Internet of Things Neural Networks: ResNet

Final Report, 06/11/2020

Kartik Kulgod, Nitish Kulshrestha, Zesen (Jason) Zhang

Abstract

Exploiting Neural Networks (NN) for radio signal classification is contingent on its efficient implementation on hardware and for this purpose we implemented a ResNet-33 on an FPGA. Previous work on Modulation Classification focused on implementing simpler NN models like VGG10 on an FPGA, however a complex and deep NN like ResNet-33 hasn't been tackled. We used the RadioML dataset [1] to train and test our modulation classifier and it achieved a peak accuracy of 93.85% for floating point parameters and a peak accuracy of 93.83% for INT8 quantized parameters. By using 1D convolution layers instead of 2D layers, our network has 137K parameters compared to 236K in [2] and 507K in [3]. Our model therefore conforms to efficient size, speed and memory requirements. In the future, we plan on synthesising the generated verilog files into a single executable unit to run on an FPGA.

Introduction

Modulation refers to the manner in which discrete information bits are transmitted as continuous waveforms. There are several methods of modulation. One example is QPSK or quadrature phase shift keying. In this method, 2 bits are coupled to give rise to one symbol/waveform. Since there are 4 ways 2 bits can be coupled (00, 01, 10, 11) and the difference between the waveforms is in the phase, we term this type of modulation as Quadrature Phase Shift Keying or QPSK.

The process of identifying the modulation type from the waveforms received is called modulation classification. In conventional modem design, the modulation type is known at the receiver and the transmitter, and so this is not a problem. However, having varying modulation type is beneficial, because it allows the communication system to adapt to a time varying channel (or the medium of transmission). For example, in a more noisy channel transmitting in a lower modulation scheme is advantageous, since it has a higher margin of error. Whereas in a less noisy channel we can transmit in a higher modulation scheme, because it can be decoded much more cleanly and as a side benefit, we are also transmitting more bits per unit time.

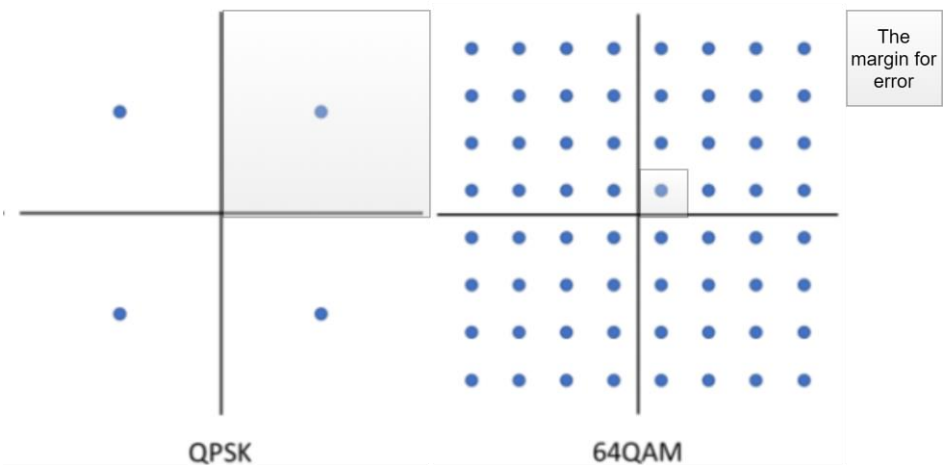

Fig. 1: A symbol in QPSK has an entire quadrant as the margin for error, whereas a symbol in 64QAM has a much smaller margin for error. So QPSK is better for noisy environments, while 64QAM is better for less noisy/noiseless environments. [4]

One approach to implement this is to predetermine the modulation schemes that the system will use and tweak the receiver so that it can support all set modulation schemes. This is however not practical, since it will take up more silicon area (which is costly) and consume more power (which is bad for mobile devices). Another approach would be to implement a Machine Learning (ML) solution which can identify the modulation scheme, which takes less silicon area, and consumes less power. The ML solution we will be implementing is a ResNet-33. ResNet-33 is a class of convolutional neural networks which has skipped connections, i.e not fully connected.

However, for this ML solution to see the light of day, it needs to be deployed on a hardware platform, and not only that but it also needs to be efficient by occupying less memory (less memory equals less silicon), and be fast enough to keep up with ever increasing data rates. So, the hardware platform of choice is an FPGA (Field Programmable Gate Array), since it allows hardware configurability, is fast enough and does not have portability issues like a clunky stack of GPU's.

The pioneers in the work of modulation classification using Neural Networks were O'Shea et al, when they first developed a CNN solution for modulation classification, however the dataset was synthetically generated and had only 11 modulation classes. In a later paper, they used ResNet-33, and a heavily expanded dataset of real radio signals of 24 modulation classes. This paper will be the foundation of our project. S. Tridgell et al, went further to implement neural networks on an FPGA. This paper will be the second foundation of our project. Using the above papers as a reference we will aim to implement a ResNet-33 on a FPGA for the purpose of modulation classification.

Our project has made the following contributions:

- Write the ResNet-33 model in Pytorch
- Quantize the model parameters using Brevitas
- Use a TWN generator to generate C/System Verilog files for the different layers in the model
- Write a testbench to simulate the model on an FPGA

Approach and Results

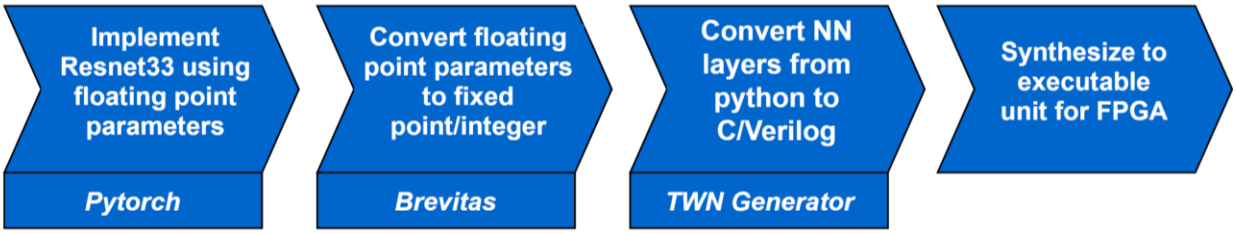

Fig.2: Workflow for the project

Floating Point Implementation

The first step of the project involved building the ResNet-33 model in Pytorch which uses floating point numbers.

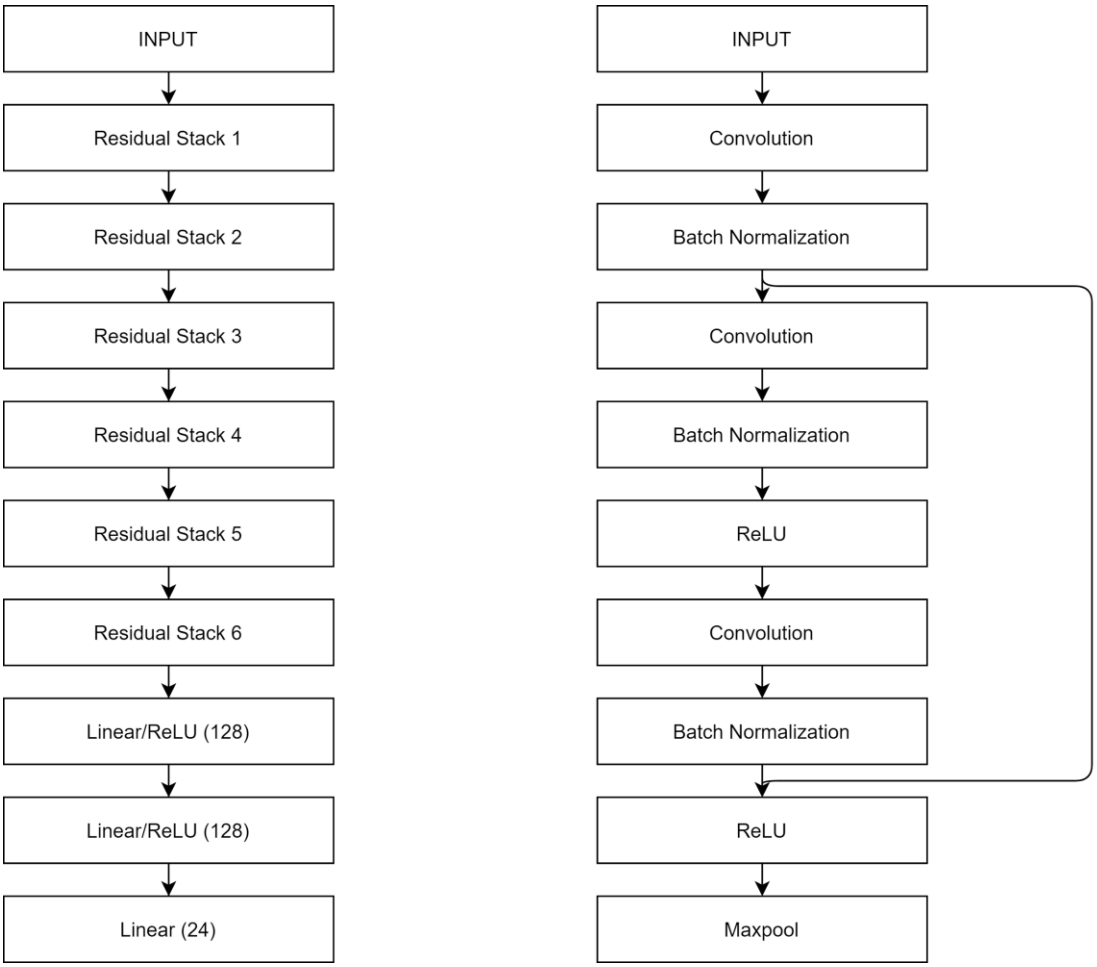

Fig. 3: (Left) ResNet-33 Architecture, (Right) Residual Stack architecture.

The details of the convolution layers are present in Appendix 1

Our model has significant differences from the ones used in [2] and [3]. For one, we used a 1D convolution layer instead of a 2D convolution layer. This resulted in a reduction in the number of parameters (see Appendix 1). We also did not use the SELU activation function used in [2] and instead opted for ReLU activation. [2] says that SELU works better than ReLU, and that is possibly why we have a slightly lower accuracy.

For our dataset we used the RadioML 2018.10A dataset [1], which consists of real radio signals of 24 modulation types collected over a SNR range from -20dB to +30 dB. We split our data in 80:20 for training and testing. Our batch size was 48 data points. We used Stochastic Gradient Descent as our learning algorithm, and the learning rate was 0.001. We ran our model for 30 epochs, with each epoch taking ~25 minutes. The training loss and training accuracy as the training proceeds is shown below.

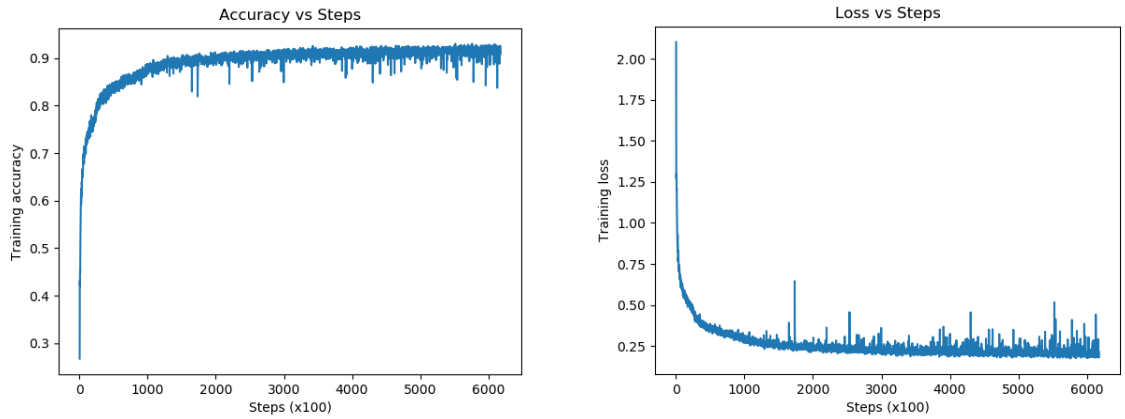

Fig.4: (Left) Training Accuracy, (Right) Training Loss

The peak accuracy we achieved was 93.85%. A variation of the accuracy varies with SNR is shown below.

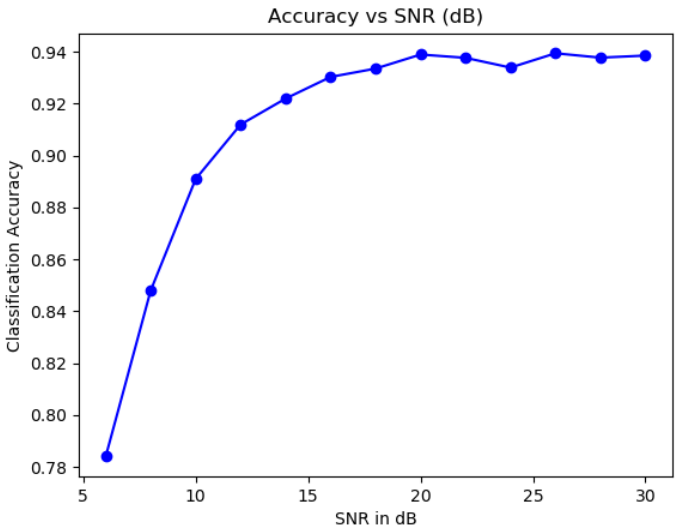

Fig. 5: Accuracy as it varies with SNR

As you would notice, we did not test our accuracy on negative SNR values since negative SNR values aren't encountered in real world applications, and negative SNR values by definition indicate more noise than signal.

Fixed Point Implementation

Having implemented the model in floating point numbers we quantize the network parameters. Quantize refers to the process where we convert from a floating point to a fixed point. Real numbers have a decimal point that 'floats', i.e the decimal point can be inserted anywhere between a series of digits. Whereas fixed point numbers have a decimal point whose position is fixed, say before the last 2 numbers. Sometimes, if the decimal point is the after the last digit, it is equal to an integer.

Now, the obvious issue with this is that quantization will reduce the accuracy of our model. However, it comes with a significant advantage. Most computers and mobiles have a separate Floating Point Unit for performing floating point arithmetic. So, floating point calculations take longer than integer arithmetic. This reasoning will allow us to process more radio signals than if floating point numbers were used.

For our project we will be using an INT8 or a 8 bit integer representation. This process of quantization is done using Brevitas [5], a Xilinx library built on Pytorch. This makes our task a lot easier since we already have built our floating point model in Brevitas. The training method for our quantization was changed, where we used a variable learning rate. For the first 3 epochs, the learning rate was 0.01 which decreased to 0.001 after that.

The training loss and training accuracy for the process is shown below.

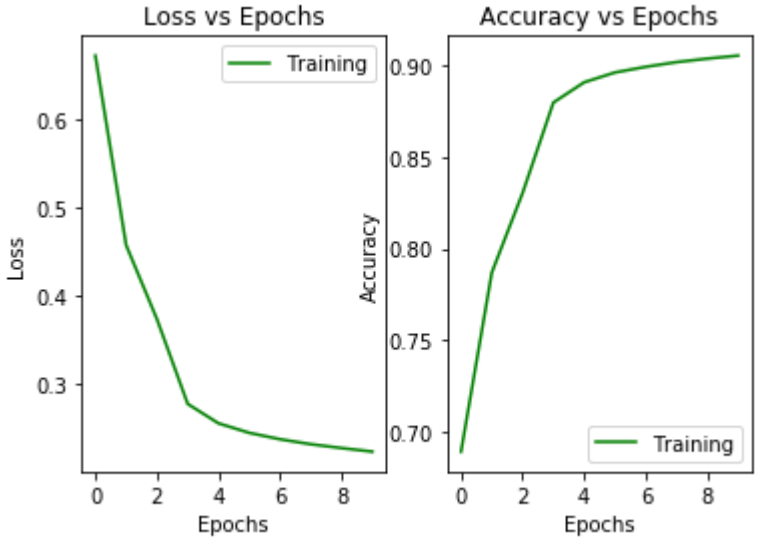

Fig. 6: (Left) Training loss for quantized model, (Right) Training accuracy for quantized model

The peak accuracy we obtained was 93.83%, which is just 0.02% short off the floating point accuracy, which confirmed our belief that the accuracy won't be affected significantly by quantization. The accuracy as it varies with SNR is shown below.

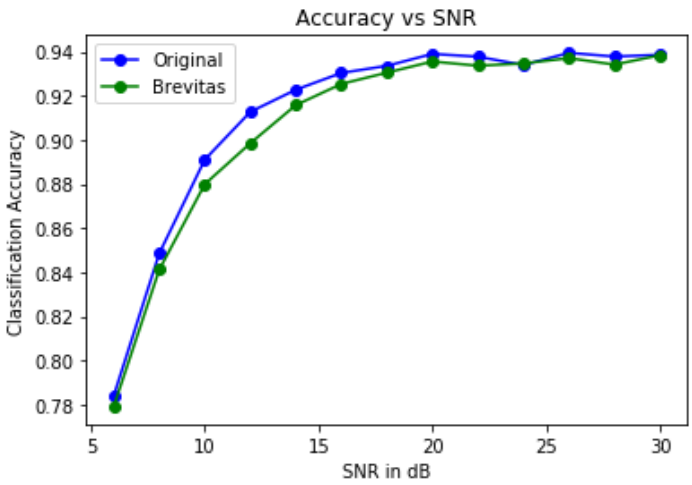

Fig. 7: SNR as it varies with accuracy for floating point and quantized model

At this point we can compare the results of our work with those of our predecessors.

Generating Hardware Files

Since a Python file isn't something that a hardware platform like an FPGA can understand, it needs to be converted to a hardware description language (HDL) file such as System Verilog, or C which is what we will be using. The final design is sometimes called the register transfer level (RTL) design.

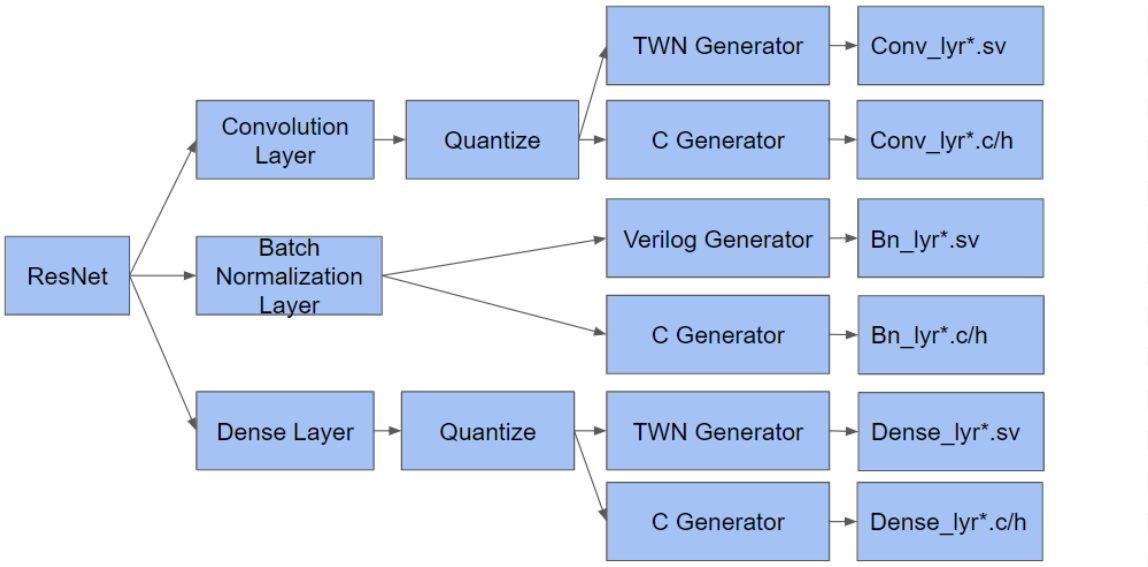

Fig. 8: C and RTL design generation process

The entire process of generation looks as shown in fig. 8. We know that the ResNet-33 model consists of 3 subclasses of layers, namely: convolution, batch normalization (BN) and dense/linear.

The convolution layer consists of a kernel with which a windowed sample of data is multiplied with. That kernel needs to be quantized.

The batch normalization layer normalizes the data to have zero mean and unit variance. From the formula we can see it does not have any dependencies apart from the data itself, which is already quantized. So this layer does not have any quantization involved and this is reflected in the flowchart as well.

The dense layer is simply put a matrix multiplication and so the matrix elements need to be quantized.

Once this is done, we use a C generator and TWN generator, 2 tools which are available at [6], to give us the necessary C, C header files and system verilog (*.sv) files.

```
#define CONV1_IN 6
#define CONV1_OUT 64
short * conv1( const short in[CONV1_IN], short out[CONV1_OUT] );
```

(a)

```
#include "conv1.h"
short * conv1( const short in[CONV1_IN], short out[CONV1_OUT] ) {
short in_6 = in[0] + in[2];
short in_7 = in[4];
short in_8 = in_6 + in_7;
short in_9 = in[0] + in[2];
short in_10 = in[4];
short in_11 = in_9 + in_10;
short in_12 = in[0] - in[4];
```

(b)

```
reg [3:0] vld_reg = 0;
wire [3:0] resets;
reg [2:0] rst_reg;
assign resets = { rst_reg, reset };
assign vld_out = vld_reg[3];
always @( posedge clock ) begin
vld_reg <= { vld_reg[2:0], vld_in };
rst_reg <= resets[2:0];
end
reg [15:0] tree_0;
always @( posedge clock ) begin
tree_0 <= ( $signed( in[0] ) ) + ( $signed( in[2] ) ) - ( $signed( 16'h0 ) )
```

(c)

Fig.9: (a) header file, (b) C file, (c) System Verilog file

A snippet of what the code looks like is presented in fig. 9. As you would notice:

- Source (.c) files have the functions performing necessary mathematical computations. As an advantage of quantization ('short' data type), only addition/subtraction is involved which is much swifter to implement than multiplication/division.
- Header (.h) files specify the size of the input taken and the output generated.
- System Verilog files describe how different hardware blocks within the FPGA are connected.

FPGA Simulation

Once the files are provided, we have an equivalent C encoding of the Python NN model. This can be used in hardware description by means of high level synthesis (HLS), which translates logic directly into hardware blocks called intellectual properties (IPs). Xilinx's Vivado HLS tool is utilized for this purpose.

Further, by writing a wrapper or top-level function that calls these layers or 'filters' (as seen by hardware) sequentially like they're supposed to, the whole design can be synthesized as a single IP. This is also desirable since we could want to contain the conceptual NN graph with all its connections in one unit.

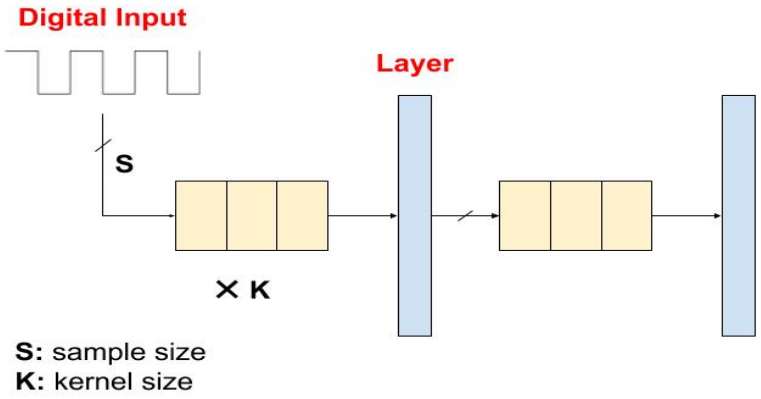

Fig. 10: Top-level logic

Fig. 10 lays out the flow of the top-level logic. It places buffers in between layers because they need to record windows of samples to match the I/O requirement. As we know for convolutions, the window size is restricted by the operational kernel size (K).

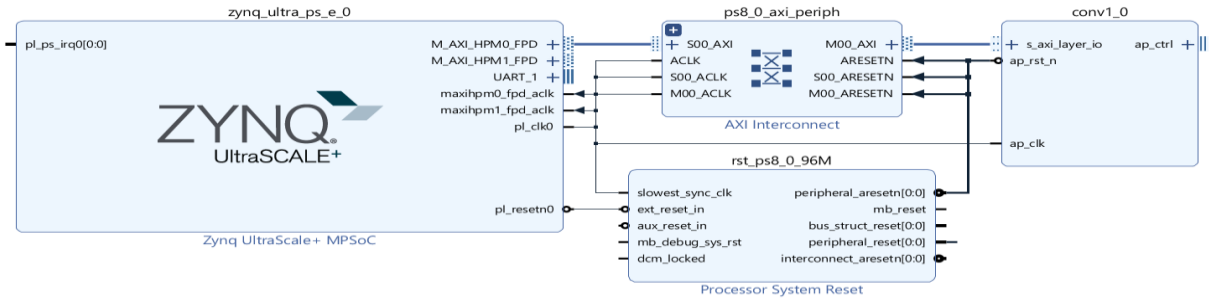

Fig. 11: Interaction of various IPs

Once the IP is generated by Vivado, it lets us interface it to a Xilinx processing chip (which would supply the signals for a physical testbench) using what's called an AXI data bus and a system controller. This is shown for a one-layer model in fig. 11. The interested reader can go through online documentation [7] to know more details.

The expected performance of our model is calculated with specifics reported by Vivado in Appendix 2. It processes 17 signal samples/s (2-channel). Considering a signal image is composed of 1024 such I/Q samples, it takes around 60.25s or roughly a minute to process each image.

FPGA Optimization

The design generated by the automated Vivado tool is seldom perfect for a model as complex as ours. Hence we tried some optimizations, explored here. This involves the various synthesis types in hardware: Most of the compute is done using filters (DSPs). Data close to compute is put into registers (FFs) and look up tables (LUTs) which are faster. All other data is put into on-chip/block RAM or off-chip/ultra RAM.

Name	BRAM	DSP	FF	LUT	URAM
Total	19	167	263430	1180412	0
Available	2160	4272	850560	425280	80
Utilization (%)	~0	3	30	277	0

Key

DSP - digital signal processors
FF - flip-flops
LUT - look up table
BRAM - block RAM
URAM - ultra RAM

Table 1: Utilization of various hardware types (top-level)

Table 1. summarizes our observation. As you might guess, in a NN we tend to run the same compute over a large chunk of data, leading to little usage of DSPs but high usage of FFs and LUTs. The latter are especially over-committed to about 2.8 times full capacity. What's also surprising is that no RAM was used. This can be attributed to the fact that by 1) creating several buffers between compute cores (here NN layers) 2) automatically inlining compute functions because of their small bodies, we're making the memory sit very close to compute, thus demanding heavy use of LUTs and FFs only.

First, we replaced the many buffers with a single reusable buffer. This was possible due to the sequential nature of our model, where calculations in one layer are dependent only on the last layer but not the last *N* layers. This reduced the build time from 7 hours to just 3-3.5 hours! Further application of directives such as `array_partition` and `resource block` [8] was rejected by the tool due to the huge synthesis overhead that our model would require, so utilization remained constant and restricted to outside RAMs.

This forced us to take a step back and look at individual layers. Clearly from Table 2, individual IPs are at around 1% (BN) and 10% (Conv, Dense) utilization, which is far from overcommitting. Hence there are advantages to a modular approach. Making the exact tradeoff is purely a design choice, but we hope this is explored in future work.

Layer	Name	BRAM_18K	DSP48E	FF	LUT	URAM
BN	Total	2	9	1013	5557	0
	Available	2160	4272	850560	425280	80
	Utilization (%)	~0	~0	~0	1	0
Conv	Total	4	0	8193	47977	0
	Available	2160	4272	850560	425280	80
	Utilization (%)	~0	0	~0	11	0
Dense	Total	4	0	9278	42371	0
	Available	2160	4272	850560	425280	80
	Utilization (%)	~0	0	1	9	0

Table 2: Utilization of various hardware types (layers)

Milestones and Challenges

Deliverable	**Current Status**	**Person(s) responsible**	**Achieve(d) in**
Torch implementation with baseline accuracy **(MVP)** ● Get existing (TensorFlow) network parameters ● Compare accuracy on common datasets ● Compare accuracy with baseline	Completed	Kartik, Jason	Weeks 3-4
Quantization of NN ● Run Brevitas and identify key changes required ● Port Torch model to it and compare models ● Hyperparameter tuning to maintain accuracy	Completed	Nitish, Kartik	Weeks 5-7
Generation of C & RTL design ● Change data dimension for TWN Generator ● Use the generator to output C and Verilog files ● FPGA simulation & optimization	Completed	Jason, Nitish	Weeks 8-10
Documentation	Completed	All	Week 11

Hardware target test	Future work	N/A	

Table 3: Milestones of the project

When we set out for our project, our minimum viable product (MVP) was a floating point ResNet-33 model in pytorch. However, we successfully completed not just that, but also its quantization and the C & RTL design generation, which are improvements. The problems we encountered while doing this project were from the lack of proper documentation/infrastructure from the sources we referred to. We list the major ones below:

- There was a lack of detail while specifying hyperparameters and architecture. Since ResNet-33 isn't a standard model like ResNet-34 it is difficult to find literature for the model apart from the papers we referred to. This took a significant amount of time in the initial stages of the project.
- The TWN generator lacks proper documentation on how to use it. It has a lot of different functions and generates multiple files. Moreover, the NN input also needs to be adapted especially for this tool. There is no theoretical support for how it works. Overall, the API acted like a black box with unclear internal organization.
- Putting our top-level model on an HLS test-bench was already complex because we had to come up with an original implementation matching the I/O requirements between several layers in logic (equivalent to matching 'shapes' in a Python model, however with restrictions on memory). It also didn't get desirable accuracies: This can be attributed primarily to no documentation of input quantization techniques in [1]-[3]; we wrongly tried typecasting like in previous students' work but the correct approach would have been using the `ap_fixed` [9] library as we learnt in week 11. Trying it out fell outside scope then.
- Since running our model on a hardware platform wasn't logistically possible due to remote work, we didn't perform the hardware target test.

Conclusion

Wireless Communication is a field that is mathematically analytical and intensive when it comes to resolving and offsetting signal distortion caused by the channel. In fact it is a field that is by nature uncertain, probabilistic and nothing can be a hundred percent sure. So, we can only go so far to achieve a desirable accuracy, and by getting one that is so high is certainly something we are proud of. We have shown a homologation of Machine Learning ideas which are smart, intelligent, efficient and practical to be implemented in today's world, which is a step forward to integrating Machine Learning techniques in this ever changing field.

To demonstrate this concept, we set out to do 4 tasks in the project, which we successfully completed to a fair degree of satisfaction. They were:

- Write the ResNet-33 model in Pytorch
- Quantize the model parameters using Brevitas
- Use a TWN generator to generate C/System Verilog files for the different layers in the model
- Simulate the model on an FPGA

Acknowledgement

We would like to acknowledge and thank Professor Ryan Kastner and Alireza Khodamoradi for the opportunity to work on this project and the immense support, feedback and help we received from them throughout.

References

[1] https://github.com/radioML/dataset
[2] O'Shea, T. J., Roy, T., & Clancy, T. C. (2018). Over-the-Air Deep Learning Based Radio Signal Classification. *IEEE Journal of Selected Topics in Signal Processing*, *12*(1), 168–179. http://doi.org/10.1109/jstsp.2018.2797022
[3] S. Tridgell, D. Boland, P.H.W. Leong, R. Kastner, A. Khodamoradi and Siddhartha "Real-time Automatic Modulation Classification using RFSoC"
[4] (n.d.). Constellation Diagrams: The Fault in Our Stars - NuWaves Engineering: Aerospace & Defense Contractors | RF & Microwave Solutions. Retrieved from https://www.nuwaves.com/constellation-diagrams/
[5] Tridgell, S., Kumm, M., Hardieck, M., Boland, D., Moss, D., Zipf, P., & Leong, P. H. W. (2019). Unrolling Ternary Neural Networks. *ACM Transactions on Reconfigurable Technology and Systems*, *12*(4), 1–23. http://doi.org/10.1145/3359983
[6] https://github.com/Xilinx/brevitas
[7] https://github.com/KastnerRG/pp4fpgas/wiki/PYNQ-example
[8] https://www.xilinx.com/html_docs/xilinx2017_4/sdaccel_doc/wqn1504034365505.html
[9] http://kastner.ucsd.edu/wp-content/uploads/2018/03/admin/pp4fpgas11.12.2018.pdf

Appendix

1. Model Architecture

This was done by printing the model summary in python:

```
ResNet-33_new(
  (conv1): Sequential(
    (0): Conv1d(1, 32, kernel_size=(3,), stride=(1,), padding=(1,))
    (1): BatchNorm1d(32, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  )
  (maxpool): MaxPool1d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  (conv2): Sequential(
    (0): Conv1d(32, 32, kernel_size=(3,), stride=(1,), padding=(1,))
    (1): BatchNorm1d(32, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  )
  (conv3): Sequential(
    (0): Conv1d(32, 32, kernel_size=(3,), stride=(1,), padding=(1,))
    (1): BatchNorm1d(32, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  )
  (conv4): Sequential(
    (0): Conv1d(32, 32, kernel_size=(3,), stride=(1,), padding=(1,))
    (1): BatchNorm1d(32, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
```

```
  )
  (conv5): Sequential(
    (0): Conv1d(32, 32, kernel_size=(3,), stride=(1,), padding=(1,))
    (1): BatchNorm1d(32, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  )
  (conv6): Sequential(
    (0): Conv1d(32, 32, kernel_size=(3,), stride=(1,), padding=(1,))
    (1): BatchNorm1d(32, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  )
  (conv7): Sequential(
    (0): Conv1d(32, 32, kernel_size=(3,), stride=(1,), padding=(1,))
    (1): BatchNorm1d(32, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  )
  (conv8): Sequential(
    (0): Conv1d(32, 32, kernel_size=(3,), stride=(1,), padding=(1,))
    (1): BatchNorm1d(32, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  )
  (conv9): Sequential(
    (0): Conv1d(32, 32, kernel_size=(3,), stride=(1,), padding=(1,))
    (1): BatchNorm1d(32, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  )
  (conv10): Sequential(
    (0): Conv1d(32, 32, kernel_size=(3,), stride=(1,), padding=(1,))
    (1): BatchNorm1d(32, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  )
  (conv11): Sequential(
    (0): Conv1d(32, 32, kernel_size=(3,), stride=(1,), padding=(1,))
    (1): BatchNorm1d(32, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  )
  (conv12): Sequential(
    (0): Conv1d(32, 32, kernel_size=(3,), stride=(1,), padding=(1,))
    (1): BatchNorm1d(32, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  )
  (conv13): Sequential(
    (0): Conv1d(32, 32, kernel_size=(3,), stride=(1,), padding=(1,))
    (1): BatchNorm1d(32, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  )
  (conv14): Sequential(
    (0): Conv1d(32, 32, kernel_size=(3,), stride=(1,), padding=(1,))
```

```
    (1): BatchNorm1d(32, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  )
  (conv15): Sequential(
    (0): Conv1d(32, 32, kernel_size=(3,), stride=(1,), padding=(1,))
    (1): BatchNorm1d(32, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  )
  (conv16): Sequential(
    (0): Conv1d(32, 32, kernel_size=(3,), stride=(1,), padding=(1,))
    (1): BatchNorm1d(32, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  )
  (conv17): Sequential(
    (0): Conv1d(32, 32, kernel_size=(3,), stride=(1,), padding=(1,))
    (1): BatchNorm1d(32, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  )
  (conv18): Sequential(
    (0): Conv1d(32, 32, kernel_size=(3,), stride=(1,), padding=(1,))
    (1): BatchNorm1d(32, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  )
  (classifier): Sequential(
    (0): Linear(in_features=512, out_features=128, bias=True)
    (1): ReLU(inplace=True)
    (2): Linear(in_features=128, out_features=128, bias=True)
    (3): ReLU(inplace=True)
    (4): Linear(in_features=128, out_features=10, bias=True)
  )
)
```

2. Throughput Calculation

From report,
Target clock = 8.75 (estimated) + 1.25 (uncertain) = 10 ns
Interval = 5884256 cycles or 0.059 s

Therefore,
Throughput (signal) = 1/0.059 = 16.995 /s
Throughput (image) = Throughput (signal) / Size = 16.995 / 1024 = 0.0166 /s
Time to process 1 image = 1/Throughput (image) = 60.25 s