Sensor fusion via QNNs

Introduction

Due to their versatility, artificial neural networks (ANNs) have been proven effective in solving complex problems in countless fields. Some of the most known are Computer Vision, Robotics or Natural Language Processing but we wanted to try its application in another not so known field: sensor fusion. This technique takes data from different sensors, combining them into an information with less uncertainty than the available when the sensors data are processed individually.

Its flexibility, easy reprogrammability and design modularity make neural networks a remarkably interesting choice to perform this task. There are many options that allow the use of neural networks to carry out sensorial fusion. For example, cloud computing could be used to process the raw sensor data, but an active internet connection would be necessary. With the aim of avoiding that internet connection, three in situ solutions appear: CPU/GPU computing, FPGA computing or the design of a specific ASIC.

These three solutions are very distanced between them. On one hand there is the CPU/GPU computing, sequential but generalizable and fast implementation. On the other, the design of a specific ASIC, a very particular option with a long design process but that achieves a very notable property, the parallelization. In the middle of these two points is the FPGA computing solution, that combines the best features of the others: fast implementation and parallel computing.

Also, taking advantage of the FPGA option, we decide to optimize the FPGA resources consumption using fixed point representation of the neural network, thus reducing the number of bits to represent the network values (inputs, parameters, outputs) and avoiding the more resource-costly usage of a floating point arithmetic.

This report presents the methodology and the results of a feed-forward quantized neural network implemented over the PYNQ's FPGA that combines the data of 16 gas sensors from an artificial noise to estimate the concentration of two gases on the atmosphere. The dataset comes from the work of Fonollosa et al. 'Reservoir Computing compensates slow response of chemosensor arrays exposed to fast varying gas concentrations in continuous monitoring'; Sensors and Actuators B, 2015 available on UCI Machine Learning Repository (https://archive.ics.uci.edu/ml/datasets/gas+sensor+array+under+dynamic+gas+mixtures).

To achieve the PYNQ's neural network implementation, we divided the project into two parts. The first one is the "virtualization part", where we have used PyTorch to design, train and test the neural model that estimates the gas concentration using the 16 chemosensor outputs. In the second part, the model has been translated into C/C++ in order to reach its HLS implementation. Then the Vivado HLS generated RTL design has been imported from Vivado IP Integrator letting us to generate a bitstream that has been uploaded to the PYNQ. Finally, a specific Python driver to manage the IP has been written, making its management better for the user.

Virtualization

The data

The Fonollosa et al. dataset is a time series that relates the readings of the 16 sensors with the concentrations of the gases over the time, as it is shown in the Figure 1.

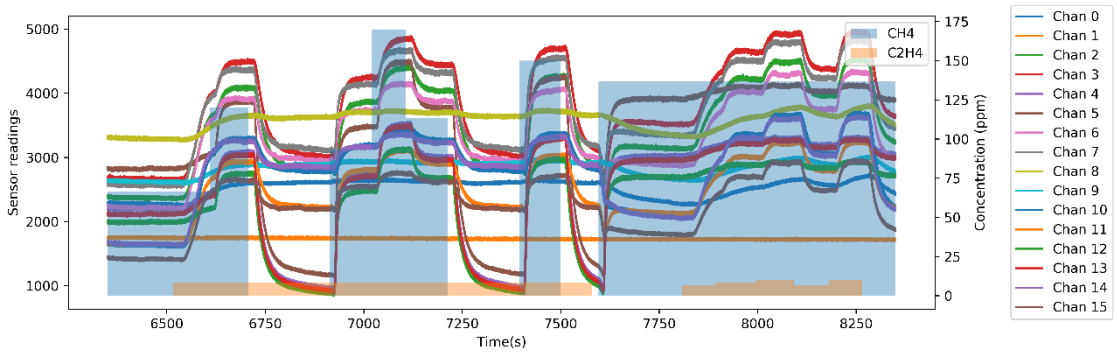

Figure 1: Raw data from the dataset.

The data shown in the figure are not suitable to an efficient network training, mainly due to two issues:

- It is not normalized. Ideally, the data to train the network have to be limited between 0 and 1, giving the same importance to all the inputs and outputs. Additionally, normalization is very important for this problem, because it ensures that all the data are in the same numerical range, letting us to establish the integer part of the fixed-point representation.
- It is not bijective. One input (a 16 sensor values vector) must be related to one output (the two gas concentration levels); however, the dynamic behavior of the gas sensors requires a specific processing to remove the output uncertainity due to the sensors output stabilization time.

In order to solve the first problem, we use the following expression to make all the inputs and output belong the [0,1] range.

$$\bar{x} = \frac{x - x_{min}}{x_{max} - x_{min}}$$

Where x is any value in the magnitude range and x_{max} and x_{min} its maximum and minimum values respectively.

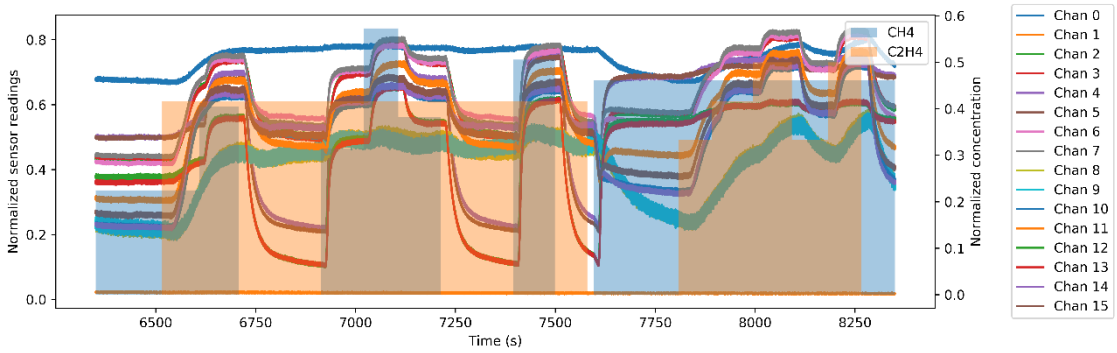

Figure 2: Normalized data

Solving the second problem is more complex. We want to recover the static behavior of the chemosensors, given by the asymptotic value they reach when the gas concentration remains constant. For this, we are using an average window for each gas constant concentration.

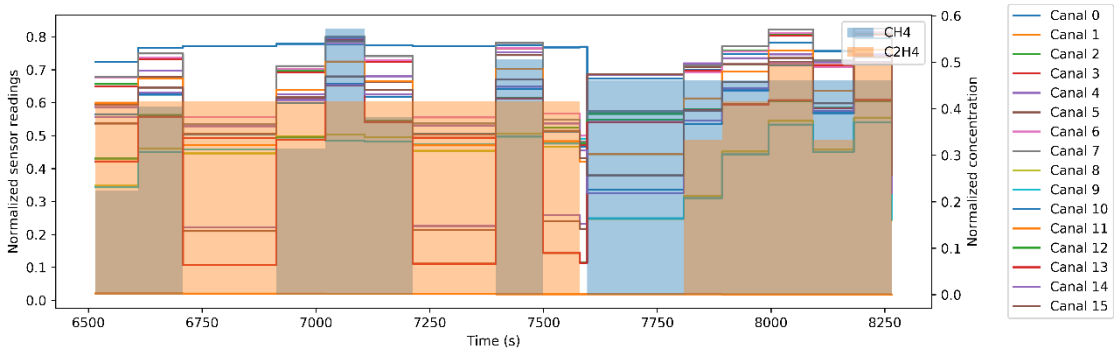

Figure 3: Left window averaged data.

With these two steps, we got a bijective data ready to train our network.

Network training

We are going to split the training process in two stages. The first one is a PyTorch regular training that allows approaching to the final learning point; but in the second stage, we are going to introduce the fixed-point quantization effect in the weights, limiting the values available for these model parameters, so that its implementation in the FPGA will use lower hardware requirements.

Network architecture

Before starting the training process, it is necessary to define the neural network architecture selected for this problem. In this case, we decided to use a feed-forward neural network with to hidden layers, the first one with 32 neurons and 12 in the second one. This makes the architecture to be a 16-32-12-2 since we have 16 sensors as inputs and 2 gas concentration levels as outputs.

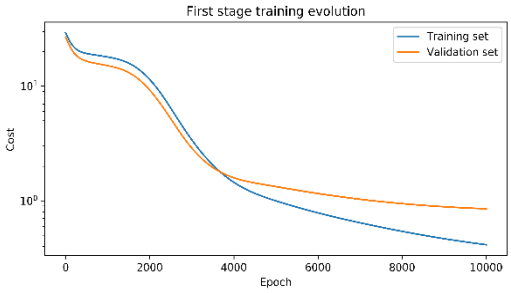

Figure 4 Evolution of the cost of the training and validation set during the first training stage.

First stage: Floating point learning

Now we are ready to start the training process. We train during 10^4 epochs using the mean square error to compute the cost value and Adam as training algorithm with a learning rate value of 10^{-4}. Figure 4 represents the training and validation sets' cost evolution during the learning. Once the model is trained, we test it over the time series. Results are shown in Figure 5.

Once we got closer to the final learning point (determined by the flattening trend in the validation cost evolution in Figure 4), we need to decide the minimum number of bits in the fixed-point representation will provide an accurate data representation. To evaluate this, we implement our neural network on C/C++, ready to be tested on HLS. Using the “ap_fixed.h”

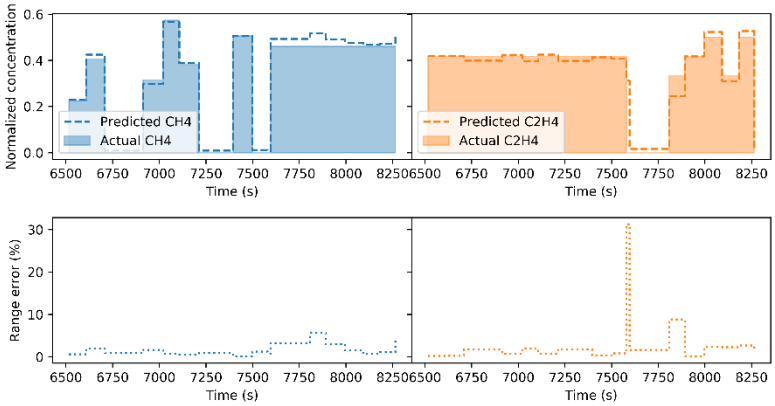

Figure 6: Predictions on the time dataset after the first stage training.

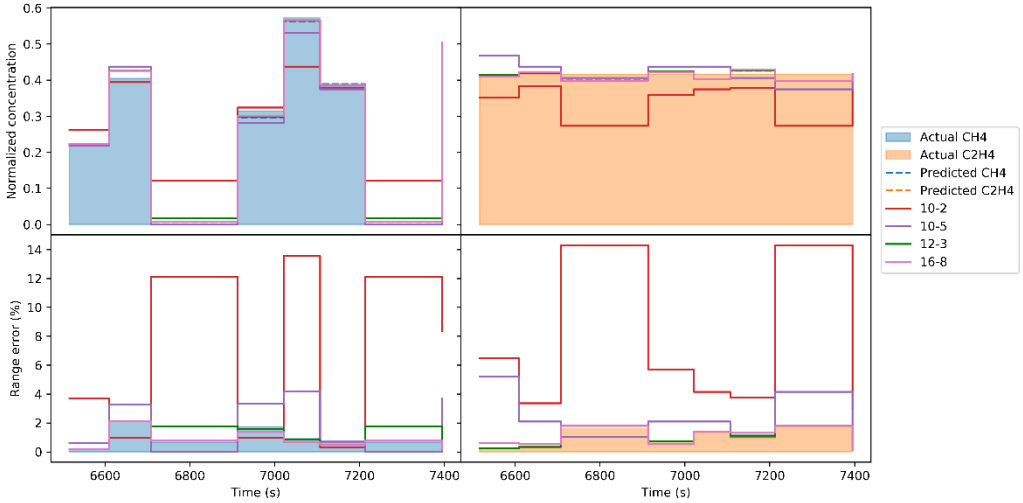

Figure 5: Neural network predictions and errors for four lengths of fixed points. The lines are labeled with the notation N-I, where N is the total number of bits used in the fixed-point value and I the number of bits used in the integer part.

we can declare fixed-point data types and “hls_math.h” give us an exponential function compatible with this data types, which will be used instead of the floating point nonlinear function in the full neural model simulated in Python. Using HLS simulation feature we obtain the results of the neural network prediction for the sensor readings using different lengths of fixed-point representation. The results are shown in Figure 6.

According to the results shown in this figure, we decided to use the 12-3 type (12 bits, 3 in the integer part), because it is very close to the PyTorch performance, but it uses 4 bit less than the 16-8 option, the other that is also closer to the PyTorch line.

Second stage: Quantized learning

Once we have decided the length of the fixed-point data type, we can start with the quantized training process. We start form the trained model in the first stage, following the training process with an extra step: the quantization of the neural parameters every 100 epochs. This make the parameters to be representable in 12-3 fixed-point type at the end of the learning process. The cost function evolution is shown in Figure 7 and the new neural model predictions in Figure 8.

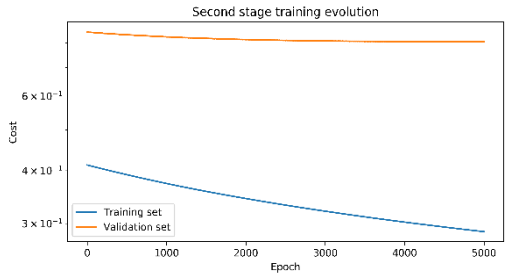

Figure 7: Second training stage evolution.

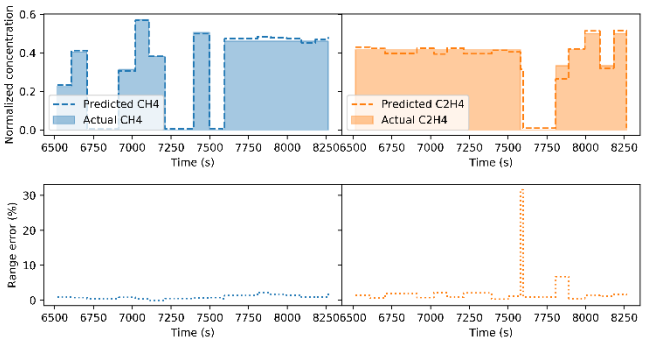

Figure 8: Results of the prediction after the second learning stage.

Error analysis

The average error is around the 2% for all the time series but there are some spikes in the error (see between 7500 s and 7750 s in Figure 8), bigger than 10%. These errors are due to singularities in the dataset, there are some asynchronous changes in the two gas concentration levels where the window average filter we used before does not perform well (as it can be seen in the Figure 3 for the same time range). As the filtered data does not recover the static value of the sensor in those points, the neural model works with inputs that are not related with the actual concentration levels in the atmosphere, showing that errors above the 10% (purple line in Figure 9).

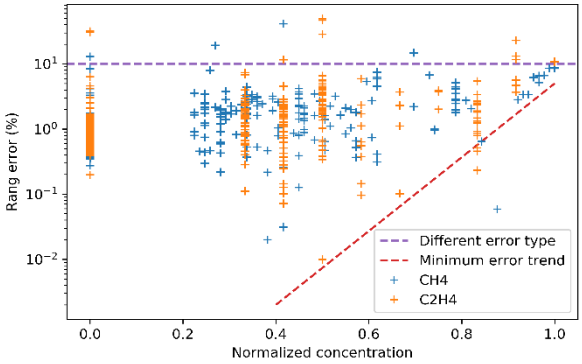

Figure 9: Plot of all errors respect of the concentration.

There is also another remarkable effect in the Figure 9. The bigger gases concentration levels, less number of measurements are available, reducing the number of training examples in this region, thus decreasing the accuracy of the network in there.

All of this Virtualization section is contained in the Virtualization.ipynb notebook.

Implementation

In the Virtualization section we defined the neural network architecture, decided the appropriate size of the fixed-point representation and trained a neural model using parameters in fixed-point representation. Now it is the moment to take all we have learned there and start with the neural network FPGA implementation.

Vivado HLS

The first step to do is the neural network model description in C/C++, available in the gas-nn.cpp source file. As we have said before, we have used it at the end of the first training stage, but for completeness we present it here. Due to its simplicity, the proposed feed forward neural network is easy to describe in C/C++. As you may know, connections between two layers are based in matrix multiplication, carrying out the expression

$$\boldsymbol{y} = W\boldsymbol{x} + \boldsymbol{b} \leftrightarrow y^i = \sum_j w_{i,j}x^j + b_i$$

Where $\boldsymbol{y}$ represents the output layer vector, y^i its components; W the weights matrix, $w_{i,j}$ its components; $\boldsymbol{b}$ the bias vector, b_i its components; and $\boldsymbol{x}$ the input layer vector and x^j its corresponding components.

In order to program this on C/C++, we need two nested loops for each connection between layers: one external loop iterating over the outputs (Layer[Orig][End]Ext in the code) and the inner one going over the inputs (Layer[Orig][End]Int in the code). Since we have four layers, we must do three matrix multiplications.

To define the fixed-point data types, we use the Vivado HLS “ap_fixed.h” library. This let us to define every number as a 12-3 fixed-point data type. Also, we need way to calculate the sigmoid output of the fixed-point data values at the hidden layers neurons. For this, we use the exp(.) function provided by the “hls_math.h” library.

Once we complete the neural network C/C++ description, we need a test bench to evaluate it over the Vivado HLS C Simulation feature. For this, we program a test bench file (also available on the sources provided), that read the network parameters and test the neural network over the complete time series. The result of the neural network HLS simulation is shown in Figure 10.

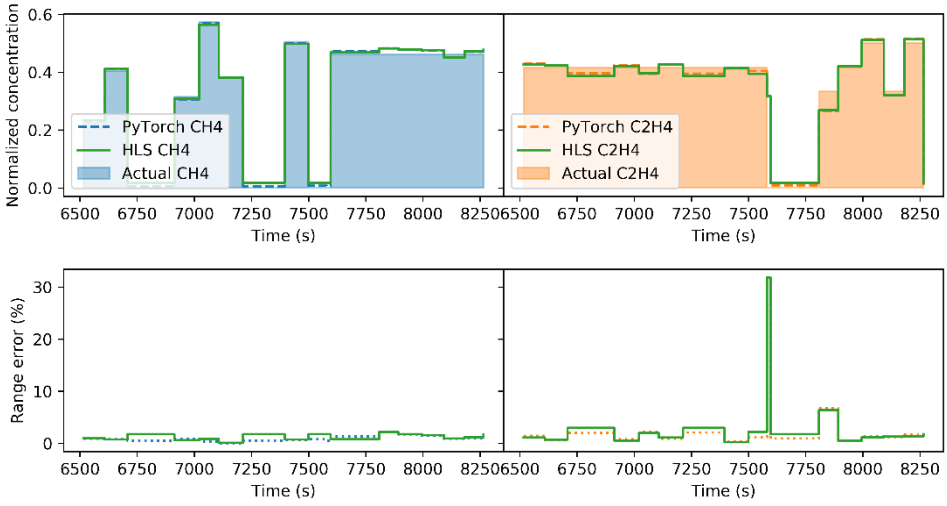

Figure 10: HLS simulated neural network predictions with the error.

As you can see, the errors obtained from the HLS simulated neural network slightly increase compared to the values provided by the PyTorch quantized version. This is because in the HLS simulated network all the parameter values are represented in fixed-point (while in PyTorch only weights are quantized), thus increasing the error at the output. Even so, the average error in all the time series keeps below 2.5%.

	Latency (cycles)	DSP (%)	FF (%)	LUT (%)
Interface-only directives	5615	5	2	11
Unroll	3797	31	6	26
Unroll+Pipeline	274	31	11	29

Table I: Latency and FPGA resources usage for three synthetizations with different directives.

Now it is time to synthesize the HLS design. The first directives we included corresponds to the interface, declaring all ports (inputs, parameters and outputs) as AXI slaves. This will let us to control them from Python when the design is loaded on the PYNQ's FPGA. So, as a starting point, we launch an interface-only synthetization to determine the initial latency in our design. With these conditions, latency time -the time the system requires to provide an output after an input is presented, is 5615 clock periods. Thus, considering a 10 ns clock period the inference time is 56.15 µs. To scale this number, the inference process programmed in Python takes 5.6 µs, 10 times less, when it is executed in an Intel i5-3470 @ 3.5 GHz CPU. On the other hand, the FPGA resources usage it is very low, as it can be seen in Table I.

Since we want to reduce the inference time to be, at least, comparable with the CPU inference time, we will apply the unroll directive over the inner loops. This let us to reduce the latency using more FPGA resources. Insomuch as the inner loops perform an accumulation function (the summation term of the neurons), the unroll directive cannot fully parallelize the process because each epoch depends on the result of the previous one. With the aim of reduce this as much as possible, we apply an additional pipeline directive on the external loops. Then by using both directives the latency time is reduced up to 274 clock cycles, namely 2.74 µs, a half of the CPU inference time.

Although we have synthesized the unrolled and pipelined design, we could not validate it due to the CoSimulation computational costs. That is why we have used the non-optimized design henceforth.

Vivado IP integrator

Using its RTL exportation feature, we can pass our design from Vivado HLS to Vivado. Taking advantage of the Vivado IP integrator and its autoconnection functionality we can connect very fast the ZYNQ processing system to the neural network module block. Once all the

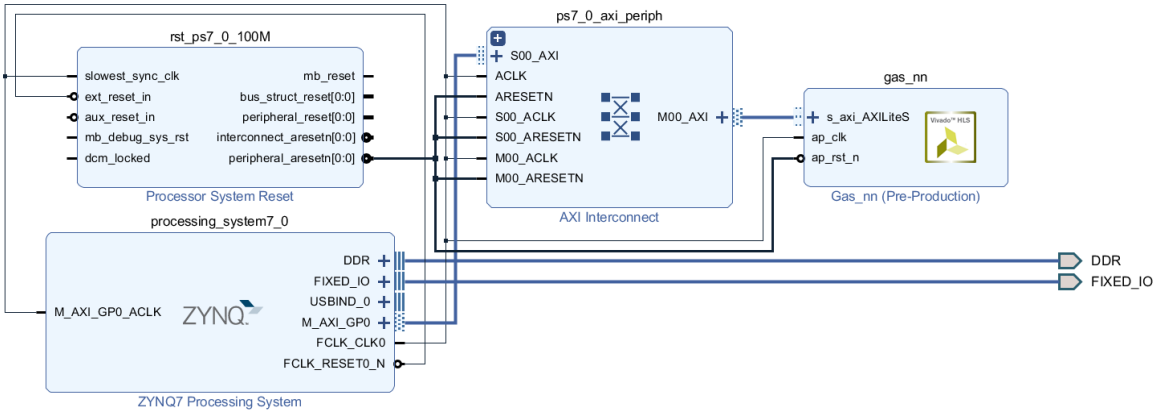

Figure 11: Block design capture. Note that the ZYNQ processing system and the gas_nn block are connected between an AXI interconnect block. The fourth block it is just a reset block.

blocks are properly connected (Figure 11), we are ready to run the synthetization and implementation of the block design, that will be followed by the bitstream generation.

Use of the FPGA implementation

In this last section we describe how to use the FPGA implemented neural network model. Once we have generated the bitstream, we upload it together with the .tcl and .hwh files of the block design to the overlays PYNQ folder. Using the overlay function in the pynq Python package we can load the design to the FPGA. Due to the use of the selected fixed point data configuration, the ports size become unrecognizable for the register map feature. In order to make the IP more user friendly, we have coded a driver using the port map generated on Vivado HLS that is shown in the _hw.h file. This driver is based on the DefaultIP driver included in the pynq package, and it is composed by two main functions: .set_params(.), that allows loading the model parameters to the IP; and .pred(.), coded to write the sensor data on the input ports and read the output ones. Both the driver and the way it can be used is available in the Usage.ipynb notebook on this project repository.

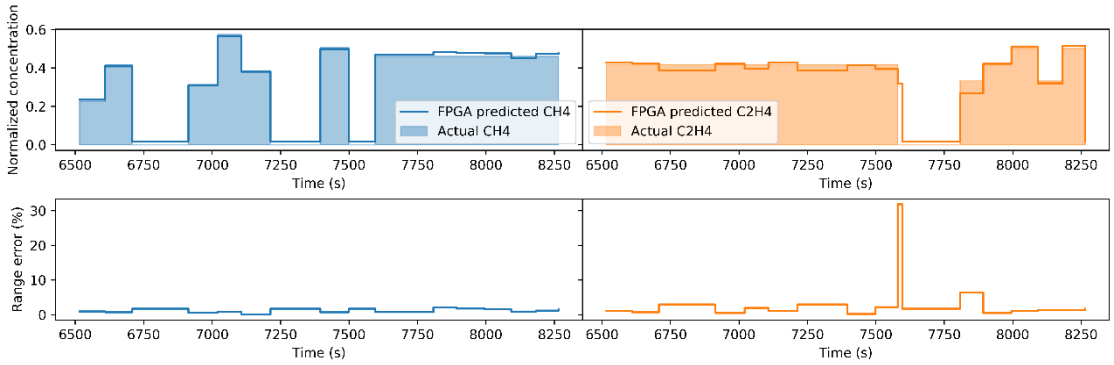

Figure 12: FPGA predicted results.