 shows the training and validation loss over the training process. The network converges after approximately 58 epochs, with the validation loss stabilising around 1.1. The validation loss consistently tracks the training loss, which suggests that the model generalises well and does not overfit. This is also a sign that the dataset provides a good range of signal scenarios for training. The training process was halted early through the early stopping mechanism as the validation loss did not continue to improve, even as the training loss was still decreasing. Had the early stopping not occurred, the validation loss would not continue to improve and instead begin to rise again. Early stopping is an excellent method for verifying the model has not overfitted.

After training, the test set is used to evaluate the model's performance on unseen data. The overall accuracy across all SNR levels is shown in Figure 4.16, the per-class performance is shown in the plot in Figure 4.17, and a confusion matrix showing the per-class correct and incorrect classifications for signals at 18dB SNR can be seen in Figure 4.18. The accuracy is reported from a range of $0 \rightarrow 1$, corresponding to $0\% \rightarrow 100\%$.

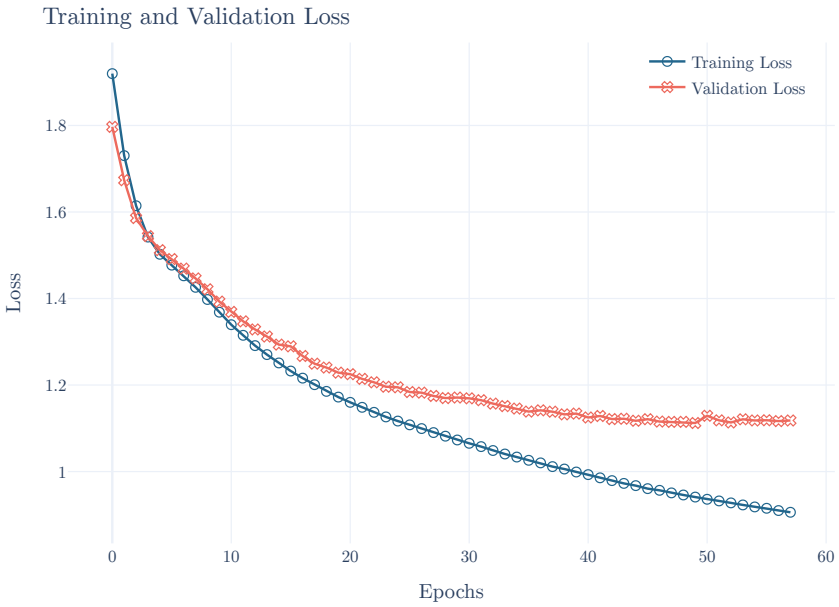



Figure 4.15 Training and validation loss during training.

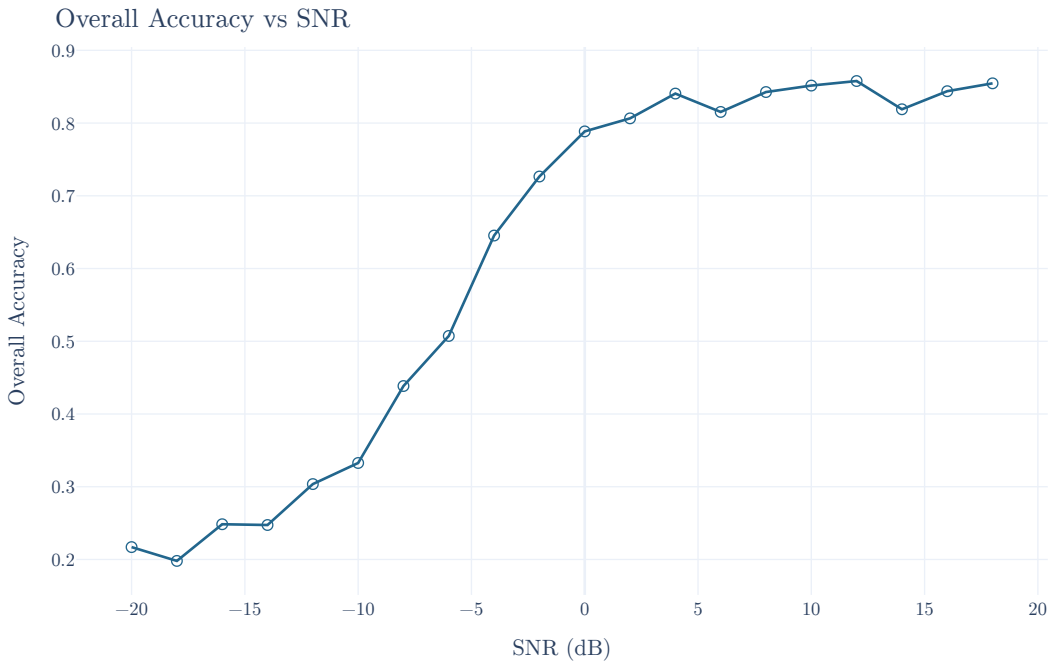



Figure 4.16 Accuracies for all classes across all SNRs.

The model performs best on modulation types with lower symbol rates, such as BPSK, GFSK, CPFSK, and PAM4. The performance of the schemes degrades as the symbol rate increases with both QAM schemes performing the worst. Overall, the model demonstrates strong classification performance on SNR values above 0dB shows good generalisation across all modulation types.

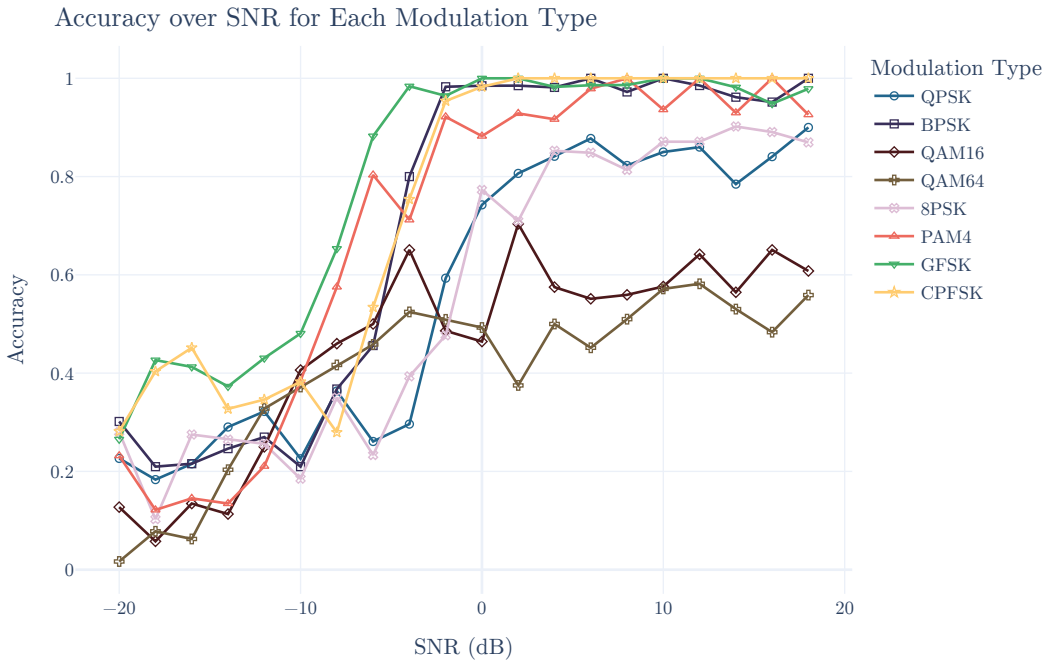



Figure 4.17 Accuracies for each class across all SNRs.

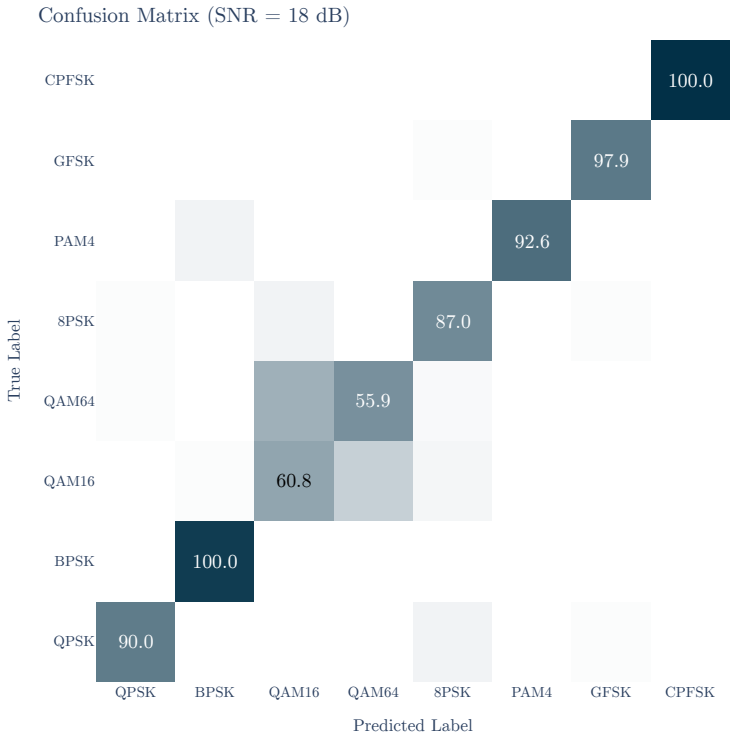



Figure 4.18 Confusion matrix of predictions vs true labels from the trained software model on the RadioML test set at SNR=18 dB.

Upon the completion of training and the evaluation of the test set to assess the performance of the trained model, the weights are preserved for use in the creation of the streaming-based CNN architecture.

The candidate network developed here is later used to construct the CNN architecture that is the main focus of this chapter. While the accuracy performance of the trained model is not optimal, the goal of this work is not to provide the best accuracy network, but rather to demonstrate how a trained network can be transferred to FPGA PL for real-time operation. The resulting accuracy for the candidate network trained on the RadioML dataset achieves a ∼85% at the higher SNRs, which is comparable to other works for networks and inputs of this size [70].

## 4.6 Design Workflow

As outlined in Section 4.4, the RadioML dataset and neural network architecture developed by O'Shea et al. are chosen as an example model and application for refining the proposed CNN architecture. Modulation classification, the task of identifying the modulation scheme applied to a received signal, can be instrumental to a spectrum sensing application in a DSA context. The RadioML dataset is widely recognised for its efficacy in demonstrating DL tasks in wireless communications. The selected neural network configuration demonstrates a variety of CNN layer features, while maintaining a moderate and manageable number of dimensions to prevent excessively long training and implementation times. With a predefined and operational neural network topology and application, the creation of customised CNN architecture stemming from fundamental DSP functionality becomes feasible.

This section will detail the design workflow and methodology behind converting the trained PyTorch model from Section 4.5.2, to a streaming dataflow CNN implementation operating in the PL. This is achieved while maintaining the core underlying functionality of the model with minimal sacrifice to the model accuracy performance. The goal of the resulting deployed model is to achieve similar classification accuracies while accelerating computation through the parallel capabilities of the logic fabric. The resulting accelerated model will receive a stream of samples and produce a prediction of the signal's modulation scheme.

The design workflow leverages the functionality from the trained CNN model discussed in Section 4.5.2, transforming it into a hardware-synthesisable format using MathWorks HDL Coder. HDL Coder enables users to generate synthesisable Verilog and VHDL code from MATLAB functions and Simulink models [101]. This tool is widely used in digital design and FPGA programming

due to its capabilities in simulating and deploying engineering algorithms efficiently on hardware platforms, and is popular with SDR engineers for its integration with Hardware Description Language (HDL) compatible MathWorks libraries like the Wireless HDL Toolbox [107].

Using a dataflow paradigm, HDL Coder within Simulink is particularly advantageous for wireless communications applications. The CNN implementation resulting from this work also exploits the data flow methodology for design. Given its compatibility with this data flow approach and its capabilities for developing hardware efficient implementations, HDL Coder is a suitable choice for realising the data flow CNN model in hardware.

The resulting hardware CNN model must have the same functionality as the trained CNN model. To maintain the same functionality, the CNN function must be built using basic functions such as additions, multiplies and on-chip memory. Hardware optimisations, like those discussed in Chapter 3, are utilised to exploit parallelism in the PL. Figure 4.19 illustrates the process of building the resulting hardware-implemented CNN.

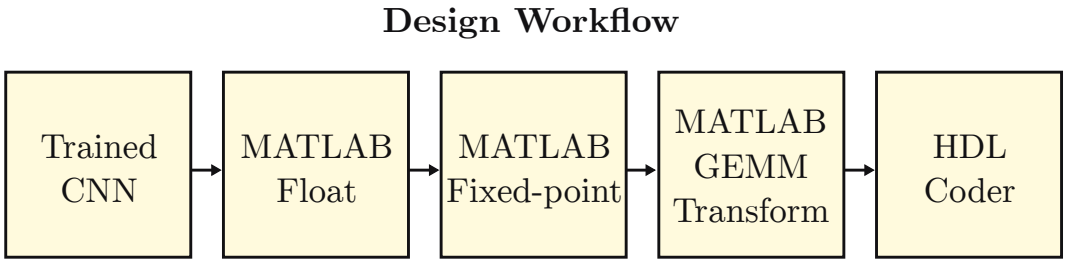



Figure 4.19 CNN architecture design workflow.

The design workflow uses a variety of design software to train and implement the CNN onto the FPGA device. To train the CNN, software tools such as PyTorch [79] or Tensorflow [80] can be used. MATLAB/Simulink is used to convert the trained weights into a hardware optimised architecture, following steps of optimisation. During the conversion from software to hardware using MATLAB and Simulink, each stage can also be simulated to verify the design at each stage of the transformation.

### 4.6.1 Trained CNN Model

In Figure 4.19, the depicted workflow illustrates the steps involved in developing a hardware-efficient CNN model, which mirrors the computational functionality of the CNN model trained in Section 4.5.2. After training and testing the

network to ensure reliable prediction of modulation schemes, the network weights are stored in a `.mat` file for integration with MATLAB.

### 4.6.2 MATLAB Floating-point Functionality

The next step in the workflow is to replicate the CNN model in MATLAB using floating-point arithmetic. This version is used to simulate the neural network inference pipeline without any quantisation or hardware-specific limitations. The goal is to verify the model's functional behaviour before transitioning to a fixed-point hardware-oriented implementation.

A convolution function, written in MATLAB, shown in Listing 4.1, was written to perform 3D convolutions across input feature maps using saved filter weights and biases. The function supports configurable stride and input dimensions, matching the structure of the trained PyTorch model. A detailed explanation of the convolutional layer was provided in Chapter 3.

Listing 4.1 **conv**: floating-point convolution function in MATLAB.

```
function out = conv(data, filt, bias, s)
%#codegen
    % Convolves 'filt' over 'data' using stride 's'
    % N - Number of filters, C - Channels, H - Height, W - Width

    [filt_N, filt_C, filt_H, filt_W] = size(filt);
    [data_C, data_H, data_W] = size(data);

    % "Dimesions of number of channels for filter and data must match"
    assert(filt_C == data_C, 'Dimensions must agree');

    % Calculate the output dimension (C,H,W)
    outdim_C = ceil((data_C - filt_C)/s)+1;
    outdim_H = ceil((data_H - filt_H)/s)+1;
    outdim_W = ceil((data_W - filt_W)/s)+1;
    outdim_N = filt_N; % Becomes the new C

    out = zeros(outdim_N, outdim_H, outdim_W);

    % Convolve the filter over every part of the image, adding the bias at each
    step.
    for curr_N = 1:filt_N
        out_h = 1;
        for curr_H = 1:s:data_H-filt_H+1
            out_w = 1;
            for curr_W = 1:s:data_W-filt_W+1
                filter = reshape(filt(curr_N,:,:,:), size(filt, 2), size(filt,3),
    size(filt,4));
                data_partial = data(:,curr_H:curr_H+filt_H-1,curr_W:curr_W+filt_W-
    1);
                out(curr_N,out_h,out_w) = sum(filter .* data_partial, 'all') +
    bias(curr_N);
                out_w = out_w + 1;
            end
            out_h = out_h + 1;
        end
    end
end
```

The input feature map is structured as a 3D tensor with dimensions for channels $C$, height $H$, and width $W$. The convolutional filters are 4D tensors of shape $(N, C, J, K)$, representing the number of filters $N$, channels $C$, kernel height $J$, and kernel width $K$. Each filter is slid across the input with the specified stride, and a bias $b$ is added after each multiply-accumulate operation.

The complete floating-point CNN model is implemented using a function `cnn_float` (Listing 4.2) that chains together two convolutional layers followed by two FC layers. ReLU activations are applied after each layer, and the final output is a raw class probability vector. This MATLAB implementation accepts the trained weights and biases exported from the PyTorch model.

Listing 4.2 `cnn_float`: floating-point CNN model in MATLAB.

```
function [conv1,act1,conv2,act2,flatten,fc1,act3,fc2, act4] = cnn_float(input,
    wconv1,wconv2,wfc1,wfc2,b1,b2,b3,b4)
%CNN_FLOAT function to process convolutional neural network for AMC without
%transforms or quantisations. Used to compare against trained data.

    % Layer 1
    conv1 = cnn.conv(input,wconv1,b1,1);
    act1 = conv1;
    act1(act1<0) = 0;

    % Layer 2
    conv2 = cnn.conv(act1,wconv2,b2,1);
    act2 = conv2;
    act2(act2<0) = 0;

    % Flatten
    conv2_2d = reshape(act2,size(act2,1),[]);
    flatten  = reshape(conv2_2d,1,[]);

    % Layer 3
    fc1 = flatten * wfc1 + b3;
    act3 = fc1;
    act3(act3<0) = 0;

    % Layer 4
    fc2 = act3 * wfc2 + b4;
    act4 = fc2;

end
```

To confirm correctness, the output feature maps from each layer of the MATLAB floating-point model are compared against those produced by the original PyTorch model using the same input data. Minor differences are expected due to precision and rounding behaviours between MATLAB and Python, but the outputs remain functionally equivalent. This step validates the floating-point implementation as a reference before moving to fixed-point.

### 4.6.3 MATLAB Fixed-point Functionality

Once the floating-point version of the CNN model is verified, the next step is to convert the implementation to use fixed-point arithmetic. Fixed-point formats are commonly used in FPGA designs, especially in DSP applications, due to their lower resource usage and faster operation compared to floating-point arithmetic.

In this implementation, the CNN model is treated similarly to a DSP pipeline. Converting to fixed-point requires careful management of numeric precision to avoid information loss from quantisation. Key design considerations include bit growth across layers, the use of appropriate dynamic ranges, and strategic truncation to balance accuracy and hardware efficiency. These are standard practices in DSP filter design and are applied here to ensure that the network remains both accurate and hardware-friendly when mapped to the PL.

#### Quantise Layer Weights

The trained model stores its weights in single-precision floating-point format, which offers a wide dynamic range but is inefficient for hardware implementation on FPGAs. Floating-point arithmetic is resource-intensive and generally avoided in favour of fixed-point formats in real-time inference applications.

Since the neural network is used purely for inference, its weights are static and can be quantised ahead of time. By analysing the distribution of trained weights in each layer, suitable fixed-point representations can be selected to minimise quantisation error while significantly reducing hardware complexity. Figure 4.20 shows histogram plots of the weight distributions across all layers of the network. These plots are used to guide the choice of bit-width and scaling factors for fixed-point representation.

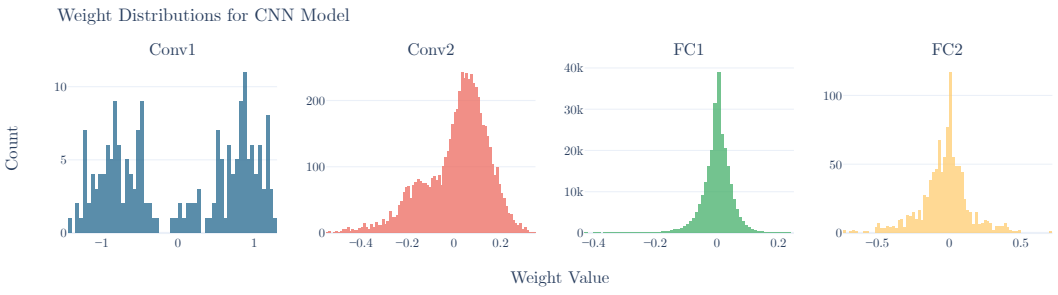



Figure 4.20 Histogram plot of weight value distribution in each layer of the trained CNN model.

As seen in Figure 4.20, for convolutional layer 1, the kernel weights are predominantly centred around $-1$ and 1, respectively, and reach a maximum just over 1 and $-1$. Convolutional layer 2 represents a more 'normal' distribution, offset and centred around 0.1. This layer has kernel weights with a maximum and minimum of $-0.45$ and 0.3, respectively. For the FC layers, both models also have a 'normal'-like distribution of weights centred around 0. The first FC layer has minimum $-0.4$ and minimum 0.2, while the second has minimum $-0.55$ and maximum 0.55. Both the FC layers have a large number of samples at or around 0.

Despite being stored in a 32-bit floating-point format, the precision offered by this format far exceeds the practical needs for the network for inference. This observation supports the use of reduced-precision representations. One focus of this chapter is how fixed-point formats can be applied to the weights, activations, and intermediate signals of the CNN to enable an efficient FPGA implementation.

The PL fabric available on AMD's UltraScale+ devices contain DSP48E2 slices which are dedicated MAC resources [108]. Figure 4.21 illustrates the internal structure of the DSP48E2 slice. To maintain a single DSP48 for each MAC operation during the implementation of the neural network in the PL, the kernel/weights bit widths are set to 16 bits and a fractional point can be chosen to best represent each layer's weights.

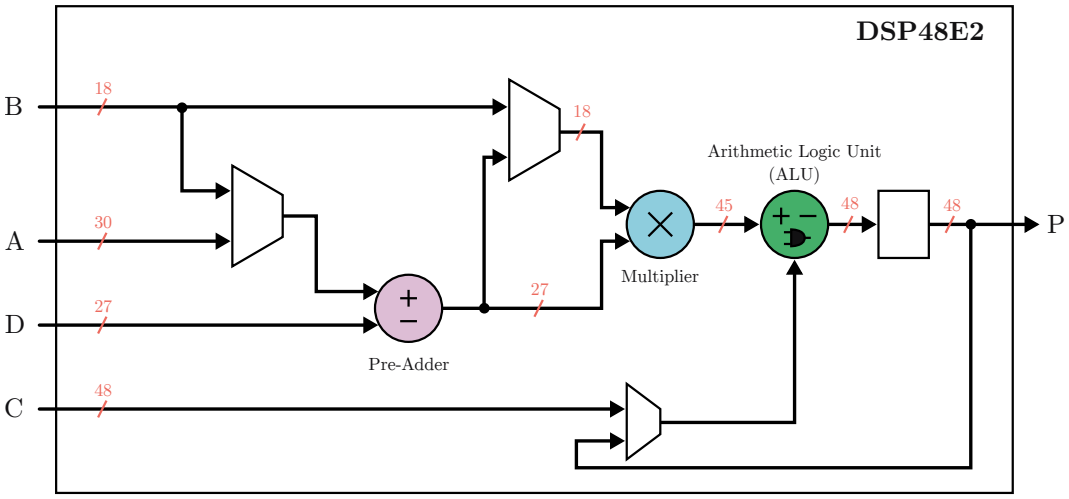



Figure 4.21 The DSP48E2 slice (simplified) [108].

In Table 4.2, each layer is assigned 16 bits to represent all of the kernels/weights in that layer. An integer width is selected depending on the maximum amplitude of any weight in the layer. After analysing the distribution of weights, a fixed-point equivalent that can represent each layer's weights is derived. The

Table 4.2 Fixed-point representation for each layer weight assignment (For Qm.n format, see App. A).

| **Layer** | **Integer Bit-width** | **Fractional Bit-width** | **Dynamic Range** | **Q(m.n) Representation (Signed)** |
|---|---|---|---|---|
| Conv1 | 2-bits | 14-bits | $-2 \rightarrow 1.999985$ | Q2.14 |
| Conv2 | 1-bits | 15-bits | $-1 \rightarrow 0.999985$ | Q1.15 |
| FC1 | 0-bits | 16-bits | $-0.5 \rightarrow 0.499985$ | Q0.16 |
| FC2 | 1-bits | 15-bits | $-1 \rightarrow 0.999985$ | Q1.15 |

number of bits to support the maximum amplitude of each distribution is selected through the integer bits and the remaining bits are allocated to the fractional bits to represent finer precision. This process is also known as PTQ. See Appendix A for a description of the Qm.n format.

#### GeMM Transform and Quantised Network

The CNN MATLAB code from Listing 4.2 is then adjusted to include the GeMM transform, discussed in Chapter 3 Section 3.3.3, and the quantised weights are analysed. By injecting the test set samples from the RadioML dataset, the inter-layer signals and activation quantisation parameters can be realised. Unlike the weight quantisation, the activations and inter-layer signal fixed-point precision need to be solved via experimentation. By using the Fixed-Point Toolbox [102] in MATLAB, the fractional bits can be automatically assigned, while the computation of the CNN is performed. Listing 4.3 shows the MATLAB code for implementing the GeMM-transformed neural network and the quantised weights and activations. Listing 4.4 shows the process of GeMM-transforming the convolutional layer weights.

The resulting inter-layer signal and activation precision are reported in Table 4.3.

The resulting activation and inter-layer precisions fluctuate in dynamic range as the signal passes through the network. This is due to the post-training representation of the floating-point signals. Later, in Chapter 6, an analysis of maintaining a more constant precision range is presented.

### 4.6.4 Dataflow Design Workflow

The dataflow design model provides a structured way to manage data in a communications pipeline, especially when the input data can flow continuously.

Listing 4.3 `cnn_gemm_amc`: GeMM-transformed CNN model with quantised weights and activations.

```
function [conv1, conv1_bias, act1, gemm_conv1, ...
          conv2, conv2_bias, act2, flattened, ...
          fc1, fc1_bias, act3, fc2, fc2_bias, act4] = ...
          cnn_gemm_amc(input, wconv1, wconv2, wfc1, wfc2, b1, b2, b3, b4)
% CNN_GEMM_AMC - 4-layer CNN inference with fixed-point GEMM operations.
% Performs inference on 16-bit fixed-point data using pre-quantised weights.

    % --- Input reshaping and padding for Conv1 ---
    input_padded = cnn.gemm.indexPaddingReplication([2, 128], [1, 3]);
    gemm_input   = fi(input(input_padded), 1, 16);  % Fixed-point conversion

    % --- Convolution Layer 1 + Activation ---
    conv1       = fi(wconv1 * gemm_input, 1, 16);
    conv1_bias  = conv1 + b1;
    act1        = fi(max(0, conv1_bias), 1, 16);  % ReLU

    % --- Reshape output of Conv1 for next layer ---
    act1_reshaped = reshape(act1, 64, 2, 126);
    act1_vector   = reshape(permute(act1_reshaped, [3, 2, 1]), 1, []);  % Flatten
    to 1D vector

    % --- Prepare input for Conv2 (GEMM transform) ---
    gemm_conv1 = act1_vector(cnn.gemm.gemm_transform_input([64, 2, 126], [2, 3]));

    % --- Convolution Layer 2 + Activation ---
    conv2       = fi(gemm_conv1 * wconv2, 1, 16).';
    conv2_bias  = conv2 + b2;
    act2        = fi(max(0, conv2_bias), 1, 16);  % ReLU

    % --- Flatten for Fully Connected Layer ---
    flattened = reshape(act2, 1, []);

    % --- Fully Connected Layer 1 + Activation ---
    fc1      = fi(flattened * wfc1, 1, 16);
    fc1_bias = fc1 + b3;
    act3     = fi(max(0, fc1_bias), 1, 16);  % ReLU

    % --- Fully Connected Layer 2 (Output) ---
    fc2      = fi(act3 * wfc2, 1, 16);
    fc2_bias = fc2 + b4;
    act4     = fc2_bias;  % Final layer has no activation

end
```

Listing 4.4 GeMM-transforming the convolutional layer weights.

```
% --- Convert and quantise convolutional layer weights ---
W1_perm = permute(Wconv1, [4, 3, 1, 2]); % Reorder dims to [N, C, H, W]
% GeMM transform and quantise to Q2.14
wconv1  = fi(cnn.gemm.roll_out_filter(W1_perm), 1, 16, 14);

W2_perm = permute(Wconv2, [4, 3, 1, 2]); % Same reordering for Conv2
% GeMM transform, quantise to Q1.15, then transpose
wconv2  = fi(cnn.gemm.roll_out_filter(W2_perm), 1, 16, 15).';

% --- Quantise fully-connected layer weights ---
wfc1 = fi(Wdense1, 1, 16, 16);  % FC1 weights to Q0.16 fixed-point
wfc2 = fi(Wdense2, 1, 16, 15);  % FC2 weights to Q1.15 fixed-point
```

It passes data between functional blocks in a streaming fashion, allowing each stage to process data concurrently. By clearly defining how data moves from

Table 4.3 Fixed-point representation for each layer activation (For Qm.n format, see App. A) .

| Signal | Integer Bit-width | Fractional Bit-width | Dynamic Range | Q(m,n) (Signed) |
|---|---|---|---|---|
| Input | 2-bits | 14-bits | $-2 \rightarrow 1.999$ | Q2.14 |
| Conv1 Out | 1-bits | 15-bits | $-1 \rightarrow 0.999$ | Q1.15 |
| Conv2 Out | 5-bits | 11-bits | $-32 \rightarrow 31.999$ | Q5.11 |
| FC1 Out | 8-bits | 8-bits | $-256 \rightarrow 255.999$ | Q8.8 |
| FC2 Out | 7-bits | 9-bits | $-128 \rightarrow 127.999$ | Q7.9 |

source to destination, it becomes easier to optimise each block independently for performance or resource usage. This also means that blocks can be reused or swapped out in other designs while still following the same dataflow approach. Benefits of using this model include improved scalability, deterministic latency, better resource utilisation, and lower overall latency. It is particularly useful for handling large amounts of data, where continuous streams of samples need to be processed without interruption.

In this work, the dataflow model is applied to the design of the CNN trained in Section 4.5.2. The trained model operates as a modulation scheme classifier where samples from an ADC are digitised, down-converted and passed into the neural network for classification. The dataflow design model allows for a theoretically indefinite stream of samples coming from the radio receiver ADC into the CNN. When implemented on PL, each layer of the CNN can operate concurrently and pass data from an input source to the classification destination.

## 4.7 The Complete RadioML CNN Architecture in Hardware

The CNN RadioML network is fully implemented in the PL of the FPGA. Figure 4.22 illustrates the RadioML CNN model built using the custom architecture layers introduced in Section 4.3.

I/Q samples enter the CNN accelerator interleaved from the Ping-Pong Buffer system, arriving as a continuous stream of frames. These samples are stored in the first convolutional layer's input buffer. Once enough data is stored, the SWG core begins the GeMM transformation and continues to

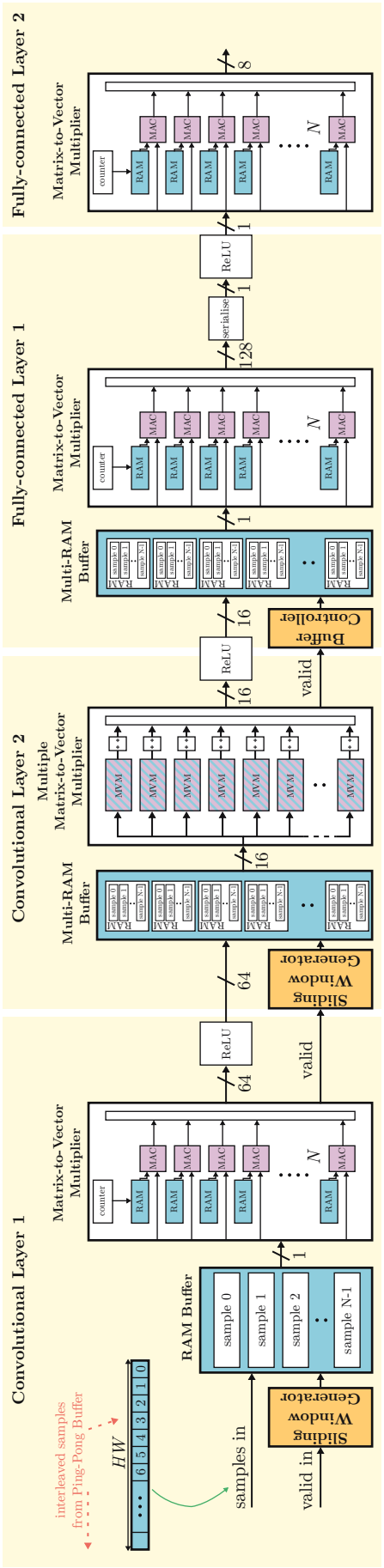



Figure 4.22 The RadioML CNN model in PL using the custom layers.

operate concurrently, producing transformed data while still receiving new samples. The resulting matrix is processed by the MVM stage, which deploys 64 parallel MACs, matching the number of filters $N$ in the first layer. A counter sequentially loads the appropriate weights for each MAC unit. After matrix multiplication, the output is passed through a ReLU activation function before being sent to the next convolutional layer.

The second convolutional layer follows a similar flow, but this time receives frames of input data. Each frame, representing a channel $C$, is written into a Multi-RAM Buffer using several parallel BRAMs, allowing an entire frame to be stored in a single clock cycle. The SWG core again performs a frame-based GeMM transform, distributing the data to 16 parallel MVM cores. Each of these cores contains 16 MACs, enabling efficient parallel computation. As before, the output passes through a ReLU activation stage.

In the first FC layer, input frames are stored in parallel buffers managed by the Buffer Controller. Since this layer does not require a GeMM transformation, samples are passed directly to the MVM stage, either one at a time or as frames (at the cost of additional MACs). Here, a parallel array of 128 MACs performs the multiplication. The accumulated outputs are then serialised and passed to the final FC layer, which consists of 8 parallel MACs. This final stage produces a classification for one of the eight modulation schemes: QPSK, BPSK, QAM16, QAM64, PAM4, 8PSK, GFSK, or CPFSK.

As discussed in Section 3.4.2, this CNN accelerator architecture uses a hybrid data reuse strategy. It combines weight stationary and output stationary techniques to reduce memory bandwidth by keeping weights local to the MACs and accumulating partial sums close to where they are computed. The system also leverages a form of input stationarity by distributing input samples across multiple MACs, each using a different set of weights.

## 4.8 Integration with Embedded FPGA Device

Having completed the process of converting the trained CNN to the custom streaming architecture in HDL Coder, the next step is to integrate it into an embedded FPGA environment for real-time execution. This section aims to provide insights into the deployment of the streaming-based CNN accelerator IP using MathWorks tools and AMD's Vivado Design Suite [109].

The AI accelerator design was built using MathWorks HDL Coder which translates block-based Simulink models into synthesisable VHDL IP Cores

that can be integrated within Vivado's IPI tool. AMD's Vivado Design Suite enables the integration of the AI accelerator with other IP cores, facilitating operations such as data movement, the communication with PS registers, and the distribution of clocks from the PL fabric. Additionally, the PYNQ software framework is required to control and interface with the AI accelerator IP through the resulting Vivado bitstream for visualisation, analysis, and testing purposes. The Vivado IPI hardware design, software drivers, and testing APIs will be explored in the following subsections. Figure 4.23 shows a flow chart of the integration with the embedded device process.

While this work focuses on the AMD RFSoC platform as the target device, the streaming CNN architecture itself is platform-agnostic. Since the accelerator is designed as a standalone IP core, it can be integrated into any FPGA-based SDR platform, such as Zynq devices with AD-FMC front-ends [110] or similar platforms, provided the FPGA fabric can meet the required input data rate.

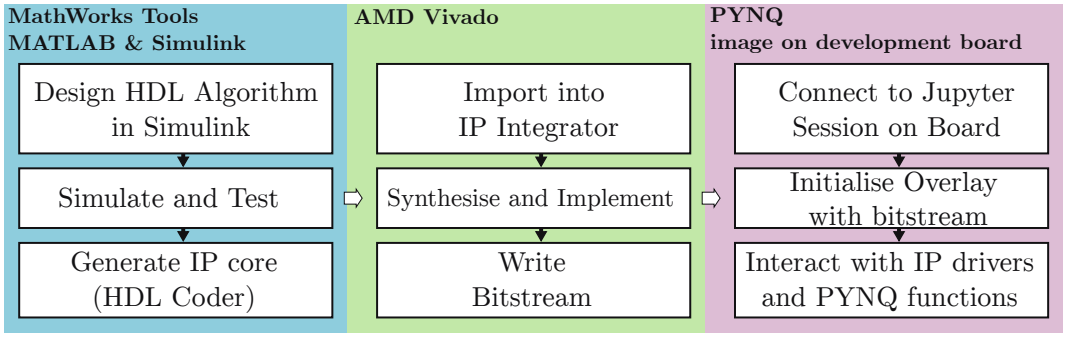



Figure 4.23 Flow chart showing process of integrating custom CNN architecture with the development platform.

### 4.8.1 IP Core Generation

Using the MathWorks HDL Coder Workflow Advisor, an IP core can be generated from a Simulink model [101]. The generated IP core is shareable and reusable, and once generated, can be integrated within a larger design for an embedded system. The HDL Workflow Advisor process allows the user to target a variety of FPGA families and devices, configure a target clock rate that the model can be tested against, and translate Simulink input and output ports into AXI4 compliant interfaces. The top level subsystem of the AI accelerator, showing input and output ports, is displayed in Figure 4.24.

HDL Workflow Advisor was configured to target the AMD Zynq UltraScale+ RFSoC device family and to generate the IP core based on the available

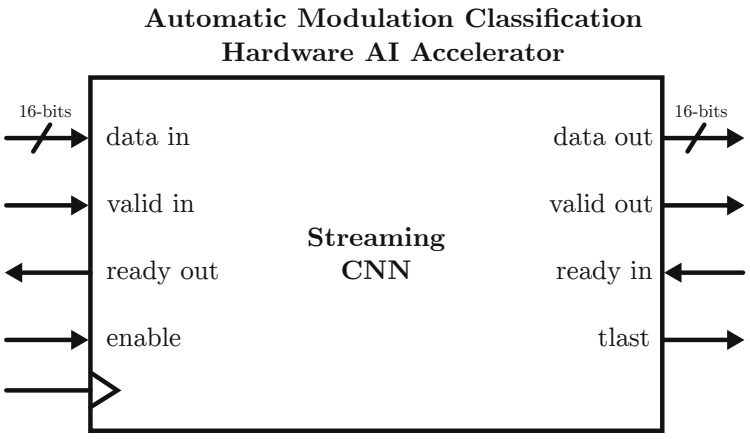



Figure 4.24 Top level AI accelerator IP showing input and output ports.

components within the target family. The IP can be used on any device that contains the same resources at the RFSoC device family. The input and output ports of the Simulink subsystem were configured for translation into AXI4 compliant interfaces. Control signals, such as `enable`, were configured to use the AXI4-Lite interface and a register map offset of **0x100**. The offset defines where in the AXI4-Lite address space the control register is located, allowing software running on the PS to interface with and manipulate the custom IP core through memory-mapped I/O. Assigning a unique offset prevents address collisions with other control registers and enables consistent access from software drivers, used by the PYNQ framework.

The data streams entering and exiting the IP core were configured to use the AXI4-Stream interface and it is primarily used for data transfer between components in a SoC architecture and is particularly useful in high throughput applications such as video processing, signal processing, and communications systems. It provides a unidirectional flow of control data for one-way transfers. AXI4-Stream is a bus interface consisting of several signals that assist with the flow of data through the embedded platform. A further explanation of the AXI4-Stream protocol is found in Section 2.1.6.

In this IP core the signals combined into an AXI4-Stream interface are displayed in Table 4.4.

HDL Workflow Advisor generates the HDL code from the Simulink design and assigns the ports to the associated AXI4 interfaces, resulting in an IP core as seen in Figure 4.25.

Table 4.4 AXI4-Stream signal assignments for AI accelerator IP core.

| Port Name | Direction | AXI4-Stream Signal Assignment | AXI4 Interface |
|---|---|---|---|
| dataIn | In | S_AXIS_TDATA | AXI4-Stream Slave |
| validIn | In | S_AXIS_TVALID | |
| readyOut | Out | S_AXIS_TREADY | |
| dataOut | Out | M_AXIS_TDATA | AXI4-Stream Master |
| validOut | Out | M_AXIS_TVALID | |
| readyIn | In | M_AXIS_TREADY | |
| tlastOut | Out | M_AXIS_TLAST | |

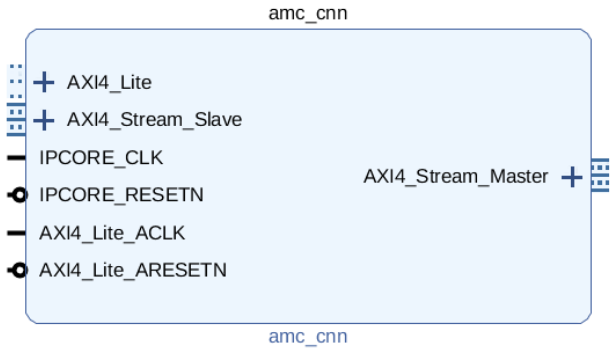



Figure 4.25 IP core generated through HDL Coder.

### 4.8.2 Integration with Vivado IP Integrator

Vivado IPI is a tool within the AMD Vivado Design Suite that allows users to create and manage complex designs using a graphical interface. It simplifies the process of integrating various IP cores and custom components into a cohesive system for AMD FPGAs and SoCs. To integrate the generated AI accelerator IP core as part of an embedded system to be programmed onto a Zynq UltraScale+ RFSoC device, the IP core is imported into Vivado IPI and connected with other components of the embedded platform. Figure 4.26 shows the Vivado IPI block design for the integration of the IP core within the embedded system.

Each numbered region in Figure 4.26 is detailed below:

1. The Zynq UltraScale+ MPSoC IP core provides interfaces from the FPGA to the PS through the `M_AXI_HPM0_FPD` and `S_AXI_HP0_FPD` ports. These two ports allow for the Direct Memory Access (DMA) IP cores to transfer data to and from the PS DDR memory and also allow for AXI4-Lite

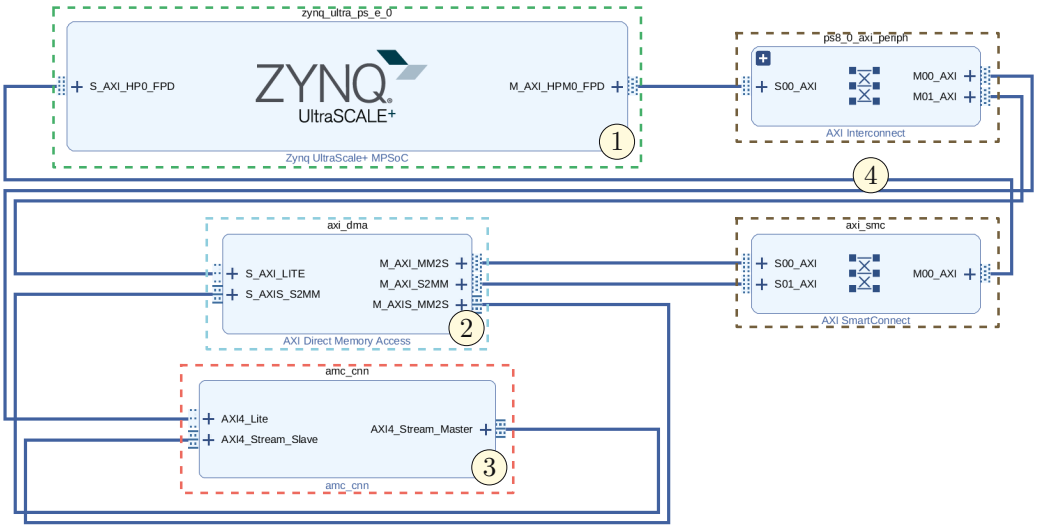



Figure 4.26 The IP Integrator block design for testing the AMC accelerator IP.

registers, within the PL, to be configured from the software operating on the PS.

2. The AXI DMA IP core is responsible for transferring bursts of data to and from the PS DDR memory. In this design, the DMA provides input feature maps to the CNN IP core and retrieves the associated classification, transferring the answer to the software on the PS.

3. The CNN IP core is the generated IP core from MathWorks HDL Coder. It is connected to the Zynq UltraScale+ MPSoC to enable AXI4-Lite register communications, and the DMA to receive inputs and send classifications. The DMA transfers the input feature map to the CNN IP, while the classification output is produced sample-by-sample. A TLAST signal is triggered when the transfer is complete. Figure 4.27 shows a timing diagram for the signals entering and exiting the CNN accelerator.

4. The AXI Interconnect manages the concurrent AXI4-Lite communications from the DMA and CNN IP core, while the AXI SmartConnect IP core manages the data transfers to and from the PS DDR memory.

The Vivado IPI block design is converted into a bitstream, which is then transferred to the Zynq UltraScale+ device for deployment. The bitstream is loaded and interacted with from the PS using the PYNQ framework.

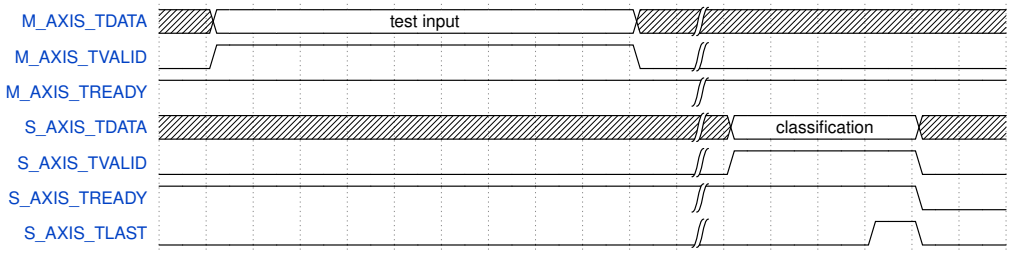



Figure 4.27 Timing diagram of AXI4-Stream signals in the CNN accelerator IP.

### 4.8.3 Control and Visualisation using PYNQ

The CNN accelerator bitstream requires the PS for control, interfacing and visualisation. The SD card located on the development board contains the PYNQ 3.0.1 image which comprises a Linux-based OS with additional functionalities for interacting with AXI connected IPs within the PL [51]. Python classes are created for the combined operation of the AXI DMA with the CNN accelerator and the visualisation widget.

The core driver files, `overlay.py` and `dma.py` host the classes *Overlay*, *DMA*, and *DefaultIP*, which enable basic interactions from PYNQ to the PL. The custom driver class, *AMCCNN*, combines the functionality of the AXI DMA and the CNN accelerator IP to create an inference testing system. The driver allows the user to send test inputs to the classifier and retrieve classifications in return. The *AMCWidget* class further expands the capabilities of the *AMCCNN* class by presenting a front-end environment powered by `ipywidgets` and `plotly`. A Unified Modelling Language (UML) diagram of the CNN accelerator control system can be seen in Figure 4.28.

#### Interfacing with AMCCNN

Interfacing with the modulation classification CNN IP core is performed through a Python interpreter provided by the PYNQ framework. The bitstream generated by Vivado is loaded onto the PL through the *Overlay* class, which exposes the AXI4-Lite connected IPs. The CNN and DMA IPs are assigned to new variables and passed into the initialisation conditions of the *AMCCNN* class, as shown in Listing 4.5.

The AMCCNN class sets up the PL IPs by applying configurations to AXI4-Lite registers and making available the functions illustrated in Figure 4.28. The `predict()` function performs a single frame query to the CNN and returns the classification result based on the input provided. In Listing 4.6, a test frame of RadioML data, `x`, is passed into the `predict()` function where

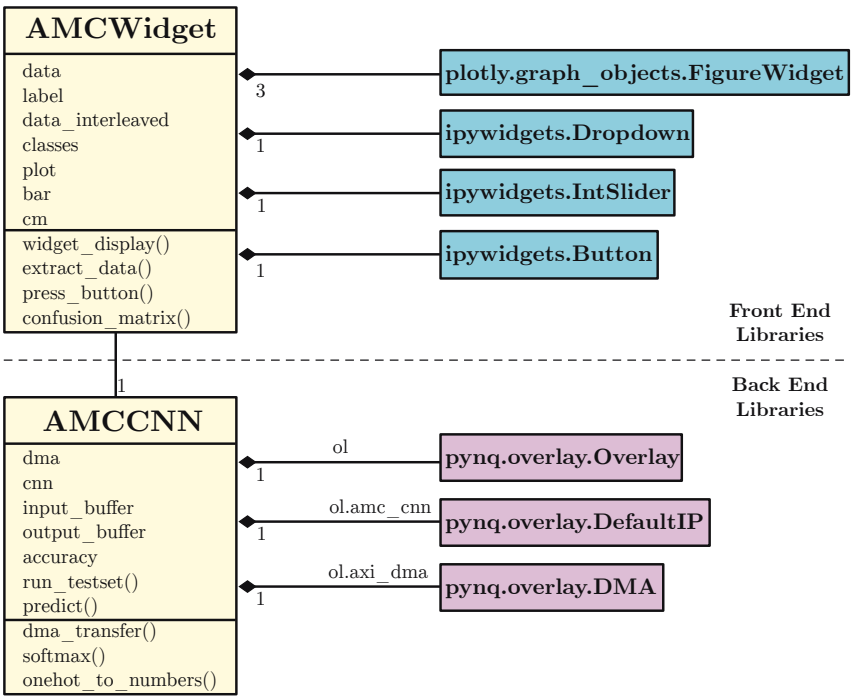



Figure 4.28 A UML diagram displaying the relationships between software drivers for controlling the bitstream.

Listing 4.5 Loading bitstream onto PL through PYNQ and passing IPs to AMCNN class.

```
from pynq import Overlay # import Overlay class
ol = Overlay("amc_cnn_bitstream.bit") # Load bitstream to the PL
dma = ol.axi_dma_0 # Assign DMA
cnn = ol.amc_cnn_0 # Assign CNN
amc = AMCCNN(dma, cnn) # Initialise AMCCNN by passing IPs to class.
```

the CNN produces a classification output, **`y_pred`**. This output is compared to the true label, **`y_true`**, for the test frame to verify if the CNN classified the input data correctly.

Listing 4.6 Predict the modulation scheme of a test frame **`x`**.

```
y_pred = amc.predict(x) # Send x to the CNN via DMA transfer, return CNN
    classification.
if y_pred == y_true: # Compare prediction with true label
    print('Correct') # Print 'Correct' if they match.
else:
    print('Incorrect') # Otherwise, print 'Incorrect'
```

The **`predict()`** function abstracts the configuration of the DMA IP as well as the AXI4-Lite enabling of the CNN IP in hardware. The CNN IP is enabled by writing a **`True`** to the AXI4-Lite register at a given offset with **`cnn.write(0x100, True)`**. To send the test frame to the CNN IP through a DMA transfer, a contiguous portion of memory is allocated through the

`pynq.allocate()` function. The DMA then performs a transfer by triggering `dma.sendchannel.transfer(data)` and then waiting for the DMA to finish the transfer with `dma.sendchannel.wait()`. This operation is repeated when receiving the classifications from the CNN IP back into the PS on `dma.recvchannel`.

The *AMCCNN* class is expanded by the *AMCWidget* class that provides an interactive version of the AMCCNN function seen in Listing 4.5. Figure 4.29 shows a screenshot of the *AMCWidget* class in operation where the input frame is plotted and the output prediction of the CNN is presented in two ways. One through a confusion matrix to compare the predicted value with the true value and the confidence of the classifier prediction. A dropdown widget allows the user to select the modulation scheme of the test frame and the 'Update' button triggers the DMA transfer to the CNN IP.

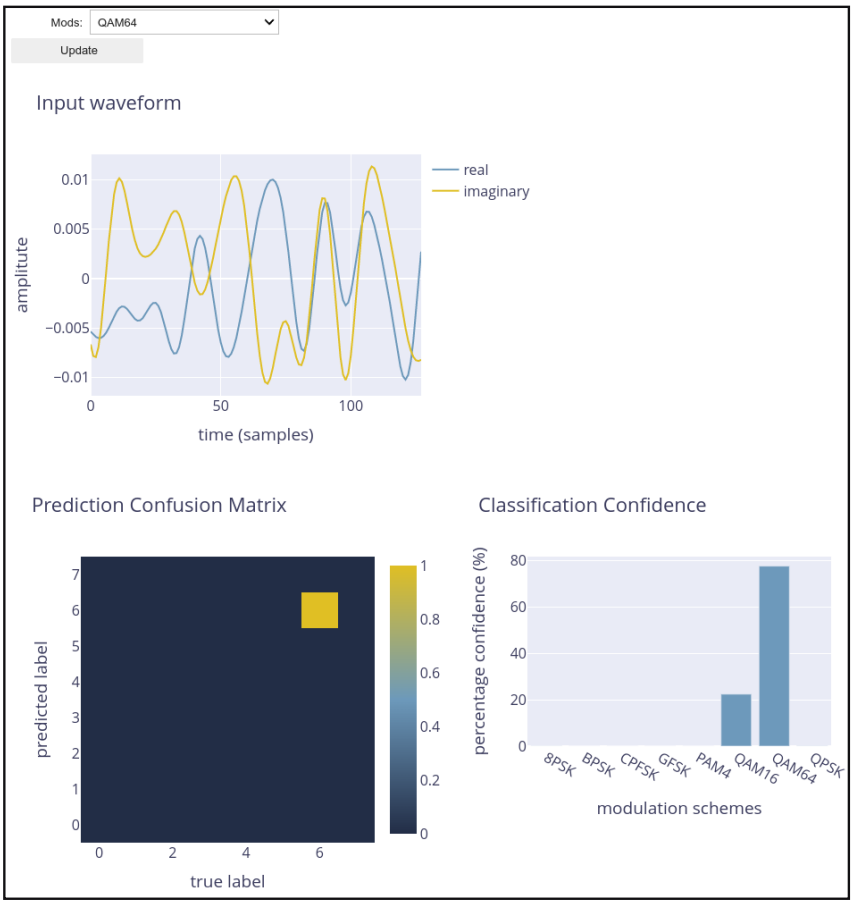



Figure 4.29 *AMCWidget* class showing interactive `ipywidgets` to control the *AMCCNN* class.

## 4.9 Architecture Evaluation with RadioML

In this section, the performance of the modulation classification CNN IP is evaluated using the RadioML dataset and results are provided for overall accuracy, per modulation scheme accuracy, and FPGA implementation utilisation. These results provide insight into the effectiveness of the architecture in identifying modulation schemes and evaluating the feasibility of the proposed streaming-based CNN accelerator. The following subsections present the overall accuracy and per-scheme accuracy of the implemented model when tested with the RadioML dataset.

### 4.9.1 Accuracy of Deployed Model

Through the abstracted class, *AMCCNN*, and associated functions the accuracy of the deployed network can be evaluated. A test set of 32,000 frames was sent to the CNN IP and the prediction of each frame was compared with its true label. The test set consists of frames with each 128 complex valued samples and modulated to one of the following modulation schemes: QPSK, BPSK, QAM16, QAM64, 8PSK, PAM4, GFSK, and CPFSK. The RadioML frames have been passed through a multipath channel with AWGN from $-20 \rightarrow 18$ dB SNR.

The overall accuracy of the deployed CNN model is calculated by sending the RadioML test set frames to the deployed model on the FPGA. The overall accuracy of the CNN model performing AMC is shown in Figure 4.30.

The results in Figure 4.30 compare the accuracy of the deployed model against the accuracy reported from the model training stage in Section 4.5.2. The accuracies of the two test scenarios on the same RadioML test set indicate that the CNN architecture accurately represents the model for performing AMC and achieves equal accuracy, despite the quantisation of the weights and activations. This is an expected and interesting result since the model is using 16-bit fixed-point values which can comfortably represent the learned floating-point values. It shows that although the system was trained in floating-point, the full precision and dynamic range of floating-point is not required to accurately represent the model.

Further results are reported in Figures 4.31 and 4.32. These show the per-class accuracies across SNR levels, and a confusion matrix of the classification for an example SNR of 18 dB, respectively.

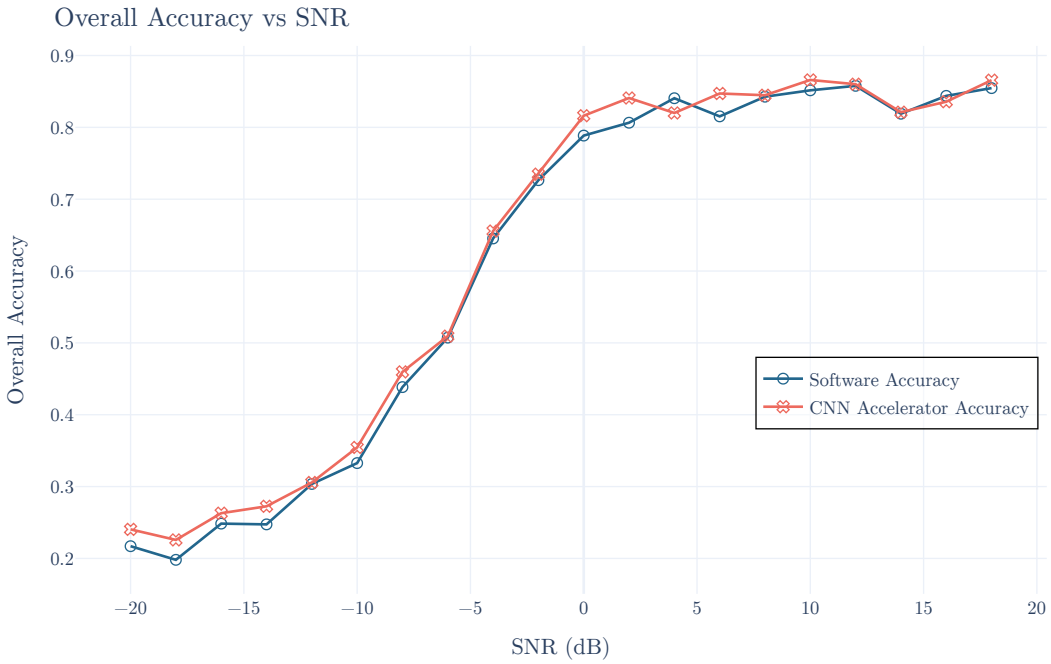



Figure 4.30 Overall accuracy of the CNN accelerator vs the overall accuracy recorded from the software model with the RadioML test set.

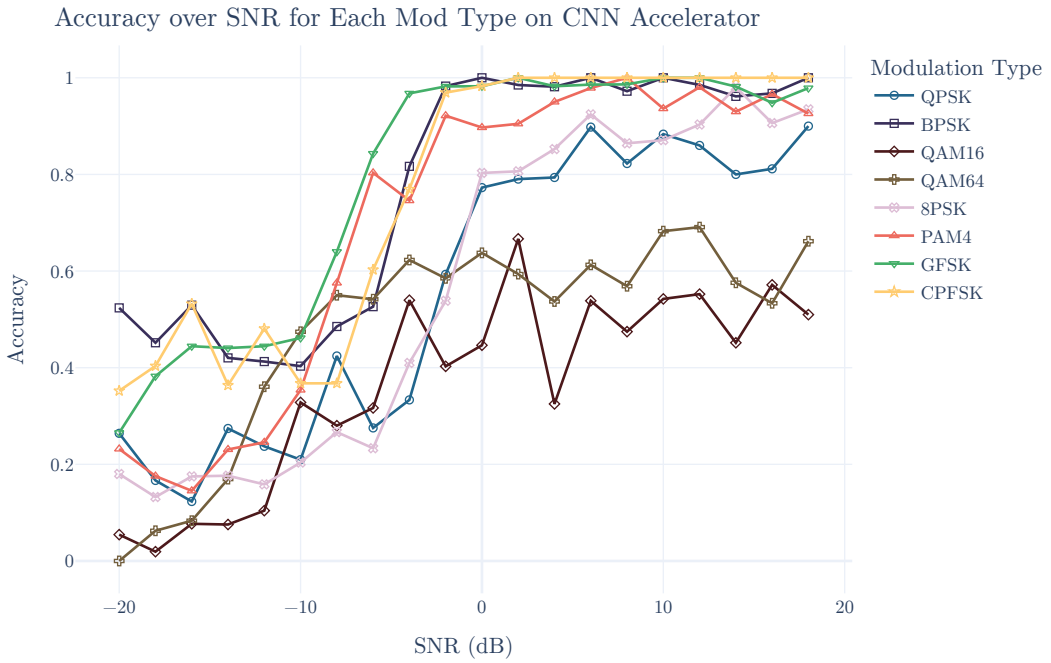



Figure 4.31 The per-class accuracy of the CNN accelerator across SNR values on the RadioML test set.

These results show that overall good classification accuracy is reported for all modulation schemes and that, as observed in Section 4.5.2, the higher bit-rate modulation schemes are the most difficult to correctly identify. Additionally,

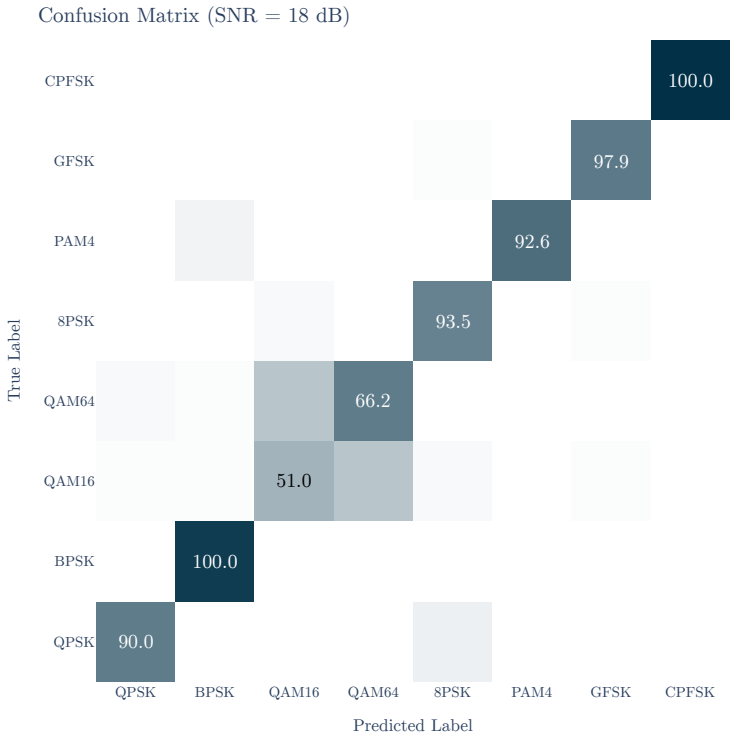



Figure 4.32 Confusion matrix of predictions vs true labels from the CNN accelerator on the RadioML test set at SNR=18 dB.

due to the similarity in the two modulation schemes, QAM16 and QAM64 are commonly mistaken for one another as seen in Figure 4.32.

### 4.9.2 Implementation Results

The system was evaluated on the AMD RFSoC 2x2 development board which hosts the Zynq UltraScale+ XCZU28DR RFSoC part. The RFSoC's PL includes 4,272 DSP slices, 1,080 BRAMs, and 80 URAMs. For the purposes of this evaluation, the RFSoC was chosen to determine whether the CNN IP could operate with its PL through DMA transfers for transferring data to the input of the model. Verifying this functionality enables the next stage of development, which will integrate the RFSoC's RF-ADCs.

#### Resource Utilisation

Table 4.5 shows the resource utilisation for the deployed CNN model in the PL. The model uses 10.67% of the available DSP slices on the chip and 15.09% of the available BRAM. A breakdown of the resource utilisation for each layer is also tabulated. The DSP slices perform the task of implementing the MAC function used in each layer. For the first convolutional layer, 64 DSP slices were

used to implement the MAC portion of the layer, which equals the number of filters ($N$) present in the layer as discussed in Section 4.3.2. Convolution layer 2 represents an implementation of multiple MVMs for a multi-channel input. In this case, 16 groups of 16 MACs were used, equalling the 256 DSP slices implemented on the device. The latter two layers of the network are output stationary and perform the MVM with MACs unrolled along the output dimensions of 128 and 8, respectively. This matches the DSP slice utilisation seen in Table 4.5.

The BRAMs and URAMs used by the CNN model facilitate the buffer of input samples entering each layer. The singular URAM used by the first convolutional layer stores the input to the model before the SWC can produce the GEMM transformation for the layer. For subsequent layers, BRAMs are used to store the inter-layer signals. Look-Up Tables (LUTs) and registers are used to store on-chip weights and perform control calculations.

Table 4.5 FPGA resource utilisation of the deployed CNN IP.

| **Model** *layer* | **Slice LUTs** | **Slice Register** | **DSPs** | **BRAMs** | **URAMs** |
|---|---|---|---|---|---|
| **CNN** | **23,721 (5.59%)** | **45,894 (5.39%)** | **456 (10.67%)** | **163 (15.09%)** | **1 (1.25%)** |
| *conv1* | 4,247 | 4,999 | 64 | 0 | 1 |
| *conv2* | 7,781 | 17,730 | 256 | 32 | 0 |
| *fc1* | 10,688 | 22,110 | 128 | 130 | 0 |
| *fc2* | 749 | 782 | 8 | 0 | 0 |

### Latency and Throughput

The model was implemented in Vivado 2020.1 and tasked with achieving a clock rate performance of 100 MHz. The deployed model achieved a maximum clock rate of 355 MHz, thus exceeding the target clock rate. At a clock rate of 100 MHz, the throughput of the model achives a 26.5k classifications per second (cps) and at its maximum achievable clock rate, 94.075k cps. The latency of the model at 100 MHz clock rate is 37.9$\mu s$ and 10.68$\mu s$ for the maximum clock rate of 355 MHz. Table 4.6 shows these reported results.

Table 4.6 Clock rate, latency, and throughput results for the deployed CNN IP.

| **Clock Rate (MHz)** | **Throughput (cps)** | **Latency ($\mu$s)** |
|---|---|---|
| 100 | 26.5k | 37.9 |
| 355 | 94.075k | 10.68 |

## 4.10 Chapter Conclusion

In PHY wireless communications, real-world deployments are increasingly realised on SDR platforms such as the RFSoC. While there are significant advancements in the field of AI for RF, transferring the algorithmic improvements to a deployable scenario is far from trivial. Other AI accelerator technologies do not take into account the unique requirement in communications systems to be able to process every sample received into the model.

This chapter has introduced a custom streaming-based CNN accelerator built specifically to process a constantly moving stream of samples. Other works [39]–[41], which focus high-throughput accelerators without a guarantee to process samples at a given input rate, this work presents an architecture that is specifically designed for high-throughput real-time RF data classification that processes every sample it receives. The target application of modulation classification was chosen to demonstrate the efficacy of this design method. A CNN model was trained on the RadioML [45] dataset and ported to the CNN accelerator architecture detailed in this chapter.

Each layer's design was covered, showing how a streaming set of samples can be processed by each of the dataflow layers, showing that all layers can concurrently operate and stream samples from one layer to the next. Through the introduction of an alternative GeMM transform, processing a streaming signal going into the model was shown to be possible.

This chapter has presented an analysis of how the weights and activations were quantised, as well as the process of designing the architecture from basic principles. The streaming-based AI accelerator architecture is developed in MATLAB/Simulink, which allows it to take full advantage of the software's simulation capabilities. As a result, engineers can integrate the CNN model with other simulation packages, such as the Communications Toolbox and the 5G Toolbox in MATLAB and Simulink [23], [111], for system-wide evaluation. This added benefit can assist with confirming the operation of an AI-based communications application before it is implemented in hardware.

The key takeaway from this chapter is that it is possible to design a streaming, real-time CNN accelerator architecture for modulation classification that achieves software-equivalent accuracy while consuming a low number of FPGA resources. The resulting deployed architecture showed that it achieved equal classification accuracy to the same model trained in software, confirming the correct computational operation of the proposed architecture. The resource utilisation of the deployed model showed acceptably low occupancy of the RFSoC's PL, occupying less that 6% of all logic fabric resources and approximately 11% of DSP slices and 15% BRAM, respectively. This allows space for other designs in a radio pipeline to fit alongside the CNN architecture. The architecture showed promising latency and throughput metrics, further supporting the design's remit as a real-time inference architecture.

While a specific CNN model deployment is reported, the architectural designs presented in this chapter can be applied to any size CNN. The number of resources a resulting CNN consumes is proportional to the number of weights in each layer (to be stored in on-chip RAM) and the input size into each layer, which determines the number of buffers required to produce a GeMM transformed input.

The next chapter explores the design choices and training requirements for deploying the CNN architecture to receive live data from the RF-ADC and operating in real-time, by removing the DMA input to the model and replacing it with a radio receiver pipeline.

# Chapter 5

# Real-time CNN Integration with Radio Receiver

This chapter details the process of validating the streaming-based CNN accelerator in a real-time radio receiver system. It outlines the design and deployment of a custom dataset, made to reflect live signal conditions captured through the RFSoC hardware, and demonstrates the full integration of the accelerator with a live signal. The chapter reports on accuracy performance with AMC and throughput and latency comparisons with other works.

## 5.1 Motivation

A key goal of the work presented in this thesis is to demonstrate a real-time CNN operating with a SDR receiving real-time data live. While much of the previous development and testing has relied on the RadioML dataset, transmitting this dataset through an RF-DAC and receiving the signal again through the RF-ADC is infeasible due to the format the dataset is stored in. Instead, this chapter focuses on improving the CNN architecture to operate successfully on received radio samples, while also demonstrating a method for the development of a custom dataset that represents the samples coming from an RF-ADC receiver stage.

Creating this dataset presents a number of challenges. Ideally, the CNN should be trained on signals that capture realistic channel and hardware effects, but doing so should not require physically building and configuring a dedicated transmitter for every modulation scheme. To address this, a hybrid dataset generation approach is adopted. Modulation signals are synthetically generated and then transmitted and received through the actual RF hardware in a

loopback configuration. This approach allows synthetic signals to be embedded with real-world hardware distortions, including the RF-ADC's digitisation artefacts, non-linearities, and tile stitching effects inherent to the RFSoC's RFDC system.

This methodology provides two key advantages. Firstly, the resulting dataset preserves the controlled channel conditions of synthetic data while incorporating the physical impairments introduced by the RF front-end. Secondly, it enables the receiver architecture to be tested under live conditions, while receiving sampled data from the RF-ADC, applying decimation and baseband mixing, and performing real-time classification using the CNN accelerator. Reliable operation under these conditions validates the accelerator as a functional and deployable component to be used in an intelligent radio system.

## 5.2 Related Work

Several works have investigated the deployment of CNN models on embedded platforms for modulation classification. In particular, related implementations targeting FPGA platforms aim to maximise inference throughput while minimising latency. Many of these efforts rely on either HLS tools or custom hardware architectures, and typically evaluate performance using the RadioML [45] dataset or similarly constructed synthetic datasets. However few have attempted real-time inference using live RF signals, and fewer still demonstrate full integration with the RF signal path of an SDR platform like the RFSoC.

To benchmark the performance of this work's deployed CNN accelerator, the comparison is made against other relevant FPGA-based CNN accelerators developed for modulation classification, including [42], [43], [46], [47]. These works differ in architecture, quantisation, dataset usage, and evaluation setup. The primary focus of this chapter is on evaluating how well a deployed model, trained on a realistic dataset, can operate in real-time on live signals.

The second aspect of related work concerns how datasets for modulation classification are constructed for training neural networks. Most existing approaches fall into two categories. The first is purely synthetic: signals are generated in software with simulated channel models such as AWGN or multipath fading, often using tools like GNU Radio [24] or MATLAB. These signals are typically normalised and presented as framed data to the network. Such examples include: [45], [112]–[114]. The second approach involves

capturing signals over the air, where modulation schemes are transmitted and received using SDR hardware, and the dataset reflects the real-world conditions of environment during capture. As a result these datasets typically record indoor channel scenarios, leading to lack of diverse channel models in the recorded data. Such examples include: [72], [115], [116]. The work in [72] first trained models on synthetic data and then fine-tuned them on over-the-air data. They reported a 16% accuracy drop when models trained solely on synthetic data were tested on over-the-air signals, while fine-tuning on over-the-air data recovered 10% of that loss.

The dataset methodology proposed in this work presents a hybrid approach. Signals are synthetically generated and channel-affected in simulation, but are then transmitted through RF hardware and re-captured. This approach bridges the gap between simulation and physical deployment.

## 5.3 Deep Learning Challenges on the RFSoC

In Chapter 4, the RadioML dataset [45] was used as a benchmark to develop and evaluate the CNN streaming accelerator architecture, demonstrating its suitability for wireless communication applications. Once the architecture was shown to operate correctly on synthetic data and successfully classify modulation schemes, the next step was to integrate the AI accelerator with a radio system receiving live signals. While synthetic benchmarks validate functional correctness, evaluating the architecture in a real-time radio environment provides a stronger indication of its practical viability and effectiveness for deployment.

### 5.3.1 Processing a Stream of Infinite Samples

Several differences arise when transitioning from a synthetic classifier scenario to a real-time and real-life data scenario. These can prove challenging when modelling a real-time AI accelerator operating on the RFSoC. One such difference is the constant stream of data samples received by the SDR RF-ADC, which could potentially be infinite instead of finite frames of data. The introduction of a constant stream of samples adds an additional challenge to the AI accelerator architecture. If the accelerator processes samples slower than they are received, the accelerator will only have the ability to process a subset of samples received by the radio. Depending on the application in which the radio AI receiver is operating, it may be acceptable to process a subset

of samples. These applications may include non-critical applications as part of a larger signal detection algorithm where data is sent asynchronously to the DL model. However, in sample-critical receiver applications such as signal decoding, spectrum monitoring, preamble detection, and channel estimation and correction, dropping samples or not processing every sample provided by the receiver can lead to a degradation of receiver performance. For a CNN accelerator to work effectively in all radio receiver scenarios, it should have the capability to process every available sample. Knowledge of the input sample rate $f_s$ for the accelerator, the dimensions of the DL model deployed, and the number of resources on chip (DSP slices, BRAMs, etc) is useful for achieving an accelerator that does not drop samples. Figure 5.1 illustrates the real-time receiver scenario when receiving a constant stream of samples.

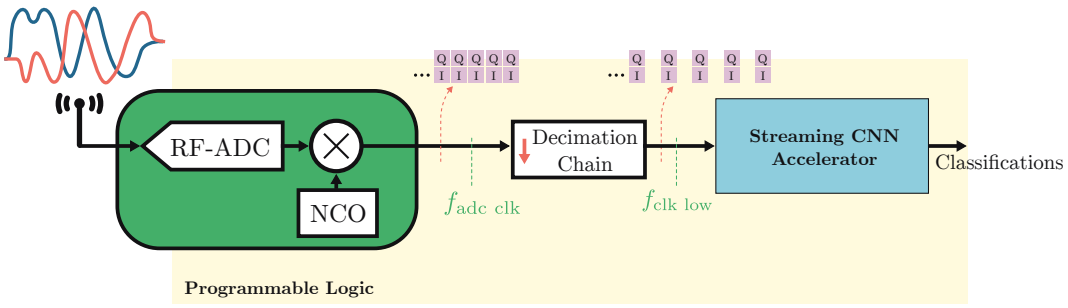



Figure 5.1 A radio receiver with a constant stream of samples entering an AI accelerator.

The architecture introduced in Chapter 4 follows the dataflow design philosophy where all layers of the DL model exist physically on-chip and data samples flow through the layers. It is important to note that this dataflow design philosophy treats the incoming data as a continuous stream of infinite samples and processes them in real-time. While the model aims to support an infinite stream of data samples entering the model, the model's Window of Focus (WoF) is still finite, depending on the parameters of the model deployed. Thus, supporting a stream of infinite samples is only an implementational design to facilitate a real-time accelerator.

A significant challenge with processing a stream of samples through the proposed dataflow model is the repeated reading of input feature map samples by the convolutional layers, where a loss of samples may occur due to the over-production of samples by the convolutional layer.

In Section 3.3.3, the GeMM transform was introduced to simplify the complexity of the 3-dimensional convolutional layers. The layers were reduced to a matrix multiplication at the cost of replication of input samples. This

replication of input samples can hinder the maximum available input data rate of the CNN accelerator. Given an input feature map with dimensions including channels $c$, height $h$, and width $w$, a convolution operation is defined by a filter kernel of height $j$ and width $k$, the number of total strides $s$ used during a convolution is defined as

$$s = (h - j + 1) * (w - k + 1). \tag{5.1}$$

The over-production of samples each convolutional layer contributes to the model can be calculated as the number of samples entering the layer and the corresponding number of samples exiting the layer. This ratio, $R_{\text{conv}}$, is calculated as shown in Equation 5.2.

$$R_{conv} = \left\lceil \frac{scjk}{chw} \right\rceil. \tag{5.2}$$

The value, $R_{\text{conv}}$, represents the ratio of output samples for a given convolutional layer. It quantifies how many output samples are produced per input sample. If $R_{\text{conv}} > 1$ then the layer produces more output samples than input samples received, which effectively limits how frequently new inputs can be accepted. To facilitate a constantly streaming convolutional layer, the AI accelerator clock rate must increase by a factor equal to the sum of all convolutional layer ratios $R_{\text{conv}}$,

$$R_{\text{conv\_total}} = \sum_{i=0}^{L_c - 1} R_{\text{conv}(i)}. \tag{5.3}$$

Equation 5.3 is the result of empirical derivation and does not appear in prior literature.

### 5.3.2 Calculating Model Clock Rates

Equations 5.2 and 5.3 can be applied to the model introduced in Section 4.4. In the designed network from Chapter 4, the first convolutional layer processes input WoF with dimensions $(c, h, w) = (1, 2, 128)$ and filter dimensions $(n, c, j, k) = (64, 1, 1, 3)$. The clock rate increase factor for this layer is denoted as $R_{\text{conv1}}$, i.e.

$$R_{\text{conv1}} = \left\lceil \frac{scjk}{chw} \right\rceil = \left\lceil \frac{252 * 1 * 1 * 3}{1 * 2 * 128} \right\rceil = \left\lceil \frac{756}{256} \right\rceil = 3. \tag{5.4}$$

The relationship in (5.2) and (5.3) can be extended and applied to all convolutional layers in the network topology. In the case of the CNN model proposed earlier, two convolutional layers exist. Additionally, an extra factor of 2 is added due to the interleaving stage before data enters the implemented CNN accelerator model. The overall upsampling ratio required for the network, assuming that a fully unrolled parallel MVM is implemented, is given by

$$R_{\text{total}} = R_{\text{conv1}} + R_{\text{conv2}} + 2 \tag{5.5}$$

where $R_{\text{conv1}}$ and $R_{\text{conv2}}$ are both equal to 3. This equation shows that the receiver system of the current network topology necessitates a minimum clock rate increase by a factor of 8, while assuming a fully unrolled parallel matrix-vector multiplier.

In the implementation described in Chapter 4, a partially unrolled MVM is implemented for each convolutional layer and FC layer, in order to save on PL resources. As a result, the clock rate is increased by a factor of 32 instead to enable unrolling.

### 5.3.3 Signal Data Path

Real-world signal environments add significant complexity to the training and inference phases of a deployed DL model. While synthetic datasets are useful for initial development and testing, they do not capture the nuances of a real-life transmission channel. These channels introduce a range of impairments, such as interleaving spurs, harmonics, power variations, clock offsets, and non-ideal frequency responses from the decimation and interpolation filters, where a perfect 'brick-wall' response cannot be achieved. A robust system must account for these imperfections to maintain a high classification accuracy under real-world conditions.

## 5.4 Transceiver-Based Dataset Construction on RFSoC - DeepRFSoC

As detailed in Section 5.3.3, real-world signal paths introduce hardware and channel impairments that synthetic datasets often fail to capture. To address this mismatch and ensure robustness under deployment conditions, a custom dataset, 'DeepRFSoC', was generated using the RFSoC platform [117].

In the generation process, synthetically generated signals were transmitted and received in loopback using the onboard RF-ADCs and RF-DACs, allowing the dataset to embed the exact non-idealities of the RFSoC signal chain. By generating training data within the target hardware environment, the resulting CNN model is exposed to the same conditions it would encounter during live inference.

### 5.4.1 Generation of Training Samples in MATLAB

The first step in creating a custom dataset is to generate a set of examples of modulated signals simulated through environmental channel effects using MATLAB and MATLAB's Communications Toolbox [23].

The MATLAB Communications Toolbox is an add-on package for MATLAB that provides tools for designing and simulating communications systems ranging from simple modulation schemes to OFDM, WiFi, LTE, and 5G systems [23]. It is widely used in wireless communications research and development. The toolbox includes a comprehensive set of algorithms and functions for tasks such as signal processing, channel modelling, error correction coding, and performance evaluation. For this research the Communications Toolbox was used to generate the modulated waveforms for each modulation type which were then 'transmitted' through a simulated channel. This work used MATLAB version 2020a.

The generation of the training samples, through MATLAB, aimed to follow a similar paradigm to the methodology detailed in RadioML [45], where different channel model parameters, frame sizes, and dataset sizes were configured. The dataset consisted of eight modulation schemes where each modulation scheme was affected by 14 AWGN values ranging from -20dB to 30dB SNR. The generation of the dataset followed the following steps (see Appendix B for a full code listing of the MATLAB generation process):

1. Generate 1024 uniformly random symbols for each modulation scheme.
2. Apply pulse shaping using a Raised Cosine filter with roll-off factor of $\beta = 0.5$, filter length of 10 symbols, and 8 SPS, resulting in 8,192 complex samples per frame.
3. Pass the signal through a Rician multipath channel using MATLAB's `comm.RicianCHannel()` function [118] configured with:
    - A sampling rate of $f_s = 128$ MHz,

- Three path delays: $\left[0, \frac{1.8}{f_s}, \frac{3.4}{f_s}\right]$,
- Average path gains: [0, -2, -10] dB,
- K-factor of 4, simulating a mildly fading indoor environment,
- Maximum Doppler shift of 4 Hz to simulate a maximum transmitter velocity of 1.7 m/s, adequately simulating person taking a brisk walk.

4. Introduce a random clock offset in the range of -5 to 5 MHz and compute the resulting frequency shift, while assuming a carrier frequency of 700 MHz.

5. Resample the signal to simulate time drift caused by sampling frequency mismatch via linear interpolation.

6. Add AWGN using SNR values ranging from -20 dB to 30 dB in 14 evenly spaced steps.

7. Crop each generated frame randomly to a fixed length of 4,096 samples.

8. Separate the real and imaginary parts (I and Q) into two channels, resulting in a final frame of tensors of shape $1,000 \times 2 \times 4,096$ per modulation scheme and SNR value.

9. Repeat the entire process for each of the 8 modulation schemes and for all SNR values.

The resulting frames for each modulation scheme and SNR were collected into a dataset ready to be transferred to the RFSoC development board.

### 5.4.2 Transmit and Receive FPGA Radio Design

The next step in creating the custom dataset was to transmit, receive, and record the generated samples through the RFSoC transmit and receive path. Figure 5.2 shows a high-level overview of the process. In this step, the RFSoC's signal path was included as part of the channel the training data passed through. The samples were sent through the RFSoC's RF-DAC transmission chain, including interpolation filtering and mixing, then sent via RF loopback. The samples were received into the RFSoC's RF-ADC, mixed down, passed through decimation filters, and capturing a small frame of this received data was captured.

To facilitate the transmission and reception of the generated signal produced in MATLAB, a PL bitstream was designed to transmit the data.

The data generated from MATLAB in Section 5.4.1 was packetised and accessed on the RFSoC's PYNQ Linux OS through the Python programming language.

As stated earlier, a single frame from the dataset has dimensions $2 \times 4,096$ and, when ready to send, it is allocated in PS DDR memory on the RFSoC development board. The data is then ready for the DMA to move it to the PL. A buffer, in the PL, waits to receive the data from the DDR memory. As the data transfer needs to be continuous, the DMA is configured in cyclic mode. This configuration allows the DMA to repeatedly transfer data from the DMA buffer to the IPs in the PL without interruption, simulating the constant stream of samples that can occur in a real-life wireless communications scenario.

The signal transferred from the DMA is passed through the Digital Up-Converter (DUC) stages and interpolated to a sampling rate of 128 MHz, then modulated to a desired carrier frequency and sent to the RF-DAC and out of the device. The signal is carried along the RF loopback cable and received into the RF-ADC, which is tuned to demodulate the signal from the same carrier frequency as applied at the transmit side.

The received signal is passed to the DDC stages where the sample rate is reduced from 128 MHz to 4 MHz, prior to the signal entering the Frame Capture IP block. The Frame Capture IP awaits an AXI4-Lite register update from the PS to instruct it to capture the current 128 samples entering the IP. Once the 128 sample frame has been captured, it is transferred to PS DDR. The data is then accessible via PYNQ and other Python libraries for visualisation and storage. The full dataset is created by iterating over all modulation schemes, and iterating over each SNR noise value, for each modulation scheme.

### Dataset Creation Block Design

The dataset creation block design was made to facilitate the generation of the modulation classification data by transmitting and receiving modulated signals on the RFSoC development board. The block design integrates RF data converters, FIR filtering stages, and efficient data transfer between the PS and PL on the RFSoC platform.

Figure 5.3 shows a high-level description of the connected IP cores used in the PL block design implemented on the RFSoC.

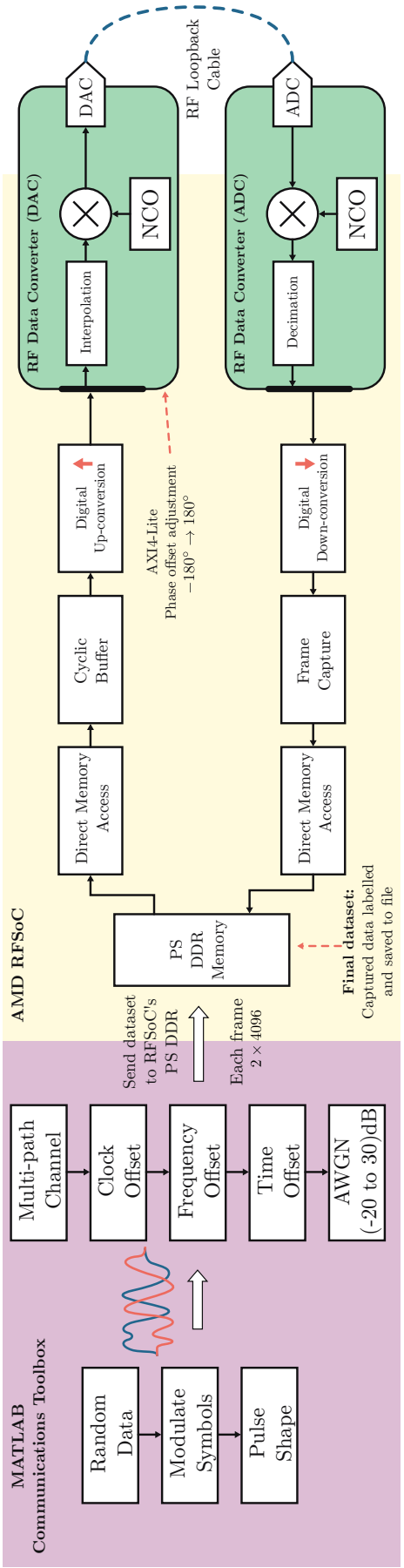



Figure 5.2 Overview of the dataset creation process, showing the transmission of MATLAB generated signals using the RFSoC's loop-back path.

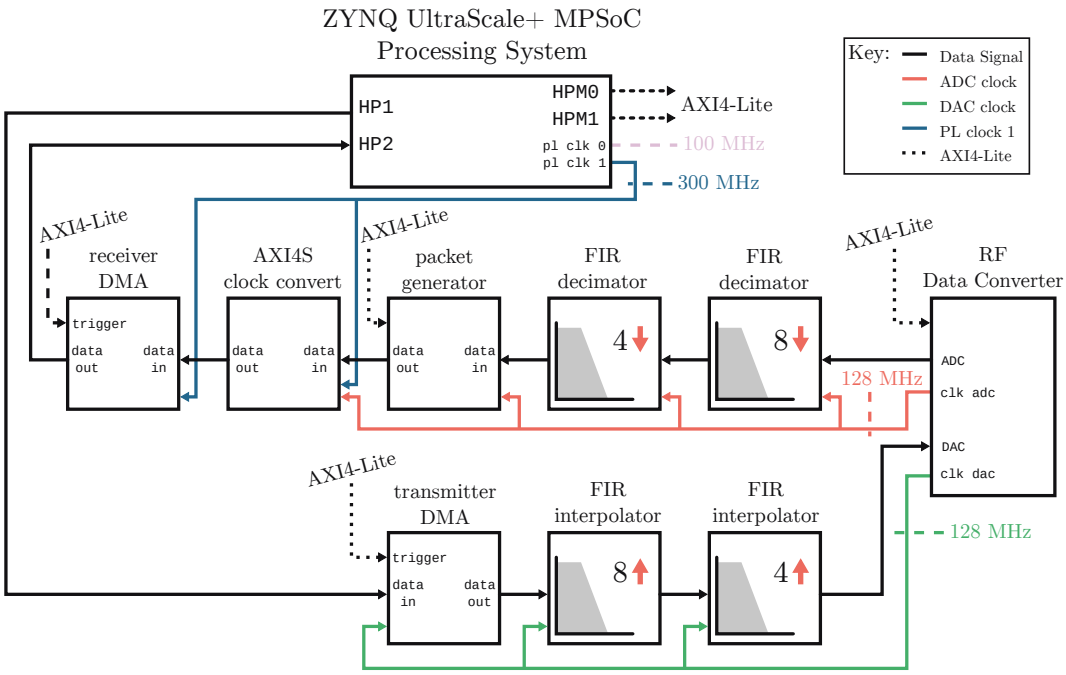



Figure 5.3 PL design for dataset creation bitstream.

The block design facilitates the functionality described in Section 5.4.2. It includes the following IP cores:

- **Zynq UltraScale+ MPSoC**: Used for configuring the PS features on the RFSoC device while also exposing the communication links between PS and PL. (Note: MPSoC and RFSoC PS are equivalent).
- **RF Data Converter**: Configured for data acquisition and transmission through ADCs and DACs and producing the clock signals for operating the transmit and receive paths.
- **FIR Interpolation Filters**: Used for increasing the data signal sampling rate while removing the generated spectral images with a low-pass filter.
- **FIR Decimation Filters**: Used for decreasing the data signal sampling rate, attenuated with low-pass filtering to prevent aliasing.
- **AXI Direct Memory Access (DMA)**: Facilitates high-speed data transfer between the PS and PL.
- **Packet Generator**: Used for generating packets of data by capturing a constant stream of samples.
- **AXIS Clock Converter**: Used to convert an AXI4-Stream signal from one clock domain to another.

**Zynq UltraScale+ MPSoC IP Core** The Zynq PS IP core provides the capabilities for configuring device-wide settings for both the PS and PL of the RFSoC. For this block design, the Zynq IP core was configured to provide two output PL fabric clocks for the DMA and AXI4-Lite IP cores. The first clock, `pl_clk0`, was set to provide a clock rate of 100 MHz, used by the AXI4-Lite interfaces on the IP cores. The second clock, `pl_clk1`, was set to provide a clock rate of 300 MHz, which was used by the AXI DMA IP cores for transferring the data signals to and from the PS. Furthermore, the Zynq PS IP core's PS to PL AXI communications signals were enabled to provide connectivity for AXI4-Lite connections and AXI DMA data transfers. These were the HPM0 and HPM1 AXI interfaces used by the receiver and transmitter AXI4-Lite connect IPs, respectively. The AXI DMAs were provided the HP1 and HP2 AXI interfaces for data transfer to and from the PS from the PL.

**RF Data Converter** The RF Data Converter (RFDC) IP core in Vivado is used to configure and control the hardened RF-ADC and RF-DAC blocks on RFSoC devices. Theses converters provide high-speed data conversion in RF applications, enabling the RFSoC to perform direct RF signal processing from within the PL [58]. For this application, the RFDC has been configured to enable one ADC and one DAC channel, as this configuration meets the requirements for a single transmit/receive system for modulation classification.

The XCZU28DR RFSoC part makes available two ADCs, one on ADC tile 224 and another on tile 226. ADC0 on tile 224 was enabled for this application as this tile receives the Phase-Locked Loop (PLL) reference clock via the most direct connection. The settings for tile 224 ADC0 were set as follows in Table 5.1.

The sampling rate configured for ADC0 on tile 224 was set to 1,024 Mega samples per second (Msps) and using a PLL reference clock of 409.6 MHz, the RFDC was instructed to provide a fabric clock of 64 MHz. The fabric clock required to support the AXI4-Stream data signal, received from the RF-ADC according to the ADC tile settings, is 128 MHz, meaning that the ADC fabric clock provided by the RFDC needs to be converted to 128 MHz using a clocking wizard. The received signal from the ADC enters the PL in the form of two AXI4-Stream signals, one each for the I and Q components.

Similarly for the DAC, the XCZU28DR RFSoC part makes available two DACs on tiles 228 and 229. In this instance, DAC pair 0 & 1 on tile 228 were

Table 5.1 ADC tile configuration details

| **ADC Tile Setting** | **Value** |
|---|---|
| Dither | Disabled |
| **Data Settings** | |
| Digital Output Data | I/Q |
| Decimation Mode | 8x |
| Samples per AXI4-Stream Cycle | 1 |
| **Mixer Settings** | |
| Mixer Type | Coarse |
| Mixer Mode | Real → I/Q |
| Frequency | Fs/2 |
| **Analogue Settings** | |
| Nyquist Zone | Zone 1 |
| Calibration Mode | Mode 2 |

enabled because this tile is the first tile to receive the PLL reference clock. The settings for tile 228 DAC 0 & 1 were set as follows in Table 5.2.

Table 5.2 DAC tile configuration details

| **DAC Tile Setting** | **Value** |
|---|---|
| Inverse Sinc Filter | Disabled |
| **Data Settings** | |
| Analog Output Data | Real |
| Interpolation Mode | 8x |
| Samples per AXI4-Stream Cycle | 2 |
| **Mixer Settings** | |
| Mixer Type | Fine |
| Mixer Mode | I/Q → Real |
| NCO Frequency (GHz) | 0.0 |
| NCO Phase | 0 |
| **Analog Settings** | |
| Nyquist Zone | Zone 1 |
| Decoder Mode | SNR Optimised |

The sampling rate configured for DAC 0 & 1 on tile 228 was set to 1024 Msps, the same as the ADC, and using a PLL reference clock of 409.6 MHz. The RFDC was instructed to provide a fabric clock of 128 MHz. The fabric clock required to support the AXI4-Stream data signal going to the RF-DAC, which is 128 MHz, according to the DAC settings. No further clocking conversions were required for the RF-DAC. The AXI4-Stream signal has been configured to contain 2 samples per AXI4-Stream cycle. This means that for each valid data

sample the AXI4-Stream data packet will contain an I sample and Q sample concatenated together.

The exposed ports and interfaces for the RFDC IP core within Vivado IPI are shown in the screenshot in Figure 5.4.

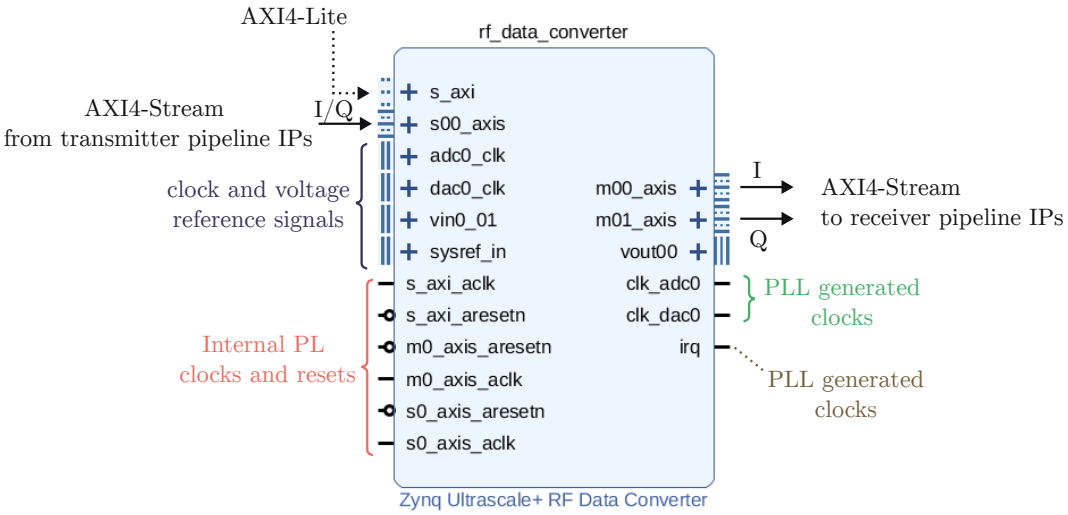



Figure 5.4 The RFDC IP core showing the exposed ports and interfaces.

The RFDC IP core exposes multiple interfaces for control, data streaming, clocking, and synchronisation, as shown in Figure 5.4. The main ports include AXI4-Lite for control and reconfiguration, AXI4-Stream for DAC and ADC data, dedicated clock and reset lines, and external reference signals. These inferfaces enable high-speed streaming and runtime control essential for SDR-based DL applications [58]. Key ports are annotated directly in the figure.

**FIR Interpolation Filters** In the block design description in Figure 5.3, two FIR interpolation filters are used in the transmitter chain. In interpolation, an upsampler inserts zero-valued samples in accordance to the upsample ratio $L$ and following then the resulting low-pass filter with ideal cut-off $f_{\text{cut\_off}} = \frac{f_s\text{out}}{2L}$ removes frequency image components.

In the application of modulation classification, the goal for the transmitter portion of the radio system is to transmit various modulated signals for the radio to receive, demodulate, and send to the CNN accelerator to classify the modulation scheme. In this design, FIR interpolation filters are used with a total interpolation factor of 32, achieved through two cascaded filters with interpolation factors of 4 and 8. An interpolation value of 32 ensures that on the receiver side, the signal can be decimated by 32, significantly reducing the

computational load for the CNN accelerator when processing the received input signal.

The FIR interpolation filter chain converts the sampling rate of a 4 Msps signal up to 128 Msps before it is passed into the RF-DAC for transmission. The interpolation is split between two filters instead of a single filter as this results in a more resource efficient design.

The first interpolation filter in the chain interpolates the signal by 4 and low-pass filters the resulting upsampled signal at a frequency cut off of

$$f_{\text{cut off}} = \frac{f_s\text{out}}{2L} = \frac{16\text{MHz}}{2 * 4} = 2\text{MHz} \tag{5.6}$$

The frequency response of the first FIR interpolator can be seen in Figure 5.5.

In the PL, the filter is implemented using fixed-point arithmetic with the coefficients being set to a representation of signed Q2.14. The filter accepts Q2.14 inputs and outputs Q4.14 samples using a symmetric rounding to zero truncation technique. The truncation technique is used by every subsequent filter. A Q2.14 fixed-point format was selected for the filter input to match the 14-bit resolution of the DAC while keeping the word length to 16-bits (2 bytes). This ensures a consistent fractional bit width across the design and allows the full DAC precision to be utilised.

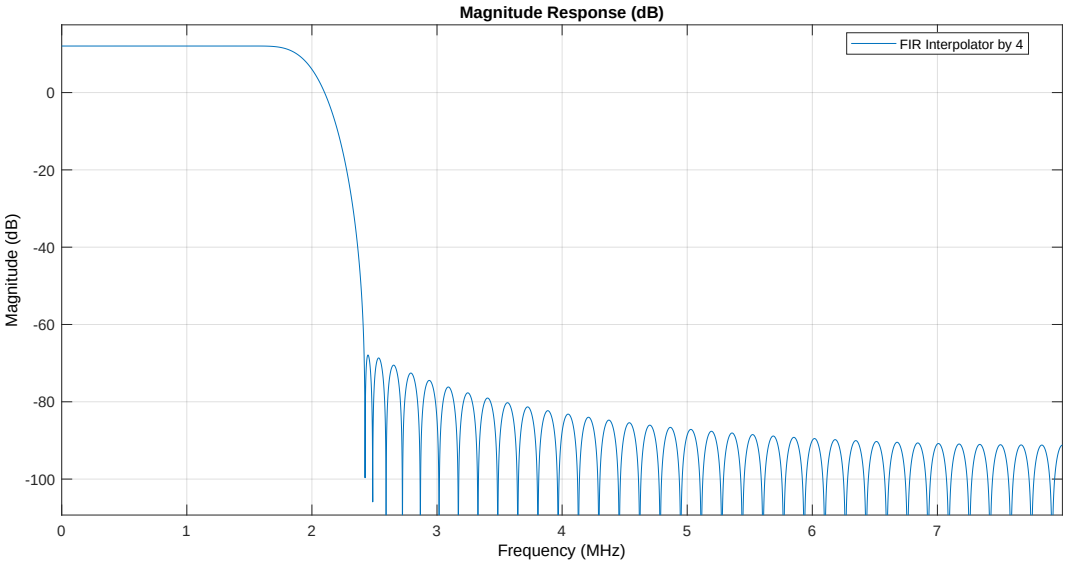



Figure 5.5 Frequency response of FIR interpolation filter by factor of 4.

The next FIR interpolation filter increases the sampling rate by a factor of 8. Similarly to the previous filter, it accepts Q2.14 samples and outputs Q4.14

samples. The coefficients are assigned to a Q2.14 fixed-point number and the filter is configured to support 2 parallel paths for the concatenated I and Q samples. Figure 5.6 shows the frequency response of the FIR interpolator by factor of 8 filter.

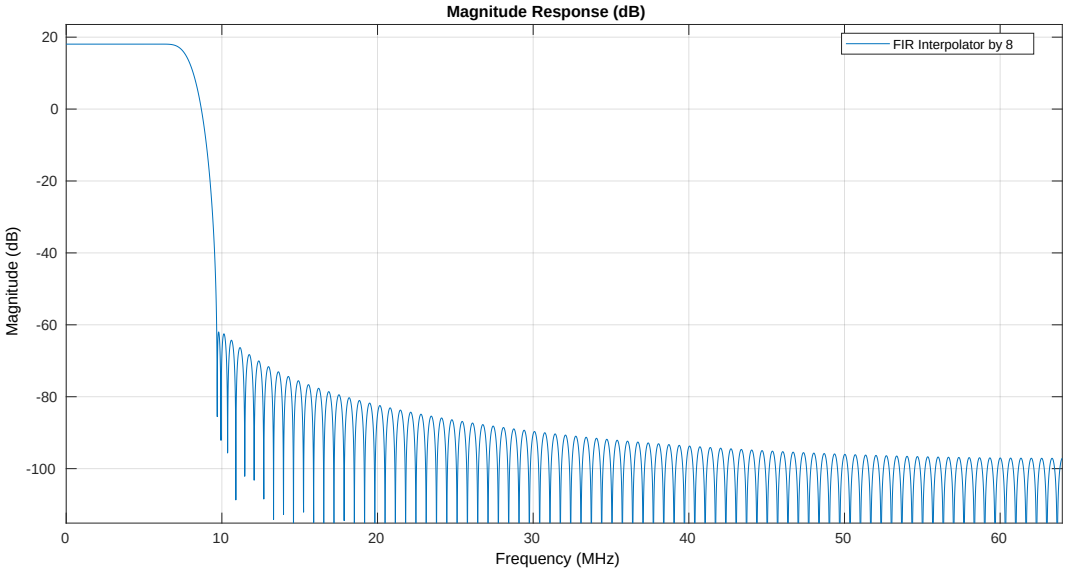



Figure 5.6 Frequency response of FIR interpolation filter by factor of 8.

The combined frequency response of both these filters connected in a filter chain is observed in Figure 5.7. The total combined filter low-pass frequency cut-off, $f_{\text{total\_cut\_off}}$, is at 2 MHz, and attenuates all frequencies to a minimum of -60 dB. The passband responses have gains above 0 dB to compensate for the effect of upsampling (inserting zeros), which reduces signal power.

The resulting signal that has been interpolated by the filter chain is sent to the RF-DAC where it is then transmitted out of the device.

**FIR Decimation Filters** In the block design description in Figure 5.3, two FIR decimation filters are used in the receiver chain. In decimation, a downsampler removes samples from a data stream in accordance to the downsample ratio $R$. Prior to the downsampler, a low-pass filter with an ideal cut-off of $f_{\text{cut\_off}} = \frac{f_s\text{out}}{2R}$ is used to remove frequency components that would otherwise alias.

As mentioned in Section 5.4.2, the total decimation ratio, $R$, used in this design is 32. This is to reduce the computational load for the AI accelerator when processing the received input signal. The data streams received from the RF-ADC are at a sample rate of 128 Msps and after passing through the FIR decimation filter chain with decimation ratio $R = 32$, the resulting sample rate

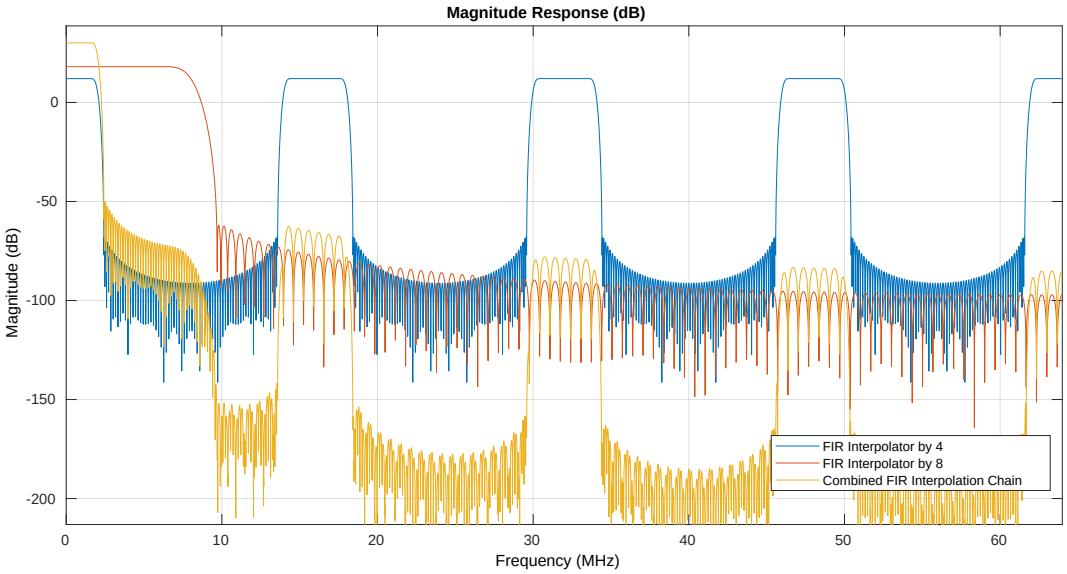



Figure 5.7 Combined frequency response of FIR interpolation chain. Total interpolation factor of 32.

is 4 Msps. At this sample rate, the maximum representable bandwidth possible is 4 MHz, which supports the 1 MHz bandwidth modulated signals being received. The decimation process is separated into two filters with decimation factors of 8 and 4, respectively. Each filters implements a low-pass filter with a frequency cut-off equal to

$$f_{\text{cut-off}} = \frac{f_s\text{out}}{2R} \tag{5.7}$$

where $R$ is the decimation ratio of each filter and $f_s$out is the new lower sampling rate. The low-pass filter is implemented prior to downsampling the data to remove frequency components that are larger than the $f_{\text{cut-off}} = \frac{f_s\text{out}}{2R}$ frequency cut-off, so that after the downsampling phase, those frequency components are not aliased down to interfere with lower frequency components. Figure 5.8 shows the combined frequency response of the FIR decimation chain with a total decimation factor of 32.

The combined FIR decimation chain achieves a low-pass cut-off of 2 MHz. The filter coefficients used are similar to the ones implemented in the FIR interpolators except there is no added gain to compensate for the inserted zeros.

**AXI Direct Memory Access (DMA)** The AXI DMA IP core in Vivado is a configurable IP block used to facilitate high-performance data transfer between system memory, such as DDR and BRAM, and AXI-based IP cores in

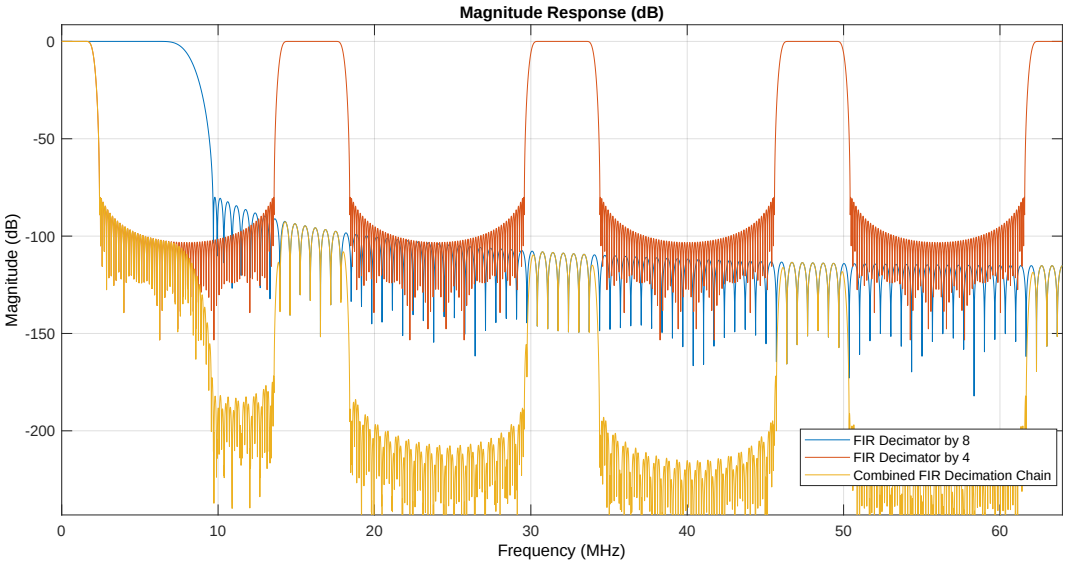



Figure 5.8 Combined frequency response of FIR decimation chain. Total decimation of 32.

FPGA designs. It supports data movement without involving the processor directly in the transfer, freeing up the processor for other PS-based tasks and allowing for efficient asynchronous data transfer between PS and PL [119].

On the transmit portion of the dataset creation block design shown in Figure 5.3, the transmitter DMA, `tx_dma`, is responsible for transferring the chosen MATLAB-generated modulated data from PS DDR to the interpolation filter chain and subsequentially to the RF-DAC. The transmitter DMA's responsibility in this design is to continually and repeatedly transfer data to the FIR interpolation filters without pauses. This is achieved by configuring the DMA in cyclic mode, where it repeatedly transmits the requested data. The transmitter DMA is configured with the following parameters given in Table 5.3.

The configuration parameters in Table 5.3 allow the transmitter DMA to be connected to the rest of the transmitter system. Figure 5.9 is a screenshot of the transmitter DMA connected to the rest of the transmitter IP cores.

The transmitter DMA's AXI4-Stream data ports are connected to the FIR interpolation filters, while its AXI4 Memory Mapped ports are connected to the Zynq UltraScale+ MPSoC IP core to facilitate transfer of the dataset frames stored in PS DDR memory to the PL. The AXI4-Lite connections to the DMA control the cyclic operation. The DMA and FIR interpolation filters are clocked by the PLL generated DAC clock (128 MHz) from the RFDC. A PL clock, operating at 300 MHz, is used to transfer data from the PS DDR memory to

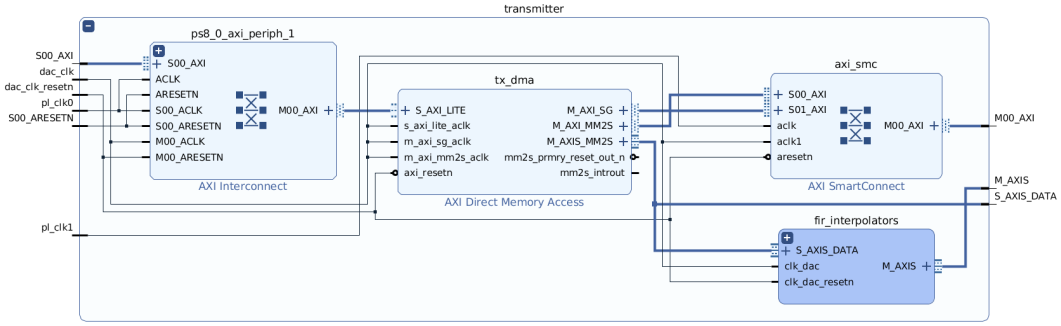



Figure 5.9 The transmitter system IP cores as seen in Vivado.

the DMA. The AXI4-Stream ports from the FIR interpolators connect to the RFDC IP.

Table 5.3 Transmitter DMA configuration parameters

| **Transmitter DMA Setting** | **Value** |
|---|---|
| Scatter Gather Mode | Enabled |
| Micro DMA | Disabled |
| Multi Channel Support | Disabled |
| Control / Status Stream | Disabled |
| Width of Buffer Length Register | 26 bits |
| Address Width | 64 bits |
| **Read Channel** | Enabled |
| Number of Channels | 1 |
| Memory Mapped Data Width | 32 bits |
| Stream Data Width | 32 bits |
| Max Burst Size | 256 |
| Allow Unaligned Transfers | Disabled |
| **Write Channel** | Disabled |
| Number of Channels | - |
| Memory Mapped Data Width | - |
| Stream Data Width | - |
| Max Burst Size | - |
| Allow Unaligned Transfers | - |

**Packet Generator** The packet generator IP core is a custom hardware module designed to generate data packets that are compliant with the AXI4-Stream protocol from continuous streams of data. In a communications system, the flow of data is constant and IP modules on the PL are constantly processing the received/transmitted samples. To facilitate the creation of the dataset, subsets of the received data streams are saved in PS DDR. Since it is not possible to transfer all the data received by the radio to the PS, instead a frame

of data is captured and transferred to the PS DDR to be stored as a part of the recorded dataset.

This data capture method is designed to enable active interaction with the deployed CNN accelerator from the PS using interactive apps. It allows the FPGA to capture data frames at specific moments in time for the purpose of building a dataset.

The packet generator IP core is used in conjunction with the AXI DMA IP core to capture a frame of data from an AXI4-Stream signal and transfer it to the PS DDR using the AXI DMA. The packet generator custom module is required because the AXI DMA itself does not hold functionality to transfer burst of data to the PS DDR. The packet generator contains an AXI4-Stream slave and master port for receiving streams of data and sending bursts of packets to the AXI DMA and has two AXI4-Lite registers used to set the capture size of the packets and another for capturing and sending the packet to PS DDR. Figure 5.10 illustrates a block diagram operation of custom packet generator IP core.

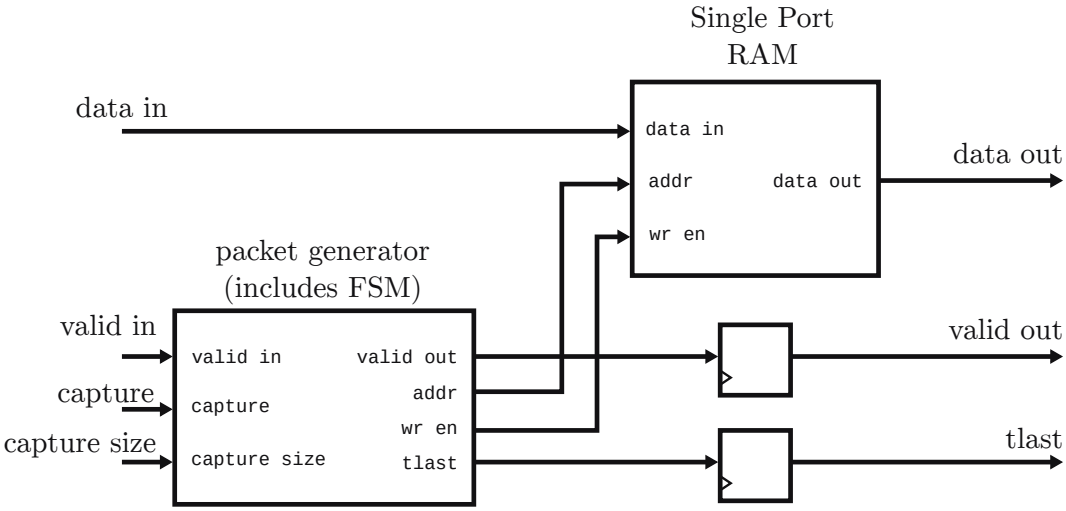



Figure 5.10 The packet generator IP core.

The packet generator IP core consists of an FSM, an on-chip RAM system, and control circuitry for managing the AXI4-Stream signals. The `capture` and `capture_size` AXI4-Lite register signals are connected directly to the state machine, and when the user sets the `capture` register to `True`, the state machine enables the write enable, `wr_en`, port on the RAM system to begin storing valid data samples. The state machine will count the number of samples being stored in the RAM system until it reaches `capture_size`. Once the state machine has fully counted the number of samples stored, the samples are released from the

RAM system and sent out of the IP using the AXI4-Stream protocol, with a TLAST signal being triggered on the final sample. Once the samples have been read out, the IP is ready to receive data again. A flow diagram of the packet generator state machine is shown in Figure 5.11.

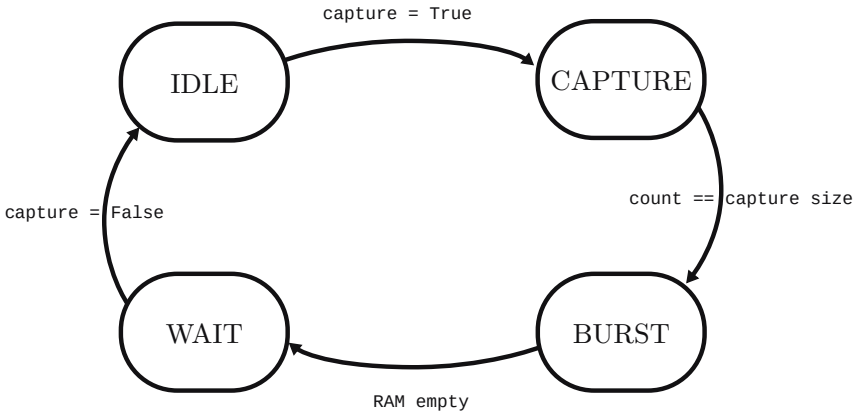



Figure 5.11 The packet generator IP finite state machine flow diagram.

The packet generator IP appears in Vivado as shown in Figure 5.12. The IP contains an AXI4-Stream port on the left-hand side for receiving the streams of received data from the RF-ADC and the FIR decimation filters. The AXI4-Lite port is connected to and communicates with the PS, from where the `capture` and `capture_size` registers are controlled.

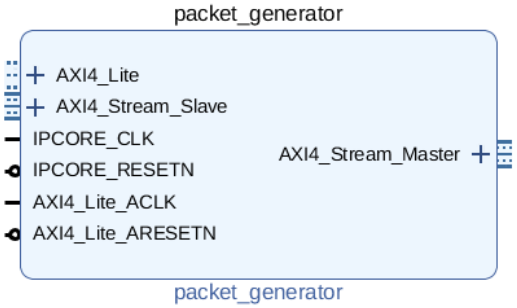



Figure 5.12 The packet generator IP as seen in Vivado.

**AXIS Clock Converter** The AXI4-Stream Clock Converter IP core in Vivado is a standard IP core provided by AMD, that enables seamless data transfer between components operating on different clock domains via AXI4-Stream interfaces. This IP core is useful in designs where different parts of the system run on distinct clock frequencies, and synchronisation is needed to ensure reliable data flow.

In the dataset creation block design diagram in Figure 5.3, the AXIS clock converter IP is used to transfer the data captured from the packet generator

from the RF-ADC generated `clk_adc`, which operates at 128 MHz, to the faster `pl_clk1`, which operates at 300 MHz. The clock domain of the captured AXI4-Stream signal is converted because a faster clock domain will ensure that the data is sent from the PL to the PS DDR at a sufficiently high rate to maintain real-time operation.

### 5.4.3 Dataset Collection Through RFSoC

This section covers the transmission, reception, and recording of the MATLAB-generated modulated data once it has been transferred to the AMD RFSoC development board as seen in Figure 5.2. The block design used in the PL of the AMD RFSoC device was described in Section 5.4.2 and this section will detail how the block design was used to transmit and record the generated modulated signals. The PL bitstream was interfaced using the PYNQ framework [51]. For the dataset creation block design, PYNQ was used to handle all aspects of data transfer and dataset labelling, as well as control of the AXI4-Lite connected PL IP cores.

The recording of the dataset was performed on the AMD RFSoC 2x2 development board. Figure 5.13 shows a picture of the AMD RFSoC 2x2 development board connected in RF loopback via the Nooelec VeGA module [120], and an in-line low-pass filter with a frequency cut-off of 2.5 GHz, which removes any unwanted high-frequency components. The board setup depicted in Figure 5.13 contains the block design shown in Figure 5.3.

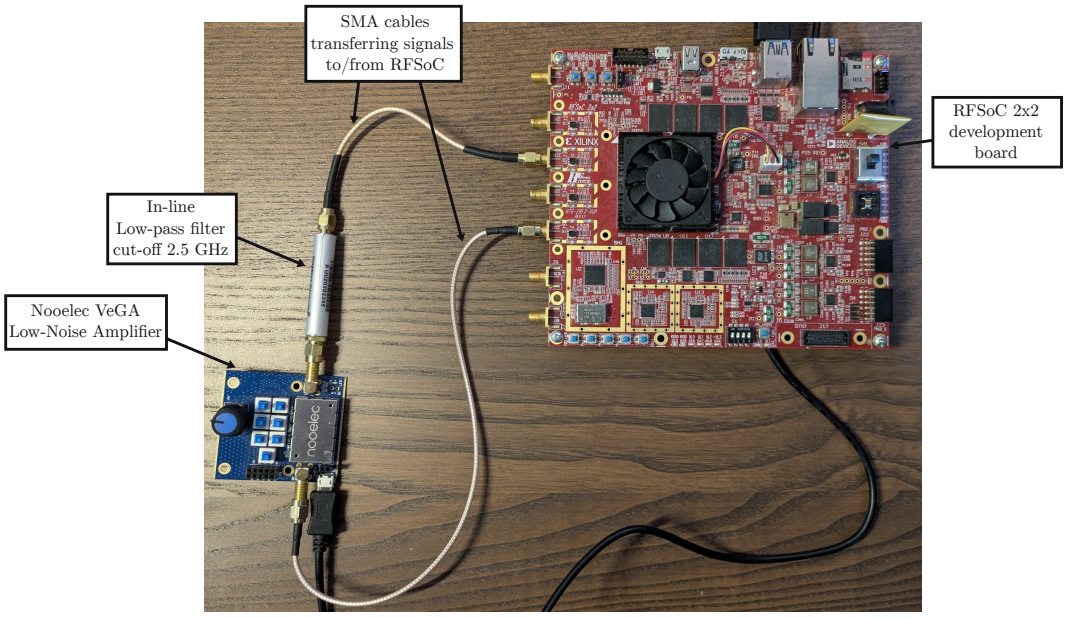



Figure 5.13 The RFSoC 2x2 connected in loopback via the Nooelec VeGA.

The enabled ADC0 on tile 224 was connected to the DAC pair 0 & 1 on tile 228 through an RF loopback connection. An RF loopback connection was used to ensure compliance with RF spectrum access regulations. The RF loopback connection consisted of SubMinature version A (SMA) connected coaxial RF cables screwed into the ADC and DAC SMA connectors on the AMD RFSoC 2x2 development board.

The ADC and DAC SMA connections were then connected to a Nooelec VeGA Barebone Ultra Low-Noise Variable Gain Amplifier (VeGA) module for RF and SDRs [120]. The VeGA is a wideband (30 MHz - 4000 MHz) high performance, general purpose Low Noise Amplifier (LNA) module with a built-in variable attenuator to control the power and amplitude of the RF signals entering the RFSoC.

The VeGA was used in this implementation to 'boost' the signal received so that the ADC on the RFSoC 2x2 utilises as much of the 14-bit resolution ADC as possible. The VeGA's gain control switches (D0 - D5) were all set to off, meaning that the VeGA was set to its minimum possible gain of $V_c = 0$V. According to the Nooelec VeGA data sheet [120], setting the control voltage, $V_c$, to 0V results in the gain of the signal being between -3dB and 20dB. The variability suggests that even when $V_c = 0$V, the device may still provide some level of amplification (up to 20dB), depending on specific conditions, such as frequency, input power, and component tolerances. Given this knowledge of the VeGA's gain range, it is important that the gain is calibrated prior to performing the dataset recording, by transmitting test signals through the loopback connection and inspecting the resolution of the received data through an Integrated Logic Analyser (ILA) in the Vivado block design. A diagram of the Nooelec VeGA is shown in Figure 5.14.

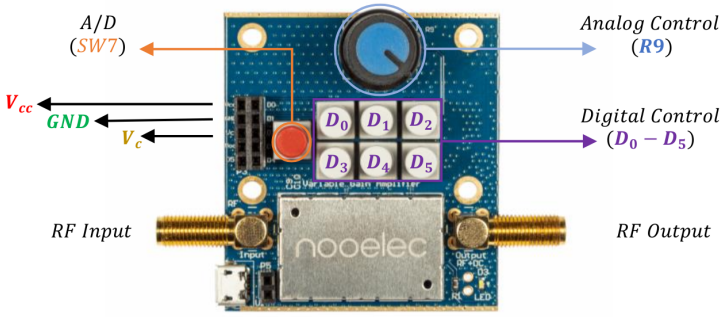



Figure 5.14 A labelled diagram of the Nooelec VeGA.

Once the RFSoC board's ADC and DAC have been connected in loopback, the PYNQ SD card is inserted into the device and powered on. The RFSoC

boot up sequence initialises the PYNQ framework with the Jupyter Labs environment where the dataset recording application files are run.

The dataset transmit and recording operates as described in Figure 5.2. Prior to transmitting, the generated MATLAB data must be transferred to the PS DDR RAM on the RFSoC device for the Jupyter session and Python files to load and inspect all of the saved modulated data. Alongside a Jupyter notebook, the generated data is stored in a folder labelled 'transmit_set', with each set of generated modulated signals for each modulation scheme under the file name `transmit_{mod}_SNR.pkl` (`mod` is replaced with the associated file for each modulation scheme).

Each file contains data for one modulation scheme, spanning the full SNR range defined in Section 5.4.1.

To automate dataset creation, a custom overlay was implemented in Python using the PYNQ framework. This overlay abstracts low-level register operations for the RFDC, DMA, and packet generator IPs and exposes a simplified interface for transmitting and receiving signal data via three methods: `send()`, `receive()`, and `stop()`. These methods were used to build a real-time loopback system for generating labeled data across a range of modulation schemes and SNR values. The main dataset creation loop is summarised below.

The following steps summarise the key operations of the data generation process:

1. **Dataset Preparation**: Each modulation scheme has an associated dataset stored in serialised files (e.g. `transmit_qpsk_SNR.pkl`). These datasets are loaded into PS DDR memory and indexed by the modulation type and SNR level using a key value pairing (e.g. `dataset['QPSK','30']`). Each modulation and SNR value pair contains 800 frames of I/Q modulated data totalling a length of 4,096 samples. The workflow accesses each of these frames sequentially.

2. **Phase Offset**: A range of phase offsets ($-180°$ to $+180°$ with $10°$ increments) is cyclically applied to the data to expose the data to a wide range of phase offset values, to induce a phase offset resilience during training. The phase offset is updated for every transmitted frame.

3. **Signal Transmission**: The modulated data is scaled to fit within a 16-bit signed integer word length, from a floating-point data type, by a factor equal to the fractional bit length configured in the PL. The 16-bit I/Q samples are interleaved to formulate a 8,192 sample long interleaved

array, which is then transferred cyclically by the transmitter DMA to the PL (using the `send()` overlay function), through the FIR interpolators and subsequentially the RF-DAC. The transmitter DMA is configured to transfer 32-bit sameples from the PS memory. Since the interleaved 16-bit I/Q samples are stored in contigous memory, the DMA collects 32 conscutive bits, resulting in each samples containing an I and Q pair, as required by the PL.

4. **Signal Reception**: The transmitted signal is looped back through the RF-ADC, where it is received and reconstructed into complex I and Q data arrays. Each frame captured is 128 I/Q samples long and is received 16 times to capture variations and increase the dataset diversity.

5. **Data Accumulation**: The received complex data is stacked across multiple frames to form a comprehensive dataset for each modulation and SNR combination. The result is a 3D array representing the accumulated signal data.

6. **Serialisation of Processed Data**: The processed data is stored in a dictionary and serialised into files (e.g. `loopback_train_QPSK.pkl`) for subsequent training and evaluation of the modulation classification system using the 'pickle' software package.

Full code listings for the overlay class and dataset creation script are provided in Appendix B.

Once the individual loopback training files (e.g. `loopback_train_QPSK.pkl`, `loopback_train_BPSK.pkl`, etc) are generated on the RFSoC board, they are copied to an external system for further processing. These files contain the received complex signal data for each modulation type and corresponding SNR levels, organised as separate serialised datasets. To streamline the training and evaluation processes, the individual files are combined into a single unified file, `loopback_train.pkl`. This consolidated file merges the datasets for all modulation schemes and SNR conditions.

In addition to recording a training dataset through the loopback process, a separate testing dataset is created by extracting a smaller subset of the transmitted data. This is to provide a smaller testing file that can be stored on the RFSoC without the need to maintain the larger generated testing set. Once the recorded set has been saved, the larger generated modulated data files are no longer required and the newly created smaller testing file,

`transmit_test_SNR.pkl`, can be stored as a source of pre-transmitted signals for testing the resulting AI accelerator.

## 5.5 Training on New DeepRFSoC Dataset

The following section describes the design and training of a neural network for modulation classification using the DeepRFSoC dataset generated with the RFSoC described in Section 5.4.3. The primary objective of training the neural network is to produce a model that accurately classifies the modulation scheme within the received signal. The resulting weights serve as the foundation for deployment within the AI accelerator, discussed in Chapter 4. This hardware-accelerated implementation is tested with signals encoded using various modulation schemes to evaluate the performance in real-time scenarios.

### 5.5.1 Dataset preparation

The DeepRFSoC dataset created in Section 5.4.3 produced a set containing 2.6 million example frames across all eight modulation schemes and SNR values. Each frame has 2 channels that are 128 samples long. Each modulation scheme holds 330,000 frames at varying SNR values. Each frame of the dataset is accessible via a key pair for the requested modulation type and the SNR value (e.g. `dataset['QAM16','16']`). The dataset accepts keys for the modulation type the signal has been encoded with and the applicable SNR value. The choice of modulation schemes is: QPSK, BPSK, QAM16, QAM64, 8PSK, PAM4, GFSK, and CPFSK, and the choice of SNR values is: -20, -16, -12. -8, -4, 0, 4, 8, 12, 16, 20, 24, 28, and 30 dB.

From a Python environment, the dataset is loaded in through the Pickle package using `dataset = pickle.load(file_name)`. The dataset frames and labels are shuffled to break up the pattern of the ordered dataset. This is to ensure, once the dataset is split between training, validation, and testing sets, that the distribution of data frames is equal between all sets and to ensure that, during training, the model does not learn patterns based on the sequence of data rather than from the data features.

The dataset is split into three sub sets, namely training, validation, and testing, as previously described in Section 4.5.1. In the case of this dataset, the set is split between training, validation, and testing at a ratio of $7 : 1 : 2$.

### 5.5.2 Neural Network

The candidate neural network trained by the newly created custom dataset mimics the neural network used in the work on RadioML in Chapter 4. The CNN topology is shown in Table 5.4. There are two convolutional layers and two FC layers, with ReLU activations after each layer except for the classification layer, where a Softmax activation exists instead to convert the output to a probability distribution.

Table 5.4 CNN Dimensions

| Layer Type | Kernels/Weights | Activations | Parameters |
|---|---|---|---|
| Input | $2 \times 128$ | - | - |
| Convolution | $64 \times 3 \times 1$ | ReLU | 192 |
| Convolution | $16 \times 3 \times 2$ | ReLU | 6,144 |
| Fully-connected | $1984 \times 128$ | ReLU | 253,952 |
| Fully-connected | $128 \times 8$ | Softmax | 1,024 |
| Output | $1 \times 8$ | - | |

The neural network dimensions were kept the same as in Chapter 4 to enable a direct comparison between the performance of the new DeepRFSoC dataset and that of the RadioML synthetically generated dataset. Maintaining the same neural network topology makes it possible to evaluate the dataset generation process.

### 5.5.3 Training

The training process aims to optimise the CNN parameters to accurately predict the modulation scheme that a signal has been encoded with. Through the PyTorch software framework, using a desktop graphics card, the dataset is used to train the neural network. It is important to note that the dataset has not been normalised. Typically a dataset is normalised prior to training, as it ensures a faster convergence (as the input features are adjusted to exist on a similar scale), as well as providing model stability [121]. When considering the custom DeepRFSoC dataset, normalisation is not possible. This is because, if the dataset was normalised, it would not accurately represent the data as it is received by the RFSoC, and therefore the AI accelerator, once deployed, would have been trained on data that is numerically different to what it would actively receive when operating in real-life.

The parameters for training the neural network are detailed in Table 5.5. An Adam optimiser method was used to provide an adjustable learning rate [87]

of $1e^{-4}$. The loss function used was cross-entropy [122], a common loss function for calculating the loss for classification-based DL tasks. The training operates for approximately 100 epochs unless early stopping is triggered. Early stopping is triggered if the validation loss has not reduced for more than the specified number of epochs referred to as patience, in this case 8, and once triggered the training is halted.

Table 5.5 Training parameters of AMC model.

| **Parameter** | **Value** |
|---|---|
| Optimiser | Adam |
| Loss function | Cross-Entropy |
| Batch size | 128 |
| Number of epochs | 100 |
| Learning rate | $1e^{-4}$ |
| Weight decay | N/A |
| Early stopping (patience) | 8 |

Figure 5.15 shows the training loss for the DeepRFSoC dataset given the parameters in Table 5.5. Although the training session was configured to train for 100 epochs, the early stopping mechanism halted the training after 20 epochs, as the validation loss showed no improvement beyond that point.

### 5.5.4 Testing

The resulting AMC model is was evaluated against the reserved testing set. The results in Figure 5.16 demonstrate the accuracy of the trained model across all modulation schemes against the noise of the signal.

A peak accuracy of 80% at the maximum SNRs and a 40% accuracy at 0dB SNR show that the method for recording a dataset through the transmission and reception of the RFSoC produces a trainable model, and can be reliably used to train, even without normalisation.

Additionally, in Figure 5.16, the accuracy of the RadioML dataset is also plotted, trained with the same CNN topology. Both datasets show a similar trend in improving accuracy as the level of noise decreases, although the model trained with the DeepRFSoC dataset improves its accuracy at higher values of SNR compared to the model trained with RadioML. The reason behind this is due to the differing channel conditions between the RadioML and DeepRFSoC datasets. While the RadioML dataset simulated a multipath channel with GNU Radio, the DeepRFSoC dataset used the MATLAB Communications

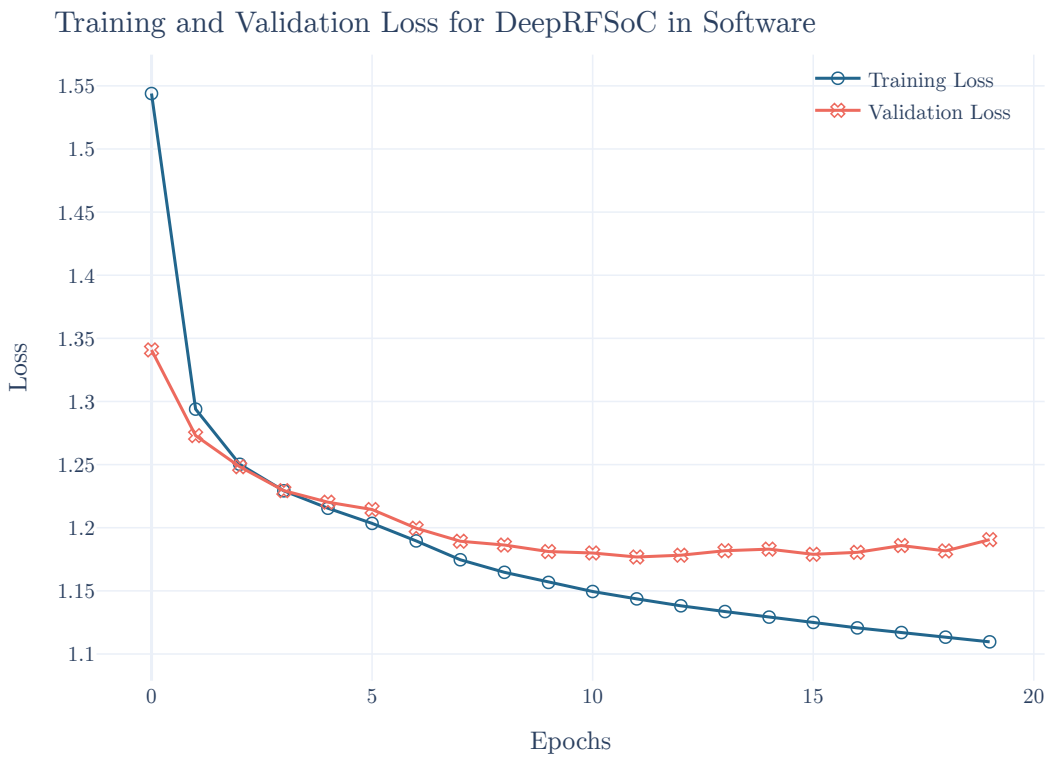



Figure 5.15 Training loss on custom dataset - DeepRFSoC.

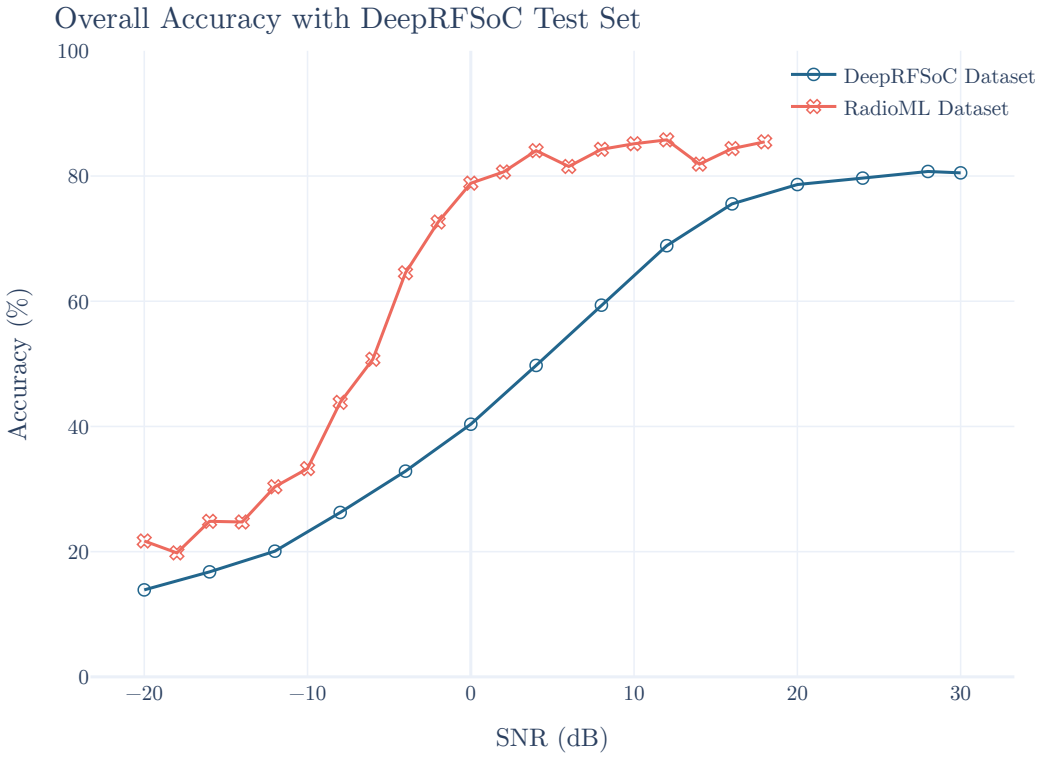



Figure 5.16 CNN model performance against testing set of DeepRFSoC, with RadioML accuracy for comparison.

Toolbox functions and a harsher channel, as well as the recorded impurities in the hardware loopback configuration. This new accuracy curve for DeepRFSoC will be used as the baseline accuracy for the training considerations explored in Chapter 6.

The results seen in Figure 5.16 confirm that the next stage of deploying the model on the streaming-based CNN accelerator from Chapter 4 can be undertaken. The next challenge is to test the CNN accelerator on signals that are received live during transmission, extending the capabilities of the model further than testing against a prerecorded dataset. This poses a significant challenge, as testing a signal on live data (as opposed to a testing set) would involve signal values that exist outside of the distribution of the dataset. For this reason, careful training and deployment must take place in order to produce a reliable model that operates on hardware to perform the classification of signals.

As stated in Chapter 4 Section 4.5.2, while the accuracy performance of the trained model is not optimal, the goal of this work is not to provide the best accuracy network, but rather to demonstrate how a trained neural network can be integrated onto a real-time radio receiver and achieve real-time operation. The resulting accuracy for both models trained on the two datasets achieves a comparable peak accuracy to other works for networks and inputs of this size [70].

## 5.6 Integration with Embedded FPGA Device

With the trained model demonstrating reliable performance on the DeepRFSoC dataset, the next step involves its integration with the embedded AI accelerator introduced in Chapter 4. This phase shifts the focus from controlled dataset evaluation to real-time, hardware-based signal classification, involving both architectural and algorithmic considerations to ensure robust performance. The deployment process involves translating the trained model's parameters into a fixed-point format within the AI accelerator and addressing hardware-specific constraints such as latency, resource utilisation, and performance once deployed on the streaming-based architecture, from Chapter 4. Additionally, the architecture and its trained weights are tested while receiving signals live, demonstrating the capabilities of the application and proving the effectiveness of training on data transmitted on the RFSoC.

The next section details how the model weights are extracted and implemented into the CNN model from Chapter 4. The conversion of the dataset building block design to include the CNN accelerator is discussed, and an overview of the software drivers for controlling the CNN accelerator and other functions is presented.

### 5.6.1 Export Model Parameters to AI Accelerator

To enable the deployment of the trained neural network on the CNN accelerator, it is essential to export the model parameters from the training environment in PyTorch and adapt them for hardware compatibility. This process begins with extracting the learned weights from the PyTorch model, which are initially stored in floating-point format. These parameters are exported to a `.mat` format for the weights to be accessible from MATLAB.

Once exported, the parameters are imported into MATLAB. Here the weights undergo PTQ. This is a technique used to reduce the size and computational complexity of a trained neural network, by converting its weights and activations from a high precision floating-point number (typically 32-bit) to a lower-precision format, such as 16-bit or 8-bit floating-point, or even fixed-point representations. This process is done after the model has been fully trained, without the need to retrain it.

In this work, PTQ was applied to the exported model parameters by converting both the weights and the inter-layer activations to a 16-bit fixed-point representation. This process applies a uniform 16-bit word length across all layers, while the fixed-point scaling is determined dynamically on a per-layer basis. This means that for each layer of the model, the fixed-point representation holds a different number of fractional bits according to the distribution of weights for that layer. This can be expressed as

$$w_q^l = \text{quant}[S_q^l(w^l - O_q^l)] \tag{5.8}$$

where $l$ and $q$ represent the current layer and quantisation configuration, respectively. The resulting quantised weight for each layer, $w_q^l$, is determined by quantising to 16-bits and scaling the resulting weight by scaling factor $S_q^l$ and offsetting the layer by $O_q^l$ [123]. In this case, the offset, $O_q^l$, for each layer is 0 and the scaling factor, $S_q^l$, is limited to a power-of-two as this is directly linked to the number of fractional bits within the fixed-point 16-bit number. Additionally, since the streaming-based CNN architecture stores all weights in

on-chip BRAM, the weight quantisations are grouped by layer instead of by channel.

The model weights and their distributions are shown in Figure 5.17.

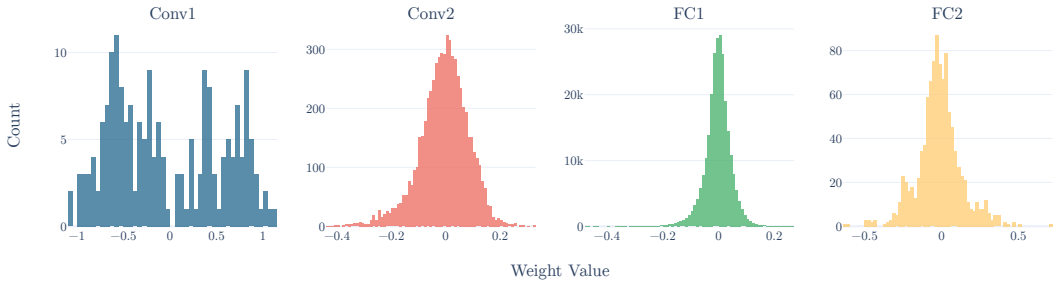



Figure 5.17 Weight distribution for each layer of the trained CNN model trained with DeepRFSoC.

The weights were quantised to a 16-bit format with 16-bit activations using the PTQ technique as detailed in Section 4.6.3.

### 5.6.2 CNN Accelerator IP Core Integration with Block Design

Similarly to Chapter 4, the generated IP core from MATLAB and Simulink HDL Coder is integrated into the Vivado IPI block design. The IP core is integrated into the same block design as described in Section 5.4.2, which was used to record the DeepRFSoC dataset for obtaining the AI accelerator. Figure 5.18 illustrates the updated block design with the new CNN accelerator IP core included.

The CNN accelerator IP core is integrated into the dataset building block design by forking the resulting signal from the FIR decimation filter chain. The signal's I and Q values are interleaved before being passed to the AMC CNN accelerator IP core. The IP core then processes every sample received by the FIR decimation chain and predicts the modulation scheme the received signal is encoded with. The resulting classifications are then packetised into an AXI4-Stream packet and sent to the `amc_dma` IP core after being converted to the `pl_clk1` clock domain, operating at 300 MHz. The packets sent from the DMA to the PS are then stored in PS DDR ready to be read by the software program operating on the PS.

The CNN accelerator IP core is tightly integrated into the communications pipeline of the receiver, enabling real-time processing of incoming signal samples.

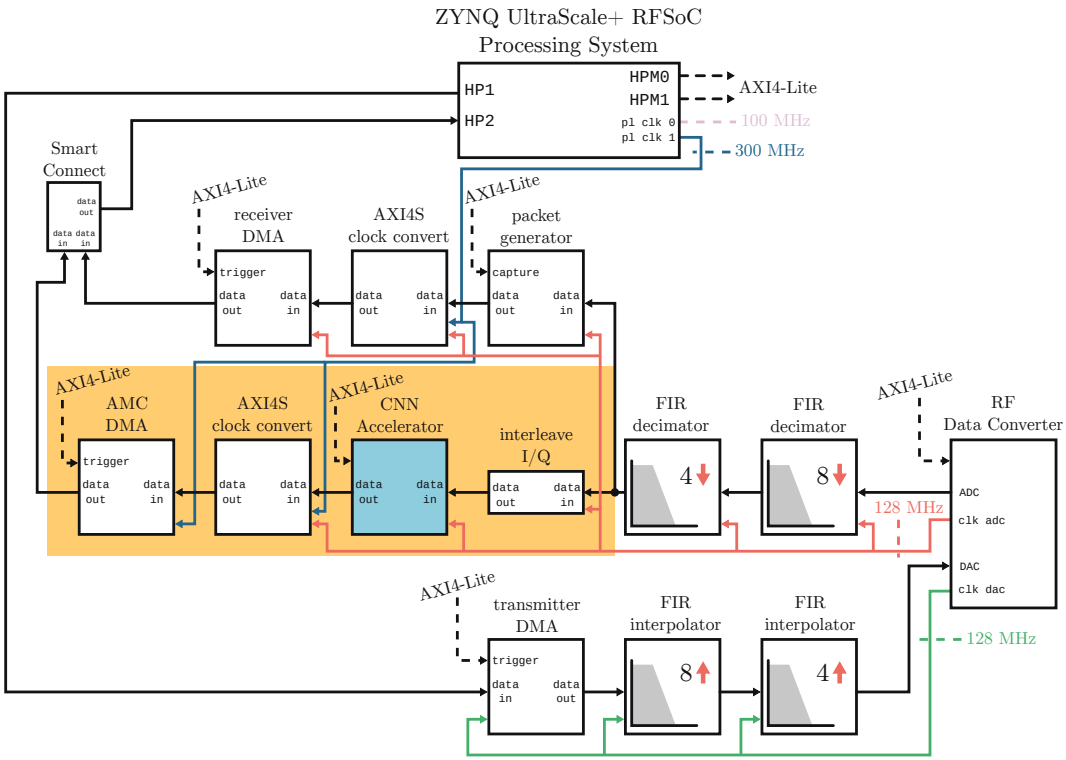



Figure 5.18 PL design with integrated AMC CNN Accelerator.

By embedding the IP core within the receiver chain, each sample can be processed sequentially as it is received, allowing for immediate classification. This eliminates the need for batch processing, reducing the overall latency and ensuring that classification results are available with minimal delay. The direct integration into the pipeline also optimises data flow, leveraging the inherent streaming nature of the architecture to maintain a high throughput while predicting the modulation scheme of the signal. This architectural approach compliments the existing flow of data that exists in radio receivers and proves that the proposed architecture type can be a viable candidate in AI solutions for PHY radio receivers.

With the CNN accelerator integrated, the updated block design is synthesised and implemented to the RFSoC PL fabric where a resulting bitstream is produced. Table 5.6 shows the resource consumption of the AI accelerator IP core in the RFSoC PL fabric.

Although the CNN accelerator IP core has been updated with new PTQ weights from the dataset recording and training, the resource allocation of the model is similar to that of the model produced in Chapter 4. The resulting bitstream produced after the integration of the CNN accelerator maintains the capabilities of: transmitting a desired signal encoded to a select modulation

Table 5.6 FPGA resource utilisation of the deployed CNN accelerator with DeepRFSoC weights.

| **Model** *layer* | **Slice LUTs** | **Slice Register** | **DSPs** | **BRAMs** | **URAMs** |
|---|---|---|---|---|---|
| **CNN** | **27,272 (6.41%)** | **40,015 (4.7%)** | **456 (10.67%)** | **169 (15.64%)** | **1 (1.25%)** |
| *conv1* | 3,237 | 4,248 | 64 | 0 | 1 |
| *conv2* | 15,105 | 20,423 | 256 | 32 | 0 |
| *fc1* | 7,837 | 14,282 | 128 | 136 | 0 |
| *fc2* | 761 | 797 | 8 | 0 | 0 |

scheme; interpolating the signal to a desired sampling rate; transmitting out of the RF-DAC; receiving the data on the RF-ADC via loopback cable; decimation from a higher to lower sampling rate; and capturing packets of the received data to send back to the PS; this is all with the added functionality of classifying the signals received from the RF-ADC. The following sections will cover the evaluation of the deployed model including interfacing with PYNQ, its accuracy performance, and the resulting latency of the deployed model as it operates with real-time data.

## 5.7 Evaluation of Real-Time Modulation Classification

In this section, the performance of the AMC CNN accelerator IP core is evaluated in a similar way to Section 4.9. The difference in this case is that the CNN accelerator is evaluated on its ability to operate in real-time while receiving signals from the RF-ADC, thus demonstrating the effectiveness of training a model with a custom generated dataset based on recording samples with the RFSoC. The results evaluated metrics are: overall accuracy, throughput, and latency.

The results captured in this section were recorded on the AMD RFSoC 2x2 development board. The board was configured with PYNQ version 3.0.1 installed on the SD card, loaded with Python files and drivers that transmit a signal encoded on a desired modulation scheme and are sent classifications from the deployed AI accelerator.

The AMD RFSoC 2x2 board was configured in RF loopback through a SMA cable and LNA configured as specified in Section 5.4.3 and shown in Figure 5.13.

### 5.7.1 Accuracy of Deployed Model

Figure 5.19 illustrates the overall accuracy of the deployed model. Its floating-point weights were quantised to 16-bits with a fractional point specified to best support the weights for each layer. Alongside the accuracy of the deployed model, the accuracy of the trained model tested against the recorded dataset is plotted as well.

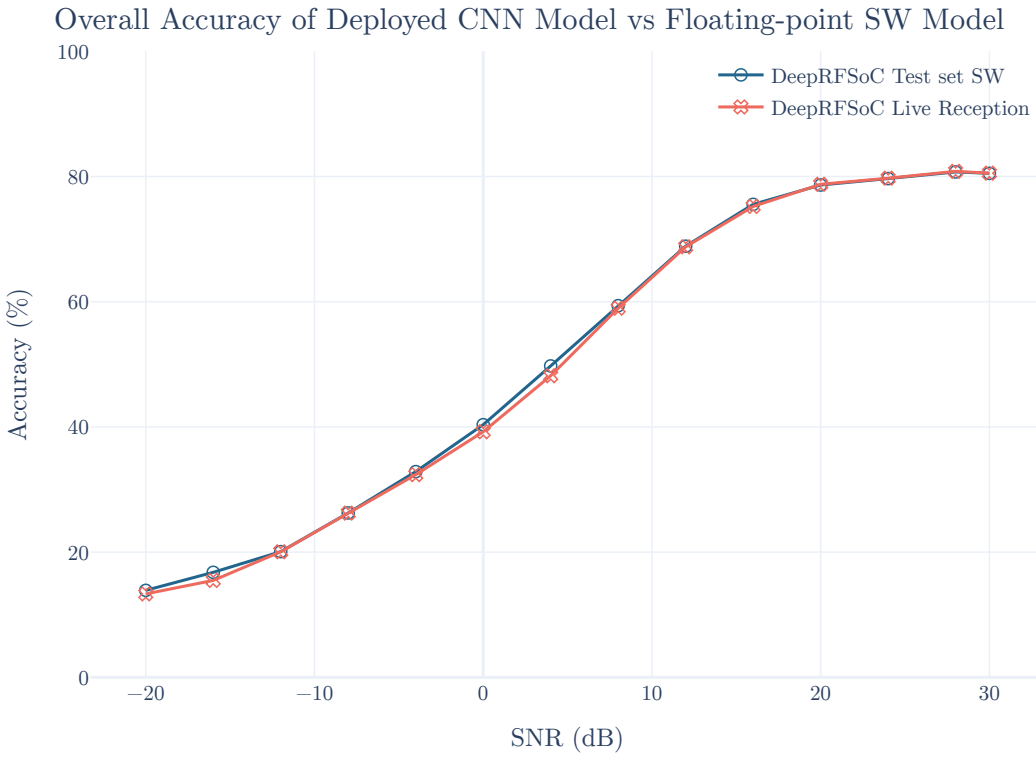



Figure 5.19 Overall accuracy of CNN accelerator IP core across SNR values.

The accuracy of the deployed model with its weights quantised to 16-bits shows performance equal to the accuracy of the trained model tested in PyTorch indicating that the PTQ performed on the model weights is successful at representing the weights from the trained model. Additionally, the quantised activations also successfully represent the inter-layer signals between the neural network layers.

### 5.7.2 Latency and Throughput

Table 5.7 presents a comparison between this work and several FPGA-based CNN accelerator implementations from the literature which have been designed for modulation classification tasks. The model featured here was trained using the DeepRFSoC dataset and deployed within the RFSoC communications pipeline using the streaming-based CNN accelerator introduced in Chapter 4. Both weights and activations are quantised to 16-bit fixed-point format, as indicated by the 16w16a label in the table.

Table 5.7 Comparison with CNN accelerators for modulation classification.

| **Accelerator** | **This work (16w16a)** | **Tridgell et al.**[47] | **FINN** [46] | **Hou et al.** [42] | **Jung et al.** [43] |
|---|---|---|---|---|---|
| **# Params** | 260k | 636k | 161k | 216m | 198k |
| **Topology** | 2 conv/2 fc | VGG10-L (128, 512) | VGG10 (64, 128) | VGG-16 | 3 conv/2 fc |
| **Quantisation (weight/act.)** | 16b/16b | 2b/incr. prec. | 4b/4b | Not Reported | 16b/16b |
| **Latency ($\mu s$)** | 29.6 | 8 | 11.7 | 234 | 12.4 |
| **Noise (dB)** | -20 to 30 | None | -20 to 30 | -2 to 10 | -20 to 30 |
| **Channel** | Multipath + RFSoC | None | Multipath | None | Multipath |
| **Clock rate** | 128 MHz | 250 MHz | 250 MHz | Not Reported | 100 MHz |
| **Accuracy (@ SNR dB)** | 81% @28dB | 80.2% @30dB | 94.1% @30dB | 92.36% @6dB | 75% @>0dB |
| **Throughput (cps)** | 34k | 488k | 120k | Not Reported | Not Reported |
| **Results from live RF** | Yes | Partial | No | No | No |

A key distinction of this work is that the results are based on real-time classification of live signals received directly from the RF-ADC, as opposed to relying solely on pre-recorded datasets or synthetic test samples. This makes the evaluation more representative of actual deployment conditions in embedded radio systems.

Notably, only Tridgell et al. [47] demonstrate partial live testing, and none of the other referenced works incorporate hardware impairments such as RF-ADC stitching artefacts or real decimation stages in their evaluations. The overall model complexity in terms of parameter count (260k) is lower than some works (e.g., Hou et al. [42] with 216 million), the architecture is optimised for low-latency operation on real data. The reported latency of 29.6µs reflects end-to-end processing from the reception of a full input frame to classification output, including clock rate increases and interleaving to account

for the overproduction of data from the convolutional layers. This is higher than works like FINN [46] and Tridgell et al. [47], which report latencies of 11.7µs and 8 µs respectively. However, these architectures assume idealised data access conditions and do not incorporate real-time live reception constraints.

In terms of throughput, the proposed architecture sustains 34k classifications per second (cps), which is lower than the 120k cps reported by FINN or the 488k cps achieved by Tridgell et al. This is a deliberate trade-off: the system was designed to operate at the line rate of the RF-ADC interface, ensuring that no incoming samples are dropped while maintaining full data fidelity. The 128 MHz clock rate is also lower than the 250 MHz used in some other designs, reflecting the integration of the accelerator into a realistic receiver chain with timing constraints.

Despite these trade-offs, the model achieves a classification accuracy of 81% at 28 dB SNR on a live signal path, closely matching or exceeding other real-time capable implementations.

Future improvements could include introducing inter-layer scheduling or deeper MAC parallelism to reduce latency and improve throughput, similar to techniques adopted in [39], [47]. However, this work prioritises end-to-end operability with real-time RF data streams, which distinguishes it from prior, purely synthetic evaluations.

## 5.8 Chapter Conclusion

This chapter has presented the integration of the streaming-based CNN accelerator with a live SDR receiver and introduced a practical method for generating a custom dataset compatible with real-time operation and live data acquisition. While the RadioML dataset served as a benchmark in earlier chapters, its fixed-length frame format and purely synthetic nature make it unsuitable for validating live classification performance. To overcome this, a hybrid approach was introduced where synthetic modulation scheme signals were passed through the RF loopback path of the RFSoC, capturing real hardware effects such as non-linearities, digitisation artefacts, and ADC tile stitching. This process produced the DeepRFSoC dataset.

This chapter has described the construction of the DeepRFSoC dataset, detailing the transmission setup and the signal processing stages used to send a synthetically generated dataset through the RFSoC's transmit/receive path. The CNN topology introduced in Chapter 4 was retrained on this new dataset,

resulting in a model that performed well despite hardware impairments and RF distortions in the dataset. Following quantisation, the trained model was deployed onto the FPGA-based accelerator and evaluated under live operating conditions.

Live classification experiments confirmed that the deployed model could match its floating-point counterpart in performance while maintaining throughput in a real-time signal environment. Finally, a comparative analysis against other FPGA-based CNN accelerators for modulation classification highlighted this work's unique focus on live RF integration, balancing latency, throughput, and deployability within a SDR signal processing pipeline.

# Chapter 6

# Low-Precision Weight Optimisation

This chapter aims to determine which quantisation strategies are best suited for real-time CNN inference on RFSoC-based radio systems. It explores two techniques for producing lower precision weights for DL models. The investigation explores a range of precisions and evaluates the resulting models with the DeepRFSoC dataset, and the classification of received live modulated signals. The accuracy performance of each technique is analysed and the resulting implementation considerations are formulated.

## 6.1 Motivation

While the CNN architecture introduced in Chapter 4 and deployed in Chapter 5 successfully performs real-time modulation classification on live RF data, it currently operates using 16-bit fixed-point weights and activations that have been quantised to represent a set of trained floating-point values. While quantising to 16-bit weights showed equal accuracy performance to the equivalent floating-point model, the question arises as to the extent that the precision of the fixed-point weights be pushed down to.

Lowering the model's precision offers a clear path to lowering memory usage, reducing latency, and improving throughput. However, lowering weight precision comes with the risk of degrading model accuracy, especially in noisy or low-SNR conditions. To address this trade-off, this chapter investigates two strategies: PTQ, where a floating-point model is quantised post-training; and QAT, where the model is trained with quantisation effects simulated during training.

Both PTQ and QAT are applied to the same CNN model trained on the DeepRFSoC dataset, with weights and precisions ranging from 16-bit down to 2-bit. Each quantised model is deployed on the PL using the same accelerator architecture and evaluated in terms of classification accuracy, generalisation to unseen data, hardware resource utilisation, and the performance on the classification of live received signals in the RFSoC.

## 6.2 Related Work

Early QAT approaches such as Jacob et al. [123] introduced training with quantised 8-bit integer networks using simulated quantisation in the forward pass during training, laying the groundwork for preserving accuracy post-quantisation. Choukroun et al. [124] improved on this by demonstrating QAT precisions down to 4-bits for hardware constrained applications. Both of these approaches demonstrate lower precision weights with floating-point scaling factors, saving on computational resources. More recent work, like Zhao et al. [125], demonstrated QAT using purely integer numbers. Lin et al. [126] also contributed to fixed-point neural networks with power-of-two scaling factors, showing that careful tuning of the quantisation parameters during training retains accuracy.

The application of QAT for low-bit inference has become a critical topic for edge devices, where memory and power are constrained. Zhu et al. [84] introduced trained ternary quantisation, allowing networks to use just two or three weight values without drops in accuracy. Other works have extended QAT to 4-bit or even binary weights, demonstrating trade-offs in model capacity and efficiency [39].

For wireless signal classification, works that implement quantised neural networks through PTQ or QAT are discussed. These works were also compared in Table 5.7 in Chapter 5. Jentzsch et al. [46] has shown that fully quantised models can achieve high throughput on FPGAs for modulation classification through QAT. Tridgell et al. [47] demonstrated low-bit quantised CNNs into a real-time SDR platform. Kumar et al. [127] demonstrated quantised neural network inference for IoT signal detection using QAT to reduce the network precision bit-width, while Hou et al. [42] and Jung et al. [43] both used PTQ for RF signal classification, trading off some accuracy for fast deployment and reduced computational complexity. However, few works explore QAT [46], [47]

for live modulation classification under fixed-point constraints, particularly power-of-two scaling factors, leaving a research gap which this thesis addresses.

Producing an SDR receiver design that is robust to real-world conditions is essential for quantised models in wireless tasks. Tridgell et al. [47] tested their quantised architecture with transmitted signals through RF loopback, showing that careful calibration of the quantised models is essential in retaining accuracy under live conditions. This thesis explores the effects of PTQ and QAT on the DeepRFSoC dataset, as well as real-time received signals transmitted with different modulation schemes to assess how these techniques perform when deployed in real-world situations.

## 6.3 Evaluation Methodology

To evaluate the impact of lower precision weights in the FPGA-based CNN accelerator, introduced in Chapter 4, a structured evaluation methodology was designed. This section outlines the hardware platform, datasets used, quantisation methods, and evaluation criteria used in the study.

### 6.3.1 FPGA Hardware Evaluation Platform

The experiments were conducted on the same AMD Zynq UltraScale+ RFSoC platform as used in earlier chapters, i.e. the AMD RFSoC 2x2 development board. The CNN accelerator used in this chapter is the streaming-based CNN architecture introduced in Chapter 4, optimised for real-time modulation classification. This implementation is deployed on the RFSoC platform using fixed-point weights and activations to perform inference and utilises on-chip memory to minimise latency and resource usage. The focus of this chapter is how the deployed accelerator performs on a stream of live signals from the RF-ADC and receiver pipelines, when different bit-widths are assigned to the weights.

The architecture of the CNN accelerator undergoes several modifications when the weight precision changes. The key components affected by quantisation include the weight representation, inter-layer activation precision, and accumulator sizes, which together influence the accuracy, computational efficiency, and resource utilisation of the design.

**Weight Precision Adjustments** In the CNN accelerator, weights are stored in fixed-point format, and their precision determines the memory footprint

and computational complexity. Higher bit-width weights (e.g. 16-bit) provide a greater numerical accuracy, but require increased memory bandwidth and storage. As the weight precision is reduced (e.g. to 8-bit, 4-bit, or 2-bit), the overall model size decreases, leading to more efficient storage. However, lower precision may introduce quantisation errors, impacting model accuracy. The FPGA resource allocation for weight storage depends on the chosen precision, with lower-bit weights requiring fewer BRAM or URAM blocks.

**Inter-layer Activation Precision** The precision of inter-layer activations directly affects the data transfer and computation between CNN layers. In the proposed architecture, discussed in Chapter 4, activations are represented in a fixed-point manner similar to the weights. Lowering the activation precision reduces the memory bandwidth required for layer-to-layer communications, as the inter-layer signals are stored in a BRAM/URAM buffer prior to being processed for the next layer's calculations. Reducing the activation precision can yield lower BRAM/URAM utilisation. However, a reduction in activation precision can lead to instability in feature propagation, requiring modifications such as batch normalisation folding for fine-tuned QAT to compensate [128]. For this work, the activation precision is kept constant at 16-bits as this provides a consistent base to assess the effectiveness of lowering the weight precision.

**Accumulator Size Adjustments** During convolution and matrix multiplication operations, intermediate results are accumulated before quantisation. The accumulator bit-width determines how many partial sums can be stored without overflow. In higher-precision weight and activation implementations (e.g. 16-bits), accumulators require a wider bit-width (e.g. 32-bit) to store results with minimal rounding errors. Conversely, when lower precision weights (e.g. 8-bit or 4-bit) are used, the accumulator size can be reduced accordingly, optimising DSP slice utilisation. However, aggressive bit-width reduction may lead to precision degradation and necessitate additional QAT techniques to maintain stability. The accumulator bit-width is directly correlated with the chosen folding factor implemented in each layer's matrix multiplication subsystem. Depending on the chosen number of MAC operations implemented in parallel, the accumulator size can be adjusted to allow for the accumulated output to be representable.

**Streaming-CNN Architecture** Figure 6.1 illustrates an overview of the streaming-based CNN architecture and shows how the three precision parame-

ters discussed relate to the custom architecture and provides an example of the weight, inter-layer activation, and accumulator precisions. At the bottom of the figure, an illustration of the MAC unit is shown, which collates all three precisions together to perform the output calculation.

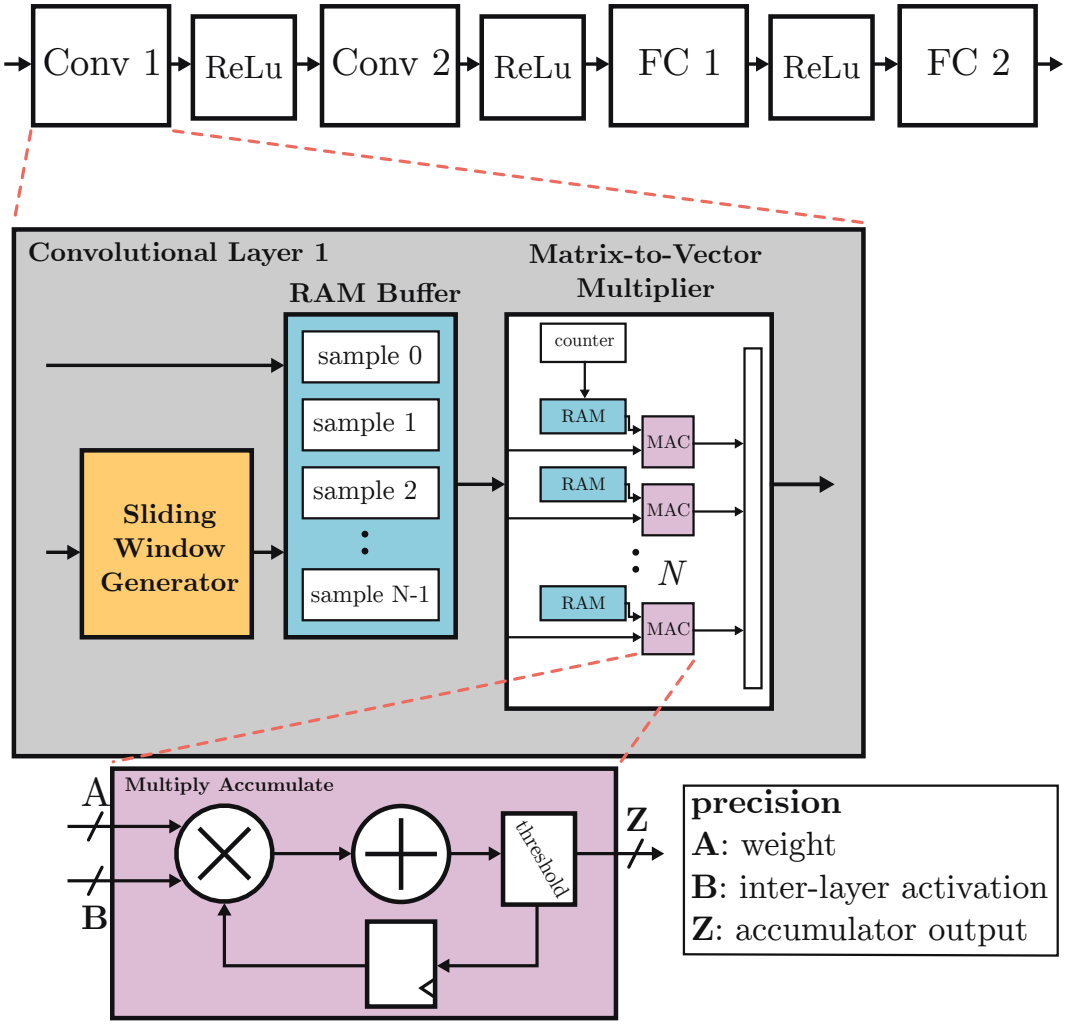



Figure 6.1 Weight, activation, and accumulator precision locations on the streaming-based CNN architecture.

The architecture uses three distinct fixed-point wordlengths for each part of the CNN pipeline:

- **Weights (A):** The weights are stored in on-chip BRAM and accessed by an address counter during MAC operations. Their precision is adjustable between 16-bit, 8-bit, 4-bit, and 2-bit bit-widths, for each separate CNN model implementation. Throughout this chapter, the weight precision is altered for each experiment.

- **Inter-layer activations (B):** These are the signals entering the input of a layer (and from the output of the previous layer's activation function) and are stored at a consistent 16-bit fixed-point precision, across all experiments. A fixed activation precision ensures consistency across experiments when weight precision is varied, isolating the impact of weight quantisation on accuracy.
- **Accumulator output (Z):** The accumulator result from the MAC operations uses a wider bit-width than the input precision to accommodate growth during accumulation and avoid overflow. The optimal accumulation precision is analysed in HDL Coder for each layer configuration.

### 6.3.2 Dataset and Preprocessing

The DeepRFSoC dataset, introduced in Chapter 5, is reused for training and evaluating the lower-precision models. This dataset includes multiple modulation types across a range of SNRs, captured in loopback configured using the RFSoC platform. The preprocessing steps and dataset split ([70:10:20]) remain unchanged.

To evaluate generalisation, the models are also tested on the RadioML dataset, unseen during training, to assess the robustness to different distributions and channel variations. This external benchmark provides an additional validation of the model's performance under quantised inference, especially with lower-precision weights.

The already trained floating-point model weights from Chapter 5 are reused in this chapter's analysis and testing.

### 6.3.3 Quantisation Methodology

To evaluate the impact of lower precision computing on the CNN accelerator, two quantisation strategies were investigated: PTQ and QAT. These methods were applied at varying weight bit-widths, specifically 16-bit, 8-bit, 4-bits, and 2-bits, to assess their effects on classification accuracy, hardware efficiency, and computational performance. Reducing the bit-widths employed by the weights can lead to potential improvements in the CNN accelerator where storage of weights can be exploited.

**Post-Training Quantisation** was performed by quantising a pre-trained, full-precision model. In this approach, weights and activations were mapped

from floating-point to fixed-point representations after training. PTQ is advantageous due to its simplicity and reduced computational cost, as it does not require modifications to the training process. However, at lower bit-widths, the abrupt reduction in numerical precision can introduce significant quantisation errors, leading to accuracy degradation [129]. The extent of this degradation depends on the bit-width used, with higher precision quantisation (e.g. 16-bit and 8-bit) generally preserving accuracy, whereas lower bit widths (4-bit and 2-bit) can induce substantial information loss. The hardware benefit of PTQ is a direct reduction in memory footprint and computational complexity, enabling more efficient FPGA deployment.

**Quantised-Aware Training** integrates quantisation constraints during training, allowing the network to adapt to lower precision arithmetic. During QAT, weight updates are computed in full precision, but quantisation effects are simulated throughout the forward and backward passes. This enables the model to learn robust weight distributions that minimise accuracy loss due to quantisation. Compared to PTQ, QAT is more computationally intensive because it requires training the network under quantisation constraints [85], [86]. However, this added complexity results in significantly better performance at lower bit-widths, as the network learns to compensate for precision loss. QAT was applied to the same bit-width configurations (16-bit, 8-bit, 4-bit, and 2-bit), allowing for a direct comparison with PTQ to determine the most effective approach for FPGA deployment.

By analysing both PTQ and QAT across multiple quantisation levels, this work aims to determine the trade-offs between accuracy, resource utilisation, and real-time performance. The following sections present the experimental results of PTQ and QAT for weights operating on the CNN accelerator during the inference of the test set and live reception of signals on the RFSoC. The implications of lower precision computation are analysed.

### 6.3.4 Evaluation Frameworks

This section details the different evaluation setups used to compare the effectiveness of the lower precision deployed models. Each evaluation scenario assesses the model's performance from software-based testing through to real-time deployment in a live signal reception scenario, providing a comprehensive understanding of accuracy, hardware efficiency and adaptability to an unseen dataset. The following evaluation platforms aim to incrementally show the

performance of the PTQ or QAT weight precision reduction techniques, as the trained model in software is deployed on the CNN accelerator operating on the PL.

### Software Test Set

The first step to verify the effectiveness of a trained model for modulation classification, trained with the DeepRFSoC dataset, is to evaluate the model's performance against a test set. The CNN model was trained using PyTorch similarly to that shown in Chapter 4. The resulting model, whether it was trained using floating-point weights and activations for PTQ, or quantised weights and activations in QAT, was evaluated in the same manner by recording its performance on the test set of the DeepRFSoC dataset.

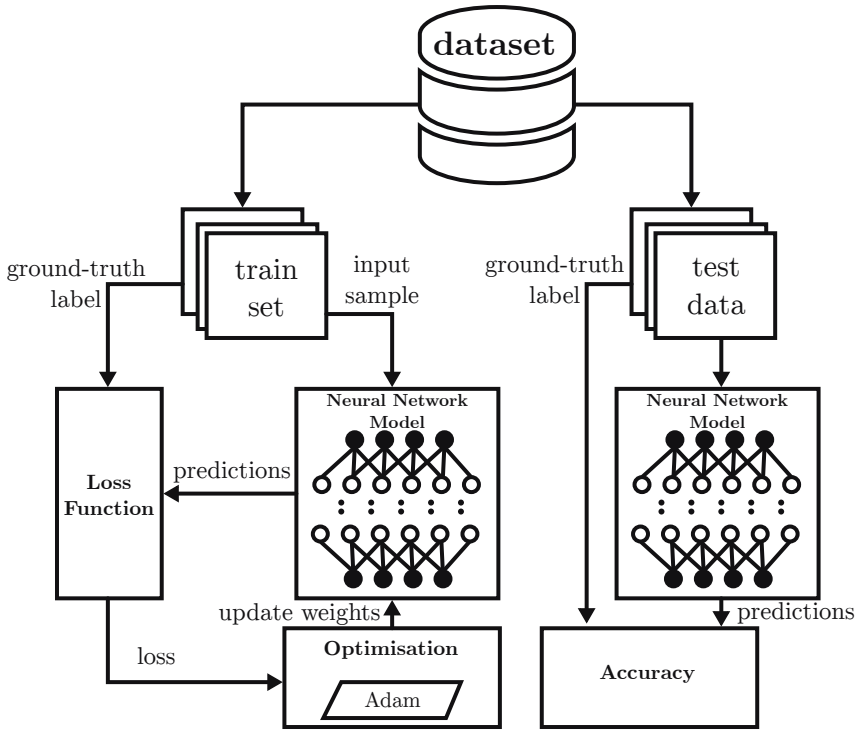



Figure 6.2 Training and testing CNN in software.

Figure 6.2 illustrates the different processes that the CNN model undergoes in software. On the left, the model is initially trained with a dataset to achieve weight values that perform the required task. In this case, the task is to train a modulation classifier that identifies the modulation scheme of the signal received on the input of the model. The dataset, shown at the top of the diagram, is the DeepRFSoC dataset (built through capturing signals with the RFSoC in Chapter 5). A batch of values is extracted and sent to the model for prediction,

where the 'Ground-Truth' label batch is extracted alongside it. The model predicts the class that the signal is encoded with. Through a loss function, in this case a cross-entropy loss function, a loss value is determined based on the 'Ground-Truth' label and the predicted label. The Adam optimiser then determines the weight update step required for the model's weights and the model is adjusted with the aim of reducing the loss function the next time the model is tested with the same batch of data. Once the whole training set has been passed through the model and weights updated based on the optimiser, the process is repeated again for several epochs until the overall loss function no longer reduces.

Once the model has been trained, with floating-point weights, the model is evaluated with the DeepRFSoC test set. This time, instead of the model predicting an answer and updating the weights depending on the output of the loss function and the optimiser, the model predicts the modulation scheme the signal is encoded with, and the overall accuracy of the model is assessed when compared to the 'Ground-Truth' label of the data it is tested with. The accuracy of the floating-point weight model serves as the baseline for comparison with all fixed-point deployment techniques.

The training and testing method detailed in this section and Figure 6.2 is a general methodology used for evaluating the trained software models trained in PyTorch and operating on the PC. It encompasses both the models trained in floating-point (PTQ) and the quantised models trained with QAT, although a more detailed version of this training/testing process for QAT is detailed in Section 6.5.1. The process detailed in Figure 6.2 is the software training/testing process used for evaluating models deployed with PTQ.

#### Test Set Evaluated on CNN Accelerator in PL

Once the accuracies for both the floating-point model and quantised models are evaluated in software after training, the next step is to evaluate the trained weights when they are deployed on the AI accelerator using the fixed-point representation from PTQ and QAT. This evaluation isolates the impact of weight quantisation on inference accuracy, independent of any front-end signal processing effects that will be analysed later in the RFSoC system.

To ensure a controlled evaluation environment, the DeepRFSoC test set is directly fed into the deployed CNN model via an AXI DMA IP core. The overall hardware setup for this evaluation is illustrated in Figure 6.3, which

depicts the data flow from DMA input into the CNN and back to DMA for result retrieval.

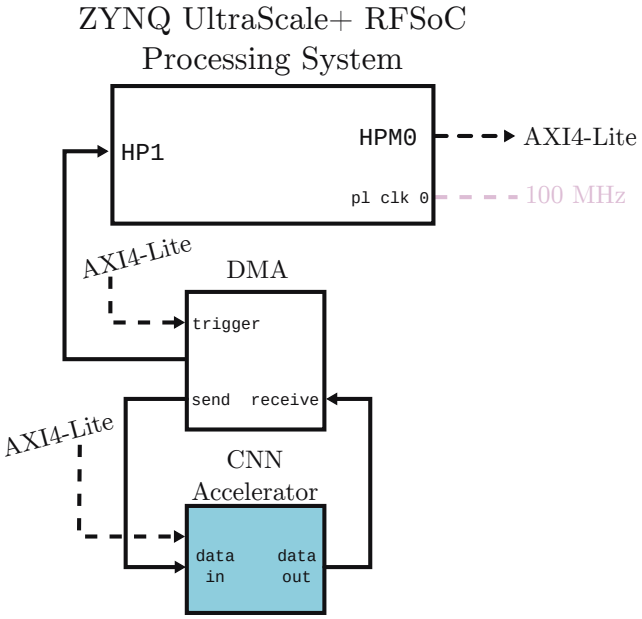



Figure 6.3 Hardware design for testing CNN IP.

The key objectives of this evaluation are:

- To compare the accuracy of PTQ and QAT models against the original floating-point and QAT baseline when executed on the custom AI accelerator in hardware.
- To analyse the impact of different fixed-point quantisation levels on classification performance.
- To validate that the CNN accelerator correctly processes the input using the quantised weight representations without introducing unintended numerical artifacts.

This test strictly evaluates classification accuracy based on the pre-recorded dataset (DeepRFSoC).

### Live Reception of Signals on RFSoC

The ultimate test to verify if the trained weights for both PTQ and QAT operate for the intended application is to evaluate the accuracy and performance of the deployed AI accelerator when receiving signals encoded with different modulation schemes live. This test evaluates the accuracy of the model while

it receives a selection of signals at the ADC. Figure 6.4 illustrates the hardware setup for deploying the CNN accelerator in the receiver pipeline.

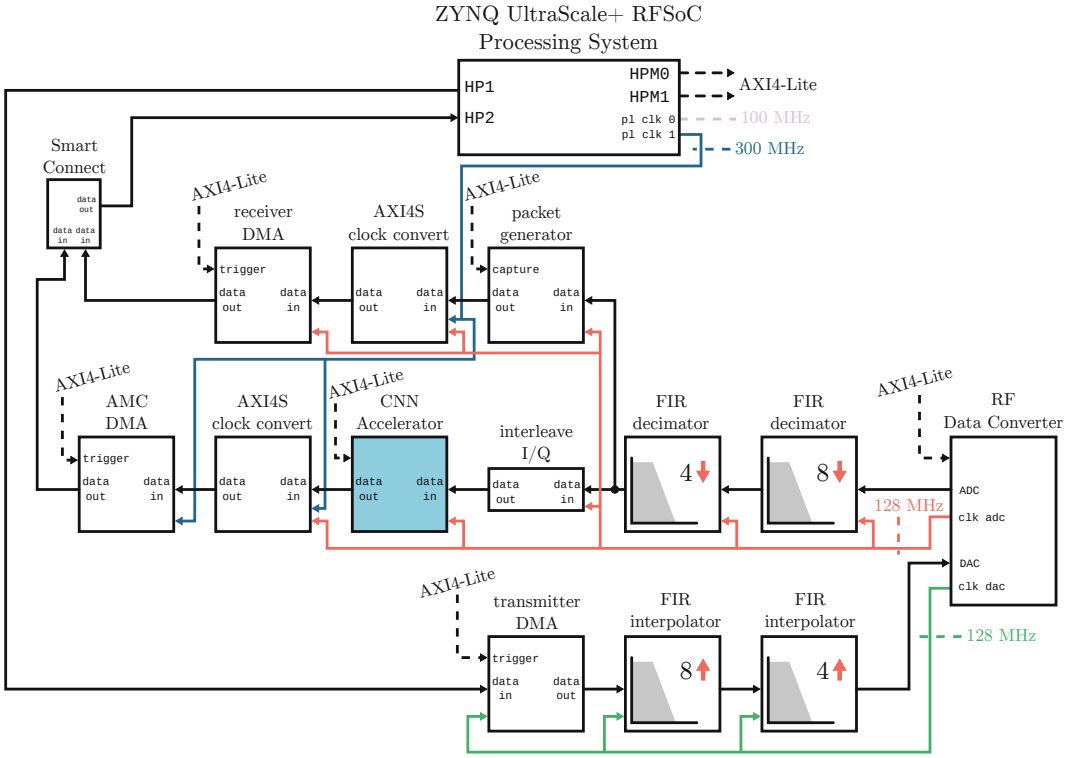



Figure 6.4 Hardware design for testing CNN IP in receiver pipeline.

The same quantised weights used for both PTQ and QAT tests are deployed on the CNN accelerator in the receiver pipeline. Demonstrating the performance of the CNN model in a receiver chain operating on live data is the ultimate test to verifying both the quantisation training techniques as well as the feasibility of the custom CNN accelerator for wireless communication applications. This test transmits and receives signals under the same conditions in which the DeepRFSoC dataset was created.

The desired pre-transmission signals, encoded with different modulation schemes, are stored in PS DDR where a transmitter AXI DMA facilitates a continual cyclic transmission, similar to that undergone in Chapter 5. The hardware-in-the-loop setup used for evaluation is the same as the method described in Chapter 5, where signals are transmitted through the FIR interpolation filter chain, looped back via the RF path, and received through FIR decimation into the AI accelerator. In this section, the deployed accelerator is evaluated under different quantised configurations (PTQ and QAT) using live loopback RF data. Each prediction is transferred back to the PS DDR memory

via AXI DMA IP cores and compared against the 'Ground-Truth' label to compute the classification accuracy.

## 6.4 Post-Training Quantisation (PTQ) Evaluation

The following sections evaluate the training effectiveness of the resulting models, and how well they map to the custom CNN architecture. An incremental evaluation, as detailed in Section 6.3.4, is discussed where an analysis of the model takes place at each stage of the path from custom dataset to deployed hardware AI accelerator operating on real-time signals.

### 6.4.1 Dataset and Training

The CNN model evaluated in this chapter is identical to the one introduced in Chapter 4 and trained as in Chapter 5 using the DeepRFSoC dataset. This dataset was generated using the RFSoC in loopback mode and contains IQ sample frames across a range of SNR levels and modulation types.

In PTQ, evaluating the accuracy of the test set in software is omitted due to the fact that, when performing PTQ, the floating-point weights from the trained model are later converted into fixed-point. In this case, the evaluation of PTQ is performed in the later sections. The model's accuracy for the task of modulation classification is assessed in floating-point when considering the DeepRFSoC test set and later, once the weights have been quantised to the desired fixed-precision, the accuracy of the PTQ models are evaluated against the original floating-point model.

#### Training Configuration and Network Architecture

The training configuration, optimiser, early stopping settings, and network topology used here are the same as those described in Chapter 5. The model consists of two convolutional layers followed by two FC layers, using ReLU activations throughout and Softmax at the output. A summary of the network dimensions is included in Table 6.1 for reference.

The model was trained over 100 epochs with early stopping and an adaptive learning rate schedule. The loss and accuracy curves matched those reported in Chapter 5, with convergence occurring within 20 epochs and no signs of overfitting.

Table 6.1 Neural Network Dimensions.

| **Layer Type** | **Dimensions** | **Activations** | **Output Dim.** | **Parameters** |
|---|---|---|---|---|
| Input | $2 \times 128$ | - | $2 \times 128$ | - |
| Convolution | $64 \times 3 \times 1$ | ReLU | $64 \times 2 \times 126$ | 192 |
| Convolution | $16 \times 3 \times 2$ | ReLU | $16 \times 1 \times 124$ | 6,144 |
| Fully-connected | $1984 \times 128$ | ReLU | $1 \times 128$ | 253,952 |
| Fully-connected | $128 \times 8$ | - | $1 \times 8$ | 1,024 |
| Output | $1 \times 8$ | Softmax | $1 \times 8$ | |

### 6.4.2 Floating-point Model Testing

The final floating-point model was evaluated on the test set to establish a baseline for quantisation-aware evaluation. The test set includes a wide range of SNR values, allowing for the measurement of the overall classification accuracy and the model's robustness to noise and channel impairments.

#### Overall Accuracy across SNR Levels

As shown previously in Chapter 5 (Figure 5.19), the floating-point model achieves strong classification performance, reaching about 81% accuracy at high SNRs and tapering off as SNR decreases.

#### Per-Modulation Accuracy across SNR levels

To further analyse performance, Figure 6.5 illustrates the classification accuracy for each modulation scheme across SNR values. The x-axis represents SNR (dB), while the y-axis shows accuracy (%) for each modulation type. Certain modulation schemes, particularly those with distinct and robust characteristics, maintain high accuracy even at lower SNRs, while others degrade more rapidly. The modulation schemes GFSK, CPFSK, PAM4, and BPSK all perform significantly better compared to QPSK, PSK8, QAM16, and QAM64. These better performing modulation schemes are designed for more robust transmission scenarios such as low bit rate satellite communications, weather balloon data, and long distance signalling [130]. The time and spectral characteristics make it easier for receivers to locate them and decode the signals. The worse performing modulation techniques in terms of classification accuracy are those used for higher data rate applications such as LTE, 5G, and WiFi. While the classification of modulation schemes has little to do with the data rates of the modulation schemes used, the time and spectral characteristics of each

modulation type affects how easy it is to classify each signal. As can be seen in Figure 6.5, the modulation schemes QAM64 and QAM16 are the lowest performing classes in terms of accuracy. The two schemes are similar in form, where QAM64 maps to more constellation points than QAM16. This can mean that determining between the two schemes is more challenging than other types of modulation scheme.

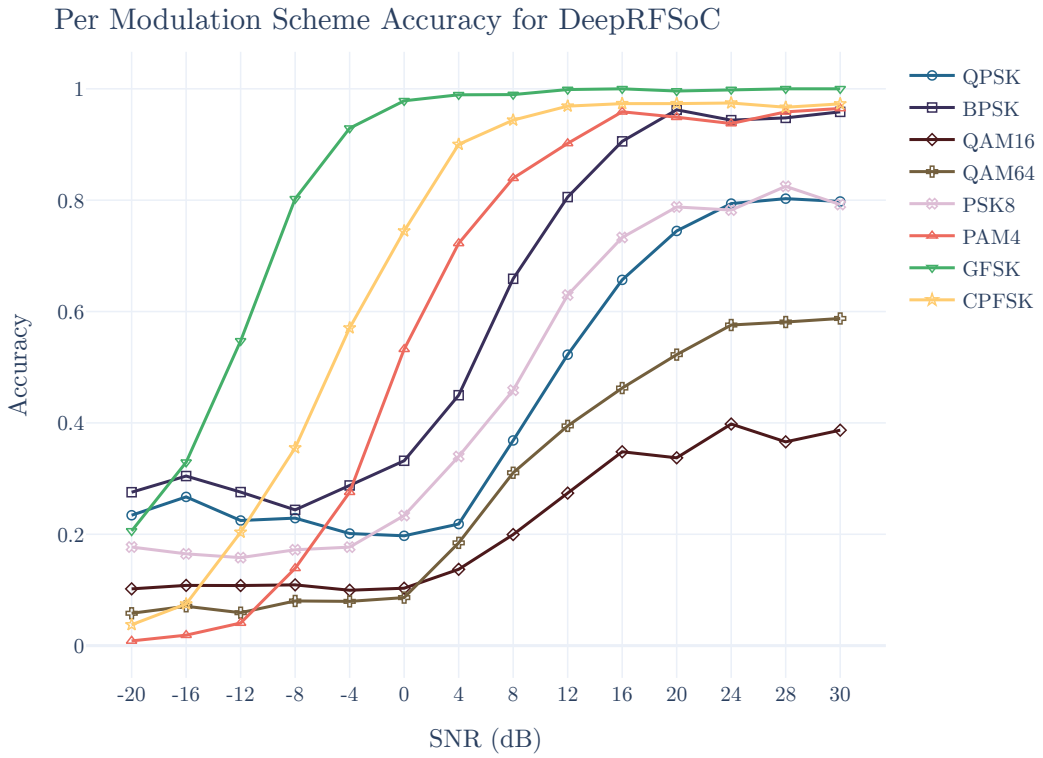



Figure 6.5 Per-modulation accuracy of floating-point model across SNR levels.

These results and those obtained in Chapter 5 serve as a baseline for subsequent quantisation experiments, allowing a direct comparison of how lower precision impacts classification performance. In the next sections, the floating-point model weights will undergo quantisation to 16-bit, 8-bit, 4-bit, and 2-bit fixed-point representations, with 16-bit activations. The effects on accuracy will be evaluated in both stored-data and real-time inference scenarios.

### 6.4.3 Evaluation of PTQ Models with AI Accelerator in PL

PTQ is a technique used to convert the trained full-precision floating point model into a fixed-point representation, typically lowering the bit-width of the model's weights and activations. PTQ is performed after training and does

not involve retraining the network under quantised constraints, making it a computationally efficient method for deploying models to hardware accelerators, such as FPGAs. The objective is to evaluate how well the model can maintain accuracy when reduced to different precision levels.

### Quantisation Procedure

The process begins with the floating-point model, which is first trained using the DeepRFSoC dataset as described in previous sections. Once the model is fully trained and evaluated, the quantisation process is applied to the weights, activations, accumulators, and inter-layer signals.

As stated in previous sections, the only quantisation variable that is altered for each fixed-point model evaluation is the weight quantisations (16-bit, 8-bit, 4-bit, and 2-bit). The activations, accumulator, and inter-layer signals are a consistent bit-width across all evaluation models, with only minor adjustments made to the fixed-point precision, such as the fractional bit location and ratio between integer and fractional bits. Since the activation, accumulator, and inter-layer signal precisions are dependant on the input and weight precisions, the weights are first quantised to the desired bit-width.

Each layer of the CNN accelerator performs a matrix multiplication using a number of MAC units in parallel, while accumulating the result over several clock cycles in an output stationary dataflow optimisation strategy. Each layer holds its own set of parallelised MACs that can operate a multiply-accumulate function for multiple weights and input samples. This means that each MAC can support one type of fixed-point multiplication and addition with a specific set of fixed-point precisions. For example, if a MAC performs a multiply and accumulate for an input sample with fixed-point 16-bit representation of Q2.14 and a weight value with 16-bit representation of Q4.12, then for the next input sample the same MAC will not be able to perform its operation with a Q5.11 precision weight.

The fixed-point value representations in this thesis use the Qm.n format to describe the integer bits (m) and fractional bits (n) (see Appendix A) [131].

With this understanding, the architecture can only support a single fixed-point precision type for each layer, making the quantisation strategy for the CNN model have layer-wise quantisation.

For each bit-width considered, a suitable precision representation must be found for each layer of the network as this will allow the MACs to support the multiplication and accumulation of multiple weights and samples. For the

PTQ technique, each layer's floating-point weights are analysed for the range they cover. A larger range means that more of the available bits are taken up for the integer section, with fewer fractional bits available, and resulting in less precise number representations.

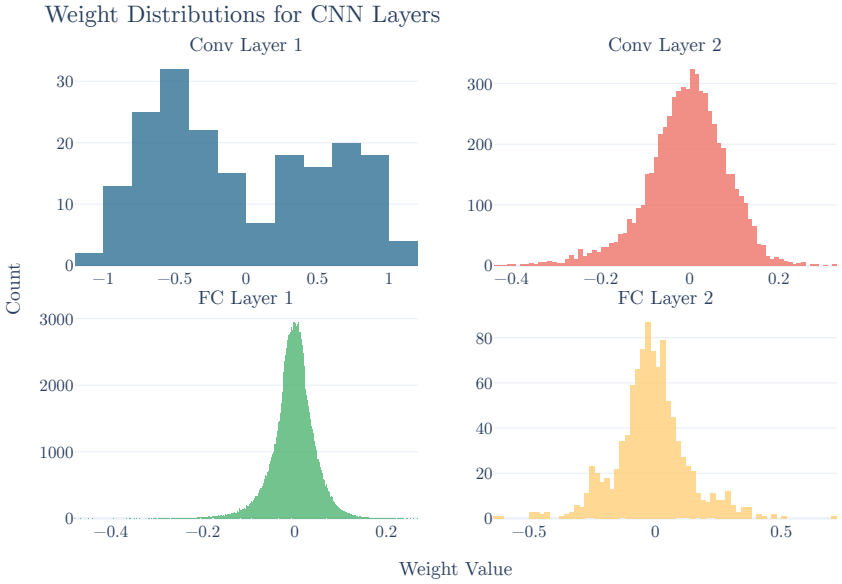



Figure 6.6 Distribution of weight values from floating-point model across four CNN layers. The x-axis represents the weight value, and the y-axis shows the number of weights within each histogram bin.

The histograms in Figure 6.6 shows the distribution of floating-point weight values for each layer of the CNN. Apart from the first convolutional layer, the model layers show a normal-like distribution of values centred around zero. Almost all layer weights have magnitudes below 0.5, except for the first fully connected layer, whose maximum value is about 0.2. With the exception of the first convolutional layer, all other layers have a handful of outlier weight values that fall outside the range of -0.5 to +0.5. Since one fixed-point precision is used per layer, the selected integer-to-fractional bit ratio allocation inevitably sacrifices fractional bits for integer bits to be able to represent these rare outliers, which can result in a degradation in accuracy of the model.

The floating-point weights shown in Figure 6.6 are mapped to each quantised weight precision detailed earlier (16-bit, 8-bit, 4-bit, and 2-bit). For each layer, an appropriate integer and fractional bit ratio is selected and shown in Table 6.2 to support representation of all the floating-point values found in each layer.

For all bit-width cases, the number of integer bits remains consistent across PTQ quantised models, where only the number of fractional bits change depending on how many available bits remain. Figure 6.7 shows the new

Table 6.2 Fixed-point precision mapping to per-layer floating-point weights.

| Fixed-Point Precision (Qm.n) | Layer 1 (Conv1) | Layer 2 (Conv2) | Layer 3 (FC1) | Layer 4 (FC2) |
|---|---|---|---|---|
| signed 16-bit | Q2.14 | Q0.16 | Q0.16 | Q1.15 |
| signed 8-bit | Q2.6 | Q0.8 | Q0.8 | Q1.7 |
| signed 4-bit | Q2.4 | Q0.4 | Q0.4 | Q1.3 |
| signed 2-bit | Q2.0 | Q0.2 | Q0.2 | Q1.1 |

distribution of weight values after quantisation for each of the varying bit-width models.

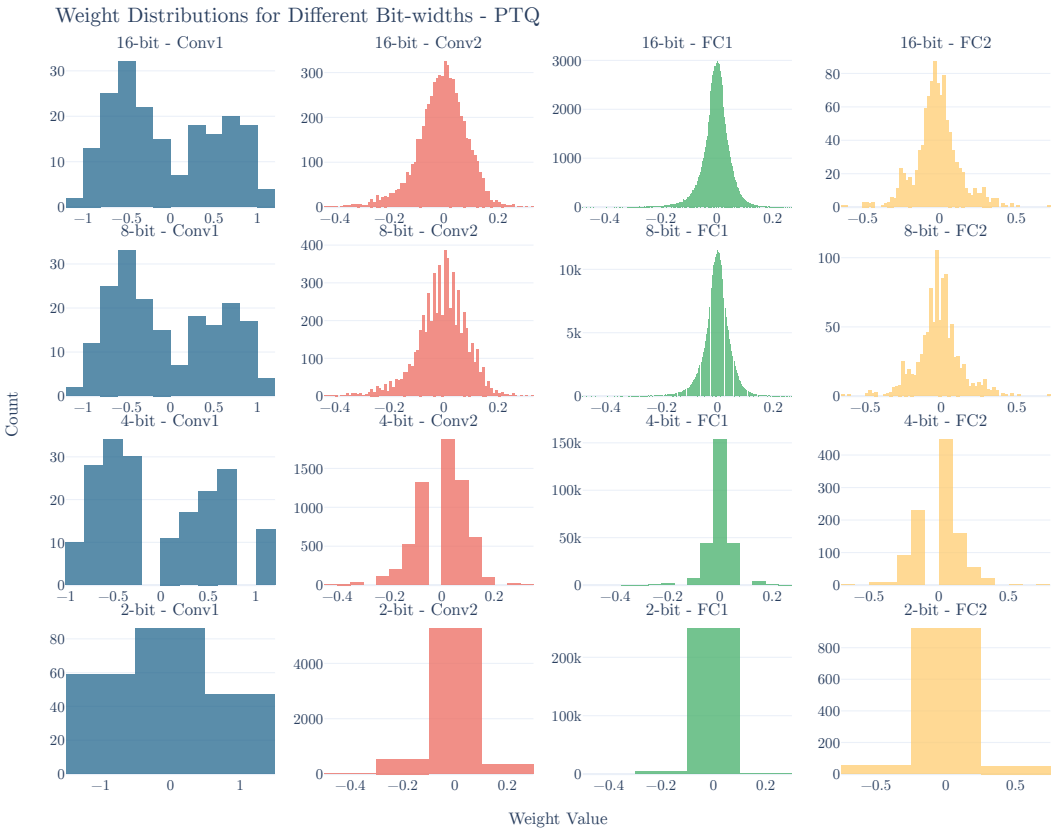



Figure 6.7 Distribution of each fixed-point layer after quantisation.

As the number of bits available to represent the layer weights drops, the weight distributions tend further from those observed in the floating-point model in Figure 6.6. Since the majority of floating-point layer weights follow a normal-like distribution, the majority of values exist near zero. Floating-point precision can accurately represent values near zero comfortably, however, when using fixed-point precision, the number of fractional bits used determines how many of these 'near-zero' values can be represented. In the lower bit-width cases

(4-bit and 2-bit), there are fewer fractional quantisation levels, and more values end up being rounded to zero. This can result in a significant degradation in accuracy.

### Inference Testing on AI Accelerator

To accurately determine the effects of PTQ on the floating-point model, each model with quantised weights is loaded into the custom CNN accelerator and evaluated as described in Section 6.3.4. With the weights loaded in the accelerator, and with knowledge that the incoming signal will be of data type Q2.14, the activations, inter-layer signals, and accumulators precisions can be calculated from the resulting MAC operations. Figure 6.8 shows a simplified dataflow diagram of the candidate CNN, where all of the activations and inter-layer signals are labelled for reference.

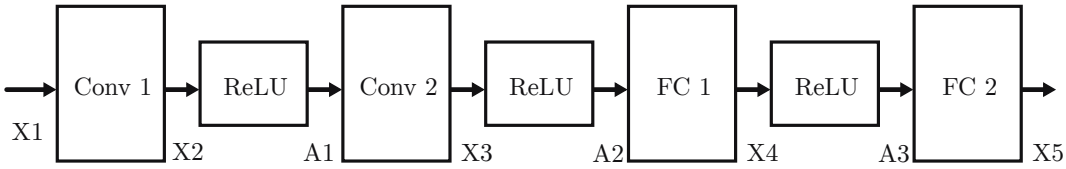



Figure 6.8 CNN model with inter-layer and activation signals labelled.

With the quantised weight values loaded into each custom AI accelerator model, the resulting activations and inter-layer signal precisions were calculated. Table 6.3 maps the labels in Figure 6.8 with the implemented precision.

Table 6.3 Fixed-point precision mapping to inter-layer signals.

| **Signal Name** | **Description** | **16-bit weight Model** | **8-bit weight Model** | **4-bit weight Model** | **2-bit weight Model** |
|---|---|---|---|---|---|
| X1 | Input Data | Q2.14 | Q2.14 | Q2.14 | Q2.14 |
| X2 | Conv1 Output | Q6.24 | Q6.24 | Q6.24 | Q6.24 |
| A1 | Act1 Output | Q3.13 | Q3.13 | Q3.13 | Q3.13 |
| X3 | Conv2 Output | Q11.30 | Q11.30 | Q11.30 | Q11.30 |
| A2 | Act2 Output | Q6.10 | Q6.10 | Q6.10 | Q6.10 |
| X4 | FC1 Output | Q19.38 | Q19.38 | Q19.38 | Q19.38 |
| A3 | Act3 Output | Q7.9 | Q7.9 | Q7.9 | Q7.9 |
| X5 | FC2 Output | Q8.8 | Q8.8 | Q8.8 | Q7.9 |
| X6 | Output Data | float32 | float32 | float32 | float32 |

Since the number of integer bits is fixed, and both the CNN architecture and network dimensions remain unchanged, the bit-width required for inter-layer signals and activations stays consistent across all PTQ models. In other

words, although the weights are quantised to different precisions (e.g. 16-bit, 8-bit, etc.), they still represent values within the same numerical range. As a result, the dynamic range of the resulting activation signals remains similar, requiring similar fixed-point formats. This behaviour is specific to PTQ, other approaches like QAT may result in different signal ranges, requiring re-analysis of the inter-layer signals and activation precision formats.

As described in Section 6.3.4, each custom CNN accelerator configured with the PTQ weights is evaluated against the DeepRFSoC test set and compared against the floating-point model performance. The custom accelerator is implemented into the PL of the RFSoC with an AXI DMA facilitating the transmission of test set frames and the reception of classes. The data sent is identical to the data used to test the floating-point equivalent model.

Figure 6.9 shows the recorded accuracy of each quantised weight model, with PTQ, on the CNN accelerator evaluated against the DeepRFSoC test set. Additionally, Figure 6.10 shows the recorded accuracy of each quantised PTQ model operating on the CNN accelerator evaluated while operating in real-time on live data being sent and received on the RFSoC development board.

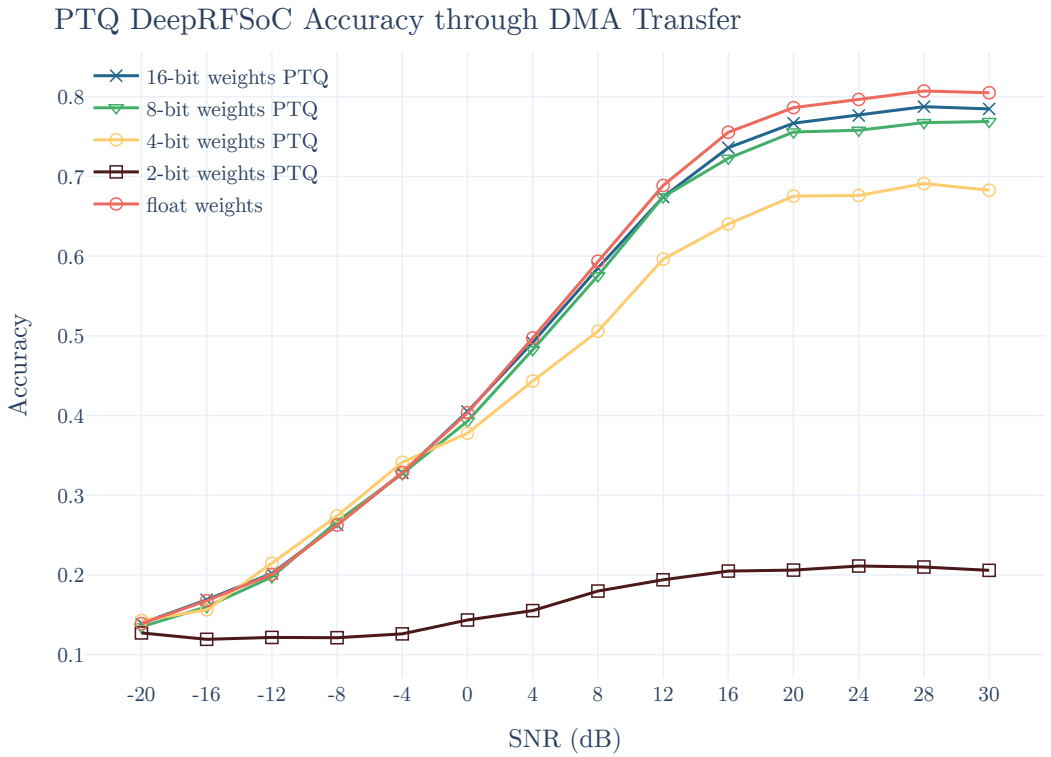



Figure 6.9 Accuracy of PTQ models with DeepRFSoC test set through DMA transfers.

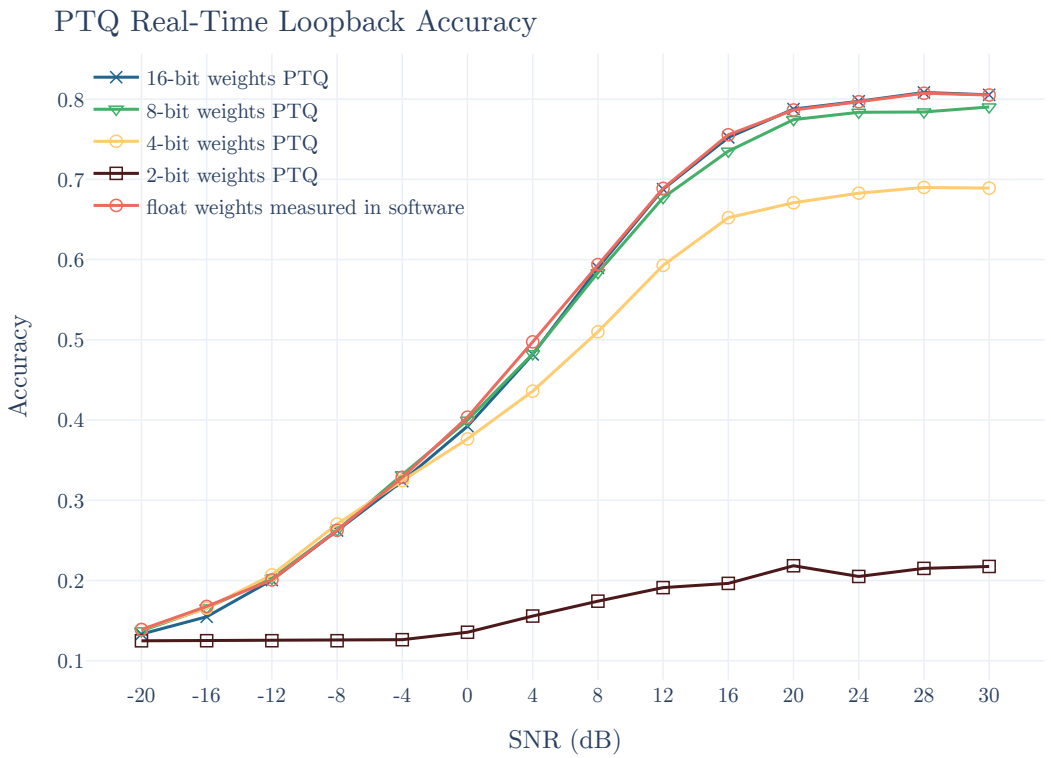



Figure 6.10 Accuracy of PTQ models in real-time RF loopback.

The accuracy results in Figure 6.9 show that the 16-bit quantised model retains most of the original performance, with only a 2% accuracy drop, suggesting that this bit-width is a viable alternative to floating-point inference while offering the capability of being deployed on the custom CNN accelerator. The 8-bit model demonstrates a 4% accuracy reduction, indicating that it remains a practical choice for deployment on resource-constrained FPGA platforms. Similarly, in the real-time RF loopback test case, the 16-bit model achieves equal accuracy performance to the floating-point equivalent model and the 8-bit model shows minimal degradation of 2% accuracy overall. As the bit-width decreases further, accuracy degradation for both test cases becomes more pronounced when quantising with the PTQ technique. The 4-bit model experiences an 11% accuracy loss, highlighting the growing impact of quantisation noise on model inference. At this bit-width, numerical precision constraints start to significantly affect classification performance, suggesting that additional techniques such as QAT may be necessary to maintain usability. The most extreme case is the 2-bit model, which suffers a 60% accuracy drop, making it impractical for real-world applications without further optimisation. This sharp decline indicates that such aggressive quantisation severely limits the model's ability to retain critical information for classification.

These results reinforce the expected trade-off between accuracy and efficiency in quantised neural networks. While 16-bit and 8-bit quantisation offer strong accuracy retention, moving to 4-bit and below introduces substantial performance loss. Selecting an appropriate quantisation level depends on the specific application's constraints, balancing memory savings, computational efficiency, and model accuracy. In the context of real-time radio modulation classification, 16-bit and 8-bit quantisation appear to be the most viable options, whereas 4-bit and lower may require additional optimisation techniques to achieve acceptable performance.

The results observed in Figures 6.9 and 6.10 show similar accuracy performance between the models evaluated against the DeepRFSoC test set and signals received live in real-time on the RF-ADC via RF loopback. The similarity in results shows the effective construction of the DeepRFSoC dataset for training DL models to perform modulation classification with the intention of deploying on an RFSoC board and perform classification in real-time.

## 6.5 Quantisation-Aware Training (QAT) Evaluation

This section will evaluate the training and effectiveness of QAT models, and how well they are mapped to the custom CNN architecture. An incremental evaluation, as detailed in Section 6.3.4, is discussed including an analysis of the model at each stage of the progression from custom dataset training, to deployed hardware AI accelerator operating on real-time signals. This is the second quantisation method discussed in this chapter after PTQ. The two techniques will later be compared in Section 6.6.

### 6.5.1 Dataset Training

In Section 6.4, the DeepRFSoC dataset was used to train a floating-point weight and activation model. The resulting model was evaluated with the DeepRFSoC test set and the model's accuracy performance was established across varying SNR values for all modulation schemes. The floating-point model's weights were then extracted and quantised to one of four selected bit-widths of 16-bit, 8-bit, 4-bit, and 2-bit to achieve a quantised model through the PTQ technique.

To implement the QAT technique, each model is quantised prior to training through quantisation-aware DL layers where the specific quantisation parame-

ters can be configured. In contrast to quantisation with PTQ, where one set of weights can be quantised to all four of the separate bit-widths, QAT requires a separate training process for all four quantised models.

### Quantised-Aware Training Method

In contrast to PTQ, QAT integrates quantisation effects during training by simulating fixed-point arithmetic while maintaining floating-point precision for gradient updates. During forward propagation, weights and activations are quantised to the target fixed-point precision, introducing the effects of reduced numerical representation. During backpropagation, gradients are computed using floating-point precision to preserve learning stability. This process helps the model to adapt to noise from the quantisation steps, resulting in improved accuracy after deployment on fixed-point hardware.

To illustrate this, Figure 6.2 includes the key points where quantisation is applied. This transformation Figure 6.11, which shows the quantised training setup. The weights are clipped to the predefined numerical range before being quantised, ensuring they remain within the representable limits of the fixed-point format. Additionally, during backpropagation, a Straight-Through Estimator (STE) is used to approximate gradients, allowing updates to propagate despite the non-differentiable quantisation operation.

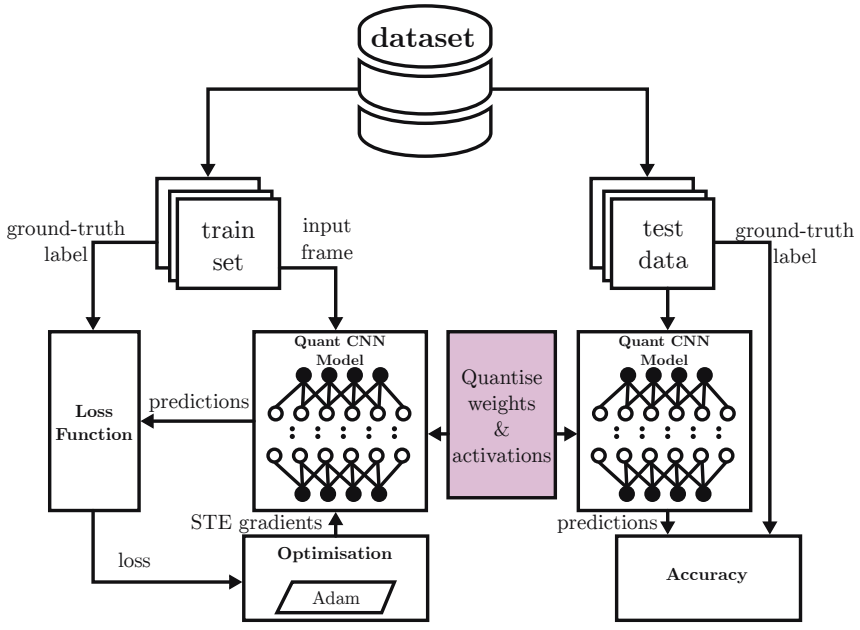



Figure 6.11 Training and testing with QAT for CNN in software.

By incorporating quantisation noise directly into training, QAT reduces accuracy degradation compared to PTQ, making it a suitable choice for ap-

plications where high precision is required post-quantisation. The evaluation of QAT models follows a similar methodology to PTQ, where performance is assessed on the DeepRFSoC test set and later on hardware with fixed-point representations on the custom AI accelerator IP core.

The resulting trained model with quantised weights and activations is evaluated against the test set of the DeepRFSoC dataset. When a forward pass on the model is performed, the learned weights and quantisation parameters are applied and produce a prediction based on the quantised weight values. The accuracy performance of the model is evaluated based on the predictions produced by these quantised weights. Alongside the accuracy results, analysis of the training performance is also performed.

### Training Setup and Configurations

The quantise-aware DL layers were configured using Brevitas, a PyTorch add-on, developed and maintained by AMD, that provides an extensive set of functionalities for lowering precision DL training [132]. The Brevitas software package allows for the specific description of the data precision for many aspects of the neural network including: weight precision and activation precision. Figure 6.12 illustrates a quantised 2D convolutional layer, showing the additional parameters and steps involved in the QAT procedure within the Brevitas framework.

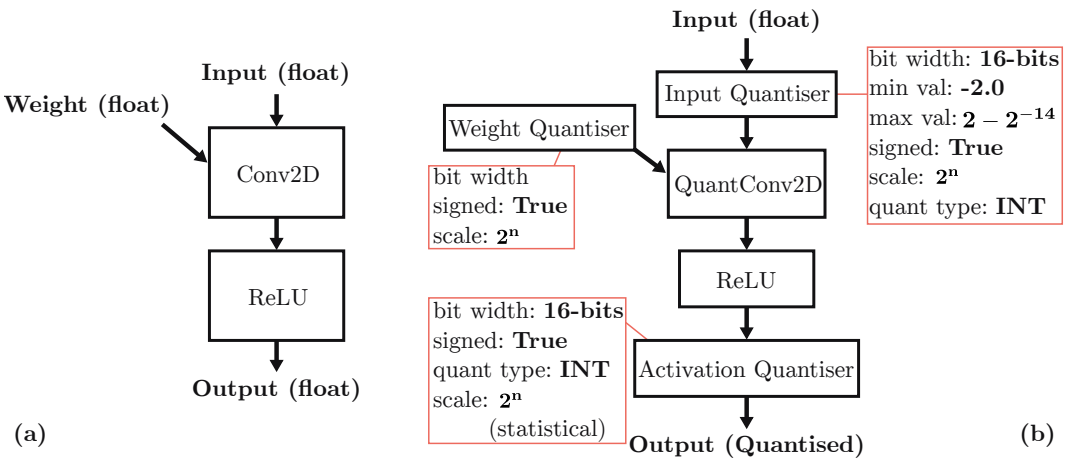



Figure 6.12 (a) floating-point Conv2D layer parameters in PyTorch. (b) Quantisation parameterse for Conv2D layer in Brevitas/PyTorch

To maintain an accurately represented quantised DL model in software, the model is configured to the desired bit-widths for all areas of the model including weights and activations. The model is configured to operate in

fixed-point 2's complement arithmetic. The scaling factor for all data types is limited to a power-of-two value, so that fractional bit selection can be used. The scaling factor is selected during training through the forward-pass and back-propagation steps, so that an optimal value is selected for the given task.

The input is explicitly quantised to a signed fixed-point data type of Q2.14 to ensure that it matches the data type assigned in hardware. The activation signals choose the scaling factors through a series of statistical techniques to determine the most effective value for the DeepRFSoC dataset [132]. Luckily, these configurations are predefined for many common bit-width scenarios and already define the fixed-point arithmetic rules within a large set of 'Quantiser' functions. As an example, the configuration of the 8-bit weight model is shown in Table 6.4, which details the Brevitas layer and quantiser classes used to achieve a fixed-point DL model, with $2^n$ scaling factor, meaning it can be represented by the fractional bits of the fixed-point number.

Table 6.4 Brevitas quantisation parameters for 8-bit weight CNN model.

| **Layer Name** | **Type** | **Bit Width** | **Quantiser** | **Scale** |
|---|---|---|---|---|
| Input | QuantIdentity | 16 | `Int16ActPerTensorFixedPointMinMaxInit` | $2^n$ |
| Conv1 | QuantConv2d | 8 | `Int8WeightPerTensorFixedPoint` | $2^n$ |
| Act1 | QuantReLU | 16 | `Int16ActPerTensorFixedPoint` | $2^n$ |
| Conv2 | QuantConv2d | 8 | `Int8WeightPerTensorFixedPoint` | $2^n$ |
| Act2 | QuantReLU | 16 | `Int16ActPerTensorFixedPoint` | $2^n$ |
| FC1 | QuantLinear | 8 | `Int8WeightPerTensorFixedPoint` | $2^n$ |
| Act3 | QuantReLU | 16 | `Int16ActPerTensorFixedPoint` | $2^n$ |
| FC2 | QuantLinear | 8 | `Int8WeightPerTensorFixedPoint` | $2^n$ |
| Output | QuantLinear | 16 | `Int16ActPerTensorFixedPoint` | $2^n$ |

### Network Architecture

The network dimensions remained the same for QAT evaluation as for PTQ evaluation. The model has four layers consisting of two 2D convolutional layers and two FC layers with ReLU activations between layers. An overview of the dimensions of each model can be seen in Table 6.1. All parameters remain the same as those configured for the floating-point model with additional parameters for achieving quantisation. For each of the candidate bit-widths, the quantiser class used for each layer changes. Table 6.5 shows the different weight quantisers used in each layer for every model.

Table 6.5 Quantisation parameters for different bit-width models.

| Model | Bit Width | Data Type | Precision Type | Scaling | Radix Point | Offset |
|---|---|---|---|---|---|---|
| 16w16a | 16 | Signed Int | Per-Tensor | $2^n$ | Computed | 0 |
| 8w16a | 8 | Signed Int | Per-Tensor | $2^n$ | Computed | 0 |
| 4w16a | 4 | Signed Int | Per-Tensor | $2^n$ | Learned (decoupled) | 0 |
| 2w16a | 2 | Signed Int | Per-Tensor | $2^n$ | Learned (decoupled) | 0 |

Table 6.5 highlights the key quantisation parameters for each candidate bit-width model. The **Bit Width** column refers to the number of bits used for representing weights, where all models use 16-bit activations, while weights vary from 16-bit to 2-bit. The **Data Type** column shows that all weight are represented as signed integers and the **Precision Type** column shows that all quantisers are configured to provide the same data type across the whole tensor/layer. The **Scaling** column indicates that all models used a power-of-two scale factor ($2^n$), ensuring that the resulting scaling factor can be transformed into fractional bit assignment for efficient fixed-point arithmetic hardware. The way the **Radix Point** (or scaling factor) is determined differs based on the bit-width. In the 16-bit and 8-bit models, the radix point is computed from the back-propagation statistics, while for the lower bit width models (4-bit and 2-bit), it is learned independently of the weights (decoupled scaling). This adjustment allows the lower-bit models to maintain accuracy despite reduced precision. Finally, the **Offset** column shows that all models use a zero offset, meaning the quantisation is symmetric around zero. This simplifies hardware implementation by avoiding the need for an additional bias term.

### Training Process

In contrast to the PTQ models, QAT requires each bit-width model to be trained separately so that the appropriate weights are learned for each quantisation limitation applied to the networks. Each model was trained for 100 epochs using a learning rate of $1e^{-4}$, with an adaptive learning rate schedule to refine convergence. The loss function used was cross-entropy and was minimised using the Adam optimiser. The performance of each model was tracked using training and validation loss/accuracy. To ensure proper generalisation and minimal over-fitting, the model's validation loss was monitored, with early stopping applied if the validation loss did not continue to improve. To assess the convergence

of each model, training and validation loss curves were recorded. Figure 6.13 presents these curves, showing that each model achieved stable convergence with minimal over-fitting when trained with the DeepRFSoC dataset.

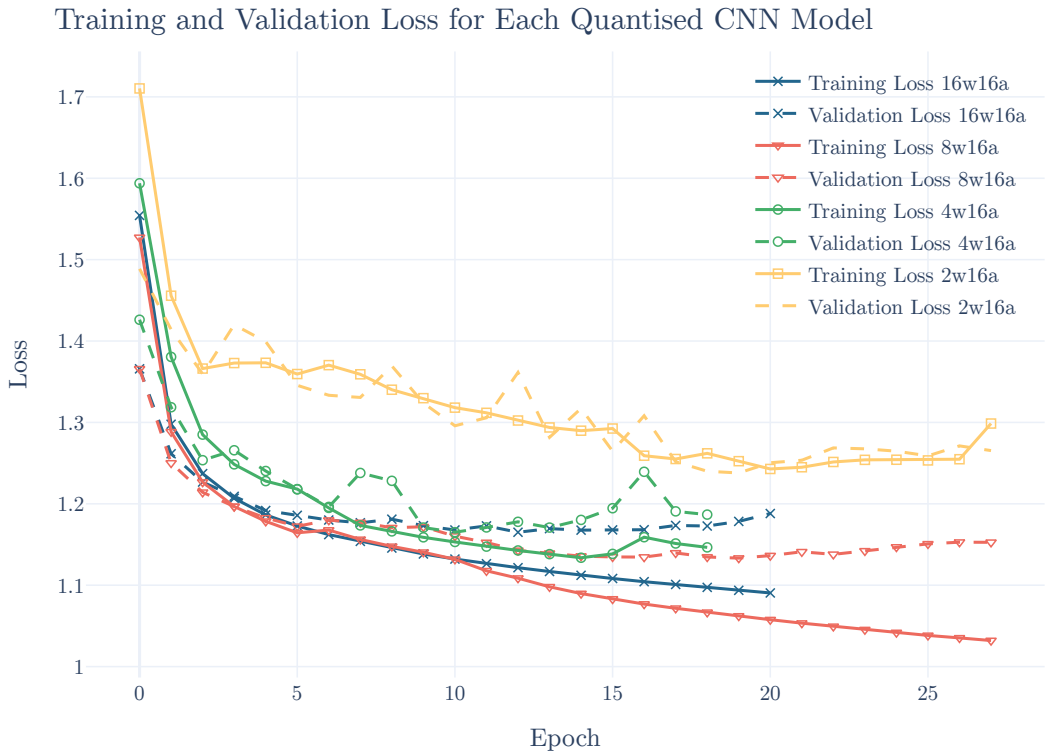



Figure 6.13 Training and validation loss plots for each quantised model training (QAT).

All four quantised models converged and triggered early stopping. Overall, the number of epochs remains comparable to the floating-point trained model which took 20 epochs to converge. The 16-bit model achieved the lowest validation loss at around 1.15, followed by the 8-bit and 4-bit models which were both measured at 1.8. The 2-bit model had the highest loss measured at 1.27; however, the convergence for this model began at a higher loss in the first epoch.

The following sections will evaluate the effectiveness of the trained weights and measure if there are any improvements to the accuracy compared to the PTQ technique for deploying on the custom CNN accelerator.

### 6.5.2 QAT Models Testing

Following the completion of training, each quantised model was evaluated on the DeepRFSoC test set to observe the accuracy performance of the model before it is transferred to the custom CNN accelerator architecture. This evaluation

provides a point of reference to compare the quantised weights, achieved through QAT, with the same weights implemented on the CNN accelerator. The QAT models are also compared against the floating-point model trained in Section 6.4.1. The test set results assess all four model's ability to classify modulation schemes across varying SNR levels.

Figure 6.14 shows the overall accuracy of each QAT model when evaluated with the DeepRFSoC test set. This test was conducted in software within the PyTorch environment to verify the accuracies of the resulting models when the quantisation parameters are applied. The four models' accuracies are plotted with the addition of the floating-point model as a baseline reference. As can be seen, the overall performance for all models is closely comparable to the floating-point model in terms of accuracy across the varying SNR values. The 2-bit model (2w16a) performs the worst due to having the lowest available bit-width, followed by the 4-bit model (4w16a) which exhibits equal performance to the floating-point model trained on the same dataset. The 16-bit (16w16a) and 8-bit (8w16a) models achieve the same accuracy, and both perform slightly better than the floating-point baseline. This may be due to quantisation noise introducing a form of regularisation, which can in some cases assist training. While this is an interesting observation, it is unlikely that fixed-point weights truly outperform floating-point representations in general. Instead, it may indicate that the floating-point model, given the current configuration, is not reaching its maximum potential accuracy. Further investigation would be required to establish whether this behaviour is consistent across architectures or specific to this setup, making it a potential direction for future work.

### Quantised Weight Analysis

In Section 6.4.3, the distribution of weight values from the floating-point model was analysed in order to assess the best quantisation strategy for PTQ. In the case of QAT, the weights are already quantised when the training process has finished; however, it is useful to observe the distribution of weight values when the model has knowledge of the quantisation limitations during training. Each layer of the CNN accelerator performs a matrix-multiplication using a selected number of MAC units in parallel, while accumulating the result over several clock cycles in an output stationary dataflow optimisation strategy. This means that each layer of the model has been configured to quantise its weights on a per-tensor basis, meaning that there is one quantised precision per matrix multiplication or layer.

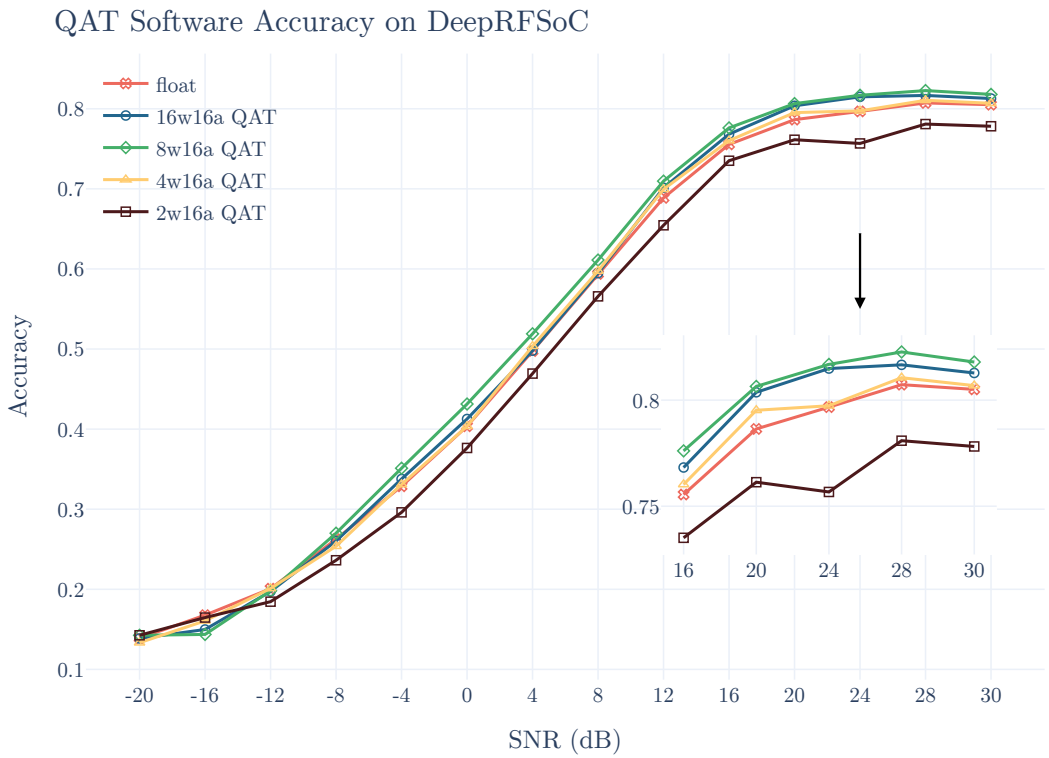



Figure 6.14 Overall accuracy of QAT models evaluated with DeepRFSoC test set in software.

The four QAT models converged to the following learned precisions per layer. Table 6.6 summarises the precision configuration for each model by layer.

Table 6.6 Fixed-point precision mapping to per-layer QAT model weights.

| **Fixed-Point Precision (Qm.n)** | **Layer 1 (Conv1)** | **Layer 2 (Conv2)** | **Layer 3 (FC1)** | **Layer 4 (FC2)** |
|---|---|---|---|---|
| signed 16-bit | Q2.14 | Q1.15 | Q0.16 | Q1.15 |
| signed 8-bit | Q2.6 | Q0.8 | Q0.8 | Q1.7 |
| signed 4-bit | Q1.3 | Q-2.6 | Q-3.7 | Q1.3 |
| signed 2-bit | Q1.1 | Q-2.4 | Q-2.4 | Q2.0 |

The precision values in Table 6.6 differ from those observed in Table 6.2 for PTQ. In the PTQ approach, the number of integer bits assigned to each layer remained consistent across models. By contrast, the QAT models independently determine the optimal precision for each layer based on its learned weights. Consequently, each layer adapts to the most appropriate precision. Notably, some layers adopt configurations where the number of fractional bits exceeds the total bit-width. Although the overall bit-width is unchanged, the radix point

is shifted to enable the representation of even smaller values (see Appendix A for an explanation on the Qm.n format).

After obtaining the learned precisions, it is insightful to examine how these settings affect the distribution of quantised weight values. A series of histogram plots for each model and each layer are presented in Figure 6.15.

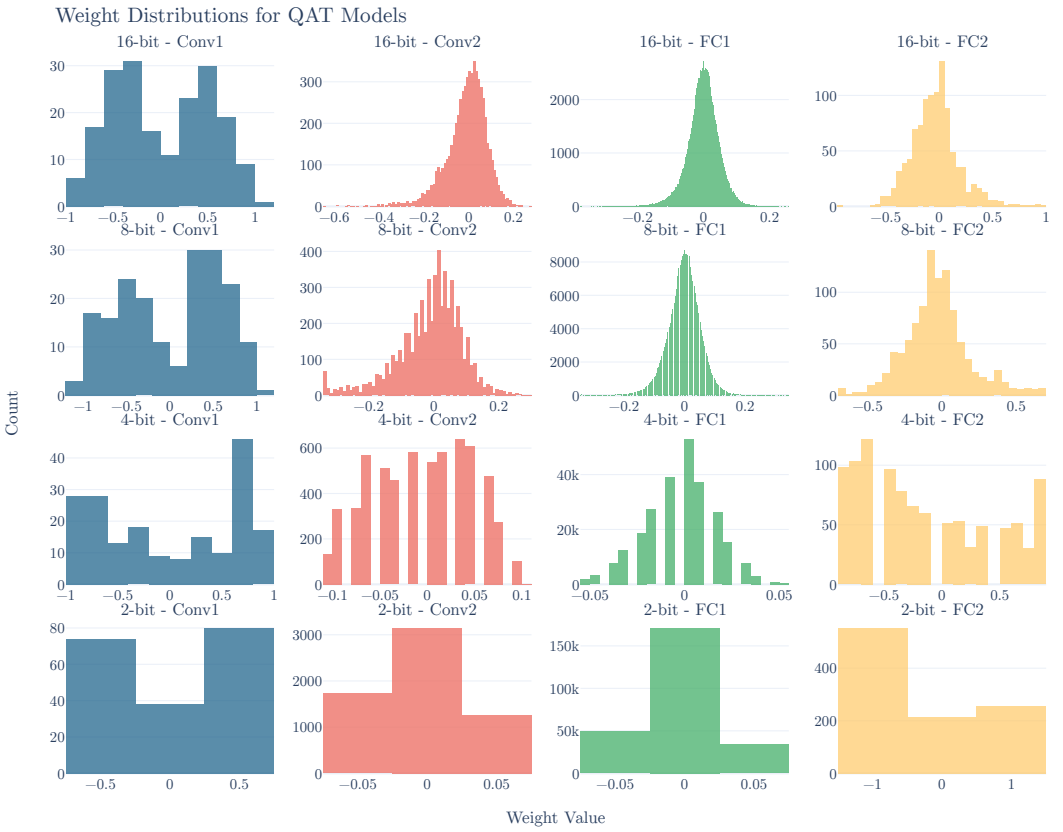



Figure 6.15 Weight distribution for each layer of each QAT model.

While the distribution of weights for higher bit-widths seems consistent with that observed in Figure 6.6, as the bit-width decreases, the distribution of weights changes for each layer. The models have learned to utilise specific, quantisation step values due to the low precision. Specifically, for the final layer of the 4-bit and 2-bit models, the distribution of weights tends away from a normal-like distribution and instead biases towards the representable limits. When comparing this to the quantised floating-point weights, in Figure 6.7, the learned QAT weights adjust for the lack of precision by allocating samples to non-zero values in order to not be overloaded by zero-valued weights and subsequently reduce the effects of accuracy degradation.

### 6.5.3 Evaluation of QAT Models with AI Accelerator in PL

To verify the effectiveness of the trained quantised weights for each QAT model, the weights are loaded into the custom CNN accelerator and evaluated as described in Section 6.3.4. Each model was trained to expect an incoming signal with data type Q2.14. The activations, inter-layer signals, and accumulator precisions were learned through the training process and set within the custom CNN accelerator. The diagram in Figure 6.8, shows a simplified dataflow diagram of the candidate CNN with all of the activations and inter-layer signals labelled for reference. The resulting activations and inter-layer signal precisions were calculated for QAT and are displayed in Table 6.7.

Table 6.7 Fixed-point accumulator and activations precision for each QAT model.

| **Signal Name** | **Description** | **16-bit weight Model** | **8-bit weight Model** | **4-bit weight Model** | **2-bit weight Model** |
|---|---|---|---|---|---|
| X1 | Input Data | Q2.14 | Q2.14 | Q2.14 | Q2.14 |
| X2 | Conv1 Output | Q6.28 | Q6.28 | Q6.28 | Q6.28 |
| A1 | Act1 Output | Q2.14 | Q2.14 | Q2.14 | Q3.13 |
| X3 | Conv2 Output | Q11.30 | Q11.30 | Q11.30 | Q11.30 |
| A2 | Act2 Output | Q2.14 | Q3.13 | Q2.14 | Q2.14 |
| X4 | FC1 Output | Q19.38 | Q19.38 | Q19.38 | Q19.38 |
| A3 | Act3 Output | Q4.12 | Q5.11 | Q3.13 | Q3.13 |
| X5 | FC2 Output | Q11.26 | Q11.26 | Q11.26 | Q11.26 |
| X6 | Output Data | float32 | float32 | float32 | float32 |

Comparing the resulting precisions from QAT in Table 6.7 to the PTQ precisions reported in Table 6.2, the QAT models use a consistent fractional bit-length across all activation layers. In contrast, PTQ show an increasing fractional bit-length as the signal moves deeper into the network. This suggests that QAT helps maintain a more normalised activation range throughout the model. That is an important point, as keeping activations normalised could improve support for adding more layers in the CNN without needing to support increasingly larger layer input values.

In terms of physical implementation performance into the CNN accelerator, no recognisable difference is observed between the model with QAT weights and the model with PTQ weights. For all models (PTQ and QAT), the inter-layer signals and activations maintain the same bit-width across each model where only the radix point is adjusted to be able to represent the resulting signals

produced from the matrix multiplication stages. The precision for the output data type of FC2 is not truncated back down to 16-bits as this output is converted to a 32-bit floating-point value.

As described in Section 6.3.4, each custom CNN accelerator, fitted with the QAT weights, was evaluated with the DeepRFSoC test set on the RFSoC PL and compared to the performance of a floating-point model for reference. Each CNN model was sent frames from the test set and the implemented AI accelerator was evaluated for its accuracy in predicting the correct classes. The test set from DeepRFSoC was identical to that used to evaluate the floating-point model and PTQ weight models.

Figure 6.16 shows the recorded accuracy of each QAT quantised weight model on the CNN accelerator, evaluated against the DeepRFSoC test set. The plot in Figure 6.16 compares each QAT model accuracy against the signal SNR value. The floating-point test set accuracy is used as a baseline comparison to the resulting performance for each quantised model. The considered QAT bit-widths are 16-bit, 8-bit, 4-bit, and 2-bit.

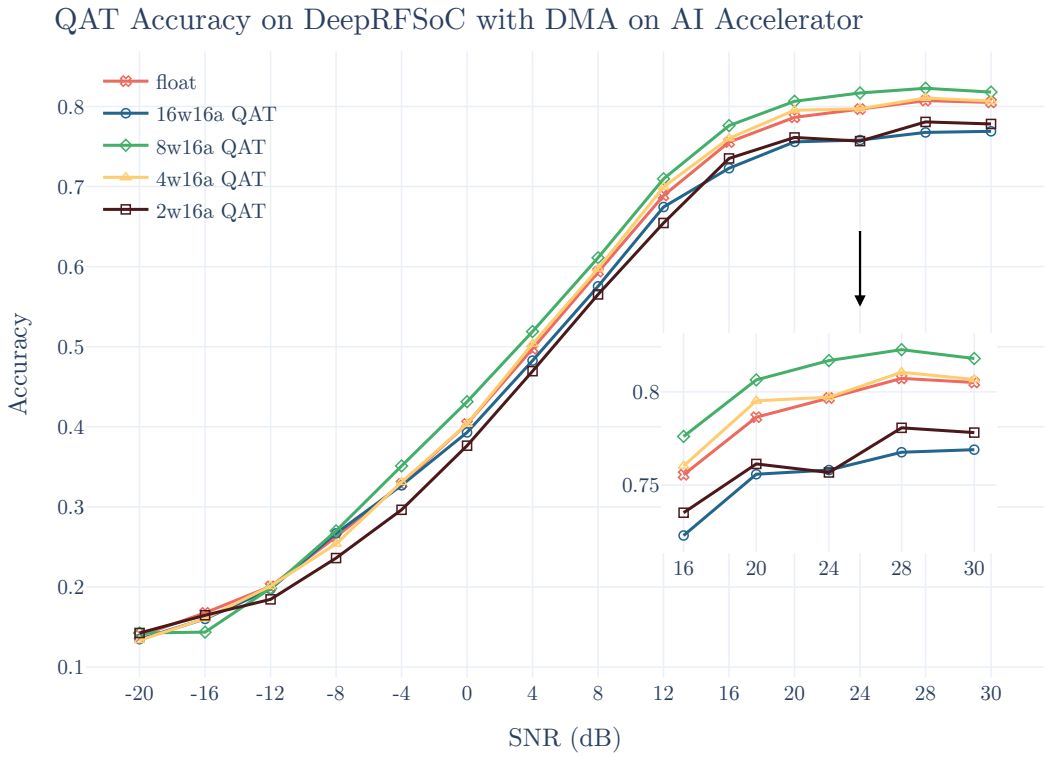



Figure 6.16 Overall accuracy of QAT models evaluated with DeepRFSoC test set in PL through DMA.

Comparing the plot in Figure 6.16, which shows the accuracy of each model when operating on the custom CNN accelerator on the PL, to the QAT models

tested in software, in Figure 6.14, similar results are observed. Firstly, the floating-point model achieves the same accuracy performance as the 4w16a model, further highlighting the effectiveness of training a network with QAT. The 2w16a model performs a little worse, with around a 2% accuracy drop. The 8w16a model again performs the best out of all the QAT models, further reinforcing that adding quantisation noise to the training process seems to provide a level of regularisation that has assisted the training process to converge on a more effective model. Furthermore, the 8w16a model shows that a weight size of 8-bits is sufficient for achieve a model with accuracies better than its floating-point counterpart.

To further evaluate the QAT weights of each model in the CNN accelerator, each model was evaluated in terms of its effectiveness in classifying the modulation schemes when operating in real-time on live signals. As described in Section 6.3.4, the QAT weights were transferred to the deployed CNN model and then evaluated in a real-time scenario by classifying the signals received from the RF-ADC. The signals sent from the RF-DAC and received into the RF-ADC were of the same configuration and parameters as those used to record the DeepRFSoC dataset.

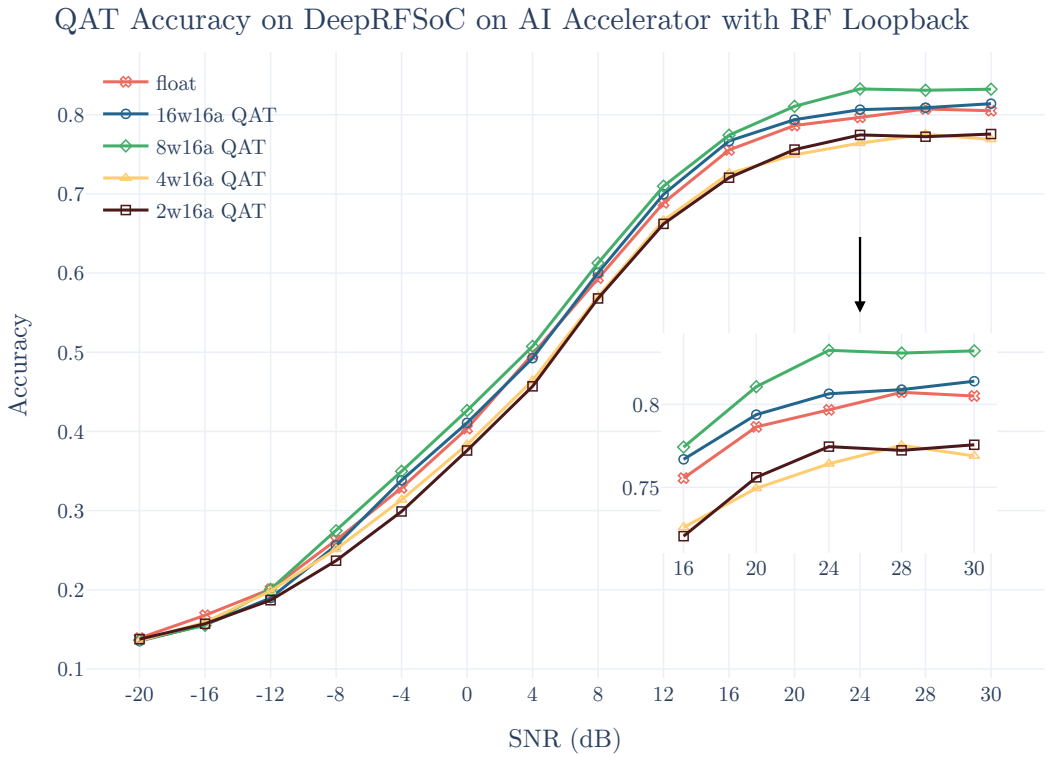



Figure 6.17 Overall accuracy of QAT models on real-time signals received from the RF-ADC.

Figure 6.17 presents the accuracy of QAT models when deployed on the CNN accelerator and classifying real-time signals from the RF-ADC. The observed trends largely align with the other results (from Figures 6.14 and 6.16), where the QAT models were evaluated against the DeepRFSoC test set.

A key observation is that the 8w16a model achieves the highest classification accuracy, surpassing the floating-point baseline by 3%. This indicates that quantisation with 8-bit weights and 16-bit activations not only preserves the network's discriminative capability, but also enhances performance in real-time inference. The 16w16a model maintains equal performance with the floating-point baseline, reaffirming that full 16-bit quantisation does not degrade accuracy in this deployment scenario.

On the other hand, lower-bit precision models such as 4w16a and 2w16a exhibit a slight accuracy reduction of approximately 3%, suggesting that, at these lower weight bit-widths, some loss of representation occurs. Despite this minor degradation, these models still demonstrate strong classification performance, making them viable candidates for edge deployment where lower precision applications are paramount.

These findings reinforce the viability of lower-precision quantisation for real-time modulation classification on the AMD RFSoC platform. Achieving accuracy on par with, or even surpassing, a floating-point baseline while running efficiently on dedicated CNN acceleration hardware highlights the effectiveness of QAT for edge inference applications. Moreover, the real-time operation of the CNN accelerator on RF signals further demonstrates its potential for practical deployment in wireless communications systems.

## 6.6 Comparison of PTQ and QAT

Quantisation is a crucial step in optimising DL models for efficient deployment on FPGA-based systems such as the AMD RFSoC. This section compares the performance of PTQ and QAT in the context of real-time modulation classification. A comparison between the two techniques is summarised in Table 6.8. The results presented in the previous sections indicate that QAT consistently achieves higher accuracy compared to PTQ. In contrast, PTQ offers a faster and more straightforward implementation process since it does not require training from scratch with quantised constraints. This trade-off between accuracy and implementation efficiency is critical when deploying

models on resource constrained edge devices for applications where retraining models consistently is a factor.

Table 6.8 Comparison of PTQ and QAT

| | **Post-Training Quantisation (PTQ)** | **Quantised-Aware Training (QAT)** |
|---|---|---|
| **Workflow** | Quantise a pretrained floating-point model without modifying the training process | Incorporates quantisation effects into the training process so the model adapts to low-precision weights |
| **Training Cost** | No additional training required | Requires training from scratch |
| **Accuracy** | Acceptable at higher precisions (e.g. 8-bit) . Degrades quickly at lower precisions (4-bit, 2-bit) | Maintains higher accuracy across all precisions, particularly at low-bit |
| **Complexity** | Simple to implement and deploy | More complex, requires changes to training process |
| **Use Case** | Suitable for rapid development or when training resources are limited | Suitable when accuracy is critical and sufficient training resources are available |

### 6.6.1 Performance Comparison

The evaluation results show that QAT-trained models outperform PTQ models when deployed on the custom AI accelerator and tasked with classifying live signals received from the RF-ADC. Specifically, the 8w16a QAT model surpassed the floating-point baseline by 3%, whereas PTQ models exhibited reduced accuracy. The accuracy degradation in PTQ suggests that applying quantisation post-training leads to a loss of representational fidelity, particularly in lower-bit precision models such as 4w16a and 2w16a.

### 6.6.2 Performance of DeepRFSoC Trained Models on RadioML Dataset

To evaluate how well the quantised models generalise, it is useful to assess their performance on a dataset they were not trained on. In this case, the RadioML 2016.10a dataset was used to test the PTQ and QAT models that were originally trained using the DeepRFSoC dataset. This helps determine how transferable the learned features are across datasets that represent similar modulation classification tasks, but differ in generation method and channel characteristics.

Figures 6.18a and 6.18b show the classification accuracy of the deployed PTQ and QAT models when tested on the RadioML dataset. Although the models were not trained on RadioML, this evaluation offers insight into the robustness of quantised models when exposed to new signal conditions.

For the PTQ models, a similar trend is observed to that previously seen for the DeepRFSoC test set, where accuracy drops as the bit-width of the weights is reduced. The 2w16a PTQ model fails to classify any signals effectively, highlighting the negative impact of extreme quantisation when not accounted for during training.

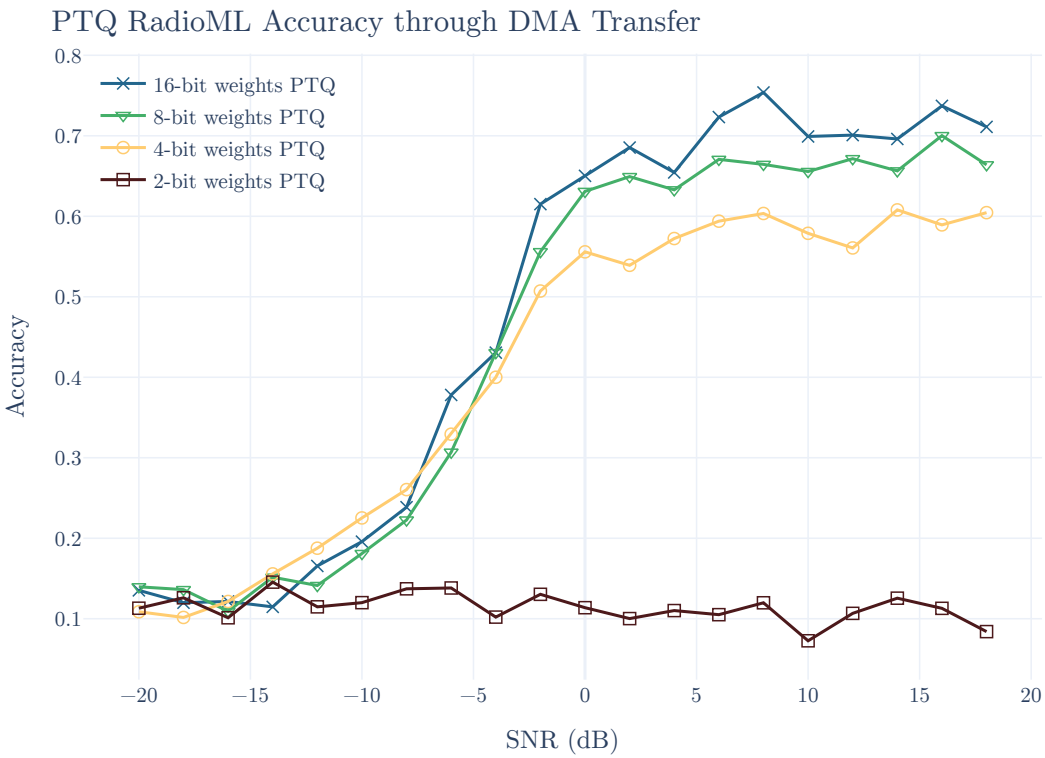
In contrast, several of the QAT models retain more consistent performance. The 8w16a and 4w16a QAT models outperform their PTQ counterparts, and surprisingly, the 2w16a QAT model shows stronger accuracy than both the 4w16a and even the 16w16a QAT model. This suggests that the QAT process is able to learn robust features even with heavily quantised weights, and in this case, may have discovered a more general representation that transfers better to the RadioML dataset.

### 6.6.3 Implementation Trade-offs

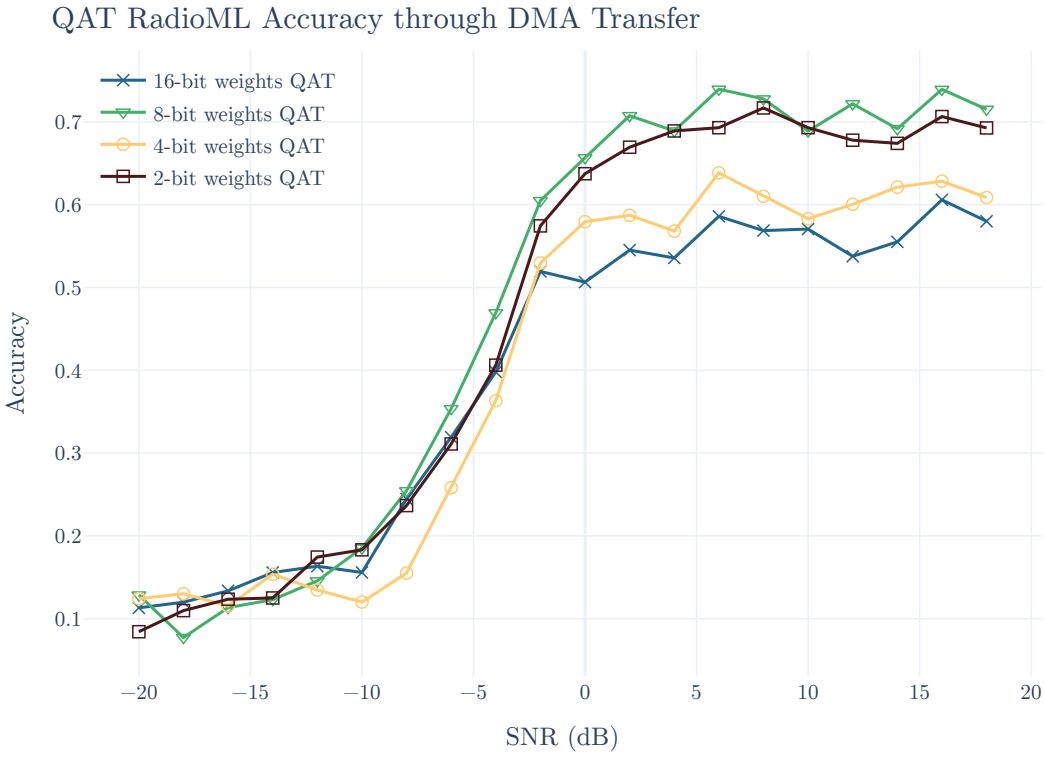
One of the primary advantages of PTQ is its ease of implementation. Unlike QAT, which requires training each model separately with quantisation constraints, PTQ applies quantisation after training. For the case where multiple quantised models are built, using PTQ significantly reduces computational training overhead involved with implementing each quantised model. PTQ requires training once, while each QAT model is trained separately. This makes PTQ a viable option when rapid deployment is necessary, such as in scenarios where training large models is impractical. However, the performance gap observed in real-time classification tasks suggests that QAT remains the superior choice for applications requiring high classification accuracy.

### 6.6.4 PL Resource Utilisation

Table 6.9 presents the PL resource usage for each quantised version of each CNN model implemented on the RFSoC platform, which applies to both the QAT and PTQ quantised models. The breakdown includes logic utilisation (LUTs and registers), DSP slices, BRAMs, and URAMs, and is separated by layer to highlight how different stages of the network contribute to overall resource consumption.




(a) PTQ models




(b) QAT models

Figure 6.18 Overall accuracy of deployed PTQ (a) and QAT (b) models on the RadioML test set while trained on DeepRFSoC.

Table 6.9 FPGA resource utilisation of each quantised CNN model.

| **Model** *layer* | **Slice (LUTs)** | **Slice (Register)** | **DSPs** | **BRAMs** | **URAMs** |
|---|---|---|---|---|---|
| **16w16a** | **34,765** | **58,517** | **456** | **169** | **1** |
| *conv1* | 4,097 | 4,879 | 64 | 0 | 1 |
| *conv2* | 18,261 | 282,83 | 256 | 32 | 0 |
| *fc1* | 11,491 | 24,554 | 128 | 136 | 0 |
| *fc2* | 662 | 534 | 8 | 0 | 0 |
| **8w16a** | **32,067** | **52,950** | **456** | **105** | **1** |
| *conv1* | 5,618 | 6,152 | 64 | 0 | 1 |
| *conv2* | 15,047 | 22,552 | 256 | 32 | 0 |
| *fc1* | 10,217 | 22,873 | 128 | 72 | 0 |
| *fc2* | 698 | 1001 | 8 | 0 | 0 |
| **4w16a** | **30,871** | **51,350** | **456** | **104** | **1** |
| *conv1* | 5,315 | 5,740 | 64 | 0 | 1 |
| *conv2* | 14,079 | 20,443 | 256 | 32 | 0 |
| *fc1* | 10,432 | 23,895 | 128 | 71 | 0 |
| *fc2* | 602 | 909 | 8 | 0 | 0 |
| **2w16a** | **28,717** | **48,286** | **456** | **82.5** | **1** |
| *conv1* | 5,002 | 5,367 | 64 | 0 | 1 |
| *conv2* | 12,683 | 18,251 | 256 | 32 | 0 |
| *fc1* | 10,026 | 23,496 | 128 | 49.5 | 0 |
| *fc2* | 528 | 821 | 8 | 0 | 0 |

Across all models, the DSP usage remains constant at 456 slices. This is because the number of MAC operations and layer-level parallelism is kept consistent regardless of weight precision. Each convolutional and FC layer uses a fixed number of parallel MACs (64, 256, 128, and 8 respectively) for all quantised model variation, which sums to the total DSP consumption.

The BRAM usage across the quantised models is largely driven by the memory demands of the first fully-connected (FC1) layer, which contains the majority of the model's weights. At 16-bit precision, FC1 alone requires 128 BRAMs to store the weights. As the weight precision is reduced, the BRAM usage for FC1 drops accordingly: 64 for 8-bit, 63 for 4-bit, and 41.5 for 2-bit weights. FC1 is the only layer in the network whose weights are stored in BRAM. The remaining BRAMs are used to store the incoming layer signals into buffers.

Notably, the BRAM usage for the 4-bit model is nearly identical to the 8-bit model. This is due to HDL Coder's default behaviour of storing data in

byte-aligned memory, meaning that the 4-bit weights are padded to 8 bits. The expected savings of half the BRAMs are not realised in this case. For the 2-bit model, HDL Coder is able to infer and apply memory packing optimisations, allowing weights to be packed more densely within each BRAM, resulting in a further reduction in utilisation.

For the remaining layers, all weights are stored in distributed LUT RAM. Between the different quantised layers, LUT utilisation also shows a consistent reduction as bit-widths decrease, dropping by roughly 2,000 LUTs between each quantised configuration.

Overall, these results show that while weight quantisation offers substantial savings in logic and memory resources, the extent of the benefit depends on the synthesis toolchain and memory alignment constraints. In practice, quantisation strategies and the synthesis toolchain are interdependent, and both must be considered when evaluating efficiency.

In addition to logic and BRAM utilisation, it is also important to assess the potential memory savings if weights were stored using efficient bit-packing. Table 6.10 presents the theoretical memory required to store the quantised weights in each layer, assuming no byte-alignment or padding is introduced during implementation. These values represent the best-case scenario for memory consumption with the proposed architecture. As the quantisation level decreases, the memory footprint reduces significantly, offering a clear benefit to using lower precision weights. Although current tools such as HDL Coder do not exploit sub-byte packing and instead allocate memory in byte increments, this analysis highlights the untapped savings that could be achieved with more aggressive memory packing techniques. The ideal savings that could be achieved from bit-packing is shown in Table 6.10. This could allow for even more compact implementations and free up additional BRAM for other functions in the system.

Table 6.10 Total memory consumption if bit packing is used to store the weights.

| **Quant Model Type** | **Conv 1 weight memory (Bytes)** | **Conv 2 weight memory (Bytes)** | **FC 1 weight memory (Bytes)** | **FC 2 weight memory (Bytes)** |
|---|---|---|---|---|
| 16w16a | 384 B | 12,288 B | 507,904 B | 2,048 B |
| 8w16a | 192 B | 6,144 B | 253,952 B | 1,024 B |
| 4w16a | 96 B | 3,072 B | 126,976 B | 512 B |
| 2w16a | 48 B | 1,536 B | 63,488 B | 256 B |

### 6.6.5 Relevance to the Streaming-based CNN Architecture

The findings from this comparison directly relate to the custom streaming-based CNN architecture introduced in Chapter 4, designed specifically for FPGA-based radio receiver systems like the RFSoC. Since this architecture is optimised for real-time signal processing, the choice between PTQ and QAT has a significant impact on its effectiveness. While PTQ provides a quicker deployment pathway, QAT aligns more closely with the performance goals of the architecture, ensuring accurate modulation classification while leveraging the CNN accelerator's capabilities.

The analysis demonstrates that while PTQ offers implementation speed and simplicity, QAT provides superior classification accuracy, making it the preferred choice for real-time inference on RFSoC-based systems. Future work could explore hybrid approaches, such as mixed-precision quantisation, to balance accuracy and efficiency for FPGA deployments.

## 6.7 Chapter Conclusion

This chapter evaluated the performance of quantised CNN models for real-time modulation classification using the DeepRFSoC dataset. This chapter focused on PTQ and QAT techniques to reduce the memory footprint while maintaining classification accuracy. Each quantised model was deployed onto the CNN accelerator presented in Chapter 4 and assessed under real-time conditions.

The quantised models demonstrated competitive accuracy compared to their floating-point counter parts, with QAT models generally outperforming PTQ equivalents in both test set evaluation and real-time live signal acquisition on the RFSoC, particularly at lower weight precisions. In some cases, QAT models even outperformed their full-precision counterparts, highlighting that quantisation can introduce beneficial regularisation effects during training. This suggests that the floating-point model may not be achieving its best possible accuracy and that implicit regularisation from quantisation could be compensating. This warrants further investigation as a direction for future work.

Resource utilisation analysis showed that while PL resources uses decreased with lower bit-widths, memory savings were limited due to HDL Coder's byte-

aligned access. A theoretical memory analysis showed significant potential for compression when more efficient packing strategies are used.

Finally, the generalisability of the quantised models was tested using the RadioML dataset. This evaluation confirmed that QAT trained models could also generalise and accommodate a different distribution of samples, further solidifying the effectiveness of incorporating quantisation during training.

Together, these results show that low-precision, real-time CNN classifiers for AMC are not only feasible but effective, and pave the way for intelligent radio systems operating on edge platforms, such as the RFSoC.

# Chapter 7

# Conclusion

This thesis has presented a custom streaming-based CNN architecture designed specifically for SDR receivers, targeting PHY wireless communication tasks that aim to integrate AI. The architecture was introduced using AMC as the demonstration case, highlighting the architecture's ability to perform fast inference on received signal data. The design was extended to support real-time operation on live radio signals, and a practical methodology was proposed for generating DL training datasets that reflect the challenges of real-world hardware considerations. The thesis has also explored the use of reduced-precision weights for the deployed CNN, providing a detailed evaluation of model accuracy and implementation trade-offs, and demonstrating the viability of low-bitwidth deployment in hardware.

The architectures and techniques developed here are designed to be reusable beyond AMC, supporting a wider range of real-time edge AI applications that rely on continuous streaming data in practical deployment settings.

## 7.1 Resume

In summary, the thesis outlined the field of PHY wireless communications and DL, followed by the core contributions from this research work.

Chapter 2 introduced key background material in physical layer wireless communications, covering digital modulation schemes, pulse-shaping, and channel modelling. Modulation classification was presented as the candidate application for this work, alongside traditional and modern techniques, as well as the RadioML dataset by O'Shea et al. [19]. This dataset laid the groundwork for the custom dataset methodology later developed in Chapter 5. The chapter concluded with an overview of the AMD RFSoC platform, the PYNQ embedded

software framework, and common RFSoC design practices, particularly for targeting its RFDC subsystem.

Chapter 3 provided an overview of DL and CNNs, including common training strategies and model optimisation techniques. The chapter also introduced hardware inference accelerators, discussing typical design trade-offs and architectural models such as SDF, relevant to embedded deployments.

Chapter 4 marked the first major contribution of this thesis. It presented a novel streaming-based CNN dataflow architecture for FPGA-based radio receivers. The design aimed to support real-time operation without dropping samples by aligning the architecture with the streaming behaviour of the input signal. The architecture was implemented on the AMD RFSoC and evaluated in terms of classification accuracy, latency, throughput, and hardware utilisation.

Chapter 5 built on this by integrating the CNN accelerator into a full real-time radio system capable of live signal classification. The chapter addressed clock rate requirements due to the overproduction of samples in the convolutional layers, and introduced the need for a dataset representative of real signals captured by the RFSoC. This led to the development of DeepRFSoC (the second contribution of this work), a novel hybrid dataset creation method that combines synthetically generated signals with real-world hardware artefacts introduced by the RF signal chain within the RFSoC. A CNN trained on this dataset was shown to generalise well, even when classifying live signals in a loopback configuration.

Finally, Chapter 6 investigated low-bit precision models for the streaming CNN accelerator. A range of weight precisions were evaluated using both PTQ and QAT. Models were tested in software, on the accelerator using the DeepRFSoC dataset, and in real-time signal classification. Accuracy, resource utilisation, and latency were analysed, demonstrating how precision choices impact both classification performance and hardware efficiency.

## 7.2 Discussion and Key Conclusions

This thesis explored how to build a streaming-compatible CNN accelerator architecture for FPGA-based SDRs, with a focus on real-time PHY inference. Across Chapters 4 to 6, the work addressed the challenges of integrating neural networks into traditional DSP pipelines and proposed both hardware and data-centric solutions.

While NPUs and TPUs deliver high throughput for batch processing, they are not well-suited for continuous data streams in low-latency embedded radio systems. Their general-purpose nature introduces memory bottlenecks and disrupts the streaming dataflow typical of signal processing pipelines. In contrast, this thesis introduced a custom CNN accelerator architecture designed with dataflow in mind, using a SDF model to propagate samples through each network layer as they arrive.

### 7.2.1 Review of Objectives

The research aim of this work was to develop a custom CNN accelerator architecture that can integrate into the streaming pipeline of FPGA-based radio receivers, to support real-world PHY wireless communications applications, while mainly focusing development on the AMD RFSoC. The following sections details the objectives accomplished to realise this aim.

#### Design a Streaming-based CNN Accelerator

This objective aimed to design and implement a CNN accelerator that integrates into a streaming pipeline for an FPGA-based radio receiver and operated at the input data rate without sample loss.

To meet this, a custom accelerator architecture was developed targetting the AMD RFSoC platform. A dataflow-based design was used to allow each layer to process samples as they arrive into the model. The architecture supports real-time operation without dropping the samples streaming from the RF-ADC. A modulation classification application was used to demonstrate the feasibility of the architecture.

#### Training Requirements and Dataset Creation

This objective involved understanding the training requirements for deploying a CNN in real-time wireless environments, which included generating data that was representative of the real-world scenario.

To address this, a custom dataset, DeepRFSoC, was created using live transmissions and loopback on the RFSoC. The dataset incorporated synthetic signals and channel impairments while transmitting and receiving the data using the RFSoC to capture hardware imperfections. This ensured that the model was exposed to deployment-like conditions during training.

The use of DeepRFSoC allowed models to generalise better during deployment, and was critical for evaluating performance under live signal capture.

#### Evaluation of Fixed-Point Quantisation

The final objective was to evaluate the effects of fixed-point quantisation on model performance in deployment.

Both PTQ and QAT were explored. Multiple weight quantisation levels (16-bit, 8-bit, 4-bit, and 2-bit) were implemented on hardware. Results showed that QAT models maintained high accuracy even at low precision, with the 8-bit QAT model outperforming its full-precision counterpart by 3%. These findings confirm that quantised inference is feasible for deployment on constrained platforms like the RFSoC.

### 7.2.2 CNN Accelerator for Streaming-based Applications

Real-time radio receiver pipelines impose strict timing requirements, where every sample must be processed without delay or loss. Many FPGA-based systems address this by adopting a dataflow model, where samples move sample-by-sample from block to block. For DL to be viable in such systems, CNN accelerators must respect this streaming behaviour and operate at line rate without dropping samples.

Chapter 4 introduced a custom CNN accelerator designed around this dataflow constraint. The architecture uses a dataflow model where MACs are parallelised to perform matrix multiplications and a SWG is used to implement convolutional layer to matrix transformations. A hybrid data reuse strategy was applied, combining weight and output stationarity, to maintain a high throughput design without compromising on resource efficiency. A modulation classification application was targetted.

Unlike other FPGA CNN frameworks such as FINN [39] and fpgaConvNet [40], which rely on HLS and static buffering, the proposed architecture was implemented using MATLAB and Simulink with HDL Coder. This allows for seamless integration with the MATLAB and Simulink ecosystem, enabling direct simulation with tools like the 5G and Communications Toolboxes, enabling end-to-end simulation with synthetic channels or RF pipelines.

A CNN model was trained using the RadioML dataset and quantised to 16-bit fixed-point weight. When deployed on the accelerator, the model achieved

comparable accuracy (85% for SNRs over 5 dB) to its floating-point software equivalent, confirming that quantisation and hardware mapping preserved inference performance.

The accelerator operated at 100 MHz and achieved a throughput of 26.5k cps, with a latency of 37.9 $\mu s$. The architecture was capable of a maximum rate of 355 MHz, resulting in a throughput of 94k cps, and a latency of 10.7 $\mu s$. Resource utilisation on the AMD XCZU28DR RFSoC device was: 5.59% LUTs, 5.39% slice registers, 10.67% DSPs, and 15.09% BRAMs. This allows reserving space for integration with other parts of a radio receiver design. These results show that the architecture is well-suited for embedded RF applications.

One design challenge involved sample overproduction at the output of convolutional layers, which required careful clock rate adjustment to maintain a streaming operation. This was addressed in Chapter 5.

### 7.2.3 Real-time CNN Integration with Radio Receiver

Many AI models for PHY wireless communications are trained on synthetic datasets, such as RadioML. While useful for simulation and prototyping, these datasets do not capture critical real-world hardware artefacts, such as: RF-ADC quantisation noise, non-linearities, and RF-ADC tile stitching, that are present during real-world deployment. Without representative training data, CNN models often degrade in performance when moved from simulation to real-world operation.

Chapter 5 introduced a custom dataset, DeepRFSoC. A hybrid dataset that bridges the gap between synthetically generated signals and channel models and the real-world artefacts of SDR systems. The dataset was generated using synthetically generated signals passed through a simulated channel model and then transmitted through the RFSoC in a loopback connection. The signals were captured, after decimation and baseband mixing, and saved to form the DeepRFSoC dataset.

A CNN model trained on DeepRFSoC maintained strong classification accuracy when deployed on the RFSoC while receiving live signals, closely matching the performance of the floating-point baseline.

To achieve a real-time performance, an alternative GeMM transform was presented, where the incoming samples were converted to matrices using the 'channels-first' approach to complement the flow of samples through the model.

Compared to prior work such as Tridgell et al. [47], Jentzsch et al. [46], Hou et al. [42], and Jung et al. [43], the proposed design achieved 81% classification

accuracy above 28 dB SNR with a throughput of 34k cps and a latency of 29.6 $\mu s$, while operating on live data. While some of these designs reported higher throughput and lower latency (such as Tridgell et al. [47] who reported 488k cps throughput and 8$\mu s$ latency), none of them demonstrated real-time inference on live, distorted RF signals using a CNN architecture on the RFSoC or similar embedded SDR platform.

The DeepRFSoC dataset highlights the value of incorporating deployment-specific characteristics during training. By combining synthetically generated data with realistic hardware distortions, the method enables robust model performance under real-time conditions, a requirement for intelligent radios operating at the edge.

### 7.2.4 Low-Precision Weight Optimisation

Chapter 6 explored the impact of lower weight precision on model performance and hardware efficiency, with the goal of compressing the CNN footprint for embedded deployment. Four fixed-point weight precisions were evaluated: 16-bits, 8-bits, 4-bits, and 2-bits. Two quantisation strategies were evaluated, namely PTQ and QAT.

PTQ models exhibited significant performance loss as weight precision decreased. While the 16-bit PTQ model achieved around 80% accuracy (similar performance to the floating-point equivalent), the 2-bit version achieved only $\sim$ 22%. In contrast, QAT models preserved much hight accurcies across all bit-widths. The 8-bit QAT model outperformed the floating-point baseline by $\sim$ 3%, likely due to regularisation effects from quantisation noise. Even at 4-bit and 2-bit precisions, QAT models maintained strong performance, with only a $\sim$ 3% drop at the lowest bit-width.

QAT models also generalised better to unseen data. When evaluated on the RadioML dataset, while trained on DeepRFSoC, AT models outperformed their PTQ equivalents, confirming their robustness under datasets with different distributions of impairment factors.

Hardware utilisation scaled with weight precision. BRAM usage reduced from 168 blocks in the 16-bit model to 82.5 blocks at 2-bits, while LUT usage decreased from 32,765 to 28,717 across the same range. This reduction reflects the more compact storage requirements of lower-precision weights and the smaller memory footprints needed for on-chip input buffer and weight storage. However, due to the fixed MAC parallelism in the architecture, the number of

DSPs remained constant at 456 for all models. DSP allocation depends on the chosen parllelism strategy.

These results confirm that significant reductions in weight precision are possible without a significant reduction in classification accuracy, when the training process uses QAT, which enables highly compressed models to be deployed on resource-constrained platforms like the RFSoC while maintaining real-time performance, assisting in future scalable and efficient intelligent radios.

### 7.2.5 Key Conclusions

This chapter evaluated the full system implementation of a streaming-based CNN accelerator for real-time modulation classification on the AMD RFSoC operating on live real-time signals. It brought together the architecture design, dataset generation, and quantised model deployment. The results show that it is possible to accelerate CNN models using low-precision weights on embedded SDR platforms like the RFSoC without sacrificing on classification accuracy and or dropping samples in real-time operation.

By combining a dataflow-based architecture with a hardware-aware training dataset (DeepRFSoC) and QAT, the system was able to operate in real-time using 16-bit, 8-bit, 4-bit, and even 2-bit weight models. The 8-bit QAT model outperformed the floating-point baseline, and even the 2-bit model maintained usable accuracy, all while maintaining line-rate classification performance.

Overall, these results show that a fully streaming, real-time CNN on RF data is not only feasible, but also effective. The accelerator makes it possible to run CNN inference directly at the edge, processing live signal data as it arrives, without needing to offload data to external processors. This lays the groundwork for future intelligent radios that are lightweight, efficient, and hold the capability of high-throughput AI at the extreme edge for PHY wireless communications.

## 7.3 Limitations and Further Work

This thesis proposed a streaming-based accelerator architecture for a specific CNN topology, which was introduced in the seminal RadioML paper by O'Shea et al. [19]. While the accelerator was built and tested successfully using this topology for real-time modulation classification, the research undertaken for this thesis did not explore a broad range of CNN variants. Future work should aim to generalise the accelerator design process, perhaps through a compilation

framework that can translate a variety of CNN topologies into streaming-compatible hardware. This would enable rapid testing and deployment of more standard models like VGGNet [133] or modern networks with skip connections, such as ResNets [134].

There are also clear opportunities to improve the performance of the existing architecture. One such area is sparsity exploitation, where many of the weights in a trained network are zero. The current accelerator does not take advantage of this, but a sparse-aware implementation could skip unnecessary computations, leading to reduced latency and higher throughput, while also lowering memory requirements. The current design already performs well without leveraging sparsity suggests there is still untapped potential for optimisation which could lead to an inference speed-up in proportion to the number of zero weights that have been trained into the model. Interestingly, the floating-point trained models exhibited a greater proportion of weights at or near zero compared to the QAT models with lower precision quantisation. This observation suggests a potential avenue for future work, namely investigating how sparsity might affect inference acceleration in the two approaches.

The dataset generation method developed in this thesis offers another promising area for refinement. By combining synthetically generated signals with the physical distortions introduced by the RFSoC hardware, it provided a practical way to train models for real-world deployment. However, the approach is time consuming, requiring physical transmission and reception for each new dataset. A possible future direction would be to model the hardware impairments, such as RF-ADC tile stitching or RF front-end non-linearities, within the simulation environment. This could remove the need for live RFSoC transmission entirely, speeding up dataset generation while preserving the hardware-specific characteristics required for model robustness.

In Chapter 6, the accuracy of reduced precision models was explored in depth. Using PTQ, 16-bit weights performed equivalently to floating-point, while with QAT, models using just 4-bit weights matched full-precision accuracy. That said, this analysis did not investigate the absolute accuracy ceiling of the floating-point model. A useful follow-up would be to assess how closely quantised models can approach that upper limit, and whether precision can be reduced even further without meaningful performance loss.

Another avenue for optimisation is the use of LUT-based MAC units for low-precision weights (4-bit and 2-bit), which removes the need to rely on DSP slices. This can lead to more efficient resource usage and potentially

higher clock frequencies. Techniques demonstrated in works such as Andronic et al. [135] and Umuroglu et al. [136] could be integrated into the architecture proposed in this thesis.

Finally, while this work focused on modulation classification as a candidate application, the proposed streaming architecture is not limited to AMC. Other PHY tasks, such as signal detection, demodulation, or spectrum sensing, could also benefit from this accelerator, with required modifications depending on the task, the size of the CNN, and whether the problem is framed as classification or regression. This offers promising directions for future exploration and validation.

## 7.4 Final Remarks

The demand for spectrum access and efficient wireless communication is greater than ever. In an age of rapid AI advancement, the convergence of wireless systems and machine learning is not only inevitable but necessary. Intelligent radios will need to adapt dynamically to their environment, operate flexibly in congested spectrum, and enable more efficient data transmission.

To realise this aim, AI models must operate on the edge, where decisions can be made quickly, locally, and without dependence on remote servers. Bringing capable models to the edge also promotes power efficiency, private, and decentralised AI systems aligning with the global trend toward smarter and more autonomous computing platforms.

This thesis presented an applications-specific CNN accelerator architecture for wireless communications scenarios. While the architecture represents an early stage prototype, it highlights key principles that can influence the development of future accelerators for SDRs. These include how to maintain a continuous stream of samples in real time, how to train models that generalise well to live RF data, and how precision-aware design and quantisation techniques can reduce resource usage without compromising performance.

# References


[1] "Connected Nations 2022: UK Report", Ofcom, Tech. Rep. 2022. [Online]. Available: https://www.ofcom.org.uk/phones-and-broadband/coverage-and-speeds/connected-nations-2022.

[2] "Spectrum Management for Next Generation Wireless Broadband: Flexible access to spectrum sharing", Ofcom, Tech. Rep., 2023. [Online]. Available: https://www.ofcom.org.uk/spectrum/innovative-use-of-spectrum/spectrum-management-for-next-generation-wireless-broadband.

[3] "IoT connections forecast – Ericsson Mobility Report", Ericsson, Tech. Rep. 2025. [Online]. Available: https://www.ericsson.com/en/reports-and-papers/mobility-report/dataforecasts/iot-connections-outlook.

[4] M. Honkala, D. Korpi and J. M. J. Huttunen, "Deeprx: Fully convolutional deep learning receiver", *IEEE Transactions on Wireless Communications*, vol. 20, no. 6, pp. 3925–3940, 2021. DOI: 10.1109/twc.2021.3054520.

[5] M. A. Abdel-Moneim, W. El-Shafai, N. Abdel-Salam, E.-S. M. El-Rabaie and F. E. Abd El-Samie, "A survey of traditional and advanced automatic modulation classification techniques, challenges, and some novel trends", *International Journal of Communication Systems*, vol. 34, no. 10, e4762, 2021. DOI: 10.1002/dac.4762.

[6] N. K. Chauhan and K. Singh, "A Review on Conventional Machine Learning vs Deep Learning", in *2018 International Conference on Computing, Power and Communication Technologies (GUCON)*, 2018, pp. 347–352. DOI: https://doi.org/10.1109/GUCON.2018.8675097.

[7] A. Krizhevsky, I. Sutskever and G. E. Hinton, "Imagenet classification with deep convolutional neural networks", in *Advances in Neural Information Processing Systems*, F. Pereira, C. Burges, L. Bottou and K. Weinberger, Eds., vol. 25, 2012.

[8] M. E. Peters *et al.*, "Deep contextualized word representations", in *2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, M. Walker, H. Ji and A. Stent, Eds., New Orleans, Louisiana: Association for Computational Linguistics, 2018, pp. 2227–2237. DOI: 10.18653/v1/N18-1202.

[9] T. O'Shea and J. Hoydis, "An introduction to deep learning for the physical layer", *IEEE Transactions on Cognitive Communications and Networking*, vol. 3, no. 4, pp. 563–575, 2017. DOI: 10.1109/TCCN.2017.2758370.

[10] L. H. Crockett, D. Northcote and R. W. Stewart, Eds., "Software Defined Radio with Zynq UltraScale+ RFSoC", 1st ed. Glasgow, U.K.: Strathclyde Academic Media, 2023. [Online]. Available: https://www.rfsocbook.com.

[11] N. P. Jouppi *et al.*, "In-Datacenter Performance Analysis of a Tensor Processing Unit", in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, Toronto, ON, Canada, 2017, pp. 1–12. DOI: 10.1145/3079856.3080246.

[12] Y.-H. Chen, T. Krishna, J. S. Emer and V. Sze, "Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks", *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, 2017. DOI: 10.1109/JSSC.2016.2616357.

[13] A. Rico *et al.*, "AMD XDNA NPU in Ryzen AI Processors", *IEEE Micro*, vol. 44, no. 6, pp. 73–82, 2024. DOI: 10.1109/MM.2024.3423692.

[14] "Deep learning", NVIDIA. (2025), [Online]. Available: https://developer.nvidia.com/deep-learning. Accessed: Jul. 22, 2025.

[15] M. Salehi and J. Proakis, "Digital Communications", 5th ed. McGraw-Hill Education, 2008.

[16] J. Browne, "2025's Top Trends: Artificial Intelligence and Machine Learning Bring Smarts to RF Systems", *Microwaves & RF*, 2024.

[17] P. L. T. Mbouembe, G. Liu, J. Sikati, S. C. Kim and J. H. Kim, "An efficient tomato-detection method based on improved YOLOv4-tiny model in complex environment", *Frontiers in Plant Science*, vol. 14, 2023. DOI: 10.3389/fpls.2023.1150958.

[18] A. Romero. "GPT-4's Secret Has Been Revealed". (2023), [Online]. Available: https://www.thealgorithmicbridge.com/p/gpt-4s-secret-has-been-revealed. Accessed: May 6, 2025.

[19] T. J. O'Shea, J. Corgan and T. C. Clancy, "Convolutional radio modulation recognition networks", in *Engineering Applications of Neural Networks*, C. Jayne and L. Iliadis, Eds., Springer International Publishing, 2016, pp. 213–226. DOI: 10.1007/978-3-319-44188-7_16.

[20] W. Shi, J. Cao, Q. Zhang, Y. Li and L. Xu, "Edge computing: Vision and challenges", *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 637–646, 2016. DOI: 10.1109/JIOT.2016.2579198.

[21] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow", *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987. DOI: 10.1109/PROC.1987.13876.

[22] "RFSoC 2x2 Kit", AMD. (2021), [Online]. Available: https://www.amd.com/en/corporate/university-program/aup-boards/rfsoc2x2.html. Accessed: April 19, 2024.

[23] "Communications Toolbox", MathWorks. (2020), [Online]. Available: https://uk.mathworks.com/products/communications.html. Accessed: May 1, 2025.

[24] "GNU Radio: The Free and Open Software Radio Ecosystem", GNU Radio Project. (2025), [Online]. Available: https://www.gnuradio.org. Accessed: May 19, 2025.

[25] I. Goodfellow, Y. Bengio and A. Courville, "Deep Learning". MIT Press, 2016. [Online]. Available: http://www.deeplearningbook.org.

[26] A. L. Ha, T. Van Chien, T. H. Nguyen, W. Choi and V. D. Nguyen, "Deep Learning-Aided 5G Channel Estimation", in *2021 15th International Conference on Ubiquitous Information Management and Communication (IMCOM)*, 2021, pp. 1–7. DOI: 10.1109/IMCOM51814.2021.9377351.

[27] H. Ye, G. Y. Li and B.-H. Juang, "Power of Deep Learning for Channel Estimation and Signal Detection in OFDM Systems", *IEEE Wireless Communications Letters*, vol. 7, no. 1, pp. 114–117, 2018. DOI: 10.1109/LWC.2017.2757490.

[28] M. H. Rahman, M. Shahjalal, M. O. Ali, S. Yoon and Y. M. Jang, "Deep Learning Based Pilot Assisted Channel Estimation for Rician Fading Massive MIMO Uplink Communication System", in *International Conference on Ubiquitous and Future Networks, ICUFN*, vol. 2021-Augus, IEEE Computer Society, 2021, pp. 470–472. DOI: 10.1109/ICUFN49451.2021.9528814.

[29] G. Shen, J. Zhang, A. Marshall, L. Peng and X. Wang, "Radio Frequency Fingerprint Identification for LoRa Using Spectrogram and CNN", in *IEEE Conference on Computer Communications (INFOCOM)*, 2021, pp. 1–10. DOI: 10.1109/INFOCOM42981.2021.9488793.

[30] J. Zhang, R. Woods, M. Sandell, M. Valkama, A. Marshall and J. Cavallaro, "Radio Frequency Fingerprint Identification for Narrowband Systems, Modelling and Classification", *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 3974–3987, 2021. DOI: 10.1109/TIFS.2021.3088008.

[31] A. Saeif, S. Savio and O. Gabriele, "The Day-After-Tomorrow: On the Performance of Radio Fingerprinting over Time", in *Proceedings of the 39th Annual Computer Security Applications Conference*, Association for Computing Machinery, 2023, pp. 439–450. DOI: 10.1145/3627106.3627192.

[32] R. Xie, W. Xu, J. Yu, A. Hu, D. W. K. Ng and A. L. Swindlehurst, "Disentangled Representation Learning for RF Fingerprint Extraction Under Unknown Channel Statistics", *IEEE Transactions on Communications*, vol. 71, no. 7, pp. 3946–3962, 2023. DOI: 10.1109/TCOMM.2023.3268286.

[33] F. Liang, C. Shen and F. Wu, "An Iterative BP-CNN Architecture for Channel Decoding", *IEEE Journal on Selected Topics in Signal Processing*, vol. 12, no. 1, pp. 144–159, 2018. DOI: 10.1109/JSTSP.2018.2794062.

[34] Y. Wang, Z. Zhang, S. Zhang, S. Cao and S. Xu, "A Unified Deep Learning Based Polar-LDPC Decoder for 5G Communication Systems", in *2018 10th International Conference on Wireless Communications and Signal Processing (WCSP)*, 2018. DOI: 10.1109/WCSP.2018.8555891.

[35] T. Huynh-The, C.-H. Hua, Q.-V. Pham and D.-S. Kim, "MCNet: An Efficient CNN Architecture for Robust Automatic Modulation Classification", *IEEE Communications Letters*, vol. 24, no. 4, pp. 811–815, 2020. DOI: 10.1109/LCOMM.2020.2968030.

[36] F. Meng, P. Chen, L. Wu and X. Wang, "Automatic Modulation Classification: A Deep Learning Enabled Approach", *IEEE Transactions on Vehicular Technology*, vol. 67, no. 11, pp. 10 760–10 772, 2018. DOI: 10.1109/TVT.2018.2868698.

[37] Y. Wang, M. Liu, J. Yang and G. Gui, "Data-Driven Deep Learning for Automatic Modulation Recognition in Cognitive Radios", *IEEE Transactions on Vehicular Technology*, vol. 68, no. 4, pp. 4074–4077, 2019. DOI: 10.1109/TVT.2019.2900460.

[38] R. M. Lima, O. Shaaban and M. Adrat, "Exploiting Transformers and Attention Mechanisms for Modulation Classification", in *2025 14th International ITG Conference on Systems, Communications and Coding (SCC)*, 2025, pp. 1–5. DOI: 10.1109/IEEECONF62907.2025.10949107.

[39] Y. Umuroglu *et al.*, "FINN: A Framework for Fast, Scalable Binarized Neural Network Inference", *International Symposium on Field-Programmable Gate Arrays (FPGA)*, pp. 65–74, 2016. DOI: 10.1145/3020078.3021744.

[40] S. I. Venieris and C.-S. Bouganis, "fpgaConvNet: A Framework for Mapping Convolutional Neural Networks on FPGAs", in *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2016, pp. 40–47. DOI: 10.1109/FCCM.2016.22.

[41] J. Duarte *et al.*, "Fast inference of deep neural networks in FPGAs for particle physics", *Journal of Instrumentation*, vol. 13, no. 07, pp. 7–27, 2018. DOI: 10.1088/1748-0221/13/07/p07027.

[42] C. Hou, C. Fang, Y. Lin, Y. Li and J. Zhang, "Implementation of a CNN identifing Modulation Signals on an Embedded SoC", in *2020 IEEE 63rd International Midwest Symposium on Circuits and Systems (MWSCAS)*, 2020, pp. 490–493. DOI: 10.1109/MWSCAS48704.2020.9184608.

[43] K. Jung, J. Woo and S. Mukhopadhyay, "On-Chip Acceleration of RF Signal Modulation Classification With Short-Time Fourier Transform and Convolutional Neural Network", *IEEE Access*, vol. 11, pp. 51–63, 2023. DOI: 10.1109/ACCESS.2023.3344175.

[44] C. Horne, N. J. Peters and M. A. Ritchie, "Classification of LoRa Signals With Real-Time Validation Using the Xilinx Radio Frequency System-on-Chip", *IEEE Access*, vol. 11, pp. 11–23, 2023. DOI: 10.1109/ACCESS.2023.3252170.

[45] T. J. O'Shea, J. Corgan and T. C. Clancy, "Radioml 2016.10a dataset", 2016. [Online]. Available: https://www.deepsig.ai/datasets/.

[46] F. Jentzsch, Y. Umuroglu, A. Pappalardo, M. Blott and M. Platzner, "RadioML Meets FINN: Enabling Future RF Applications With FPGA Streaming Architectures", *IEEE Micro*, vol. 42, no. 6, pp. 125–133, 2022. DOI: 10.1109/MM.2022.3202091.

[47] S. Tridgell, D. Boland, P. H. Leong, R. Kastner, A. Khodamoradi and Siddhartha, "Real-time automatic modulation classification using RFSoC", *IEEE 34th International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 82–89, 2020. DOI: 10.1109/IPDPSW50202.2020.00021.

[48] A. Maclellan. "Axdy/rfsoc_quant_amc". (2025), [Online]. Available: https://github.com/axdy/rfsoc_quant_amc.

[49] B. Sklar, "Digital Communications: Fundamentals and Applications", 2nd ed. Prentice Hall, 2001.

[50] A. Goldsmith, "Wireless Communications", 1st ed. Cambridge University Press, 2005. DOI: 10.1017/CBO9780511841224.

[51] "PYNQ", AMD. (2017), [Online]. Available: http://www.pynq.io/. Accessed: May 6, 2025.

[52] L. H. Crockett, D. Northcote, C. Ramsay, F. D. Robinson and R. W. Stewart, "Exploring Zynq MPSoC: With PYNQ and Machine Learning Applications", 1st ed. Strathclyde Academic Media, 2019.

[53] C. R. Harris *et al.*, "Array programming with NumPy", *Nature*, vol. 585, no. 7825, pp. 357–362, 2020. DOI: 10.1038/s41586-020-2649-2.

[54] "Plotly: Low-Code Data App Development", Plotly. (2015), [Online]. Available: https://plotly.com/. Accessed: April 19, 2024.

[55] Project Jupyter. "Jupyter widgets, version 8.1.2 documentation". (2024), [Online]. Available: https://ipywidgets.readthedocs.io/en/stable/. Accessed: February 13, 2024.

[56] T. Kluyver *et al.*, "Jupyter Notebooks – a publishing format for reproducible computational workflows", in *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, F. Loizides and B. Schmidt, Eds., IOS Press, 2016, pp. 87–90. DOI: 10.3233/978-1-61499-649-1-87.

[57] G. Van Rossum and F. L. Drake, "Python 3 Reference Manual". Scotts Valley, CA: CreateSpace, 2009.

[58] "RFSoC RF Data Converter v2.6 Gen 1/2/3 DFE LogiCORE IP Product Guide (PG269)", AMD. (2020), [Online]. Available: https://docs.amd.com/r/en-US/pg269-rf-data-converter. Accessed: May 9, 2025.

[59] ARM. "AMBA AXI-Stream Protocol Specification". (2020), [Online]. Available: https://developer.arm.com/documentation/ihi0051/latest/. Accessed: May 1, 2025.

[60] R. G. Lyons, "Understanding Digital Signal Processing", 2nd ed. Prentice Hall PTR, 2009.

[61] C. S. Weaver, C. A. Cole, R. B. Krumland and M. L. Miller, "The automatic classification of modulation types by pattern recognition.", Defense Technical Information Centre, 1969.

[62] J. L. Xu, W. Su and M. Zhou, "Likelihood-Ratio Approaches to Automatic Modulation Classification", *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 41, no. 4, pp. 455–469, 2011. DOI: 10.1109/TSMCC.2010.2076347.

[63] Z. Zhu and A. K. Nandi, "Automatic Modulation Classification: Principles, Algorithms and Applications". John Wiley  Sons, Ltd, 2014. DOI: 10.1002/9781118906507.

[64] A. Papoulis and S. U. Pillai, "Chapter 5.4: Moments", in *Probability, Random Variables and Stochastic Processes*, 4th, McGraw-Hill Europe, 2002.

[65] P. J. Smith, "A Recursive Formulation of the Old Problem of Obtaining Moments from Cumulants and Vice Versa", *The American Statistician*, vol. 49, no. 2, pp. 217–218, 1995. DOI: 10.1080/00031305.1995.10476146.

[66] O. Dobre, A. Abdi, Y. Bar-Ness and W. Su, "Survey of automatic modulation classification techniques: Classical approaches and new trends", *IET Communications*, vol. 1, no. 2, pp. 137–156, 2007. DOI: 10.1049/iet-com:20050176.

[67] L. Deng, "The MNIST database of handwritten digit images for machine learning research", *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012. DOI: 10.1109/MSP.2012.2211477.

[68] O. Russakovsky *et al.*, "ImageNet large scale visual recognition challenge", *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015. DOI: 10.1007/s11263-015-0816-y.

[69] A. Krizhevsky, "Learning multiple layers of features from tiny images", *University of Toronto*, 2009.

[70] N. E. West and T. O'Shea, "Deep architectures for modulation recognition", *IEEE International Symposium on Dynamic Spectrum Access Networks (DySPAN)*, pp. 1–6, 2017. DOI: 10.1109/DySPAN.2017.7920754.

[71] S. Peng *et al.*, "Modulation classification based on signal constellation diagrams and deep learning", *IEEE Transactions on Neural Networks and Learning Systems*, vol. 30, no. 3, pp. 718–727, 2019. DOI: 10.1109/TNNLS.2018.2850703.

[72] T. J. O'Shea, T. Roy and T. C. Clancy, "Over-the-Air Deep Learning Based Radio Signal Classification", *IEEE Journal of Selected Topics in Signal Processing*, vol. 12, no. 1, pp. 168–179, 2018. DOI: 10.1109/JSTSP.2018.2797022.

[73] J. Krzyston, R. Bhattacharjea and A. Stark, "Complex-valued convolutions for modulation recognition using deep learning", in *IEEE International Conference on Communications Workshops (ICC Workshops)*, 2020, pp. 1–6. DOI: 10.1109/ICCWorkshops49005.2020.9145469.

[74] K. Gurney, "An Introduction to Neural Networks". CRC Press, 2018.

[75] M. Verhelst, "Hardware-efficient machine learning: Designing across the circuit architecture-scheduling stack", Course, IMEC Leuven, Belgium, 2024.

[76] P. Ramachandran, B. Zoph and Q. V. Le, "Searching for activation functions", *ArXiv*, vol. abs/1710.05941, 2018.

[77] D. Misra, "Mish: A Self Regularized Non-Monotonic Activation Function", in *British Machine Vision Virtual Conference*, 2020. DOI: 10.48550/arXiv.1908.08681.

[78] B. Moons, D. Bankman and M. Verhelst, "Embedded Deep Learning: Algorithms, Architectures and Circuits for Always-on Neural Network Processing". Cham: Springer International Publishing, 2019. DOI: 10.1007/978-3-319-99223-5.

[79] A. Paszke *et al.*, "Pytorch: An imperative style, high-performance deep learning library", in *33rd International Conference on Neural Information Processing Systems*, 2019.

[80] Martín Abadi *et al.*, "TensorFlow: Large-scale machine learning on heterogeneous systems", Software available from tensorflow.org, 2015. [Online]. Available: https://www.tensorflow.org/.

[81] Stanford Univeristy, "CS231n: Deep Learning for Computer Vision", 2024. Accessed: April 8, 2025.

[82] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv and Y. Bengio, "Binarized Neural Networks", in *Advances in Neural Information Processing Systems*, vol. 29, Curran Associates, Inc., 2016.

[83] B. Moons, K. Goetschalckx, N. Van Berckelaer and M. Verhelst, "Minimum energy quantized neural networks", in *51st Asilomar Conference on Signals, Systems, and Computers*, 2017, pp. 1921–1925. DOI: 10.1109/ACSSC.2017.8335699.

[84] C. Zhu, S. Han, H. Mao and W. J. Dally, "Trained ternary quantization", in *International Conference of Learning Representations (ICLR) Poster*, 2017. DOI: arXiv:1612.01064.

[85] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv and Y. Bengio, "Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations", *Journal of Machine Learning Research*, pp. 1–30, 2018.

[86] Y. Bengio, N. Léonard and A. C. Courville, "Estimating or propagating gradients through stochastic neurons for conditional computation", *CoRR*, vol. abs/1308.3432, 2013. arXiv: 1308.3432.

[87] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization", 2017. arXiv: 1412.6980.

[88] L. Prechelt, "Early Stopping — But When?", in *Neural Networks: Tricks of the Trade: Second Edition*, G. Montavon, G. B. Orr and K.-R. Müller, Eds., Berlin, Heidelberg: Springer, 2012, pp. 53–67. DOI: 10.1007/978-3-642-35289-8_5.

[89] A. Y. Ng, "Feature selection, L1 vs. L2 regularization, and rotational invariance", in *Twenty-First International Conference on Machine Learning*, Association for Computing Machinery, 2004, p. 78. DOI: 10.1145/1015330.1015435.

[90] C. M. Bishop, "Training with Noise is Equivalent to Tikhonov Regularization", *Neural Computation*, vol. 7, no. 1, pp. 108–116, 1995. DOI: 10.1162/neco.1995.7.1.108.

[91] "Deep learning with intel avx-512 and intel dl boost", Intel. (2025), [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/guide/deep-learning-with-avx512-and-dl-boost.html. Accessed: April 9, 2025.

[92] J. Bottleson, S. Kim, J. Andrews, P. Bindu, D. N. Murthy and J. Jin, "clCaffe: OpenCL Accelerated Caffe for Convolutional Neural Networks", in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2016, pp. 50–57. DOI: 10.1109/IPDPSW.2016.182.

[93] S. Chetlur *et al.*, "cuDNN: Efficient Primitives for Deep Learning", 2014. DOI: 10.48550/arXiv.1410.0759.

[94] M. Horowitz, "Computing's energy problem (and what we can do about it)", in *IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, 2014, pp. 10–14. DOI: 10.1109/ISSCC.2014.6757323.

[95] N. Jouppi *et al.*, "TPU v4: An Optically Reconfigurable Supercomputer for Machine Learning with Hardware Support for Embeddings", in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, Orlando, FL, USA: Association for Computing Machinery, 2023. DOI: 10.1145/3579371.3589350.

[96] S. Colleman, M. Shi and M. Verhelst, "COAC: Cross-Layer Optimization of Accelerator Configurability for Efficient CNN Processing", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 31, no. 7, pp. 945–958, 2023. DOI: 10.1109/TVLSI.2023.3268084.

[97] L. Bai, Y. Lyu and X. Huang, "A Unified Hardware Architecture for Convolutions and Deconvolutions in CNN", in *IEEE International Symposium on Circuits and Systems (ISCAS)*, IEEE, 2020, pp. 1–5. DOI: 10.1109/iscas45731.2020.9180842.

[98] C. Deng, Y. Sui, S. Liao, X. Qian and B. Yuan, "GoSPA: An Energy-efficient High-performance Globally Optimized SParse Convolutional Neural Network Accelerator", in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021, pp. 1110–1123. DOI: 10.1109/ISCA52012.2021.00090.

[99] X. Liu *et al.*, "Collaborative Edge Computing With FPGA-Based CNN Accelerators for Energy-Efficient and Time-Aware Face Tracking System", *IEEE Transactions on Computational Social Systems*, vol. 9, no. 1, pp. 252–266, 2022. DOI: 10.1109/TCSS.2021.3059318.

[100] A. Parashar *et al.*, "SCNN: An accelerator for compressed-sparse convolutional neural networks", in *44th Annual International Symposium on Computer Architecture (ISCA)*, 2017, pp. 27–40. DOI: 10.1145/3079856.3080254.

[101] "Hdl coder", MathWorks. (2024), [Online]. Available: https://www.mathworks.com/products/hdl-coder.html. Accessed: February 28, 2024.

[102] "Fixed-point designer", MathWorks. (2025), [Online]. Available: https://uk.mathworks.com/products/fixed-point-designer.html. Accessed: May 5, 2025.

[103] L. S. Blackford *et al.*, "An updated set of basic linear algebra subprograms (BLAS)", *ACM Transactions on Mathematical Software*, vol. 28, no. 2, pp. 135–151, 2002. DOI: 10.1145/567806.567807.

[104] J. Volder, "The CORDIC computing technique", in *Western Joint Computer Conference*, Association for Computing Machinery, 1959, pp. 257–261. DOI: 10.1145/1457838.1457886.

[105] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines", in *Proceedings of the 27th International Conference on International Conference on Machine Learning*, 2010, pp. 807–814.

[106] "GeForce RTX 20 Series Graphics Cards and Laptops", NVIDIA. (2025), [Online]. Available: https://www.nvidia.com/en-gb/geforce/20-series/. Accessed: May 7, 2025.

[107] "Wireless HDL Toolbox", MathWorks. (2025), [Online]. Available: https://uk.mathworks.com/products/wireless-hdl.html. Accessed: May 7, 2025.

[108] AMD. "Vivado Design Suite Reference Guide: Model-Based DSP Design Using System Generator (UG958)". (2025), [Online]. Available: https://docs.amd.com/r/en-US/ug958-vivado-sysgen-ref/DSP48E2. Accessed: May 8, 2025.

[109] AMD. "AMD Vivado Design Suite". (2025), [Online]. Available: https://www.amd.com/en/products/software/adaptive-socs-and-fpgas/vivado.html. Accessed: April 18, 2025.

[110] Analog Devices, "AD-FMCOMMS3-EBZ Wideband RF Transceiver Evaluation Board", 2025. [Online]. Available: https://www.analog.com/en/products/ad-fmcomms3-ebz.html.

[111] MathWorks, "5G Toolbox", 2025. [Online]. Available: https://uk.mathworks.com/products/5g.html. Accessed: May 9, 2025.

[112] C. Spooner, J. Snoap, J. Latshaw and D. Popescu, "Synthetic digitally modulated signal datasets for automatic modulation classification", 2022. DOI: 10.21227/pcj5-zr38.

[113] K. Tekbıyık, C. Keçeci, A. R. Ekti, A. Görçin and G. K. Kurt, "HisarMod: A new challenging modulated signals dataset", 2019. DOI: 10.21227/8k12-2g70.

[114] X. Zhang, J. Sun and X. Zhang, "Data of novel features and FS_DT-SSVM", 2019. DOI: 10.21227/0m92-z416.

[115] K. Doke, W. Xiong, P. Bogdanov and M. Zheleva, "Modulation recognition iq samples across heterogeneous sensors", 2022. DOI: 10.21227/cp3b-aq35.

[116] R. Utrilla, "MIGOU-MOD: A dataset of modulated radio signals acquired with MIGOU, a low-power IoT experimental platform", 2020. DOI: 10.17632/fkwr8mzndr.1. Accessed: May 14, 2025.

[117] A. Maclellan, L. H. Crockett and R. W. Stewart, "Deeprfsoc dataset for modulation classification", 2025. DOI: {10.15129/95f907fb-4cb2-4365-93ac-c36165053999}.

[118] "Multipath Fading Channel", MathWorks. (2024), [Online]. Available: https://uk.mathworks.com/help/comm/ug/multipath-fading-channel.html. Accessed: May 9, 2025.

[119] AMD, "AXI DMA LogiCORE IP Product Guide (PG021)", 2025. [Online]. Available: https://docs.amd.com/r/en-US/pg021_axi_dma. Accessed: May 9, 2025.

[120] "Nooelec - VeGA Barebones - Ultra Low-Noise Variable Gain Amplifier (VGA) Module for RF & Software Defined Radio (SDR). Highly Linear & Wideband 30MHz-4000MHz Frequency Capability", Nooelec. (2020), [Online]. Available: https://www.nooelec.com/store/vega-barebones.html. Accessed: May 9, 2025.

[121] Y. A. LeCun, L. Bottou, G. B. Orr and K.-R. Müller, "Efficient BackProp", in *Neural Networks: Tricks of the Trade: Second Edition*, G. Montavon, G. B. Orr and K.-R. Müller, Eds., Berlin, Heidelberg: Springer, 2012, pp. 9–48. DOI: 10.1007/978-3-642-35289-8_3.

[122] C. M. Bishop, "Pattern Recognition and Machine Learning" (Information Science and Statistics). Springer, 2007.

[123] B. Jacob *et al.*, "Quantization and training of neural networks for efficient integer-arithmetic-only inference", in *Conference on Computer Vision and Pattern Recognition*, IEEE, 2018, pp. 2704–2713. DOI: 10.1109/cvpr.2018.00286.

[124] Y. Choukroun, E. Kravchik, F. Yang and P. Kisilev, "Low-bit quantization of neural networks for efficient inference", in *International Conference on Computer Vision Workshop (ICCVW)*, 2019, pp. 3009–3018. DOI: 10.1109/ICCVW.2019.00363.

[125] H. Zhao, D. Liu and H. Li, "Efficient Integer-Arithmetic-Only Convolutional Neural Networks", 2020. DOI: 10.48550/arXiv.2006.11735.

[126] D. D. Lin, S. S. Talathi and V. S. Annapureddy, "Fixed point quantization of deep convolutional networks", in *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, 2016, pp. 2849–2858.

[127] S. Kumar, R. Mahapatra and A. Singh, "Automatic Modulation Recognition: An FPGA Implementation", *IEEE Communications Letters*, vol. 26, no. 9, pp. 2062–2066, 2022. DOI: 10.1109/LCOMM.2022.3184771.

[128] E. Yvinec, A. Dapogny and K. Bailly, "To Fold or Not to Fold: a Necessary and Sufficient Condition on Batch-Normalization Layers Folding", in *International Joint Conference on Artificial Intelligence*, 2022, pp. 1576–1582. DOI: 10.24963/ijcai.2022/220.

[129] X. Zhao, R. Xu and X. Guo, "Post-training Quantization or Quantization-aware Training? That is the Question", in *China Semiconductor Technology International Conference (CSTIC)*, 2023, pp. 1–3. DOI: 10.1109/CSTIC58779.2023.10219214.

[130] J. L. DuBois, R. P. Multhauf and C. A. Ziegler, "Invention and Development of the Radiosonde with a Catalog of Upper-Atmospheric Telemetering Probes in the National Museum of American History", in *Smithsonian Contributions to History and Technology*, 2002. DOI: 10.5479/si.00810258.53.1.

[131] "TMS320C64x DSP Library Programmer's Reference", Dallas, Texas, USA: Texas Instruments Incorporated, 2003. [Online]. Available: https://www.ti.com/lit/ug/spru565/spru565.pdf. Accessed: August 31, 2025.

[132] G. Franco, A. Pappalardo and N. J. Fraser, "Xilinx/brevitas", version 0.12.0, 2025. DOI: 10.5281/zenodo.3333552.

[133] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition", in *3rd International Conference on Learning Representations (ICLR)*, 2015, pp. 1–14. DOI: 10.48550/arXiv.1409.1556.

[134] D. Wu, Y. Wang, S.-T. Xia, J. Bailey and X. Ma, "Skip Connections Matter: On the Transferability of Adversarial Examples Generated with ResNets", in *International Conference on Learning Representations (ICLR)*, 2020. DOI: 10.48550/arXiv.2002.05990.

[135] M. Andronic and G. A. Constantinides, "Polylut: Learning piecewise polynomials for ultra-low latency fpga lut-based inference", in *2023 International Conference on Field Programmable Technology (ICFPT)*, IEEE, 2023. DOI: 10.1109/icfpt59805.2023.00012.

[136] Y. Umuroglu, Y. Akhauri, N. J. Fraser and M. Blott, "Logicnets: Co-designed neural networks and circuits for extreme-throughput applications", in *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*, IEEE, 2020, pp. 291–297. DOI: 10.1109/fpl50879.2020.00055.

# Appendix A

# Qm.n Format

Throughout the work presented in this thesis, fixed-point numbers are described using the Qm.n format. This format allows for the short-hand description of signed fixed-point numbers that include a portion of the wordlength allocated to integer $m$ and fractional bits $n$. An overview of how this format is interpreted is presented in this Appendix. Throughout the work presented in this thesis, fixed-point numbers are described using the Qm.n format. This format allows for the short-hand description of signed fixed-point numbers that include a portion of the wordlength allocated to integer $m$ and fractional bits $n$ [131]. An overview of how this format is interpreted is presented in this Appendix.

Figure A.1 illustrates three examples of how an 8-bit signed fixed-point number can be represented in different ways by allocating integer and fractional bits differently.

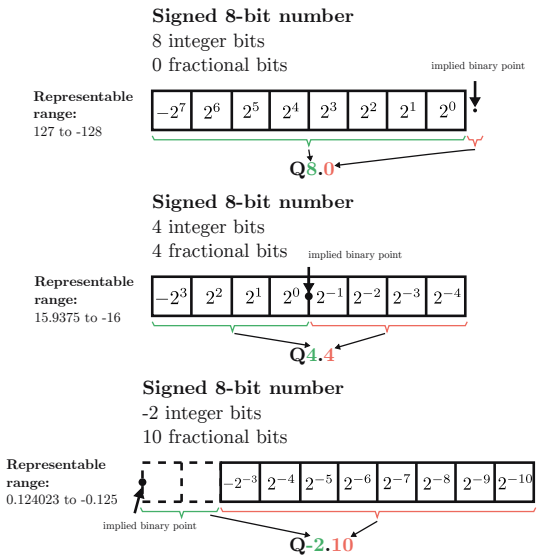



Figure A.1 Examples of Qm.n format used for representing different configurations for a 8-bit fixed-point number.

In the first example, an 8-bit number is interpreted as a whole integer, with all bits allocated to the integer part. This allows for a representable range of -128 to 127.

In the second example, the 8 bits are split into 4 integer bits and 4 fractional bits. This enables the representation of values less than 1, at the cost of reduced range. By placing the binary point in the middle, the representable range becomes -16 to 15.9375.

This idea can be extended further by shifting the binary point entirely to the left. In the third example, all 8 bits are used for the fractional part, meaning the number can represent fine-grained values below 1, but no integer values. To clearly define the format, a negative value is used to indicate how far the binary point has shifted out of the fractional range. The sum of the integer and fractional bit counts still totals 8 bits, preserving the overall wordlength. The resulting representable range becomes -0.125 to 0.124023.

# Appendix B

# Dataset Generation Software

This appendix details the MATLAB software files and PYNQ Python drivers for creating the DeepRFSoC dataset.

## B.1 MATLAB Generation Code

The following code listings contain the MATLAB code used to create the DeepRFSoC dataset prior to transmission through the RFSoC loopback connection.

Listing B.1 defines the number of frames to be generated and transmission parameters, including the channel models, frequency shifts, and timing offsets.

Listing B.1 Pulse-shaping filter and Rician channel configuration in MATLAB.

```
% Parameters for modulation and frames
numFramesPerModType = 1000; % Number of frames per modulation type and SNR value
sps = 8;                    % Samples per symbol
spf = 8192;                 % Samples per frame
symbolsPerFrame = spf / sps; % Symbols per frame
rolloff = 0.5;              % Rolloff factor for pulse shaping
filterLength = 10;          % Filter length (in symbols)
fs = 128e6;                 % Sampling frequency

% Generate filter coefficients (Raised Cosine Filter)
filterCoeffs = rcosdesign(rolloff, filterLength, sps);

% Channel configuration
maxOffset = 5;              % Maximum timing offset
fc = 300e6;                 % Carrier frequency
fs = 128e6;                 % Sampling frequency (repeated for clarity)

% Rician multipath channel model
multipathChannel = comm.RicianChannel(...
    'SampleRate', fs, ...
    'PathDelays', [0 1.8 3.4] / fs, ... % Path delays in seconds
    'AveragePathGains', [0 -2 -10], ... % Path gains in dB
    'KFactor', 4, ...                   % Rician K-factor
    'MaximumDopplerShift', 4);          % Maximum Doppler shift in Hz

% Frequency offset configuration
frequencyShifter = comm.PhaseFrequencyOffset(...
    'SampleRate', fs); % Frequency offset model
```

Listing B.2 describes the process for generating the synthetic data signals for each modulation scheme, including: QPSK, BPSK, QAM16, QAM64, 8PSK, PAM4, GFSK, and CPFSK. Each modulation scheme is affected by a Rician channel model, clock offset, frequency shift, timing drift, and AWGN. The signals are generated into complex waveforms, with I and Q separated and concatenated together.

Listing B.2 Modualtion scheme frame generation method in MATLAB.

```
%% QPSK

QPSK = zeros(numFramesPerModType,2,spf/2);

for i=0:size(QPSK,1)-1

    d = randi([0 3], spf/sps, 1);
    % d = [0; 0; 0; 0; d; 1];

    syms = pskmod(d,4,pi/4);

    % Pulse shape
    tx = filter(filterCoeffs, 1, upsample(syms,sps));

    % Channel
    reset(multipathChannel);
    outMultipathChan = multipathChannel(tx);

    % Clock Offset factor
    clockOffset = (rand()* 2*maxOffset) - maxOffset;
    C = 1 + clockOffset / 1e6;

    % Add frequency offset
    frequencyShifter.FrequencyOffset = -(C-1)*fc;
    outFreqShifter = frequencyShifter(outMultipathChan);

    % Add sampling time drift
    t = (0:length(tx)-1)' / fs;
    newFs = fs * C;
    tp = (0:length(tx)-1)' / newFs;
    outTimeDrift = interp1(t, outFreqShifter, tp);

    % numFramePerModTypeAdd noise
    rx = awgn(outTimeDrift,SNR,0);

    % Get frames
    framesComplex = frameGenerator(rx,spf/2,spf/2,50,sps);

    % Real-frames
    I = permute(real(framesComplex), [3 1 4 2]);
    Q = permute(imag(framesComplex), [3 1 4 2]);
    framesReal = cat(1, I, Q);

    QPSK(i+1,:,:) = framesReal;

end


% QPSK = [real(out), imag(out)];
%% BPSK

BPSK = zeros(numFramesPerModType,2,spf/2);

for i=0:size(BPSK,1)-1

    d = randi([0 1], spf/sps, 1);

```

```
    syms = pskmod(d,2);

    % Pulse shape
    tx = filter(filterCoeffs, 1, upsample(syms,sps));

    % Channel
    reset(multipathChannel);
    outMultipathChan = multipathChannel(tx);

    % Clock Offset factor
    clockOffset = (rand()* 2*maxOffset) - maxOffset;
    C = 1 + clockOffset / 1e6;

    % Add frequency offset
    frequencyShifter.FrequencyOffset = -(C-1)*fc;
    outFreqShifter = frequencyShifter(outMultipathChan);

    % Add sampling time drift
    t = (0:length(tx)-1)' / fs;
    newFs = fs * C;
    tp = (0:length(tx)-1)' / newFs;
    outTimeDrift = interp1(t, outFreqShifter, tp);

    % numFramePerModTypeAdd noise
    rx = awgn(outTimeDrift,SNR,0);

    % Get frames
    framesComplex = frameGenerator(rx,spf/2,spf/2,50,sps);

    % Real-frames
    I = permute(real(framesComplex), [3 1 4 2]);
    Q = permute(imag(framesComplex), [3 1 4 2]);
    framesReal = cat(1, I, Q);

    BPSK(i+1,:,:) = framesReal;

end
%% QAM16

d = randi([0 15], spf/sps, 1);

syms = qammod(d,16,'UnitAveragePower',true);

out = filter(filterCoeffs, 1, upsample(syms,sps));

QAM16 = zeros(numFramesPerModType,2,spf/2);

for i=0:size(QAM16,1)-1

    d = randi([0 15], spf/sps, 1);

    syms = qammod(d,16,'UnitAveragePower',true);

    tx = filter(filterCoeffs, 1, upsample(syms,sps));

    % Channel
    reset(multipathChannel);
    outMultipathChan = multipathChannel(tx);

    % Clock Offset factor
    clockOffset = (rand()* 2*maxOffset) - maxOffset;
    C = 1 + clockOffset / 1e6;

    % Add frequency offset
    frequencyShifter.FrequencyOffset = -(C-1)*fc;
    outFreqShifter = frequencyShifter(outMultipathChan);

    % Add sampling time drift
    t = (0:length(tx)-1)' / fs;
    newFs = fs * C;
```

```
    tp = (0:length(tx)-1)' / newFs;
    outTimeDrift = interp1(t, outFreqShifter, tp);

    % numFramePerModTypeAdd noise
    rx = awgn(outTimeDrift,SNR,0);

    % Get frames
    framesComplex = frameGenerator(rx,spf/2,spf/2,50,sps);

    % Real-frames
    I = permute(real(framesComplex), [3 1 4 2]);
    Q = permute(imag(framesComplex), [3 1 4 2]);
    framesReal = cat(1, I, Q);

    QAM16(i+1,:,:) = framesReal;

end
%% QAM64

QAM64 = zeros(numFramesPerModType,2,spf/2);

for i=0:size(QAM64,1)-1

    d = randi([0 63], spf/sps, 1);

    syms = qammod(d, 64, 'UnitAveragePower', true);

    tx = filter(filterCoeffs, 1, upsample(syms,sps));

    % Channel
    reset(multipathChannel);
    outMultipathChan = multipathChannel(tx);

    % Clock Offset factor
    clockOffset = (rand()* 2*maxOffset) - maxOffset;
    C = 1 + clockOffset / 1e6;

    % Add frequency offset
    frequencyShifter.FrequencyOffset = -(C-1)*fc;
    outFreqShifter = frequencyShifter(outMultipathChan);

    % Add sampling time drift
    t = (0:length(tx)-1)' / fs;
    newFs = fs * C;
    tp = (0:length(tx)-1)' / newFs;
    outTimeDrift = interp1(t, outFreqShifter, tp);

    % numFramePerModTypeAdd noise
    rx = awgn(outTimeDrift,SNR,0);

    % Get frames
    framesComplex = frameGenerator(rx,spf/2,spf/2,50,sps);

    % Real-frames
    I = permute(real(framesComplex), [3 1 4 2]);
    Q = permute(imag(framesComplex), [3 1 4 2]);
    framesReal = cat(1, I, Q);

    QAM64(i+1,:,:) = framesReal;

end
%% 8PSK

PSK8 = zeros(numFramesPerModType,2,spf/2);

for i=0:size(PSK8,1)-1

    d = randi([0 7], spf/sps, 1);

    syms = pskmod(d, 8);
```

```
198
199     tx = filter(filterCoeffs, 1, upsample(syms,sps));
200
201     % Channel
202     reset(multipathChannel);
203     outMultipathChan = multipathChannel(tx);
204
205     % Clock Offset factor
206     clockOffset = (rand()* 2*maxOffset) - maxOffset;
207     C = 1 + clockOffset / 1e6;
208
209     % Add frequency offset
210     frequencyShifter.FrequencyOffset = -(C-1)*fc;
211     outFreqShifter = frequencyShifter(outMultipathChan);
212
213     % Add sampling time drift
214     t = (0:length(tx)-1)' / fs;
215     newFs = fs * C;
216     tp = (0:length(tx)-1)' / newFs;
217     outTimeDrift = interp1(t, outFreqShifter, tp);
218
219     % numFramePerModTypeAdd noise
220     rx = awgn(outTimeDrift,SNR,0);
221
222     % Get frames
223     framesComplex = frameGenerator(rx,spf/2,spf/2,50,sps);
224
225     % Real-frames
226     I = permute(real(framesComplex), [3 1 4 2]);
227     Q = permute(imag(framesComplex), [3 1 4 2]);
228     framesReal = cat(1, I, Q);
229
230     PSK8(i+1,:,:) = framesReal;
231
232 end
233 %% PAM4
234
235 PAM4 = zeros(numFramesPerModType,2,spf/2);
236
237 for i=0:size(PAM4,1)-1
238
239     amp = 1 / sqrt(mean(abs(pammod(0:3, 4)).^2));
240
241     d = randi([0 3], spf/sps, 1);
242
243     syms = pammod(d,4);
244
245     tx = complex(filter(filterCoeffs, 1, upsample(syms,sps)));
246
247     % Channel
248     reset(multipathChannel);
249     outMultipathChan = multipathChannel(tx);
250
251     % Clock Offset factor
252     clockOffset = (rand()* 2*maxOffset) - maxOffset;
253     C = 1 + clockOffset / 1e6;
254
255     % Add frequency offset
256     frequencyShifter.FrequencyOffset = -(C-1)*fc;
257     outFreqShifter = frequencyShifter(outMultipathChan);
258
259     % Add sampling time drift
260     t = (0:length(tx)-1)' / fs;
261     newFs = fs * C;
262     tp = (0:length(tx)-1)' / newFs;
263     outTimeDrift = interp1(t, outFreqShifter, tp);
264
265     % numFramePerModTypeAdd noise
266     rx = awgn(outTimeDrift,SNR,0);
267
```

```
    % Get frames
    framesComplex = frameGenerator(rx,spf/2,spf/2,50,sps);

    % Real-frames
    I = permute(real(framesComplex), [3 1 4 2]);
    Q = permute(imag(framesComplex), [3 1 4 2]);
    framesReal = cat(1, I, Q);

    PAM4(i+1,:,:) = framesReal;

end
%% GFSK

GFSK = zeros(numFramesPerModType,2,spf/2);

for i=0:size(GFSK,1)-1

    d = randi([0 1], spf/sps, 1);
    M = 2;
    mod = comm.CPMModulator(...
        'ModulationOrder', M, ...
        'FrequencyPulse', 'Gaussian', ...
        'BandwidthTimeProduct', 0.5, ...
        'ModulationIndex', 1, ...
        'SamplesPerSymbol', sps);
    meanM = mean(0:M-1);

    tx = mod(2*(d-meanM));

    % Channel
    reset(multipathChannel);
    outMultipathChan = multipathChannel(tx);

    % Clock Offset factor
    clockOffset = (rand()* 2*maxOffset) - maxOffset;
    C = 1 + clockOffset / 1e6;

    % Add frequency offset
    frequencyShifter.FrequencyOffset = -(C-1)*fc;
    outFreqShifter = frequencyShifter(outMultipathChan);

    % Add sampling time drift
    t = (0:length(tx)-1)' / fs;
    newFs = fs * C;
    tp = (0:length(tx)-1)' / newFs;
    outTimeDrift = interp1(t, outFreqShifter, tp);

    % numFramePerModTypeAdd noise
    rx = awgn(outTimeDrift,SNR,0);

    % Get frames
    framesComplex = frameGenerator(rx,spf/2,spf/2,50,sps);

    % Real-frames
    I = permute(real(framesComplex), [3 1 4 2]);
    Q = permute(imag(framesComplex), [3 1 4 2]);
    framesReal = cat(1, I, Q);

    GFSK(i+1,:,:) = framesReal;

end
%% CPFSK

CPFSK = zeros(numFramesPerModType,2,spf/2);

for i=0:size(CPFSK,1)-1

    d = randi([0 1], spf/sps, 1);
    M = 2;
    mod = comm.CPFSKModulator(...
```

```
        'ModulationOrder', M, ...
        'ModulationIndex', 0.5, ...
        'SamplesPerSymbol', sps);
    meanM = mean(0:M-1);

    tx = mod(2*(d-meanM));

    % Channel
    reset(multipathChannel);
    outMultipathChan = multipathChannel(tx);

    % Clock Offset factor
    clockOffset = (rand()* 2*maxOffset) - maxOffset;
    C = 1 + clockOffset / 1e6;

    % Add frequency offset
    frequencyShifter.FrequencyOffset = -(C-1)*fc;
    outFreqShifter = frequencyShifter(outMultipathChan);

    % Add sampling time drift
    t = (0:length(tx)-1)' / fs;
    newFs = fs * C;
    tp = (0:length(tx)-1)' / newFs;
    outTimeDrift = interp1(t, outFreqShifter, tp);

    % numFramePerModTypeAdd noise
    rx = awgn(outTimeDrift,SNR,0);

    % Get frames
    framesComplex = frameGenerator(rx,spf/2,spf/2,50,sps);

    % Real-frames
    I = permute(real(framesComplex), [3 1 4 2]);
    Q = permute(imag(framesComplex), [3 1 4 2]);
    framesReal = cat(1, I, Q);

    CPFSK(i+1,:,:) = framesReal;

end
```

Listing B.3 shows the MATLAB code that repeats the modulation scheme generation process for each SNR value, ranging from: -20 dB to 30 dB.

Listing B.3 Generate modulated frames for each SNR value.

```
% Define the range of SNR values
SNRs = [-20, -16, -12, -8, -4, 0, 4, 8, 12, 16, 20, 24, 28, 30];

% Loop through each SNR value
for i = 1:length(SNRs)
    % Set the current SNR value
    SNR = SNRs(i);

    % Call the function/script to generate modulation schemes for the current SNR
    generateModulationSchemesSNR;
end
```

## B.2 PYNQ Overlay Class and Functions

The following Python code is related to the second portion of the DeepRFSoC generation process where the generated signals from MATLAB are transmitted on the RFSoC through a loopback connection. The following code listings

detail the PYNQ drivers and functions used to control a design on the PL. The design cyclically transmited a signal out of the RF-DAC and received the signal on the RF-ADC, before it was created into the DeepRFSoC dataset.

The process starts from an FPGA bitstream, generated from the Vivado software in Section 5.4.2.

The bitstream was downloaded to the PL through the `pynq.overlay.Overlay()` class from the PYNQ framework. Once downloaded, the IP cores on the device were then configured for the dataset recording task. A custom overlay class, `dataset_building.overlay.Overlay()`, was created that inherits from the PYNQ `pynq.overlay.Overlay()` class. The custom overlay class is imported as shown in Listing B.4.

Listing B.4 Custom overlay initialisation

```
# Import the Overlay class from the dataset_building.overlay module
from dataset_building.overlay import Overlay

# Create an instance of the Overlay class with the specified bitstream file
ol = Overlay('bitstream/dataset_building.bit')
```

In this custom overlay class, the IP cores, located in the PL, are automatically configured such as the RFDC, AXI DMA, and packet generator. In Listing B.5 the RFDC is configured using the PYNQ framework in Python.

Listing B.5 RFDC configuration in PYNQ.

```
# Initialise the RFDC with default configurations
# Get the rf components
self.rf = self.usp_rf_data_converter_0
self.adc_tile = self.rf.adc_tiles[0]
self.adc_block = self.adc_tile.blocks[0]
self.dac_tile = self.rf.dac_tiles[0]
self.dac_block = self.dac_tile.blocks[0]

# Set reference clocks for RF components
xrfclk.set_ref_clks()

# Set sane DAC defaults
# Configure DAC PLL with reference clock and output frequency
self.dac_tile.DynamicPLLConfig(1, 409.6, 1024.0)
# Set Nyquist zone for DAC block
self.dac_block.NyquistZone = 1
# Configure DAC mixer settings
self.dac_block.MixerSettings = {
    'CoarseMixFreq':  xrfdc.COARSE_MIX_BYPASS,
    'EventSource':    xrfdc.EVNT_SRC_IMMEDIATE,
    'FineMixerScale': xrfdc.MIXER_SCALE_0P7,
    'Freq':           700,
    'MixerMode':      xrfdc.MIXER_MODE_C2R,
    'MixerType':      xrfdc.MIXER_TYPE_FINE,
    'PhaseOffset':    0.0
}
# Update DAC mixer event
self.dac_block.UpdateEvent(xrfdc.EVENT_MIXER)
# Enable FIFO for DAC tile
self.dac_tile.SetupFIFO(True)

# Set sane ADC defaults
```

```
# Configure ADC PLL with reference clock and output frequency
self.adc_tile.DynamicPLLConfig(1, 409.6, 1024.0)
# Set Nyquist zone for ADC block
self.adc_block.NyquistZone = 1
# Configure ADC mixer settings
self.adc_block.MixerSettings = {
    'CoarseMixFreq':  xrfdc.COARSE_MIX_BYPASS,
    'EventSource':    xrfdc.EVNT_SRC_TILE,
    'FineMixerScale': xrfdc.MIXER_SCALE_1P0,
    'Freq':           -700,
    'MixerMode':      xrfdc.MIXER_MODE_R2C,
    'MixerType':      xrfdc.MIXER_TYPE_FINE,
    'PhaseOffset':    0.0
}
# Update ADC mixer event
self.adc_block.UpdateEvent(xrfdc.EVENT_MIXER)
# Enable FIFO for ADC tile
self.adc_tile.SetupFIFO(True)
```

The settings configured in Listing B.5 match the configurations described in Tables 5.1 and 5.2. The custom overlay class provides three new functions to the user: **`send()`**, **`receive()`**, and **`stop()`**. The **`send()`** function accepts a **`data_buffer`** as an input and coordinates the communications with the **`tx_dma`**, in the PL, and configures the DMA to transmit the data cyclically. Listing B.6 shows the **`send()`** function.

Listing B.6 The **`send()`** function.

```
def send(self, data_buffer):
    """ Send data from PS memory to the DUC chain
    by setting the dma to operate cyclically.
    [dma -> fir_interp_4 -> fir_interp_8 -> rf_dac]
    The sent data is transmitted continuously.
    """

    # Store the length of the data buffer
    self._data_length = len(data_buffer)
    # Allocate memory for the input buffer with the same length as data_buffer
    input_buffer = allocate(shape=(self._data_length,), dtype=np.int16)
    # Copy data from data_buffer to input_buffer
    input_buffer[:] = data_buffer

    # Stop sending the previous data
    self.tx_dma.sendchannel.stop()
    # Start cyclically sending the new data
    self.tx_dma.sendchannel.transfer(input_buffer, cyclic=True)
```

The **`receive()`** function coordinates the communications between the packet generator and the receiver DMA. The function configures the packet generator to record I/Q packets that are 128 samples long and transfer the packets to the receiver DMA through the AXI4-Stream protocol. The function configures the DMA to receive the packet when then packet has been triggered and to store the received packet in a buffer called **`pkt_buffer`**. Listing B.7 shows the **`receive()`** function in the overlay class.

Listing B.7 The **`receive()`** function.

```
def receive(self):
    # Set the packet generator to generate 128 packets
```

```
    self.pkt_gen.write(0x104, 128)

    # Allocate memory for the packet buffer to store 128 packets
    pkt_buffer = allocate(shape=(128,), dtype=np.uint32)

    # Start the DMA transfer to receive packets into the buffer
    self.dma_pkt.recvchannel.transfer(pkt_buffer)

    # Start the packet generator
    self.pkt_gen.write(0x100, 1)

    # Wait for the DMA transfer to complete
    self.dma_pkt.recvchannel.wait()

    # Stop the packet generator
    self.pkt_gen.write(0x100, 0)

    # Return the received packet buffer
    return pkt_buffer
```

The **stop()** function has been created to stop the **tx_dma** operating in **cyclic** mode. The **stop()** function is shown in Listing B.8.

Listing B.8 The **stop()** function.

```
def stop(self):
    # Stop the dma operating in 'cyclic' mode.
    self.tx_dma.sendchannel.stop()
```

Abstracting the register-level functionality within the custom overlay class streamlines the implementation of the dataset creation system by encapsulating the low-level register operations, making the code cleaner, easier to read, and more maintainable. The functions described are used to create the data set from a Jupyter notebook with snippet of code shown in Listing B.9.

Listing B.9 The data set creation code in PYNQ

```
# Set phase offset array
phase_offset = range(-179, 179, 10)
mods = ['QPSK', 'BPSK', 'QAM16', 'QAM64', 'PSK8', 'PAM4', 'GFSK', 'CPFSK']
snrs = ['-20', '-16', '-12', '-8', '-4', '0', '4', '8', '12', '16', '20', '24', '
    28', '30']

for mod in mods:
    print(f'Starting {mod}...')
    # Load the dataset for the current modulation scheme
    with open(f'./transmit_set/transmit_{mod.lower()}_SNR.pkl', 'rb') as f:
        dataset = pickle.load(f)
    data_dict = {}
    for snr in snrs:
        data_mod = dataset[mod, snr]
        # Initialize an empty array to store complex modulation data
        complex_mod = np.array([])
        i_po = 0  # phase offset index
        for i in range(data_mod.shape[2]):
            data = data_mod[:, :, i]
            # Multiply to fit number in int16 wordlength
            y = np.int16(data * np.int16(pow(2, 14)))
            # Prepare sending variable
            z = np.zeros(2 * 4096, dtype=np.int16)
            # Interleave samples
            z[0::2] = y[0, :]
            z[1::2] = y[1, :]
            # Set phase offset
```

```
            po = phase_offset[i_po % len(phase_offset)]
            ol.phase_offset_tx = po
            i_po += 1
            # Transmit data through DAC
            ol.send(z)

            # Receive multiple times
            for j in range(16):
                # Receive data through ADC
                re_data, im_data = ol.receive_data()
                # Data received as s16_14
                complex_data = np.vstack([re_data, im_data])
                complex_split = complex_data
                # Stack onto accumulative variable to store in dict
                if complex_mod.size == 0:
                    complex_mod = complex_split
                else:
                    complex_mod = np.dstack([complex_mod, complex_split])
            # Stop sending data
            ol.stop()
            print('.', end='')
        # After looping through all frames, save to dict
        data_dict[mod, snr] = complex_mod

    # Write data to file
    with open(f'./received_set/loopback_train_{mod}.pkl', 'wb') as fpkl:
        pickle.dump(data_dict, fpkl, protocol=pickle.HIGHEST_PROTOCOL)
    print(' ')
    print(f'{mod} file written!')
```

The workflow in Listing B.9 iterates over a set of predefined modulation schemes, **`mods`**, including QPSK, BPSK, QAM16, QAM64, 8PSK, PAM4, GFSK, and CPFSK, and multiple SNR levels ranging from -20 dB to 30 dB.

The resulting **`looback_train_{mod}.pkl`** files are combined to create the DeepRFSoC dataset.

# Appendix C

# Demonstrator

An interactive demonstrator was created as a part of this work. The demonstrator provides an interactive **`ipywidgets`** app to test the modulation classification performance of three quantised models concurrently. The demo is available for download at: https://github.com/axdy/rfsoc_quant_amc. Figure C.1 shows a screenshot of the **`ipywidgets`** demo in Jupyter Notebooks.

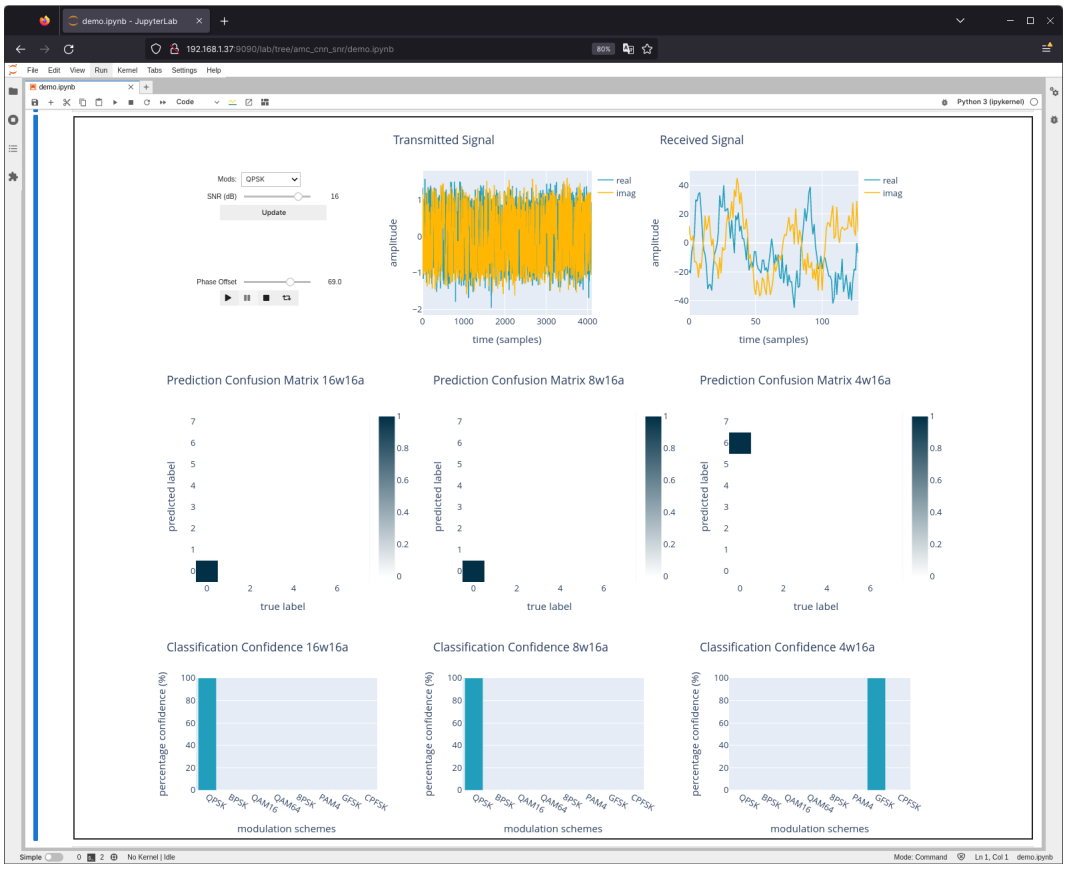



Figure C.1 Modulation classification demonstrator in Jupyter.