1 Python

1.1 Keras Model

In the test and training data that was provided, there were 10,000 waveform events, each of which had 100 instances (Fig. 1). The *tensorflow* package was used to build a simple Keras model to predict the mean and spread, sigma, of the waveforms. Originally, it was used to predict the means, sigma, pedestal, and height. But all four parameters used too many resources on the FPGA, so the model was simplified. The sequential keras model has an input layer of 100 neurons, one hidden layer with 64 neurons using a *relu* activation function, and a final layer of 2 neurons, for the predicted parameters. I used the Adam optimizer with a learning rate of 0.001, MSE for the loss, and MAE for the metric. From trial and error, I found that an epoch number of 60 works the best for this data and model. I had 20% of the data used for validation results to get the validation loss (Fig. 2).

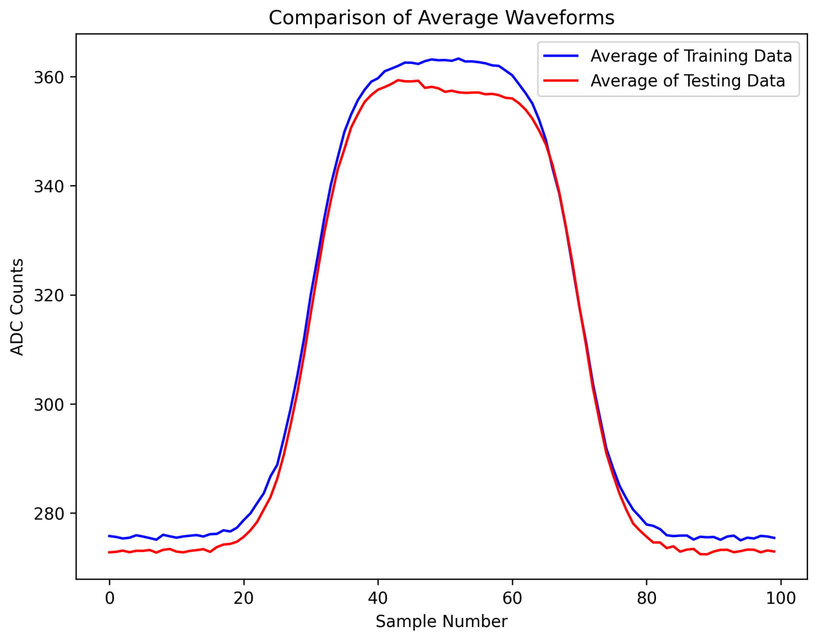


Figure 1: *The averages of both training and testing data plotted to show the differences in the waveforms.*

When predicting the waveform parameters, I calculated the mse, mae, and r^2 metrics to show the precision of the prediction. I paid closer attention to the r^2 metric, as this value represents how close the prediction is to the true value. Where the closer it is to 1, the more accurate and the further, or more negative, the less accurate. This value ranged from 0.71 to 0.99, which is good for such a small model like this. The predictions were plotted against the true values, seen in Fig. 3, and can be seen to have a very wide spread, especially the sigma. I wish more layers and neurons could

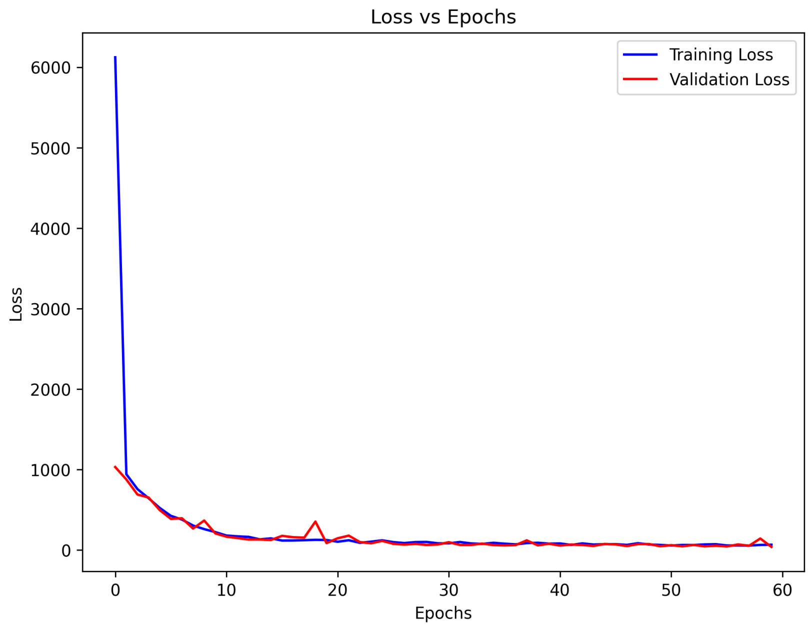


Figure 2: *Plotting the loss and validation loss vs the number of epochs, 60. This number was chosen because the losses would flatten out after a certain number of epochs and begin to over-train the data.*

have been used for building the model, but the FPGA resources could not handle it. The model is then saved as a *.h5* file for later use using the *save* function provided by *tensorflow*.

1.2 HLS4ML Model

Moving on to build the hls4ml model using the *hls4ml* package. This would not work on MAC or Windows, so I had to install a Linux VM and test on there. This is used to convert the keras model into *C++* and make a *build.tcl* file to be synthesized by Vitis-HLS, Sec. 2. Before converting the keras model, the configuration of the hls4ml model is done. This is to quantize the model by specifying the bit precision, re-use factors, and strategy of the hls4ml model. The bit precision is denoted by $< x, y >$, where x is the total number of bits and y is the number of bits representing the signed number above the binary point (Fig. 4). The higher number of bits, the more resources that are used by the FPGA. To minimize the resource usage, I tested each layer's precision through trial and error. Looking at Fig. 5, you can see the layer's weights and biases fit within each of their specified precisions. After, you can use a *ReuseFactor* for specific layers which use too many resources. I used a factor of 32 for the global configuration and 64 for the input and dense layers. This increases the latency of the model build, but will fit your model onto an FPGA which has minimal resources. Another variable that helps with this is the *Strategy* called *Resource*. This

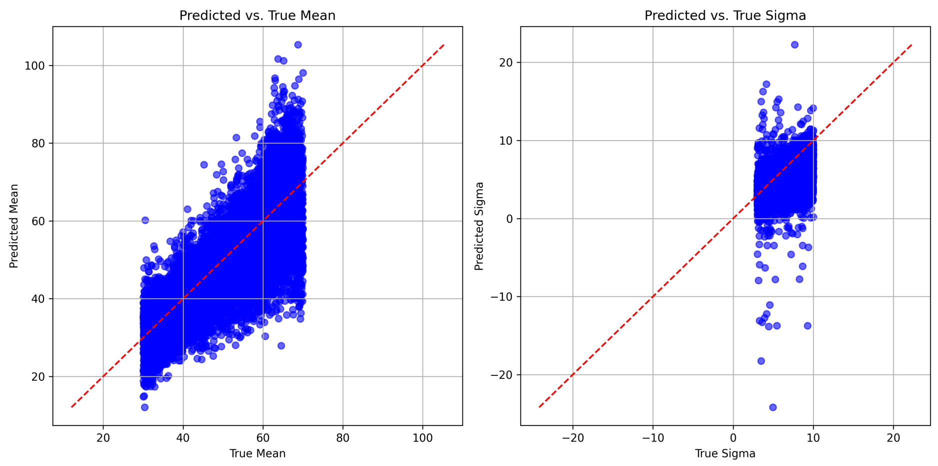


Figure 3: *Mean and Sigma prediction from the Keras model vs the True values from the testing data. This is to compare the differences and look at the spread of the values. Using the red line as a reference, where this is the perfect/best case scenario for predictions.*

does the same and will effectively split the resource usage by half, while increasing the latency.

After the configuration, the model is then compiled and from this, you can predict the waveform parameters. The concept is the same as using the keras model for predictions, where you use *mse*, *mae*, and r^2 to test the precision of the predictions. Plotting the hls4ml predictions against the keras predictions to compare the differences, or to test how close the hls4ml model is to the keras model (Fig. 6).

2 Vitis-HLS

I was not able to get the *build* function in *hls4ml* to work, so I add to use Vitis-HLS on the lab computers which use Windows. Using the *build_prj.tcl* file, which *hls4ml* produced when converting the keras model to a hls4ml model, to produce the C-Sim, RTL Synthesis, and CO-Sim. The RTL synthesis showed the estimated latency to be 12.73 ns, where the target is 15 ns. BRAM_18K, DSP, FF, and LUT usages were 14%, 34%, 10%, and 21%, respectively. This was very good, where the usage was around 100x this when using 4 parameters and not using the *Resource Strategy*. After, the RTL was exported as an IP core for Verilog to use.

3 Verilog

A basic testbench was implemented alongside the IP core from Vitis-HLS to simulate the results of the model. The event number 4,321 was used as the the 1800-bit

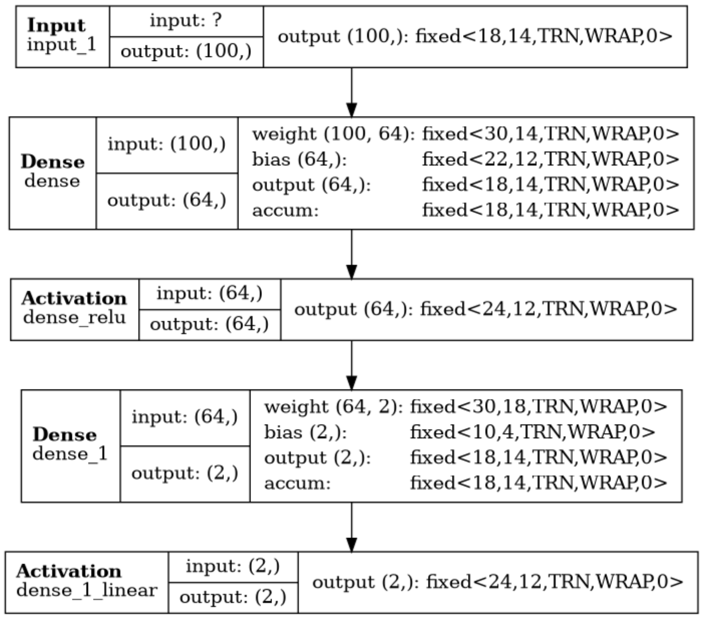

Input
input_1
input: ?
output: (100,)
output (100,): fixed<18,14,TRN,WRAP,0>
Dense
dense
input: (100,)
output: (64,)
weight (100, 64): fixed<30,14,TRN,WRAP,0>
bias (64,): fixed<22,12,TRN,WRAP,0>
output (64,): fixed<18,14,TRN,WRAP,0>
accum: fixed<18,14,TRN,WRAP,0>
Activation
dense_relu
input: (64,)
output: (64,)
output (64,): fixed<24,12,TRN,WRAP,0>
Dense
dense_1
input: (64,)
output: (2,)
weight (64, 2): fixed<30,18,TRN,WRAP,0>
bias (2,): fixed<10,4,TRN,WRAP,0>
output (2,): fixed<18,14,TRN,WRAP,0>
accum: fixed<18,14,TRN,WRAP,0>
Activation
dense_1_linear
input: (2,)
output: (2,)
output (2,): fixed<24,12,TRN,WRAP,0>

Figure 4: *Showing the bit precision of each layer and the number of neurons used.*

input, 18 bits per instance concatenated, and 2, 24-bit outputs were produced, the mean and sigma. The results of the simulation can be seen in Fig. 7. Once done, the predictions $Mean = 033400 = 212,992$ and $Sigma = FFF400 = 16,773,120$ can be converted to fixed-point. Where the bit-precision for this layer was $< 24, 12 >$, which yields a fixed-point format of Q12.12. The final results are the following,

$$\frac{212,992}{4,096} = 52.0 \quad \& \quad \frac{16,773,120 - 16,777,216}{4,096} = -1.0$$

4 Conclusion and Discussion

The simulation predictions align with that of the keras and hls4ml predictions. This means that the IP core is working correctly, in the simulation at least. I was not able to implement the model onto a FPGA directly, but these results show that if this was done correctly, the results would align with the other model predictions. In the future, I want to use all 4 parameters with the models and get it working on an FPGA. I believe that with the *Resource Strategy* the resources usage can decrease enough so that all 4 parameters can be used. The keras model would need a lot of tweaking to get good predictions on limited resources, but I believe this can be done.

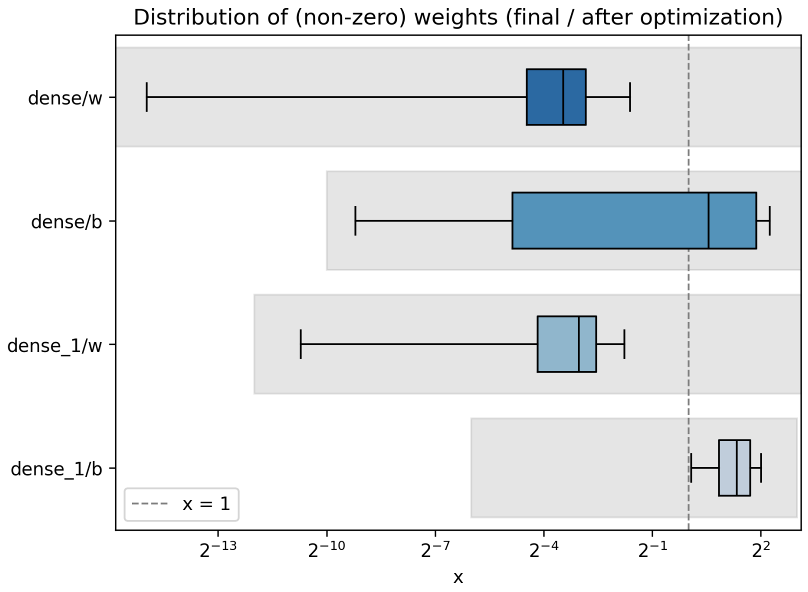


Figure 5: *This represents the numerical model that best quantizes each of the layer's weights and biases. The bit precision can be fixed based on how much precision the layer needs, the cross hairs. The shaded region represents the bit precision you are using and if the hairs fit inside this region, the precision is enough.*

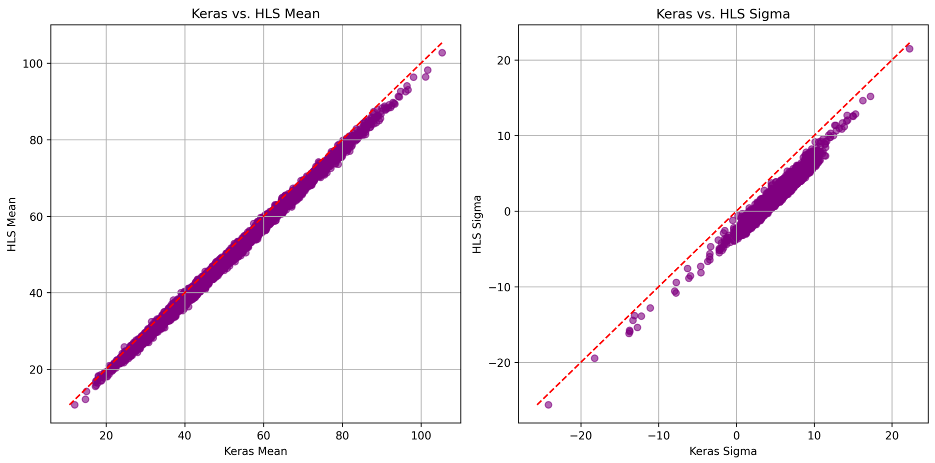
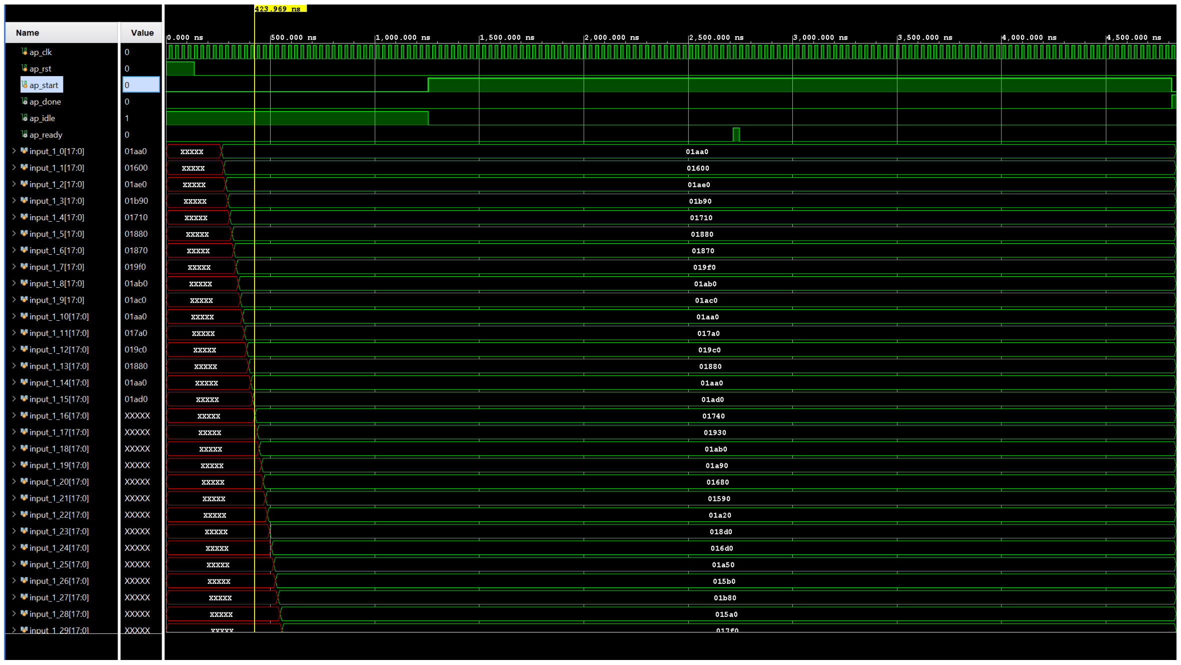
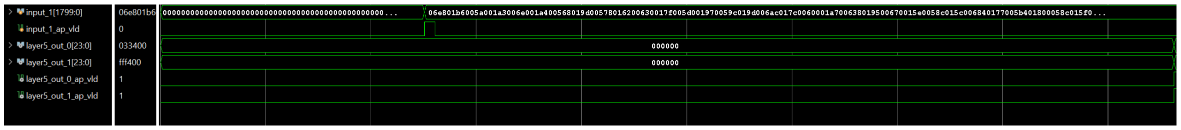


Figure 6: *hls4ml prediction vs the keras predictions to test how accurate the hls4ml model is to the keras model. Same concept shown in Fig. 3.*



Name	Value
ap_clk	0
ap_rst	0
ap_start	0
ap_done	0
ap_idle	1
ap_ready	0
input_1_0[17:0]	01aa0
input_1_1[17:0]	01600
input_1_2[17:0]	01ae0
input_1_3[17:0]	01b90
input_1_4[17:0]	01710
input_1_5[17:0]	01880
input_1_6[17:0]	01870
input_1_7[17:0]	019f0
input_1_8[17:0]	01ab0
input_1_9[17:0]	01ac0
input_1_10[17:0]	01aa0
input_1_11[17:0]	017a0
input_1_12[17:0]	019c0
input_1_13[17:0]	01880
input_1_14[17:0]	01aa0
input_1_15[17:0]	01ad0
input_1_16[17:0]	XXXXX
input_1_17[17:0]	XXXXX
input_1_18[17:0]	XXXXX
input_1_19[17:0]	XXXXX
input_1_20[17:0]	XXXXX
input_1_21[17:0]	XXXXX
input_1_22[17:0]	XXXXX
input_1_23[17:0]	XXXXX
input_1_24[17:0]	XXXXX
input_1_25[17:0]	XXXXX
input_1_26[17:0]	XXXXX
input_1_27[17:0]	XXXXX
input_1_28[17:0]	XXXXX
input_1_29[17:0]	XXXXX

(a)



Name	Value
input_1[1799:0]	06e801b6
input_1_ap_vld	0
layer5_out_0[23:0]	033400
layer5_out_1[23:0]	fff400
layer5_out_0_ap_vld	1
layer5_out_1_ap_vld	1

```
0000000000000000000000000000000000000000000000...
06e801b6005a001a3006e001a400568019d00578016200630017f005d001970059c019d006ac017c0060001a700638019500670015e0058c015c006840177005b401800058c015f0...
000000
000000
```

(b)

Figure 7: *(a) Showing the top of the simulation results, where each instance of the IP module is triggered and the simulation is complete when done=1. (b) Showing the bottom results which displays the 1800-bit input and 2, 24-bit outputs, or predictions.*