University of Strathclyde Glasgow

AMD XILINX

EE997: MSc Project

Machine Learning For Radio Spectrum Analysis

Student: Devjyoti Badal Das (202286524)

Supervisor: Professor Robert W. Stewart

Co-Supervisor: Mr Sarunas Kalade

MSc Machine Learning and Deep Learning,

The Department of Electronic and Electrical Engineering

The University of Strathclyde

18th August 2023

Declaration of Authorship

I, Devjyoti Badal Das, hereby declare that this work has not been submitted for any other degree/course at this University or any other institution and that, except where reference is made to the work of other authors, the material presented is original and entirely the result of my own work at the University of Strathclyde under the supervision of Professor Robert W. Stewart.

Signed: Devjyoti Badal Das

Date: 18/08/2023

Acknowledgement

I would like to extend my sincere appreciation to the University of Strathclyde for enriching my academic journey profoundly and for the strong support provided to me to advance my educational trajectory. The MSc program offered me an exceptionally captivating and insightful experience shaping my scholarly pursuits and personality remarkably.

I am grateful to my esteemed supervisor Professor Robert W. Stewart for bestowing upon me this wonderful opportunity to engage in this one-of-a-kind project and for consistently supporting and guiding me throughout the course of the project which has been an invaluable experience.

I am also extremely thankful to AMD UK and my co-supervisor Mr Sarunas Kalade for presenting a highly valuable industry-engaged learning opportunity and for furnishing the necessary AMD project-based resources. Mr Kalade's consistent presence throughout the research and experimentation has been a testament to his dedication towards sharing perspectives derived from extensive industry research expertise.

I also wish to express my acknowledgement to Dr Javier Moya Paya for approving my registration to the ETH Zurich HACC University program and providing requisite guidance pertaining to the HACC server, Vitis AI Docker and ROCm GPU access. The availability of these resources proved pivotal since they played an indispensable role in turning this project into a successful reality.

Finally, my heartfelt gratitude is extended to all my fellow colleagues, teachers, tutors, and the entire community of RFML researchers. Their collaborative efforts have been instrumental in aiding me to gain insightful knowledge about this wide field of Machine Learning.

Abstract

A rapid growth in wireless technology across different domains of applications has resulted in heavy usage of radio spectrum bands. The rise in demand for these bands creates scarcity in spectrum allocation because the electromagnetic spectrum is a finite resource. To alleviate this scarcity problem, smart spectrum analysis is required through cognitive radio solutions to perform automatic bandwidth allocation. In recent years, state of art Computer Vision algorithms has shown significant capabilities to enable technologies like Cognitive Radio applications to address efficient spectrum management and enhance the capabilities of signal detection in cognitive radio. In particular, research focusing on wideband frequency spectrum analysis using machine learning is scarce and holds a lot of potential to be explored. This project aims on examining the capabilities of machine learning models for radio frequency signal recognition tasks. The prime directive is to obtain a potential spectrogram dataset using state of art tools and applying end-to-end object detection models for spectrogram analysis. Subsequently, to evaluate the entire data and model pipeline, an end-to-end deployment of the model in a low-power hardware architecture is presented to examine the real-world scenario. The obtained result for this project showcases that low-latency object detection models are one of the modern-day architectures that can outperform traditional techniques and is capable enough to work on real-time signals.

Key Words: RFML, Automatic Signal Detection, Object Detection, Spectrum Analysis, YOLO models, Quantization Analysis, Model Deployment, FPGAs, Hardware Accelerators

Table of Contents

List of Figures

List of Tables

List of Appendices

Introduction

In the field of signal processing domain, signal detection and estimation are essential tasks for all telecommunication systems. Where traditional data detection and estimation techniques like matched filtering and Spectrum Analysis through Fourier Transform are facing significant challenges when applied to real-time complex high dimensional noisy signals [1]. These challenges could be further addressed as the inability to recognize a signal in a time-frequency domain, so by adapting an advanced machine learning approach, it is advantageous to analyse the spectrum and intelligently detect signals and their estimation more precisely [2]. Thus, making the Radio Frequency Machine Learning (RFML) domain versatile to adapt to nearly all telecommunication systems.

1.1 Project Overview

With this increasing importance of adapting the machine learning approach in the RF spectrum analysis for automatic spectrum management, the AMD-Xilinx Industrial Supervised Project presented a unique and promising opportunity to dive deeper into resolving the intricate challenges inherent in RFML like high throughput and low latency by leveraging the potential of Spectrum Machine Learning to deploy tailored innovative solutions on AMD-based FPGAs. This project concentrates on automating RF Spectrogram sensing, ultimately leading to an elevated level of utilization of the wideband spectrum. The project will involve a structured process, which is detailed as under:

Initiating the project involves Dataset Creation. This pivotal phase begins with the acquisition of prospective wideband RF Spectrogram generation. The creation of this dataset can be accomplished through two primary methods: firstly, by employing Software Defined Radio (SDR) to capture real-time instances of waterfall plots, thus generating authentic data and secondly, through simulations using tools such as TorchSig Toolkit, GNU Radio, Python Libraries or Matlab Libraries [3]. This scrupulous dataset creation serves as the base for the subsequent stages of the project, enabling robust exploration and analysis. Further, it is imperative to ensure the dataset's comprehensiveness, necessitating the inclusion of a spectrum of modulation schemes to work with.

Thus, the strategy revolves around generating synthetic real-time impaired data through TorchSig toolkit, particularly during the Testing and Deployment phase. This approach substantially enhances the dataset's authenticity and relevance, enabling an effective exploration of various modulation scenarios.

Upon the completion of dataset generation for the designated training set, the subsequent step entails Data Preparation. This consists of the assignment of labels to the data as an object through bounded boxes, effectively rendering the working of object recognition. This pivotal process serves as a catalyst, significantly contributing to the attainment of a rigorously verified object detection model. The central goal is to enable the said model to proficiently identify the modulation type of each spectrogram signal. The process of Data Preparation also consists of several essential phases of data pre-processing phases like feature selection, grouping modulation/ signal classes, data augmentation and imposing synthetic impairments in data (if the data is synthetically obtained). These pre-processing phases are of utmost importance, as they collectively contribute to refining the dataset's quality and strengthening its suitability for subsequent analysis.

Moving forward, the next step in the process is the establishment of a resilient RF Machine Learning Model Architecture which revolves around comprehensively analysing the carefully prepared wideband RF dataset. This involves the diligent implementation of a vigorous object detection algorithm, capitalizing on the available architectures such as R-CNN, R-FCN, YOLO, SSD, YOLOv5, DERT, and U-Net. By integrating these robust architectures, the dataset can be effectively fed and processed, paving the way for the development of a simple and streamlined Automatic Modulation Classification (AMC). The process of Parameter/ Hyperparameter tuning assumes a position of paramount importance involving the precise calibration of our model for this particular task by adjusting the parameters during the training phase, coupled with a rigorous verification of the model architecture through the validation phases. This approach strives for insightful classification results, showcasing the project's commitment to excellence in RF analysis and modulation classification through refinement of the model's performance.

Subsequently, the next step is Performance Evaluation, wherein the efficacy of all the desired model phases is gauged through the adoption of established assessment metrics like Precision, Recall, Mean Average Precision (MAP) or Mean Average Recall (MAR) which are widely used in the domain of object detection RF vision [4].

Lastly, Model Deployment will be carried out and once the metric evaluation is completed, the implementation of optimization methods like that of network compression would aid to augment the efficiency of the network architecture, enabling seamless real-time operation. To realise this approach, experimentation with quantization or pruning methods can be undertaken. Ultimately, the deployment onto an AMD-Xilinx RFSoC PNYQ or on an accelerator card can be seamlessly attained by leveraging the capabilities of the Vitis AI development environment. Hence, this project serves as an ideal source of inspiration with the prospect to make a meaningful contribution to the Machine Learning-based RF Analysis community.

1.2 Applications

The applications of this project do not just limit to real-time signal detection but instead provide a vast amount of various telecommunication domain applications such as automating signal processing using these transferable object detection models, finding new patterns in existing signals and can even work as a stand-alone signal modulation classifier [5].

Specifically, this project is ventured into based on the keen interest in pursuing Radio Frequency Machine Learning (RFML) research for spectral analysis, stemming from the resolute recognition of the existing gaps in RF resources. These gaps present opportunities to support a wide range of wireless communication applications, spanning from improved device traffic management to the acceleration of satellite communication capabilities.

1.3 Motivation

The pervasive utilization of electronic devices on a global scale has compounded the necessity for a comprehensive approach to wideband dynamic spectrum management. This project underscores the urgency of addressing this vital need as the existing literature in this domain is extremely scant, if not entirely absent.

The primary impetus driving this project is grounded in the fact that there is the astonishingly scarce availability of datasets particularly centred around the attainment of real-time signals through an SDR coupled with a GNU Radio (Python) and a vector signal generator. Furthermore, it is an intricate and time-consuming process to undertake the separation of a minimum of 10,000 sample frames as mandated by the exigencies of a convolutional-based neural network to obtain desired results, from different modulation classes/ signal standards and parallelly label the spectrograms. Therefore, a solid foundational automatic dataset generation toolkit was ingeniously devised by Boegner *et al.* (2022) based on leveraging the prowess of PyTorch RFML, known as TorchSig [4]. It is a general-purpose RFML data generation toolkit that serves as an exemplar to examine the widely used 53 different RF signal modulate schemes. The toolkit consists of 2 predefined datasets namely Sig53 and WBsig53 that utilize 6 fundamental family classes of modulations namely: Amplitude-Shift Keying (ASK), Frequency-Shift Keying (FSK), Phase-Shift Keying (PSK), Pulse-Amplitude Modulation (PAM), Quadrature Amplitude Modulation (QAM), and Orthogonal Frequency-Division Multiplexing (OFDM), that is best suitable for this project. The benefit of this toolkit usage lies in its unparallel capacity to personalize complex impairments and transformation environmental variables during the data generation process thus accomplishing the complexities and nuances of dynamic environmental effects in the obtained dataset.

Additionally, Boegner *et al.* (2022) research showcases the generation of a wideband Sig53 (WBSig53) dataset which serves as a pivotal tool for detecting the signal modulation class [4]. Furthermore, there is an abundance of untapped potential in establishing an object detection algorithm tailored especially for the WBSig53 dataset and integrating the application of quantization techniques to the bits of the algorithm along with implementing a pipeline to deploy the model via AMD DPU. In conclusion, this literature gives us the advantage to use transfer learning across generated datasets and compare the results. Thus, this project tries to fill the research void for end-to-end RFML spectrogram-based deployment. Through this dedicated research and development in this field, we possess the ability to create self-aware networks that optimize their behaviour based on real-time spectrum analysis, providing outcomes that have increased efficiency and also exhibit wireless communications that are marked with high resilience.

1.4 Objectives

With the advancements achieved in the domain of wideband RF data analysis, there remains a notable dearth of research concerning the deployment and application of such Radio Frequency Machine Learning (RFML) models. This situation gives rise to a significant void, providing us with an opportunity to set forth clear goals:

- Acquiring an initial robust RFML dataset that is personalized for specifically object detection model,
- Subsequently, thorough object detection model analysis for this custom dataset should be performed to understand the performance metric of the object detection model compared to other techniques, and
- Lastly, by leveraging FPGA (Field Programmable Gate Array) or an accelerator card we can seamlessly deploy the entire data pipeline and model stream.

Literature Review and Methodology

2.1 Literature Review

A series of complexities have been engendered in the realm of spectrum resource allocation due to the exponential proliferation of wireless communication. In order to mitigate the issue of spectrum scarcity, it becomes vital to effectively manage the data traffic within the radio frequency (RF) spectrum and enhance the efficiency of the desired spectrum allocation for optimal outcomes [6]. One of the promising technological advancements aimed at addressing this challenge is the implementation of Cognitive Radio solutions which are innovative solutions having the potential to discern signals from the Primary User (PU) and facilitates the assignment of requisite spectrum access [7]. Contemporary advancements in Cognitive Radio are distinctly focused on the creation of a proficient solution in wide-band spectrum sensing which demonstrates the capacity to detect PU signals and dynamically harness wide-band white frequency sub-bands to their full potential [8] [9].

Over the course of time, the field of RF signal detection and analysis has been delved into extensively by utilizing signal-processing methodologies, focusing on either time or frequency estimation, however, the inherent versatility offered by neural architecture presents a markedly superior capacity for achieving heightened levels of generalizations across a wide array of applications. Therefore, the integration of Machine Learning within wide-band spectrum sensing stands as one of the state-of-the-art techniques employed for the analysis of autonomous RF spectrum. Marking a substantial leap in the pursuit of advanced RF spectrum analysis; this approach holds a distinct advantage over individual spectrum sensing as well as cooperative spectrum sensing techniques. These conventional model-centric methodologies exhibit limitations when operating in a time-varying dynamic wireless environment thereby necessitating the adoption of a machine-learning paradigm to enhance spectrum sensing capabilities [10]. Nonetheless, the utilization of Machine Learning mandates extensive datasets for training and also requires substantial computing power to work with such as Graphical Processing Unit (GPU), posing a huge obstacle to deploy these machine learning models for spectrum sensing in hardware systems. To tackle the limitations associated with datasets within the machine learning approach, pioneering researchers such

as O'Shea *et al.* (2018) presented a standardized dataset focused on narrow-band RF signals to benefit the machine learning community and this resource was extensively used to cultivate and refine deep learning architectures [11]. This class-restricted dataset garnered considerable attention in the domain of RF analysis, but its scope was confined to narrowband signals presented in time series data, lacking the advantageous spectrogram representation. As a consequence, because of poor analysis, its effectiveness in diverse scenario applications was compromised. Thereupon, some contemporary researchers and institutions, exemplified by Vagollari *et al.* (2021) and IEEE SPAWC 2021 Challenge organized by DeepSig, have taken proactive steps to overcome this shortfall by undertaking endeavours to introduce a series of multiple limited wideband spectrogram datasets through simulation. Furthermore, they have also embraced advanced object detection algorithms like YOLO and U-Net to facilitate thorough analysis of signals categorized by modulation classes including PASK, QAM and PAM [12]. While researchers like Nguyen *et al.* (2021) have made noteworthy strides into real-time data analysis of signal protocols like Bluetooth, Wi-Fi, XPD, Lightbridge and Zigbee by involving the utilization of the YOLO approach [13].

The promotion of these datasets has remarkably enhanced the availability of RF data and broadened the scope of research perspective within the RF spectrum. Yet, it is crucial to acknowledge that the substantial complexity and size of these datasets, often reaching Terabytes or large Gigabytes, present difficulties when configuring them for implementation on simpler machine-learning models. Consequently, this complexity can impede the seamless deployment of such models on hardware accelerators, thus making it imperative to address these considerations in order to facilitate a smoother integration of these valuable datasets into practical applications [14]. Moreover, noteworthy advancements like the work by Truong *et al.* (2022) have tried to apply the 'You Only Look Once-version 5 (YOLOv5) framework' for the analysis of Signal to Noise Ratio (SNR) based spectrogram captured dataset and yielded an astonishing accuracy of 99.3% in a dependable environment [15]. Likewise, other researchers like Wicht *et al.* (2021) put forth an innovative proposition involving the utilization of YOLOv4-based real-time signal (Wi-Fi) frame detection which undeniably bolstered the frame capacity of the wireless networks [16].

Training a model for a computer vision problem can be easier and can yield good accuracy results however, deploying it on hardware comes with a big drop in accuracy and RFML Quantization research is very limited to IQ samples and not many have explored the RF spectrogram samples to deploy the model. Soltani *et al.* (2019), have tried to design an RF signal classifier on an FPGA using Software-Defined Radio (SDR) called DeepRadio in real time but with RF front end [17]. However, the experiment was performed in a very regulated environment and the model used was the feed-forward neural network (FNN) that would mostly suffer huge losses if real-world impairments were introduced with the signals. In a similar way, Jentzsch *et al.* (2022) implemented a simple (DNN) architecture VGG10 used for the RadioML dataset using the platform FINN and it was successfully deployed using an FPGA device, but the speed of the models can be a little setback considering the industry standards [18]. Although they have used Quantization Aware training of 4 bits and achieved a good accuracy of 90%, the deployment on a real-time RFSoC is still yet to be explored. This creates a good research space where the RFML models are working on par with traditional models and has the potential to work at a very high speed.

2.2 Methodology

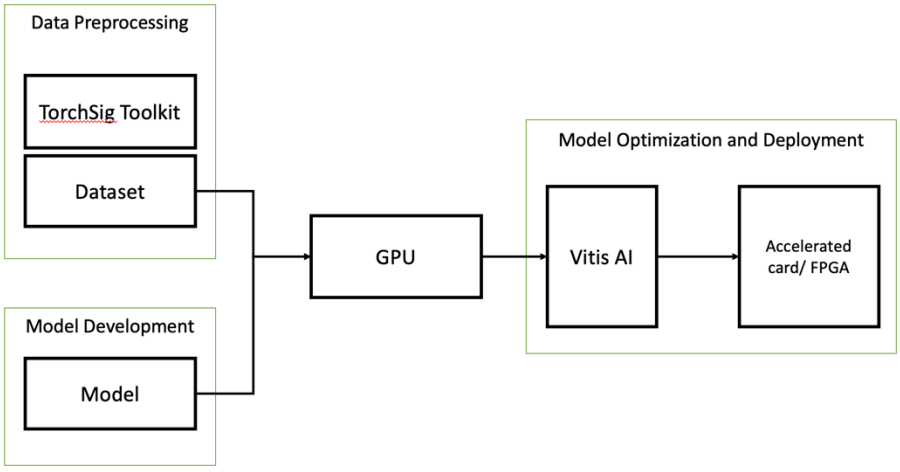

Figure 1: *Block Diagram of Project Workflow.*

This project adopts an Experimental methodology and is systematically structured into distinct strategic phases which are designed to orchestrate the fruitful execution of a fully functional RF Spectrum Analyzer system as outlined below and as shown in Figure 1:

Data Processing: A predetermined open-source dataset toolkit like TorchSig presents a strategic advantage while initiating this project, owing to its provision of valuable resources like effortless data fetching through PyTorch. Employing this approach represents a judicious strategy for gathering training data since the emulation of real-world signal; extensive data diversity and inherent inclusion of impairment diversity of TorchSig will not only expedite the project's inception but also yield considerable savings in time while gathering the dataset. For a simpler approach, an alternative path involves experimenting with verified datasets like WBSig53 for easier variability and implementation of transfer learning approaches.

Upon the dataset generation using TorchSig, a subsequent imperative involves the need for the spectrogram to be complexly transformed plus the labels associated must be target transformed to attain simplification of the computer vision problem. This strategic manoeuvre effectively decreases the dimensions from (3,512, 512) to (2,512, 512). Following this pre-processing phase, the dataset would be ready for examination and analysis.

Post a victorious model architecture development through TorchSig usage and the realization of the sought-after outcomes in the deployment phases, the logical progression entails embarking on the next step of the utilization of an RFSoC to capture a real-time [11].

Model Development (Using Python and PyTorch): It is requisite to undertake systematic experimentation and necessary enhancements with transfer learning models such as DERT, as harnessed in the context of WBSig53 literature shall be applied to evaluate the performances and then similar Object-Oriented Models like Ultralytics module based You Only Look Once (YOLOv5 and YOLOv3) can be tested with an ambition to attain an optimal outcome, characterized by an accuracy level ranging from at least 80% to 90% accuracy. The rationale behind the criticality of attaining such high accuracy stems from the significant amount of accuracy drops that can be noticed during the process of optimizing the model.

While working with the model architecture, there are chances of frame collision taking place due to the occurrence of a dense sequence of short frames. Accordingly, these models heavily depend on mean Average Precision (mAP) with 50-90% confidence (mAP50-90%) ratios and consequently, to improve the model accuracy, it becomes a pre-requisite to hyper-tune the larger step sizes, batch sizes, types of optimizes and learning rates. Thus, there exists the potential for a total reiteration of the complete development process, as the performance of the model in the quantization phase might prove to be dissatisfactory, despite the model achieving commendable accuracy in both the training and evaluation sets.

Model Optimization and Deployment: After tuning and finalizing the trained model, an indispensable step involves integration between the model framework and the hardware infrastructure. To accomplish this, adaptation to the development environment like Vitis AI becomes obligatory. Vitis AI would furnish this project with the capability to leverage tools such as AI Quantizer, Compilers, and Deep Learning Processing Units (DPU). Upon inputting the model into the AI Quantizer integrated within this environment, a transformation would occur, converting the model's single-precision floating-point into an 8-bit integer format. This conversion serves the dual purpose of decreasing memory consumption and raising the data transmission speed of the targeted FPGA (hardware) [19] [26].

AI compiler takes the Quantized neural architecture for generating the DPU-specific model adaptable file. Also, AMD has multiple accumulation operation methods using INT8 that can be executed concurrently on an accelerator card or a development board for inference of the INT8 model [20].

The methodology employed in this project to develop the real-time spectrum monitoring system would pave the way for efficient spectrum utilization that will indeed improve the network performance and enhance the security of wireless communications.

Dataset Description

3.1 TorchSig Toolkit and Architecture

The TorchSig toolkit is a next-generation open-source state of art synthetic signal processor and generator that generates 5 million samples across 53 different modulation types. The toolkit is based on the PyTorch data handling pipeline and also provides multi-data transmission options such as wide-band and narrow-band transmissions covering a frequency range from 10kHz to 10GHz, where all the signals' centre frequencies range from -0.4 to 0.4 of the normalized bandwidths [25]. The signal generation framework of the toolkit can be described as follows:

3.1.1 Carrier Signal and Gaussian Noise

The carrier wave is a high-frequency signal represented as an exponential term with a fixed frequency and amplitude. To generate this carrier wave the following equation is used:

$$\mathrm{C(t)} = \mathrm{A} \,.\, e^{\mathrm{j}(2 * \pi * \mathrm{fc} * \mathrm{t} + \emptyset)}$$

Where A is the amplitude, fc is the carrier frequency in Hertz, t is the time and Ø is the phase offset in radians. This carrier wave is mixed with random Gaussian noise to simulate random amplitude changes which help to generate variations in amplitude. Similarly, by changing other factors such as frequency, pulse, subcarriers, quadrature, and phase, we can achieve different RF modulation variations as shown in Figure 2.

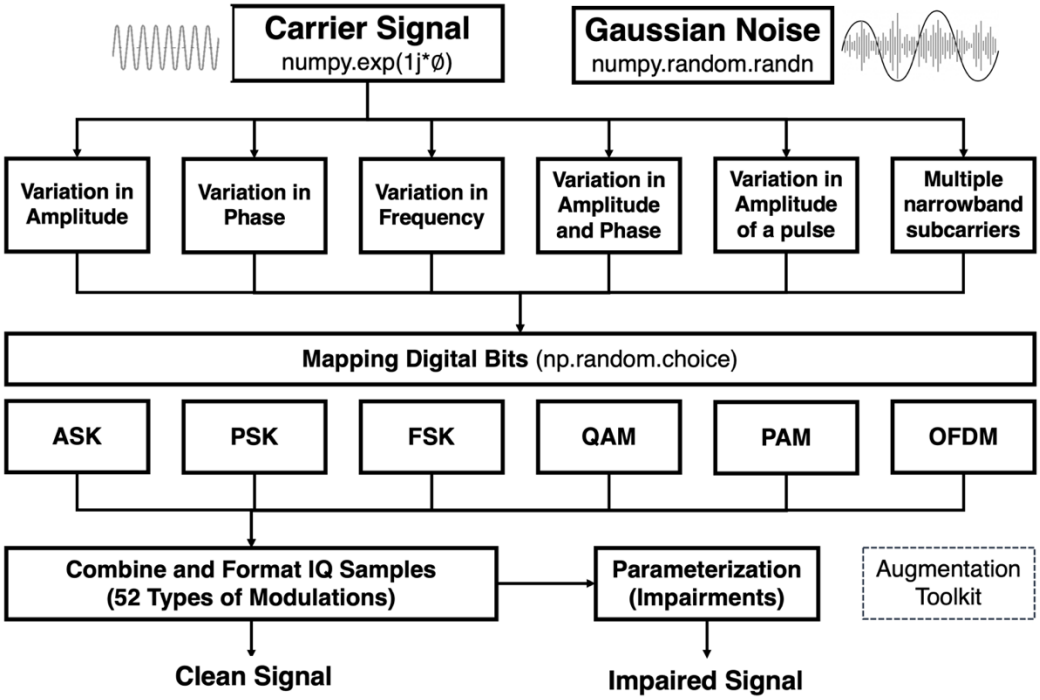

Figure 2: *Block Diagram of TorchSig Toolkit.*

To maximize the research potential of this dataset the toolkit lays out target transforms that map down 53 signal classes to 6 modulation families these include Amplitude Shift Keying (ASK), Frequency Shift Keying (FSK), Phase Shift Keying (PSK), Pulse Amplitude Modulation (PAM), Quadrature Amplitude Modulation (QAM), and Orthogonal Frequency Division Multiplexing (OFDM). These modulations families are represented through 144 complex-valued samples that are called IQ samples. A more detailed description of the 53 signal classes can be found in the Large Scale WBSig53 Torchsig Paper Boegner et., al [4].

3.1.2 Digital Mapper

A digital bits mapper collects all modulated variations and transforms them into synthetic time domain IQ samples by mapping the digital bits. These IQ samples are used by different modulation family functions to generate appropriate sub-modulation classes (53 types of modulations) by changing the frequencies.

3.1.3 Combine and Format IQ Samples

Once the IQ samples are generated for each modulation family, they are combined and formatted to create the final dataset.

3.1.4 Data Formation

The final dataset is a combination of different IQ samples that uses 512-point Fast Fourier Transform (FFT) with zero overlapping to create a 512 x 512 complex-valued spectrogram. These spectrograms are infused with a range of signal-to-noise ratios (SNR) typically ranging from 20 to 40 dB to form Clean Spectrograms. These clean spectrograms are formatted to calculate the respective signal labels through spectrogram sample coordinates. These labels are then encrypted in numpy format and are transformed and combined with the clean spectrograms. The clean spectrograms may have multi-carrier OFDM signals, but these are still treated as single signals and both OFDM and non-OFDM signals may have burst substructures with a probability of 0.2.

3.2 Parameterization Impairments and Augmentations

The TorchSig toolkit comes with additional settings to impose synthetic impairments to the clean IQ samples to resemble the dataset as close to a real-world signal. The impaired diversity functions emulate 5 main synthetic environmental and real-world system RF impairments such as Time Shift (randomizing 25% of time shifts with zero padding), Frequency Shift (randomizing 25% of frequency shifts by applying filters), Random Resampling (25% random resampling to the input data), Spectral Inversion (50% frequency component of the input signals undergoes spectral inversion), and Additive White Gaussian Noise (AWGN is added to the 100% of input data). These impairments are carried out only on IQ samples and each impairment always inspects the respective labels before performing any operations to avoid aliasing.

Impairments like Time Shift and Frequency Shift are introduced to the synthetic signal considering that in the real world, an RF signal undergoes path loss and the Doppler effect which mimics the dynamics of a real signal. Likewise, all real-world RF signals are resampled by various devices (electronic components) that induce Gaussian noise, so AWGN and random resampling is necessary for depicting a real signal.

Apart from the impairments, the toolkit offers added domain-specific data augmentations such as Time Reversal, Channel Swap, Amplitude Reversal, Quantize, Cutout, Patch Shuffle, Local Oscillator Drift, Time-Varying Noise, Clip, Add Slope, Gain Drift, Automatic Gain Control, Spectrogram Random Resize Crop, MixUp Transform, and CutMix Transform. These augmentations are not directly applied to the dataset rather they are selectively chosen while TorchSig model training.

3.3 Data Generation

This research uses a wideband Sig53 (WBSig53) modulation Dataset to cover the massive RF modulation types. This Dataset contains 2 million signals that are subdivided into: Clean Training (250,000 samples), Clean Validation (25,000 samples), Impaired Training (250,000 samples) and Impaired Validation (25,000 samples). The clean and impaired samples of data can be seen in Figure 3.

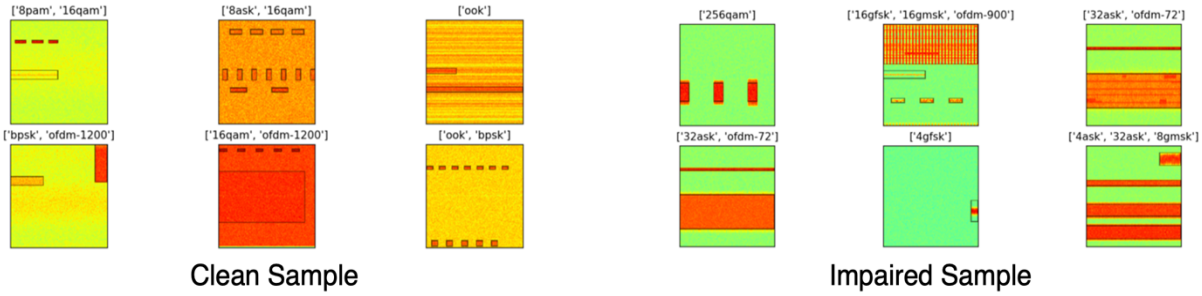

Figure 3: *Clean and Impaired spectrogram examples.*

To make this project versatile it is necessary to use both clean and impaired versions of the WBSig53 Dataset. The synthetic impairment will simulate the RFML models to work for real-world captured signals and the clean signals would help to generalize the ML models. The impaired training and validation data can be loaded using the inbuilt dependencies and classes of the toolkit. The toolkit also enables the creation of custom-impaired datasets with the option to choose modulation classes, FFT size, data transform and target transform. To access this dataset the following procedure is performed in a conda (a Python language package management tool) environment:

3.3.1 Installation of TorchSig

The installation is carried out by cloning the GitHub repository of the TorchSig toolkit and installing the necessary dependencies and libraries in the local system for version control. After installation of the toolkit, PyTorch (a machine learning framework) is installed in the same conda environment to support the data handling pipelines.

3.3.2 Data Accessibility

The WBSig53 dataset can be accessed through a Jupyter notebook by calling crucial TorchSig libraries and instantiating the wideband modulation dataset class by passing a list of inputs such as modulation list, FFT size, number of samples, number of classes, required spectrogram transform and target transform (transforms signal description into mask class labels). During instantiating, the WBSig53 dataset can be tuned in three ways: clean data, impaired data, or custom modulation class data. To apply object detection models for this project, it is necessary to have diversified signals for better generalization, hence both the impaired data and custom modulation class data options are selected and passed to a data loader to form batches of data (by specifying the batch size) for simpler data pre-processing.

Note: Without the TorchSig libraries, the data files cannot be accessed directly since the source dataset files are saved in a database markup language (.dbm format).

3.3.3 Data Visualization

The data loader can be visualized by passing it to the visualizer class and specifying the input that was used while data instantiating. This visualizer class uses ‘matplotlib.pyplot’ library to show the spectrogram images while the labels associated with the spectrogram images are transformed from signal description to mask class annotation using the ‘MaskClassVisualizer’ function. Above every respective spectrogram image, these annotations can be seen as depicted in Figure 3. The ‘MaskClassVisualizer’ function also enables the visualizer class to draw the bounding boxes across every signal in a sample.

3.3.4 Data Analysis

To analyze the spectrogram visualizations further, the WBSig53 dataset is re-instantiated without target transforms. This re-instantiated data is looped through a class counter to count all the class names and the number of classes to plot the distribution of modulation classes per batch. Figure 4 shows a class distribution of a particular batch and looking at it, we can easily deduce that modulation class distribution is random in nature which can cause an imbalance in the entire dataset when all the batches are retrieved. Thus, this dataset needs to be thoroughly balanced before performing training on classification tasks with an object detection model. One of the ways to mitigate this imbalance can be to oversampling the dataset.

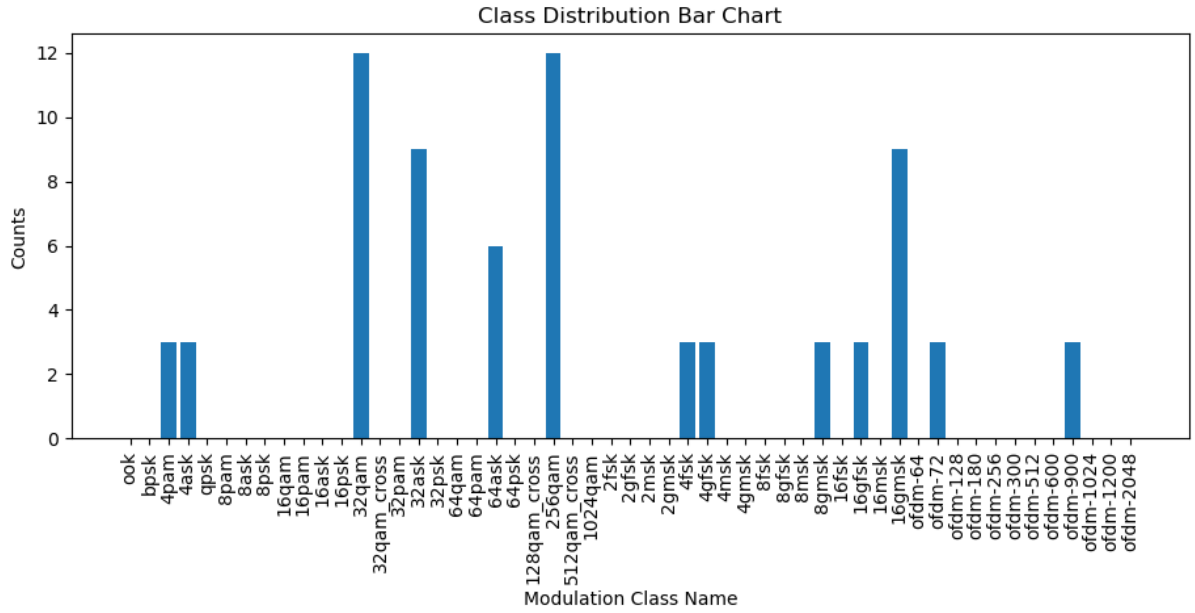

Figure 4: *Bar Chart for Modulation Class Distribution.*

Similarly, Signal to Noise Ratio (SNR) is another parameter that is important for determining the quality and detectability of a spectrogram signal. To analyse this parameter for the given data, the re-instantiated data is looped through the SNRs of each signal to plot the distribution. The distribution as shown in Figure 5, portrays the wide variety of SNR conditions. These irregularity of SNR bins are necessary for training a robust object detection model to achieve realism and diversity that can contribute to the model's ability to work with real-world signals.

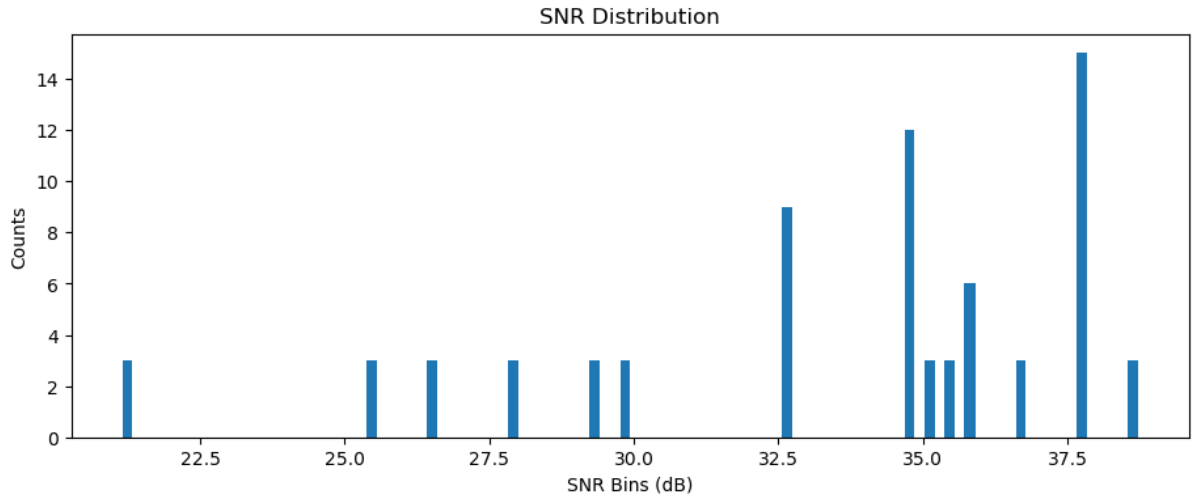

Figure 5: *Bar Chart for SNR Distribution.*

This diversity ensures different-sized object localization across different signals which increases the model performance by enhancing the generalization. Additionally, for the precise detection tasks, the data labels for WBSig53 accurately depict spatial and temporal features of the objects within a spectrogram.

Therefore, this dataset is ideal for object detection-based RF Spectral analysis and can be preferred over the RadioML datasets due to its massive range of signal qualities. However, the only disadvantage with this dataset can be due to unbalanced signal classes that can be mitigated using data pre-processing techniques such as controlled custom-modulation dataset generation or oversampling the data through augmentation functions. A full detailed data generation and analysis code can be accessed from Appendix 2 and Appendix 1 respectively.

Dataset Preprocessing

4.1 Overview of Data Preparation

Spectrogram datasets can be directly obtained using the TorchSig toolkit, however, implementing a custom object detection model while passing the data pipeline is a complex procedure since the dataset is encrypted in Pytorch-based tensors but most model architectures naturally expect a numpy array as an input.

To solve this issue, we have developed a custom algorithm for changing the datatype by utilizing the Visualizer Transform class to iterate over the data pipeline and transform spectrogram data from the '.dbm' format to the '.png' format. This change is beneficial for spectrogram cleaning as some synthetic data might contain insufficient labels. Also, most transfer learning object detection models expect image format to be '.jpeg' or '.png' format.

Similarly, the labels of each spectrogram are separated from the Pytorch tensor format and are stored into a variable to get transformed into a mask-class tuple. These Tuples are further converted into a list where they are represented as {class name}, {x and y coordinates} of the object's centre from the top left corner of the image, and {height and width} of the object itself. These values of x,y, height and width (x,y,h,w) are float data types and together they form a bounding box that represents the coordinate and size of the signal. However, the {class name} is stored as a string data type and needs to be converted to a numerical format for passing it to the model input. Thus, the {class name} of all the signals in a spectrogram image are target encoded by using the 'MaskClassVisualizer' function and are stored as {class Id}. Furthermore, a single spectrogram image may have multiple signals, thus for every signal there should be a label representing that signal. A clear representation of a spectrogram and its labels is shown in Figure 6.

```
42 0.4990234375 0.79296875 0.998046875 0.078125
50 0.4990234375 0.5654296875 0.998046875 0.103515625
21 0.4990234375 0.443359375 0.998046875 0.046875
26 0.4990234375 0.1474609375 0.998046875 0.294921875
```

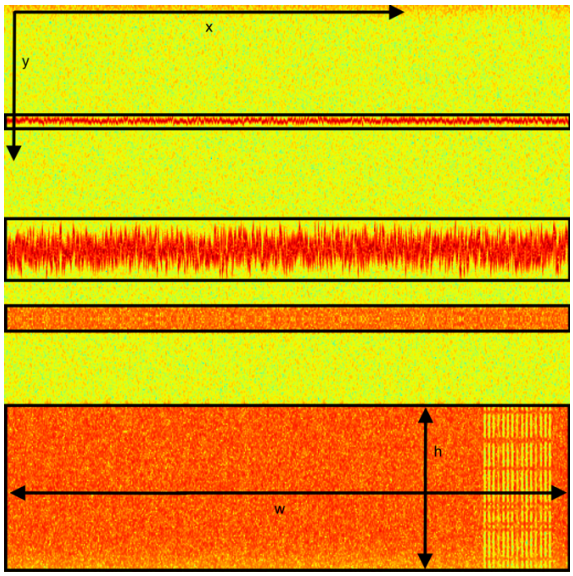

Figure 6: *Spectrogram sample with target object labels.*

Various labels of a single spectrogram are stored in a '.txt' file format with the same name as the relative spectrogram image representing the labels (for example, an image named spectrogram1.png should have its respective label as spectrogram1.txt). Hence, we are creating two different folders one with spectrogram images and another with their respective labels. This prepared data structure is known as the YOLO dataset format which is one of the most commonly used object detection dataset formats to implement YOLO based real-time object detection model. Since our primary object detection model uses custom YOLOv5 architecture, we have tailored the dataset as per YOLO architecture and requirements.

Additionally, this YOLO data structure doesn't come with an annotation that can represent the image and data path because some older YOLO models like YOLOv3 (that we have designed from scratch as a secondary option) require annotations (stored as .csv file) that can infer the paths of an image and label file. Specifically, for designing and using the YOLOv3, we have also included a CSV generator which takes image and label names from respective folders and forms the annotation. This step also helps to make our custom pre-processed dataset reproducible for other kinds of object detection models as well.

Similarly, to use the Ultralytics-based YOLO models, we have provided custom 'data.yaml' file which contains the path and classes to the spectrogram images and labels folder of train, test and validation. The code for the CSV generator and yaml file can be found in Appendix 2.

4.2 Steps to Generate Custom YOLO Dataset

4.2.1 Instantiate TorchSig Dataset

The TorchSig libraries and modulation list containing all 53 signals are passed as an input argument to the 'wideband modulation dataset' class. This class is also given additional settings such as an FFT size of 512 with the number of samples set to 1680. We have also passed the spectrogram data transform argument to achieve spectrogram samples and similarly, the Description to mask class transform argument is used to convert the labels into mask class. The 'wideband modulation dataset' class creates the data in a tensor format and stores it in a variable. Also, the benefit of using the 'wideband modulation dataset' class is that it includes default real-world impairments at random.

4.2.2 Data Splitting

The splitting is performed on the tensor variable (which stores the dataset) and a random split ratio of 80% for training, 20% validation and 20% for testing is applied to divide the dataset. Therefore, we passed a batch size of 1200 samples for training, 240 samples for validation and 240 samples for testing while keeping the image shuffling turned on.

1200 high-quality samples for a particular signal detection task are enough to provide ample exposure for training a robust object detection model. 240 samples in the validation set would give us an unbiased estimation of how the model performs for an unseen sample of data thereby giving us an overview of model behaviour.

4.2.3 Data Up-sampling

The visualizer class and the 'MaskClassVisualizer' function convert the raw tensor variables of train, validation, and test into a Numpy array and separate the images and their respective labels into different variables. Further, the image variables are up-sampled by setting up a higher dpi of 300, which leads to an image size of 1386 pixels.

The up-sampling from 512 x 512 pixels to 1386 x1386 pixels is performed to increase the localization precision since the pixel granularity increases and reduces the risks of signal occlusions in the custom dataset.

The data up-sampling would result in an increase in all the image sizes from 200 kilobytes (KB) to 1.5 megabytes (MB). This increase in image size may cause dataset transfer a little bit slower if more than 2,000 samples are used so we have kept the total number of samples at 1680.

4.2.4 Detection-based Data Changes

The labels associated with each up-sampled spectrogram are saved as a {class Id, x,y,h,w} format. However, our research focuses on building a signal-detection model that needs to detect the signal and not the signal modulation type. So, we pass an extra argument (number of classes = 0) to the visualizer class to combine all the target classes into a single class that can be called a signal. This transformation of labels will help to eliminate any modulation bias. The spectrogram image and the labels after data changes can be visualized in Figure 7.

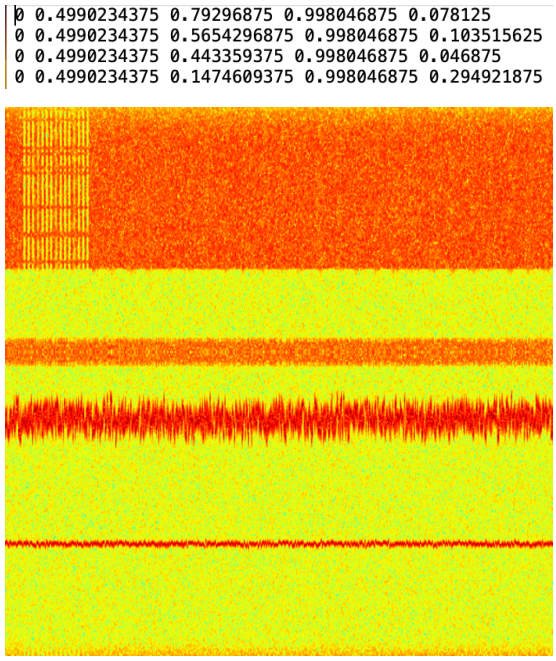

Figure 7: *Detection-based data change sample.*

4.2.5 Data Structure

After up-sampling the image and extracting the respective transformed labels by converting it into numerical format (.txt); the train, validation and test files are formed and saved separately with the images and labels folder that contains the split data image and labels accordingly.

4.2.6 Data Cleaning

For some instances of data samples in all three splits (train, validation, and test), we observed that there are instances where some signal labels are missing/ incorrect. Hence these signals were manually eliminated to reduce training loss for custom object detection models (YOLOv3 and YOLOv5). Also, by eliminating these instances the models avoid learning the incorrect association between features and labels.

Mostly these eliminated signals belong to burst mode modulation classes and can create a minute imbalance in the dataset classes. In conclusion, we have increased the quality and precision of the signal spectrograms and reduced the biases by this thoroughly processed dataset. As a result, this dataset is ready to be used for all the object detection models used in this project. Appendix 2 demonstrates the Jupyter Notebook with the code to generate a custom YOLO Dataset using the TorchSig toolkit.

In a brief, the modifications to the TorchSig toolkit make it a state-of-the-art highly valuable YOLO resource that defines new insights for RFML researchers. The comprehensive data preparation is tailored to effectively cater to the multifaceted demands posed by advanced signal processing and detection scenarios.

Model Selection

In a signal detection task, the most pivotal aspect of a machine learning algorithm is to identify the precise position of the signal (object) within a spectrogram image. State of art machine learning architectures like Object detection algorithms are designed to recognise spatial arrangements of a signal using a technique called as spatial localisation. Spatial localization involves outlining a bounding box across every signal in a spectrogram to know its exact location. These bounding boxes are visual indicators and are calculated by an object detection model using the (x1, x2, y1, y2) coordinates of a signal, also these coordinates are mostly represented in the dataset by the Axis-Aligned Bounding Box (AABBs) i.e., (x,y,h,w) format. This technique helps to localize multiple signal types of varying sizes in a spectrogram image simultaneously. Moreover, in a complex overlapping environment of signals in a spectrogram, the object detection models can accurately distinguish between signals by pointing to the boundaries of the bounding box coordinates. Overlapping of RF signals usually occurs due to shared modulation schemes, and multipath propagation.

The selected model's objective will be to correctly pinpoint the exact location of the real-world real-time RF signals and thus non-object detection models like image classification or semantic segmentation can be inappropriate because we are not focusing on classifying a single signal within a spectrogram. Hence, using an object detection model for the custom TorchSig dataset will lead to more accurate results.

5.1 Understanding Model Architectures

There are two main types of object detection architectures namely: single-stage object detectors and two-stage object detectors. Single-stage object detectors directly focus on classifying object (signal) anchor boxes without extracting the region of interest (ROI) for example YOLO Model family, and the CenterNet Model Family (SDD). While Two-stage object detectors first try to extract the ROI and then classify the ROI to get the detection results, examples of two-stage architectures are Image Transformers, R-CNN Faster-RCNN and Mask R-CNN.

The single-stage object detection architectures are simpler to implement with fewer hyperparameters and generally have a faster inference time making them ideal for real-time applications that can result in slightly lower accuracy. The single stage also helps to develop end-to-end deployment pipelines which is necessary for our project. Contrary to single-stage detection, multi-stage detection involves more accurate localization and supports better handling of overlapping instances. Therefore, there might be an increase in accuracy, however, the inference time and model complexity make the model difficult to train even via transfer learning. Hence, for this research, we have conducted extensive model experimental analysis for both single-stage and multi-stage object detection architectures.

Both single-stage and multi-stage object detection algorithms are assessed using a few common metrics such as Precision, Recall, Intersection over Union (IoU), Average Precision (AP), and Mean Average Precision (mAP). Out of these, Precision and Recall are the most straightforward metric that measures the accuracy of true predictions and measures true predictions for all correctly predicted labels respectively. IoU calculates the difference between the ground truth labels and the predicted bounding boxes; for this research, we have used IoU for the non-max suppression function in our models.

While the metrics like AP compute the area of the precision-recall curve by changing the confidence threshold of a single object for a model, mAP is an extension where it takes the mean of APs for all the objects. The AP and mAP scores are versatile in nature because they provide a more nuanced evaluation at different confidence levels and describe the true performance of an object detection model. Hence, we would be relying mostly on mAP for better judgment.

However, these metrics alone do not decide the success of a computer vision model. The success of real-time RF signal detection tasks heavily depends on the attributes of an object detection model such as high frames per second (fps) to identify rapidly changing signals, accurately distinguishing between noisy/ complex signals, adaptability to point out diversified signals and working within a conserving computational resource. To fulfil this criterion, we have selected two single-stage architectures: YOLOv5 and YOLOv3; and one multi-stage architecture Detection Transformer (DETR).

The reason for selecting these specific object detection models is, firstly, the YOLOv5 and YOLOv3 architecture both have simpler parameters and training loops compared to other advanced YOLO family members. Also, the faster inference time and excellent object prediction mechanism make the YOLOv5 and YOLOv3 model ideal for intelligent signal analytics. Secondly, DETR has one of the advanced innovations in multi-stage detectors that provides a wide range of backbone scaling and size which ultimately results in a good balance between accuracy and computational resources. The DETR employs a bi-directional feature pyramid network mechanism that adapts to different image aspect ratios, making the model robust.

5.2 Performance Analysis of the Models

To understand each model architecture used in this research in detail, we experiment with some convolutional-based neural approaches like YOLOv3m, YOLOv3-from-scratch, YOLOv5x, YOLOv5m, and YOLOv5s and a transformer-based approach like DETR architecture with EfficientNet backbones and XCiT transformer.

5.2.1 YOLOv3m (Ultralytics Model)

The YOLOv3 model uses a Darknet 53 architecture as the backbone and has been built on the framework of YOLOv2. Similar to YOLOv2, the YOLOv3 model is trained on various datasets like MS COCO, COCO and PASCAL VOC. The Darknet 53 (a convolutional neural network that has 53 deep layers) acts as a feature extractor which captures heretical features across the depth of the network resulting in feature maps. The Detection Head takes these feature maps, which utilize individual gird cells to predict the bounding boxes and confidence score around the signals. The detection head uses a predefined scale and aspect ratio of the anchor box to automatically adjust the dimension and plot bounding boxes.

Further, if there are multiple classes in a dataset, the YOLOv3's head uses softmax activation to show the probability over each class. However, in our custom dataset, we have only one class (called signal), thus there is no need to apply softmax activation in our YOLOv3 model detection head.

After the detection head plots the bounding boxes, the Non-Max Suppression (NMS) function removes duplicate bounding box predictions by measuring the confidence threshold [22].

YOLOv3 was the first detection model to utilise anchor boxes with advanced feature extraction and its performance surpasses that of SSD. We have adapted Ultralytics YOLOv3m which was originally developed in Pytorch and trained using the MS COCO dataset and also supports transfer learning. The YOLOv3m is a medium-sized model with 23 million parameters and uses a swish mathematical function (SiLU) for activation [22]. This model has been selected out of all the other YOLOv3 family models to ensure a balance between efficiency and inference time. Other YOLOv3 family models like YOLOv3l and YOLOv3s have extreme properties which might be bad for the model deployment phase.

The YOLOv3m model is implemented by cloning the GitHub repository from the Ultralytics webpage and installing the necessary dependencies. The YOLOv3 repository includes the training script for training the model and the validation script for validating the model. These training and validation script uses utility-file functions to perform different tasks like auto anchor box creation, evaluation metric calculation, loss computation and model early stopping. Apart from the Utility file the test and validation scripts also include the model files which store the entire float32 YOLOv3 model with a configuration option to select YOLOv3m.

After installing the dependencies, setting up the conda environment and configuring the YOLOv3m option, the training script of the pre-trained YOLOv3m model can be executed using a Jupyter notebook. Before running the script, adding additional parameters such as image size, batch size, number of epochs, and number of parallel workers as input arguments to the model is necessary (here the number of workers signifies parallel threads and helps the data logger to load the data faster during the training process). Above all, the path of the data annotation (‘data.yaml’ file) should be pointed to the training script before model training so as to effectively supply the training and validation samples to the model.

During training experimentation with YOLOv3m, we used the Adam optimizer which uses extended Stochastic Gradient Descent (SGD) that helps to dynamically adjust learning weights by referring to the previous individual weights. To tailor the model, pre-trained COCO weights

that come with the YOLOv3 model as a default are passed while training. The purpose of using the pre-trained weights is to accelerate the model training to attain fast convergence for less computational cost (FLOPS). Also, after tuning the hyper-parameters during the model training the learning rate was set at 0.01, with a weight decay of 0.0005 and a batch size of 32 for best accuracy. After training the model for 300 epochs and parallelizing data loading with 8 workers, we get a precision score of 0.722 within an inference time (per batch of data) of 6.1 milliseconds and a strict mean average precision (mAP50-90) of 0.355. A mAP50-90 is a metric score that calculates mean AP keeping the confidence threshold value between 0.6 to 0.9, thus it yields a comprehensive understanding of model performances and is the best criterion to finalize a model. The codebase for YOLOv3m can be found in Appendix 3.

A mAP50-90 score can be deduced as the model architecture is underfitting the data and the model will perform poorly in the test phase. We also tried to train the YOLOv3m without the pre-trained weights and got a mAP50-90 of 0.30. Hence, the pre-trained weights give a better initialization and improve model performance. A mAP50-90 score of 0.355 is not a desired score as it can be verified using Figure 8; because this project aims to quantize a model which will drop the accuracy further, so a model with sufficient accuracy is required to detect real-time RF signals. Thereby subsequently, we tried to implement a personalized YOLOv3 from scratch to establish a model only for our dataset.

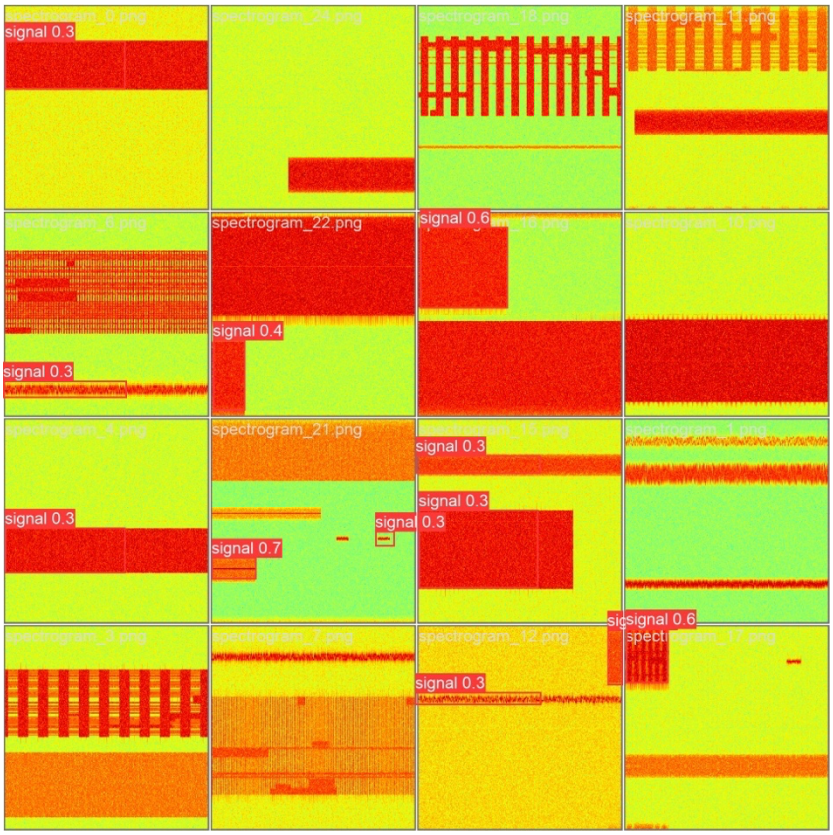

Figure 8: *YOLOv3 predicted labels.*

5.2.2 YOLOv3 (from scratch)

The YOLOv3 architecture when designed from scratch uses the same Backbone layers as in the Ultralytics YOLOv3 model. However, the augmentation library is changed to 'Albumentations' for introducing data augmentations and the CNN block now uses a Leaky ReLU as an activation function. The reason to use Leaky ReLU instead of using SiLU is to reduce the complexity of the activation function and avoid the problem of vanishing gradient. The leaky ReLU function provides output for both positive inputs and negative inputs, the only difference between the outputs is the output for negative values always tends to be zero. Leaky ReLU is further preferred due to model deployment purposes which are discussed ahead in the Model Development section of this report. Also, a simpler version of IOU, a non-max suppression function (to reduce anchor boxes), mean average precision and loss functions are also customised for this model.

Firstly, the dataset argument uses a dataset path, image and dataset annotation file (in .csv format) to direct the data loaders objects for constructing the train data loader and test data loader. Secondly, data augmentations like random crop, longest max size, horizontal flip, blur, and normalise are introduced to the incoming custom train dataset. Thirdly, a loss function is created using 3 different types of losses: MSE loss (Mean Squared Error Loss), BCE with logit loss (Binary Cross Entropy with logit loss), and cross-entropy loss to adapt and validate object and class losses respectively. Lastly, a checkpoint function is introduced to save all the confidence intervals while training to manage future transfer learning perspectives. These saved checkpoints include model state, model parameters, model weights and optimizer used.

After designing these modifications and parameters for the YOLOv3 model the training loop is setup by giving input arguments like the batch size of 16, the learning rate is kept at 0.0001 (since training from scratch requires gradual optimization), and the confidence threshold is kept at 0.05 for 100 epochs. After 100 epochs the mAP50-90 remains at 0.1 and the model undergoes early stopping. Therefore, the mAP is relatively low and suggests that the YOLOv3 architecture hasn't converged (might be due to vanishing gradient) and must require more complexity to learn the feature maps. Appendix 4 highlights the code for the designing and training of the YOLOv3-from-scratch.

We decided to omit further model experimentation on YOLOv3 architecture since it is computationally very expensive to train the network on an Online Cluster Server and adopt a slightly more complex and computationally lightweight model. The successor to the YOLOv3 architecture is the state of art YOLOv5 family architecture. Finally, building a model from scratch can be an innovative option to avoid transfer learning dependencies but the performance tuning can be very challenging and time-consuming for an object detection task.

5.2.3 YOLOv5 Family (Ultralytics Model)

The YOLOv5 is a single-stage object detection architecture that was designed in 2020 by Jocher *et al.,* (2022). This high-level architecture consists of three main sections: Backbone, Neck and Output. The function of the backbone in YOLOv5 is to increase channel resolution to extract features of input spectrogram images [21].

Once the features are extracted the neck section is responsible for building the feature pyramid for object generalization. The output head then calculates bounding boxes by applying anchor boxes to the feature pyramids to generate objectness scores for a particular class.

Like the YOLOv3 architecture, the backbone of YOLOv5 uses a DarkNet53 with the Cross-Stage Partial (CSP) network. The CSP network truncates the gradient flow by using residual and dense blocks by dividing the base layer feature map and applying cross-stage hierarchy. This technique helps to reduce the number of parameters and increases the model inference speed by three times. To further elaborate on the architecture, the neck of the YOLOv5 is built using Path Aggregation Network (PANet) that uses a Spatial Pyramid Pooling (SPP) at the input to aggregate variable-sized inputs and forwards fixed-length outputs. Thereby, SPP segregates the best-matching features from the individual feature maps. The next stage in YOLOv5 architecture is the detection head which predicts the location of bounding boxes (x,y,h,w), accuracy metrics, and calculates object loss by using BCE.

A more comprehensive diagram of YOLOv5 architecture is shown in Figure 9, where the Focus layer highlights the input spectrogram images and passes it to the convolutional block to perform Similarity Based Learning (SBL), batch normalisation and striding. After extracting important feature maps, CSP layers recursively extract the feature pyramid and finally pass it to the SPP block. The bottleneck in the SPP layer is up-sampled and concatenated in the PANet section for better learning capability while model training. These bottlenecks in the PANet section produce 2D Convolution of the spatial image maps which helps the output section to apply anchor boxes and IoU for predicting desired objects. The authors of this YOLOv5 model use the SiLU activation function as a default function in the hidden layers and this model is originally trained for the MS COCO dataset.

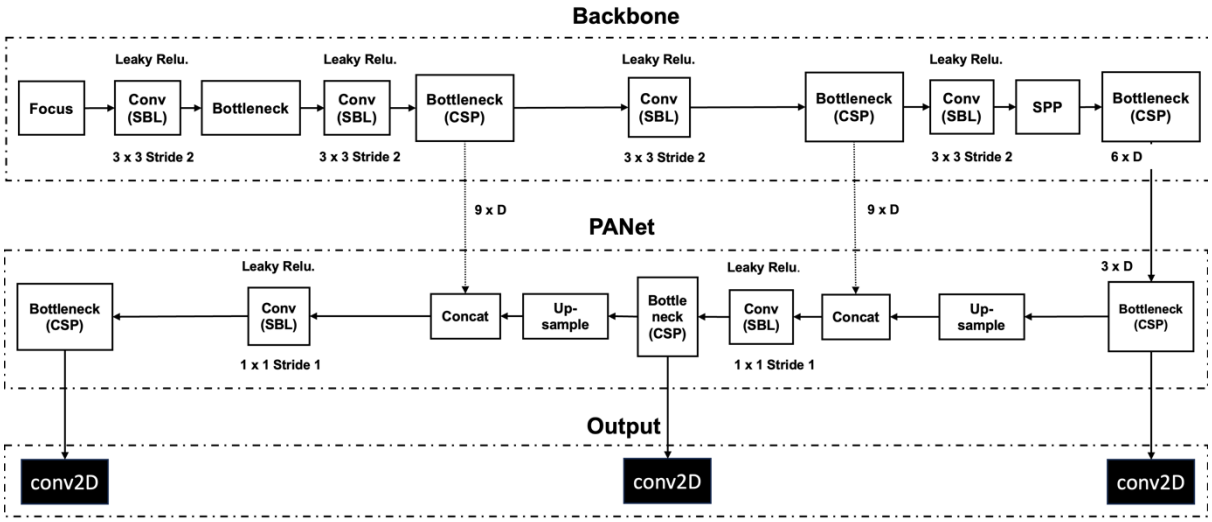

Figure 9: *YOLOv5 architecture.*

The YOLOv5 family has five architectural size variations: nano (n), small (s), medium (m), large (l), and extra-large (xl). All these different size variations share the same characteristics framework except for the difference in the number of layers and parameters. Although, there is a significant performance difference between all the models with YOLOv5n performing the least and sharing the lowest number of parameters and inference time, and YOLOv5x performing the best but has the highest parameters.

To access these YOLOv5 architectures via Ultralytics, firstly, there is a need to clone the GitHub repository while setting up a conda environment and install the required dependencies for running the architecture. Secondly, place the annotation file (data.yaml) inside the data folder of the YOLOv5 repo and import the custom TorchSig-based YOLO dataset in the same directory. This annotation file format is the same as the YOLOv3m annotation file that contains the train and test data sample paths and single signal class. And finally, to train a YOLOv5 model for our custom dataset, a Jupyter notebook can be set up in the same conda environment and the training script can be passed into the Jupyter notebook. Before running the script, input parameters need to be given such as image size, batch size, number of epochs, number of parallel workers and model configuration file that is specific to types of YOLOv5.

All the YOLOv5 models are paired with the Adam optimizers as a default optimizer to learn the neural network weights and the activation functions of the models are changed to Leaky ReLU (by 0.1) to construct a simpler activation function that can be quantized during the model deployment phase if the YOLOv5 model gets selected. YOLOv5 activation functions are stored in common and experimentation Python files inside the 'model' folder in the YOLOv5 repository. These files are updated by adding the Leaky ReLU activation function and deleting the SiLU activation functions as portrayed in the code of Appendix 7.

The principal advantage of using YOLOv5 from Ultralytics is that the YOLOv5 model comes with advanced input image augmentations like Mosaic, Random selection, Image MixUp and HSV (change in hue and saturation of the image). These augmentations introduce generalization to the model input and the model learns better even with a low learning rate.

To add variation in the selection of the YOLOv5 architecture, we have experimented with three different-sized YOLOv5 models: YOLOv5s, YOLOv5m, and YOLOv5x to get a brief understanding of how the custom dataset performs on the YOLOv5 framework. In this experimentation, we have used the pre-trained COCO weights to train the models, since in the YOLOv3 experiment we saw that training a model from scratch without any weights can be disadvantageous for model learning. Furthermore, we have set the learning rate to 0.01, with a weight decay of 0.0005 and a batch size of 32 as the parameters for running the training script. These parameters were specifically selected because we achieve the best results with these values. Appendix 5 follows through the links for the training code of the YOLOv5x, YOLOv5m and YOLOv5s respectively.

After training the model for 300 epochs and parallelizing data loading with 8 workers, we get a precision score of 0.533 within an inference time of 7.22 milliseconds and a mAP50-90 of 0.373 for a YOLOv5s framework. Similarly, we get a precision score of 0.902 within an inference time of 6.67 milliseconds and a mAP50-90 of 0.694 for a YOLOv5m framework. And lastly, for a YOLOv5x framework, we get a precision score of 0.918 within an inference time of 6.67 milliseconds and a mAP50-90 of 0.373.

These mAP50-90 results suggest that the YOLOv5s model architecture is not capable to establish correct feature maps to predict the right labels associated with a signal for a spectrogram sample, thus causing underfitting. Whereas the YOLOv5x model gives the highest performance and learns really well due to the larger model depth that produces a more accurate feature pyramid that helps the model to scan the spectrogram samples thoroughly. However, the YOLOv5x requires high computational power (203.6 GFLOPS) and a very high inference time (per batch of data) of 30 milliseconds, therefore, is not suitable for deployment in FPGAs (Field Programmable Gate Arrays).

The most appropriate model that we can visualize by looking into the performance is the YOLOv5m, this model comes with a medium depth and constructs the feature pyramid accurately similar to the YOLOv5x. As seen in Figure 10, the box loss and object loss descend towards 0 while the precision and recall ascend towards 1 thus signifying that the model is constantly improving by learning the feature maps more accurately.

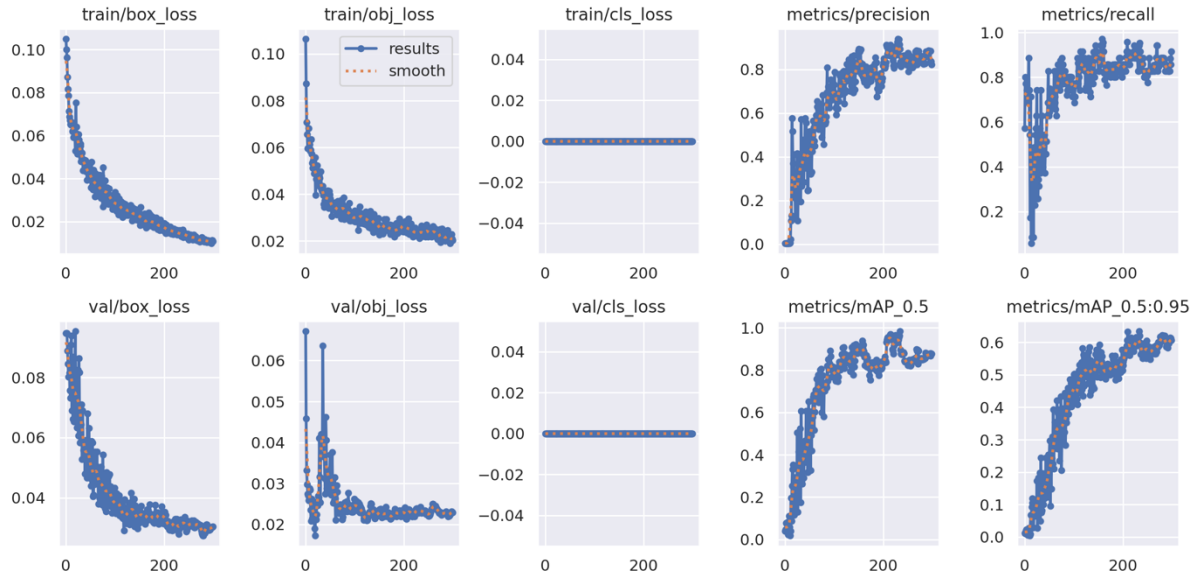

Figure 10: *YOLOv5m Torch-metric.*

Moreover, this model has many benefits over the YOLOv5x including faster inference time, minimum weight size and lesser layer complexity. This can be validated by running the evaluation (by using an evaluation script on the repo) on the YOLOv5m model where we get a validation accuracy of 0.86 by applying trained weights from the model training.

Output predicted bounding boxes of YOLOv5m are very close to the YOLOv5x model as shown in Figure 11. Thus, the YOLOv5m model is ideal for the deployment phase and demonstrates good accuracy and a reasonable model size.

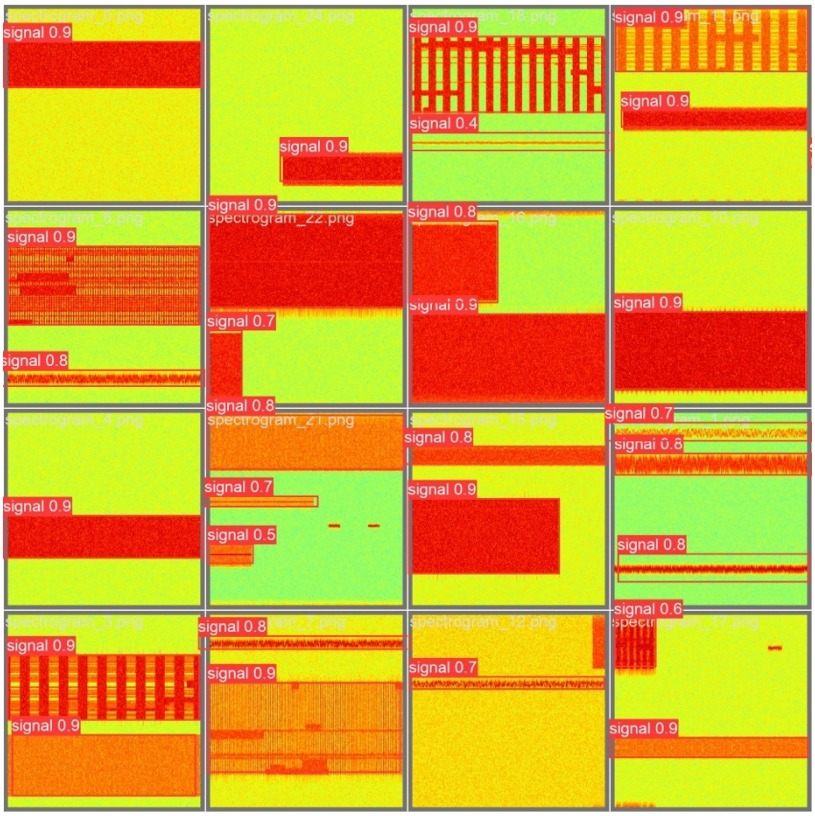

Figure 11: *YOLOv5m predicted labels.*

The experimentation with the convolution neural network algorithms can be finalised by selecting YOLOv5m as an all-rounder fast model that outperforms every other model tested so far. The next step is to explore the transformer domain to test the state of art Image Transformers and their capabilities to perform for our custom dataset.

5.2.4 DETR B0 Nano (Image Transformer)

An image transformer is a powerful approach when we are trying to detect multiple signals in a given spectrogram. One of the most recent advancements made by Facebook researchers, Carion *et al.,* (2020) is the discovery of the detection transformer (DETR) [24]. This DETR model consists of a backbone of ResNet-50 followed by a transformer encoder,

transformer decoder, and prediction heads. Originally, DETR is trained in Pytorch using the COCO dataset and has outperformed many two-staged object detection models like R-FCNN.

Authors of the TorchSig toolkit paper have performed some modifications to the original DETR model to customise it as per the dataset [4]. By changing the ResNet-50 backbone with an EfficientNet and the Detection Head with the XCiT Transformer an image transformer is established. This modification helps the authors to adjust backbone network complexity by selecting various scales from B0 to B7. Also, the scales of XCiT can be adjusted at the detection head, and the scales are set to the lowest (nano). As a result, the TorchSig toolkit gets introduced with three Image transformer variations namely the DETR-B0-Nano, DETR-B2-Nano, and DETR-B4-Nano. The models operate in the signal vision domain where the detection head calculates bounding boxes in a different convention. Unlike the vision domain, the signal domain uses {tc, fc, d, and B}; where tc is the centre time, fc is the centre frequency, d is the signal duration and B represents the bandwidth of the signal.

Hence the DETR models can be directly adapted by the TorchSig toolkit (Transfer Learning) and can be called using 'TorchSig.model' functions. The model is designed to undergo target transform and hence the data pipeline of the TorchSig dataset can be fed directly without creating a custom YOLO-based dataset.

For experimentation, we tried to reproduce the DETR B0 Nano for 50 epochs by following the example folders of the TorchSig repository. However, the accuracy of mAP50-90 remained very low while training and resulted in 0.3. While trying to test the pre-trained weights issues by TorchSig, the model accuracy does not change, and the pre-trained weights predicted wrong bounding boxes as shown in Figure 13. Furthermore, the model cannot be separately imported to Pytorch environments since it can only be used using TorchSig dependencies, making the model non-transferable in nature. The entire code for executing the DETR B0 nano can be found in Appendix 6.

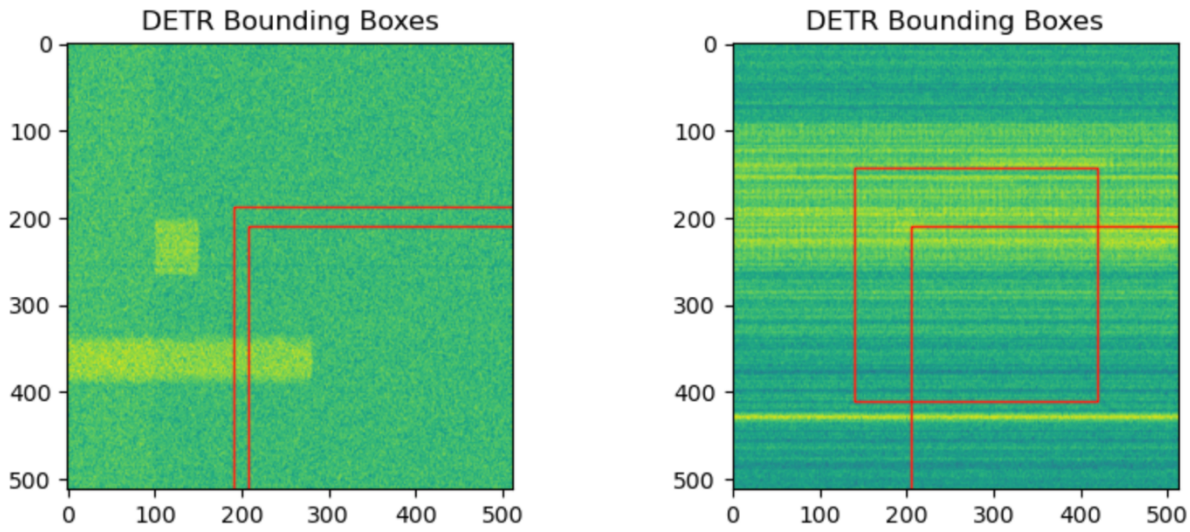

Figure 12: *DETR B0-nano predicted labels.*

Though there might be a high possibility that the other DETR network (DETR B2, and DETR B4) can have better performance, however, experiments on DETR B2 and B4 nano are not performed because of network size and complexity. Mainly due to Vitis AI support-layer limitation in the Quantization phase which restricts the XCiT framework from Quantization implementation resulting in a non-selection of this model types [23].

5.3 Cumulative Results of the Models

Summing up, we have explored the architecture and performances of both single-stage models and multi-stage models while keeping the Quantization phase in mind. The best architecture across single-stage and multi-stage models according to Torch metric criteria is YOLOv5m and has the most potential for performance improvement through fine-tuning. An overview of the model comparison can be seen in Table 1.

These results match our expectations like our estimate that there should have been a trade-off between the speed (NMS/Image) and accuracy (mAP50-90) of a model. This can be easily seen by comparing the YOLOv5x and YOLOv5s Torch metrics. From this entire comparison, we learn that for object detection using spectrograms, a light model will mostly struggle to perform and will suffer with high box loss resulting in overall lower mAP.

Table1: *Object detection model comparison for signal detection on TorchSig dataset*

Model	Box Loss	Precision	Recall	mAP 50	mAP 50-90	Weight Size (MB)	NMS/ Image (ms)
YOLOv5x	0.0070	0.918	0.971	0.956	0.891	166	30
YOLOv5m	0.0091	0.902	0.943	0.941	0.7	43	6.2
YOLOv5m (Pruned 0.2)	0.0082	0.967	0.886	0.897	0.813	32	5.7
YOLOv5s	0.019	0.533	0.771	0.719	0.37	14.5	7.2
YOLOv3m	0.0123	0.869	0.517	0.79	0.35	113	6.1
Scratch YOLOv3	0.3469	0.239	0.422	0.322	0.1	96	-
DETR B0 Nano	0.056	0.36	0.52	0.44	0.3	33	-

Finally, all the models are referenced and pulled from the original authors who have developed the algorithms and then we have modified it for our project experimentation purposes. Also, all the networks were in float32 format and are trained using AMD ROCm MI210 Graphical Processing Unit (GPU) from the native HACC cluster including custom YOLO dataset generation code.

Model Development

In the computer vision domain, native hardware like system on chip (SoC) and development boards often comes with low memory and storage space. Alternatively, most high-performing neural networks require high computation power to operate which creates a dilemma between the efficiency and cost of computational memory. The aim of major model deployment tasks is to get a high-efficiency model with the lowest computational speed, and this can be only achieved through model fine-tuning. For object detection algorithms fine tuning can be obtained through four different methods: Pruning, Quantization, Factorization and Distillation. Out of which we have applied pruning and quantization for this research project.

Pruning can be defined as the method to cut off the model weights or skip unnecessary convolutional layers, to help consume less computational power and energy without disturbing the network's performance metric. Pruning can be divided into two main types: Structured Pruning and Unstructured Pruning. Structured pruning tries to remove the building layers in a neural network which has a low impact on the output, whereas Unstructured pruning converts model weights and biases (that are closer to zero) into zero.

Thus, unstructured pruning might help to increase the computation expense of the selected model 'YOLOv5m'. By taking advantage of this technique, the YOLOv5m model can be easily fine-tuned to further improve the model performance. To apply pruning to the YOLOv5m model, the validation file of the repo (val.py) is to be modified by adding a pruning argument from the 'torch.nn' module to the model with a desired threshold. This unstructured pruning argument imposes iterative pruning of L2 regularization to the loss function (BCE for YOLOv5m) to push the weights to zero [28]. This helps the YOLOv5m model to convert from a dense network to a sparse network. For this task, we have selected a pruning of 0.2 threshold value, which means 20% of the model weights that tend to zero will be converted to zero. After adjusting the pruning code in the validation script, the entire script can be run using the trained weight to initiate the pruning of the model. The pruned model prediction labels can be found in Figure 13 to see how accurately the pruned model works.

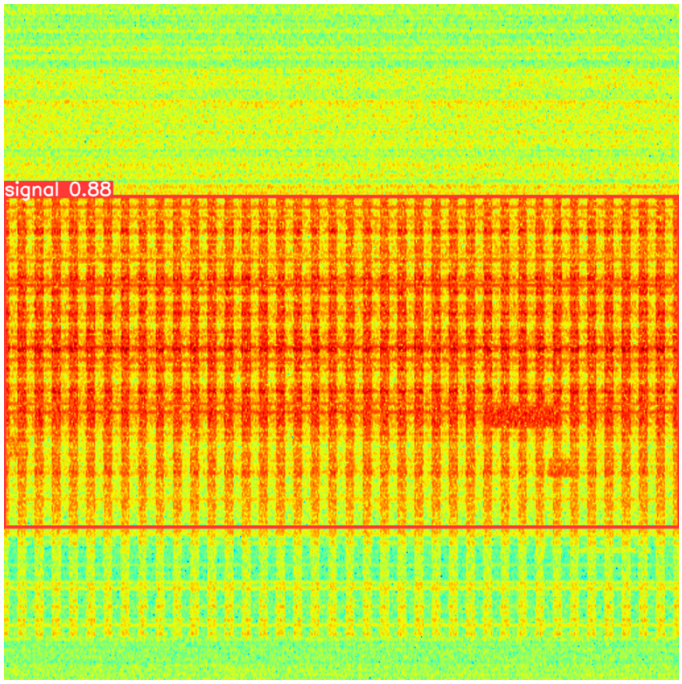

Figure 13: *Pruned YOLOv5m predicted labels.*

The result of applying pruning is quite satisfying since the pruned version of the model doesn't lose accuracy and the computational expense (GFLOPs) in comparison to the dense YOLOv5m model which reduces to half as seen in Figure 14.

```
val: data=/pub/scratch/devdas/ML_for_Spectral_Analysis_on_SoC/yolov5/data/data.yaml, weights=['myweights.pt'], batch_size=32, imgsz=640, conf_thres=0.001, iou_thr
es=0.6, max_det=300, task=val, device=, workers=8, single_cls=False, augment=False, verbose=False, save_txt=False, save_hybrid=False, save_conf=False, save_json=F
alse, project=runs/val, name=exp, exist_ok=False, half=False, dnn=False
YOLOv5 🚀 v7.0-196-gacdf73b Python-3.9.7 torch-2.0.1+rocm5.4.2 CUDA:0 (AMD Instinct MI210, 65520MiB)

Fusing layers...
YOLOv5m summary: 212 layers, 20852934 parameters, 0 gradients, 47.9 GFLOPs
Model pruned to 0.2 global sparsity
val: Scanning /pub/scratch/devdas/ML_for_Spectral_Analysis_on_SoC/Data/test/labe
                 Class     Images  Instances          P          R      mAP50
                   all         17         35      0.967      0.886      0.897      0.813
Speed: 0.4ms pre-process, 6.2ms inference, 5.7ms NMS per image at shape (32, 3, 640, 640)
Results saved to runs/val/exp12
```

Figure 14: *Pruned YOLOv5m results.*

This pruned YOLOv5m model maintains a mAP50-90 score of 0.813 which is very high and stable compared to the accuracy of the dense YOLOv5 model. Also, to further analyse the sparse YOLOv5m model, we have plotted the F1-Confidence curve and the Precision-Recall curve in Figure 15. The F1-Confidence curve demonstrates that there is a balance between the Precision and Recall value when the confidence threshold is between 0.2 to 0.4.

Thereby the pruned model showcases that in the overconfidence area, the model can perform very badly. Similarly, the Precision-Recall curve demonstrates an almost zero trade-off between precision score and recall score due to the balanced dataset. The overall performance of the model is impressive since the curve doesn't fluctuate much.

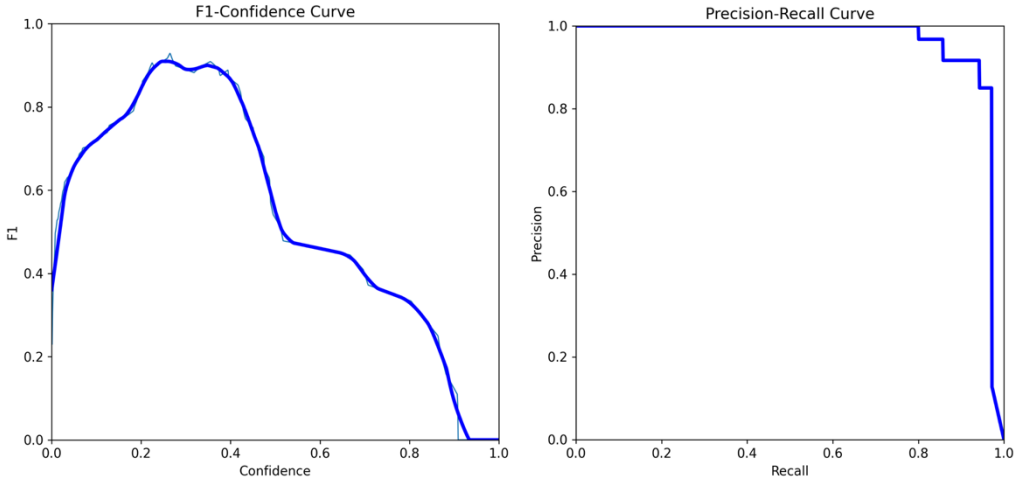

Figure 15: *F1-Confidence and Precision-Recall Curve for Pruned YOLOv5m.*

Although the model has been successfully pruned, the precision type is insufficient for the deployment phase and remains the same i.e., Float32, but most of the FPGAs and development hardware (lower-end devices) are designed to perform better with INT8 model format. So, to further reduce the model size and computational expense for easy model deployment on AMD hardware, the next step would be to quantize this pruned model.

Post Training Quantization

The fundamental steps to deploying an object detection model for real-time signal detection are explored in this section. Model Optimization is one of the first pivotal steps that ensure the model compatibility with the designated deployable hardware by matching the operator types of hardware and the model, before deploying any model. One of the key areas of Model Optimization is Post Training Quantization which highlights reducing memory consumption by 10x and converting the full-precision Float 32 bits values to lower-precision INT 8 bits values with considerable accuracy loss after the model has been fully trained [29] [30]. The post-training quantization requires a model optimizer to implement it over a model. The advantage of using post-training quantization over other quantization techniques is that it is easier to implement it on a model and analyzes the distribution of activations which can make it faster to implement.

The Model Optimizer is a software that is platform-specific and since this project is governed by AMD, we are using the Vitis AI platform to quantize our pruned YOLOv5m model. Vitis AI is a cutting-edge and robust comprehensive inference development software that is designed to specifically work along with AMD devices, FPGAs, SoC, GPU, DPU, development boards and accelerator cards. The Vitis AI model quantizer would be perfect to use on a pruned YOLOv5m model because it will help the model to get more compact and amenable to deploy on AMD-based hardware fulfilling our ultimate goal for this research. Hence, Vitis AI is a powerful tool to design the entire signal detection inference. Contradictorily, the Vitis AI quantizer can be challenging to implement a custom ML model, example for this research the Vitis AI will not support the forward function mask segment instance of the ‘yolo’ file in the YOLOv5 repo, so it needs to be changed to a simple inference output forward function.

7.1 AMD Resources and Vitis AI Platform Setup

The Vitis AI software module can only be implemented using the official Docker images (x86) from the Vitis AI websites and is strictly supported on Linux Operated machines only. For comfortable access to various AMD hardware for our research, we have registered to the Heterogeneous Accelerated Compute Clusters (HACC) program to execute the entire project

for accessing AMD resources. The reason to select the HACC program over implementing the project into a native machine is the flexibility to deploy the final signal detection algorithm on edge devices, data centre accelerator cards and even cloud deployments for free.

The HACC server of ETH Zurich provides high-end AMD GPUs such as MI210, reconfigurable accelerator cards like Alveo VCK5000 and FPGA hardware like Alveo U55C. Figure 16 portrays the block diagram of the HACC box where all the accelerator cards and GPUs are connected to an ethernet interface (100 GbE).

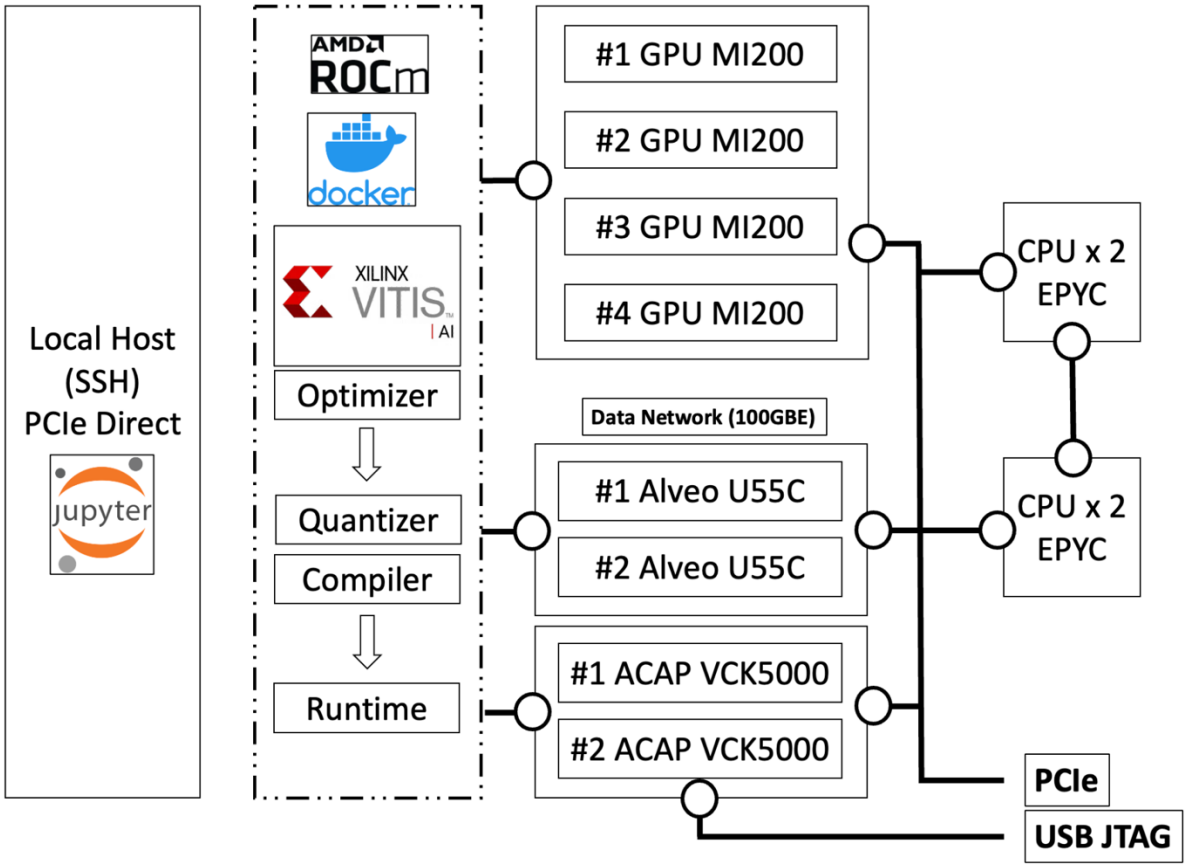

Figure 16: *Block Diagram of ETH Zurich HACC server.*

To access this HACC box from the local host, we use a secure shell (ssh) port via ETH Zurich VPN. The ssh helps to establish a connection between the HACC server and the local machine and Figure 17 shows the establishment of ssh connection to the server.

```
(base) devjyotidas@inf-docking-vpn-1-99 ~ % ssh -L 8888:localhost:8888 devdas@hacc-box-03.ethz.ch
devdas@jumphost.inf.ethz.ch's password:
hacc-box-03.ethz.ch - Ubuntu 20.04 (x86_64) [systems_cluster]
+-------------------------------------------------------------+
|                                                             |
|     Heterogeneous Accelerated Compute Clusters (HACC)       |
|            AMD Xilinx University Program (XUP)              |
|                                                             |
|         The HACC - XUP is a special initiative to           |
|  support novel research in adaptive compute acceleration    |
|                for high-performance computing               |
|                                                             |
|          Learn more at https://www.amd-haccs.io             |
|                                                             |
|         ---------------------------------------             |
|                                                             |
|     Institute for Computing Platforms - Systems Group       |
|                                                             |
|                      ETH Zürich - 2022                      |
|                                                             |
+-------------------------------------------------------------+
```

Figure 17: *ssh local host connect to the ETH Zurich HACC server.*

Within the HACC server, there is a native user directory to load the datasets or dependencies to perform various ML operations. The AMD GPUs in the HACC have ROCm as a parallel computing software to directly access the GPUs. To initiate the Vitis AI Quantizer Pytorch framework 'vai_q_pytorch', the Vitis AI repository is cloned and the docker container within the Vitis AI directory is called to pull the Vitis-AI PyTorch ROCm image (where the GPU backend is managed by the ROCm layer which acts similarly to the CUDA base by Nvidia) and opened to create the docker environment as shown in Figure 18. Appendix 8 shows the GPU hardware and software availability inside the HACC box.

```
(base) devdas@hacc-box-03:/mnt/scratch/devdas/Vitis-AI$ ./docker_run.sh xilinx/vitis-ai-pytorch-rocm:latest
WARNING: Please start 'docker_run.sh' from the Vitis-AI/ source directory
latest: Pulling from xilinx/vitis-ai-pytorch-rocm
Digest: sha256:65c55c064a36902bb8a75012e01414860b99b3619a66280550b917e2fb34141a
Status: Image is up to date for xilinx/vitis-ai-pytorch-rocm:latest
docker.io/xilinx/vitis-ai-pytorch-rocm:latest
Setting up devdas 's environment in the Docker container...
Running as vitis-ai-user with ID 0 and group 0

==========================================

__      ___ _   _                 _____
\ \    / (_) | (_)          /\   |_   _|
 \ \  / / _| |_ _ ___ _____ /  \    | |
  \ \/ / | | __| / __|_____/ /\ \   | |
   \  /  | | |_| \__ \     / ____ \ _| |_
    \/   |_|\__|_|___/    /_/    \_\_____|

==========================================

Docker Image Version: ubuntu2004-3.5.0.306    (ROCM)
Vitis AI Git Hash: 6a9757a
Build Date: 2023-06-26
WorkFlow: pytorch
```

Figure 18: *Vitis AI Docker Image on the ETH Zurich HACC server.*

7.2 Vitis AI Quantization

Once the Vitis AI Docker is set up and running, we upload the datasets and models to the same directory as the Vitis AI repository. Since the pruned model is ready to be quantized, we developed a quantised Python file to convert the pruned model to a quantised model which can be found in Appendix 9. In this Python file, we created a 'CustomImageDataset' class to manage the custom YOLO-based dataset pipeline (but without the labels just passing the validation images to the data loaders) and have defined a quantization function that takes model-pertained weights, dataset pipeline, quantization mode, and the YOLOv5m model (which is named as DetectMultiBackend) as input to produce an output 'quant config()' file via 'torch.nndct' module of Vitis AI (the nndct module is a Vitis AI quantizer module that comes with the Vitis AI docker.). This config file holds the whole quantized YOLOv5m model configuration with INT8 weights and can only be generated when the quantization mode is set to 'calib'.

Therefore, in the 'calib' mode of the quantization, two files get generated: the quantized YOLOv5m model file which is saved as a python file named 'DetectMultiBackend.py' and has an INT8 converted model. And the other file which is generated is the 'quant info' file which has all the respective quantized weights. Now, to capture the activation statistics and the accuracy of the quantized model the quantization script is configured in test mode.

In the test mode, the quantization script uses the 'non-max suppression' function to calculate the predicted bounding box across each iteration of inference inside the 'quantize' function. Now, the output of the non-max suppression function is in tensor and is not normalized. Thus, it is normalized first, and the detection labels and boxes are calculated for each image, this can be observed in Figure 19.

```
vitis-ai-user@hacc-box-03:/workspace/yolov5$ python qat.py --build_dir runs/train/exp --quant_mode test --weights FINAL.pt --dataset Data/test/

[VAIQ_NOTE]: Loading NNDCT kernels...

--------------------------------------------------
PyTorch version :  1.13.1+git3aa2ef3
3.8.6 | packaged by conda-forge | (default, Oct  7 2020, 19:08:05)
[GCC 7.5.0]
--------------------------------------------------
 Command line options:
--build_dir    :  runs/train/exp
--quant_mode   :  test
--weights    :  FINAL.pt
--dataset    :  Data/test/
--------------------------------------------------
Fusing layers...
YOLOv5m summary: 212 layers, 20852934 parameters, 0 gradients

Image 17
Image 17 Detections:
  Co-ordinates: tensor([ 6.87500,  6.12500, 20.12500, 19.37500], device='cuda:0'), Confidence: 0.8778125047683716, Class Index: 0.0
  Co-ordinates: tensor([ 4.12500,  3.50000, 13.87500, 14.00000], device='cuda:0'), Confidence: 0.604687511920929, Class Index: 0.0
  Co-ordinates: tensor([ 2.37500,  4.00000,  8.62500, 10.00000], device='cuda:0'), Confidence: 0.11874999850988388, Class Index: 0.0
 mAP@50-90: 0.5625
Inference time: 1.33 ms


[VAIQ_NOTE]: DetectMultiBackend_int.pt is generated.(quantize_result/DetectMultiBackend_int.pt)

[VAIQ_NOTE]: DetectMultiBackend_int.onnx is generated.(quantize_result/DetectMultiBackend_int.onnx)

[VAIQ_NOTE]: =>Converting to xmodel ...

[VAIQ_NOTE]: =>Successfully convert 'DetectMultiBackend' to xmodel.(runs/train/exp/quant_model/DetectMultiBackend_int.xmodel)
```

Figure 19: *Vitis AI Quantization in 'test' mode.*

After iterating for all the images, a cumulative mAP50-90 is calculated for all the image instances. To get an inference on the model a batch size of 17 is used from the validation set of the custom YOLO-based dataset and achieved an overall mAP50-90 of 0.5625 and 1.33 milliseconds of NMS/image which is very good considering there is only a 20% decrease in performance for the amount of data (240 samples) that we have passed to this quantized model. The test configuration of the quantization script also produces different file formats of the quantized model like '.pt' for the Pytorch framework and '.onnx' for the ONNX framework. Simultaneously, a '.xmodel' format is also generated, this framework is supported for all the AMD hardware and is used to compile the quantized model for a Data Processing Unit (DPU) and can be accessed from Appendix 10. With this, the Vitis AI Quantization is successfully implemented with a good mean average precision score and an ideal inference time that can be used to compile and deploy it in real-time AMD hardware.

Model Deployment

The HACC cluster consists of an Adaptive Compute Acceleration Platform (ACAP) which combines FPGA, SoC, and hardware acceleration. These are commonly known as accelerator development cards that are highly efficient to simulate an ML model and can run simulated inferences. That means, should the ML model prove to be effectively running on an ACAP, the likelihood of its smooth implementation is high in a local FPGA or SoC hardware.

The ETH Zurich HACC has the Versal VCK5000 which is a state of art ACAP belonging to the Xilinx VC1920 family and uses a DPU architecture of DPUCVDX8H which is a high-performance CNN inference accelerator.

To compile the quantized model (.xmodel) framework a Vitis AI compiler is used. The Vitis AI compilers are called in the terminal window of the docker utilizing the 'vai_q_pytorch' method, where the 'vai_q_pytorch' takes the quantized '.xmodel', and the desired architecture path for compilation and then produces a 'my_model.xmodel' file at the output which is DPU deployable. The Vitis AI compiler when specified with a DPU architecture of DPUCVDX8H, maps the quantized YOLOv5 model into a highly optimised DPU instruction sequency. The instruction sequence then calibrates the quantized model into a DPU-deployable model as shown in Figure 20.

```
* VITIS_AI Compilation - Xilinx Inc.
**********************************************
[UNILOG][INFO] Compile mode: dpu
[UNILOG][INFO] Debug mode: null
[UNILOG][INFO] Target architecture: DPUCVDX8H_4PE_MISCDWC
[UNILOG][INFO] Graph name: DetectMultiBackend, with op num: 616
[UNILOG][INFO] Begin to compile...
[UNILOG][INFO] Total device subgraph number 5, DPU subgraph number 1
[UNILOG][INFO] Compile done.
[UNILOG][INFO] The meta json is saved to "/workspace/./meta.json"
[UNILOG][INFO] The compiled xmodel is saved to "/workspace/.//my_model.xmodel"
[UNILOG][INFO] The compiled xmodel's md5sum is 62d0c82db1d79ac8b74c0f81846d90a4, and has been saved to "/workspace/./md5sum.txt"
```

Figure 20: *Vitis AI Compiling VCK5000.*

Finally, the model is deployed on a Versal VCK5000 where it can be executed with 1-millisecond runtime (NMS/image). This is very close to real-time execution and has the potential to work on an RF-SoC with live data feed.

If the quantized model worked on the accelerator card Versal VCK5000, it shall be able to deliver the same results on a DPU on the RFSoC. The reason for an easy port to other DPUs is that the other DPUs share a very similar architecture to that of a VCK5000 making it compatible. In a nutshell, the compatibility of compiled sparse YOLOv5m looks promising enough to deliver good frames per second for static pipelines across various FPGAs.

Discussion

Based on the research motivation, the project objective was divided into three stages: firstly, to create a custom RFML spectrogram dataset; secondly, to develop a CNN-based object detection model to analyse the custom RFML spectrogram dataset, and lastly, to deploy this RFML spectrogram on an RF SoC/ a development board/ an accelerator card for both real-time signal and static signal. For fulfilling this project's objectives, the following process was undertaken which helped us to gather a wider perspective in our research domain as explained hereunder.

In pursuit of accomplishing the primary objective to create a custom RFML spectrogram dataset, several approaches were tried like collecting simulated sample data by varying frequency time and phase through SDR and a total of 100 images were collected. But since the dataset was extremely straight-forward and there were no impairments, this dataset had to be rejected leading to a loss of time in an extremely time-crunched project. Despite this setback, the flame of enthusiasm and inquisitiveness was sustained in this field which served to propel further and explore new datasets like the Fronhoffer Wideband dataset [27]. Yet, we had to reject this dataset because of the difficulty in its extraction due to the data size being over 180 GB, causing further hurdles in this project. These trial-and-error methods of identifying datasets served as a stepping-stone to understanding the specific type of dataset that function optimally with the requirements of the project. Noting the feedback from this journey, we were then able to successfully identify the TorchSig dataset and progress to the subsequent objective.

Proceeding towards attaining the second objective of developing a CNN-based object detection model to analyse the custom RFML spectrogram dataset, we initiated the implementation of a transfer learning approach taking YOLOv5n model with pre-defined weights of TorchSig models. However, the model weights were not effective, and the model underperformed.

Afterwards, we executed DETR B0 nano with pre-trained weights of TorchSig, but the predicted bounding boxes were producing wrong labels as well as transferring learning of the DETR model to a new dataset was very challenging computationally, because of this we did not try to implement further variations of DETR like B2 and B4. A notable learning in this process was that using a GPU before training is a necessity, considering these TorchSig models were executed in an arm-based CPU and presented many difficulties due to low computation. Subsequently, we implemented YOLOv3m by Ultralytics and the model was insufficient to learn the custom dataset. And then we also tried to implement YOLOv3 from scratch but again to our dismay the performance and inference time of that model was even way lower than the pre-trained Ultralytics YOLOv3m. Consolidating the lessons learnt from these trials, we realised continuing hereon with the testing of complex YOLO family models like YOLOv5x, YOLOv5s, YOLOv5m would be fruitful and indeed this attempt turned out to be productive leading us to finalise working on the YOLOv5m model.

Ultimately, we strived to realize the third objective of deploying the RFML spectrogram on an RF SoC/ a development board/ an accelerator card for both real-time signal and static signal. To gain this we undertook efforts to import the Vitis AI GitHub Repo and call Vitis AI image docker to the local machine, but as we were employing the arm-based CPU, we recognized that Vitis AI only supports x86 Linux Ubuntu machines. This stood out as the key lesson we procured from the experimentation experience, driving us to attempt the implementation of the Vitis AI on an Ubuntu machine. Despite this attempt, we observed that the RoCm workflow base was missing, and this helped us to comprehend that the ROCm base is necessary to deploy the model on RF SoC. Thereupon, finally, we made a deliberate choice to utilize the HACC Cluster which was offered by the ETH Zurich AMD program.

The entire model workflow from training to deployment was carried out via HACC Cluster, realising that both static and real-time signals can be analysed giving us a positive result in turn leading to the achievement of the third objective. The journey of achieving all three objectives was possible due to vigorous project management like using Gantt Chart to adhere well to the project lifecycle stages and working on a phase-based milestone tracker providing ample buffer time to quickly learn and adapt. Keeping a change agility mindset and using agile

principles also proved to be highly efficient to effectively tackle the emerging issues while undergoing each phase thereby leading to fruitful outcomes.

Despite achieving all three objectives with our best efforts, we were unable to deliver and test the model on the local RFSoC device. One of the major challenges faced was the deployment of the compiled model on an AMD RFSoC which would require us to design a binary dataflow pipeline that processes the data and feeds it to the model with an efficient mechanism to manage overflow. Additionally, integration of the entire data pipeline with the model (YOLOv5m) using the RFSoC API can be an extremely time-consuming task given the timeframe of the entire project. We recognize the importance of this step for research in this field of RFML and would aim to continue further research and execution of the RFSoC deployment scope in the future by managing the time constraints.

Building upon these outcomes and the abovementioned issues, we are of the opinion that substantial opportunities exist for forthcoming research endeavours in the field of RFML spectrogram. A few of the opportunities are mentioned hereafter. Even though the TorchSig toolkit is still in a development beta state, the code that we have developed through this project will help to generate both detection and classification datasets to outsource the TorchSig generator for different types of object detection models. Next, the YOLOv5m architecture that is used for the signal detection task can be easily used as a transfer learning approach for the signal classification task as well and holds a future potential to analyse 53 different classes of TorchSig dataset toolkit. Another area to research is in connection with the fact that YOLOv5m was released in 2020 by Ultralytics and since then much more complex smaller and better architecture has been invented namely, the YOLOv8 family and YOLO-NAS. However, due to layer complexities and activation functions not being compatible with the Vitis AI environment, these models have not been experimented on to date. But there is immense scope if the model architecture is changed to support Vitis AI then there are high chances that they can even outperform YOLOv5m.

An additional avenue for exploration lies in the quantization layer of Vitis AI quantizer which supports both post-training quantization and quantization-aware training. The post-training quantization may induce lower precision and lossy process after quantizing the YOLOv5m and to diminish this loss and significantly level up the accuracy, quantization-aware training can be performed. In quantization-aware training the forward pass simulates low precision behaviour, this leads to a certain degree of quantization error where the optimizer tries to minimise this quantization error by adjusting the parameters accordingly. This procedure can result in lossless quantization and has enormous possibilities for YOLOv5m in the future.

By endeavouring into these unexplored realms of study, the RFML research communities can enhance their understanding of RFML spectrogram-based object detection and open doors for innovative data-driven insights which will prove to be useful to the industry at large.

Conclusion

The applications of using machine learning in spectrum management are ever-increasing due to the robustness to adapt to a continuously changing spectrum environment in real-time conditions. Most importantly the property of neural networks to detect complex signals automatically with a high efficiency makes machine learning techniques the future adaptation to the spectrum analysers. With this research project, we tried to understand the state of art model architectures used in this RFML research space and what are the potential literature voids that can be experimented on.

By performing an extensive literature review it is safe to say that there is very less focus on object detection models to automate signal analysis work pursued on standard wide-band spectrogram datasets that cover a broad range of modulation signals. We have tried to fill in this potential gap in research by not only experimenting with the state of art RFML spectrogram dataset and object detection models but also deploying them on low-power hardware. Moreover, noting that as per the literature review, the researchers have not been able to thoroughly establish the quantization analysis of the latest object detection architecture for the RF spectrogram, the project aimed to perform a detailed quantization analysis on object detection architectures for the RFML workspace.

To achieve the project goal, the process involved discovering that the TorchSig toolkit is a wonderful software which can outperform even the traditional RadioML dataset for providing a wide range of modulation types of synthetic signal generation. Also, key features to add impairments gives extra points to use this dataset for model experimentation. Additionally, the dataset becomes ideal when we try to generate a customized real-time spectrogram specifically for this project. Therefore, we developed an algorithm to generate the customized spectrogram.

Furthermore, to utilize the most out of the customized dataset, the model selection procedure is undertaken to examine the best neural architecture available judging by the criteria of model speed and accuracy. By exploring deeper into neural networks, we realized that the object detection model relatively performs better than other model techniques as real-time signals sometimes overlap with each other and the model without bounding boxes predictions does not recognise the difference in signals and hence fails to classify the signals. While during the model training, we learnt that using pre-trained weights for training is more helpful since building a model from scratch might burn out more computational resources and can still provide low accuracy. The experiment shows us that the model with a balanced nature should be selected because the signal detection tasks focus on getting both high speed and accuracy. Therefore, after applying experimentation and a fast failure approach, this project examined using niche low-latency object detection algorithms like YOLOv3m, YOLOv3-from-scratch, YOLOv5s, YOLOv5m, YOLOv5x, and DETR B0 nano to identify which model is successful in accomplishing both the aspects of high speed and accuracy. Comparing the output of each model, YOLOv5m turned out to be the best-fit option and that is the core reason why YOLOv5m was successfully deployed.

However, for the successful deployment of the YOLOv5m model, it is highly recommended that optimization of this model is adopted using a technique called pruning since it reduces the model weight size. Post the conduction of this pruning, further quantization should be applied to assist in reducing the model size using the AMD-specific platform: Vitis AI. This platform has its own quantizer that runs quantization through the layers of the YOLOv5m model. Nevertheless, prior to running quantization on this platform, attention must be paid to checking Hardware Support, Vitis AI Quantizer Support, and GPU availability. After checking these parameters, the YOLOv5m model needs to undergo architectural changes according to Vitis AI layer support to be applicable for quantization from the Vitis AI quantizer. This Vitis AI quantizer converts the dense YOLOv5m (float32) model to the sparse YOLOv5m (INT8) model. Once this step is done, the model size is reduced by ten times and is ready to be implemented on low-power AMD hardware.

Since all the above actions were conducted using GPUs of a HACC cluster and it is easier to export the quantized YOLOv5m (sparse) model to the accelerator card installed in this cluster, thereby we deployed this model on the accelerator card (VCK5000). This card is very versatile and if a model is deployed on this card, then RF SoC ought to generate the same level of performance as that of VCK5000. Following the quantization and deployment, we achieved a remarkable result with a speed of 1 millisecond (pprox.. 70 frames per second) and an accuracy of 65%. This result can be considered a good score and this model has the potential to work on a real-time signal on an RF SoC board.

In summary, this project has conducted a thorough analysis of object detection models for signals detection using spectrograms and has found the YOLOv5m model as an apt option for this research. And this project will support RFML communities and researchers to venture deeper into the domain of RFML spectrogram analysis.

References

[1] S. Wang, Y. E. Sagduyu, J. Zhang, and J. H. Li, “Spectrum shaping via network coding in cognitive radio networks,” IEEE Xplore, Apr. 01, 2011. https://ieeexplore.ieee.org/document/5935190

[2] T. J. O’Shea, J. Corgan, and T. C. Clancy, “Convolutional Radio Modulation Recognition Networks,” Engineering Applications of Neural Networks, pp. 213–226, 2016, doi: https://doi.org/10.1007/978-3-319-44188-7_16.

[3] Y. Sagduyu, S. Soltani, T. Erpek, Y. Shi, and J. Li, “A Unified Solution to Cognitive Radio Programming, Test and Evaluation for Tactical Communications,” IEEE Communications Magazine, vol. 55, no. 10, pp. 12–20, Oct. 2017, doi: https://doi.org/10.1109/mcom.2017.1700222.

[4] L. Boegner, G. Vanhoy, P. Vallance, M. Gulati, D. Feitzinger, B. Comar, and R. D. Miller, “Large Scale Radio Frequency Wideband Signal Detection & Recognition,” arXiv.org, 2022, doi: 10.48550/arXiv.2211.10335.

[5] T. J. O’Shea, K. Karra, and T. Charles Clancy, “Learning to communicate: Channel auto-encoders, domain specific regularizers, and attention,” International Symposium on Signal Processing and Information Technology, Aug. 2016, doi: https://doi.org/10.1109/isspit.2016.7886039.

[6] Y. Long, H. Li, H. Yue, M. Pan and Y. Fang, "Spectrum utilization maximization in energy limited cooperative cognitive radio networks," 2014 IEEE International Conference on Communications (ICC), Sydney, NSW, Australia, 2014, pp. 1466-1471, doi: 10.1109/ICC.2014.6883528.

[7] A. Ali and W. Hamouda, "Advances on Spectrum Sensing for Cognitive Radio Networks: Theory and Applications," in IEEE Communications Surveys & Tutorials, vol. 19, no. 2, pp. 1277-1304, Second quarter 2017, doi: 10.1109/COMST.2016.2631080.

[8] A. Gharib, W. Ejaz and M. Ibnkahla, "Enhanced Multiband Multiuser Cooperative Spectrum Sensing for Distributed CRNs," in IEEE Transactions on Cognitive Communications and Networking, vol. 6, no. 1, pp. 256-270, March 2020, doi: 10.1109/TCCN.2019.2953661.

[9] H. Qi, X. Zhang and Y. Gao, "Channel Energy Statistics Learning in Compressive Spectrum Sensing," in IEEE Transactions on Wireless Communications, vol. 17, no. 12, pp. 7910-7921, Dec. 2018, doi: 10.1109/TWC.2018.2872712.

[10] W. Fan, L. He, Y. Long, H. Ju and S. Lin, "CNN-Based Distributed Learning for Spectrum Sensing in Cognitive Radio Networks," 2021 IEEE/CIC International Conference on Communications in China (ICCC), Xiamen, China, 2021, pp. 1137-1142, doi: 10.1109/ICCC52777.2021.9580342.

[11] T. J. O'Shea, T. Roy and T. C. Clancy, "Over-the-Air Deep Learning Based Radio Signal Classification," in IEEE Journal of Selected Topics in Signal Processing, vol. 12, no. 1, pp. 168-179, Feb. 2018, doi: 10.1109/JSTSP.2018.2797022.

[12] A. Vagollari, V. Schram, W. Wicke, M. Hirschbeck and W. Gerstacker, "Joint Detection and Classification of RF Signals Using Deep Learning," 2021 IEEE 93rd Vehicular Technology Conference (VTC2021-Spring), Helsinki, Finland, 2021, pp. 1-7, doi: 10.1109/VTC2021-Spring51267.2021.9449073.

[13] H. N. Nguyen, M. Vomvas, T. D. Vo-Huu and G. Noubir, "WRIST: Wideband, Real-time, Spectro-Temporal RF Identification System using Deep Learning," in IEEE Transactions on Mobile Computing, doi: 10.1109/TMC.2023.3240971.

[14] L. J. Wong, W. H. Clark, B. Flowers, R. M. Buehrer, W. C. Headley and A. J. Michaels, "An RFML Ecosystem: Considerations for the Application of Deep Learning to Spectrum Situational Awareness," in IEEE Open Journal of the Communications Society, vol. 2, pp. 2243-2264, 2021, doi: 10.1109/OJCOMS.2021.3112939.

[15] T. L. Truong, N. T. Le and W. Benjapolakul, "An Application of Deep Learning YOLOv5 Framework to Intelligent Radio Spectrum Monitoring," 2022 37th International Technical Conference on Circuits/Systems, Computers and Communications (ITC-CSCC), Phuket, Thailand, 2022, pp. 1-4, doi: 10.1109/ITC-CSCC55581.2022.9894862.

[16] J. Wicht, U. Wetzker and A. Frotzscher, "Deep Learning Based Real-Time Spectrum Analysis for Wireless Networks," European Wireless 2021; 26th European Wireless Conference, Verona, Italy, 2021, pp. 1-6.

[17] S. Soltani, Y. E. Sagduyu, R. Hasan, K. Davaslioglu, H. Deng, and T. Erpek, “Real-Time and Embedded Deep Learning on FPGA for RF Signal Classification,” IEEE Xplore, Nov. 01, 2019. https://ieeexplore.ieee.org/document/9021098.

[18] 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, Nov. 2022, doi: https://doi.org/10.1109/MM.2022.3202091.

[19] F. Jentzsch, Y. Umuroglu, A. Pappalardo, M. Blott and M. Platzner, "RadioML Meets FINN: Enabling Future RF Applications With FPGA Streaming Architectures," in IEEE Micro, vol. 42, no. 6, pp. 125-133, 1 Nov.-Dec. 2022, doi: 10.1109/MM.2022.3202091.

[20] A. Ushiroyama, M. Watanabe, N. Watanabe and A. Nagoya, "Convolutional neural network implementations using Vitis AI," 2022 IEEE 12th Annual Computing and Communication Workshop and Conference (CCWC), Las Vegas, NV, USA, 2022, pp. 0365-0371, doi: 10.1109/CCWC54503.2022.9720794.

[21] G. Jocher et al., “ultralytics/yolov5: v7.0 - YOLOv5 SOTA Realtime Instance Segmentation,” Zenodo, Nov. 22, 2022. https://doi.org/10.5281/zenodo.7347926

[22] J. Redmon and A. Farhadi, “YOLOv3: An Incremental Improvement,” arXiv.org, 2018. https://arxiv.org/abs/1804.02767

[23] Vitis AI User Guide, “AMD Adaptive Computing Documentation Portal,” UG1414 (v3.0), docs.xilinx.com. https://docs.xilinx.com/viewer/book-attachment/kKd~l_NZ4zoS2LugELQQ0Q/5LDZhuylUJ0TulQvqJ~1lQ

[24] N. Carion, F. Massa, G. Synnaeve, N. Usunier, A. Kirillov, and S. Zagoruyko, “End-to-End Object Detection with Transformers,” arXiv:2005.12872 [cs], May 2020, Available: https://arxiv.org/abs/2005.12872

[25] L. Boegner, G. Vanhoy, P. Vallance, M. Gulati, D. Feitzinger, B. Comar, and R. D. Miller, “Large Scale Radio Frequency Signal Classification,” arXiv.org, 2022, doi: 10.48550/arXiv.2207.09918.

[26] S. Tridgell, D. Boland, P. H. W. Leong, R. Kastner, A. Khodamoradi, and Siddhartha, “Real-time Automatic Modulation Classification using RFSoC,” IEEE Xplore, May 01, 2020. https://ieeexplore.ieee.org/document/9150443

[27] N. West, T. O'shea, and T. Roy, “A Wideband Signal Recognition Dataset.” Accessed: Aug. 18, 2023. [Online]. Available: https://arxiv.org/pdf/2110.00518.pdf

[28] T. Liang, J. Glossner, L. Wang, S. Shi, and X. Zhang, “Pruning and Quantization for Deep Neural Network Acceleration: A Survey,” arXiv:2101.09671 [cs], Jun. 2021, Available: https://arxiv.org/abs/2101.09671

[29] B. Jacob et al., “Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference,” arXiv:1712.05877 [cs, stat], Dec. 2017, Available: https://arxiv.org/abs/1712.05877

[30] M. Nagel, M. Fournarakis, R. A. Amjad, Y. Bondarenko, M. van Baalen, and T. Blankevoort, “A White Paper on Neural Network Quantization,” arXiv:2106.08295 [cs], Jun. 2021, Available: https://arxiv.org/abs/2106.08295

Appendices

Appendix 1

Code for TorchSig Dataset Analysis:

https://github.com/Er-Devjyoti/ML_for_Spectral_Analysis_on_SoC/blob/main/Data%20Analysis%20and%20Preprocessing/Data%20Analysis.ipynb

Appendix 2

Code for Detection Dataset Creation:

https://github.com/Er-Devjyoti/ML_for_Spectral_Analysis_on_SoC/blob/main/Dataset%20Generation/Detection%20Dataset%20Creation.ipynb

Code for CSV Generator:

https://github.com/Er-Devjyoti/ML_for_Spectral_Analysis_on_SoC/blob/main/Model%20from%20scratch/csv_generator.ipynb

Data.yaml file:

https://github.com/Er-Devjyoti/ML_for_Spectral_Analysis_on_SoC/blob/main/Model%20from%20scratch/data.yaml

Appendix 3

Code for Ultralytics YOLOv3m:

https://github.com/Er-Devjyoti/ML_for_Spectral_Analysis_on_SoC/blob/main/Yolov3m/YOLOv3m.ipynb

Appendix 4

Code for YOLOv3 from scratch:

https://github.com/Er-Devjyoti/ML_for_Spectral_Analysis_on_SoC/blob/main/Model%20from%20scratch/Model.ipynb

Appendix 5:

Code for training and validating YOLOv5x and YOLOv5m Ultralytics model:

https://github.com/Er-Devjyoti/ML_for_Spectral_Analysis_on_SoC/blob/main/YOLOv5/YOLOv5x.ipynb

Code for training and validating YOLOv5s Ultralytics model:

https://github.com/Er-Devjyoti/ML_for_Spectral_Analysis_on_SoC/blob/main/YOLOv5/YOLOv5.ipynb

Appendix 6:

Code for DETR B0 nano:

https://github.com/Er-Devjyoti/ML_for_Spectral_Analysis_on_SoC/blob/main/Image%20Transformer/DETRBOnano.ipynb

Appendix 7: (Python code for YOLOv5m Ultralytics model changes to adapt Vitis AI support)

common.py:

https://github.com/Er-Devjyoti/ML_for_Spectral_Analysis_on_SoC/blob/main/YOLOv5/common.py

experimental.py:

https://github.com/Er-Devjyoti/ML_for_Spectral_Analysis_on_SoC/blob/main/YOLOv5/experimental.py

train.py:

https://github.com/Er-Devjyoti/ML_for_Spectral_Analysis_on_SoC/blob/main/YOLOv5/train.py

val.py:

https://github.com/Er-Devjyoti/ML_for_Spectral_Analysis_on_SoC/blob/main/YOLOv5/val.py

yolo.py:

https://github.com/Er-Devjyoti/ML_for_Spectral_Analysis_on_SoC/blob/main/YOLOv5/yolo.py

Appendix 8:

System Check for RoCm and Vitis Ai support on HACC:

https://github.com/Er-Devjyoti/ML_for_Spectral_Analysis_on_SoC/blob/main/YOLOv5/SYSTEMCHECK.ipynb

Appendix 9:

Quantization code for YOLOv5m:

https://github.com/Er-Devjyoti/ML_for_Spectral_Analysis_on_SoC/blob/main/YOLOv5/qat.py

Appendix 10:

Quantization models in various formats:

https://github.com/Er-Devjyoti/ML_for_Spectral_Analysis_on_SoC/tree/main/Quantized%20Model