DSTA

Internship Project

Report and Documentation

RF Signal Classification using Deep Learning on the edge

30 November 2020 - 30 December 2020

Sim Shang En

sim@shangen.org

Table of Contents

1. Abstract

We demonstrate the embedded implementation of a deep learning-based RF modulation classifier that is implemented on the embedded ARM CPU of the Raspberry Pi 2 connected to the software-defined radio (SDR) platform, ADALM-PLUTO. Supported by low-power embedded computing, the received signals are classified into different modulation types in real-time. The deep neural network that is used for the RF signal classifier runs directly on the Raspberry Pi without an external Internet connection. In the demonstration setup, a HackRF One radio transmits signals with different modulation types and ADALM-PLUTO with Raspberry Pi classifies each received signal (I/Q samples) to its modulation type.

2. Introduction

2.1. Background

Radio frequency (RF) signal classification is an important task for wireless systems to be able to learn and characterize the underlying channel, interference, and traffic effects. For that purpose, wireless receivers need to collect I/Q data and classify them based on the different waveform and device properties. Supported by a better understanding of spectrum dynamics, wireless systems can then pursue different applications such as dynamic spectrum access (DSA), waveform adaptation, jammer detection, and device authentication. One such application is in Cognitive Radio (CR). CR, through opportunities, utilizes the available spectrum of the existing users to provide a solution to the problem of insufficient spectrum utilization [1]. Hence, one of the key tasks of CR is spectrum sensing [2], which collects information in spectrum scenarios. This information needs to be able to evaluate the possibility of interference with other users, especially the primary users and set the corresponding operating parameters. There are even plans to allocate the unused spectrum to television broadcasting services through CR [3].

Wireless signals can be classified by likelihood-based methods or feature-based analysis on the received I/Q samples. These approaches require prior knowledge of signals and the spectrum environments. On the other hand, machine learning can be applied as a data-driven approach to classifying signals. Conventional machine learning algorithms such as Support Vector Machine (SVM) fall short from extracting the intrinsic properties of wireless signals and reliably classifying them. Deep learning has emerged as a powerful way to classify wireless

signals by extracting and operating on underlying representations hidden in spectrum data. A prominent example of wireless signal classification that has gained recent popularity is modulation classification, namely classifying the received signals to the type of modulation used by the transmitter.

This deep learning has traditionally been solely performed on servers and high-performance machines. However, advances in chip technology have given us miniature libraries such as TensorFlow Lite that fit in our pockets and mobile processors have vastly increased in capability narrowing the large gap between the simple microprocessors embedded in such things and their more complex counterparts in personal computers.

2.2. Aims

This project aims to focus on the practical application of deep learning-based RF modulation classifiers in the field. The typical practice for modulation classification with deep learning is to use either simulated or over-the-air captured signals for testing and training purposes, while the underlying computation is performed off-line with central processing unit (CPU) or graphics processing unit (GPU) resources without answering the needs of embedded (on-device) computation. However, for some real-time applications that need to make a quick decision in microseconds timeframe, the latency to move the data to the cloud for deep learning is not feasible. To that end, we investigated the feasibility of performing classification on the embedded device itself. Additionally, we will investigate the performance of a model trained on a synthetic dataset on real-life RF samples.

2. Overview of the ADALM-PLUTO and Raspberry Pi

2.1. ADALM-PLUTO

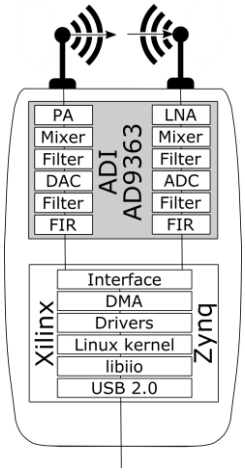

Figure 2.1.1: Overview of the ADALM-PLUTO

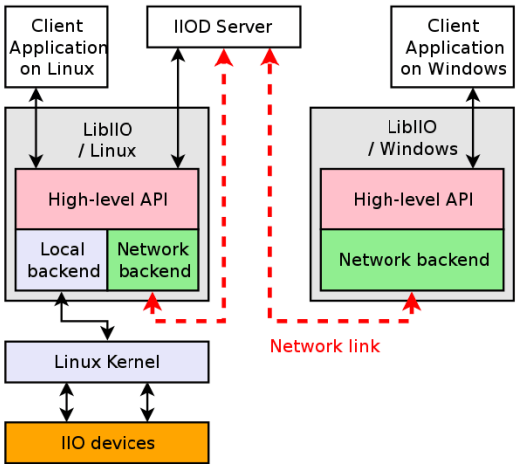

Figure 2.1.2: Software stack of the ADALM-PLUTO

The ADALM-PLUTO is equipped with an Analog Devices AD9361 RF transceiver and Xilinx Zynq XC7Z010-1CLG225C FPGA with ARM Cortex Processor (Single-core ARM® Cortex™-A9 MPCore™). The FPGA has 28K System Logic Cells, 2.1Mb Memory, 80DSP Slices. The MIMO-capable RF transceiver covers 325MHz to 3.8GHz with a tuneable instantaneous bandwidth of between 200kHz to 20MHz [4]. The ADALM-PLUTO was chosen as it ran embedded Linux, allowing us to build custom applications for the ADALM-PLUTO.

An overview of the ADALM-PLUTO architecture is shown in Figure 2.1.1. The software stack of the ADALM-PLUTO is shown in Figure 2.1.2. Our application will be interfacing with the IIOD server over the network link.

2.2. Raspberry Pi

While it is possible to run classification directly on the ADALM-PLUTO, due to time restrictions and technical restrictions[1], I chose to implement our classification model on the Raspberry Pi itself.

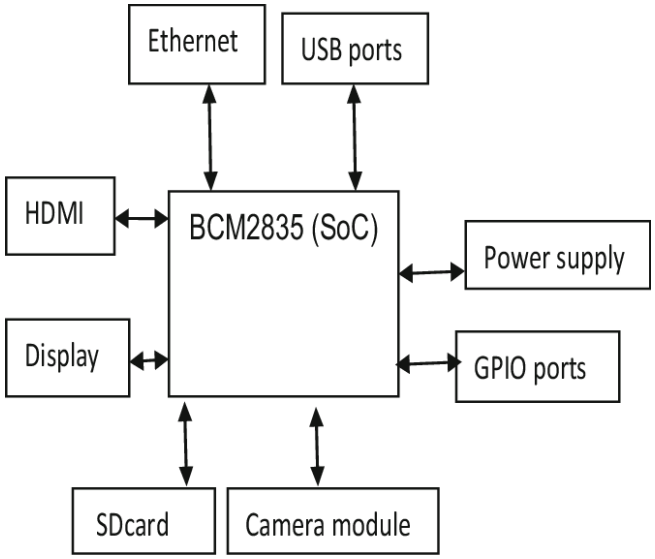

Figure 2.2.1: Overview of the Raspberry Pi

The Raspberry Pi 2 was chosen as it had a 900MHz quad-core ARM Cortex-A7 CPU, 1 GB of RAM and 4 USB ports with 100 Base Ethernet and 40 GPIO pins (refer to Figure 2.2.1) [5]. The onboard peripherals will allow us to interact with other embedded devices or share data with over USB or Ethernet. Additionally, the integrated GPIO pins allow us to interact with embedded devices like microcontrollers or build a human user interface using a 16x2 LCD for instance.

[1] More details about my attempts to run the classification on the ADALM-PLUTO can be found in **APPENDIX I**

3. Implementation

3.1. Previous work

In [6], they explored using CNN to extract features from I/Q time-series. O'Shea et al. [6]–[8] propose several variations of Convolutional Neural Networks for the task of modulation classification. 1D convolutional layers have proven to be helpful for time-series analysis in tasks such as human activity recognition and financial time-series forecasting, so it makes sense that 1D convolutional layers would also help extract features from the I/Q time-series. The different architectures are elaborated on in the following sections.

3.1.1. Basic CNN

A basic CNN with 2 convolutional layers and 2 dense layers is used for signal classification. These convolutional layers can be used to average the noise out as a way to optimize the signal-to-noise ratio (SNR)[2], cleaning the signal [9]. The choice of filter sizes (1x3, 2x3) and filter numbers in each layer were determined through trial and error [9].

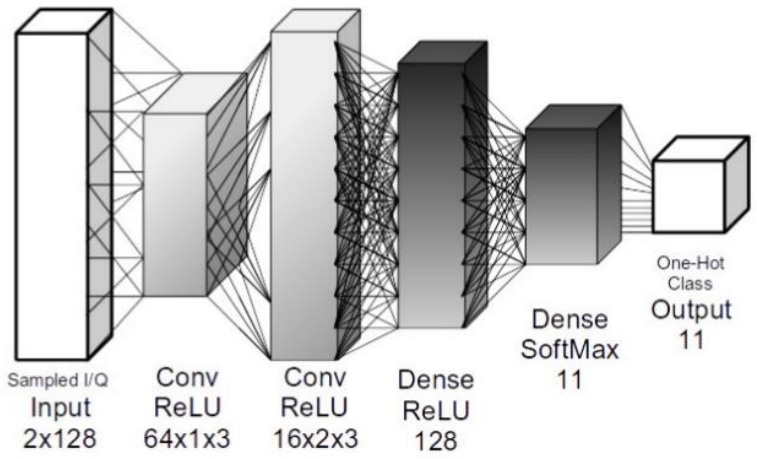

Figure 3.1.1: Architecture of basic CNN with 2 convolutional layers and 2 dense layers

3.1.2. Inception-based CNN

The inception architecture serves as a way to increase network depth and ability to generalize to features of varying scales while still managing complexity [8]. Here, 1x1, 1x3 and 1x8 filter sizes are used. This network consists of repeated inception modules, allowing processing of

[2] SNR is defined as the ratio of signal power to the noise power, often expressed in decibels. A ratio higher than 1:1 (greater than 0 dB) indicates more signal than noise.

spatial information at various scales, which are then aggregated when the filter outputs are concatenated together [6].

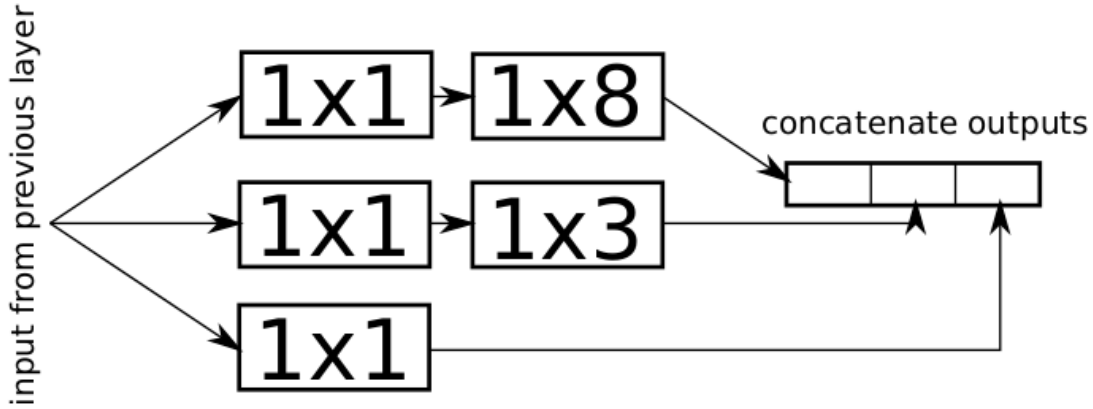

Figure 3.1.2: Architecture for inception-based CNN

3.1.3. Resnet

The Resnet model is effective in overcoming the Vanishing Gradient problem, which is when the gradients of the loss function approaches zero as more epochs are trained, making it harder to train the network [10]. This is done by using a skip-connection as shown in Figure 3.1.3. The skip-connection serves as a route for earlier features to operate at multiple scales and depths throughout the neural network [11]. This tackles the vanishing gradient problem and makes the training of much deeper networks possible [11].

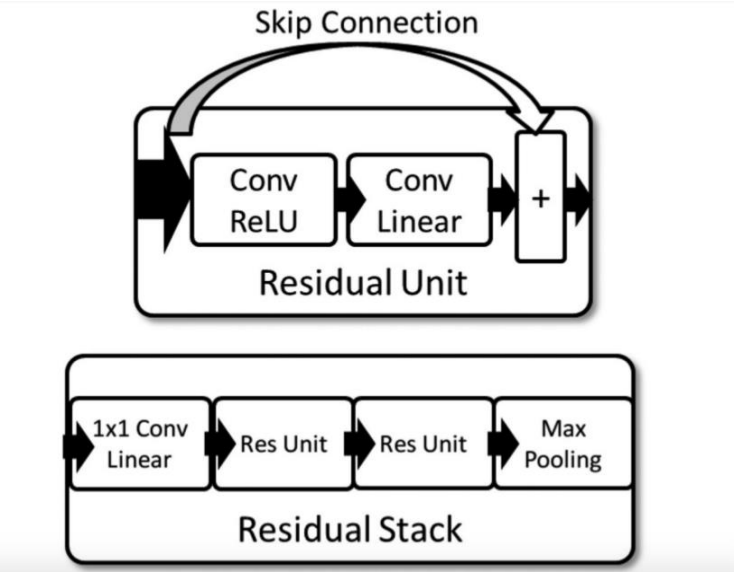

Figure 3.1.3: Skip-connection feature for Resnet

3.1.4. CLDNN

CLDNNs make use of convolutional layers followed by Long Short-Term Memory (LSTM) cells. LSTMs consist of several gates that control how long history is maintained [8]. It has connections that bypass layers so as to provide a longer time context for the features [8]. This is especially useful in this project, which makes use of time-series I/Q data. In CLDNN architectures, the long input sequences are turned into much shorter representations of high-level features through feature extraction and dimensionality reduction process, which is done by the convolutional layers with pooling. Sequentially, the shorter sequences become the input for subsequent LSTM layers to learn long term temporal coherence of different modulation types.

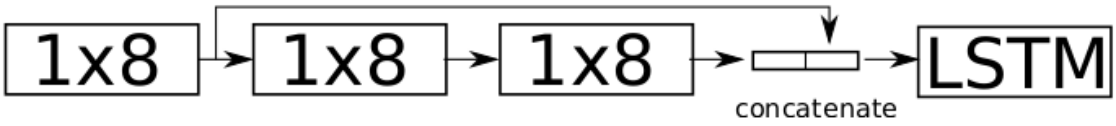

Figure 3.1.4: Architecture of CLDNN

3.1.5. Results

Models		**Classification accuracy (%) at:**			
Architecture	**Input**	**Overall**	**High[3] SNR**	**Med[4] SNR**	**Low[5] SNR**
Basic CNN	I/Q	55.78	81.24	60.70	13.14
	A/P	54.92	86.67	53.90	12.92
Inception	I/Q	44.52	64.61	46.81	13.35
ResNet	I/Q	55.84	82.36	59.61	13.52
	A/P	54.09	86.52	51.91	12.96
CLDNN	I/Q	56.73	81.70	62.23	13.86
	A/P	58.98	93.60	58.67	11.92
Simple LSTM	A/P	55.55	87.84	54.46	12.90
Constellation (AlexNet)	32x32x1	-	90.93	51.69	-
	32x32x3	-	89.37	50.67	-
	48x48x1	-	91.82	51.83	-
	48x48x3	-	92.14	51.69	-

Figure 3.1.5: Table showing classification accuracies of different models at three SNR ranges using the RadioML dataset [6]. *The A/P CLDNN (highlighted in red) performed the best in overall accuracy.*

[3] SNR: 6 dB to 18 dB
[4] SNR: -10 dB to 5 dB
[5] SNR: -20 dB to -11 dB

[6] found that

1. The CLDNN model which combined both convolutional and LSTM layers was the best performing on the cleaner radioML dataset (2×128 time-series) (see Figure 3.1.5 for the classification accuracies of different models on the RadioML dataset). It was trained on amplitude-phase time-series and outperformed the rest of the models significantly at high SNR.
2. Across all models, training with amplitude-phase data yielded significantly higher (5%) accuracy at high SNRs.

Hence, I will be implementing CLDNN trained with the synthetic RadioML amplitude-phase (AP) data as the input to classify different modulation schemes on the Raspberry Pi with the ADALM-PLUTO since our test environment is relatively noise-free.

3.2. System setup

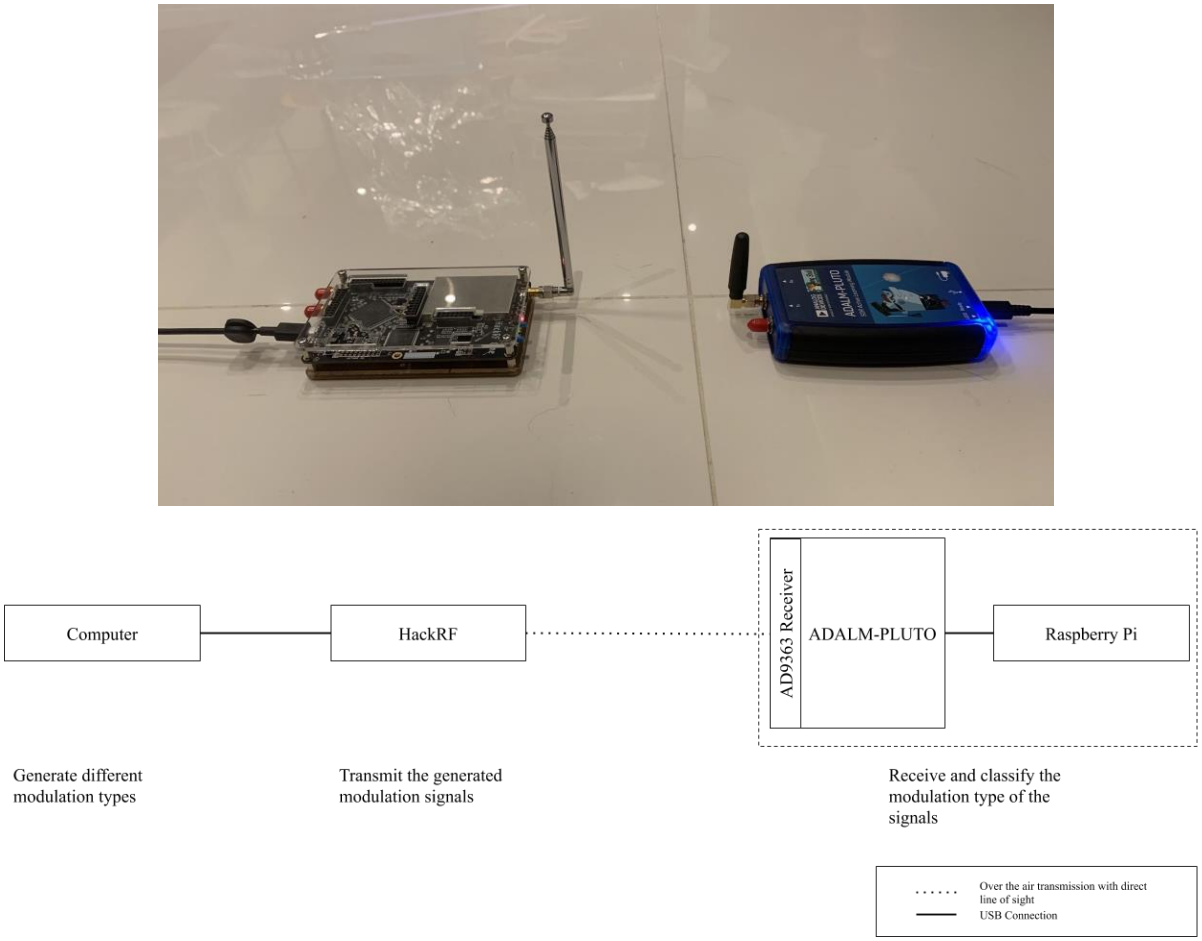

Figure 3.2.1: Validation set-up

The system setup is shown in Figure 3.2.1. There are two major components, a transmitter and a receiver.

- There is one HackRF One that is controlled by a computer as an RF front end. This HackRF One either waits or transmits narrowband signals at 1.41 GHz frequency. Each transmitted signal is modulated with one of six different modulation types, namely
    - Quadrature amplitude modulation 16 (QAM16)
    - Quadrature amplitude modulation 64 (QAM64)
    - Gaussian frequency-shift keying (GFSK)
    - Binary Phase Shift Keying (BPSK)
    - Quadrature Phase Shift Keying (QPSK)
    - Continuous phase modulation (CPFSK)
    - 4 Level Pulse Amplitude Modulation (PAM4)
    - Phase-shift keying 8 (8PSK)
- As a receiver, ADALM-PLUTO collects 128 samples of the signal into its buffer and passes it on to the Raspberry Pi. The Raspberry Pi (the classifier) runs a signal classifier by taking the received signals (I/Q samples) as input data and determines whether the received signal or it is a signal transmitted with one of the eight modulation types (labels 1-8) dynamically in real-time by running a deep learning classifier.

3.3. Methodology

To transmit a clean signal of a particular modulation scheme, the In-phase/Quadrature (I/Q) file is needed to transmit the signal. Thus, a MATLAB program (Figure 3.3.1) has been created to produce I/Q data for the eight modulation schemes specified above.

```
O = 15
d = randi([0 O-1], 2000000, 1);
% 16PSK modulation
syms = pskmod(d,16,pi/16);

%32PSK modulation
%syms = pskmod(d,32,pi/32);


scatterplot(syms);
IQ = [];
for i = 1:length(syms)
    x =  real(syms(i));
    y =  imag(syms(i));
    IQ(end+1) = (x);
    IQ(end+1) = single(y);
end



fileID = fopen('IQ16psktransmit.bin','w');
fwrite(fileID,IQ,'single');
fclose(fileID);
```

Figure 3.3.1: MATLAB code to generate I/Q files for 16PSK and 32PSK modulation schemes

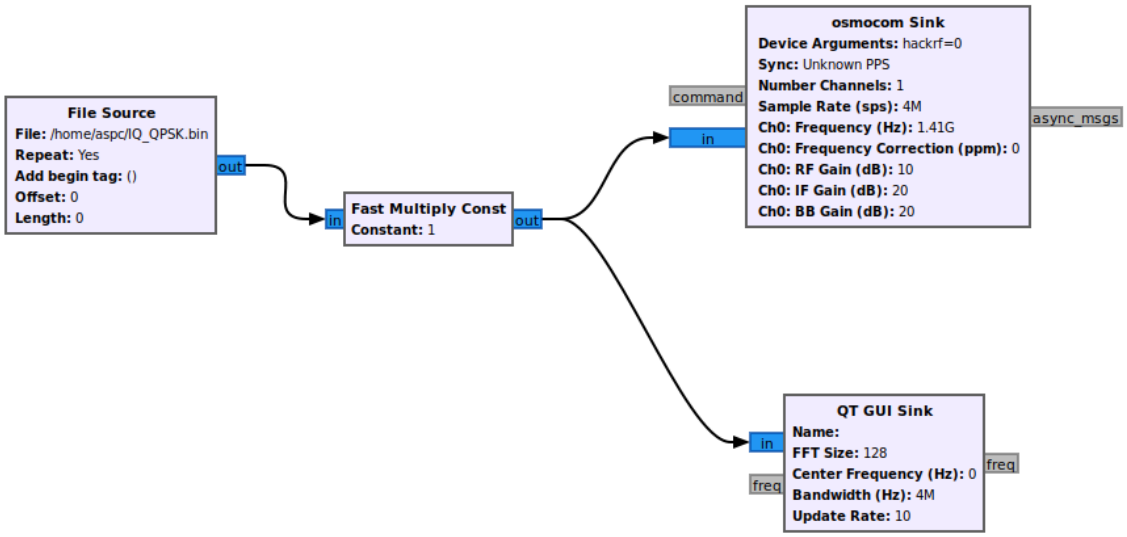

Figure 3.3.2: GNURadio Blocks responsible for transmitting the different modulation signals

The files are then passed into the file source block in GNURadio (Figure 3.3.2), where the signals will be transmitted by the transmitter.

Figure 3.3.3: IMDA Frequency band allocation chart [12]. *Shaded region indicates the spectrum is unallocated*

To prevent interference from other sources, a frequency of 1.41GHz is used for transmitting the signal as 1400 - 1429 MHz was unallocated (Figure 3.3.3). Additionally, the 4 mW transmission power of the HackRF One at 1.41 GHz is unlikely to cause any interference with devices operating in the 1.41 GHz band. To ensure no interference exists at 1.41 GHz, a preliminary scan of the frequency was performed.

The received signals are passed onto a TensorFlow Lite interpreter and run through a deep neural network[6] that returns the signal labels (one of eight modulation types). TensorFlow Lite is a set of tools to help developers run TensorFlow models on mobile, embedded, and IoT devices. It enables on-device machine learning inference with low latency and small binary sizes [13].

[6] The code for this project can be found in [17].

4. RF Classification Performance

We measured that the RF signal classifier implemented on the ADALM-PLUTO achieves high accuracy during our testing.

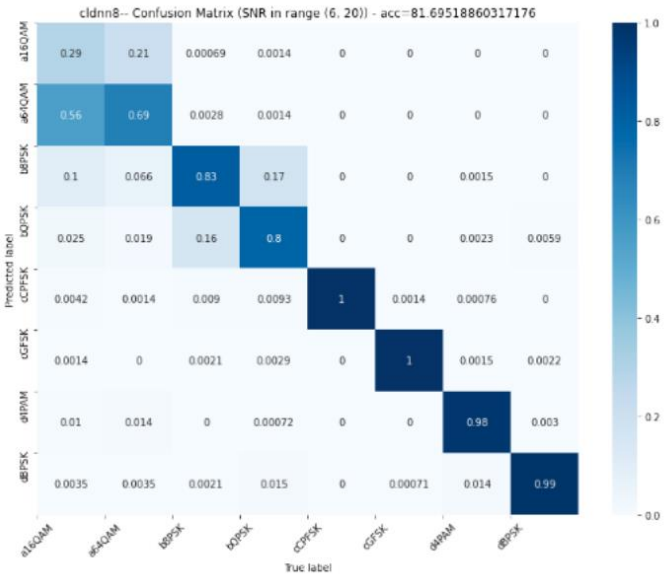

Figure 4.1.1: Confusion matrix of RF signal classifier (CLDNN) when run on the RadioML synthetic dataset

Figure 4.1.2: Confusion matrix of RF signal classifier running on the ADALM-PLUTO. The overall accuracy is 33.75%

The confusion matrix of the RF signal classifier running on the ADALM-PLUTO is shown in Figure 4.1.2 while the confusion matrix of the RF signal classifier running on RadioML synthetic dataset is shown in Figure 4.1.1. When real signals were passed into the model, the model achieved an accuracy of 33.75% accuracy. This low accuracy could be due to the noisier nature of the real signals as compared to the synthetic RadioML dataset that the model was trained on. [14] observed that when the model is trained with high-quality data (high SNR) but tested on lower-quality data (low SNR), the accuracy is low; when the model is trained with low SNR but tested on high SNR, the accuracy is high. This suggested that in general, adding noise to the training data makes the network more robust, that is, less affected by the presence

of noise in the test set. Additionally, [6] found that the models that they tried adapted poorly when tested on a different dataset than it was trained on.

5. Future work and conclusion

5.1. Conclusion

In conclusion, the model's impressive performance on the synthetic RadioML dataset did not translate to good performance when raw samples from the ADALM-PLUTO were classified. While the model used demonstrated impressive performance on the synthetic RadioML dataset, this did not translate to good performance when raw samples from the ADALM-PLUTO were classified. This could be because, in the real world, it is impossible to control how much noise is being detected by the receiver. Uncontrollable environmental factors such as temperature and position of the receiver can affect the SNR of the signal received, explaining the bad performance of the model. In contrast, the RadioML dataset that the model was trained on was much cleaner as compared to the real-life samples collected.

Matthew has also shown that retraining on real-life data increases the accuracy of the classifier. This shows that synthetic data and real data is very different and that the model needs to be retrained on real data.

Additionally, we have shown it is possible to run the models on embedded devices with limited resources by running the model on the Raspberry Pi.

5.2. Future Work

Given the short one-month internship period, further investigation into whether if there is a performance difference between a model trained on a noisy synthetic dataset and noisy real-life dataset. Additionally, to investigate the effect training the model on real-life data, the model can be retrained on data collected in the real world. For example, the model can be trained using data from varying the distances between the transmitter and the receiver, as well as to add a jamming unit to introduce more noise to the signal. This further stimulates the real conditions that this RF signal classification system will be subjected to.

Next, another possible direction would be to try to integrate the model directly into the ADALM-PLUTO. Since the ADALM-PLUTO has the sufficient processing resources and memory to hold our program and built-in General-purpose input/output (GPIO) pins, the

ADALM-PLUTO can perform classification without a host computer while interacting with other embedded devices.

Finally, the model's performance when trained with real data and evaluated on synthetic data can be investigated. This might give some insight into whether synthetic and real data significantly affects the model's performance. Should there be no significant difference between the two different methods of data collection, the cheaper to obtain synthetic data can be used to train future ML models.

6. APPENDIX I

6.1. Attempts to run inference on the ADALM-PLUTO

Initially, I had planned to run the model directly on the ADALM-PLUTO since it had enough processing power and memory to run the model directly. When packaged the Python program took up only 16 Megabytes of memory. This fits within the limit of 32 Megabytes of the ADALM-PLUTO's flash memory.

However, when implementing this, I encountered errors that prevented me from running the Python code on the ADALM-PLUTO. Firstly, when packaging the Python code into a standalone executable using pyinstaller[7], certain libraries such as `pylibiio` did not work in the environment of the ADALM-PLUTO while it worked on the armhf build environment (see Figure 6.1.1).

```
# ./app_py
Traceback (most recent call last):
  File "app.py", line 5, in <module>
  File "<frozen importlib._bootstrap>", line 983, in _find_and_load
  File "<frozen importlib._bootstrap>", line 967, in _find_and_load_unlocked
  File "<frozen importlib._bootstrap>", line 677, in _load_unlocked
  File "/home/pi/.local/lib/python3.7/site-packages/PyInstaller/loader/pyimod03_importers.py", line 493, in exec_module
  File "adi/__init__.py", line 34, in <module>
  File "<frozen importlib._bootstrap>", line 983, in _find_and_load
  File "<frozen importlib._bootstrap>", line 967, in _find_and_load_unlocked
  File "<frozen importlib._bootstrap>", line 677, in _load_unlocked
  File "/home/pi/.local/lib/python3.7/site-packages/PyInstaller/loader/pyimod03_importers.py", line 493, in exec_module
  File "adi/ad936x.py", line 34, in <module>
  File "<frozen importlib._bootstrap>", line 983, in _find_and_load
  File "<frozen importlib._bootstrap>", line 967, in _find_and_load_unlocked
  File "<frozen importlib._bootstrap>", line 677, in _load_unlocked
  File "/home/pi/.local/lib/python3.7/site-packages/PyInstaller/loader/pyimod03_importers.py", line 493, in exec_module
  File "adi/context_manager.py", line 34, in <module>
  File "<frozen importlib._bootstrap>", line 983, in _find_and_load
  File "<frozen importlib._bootstrap>", line 967, in _find_and_load_unlocked
  File "<frozen importlib._bootstrap>", line 677, in _load_unlocked
  File "/home/pi/.local/lib/python3.7/site-packages/PyInstaller/loader/pyimod03_importers.py", line 493, in exec_module
  File "iio.py", line 236, in <module>
  File "ctypes/__init__.py", line 369, in __getattr__
  File "ctypes/__init__.py", line 374, in __getitem__
AttributeError: ./app_py: undefined symbol: iio_get_backends_count
[1459] Failed to execute script app
#
```

Figure 6.1.1: Traceback of the pyinstaller executable using pylibiio

[7] PyInstaller freezes (packages) Python applications into stand-alone executables, under Windows, GNU/Linux, Mac OS X, FreeBSD, Solaris and AIX.

I speculated that this was because the ADALM-PLUTO did not come with the Python bindings installed. Hence, I removed the dependency on the `pyadi-iio` and `pyiio` libraries by interacting with `libiio` using the userspace command-line tools. However, I ran into my second issue which was that the Python TensorFlow Lite interpreter did not run on the ADALM-PLUTO. I tried switching to using Nuitka[8] to compile the Python code into C code. However, this produced the same error[9] as pyinstaller.

6.2. Potential solutions

The solution to the problems above is to write the code interfacing with `libiio` and TensorFlow Lite in C++ and link it against the sysroot provided by Analog Devices. Unfortunately, due to time constraints, I was unable to test this solution.

Additionally, an alternative approach based on [15] is to create an Ubuntu armhf userland and modify the ADALM-PLUTO's firmware. However, since this entails modifying the firmware of the ADALM-PLUTO which risks bricking the device, I did not test this approach.
The approach is as follows:

1. Build the Pluto firmware from source. Instructions can be found in [16].
2. Modify the Buildroot image to mount the USB drive and switch_root as init process. This means replacing the Buildroot / BusyBox init script (in the root of the initramfs image) with a custom one.
3. Create Ubuntu armhf userland using qemu and debootstrap.
4. Install Python 3.9 and libiio, libad9361-iio (also OpenSSH-server, etc.) into userland.
5. Copy userland to ext4 flash drive.
6. Connect flash drive and the WiFi dongle (and/or USB - UART adapter).
7. Boot, SSH in, run the Python script.

[8] Nuitka is a source-to-source compiler which compiles Python code to C source code, applying some compile-time optimizations in the process such as constant folding and propagation, built-in call prediction, type inference, and conditional statement execution.
[9] The error was "Illegal Instruction"

7. References

[1] T. Yucek and H. Arslan, "A survey of spectrum sensing algorithms for cognitive radio applications," *IEEE Communications Surveys & Tutorials*, vol. 11, no. 1, pp. 116–130, 2009, doi: 10.1109/SURV.2009.090109.

[2] O. A. Dobre, "Signal identification for emerging intelligent radios: classical problems and new challenges," *IEEE Instrumentation & Measurement Magazine*, vol. 18, no. 2, pp. 11–18, 2015, doi: 10.1109/MIM.2015.7066677.

[3] N. Venkatesh, "Sharing the Broadcasting Spectrum: digital dividend, white spaces, power line telecommunication (PLT) system," *ITU*. https://www.itu.int/net/ITU-R/information/promotion/e-flash/4/article4.html (accessed Dec. 30, 2020).

[4] amiclaus, "ADALM-PLUTO Detailed Specifications [Analog Devices Wiki]," *Analog Devices*. Nov. 2018, [Online]. Available: https://wiki.analog.com/university/tools/pluto/devs/specs.

[5] The Raspberry Pi Foundation, "Buy a Raspberry Pi 2 Model B," *Raspberry Pi*. https://www.raspberrypi.org/products/raspberry-pi-2-model-b/.

[6] K. W. Yoke, "Internship Project Report and Documentation: RF Signal Classification using Deep Learning," Aug. 2020. [Online]. Available: https://github.com/kwyoke/RF_modulation_classification/blob/master/Report_RFmodclass.pdf.

[7] OREILLY, "10 Modulation Classifier Design for Military Applications - Automatic Modulation Classification: Principles, Algorithms and Applications [Book]," *OREILLY*. https://www.oreilly.com/library/view/automatic-modulation-classification/9781118906514/c10.xhtml (accessed Dec. 30, 2020).

[8] N. E. West and T. O'Shea, "Deep architectures for modulation recognition," in *2017 IEEE International Symposium on Dynamic Spectrum Access Networks (DySPAN)*, Mar. 2017, pp. 1–6, doi: 10.1109/DySPAN.2017.7920754.

[9] T. J. O'Shea, J. Corgan, and T. C. Clancy, "Convolutional Radio Modulation Recognition Networks," in *Engineering Applications of Neural Networks*, vol. 629, C. Jayne and L. Iliadis, Eds. Cham: Springer International Publishing, 2016, pp. 213–226.

[10] C.-F. Wang, "The Vanishing Gradient Problem," *Medium*, Jan. 2019. https://towardsdatascience.com/the-vanishing-gradient-problem-69bf08b15484.

[11] L. Kerbs, "Deep Learning for Radio Waves," *Medium*. Sep. 2019, [Online]. Available: https://medium.com/gsi-technology/deep-learning-for-radio-waves-c240446711d1.

[12] Infocomm Media Development Authority, “Singapore Spectrum Allocation Chart.” Infocomm Media Development Authority, [Online]. Available: https://www.imda.gov.sg/-/media/Imda/Files/Regulation-Licensing-and-Consultations/Frameworks-and-Policies/Spectrum-Management-and-Coordination/SpectrumChart.pdf?la=en.

[13] TensorFlow Contributors, “TensorFlow Lite guide,” *TensorFlow*. [Online]. Available: https://www.tensorflow.org/lite/guide.

[14] T. Jian *et al.*, “Deep Learning for RF Fingerprinting: A Massive Experimental Study,” *IEEE Internet of Things Magazine*, vol. 3, no. 1, pp. 50–57, Mar. 2020, doi: 10.1109/IOTM.0001.1900065.

[15] iracigt, “GNU Radio *on* the PlutoSDR: Proof of Concept - Discussions - Virtual Classroom for ADI University Program - EngineerZone,” *Analog Devices*, Jul. 09, 2018. https://ez.analog.com/adieducation/university-program/f/discussions/98761/gnu-radio-on-the-plutosdr-proof-of-concept (accessed Dec. 30, 2020).

[16] rgetz, “Building the Firmware Image [Analog Devices Wiki],” *Analog Devices Wiki*, Jun. 2020. https://wiki.analog.com/university/tools/pluto/building_the_image.

[17] S. Shang En, “DSTA-Internship-Project: Code for Report,” Dec. 2020, doi: 10.5281/ZENODO.4407303.