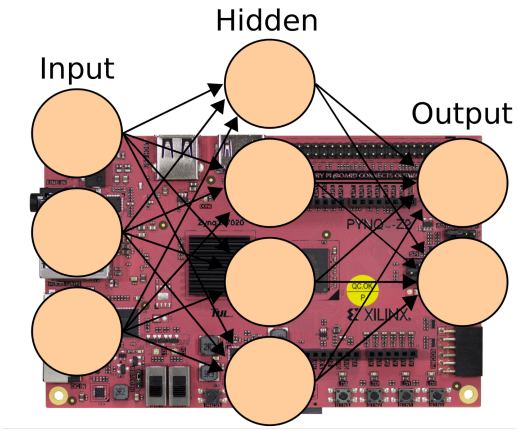
ECG interpretation on FPGA for Edge AI in the StorAIge project

October 1, 2024

Overview

This document presents the development experience and results obtained within the *StorAge* project. One of the parts of the project was the development of an electrocardiogram (ECG) interpretation system on a field programmable gate array (FPGA) platform. This document refers specifically to that part of the *StorAIge* project, omitting everything that does not concern the magic of AI and its implementation on the FPGA.



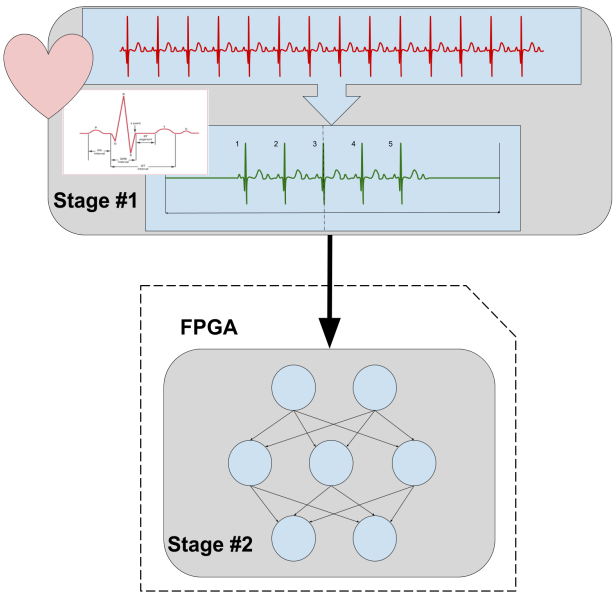
We use the following open-source software and technologies - Tensorflow/Keras/Pynq/HLS4ML as high-level tools to achieve our goal. Thus, this document is essentially a digest of all the most interesting things that we encountered while working on the project and what was hidden in open sources or not clearly described concerning this software. Of course, the emphasis is on practical value for additional study and in no way a comprehensive guide to learning AI. Some things in the document are presented as well known, but make reading more comfortable for an uninitiated specialist. It is impossible to create a document that describes in detail the process of developing a deep learning system from building and training a model in Python to implementing a working solution on an FPGA through conversion chains and synthesis, so some things remain beyond our attention.

Content

HW/SW solution in general

In a nutshell, we developed FPGA-based Neural Network for Multi-Class Classification of preprocessed ECG. Diagnosis of heart rhythms is solved using the canonical Machine Learning method of classification using a pre-trained model. However, the technical solution to this problem is made by a non-trivial way of interpreting segmented electrocardiogram (ECG) signals on a field programmable gate array (FPGA) platform. Segmentation of the ECG signal in the first stage and, therefore, possible classification by a compact model in the second stage, allows achieving the results required for Edge AI. Hardware implementation of ML algorithms further increases this efficiency.

Two stages processing of ECG for resource-efficient classification:



- **Resampling and amplitude normalization**
- **5 QRS peaks segmentation**
- **Spatial normalization in 8.5 s window**
- **Low / High pass Filtering (by DWT)**
- **Denoising with wavelets (by DWT)**
- **Removing baseline wander (by DWT)**
- **Augmentation**

- **2 x 2831 samples input (5-QRS segments)**
- **2 x Convolutions extract spatial features**
- **3 x Fully-connected layers**
- **4 Classes categorical classification**

Segmentation and spatial normalization allow the use of a small Neural Network with just **22626** (88.38 KB) parameters.

ECG data

Minimal knowledge of ECG and dataset

ECG is the primary, non-invasive, and inexpensive technique popular for heart abnormality diagnosis. An ECG is a graphic recording of electrical impulses and stimulus variations to the time captured using electrodes. The ECG machine processes the signals picked up from the skin by electrodes and produces a graphic representation of the electrical activity of the patient's heart. The basic pattern of this electrical activity was first discovered over a hundred years ago. It comprises three waves, which have been named P, QRS (a wave complex), and T (see Figure 1) [1].

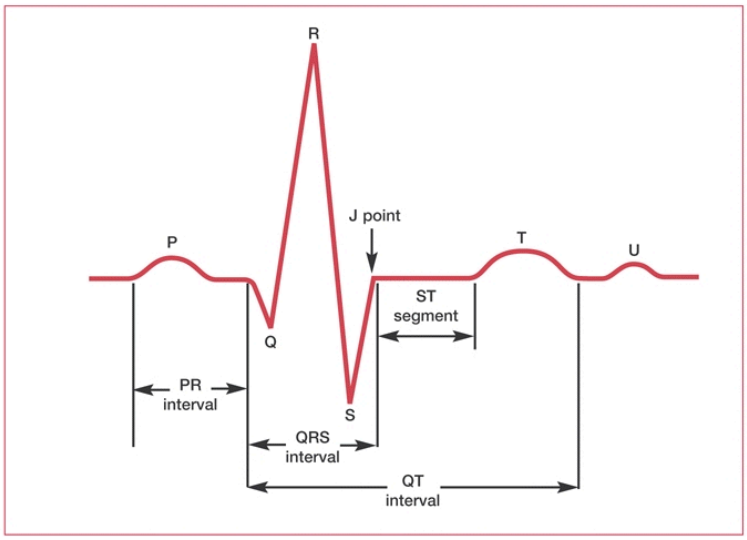


Figure 1. The basic pattern of electrical activity across the heart.

Any variation in the source, rhythm, and rate of these electrical stimuli is reflected in the characteristic P-QRS-T waveforms that imply cardiac arrhythmia. As a non-invasive test, long term ECG monitoring is a major and vital diagnostic tool for detecting these conditions. This practice, however, generates large amounts of data, the analysis of which requires considerable time and effort by human experts. Advances in modern Machine Learning and statistical tools enable training on high-quality big data to achieve exceptional levels of automated diagnostic accuracy. Such classification methods require large size data that contain all prevalent types of conditions for algorithm training purposes. We use this Chapman University database that we found for free download and which is described in detail in the scientific paper [2]:
https://www.nature.com/articles/s41597-020-0386-x

The dataset contains 12-lead ECGs from 10,646 patients, containing 11 common rhythms and 67 additional cardiovascular conditions, all labeled by professional experts. Data can be downloaded from [3]:
https://figshare.com/collections/ChapmanECG/4560497/2
We need 3 files `Diagnostics.xlsx` , `ECGData.zip` , `ECGDataDenoised.zip` that we need to be placed/unzipped into:

```
~/StorAIge/SignalDatabase/Chapman/ECGData/
~/StorAIge/SignalDatabase/Chapman/ECGDataDenoised/
~/StorAIge/SignalDatabase/Chapman/Diagnostics/
```

Data recorded as CSV files contain 5000 rows and 12 columns with header names presenting the ECG lead.

The standard ECG uses 10 cables to obtain 12 electrical views of the heart [1]. The different lead signals reflect the angles at which electrodes "look" at the heart and the direction of the heart's electrical depolarization. The first 6 leads are Limb leads. The rest 6 leads are Chest leads. Three bipolar leads and three unipolar leads are obtained from three electrodes attached to the left arm, the right arm, and the left leg, respectively. (An electrode is also attached to the right leg, but this is an earth electrode.) The bipolar limb leads reflect the potential difference between two of the three limb electrodes:

- lead I: right arm–left arm
- lead II: right arm–left leg
- lead III: left leg–left arm

The unipolar leads reflect the potential difference between one of the three limb electrodes and an estimate of zero potential – derived from the remaining two limb electrodes. These leads are known as augmented leads. The augmented leads and their respective limb electrodes are:

- aVR lead: right arm
- aVL lead: left arm
- aVF lead: left leg

By building test Machine Learning models on datasets with varying numbers of ECG leads, we found that the simplest release could only use the first 2 leads. Testing did not reveal a significant deterioration in accuracy when classifying different types of arrhythmias. So, for simplicity, we only take the first two measurements from the ECG data files. These CSV files are named by unique IDs. We only use **denoised ECG data files** and **diagnoses files**.

Continuing simplifications, we also reduced the number of diagnoses to 4.

We transformed the Chapman University dataset into a significantly different one:

- Using 2 leads only
- Label quantity reduced from 11 to 4
- Amplitude and spatial normalization
    - Sampling rate reduced from 500 to 333 samples per second
    - 10-second sample reduced to 8.5-second
    - The sample is segmented with 5-heart beats with 3-rd heartbeat placed in the center

Resampling, reducing label quantity and amplitude normalization

Our python script ***ecg_saple_class.py*** defines source labels as the following:

```
#  Acronym Name | Rhythm                                          | Frequency (%)
#  -------------|-------------------------------------------------|---------------
self.ECG_labels = [
       'AF',    # Atrial Flutter                                  | 445 (4.18)
       'AFIB',  # Atrial Fibrillation                             | 1,780 (16.72)
       'SR',    # Sinus Rhythm                                    | 1,826 (17.15)
       'AT',    # Atrial Tachycardia                              | 121 (1.14)
       'AVNRT', # Atrioventricular Node Reentrant Tachycardia     | 16 (0.15)
       'AVRT',  # Atrioventricular Reentrant Tachycardia          | 8 (0.07)
       'SA',    # Sinus Arrhythmia (SI - Sinus Irregularity)      | 399 (3.75)
       'SAAWR', # Sinus Atrium to Atrial Wandering Rhythm         | 7 (0.07)
       'SB',    # Sinus Bradycardia                               | 3,889 (36.53)
       'ST',    # Sinus Tachycardia                               | 1,568 (14.73)
       'SVT',   # Supraventricular Tachycardia                    | 587 (5.51)
]
#  ------------------------------------------------------------------------------
#                                                        Total: | 10,646 (100)
```

These 11 labels are reduced to 4 ones:

```
self.labels =  ['SR', 'SB', 'AF', 'GSVT']
```

In the following way:

```
labels_dict = dict()
for i in range(len(self.ECG_labels)):
    if self.ECG_labels[i] in ['SR', 'SA']:
        labels_dict[self.ECG_labels[i]] = 0
    elif self.ECG_labels[i] == 'SB':
        labels_dict[self.ECG_labels[i]] = 1
    elif self.ECG_labels[i] in ['AF', 'AFIB']:
        labels_dict[self.ECG_labels[i]] = 2
    else:
        labels_dict[self.ECG_labels[i]] = 3
```

Primary Sample Extractor ***pse.py*** is exactly the script that should be executed first and that extracts data from the original Chapman University dataset doing validation, resampling, reducing label quantity and amplitude normalization. At this stage, the simplification ends, but a rather complex data transformation begins.

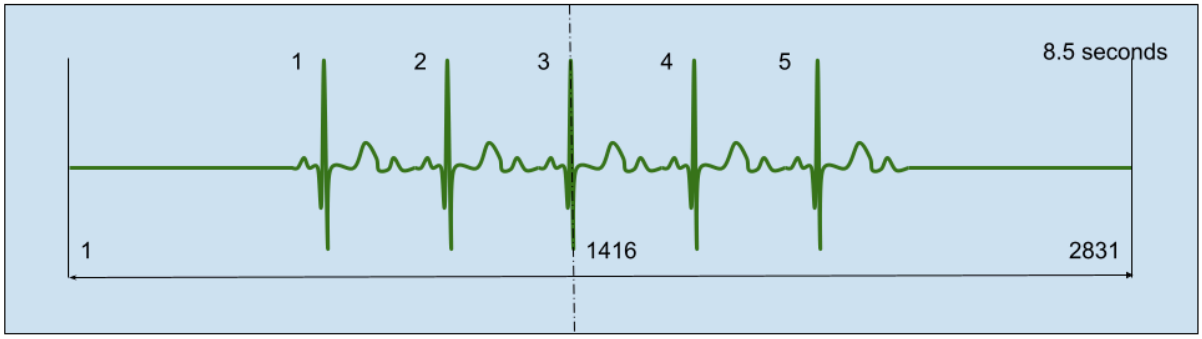
Note: These above mentioned scripts and the rest files of the project can be obtained from the gitlab repository [4]: https://git.strikersoft.com/edgeai/ecg

```
~ cd StorAIge
~/StorAIge/ git clone https://git.strikersoft.com/edgeai/ecg
~/StorAIge/ git switch dev
```

Spatial normalization

We found an opportunity to convert ECG data into a form that can be analyzed with sufficient accuracy even by a simple ML algorithm that does not require excessive resources. This conversion is done at the stage before the execution of the AI algorithms in a software way, but can be embedded in the solution on the FPGA. The main idea is spatial normalization. The proposed Machine Learning model makes inference from the spatial features of extracted and processed time-window of original ECG and classifies the signal into four arrhythmia types. The 5 heartbeat segments are extracted and aligned to be in a time window of 2831 samples of 8.5

seconds in length such that the 3-rd beat segment is fixed by keeping the R peak in the center (sample 1416). The signals are cropped/padded if they are longer/shorter than 2831 samples.



To get it smooth and right the Discrete Wavelet Transform (DWT) is used. Corrected DWT coefficients of DWT-decomposed segment window signals for each ECG lead are used for filtering, denoising and removing baseline wander while DWT-reconstracting. Such preprocessing allows to reach good results without extracting spectral features and keep the model simple doing the Deep Learning AI algorithms.

ECG data as input for AI/ML

An AI/ML input refers to an example (sometimes also known as sample, observation or data point) x from a dataset that we pass to the AI/ML model. Since we have an ECG in the form of a time series of samples x_j for j from 1 to 2831, we will use the term sample x_j to denote the point on the ECG received from the ADC, and for the AI/ML input we will use the term example x_i. Each example x_i in turn is a vector that consists of samples x_j. We collect pre-processed (segmented, normalized, etc.) ECG examples x_i as labeled dataset $\mathbf{D = \{(x_i,y_i)\}^N_{i=1}}$, where x_i is the $\mathbf{i}^{th}$ input and $\mathbf{y_i}$ the corresponding label (aka target or output). Thus AI/ML is the function $f{:}X \to Y$, then $x \in X$ is the input and $f(x) = y \in Y$ is the output of the function for that input vector or tensor x.

The ***pse.py*** script we run first to get the data we need from the primary Chapman University dataset gives us the following report:

```
100%|██████████| 10646/10646 [03:18<00:00, 53.50it/s]
Dataset len: 10588
100%|██████████| 10588/10588 [00:07<00:00, 1440.59it/s]
(10588, 3333, 2) (10588, 4)
```

```
'X' is (10588, 3333, 2), 10588 samples x 3333 timesteps. 'y' is (10588, 4) labels, 10588 length
Sample points: 3333
Sample leads:  2
LABELS:        ['SR', 'SB', 'AF', 'GSVT']
====================
```

This means that of the 10,646 records found in the Chapman University dataset, only 10,588 were valid for our follow-up and the first two ECG leads. Resampling was performed and an intermediate dataset with shape (10588, 3333, 2) was formed. That is, 3333 samples of a 10-second window for two ECG leads in a dataset of 10588 examples. Every example is labeled with one of four possible abbreviations ['SR', 'SB', 'AF', 'GSVT']. Looking at the ***pse.py*** and ***ecg_saple_class.py*** scripts, we can see that the scripts further standardize (normalize) and convert the ECG signals to numpy arrays and represent the labels as OneHot encoding vectors. This intermediate dataset is written to the ***pse/ecg_data_set.npz*** file.

The following ***segmentation.py*** script launched for spatial normalization converts the intermediate dataset into the target one with the following report:

```
X:  (10588, 3333, 2)
sample_index:  None debug:  False
 83%|████████▎ | 8789/10588 [01:59<00:26, 66.87it/s]Exception:  index 0 is out of bounds for axis 0 with size 0, 8778, SB
100%|██████████| 10587/10588 [02:34<00:00, 34.47it/s]
segments:  (23511, 2831, 2)

dwt_high_coeffs:  (23511, 363, 2)
dwt_mid_coeffs:  (23511, 187, 2)
dwt_low_coeffs:  (23511, 99, 2)
100%|██████████| 10588/10588 [02:36<00:00, 67.71it/s]
```

This means that example 8789 was rejected because it failed the 5-peak segmentation test. Another very important feature of this script is the augmentation of the data set to the level of 23511 examples. This became possible because from ECG signals that have a large (different) number of QRS-peaks, we form signals that have exactly 5 peaks. But, it is clear that segmentation is the most interesting. This segmented dataset is saved in ***pse/ecg_data_set_segments.npz*** as a numpy array and is used as an input source for the next script that should prepare input data for AI/ML. This ***start.ipynb*** script is Jupyter Notebook that splits dataset into train and test parts with the following report from the 3-rd cell:

```
Loading of Segmented ECGs ...
Segmented ECGs are loaded and 'X', 'y', and 'Age' are prepared and saved
The type of 'X' data is  <class 'numpy.ndarray'>
The type of 'y' data is  <class 'numpy.ndarray'>
'X' is (23511, 2831, 2), 23511 samples x 2831 timesteps. 'y' is (23511, 4) labels, 23511 length
====================
```

```
###############################################
Sample points: 2831
Sample leads:  2
X_ecg_train shape: (19511, 2831, 2), X_ecg_test shape: (4000, 2831, 2)
X_age_train shape: (19511,)       , X_age_test shape: (4000,)
y_train shape:     (19511, 4)     , y_test shape:     (4000, 4)
LABELS:            ['SR' 'SB' 'AF' 'GSVT']
```

The result of the script is folder ***work/model_1*** with the set of 5 files:

X_ecg_train.npy

X_ecg_test.npy

y_train.npy

y_test.npy

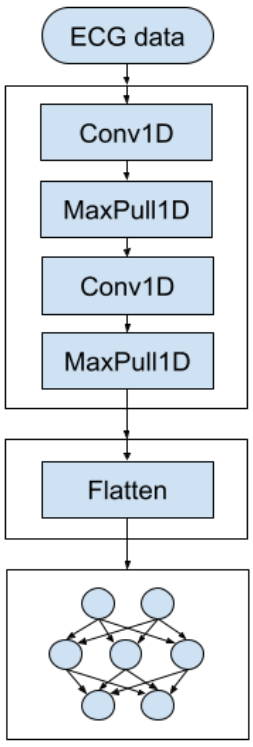
labels.npy

We use this set of 5 files for further experiments and as a target dataset for AI/ML models. Such preprocessing was found to be optimal together with the optimal AI/ML model as a result of repeated tests, analysis, changes, and modifications of all components of the system, from the type of input data to the model synthesized in the FPGA. Of course, the optimality of this solution was determined by many factors, one of which is the simplicity of the entire system design cycle and the compactness of the solution.

At this stage, preprocessing ends and the work of AI/ML models begins.

AI/ML, what exactly are we dealing with?

AI is a set of technologies built into a system that allows it to reason, learn, and act to solve complex problems by mimicking the cognitive functions associated with human intelligence. It sounds cool, abstract and incomprehensible. But slam the door and dive into the essence of the project! Rather, we have developed machine learning (ML), which is a type of artificial intelligence that automatically allows a machine or system to continue learning and improving on its own based on experience using a mathematical data model, helping it learn without direct instructions. In short, ML is AI that can automatically adapt with minimal human interference. Deep Learning is a subset of ML that uses artificial neural networks to mimic the learning process of the human brain. **From a marketing point of view, it is better to say that we made a system with AI, but from a technical point of view it is more correct to call it ML or even Deep Learning(DL)**. Deep Learning studies the development of mathematical methods that allow software to learn autonomously, without explicitly describing each operating rule. In our case, its goal is to extract hidden features from ECG data that allow us to assign each input signal to a specific class. The catch is that there is no rule statement, but there is an adaptive model that adjusts its parameters according to the input data it receives, improving the resulting estimates as new input samples arrive during training. Later we are going to explain NNs in detail by going through their components step by step. For now, given the diagram, just keep in mind that we use a few types of neural networks as part of our artificial intelligence algorithm. Let's look at our model without details. 1-Dimensional Convolution Neural Network (CNN) layers create feature maps that extract a region of the 5-peak ECG signal, which is divided into segments and sent for nonlinear processing after flatten. Each neuron only processes the information from a small part of the ECG. This processing uses principles from linear algebra, especially matrix multiplication, to detect patterns within ECG signals. The CNN contains two convolutional layers followed by pooling. In this way, we can extract some features from the ECG data and feed them into the next chain of the AI algorithm for classifying. All these algorithms being implemented on



FPGA meet or exceed the performance of GPUs but have a lower power consumption that is well suited for embedded applications.

Something from AI/ML that we use here

- An AI system performs its task by typically predicting output values based on given inputs using a *machine learning algorithm.*
- In general, ML is all about making *predictions* and *classifications*. In this project, we use a learning algorithm to make categorical classification. In a classification problem, we use the information contained in the data to predict the class $\mathbf{y_i}$ of the example x_i.
- Most ML algorithms are broadly categorized as being either *supervised* or *unsupervised*. In this project, we use the algorithms of supervised classification. These algorithms deal with clearly labeled ECG data. Labels (classes) are stored in the ***labels.npy*** file.
- We need the ECG data to:
    - "*Train*" the machine learning methods using ***X_ecg_train.npy*** and ***y_train.npy*** files.
    - "*Test*" the machine learning methods using ***X_ecg_test.npy*** and ***y_test.npy*** files.

Neural Networks

There are several types of machine learning algorithms. We use Artificial Neural Networks. ANNs, or simply Neural Networks (NNs), are groups of algorithms that recognize patterns in input data using building blocks called neurons. These neurons are much like the neurons in the human brain.

Convolutional Neural Networks (CNNs)

A convolutional neural network (CNN) is a type of neural network that learns features by itself via filter (or kernel) optimization. There is no special “physical” meaning in the 1D convolution operation, except that by convolving the ECG signal with a kernel (another signal), we obtain a new signal, which is the sum of an infinite number of copies of the impulse response, each of which is shifted by a delay time. Something like a “weighted sum of memories” or echoes. Thus a discrete-time convolution generalizes a moving average. Like a moving average, it smooths a time series, as we shall see. A CNN is effective at identifying simple patterns in data, which are then used to form more complex patterns in higher layers. We use a 1D CNN, which uses multiple convolutional layers that are not fully connected but pooled. Pooling involves chunking a vector into non-overlapping equal-sized groups or ‘pools’, and then taking a summary statistic for each group. This further smooths out noise in the local dynamics and downsamples the tensor that is passed on to subsequent layers. CNN layers create feature maps that record a region of the ECG signal, which is eventually divided into segments and sent for nonlinear processing.

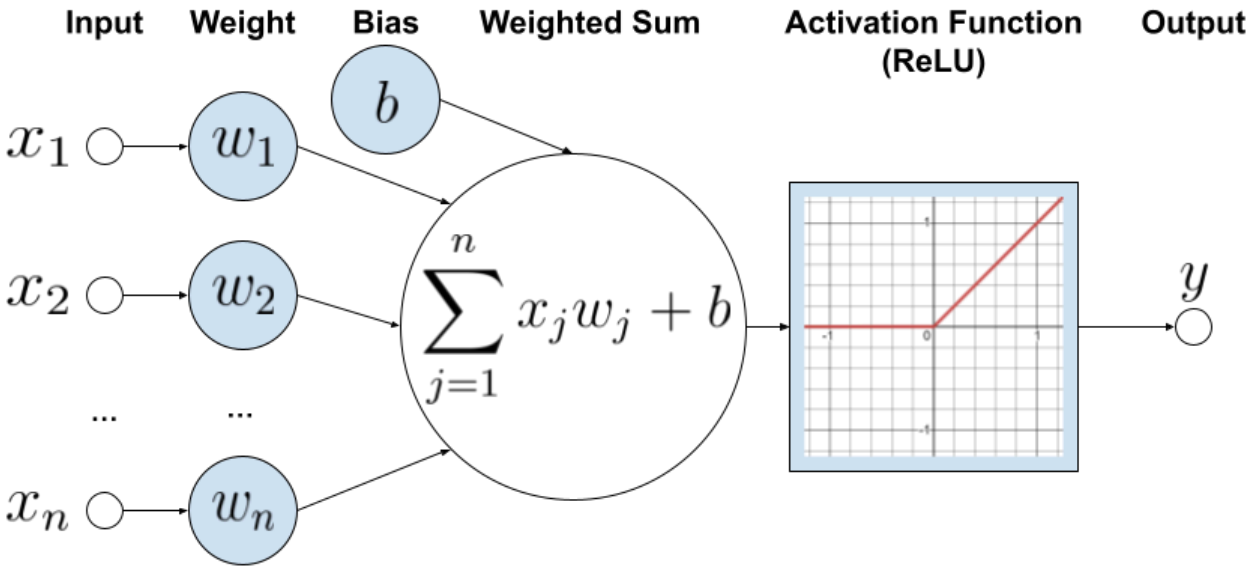
Dense Neural Networks (Fully Connected layers)

A Neural Network in machine learning model in which each layer is deeply connected to the previous layer. One of the simplest options for neural networks. They transmit information in one direction through various input nodes until it reaches the output node. Since this is a supervised learning algorithm that classifies data into categories, it is a classifier with neurons in the last layer containing the probabilities that the ECG under test belongs to a certain class.

As we can see in the diagram given above, the Neural Network consists of an input layer, an output (target) layer, and a hidden layer in between the two. Each layer consists of nodes. The layers are connected, and these connections form a network of interconnected nodes. In general, a Neural Network consists of Nodes and Connections between the Nodes.

Understanding Neurons in Deep Learning

Neurons are a critical component of any AI machine learning algorithm. Neurons in ML are Nodes through which data and computation flow. Here is a diagram of the functionality of a Neuron:



Once a Neuron receives its inputs from the neurons in the preceding layer, it adds up each signal multiplied by its corresponding weight. The bias value is included in this value. This sum is passed on to an activation function. The activation function calculates the output value for the Neuron. There are many bent or curved lines that we can choose for an activation function:

- ReLU - Rectified Linear Unit f(x) = max(0, x)
- softplus f(x) = log(1 + e^x)
- sigmoid f(x) = e^x / (e^x + 1)
- hyperbolic tangent f(x) = tanh(x) = sinh(x) / cosh(h) = (e^x - e^-x) / (e^x + e^-x)

We consider a slightly modified version of the simplest model of a neuron proposed by Frank Rosenblatt in 1957, called 'Perceptron'. All modifications are made for the sake of simplicity, because we are not going to deeply explain the operation of neural networks. Our neuron is a mathematical function:

$$y = max\left(0, \left(\sum_{j=1}^{n} x_j w_j + b\right)\right)$$

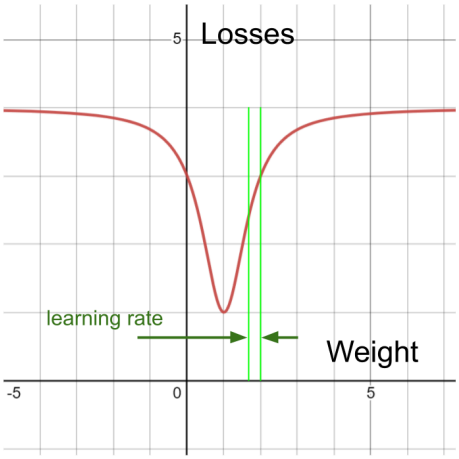
It takes n numbers as input, which we denote as x_j where j stands for index of input. For each input x_j neuron assigns another number w_j. A vector consisting of these numbers w_j is called Weights vector or parameters. This vector also includes bias b. These parameters make each neuron unique and are fixed during "Test"-ing. But during "Train"-ing we are going to change these numbers to "tune" our network.

A neural network is also a mathematical function. It is defined by a bunch of neurons connected to each other. This means that the output of one nonlinear function is used as input to other nonlinear functions, and all functions have trained parameters and operate on vectors. Thus, neurons, being mathematical functions with input data of a fixed length, when connected to a network, will also require data of a fixed length at the input layer of this network. We're going to pass our ECG samples as inputs to our neural network. These ECG samples are an array of numbers, each of which indicates the amplitude of the signal in the ECG lead during analog-to-digital conversion. We cut a window in the multi-lead ECG data, and pass all these numbers as a vector to the input layer of the network. After a chain of calculations in the previous layers, our neuron, being inside the network, receives data in the form of an array of numbers, each of which indicates some information associated with some feature of the ECG data that we fed to the input of our network.

With a nonlinear calculation, the neuron produces data where information associated with certain features of the ECG data is more characteristic. When classifying an ECG, we need the output of our neural network to be a number between 0 and 1 so that we treat it as a probability. However, our solution does some tricks to avoid complex nonlinear calculations keeping just simple ReLU ones. More details will be presented later in this document.

Loss Function and Training

During training, we pass each ECG example x_i through the network to produce 4 probability numbers as a product of activation functions of 4 neurons on the last layer. Loss function tells us how good our neural network is for our task. The purpose of the loss function is to take the output probabilities and measure the distance from the true values. The Categorical Cross-Entropy function solves this problem. During model training, the model weights are iteratively adjusted accordingly with the aim of minimizing the Cross-Entropy loss.The process of adjusting the weights is what defines model training and as the model keeps training and the loss is getting minimized, we say that the model is learning. When adjusting the neurons' weights and biases, the optimization algorithm uses a learning rate parameter, which effectively tells it how big the adjustments should be. Large values for the learning rate translate as large adjustments, and smaller values translate as small adjustments. Large adjustments can change the values of the weights and bias by too much, causing the neuron's output to diverge even more from the expected output. Very small adjustments might mean that the algorithm is taking too long to learn. In a neural network we think of inputs x, and outputs **y** as fixed numbers. The variable with respect to which we're going to be taking our derivatives are weights w, since these are the values we want to change to improve our network. If we compute the gradient of the loss function with reference to our weights and take small steps in the opposite direction of gradient our loss will gradually decrease until it converges to some local minima.



Working on the form and content of the input data, on the structure and parameters of the ML model, we constantly trained and tested the model, finding out what accuracy indicators were achieved as a result of certain changes made. The process of improving the performance of the entire system is similar to how the ML model is trained, only the criterion for improving the system is increased accuracy and fewer resources used, and for the ML model there is a loss.

Tensorflow/Keras/QKeras

TensorFlow is a rich system for managing all aspects of a machine learning system. It is developed by Google and is the de facto industry standard. TensorFlow is used to build and train deep learning models as it facilitates the creation of computational graphs and efficient execution on various hardware platforms. Keras is a high-level neural networks API, capable of running on top of Tensorflow. It enables fast experimentation through a high-level, user-friendly, modular and extensible API. Keras can also be run on both CPU and GPU. QKeras is a quantization extension to Keras that provides drop-in replacement for some of the Keras layers, especially the ones that create parameters and activation layers, and perform arithmetic operations, so that we can quickly create a deep quantized version of Keras network. The quantized model generally refers to models that use an 8-bit signed or unsigned integer data format to encode each weight and activation. All experiments we do with Keras model that uses 32-bit float point format for all data that are in computation flow. After this Keras model is prepared to be QKeras with reduced data format and then is re-trained in a manner of quantization-aware training (QAT).

We use TensorFlow/Keras for developing the model and finding optimal solutions, but at the implementation stage we use TensorFlow/QKeras. This is general practice for Edge ML.

ML model that we build

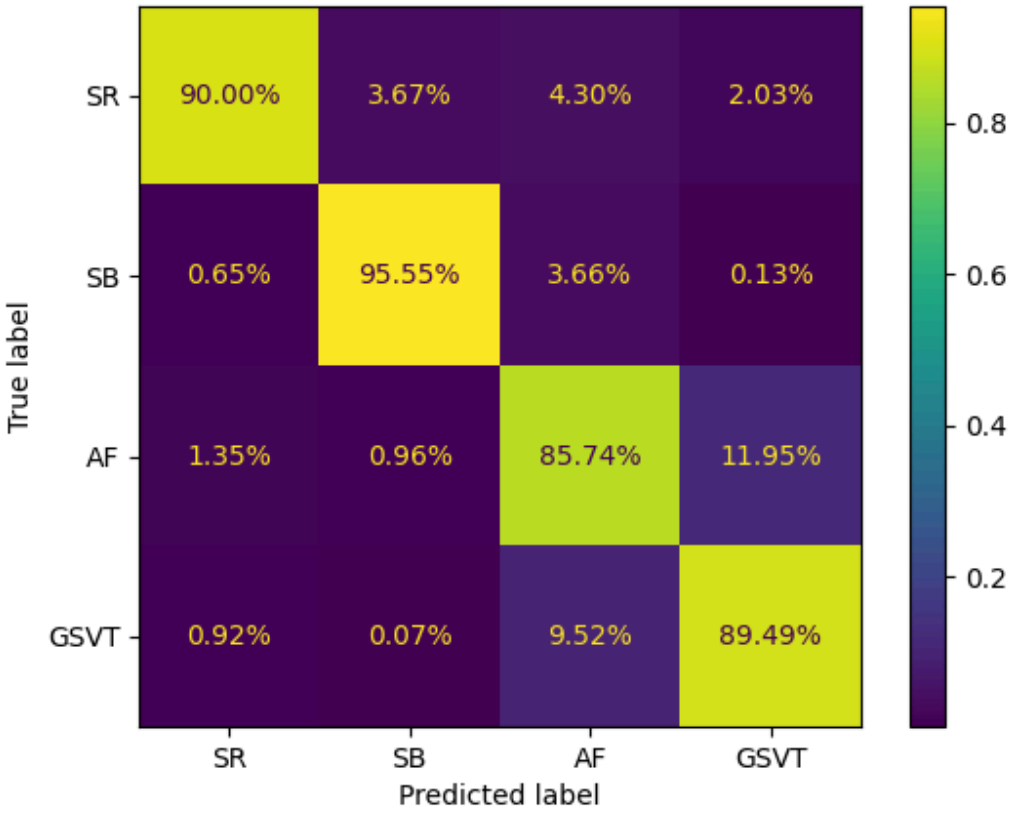
Below is a summary of the quantized model we use for classification in the project. A simplified diagram of this model is shown at the beginning of this brochure. This summary provides precise, layer-by-layer information on how the model sees tensorflow:

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv_1 (QConv1D)            (None, 1408, 4)           140

 bn_conv_1 (QBatchNormaliza  (None, 1408, 4)           16
 tion)

 relu_1 (QActivation)        (None, 1408, 4)           0

 pool_1 (MaxPooling1D)       (None, 704, 4)            0

 dropout_1 (Dropout)         (None, 704, 4)            0

 conv_2 (QConv1D)            (None, 345, 2)            122

 bn_conv_2 (QBatchNormaliza  (None, 345, 2)            8
 tion)

 relu_2 (QActivation)        (None, 345, 2)            0

 pool_2 (MaxPooling1D)       (None, 172, 2)            0

 dropout_2 (Dropout)         (None, 172, 2)            0

 flatten (Flatten)           (None, 344)               0

 dense_1 (QDense)            (None, 64)                22080

 relu_3 (QActivation)        (None, 64)                0

 dropout_3 (Dropout)         (None, 64)                0

 output_dense (QDense)       (None, 4)                 260

=================================================================
Total params: 22626 (88.38 KB)
Trainable params: 22614 (88.34 KB)
Non-trainable params: 12 (48.00 Byte)
_________________________________________________________________
```

This information we can find in the output cell **№7:quantization.ipynb** of quantization.ipynb script that is Jupyter Notebook. The model itself is specified in code cell **№7:quantization.ipynb**. The model data quantization formats are specified in the previous cell **№6:quantization.ipynb** as:

```
model_kernel_quantizer = "quantized_bits(8, 0, alpha = 1)"
model_bias_quantizer = "quantized_bits(8, 0, alpha = 1)"
model_activation = "quantized_relu(8)"
```

The model is trained in the cell **№10:quantization.ipynb** with the following result:



The next cells perform converting the model to FPGA firmware with HLS4ML codesign. Overview of the HLS4ML codesign

The HLS4ML is a Python package that translates neural networks into FPGA firmware for inference with very low latency and power consumption on FPGAs. It achieves this by utilizing High-Level Synthesis (HLS) techniques, which convert high-level programming languages like C++ or Python into digital hardware description languages such as VHDL or Verilog. These hardware description languages can then be used to implement the models on FPGA or ASIC platforms. The use of HLS with HLS4ML significantly increases accessibility to a wide user community and greatly reduces firmware development time. By taking a neural network model as input, HLS4ML generates C/C++ code that can be transferred into FPGA firmware by using an HLS library. A significant outcome of the HLS4ML is the companion compiler, which translates ML models from popular open-source software packages like Keras and PyTorch into RTL (Register-Transfer Level) abstraction for FPGAs, using HLS tools. The HLS4ML tool enables to prototype ML algorithms quickly for both firmware feasibility and performance without requiring extensive Verilog/VHDL expertise. This significantly reduces the time required for algorithm development cycles while preserving engineering resources.

Converting the model to FPGA firmware using HLS4ML codesign

Now we will go through the steps to convert the model we trained to a resource-efficient but quite low-latency optimized FPGA firmware with hls4ml. First, we will evaluate its classification performance to make sure we haven't lost accuracy using the fixed-point data types. Then we will synthesize the model with Vivado HLS and check the metrics of latency and FPGA resource usage.

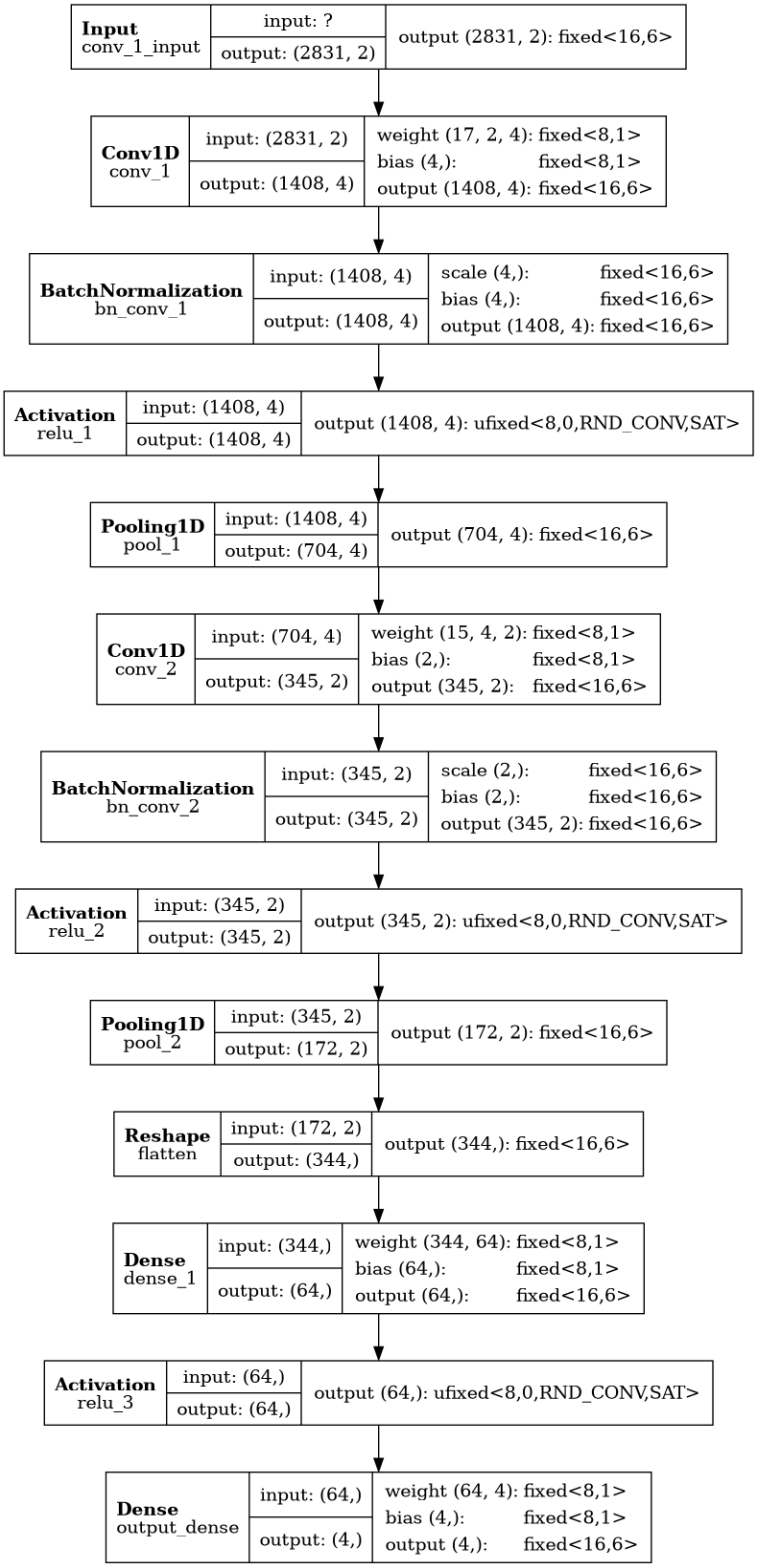
We choose 16-bit computation for all internals of the machine learning algorithms that hls4ml converts using **qmodel** as a source, which in turn is a pre-trained Tensorflow/QKeras model coded in Python. The **hls_model** is a Python representation of the hls4ml conversion with not only the computations fully defined, but also a C++ implementation that is ready for high-level synthesis. There are a few important items that configure hls4ml. Parameter **io_stream** hls4ml to build algorithms taking into account that the model contains convolutions The xc7z020clg400-1 is an FPGA chip of the PYNQ-Z2 board. The **/hls4ml_prj** parameter defines the

path to the Vivado project, in which the hls4ml scripts and the Vivado scripts will generate a full Vivado project and then synthesize the hardware description files with detailed reports.

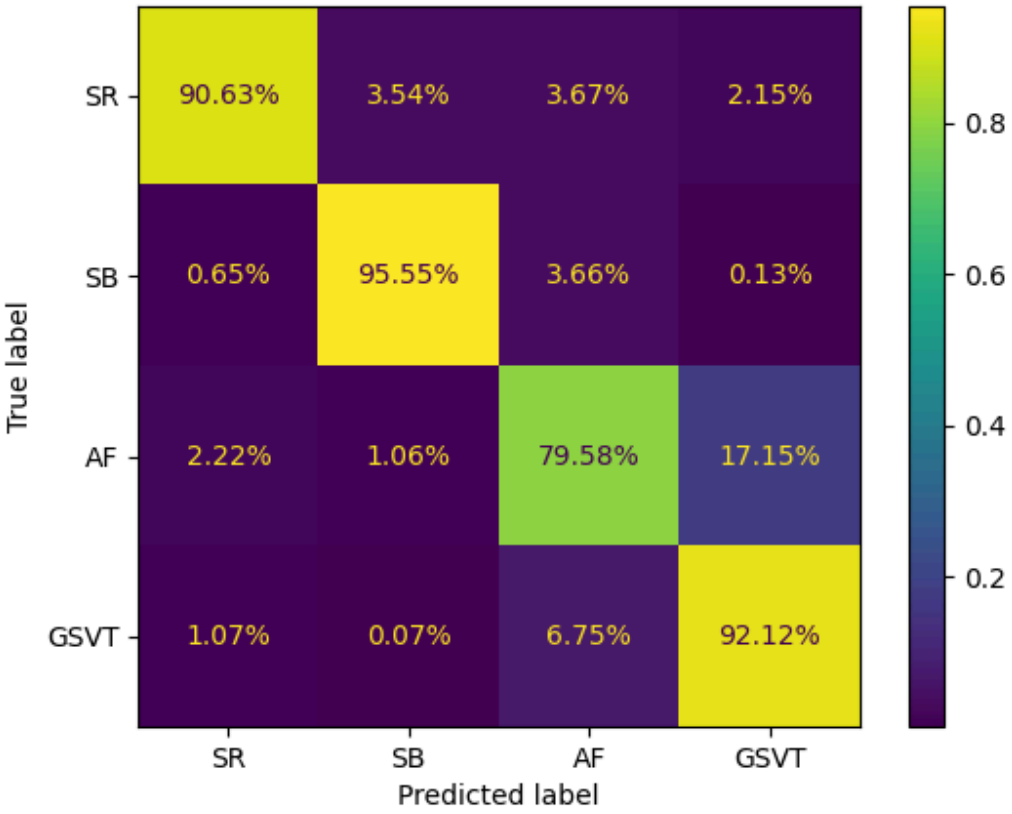
Cell **№13:quantization.ipynb** makes such conversion:

```
import hls4ml
import plotting

config = hls4ml.utils.config_from_keras_model(qmodel, granularity = 'name')

# Set the precision, reuse factor, and strategy for the full model
config['Model']['Precision'] = 'ap_fixed<16,6>'

for Layer in config['LayerName'].keys():
    config['LayerName'][Layer]['Strategy'] = 'Resource'
    config['LayerName'][Layer]['ReuseFactor'] = 256

print("-----------------------------------")
plotting.print_dict(config)
print("-----------------------------------")
hls_model = hls4ml.converters.convert_from_keras_model(
    qmodel,
    hls_config = config,
    io_type = 'io_stream',  # Must set this if using CNNs!
    output_dir = str(path_to_model_dir) + '/hls4ml_prj',
    part = 'xc7z020clg400-1',            # Just 280 BRAM_18K, 220 DSP48E, 106400 FF, 53200 LUT available
)
hls_model.compile()
```

After this conversion we can see the detailed structure of the ML model including precision of calculation on all layers and precision of all parameters. Also, we can check accuracy for bit-accurate emulation of the final solution that we expect on an FPGA.


Input
conv_1_input
input: ?
output: (2831, 2)
output (2831, 2): fixed<16,6>
Conv1D
conv_1
input: (2831, 2)
output: (1408, 4)
weight (17, 2, 4): fixed<8,1>
bias (4,): fixed<8,1>
output (1408, 4): fixed<16,6>
BatchNormalization
bn_conv_1
input: (1408, 4)
output: (1408, 4)
scale (4,): fixed<16,6>
bias (4,): fixed<16,6>
output (1408, 4): fixed<16,6>
Activation
relu_1
input: (1408, 4)
output: (1408, 4)
output (1408, 4): ufixed<8,0,RND_CONV,SAT>
Pooling1D
pool_1
input: (1408, 4)
output: (704, 4)
output (704, 4): fixed<16,6>
Conv1D
conv_2
input: (704, 4)
output: (345, 2)
weight (15, 4, 2): fixed<8,1>
bias (2,): fixed<8,1>
output (345, 2): fixed<16,6>
BatchNormalization
bn_conv_2
input: (345, 2)
output: (345, 2)
scale (2,): fixed<16,6>
bias (2,): fixed<16,6>
output (345, 2): fixed<16,6>
Activation
relu_2
input: (345, 2)
output: (345, 2)
output (345, 2): ufixed<8,0,RND_CONV,SAT>
Pooling1D
pool_2
input: (345, 2)
output: (172, 2)
output (172, 2): fixed<16,6>
Reshape
flatten
input: (172, 2)
output: (344,)
output (344,): fixed<16,6>
Dense
dense_1
input: (344,)
output: (64,)
weight (344, 64): fixed<8,1>
bias (64,): fixed<8,1>
output (64,): fixed<16,6>
Activation
relu_3
input: (64,)
output: (64,)
output (64,): ufixed<8,0,RND_CONV,SAT>
Dense
output_dense
input: (64,)
output: (4,)
weight (64, 4): fixed<8,1>
bias (4,): fixed<8,1>
output (4,): fixed<16,6>

The following confusion matrix is the result that we should achieve at the validation stage by running the test on a real PYNQ-Z2 board with FPGA:



We do not see sufficient degradation of accuracy compared to the previous matrix. It is very important to check! The second important message is that this result is exactly what we should get when testing the whole solution on a PYNQ-Z2 board with a real FPGA.

At cell **№19:quantization.ipynb** we built the Vivado project using the Vivado HLS backend.

The last cell **№20:quantization.ipynb** makes the report. Let's look at the part of the report that inform us about used resources of FPGA:

```
================================================================
== Utilization Estimates
================================================================
* Summary:
+-----------------+---------+-------+--------+-------+-----+
|       Name      | BRAM_18K| DSP48E|   FF   |  LUT  | URAM|
+-----------------+---------+-------+--------+-------+-----+
|DSP              |        -|      -|       -|      -|    -|
|Expression       |        -|      -|       0|      2|    -|
|FIFO             |       32|      -|    1796|   5044|    -|
|Instance         |       31|    135|   33228|  36912|    -|
|Memory           |        -|      -|       -|      -|    -|
|Multiplexer      |        -|      -|       -|      -|    -|
|Register         |        -|      -|       -|      -|    -|
+-----------------+---------+-------+--------+-------+-----+
|Total            |       63|    135|   35024|  41958|    0|
+-----------------+---------+-------+--------+-------+-----+
|Available        |      280|    220|  106400|  53200|    0|
+-----------------+---------+-------+--------+-------+-----+
|Utilization (%)  |       22|     61|      32|     78|    0|
+-----------------+---------+-------+--------+-------+-----+
```

This report shows the use of "clean" FPGA resources without the overhead of accompanying hardware, which we use for comfortable testing when creating hardware accelerators that are connected to Python code as an overlay, without deep diving into FPGA programming.

Preparing bitstream

Bitstream notebook **bitstream.ipynb** builds HW Accelerator wrapping our ML solution with all necessary hardware to have the possibility to load the solution into FPGA directly from Python code. That friendly way to reprogram FPGA is possible due to PYNQ technology.

The cell **№4:bitstream.ipynb** builds the PYNQ project in the dedicated folder **/hls4ml_prj_pynq**. The differences between the configuration that we did in **quantization.ipynb** are: **backend = 'VivadoAccelerator'** and **board='pynq-z2'**.

Of course, the confluence matrix in the output cell **№6:bitstream.ipynb** shows the same result as the matrix in the output cell **№21:quantization.ipynb**:

At the validation stage we'll transfer a few files from the earlier development and this notebook into the `~/broshure_test` directory on the PYNQ-Z2 board. The following commands in cell **№9:bitstream.ipynb** archive these files into `**work/model_q/hls4ml_prj_pynq/package.tar.gz**` that can be copied over to the PYNQ-Z2 board and then extracted:

```
!mkdir -p work/model_q/hls4ml_prj_pynq/package
!cp work/model_q/hls4ml_prj_pynq/myproject_vivado_accelerator/project_1.runs/impl_1/design_1_wrapper.bit work/model_q/hls4ml_prj_pynq/package/hls4ml_nn.bit
!cp work/model_q/hls4ml_prj_pynq/myproject_vivado_accelerator/project_1.srcs/sources_1/bd/design_1/hw_handoff/design_1.hwh work/model_q/hls4ml_prj_pynq/package/hls4ml_nn.hwh
!cp work/model_q/hls4ml_prj_pynq/axi_stream_driver.py work/model_q/hls4ml_prj_pynq/package/
!cp work/X_ecg_test.npy work/y_test.npy work/model_q/hls4ml_prj_pynq/package
!cp deployment.ipynb work/model_q/hls4ml_prj_pynq/package
!tar -czvf work/model_q/hls4ml_prj_pynq/package.tar.gz -C work/model_q/hls4ml_prj_pynq/package/ .
```

At the end of this notebook we can find the report that shows the FPGA resource utilization taking into account the overhead of the associated hardware that we use to connect to the Python code as an overlay:

```
+---------------------------+-------+-------+-----------+-------+
|          Site Type        |  Used | Fixed | Available | Util% |
+---------------------------+-------+-------+-----------+-------+
| Slice LUTs                | 25483 |     0 |     53200 | 47.90 |
|   LUT as Logic            | 25237 |     0 |     53200 | 47.44 |
|   LUT as Memory           |   246 |     0 |     17400 |  1.41 |
|     LUT as Distributed RAM |    22 |     0 |           |       |
|     LUT as Shift Register  |   224 |     0 |           |       |
| Slice Registers           | 36988 |     0 |    106400 | 34.76 |
|   Register as Flip Flop   | 36988 |     0 |    106400 | 34.76 |
|   Register as Latch       |     0 |     0 |    106400 |  0.00 |
| F7 Muxes                  |  1407 |     0 |     26600 |  5.29 |
| F8 Muxes                  |   396 |     0 |     13300 |  2.98 |
+---------------------------+-------+-------+-----------+-------+
```

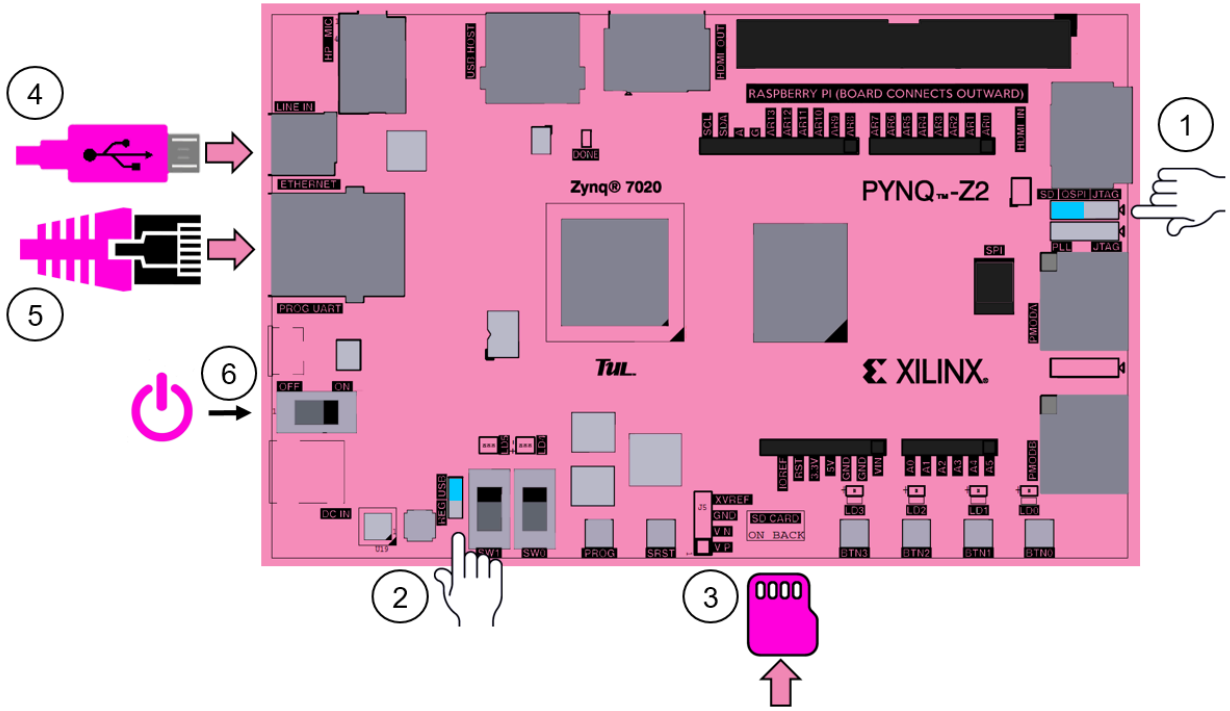
```
================================================================
== Utilization Estimates
================================================================
* Summary:
+-----------------+---------+-------+--------+-------+-----+
|       Name      | BRAM_18K| DSP48E|   FF   |  LUT  | URAM|
+-----------------+---------+-------+--------+-------+-----+
|DSP              |        -|      -|       -|      -|    -|
|Expression       |        -|      -|       0|     20|    -|
|FIFO             |        8|      -|     179|    552|    -|
|Instance         |       63|    135|   36630|  43416|    -|
|Memory           |        -|      -|       -|      -|    -|
|Multiplexer      |        -|      -|       -|     36|    -|
|Register         |        -|      -|       4|      -|    -|
+-----------------+---------+-------+--------+-------+-----+
|Total            |       71|    135|   36813|  44024|    0|
+-----------------+---------+-------+--------+-------+-----+
|Available        |      280|    220|  106400|  53200|    0|
+-----------------+---------+-------+--------+-------+-----+
|Utilization (%)  |       25|     61|      34|     82|    0|
```

```
+----------------+---------+-------+--------+-------+-----+
```

Real HW test

We test the solution on the PYNQ-Z2 board collecting ML inferences with the test dataset. In such a way we test ML algorithms, HLS4ML conversion, Vivado HLS synthesis, and operating the FPGA with developed bitstream. The heart of the PYNQ-Z2 board is a Xilinx system-on-a-chip that contains an embedded Linux machine with a Cortex A processor and an FPGA on a single die. We configure the FPGA as a hardware accelerator that we call from Python code as an overlay function. This function fully implements our ML algorithm and works as an external Neural Network processor for Python code. On a Linux system, Python code is only needed as a convenient interface to this network.

First, you'll need to follow the [setup instructions for the PYNQ-Z2 board](https://pynq.readthedocs.io/en/latest/getting_started/pynq_z2_setup.html). Typically, this includes connecting the board to your host via ethernet and setting up a static IP address for your host (192.168.2.1). The IP address for the PYNQ-Z2 is 192.168.2.99. The default username and password is **xilinx**.



Next, you'll transfer the following files from the earlier development into `~/broshure_test` directory on the PYNQ-Z2:

```
./X_ecg_test.npy
./y_test.npy
./hls4ml_nn.hwh
./deployment.ipynb
./hls4ml_nn.bit
./axi_stream_driver.py
```

You can just simply copy this archive `**work/model_q/hls4ml_prj_pynq/package.tar.gz**` to the PYNQ-Z2 and extract it. To copy it to the PYNQ-Z2 with the default settings, the command would be:

```
scp model_3/hls4ml_prj_pynq/package.tar.gz xilinx@192.168.2.99:~/broshure_test
```

Or, you can navigate your web browser to http://192.168.2.99. You will see the JupyterHub running on the PYNQ-Z2. And then copy the archive by Jupyter.

Open a terminal and extract the tarball:

```
cd ~/broshure_test
tar xvzf package.tar.gz
```

Or, just create new script and run Jupyter Notebook cell like:

`!tar xvzf package.tar.gz`

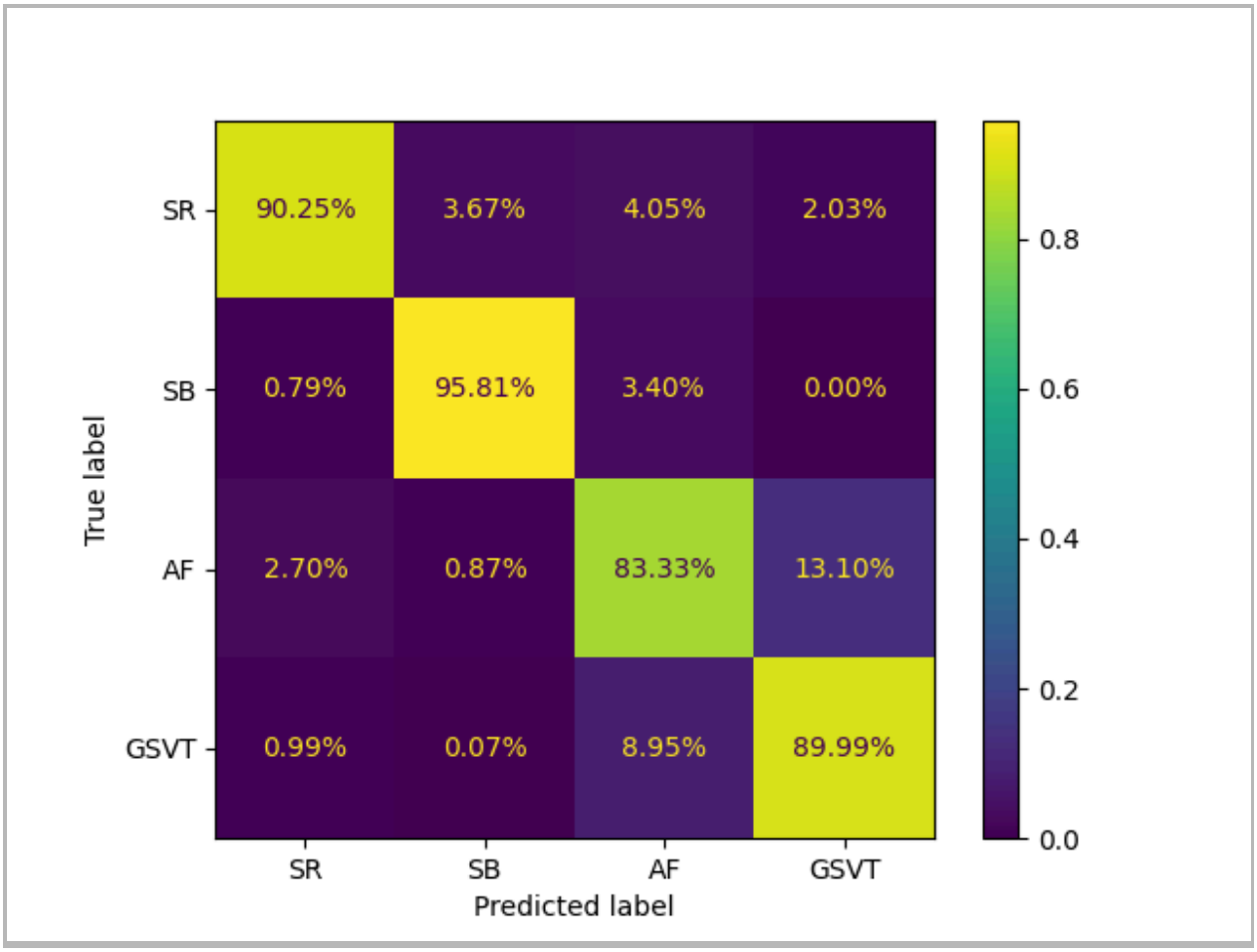
Open and run the **deployment.ipynb** script. Restart the kernel and then re-run the whole notebook. The Python code loads the bitstream into the FPGA, which takes our time. After a few seconds, the FPGA is ready and the actual hardware test begins. As a result, the **y_hw.npy** file with a collection of inferences will be created. The report tells us about 359 inferences per second, which is a good performance.

```
------------------------
(0, 4)
idx_start: 0, idx_end: 500
X_test_chunk shape:  (500, 2831, 2)
y_test_chunk shape:  (500, 4)
Classified 500 samples in 1.392252 seconds (359.1303873149401 inferences / s)
500 inferences have been got
idx_rest: 3500
========== len(y_hw):  500
```

Validation

We come to the final stage of development to validate the whole solution. We have to copy the **y_hw.npy** file from our PYNQ-Z2 board into the `**work/model_q**` folder of our instrumental platform (Ubuntu host).

We run **validation.ipynb** notebook and take the following confusion matrix:



Congratulations! We have got a validated ML solution based on FPGA with good accuracy and performance.

Vivado side details

Let's run Vivado to see the project as an FPGA-based solution with implemented hardware details. Just type the following in bash terminal:

```
/tools/Xilinx/Vivado/2019.2/bin/vivado
```

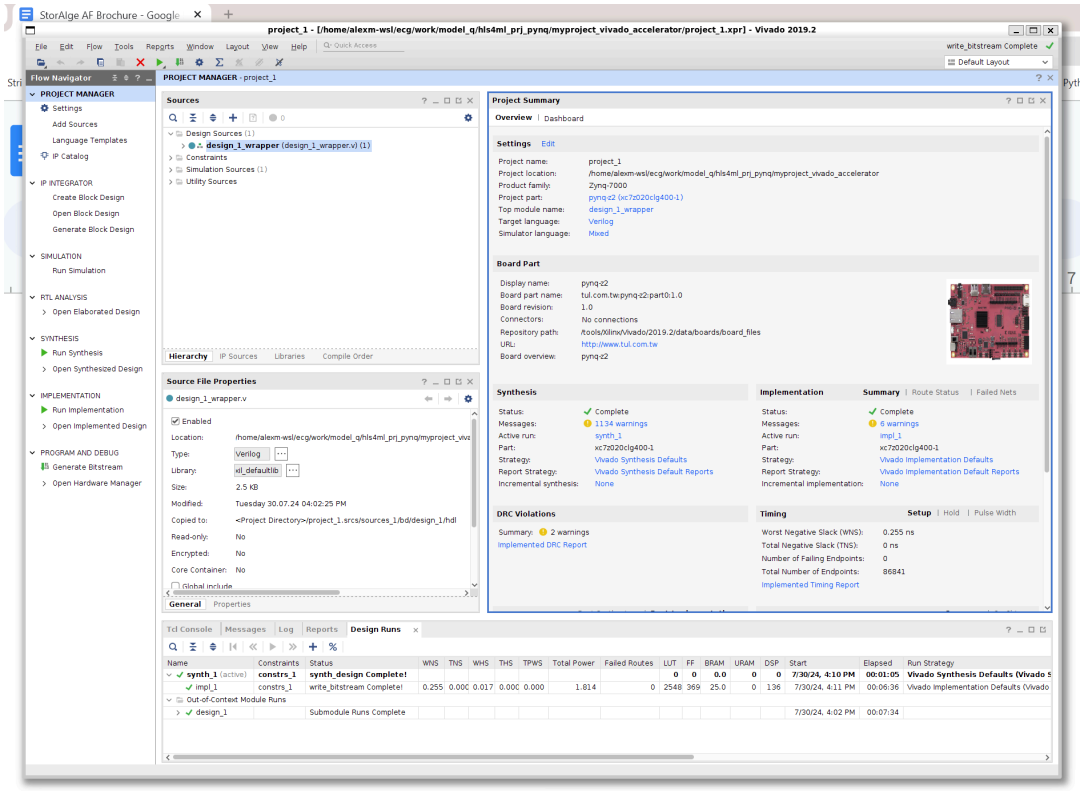
After launching the Vivado development environment, go to the top bar and open the project:

```
File -> Project -> Open -> ecg/work/model_q/hls4ml_prj_pynq/myproject_vivado_accelerator/project_1.xpr
```

Alternatively you can type the following in bash terminal:

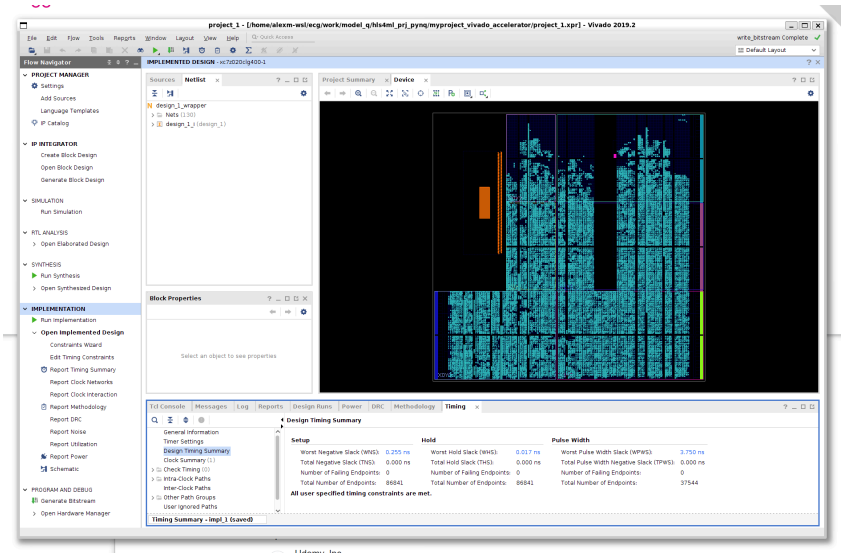
```
/tools/Xilinx/Vivado/2019.2/bin/vivado ~/ecg/work/model_q/hls4ml_prj_pynq/myproject_vivado_accelerator/project_1.xpr
```

You should get something like the following:



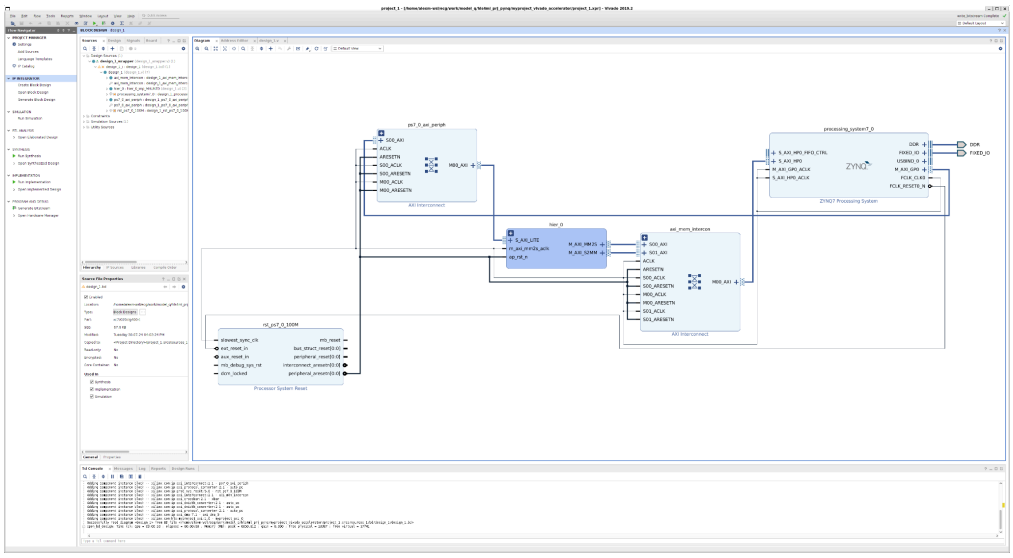
As you can see we have a full Vivado project with a successfully generated FPGA-bitstream that can be downloaded into PYNQ-Z2 as Python overlay. And the project is already completely built.

The HLS4ML scripts created the project under the hood without burdening us with this work, but we can study everything in the Vivado and Vivado HLS integrated environments as if we had created it. Let's look at the visualization of the floorplan of the FPGA-implemented accelerator by clicking "Open Implemented Design" in the Vivado GUI:



This picture gives an idea of the distribution of FPGA hardware resources for the implementation of the functions of the model itself and auxiliary functions that together implement the ML accelerator as a Python overlay.

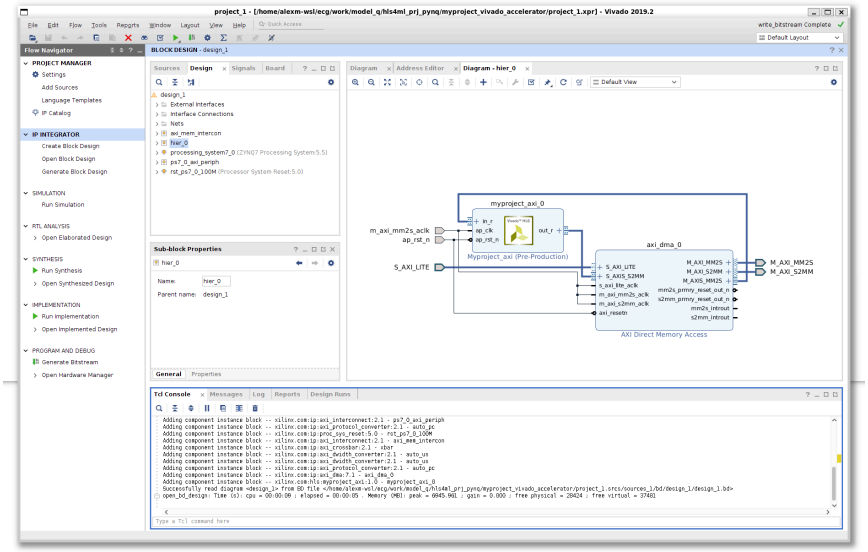
Obviously we are more interested in the functions of the model itself and all sorts of reports that we can extract from the Vivado integrated development environment to evaluate the effectiveness of the synthesis. Let's study the project in more detail.



On the top of the project we have wrapped block design that looks like the following:

You can see the Processing System with some blocks of Programmable Logic.

Ignoring the reset block, peripherals, and interconnection blocks, let's move on to the **hier_0** block, which implements ML as a hardware accelerator with Direct Memory Access:



Block **myproject_axi_0** is the hardware itself that implements our pre-trained ML model as an IP module synthesized by Vivado HLS. In Xilinx Vivado, an IP module (Intellectual Property module) refers to pre-designed, pre-verified blocks of logic or cores that can be used to accelerate the design process.

Let's look into **myproject_axi_0** block as a product of Vivado HLS. The log file is the key for understanding and is located at:
~\ecg\work\model_q\hls4ml_prj_pynq\vivado_hls.log

```
/tools/Xilinx/Vivado/2019.2/bin/vivado_hls
```

Appendix

Installation of development environment on Linux Ubuntu 22.04.03

Note! You should have at least 250GB of free space available to install all of the HLS4ML/Xilinx tools and 32GB - 64GB of RAM with at least 8 CPU cores you can dedicate to them.

Note2! Always keep your Linux system up-to-date with APT (Advanced Packaging Tool):

```
~$ sudo apt update && sudo apt -y upgrade && sudo apt -y autoremove
```

Note3! For a completely fresh/clean installation you probably need the following list of the required package dependencies installed:

```
$ sudo apt install make
$ sudo apt-get install git
```

Note4! Before downloading anything from Xilinx's website, there are some dependencies required by Vivado/Vitis/PetaLinux that need to be installed beforehand. Many of the libraries and tools PetaLinux has dependencies on in the host machine have 32-bit libraries, thus the 32-bit architecture i386 needs to be added to the host to compile them. This is done using the package management system, dpkg:

```
$ sudo dpkg --add-architecture i386
$ sudo apt install libc6-dev-i386 net-tools
$ sudo apt install libtinfo5
```

Install Google Chrome:

```
~$ wget https://dl.google.com/linux/direct/google-chrome-stable_current_amd64.deb
~$ sudo apt -y install ./google-chrome-stable_current_amd64.deb
~$ google-chrome --version
```

Installing Miniconda

```
$ mkdir -p ~/miniconda3
$ wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh -O ~/miniconda3/miniconda.sh

~$ bash ~/miniconda3/miniconda.sh -b -u -p ~/miniconda3
~$ source ~/.bashrc
~$ ~/miniconda3/bin/conda init bash
~$ rm -rf ~/miniconda3/miniconda.sh
~$ conda config --set auto_activate_base false
```

At the moment of composing the manual, there was an installed **conda** version:

```
~$ conda -V
conda 23.10.0
```

Installing HLS4ML

Running C simulation from Python requires a C++11-compatible compiler. On Linux, a GCC C++ compiler g++ is required:

```
$ sudo apt install g++
```

The hls4ml library depends on a number of Python packages and external tools for synthesis and simulation. These libraries will be installed together with HLS4ML.

Create conda environment that we are going to use with HLS4ML:

```
~$ conda create --name hls4ml python=3.11.5 pip jupyterlab=4.0.8
~$ conda activate hls4ml
```

Correct required version dependencies (actual on 2024 March 12)

```
~$ pip install tensorflow-model-optimization==0.7.5
```

Optional:

```
~$ pip install tensorflow==2.15.0
```

Install HLS4ML

```
~$ pip install hls4ml[profiling]==0.8.1
```

Install additional dependencies

```
~$ conda install pywavelets
~$ conda install openpyxl
~$ conda install -c conda-forge pydot
~$ conda install -c conda-forge graphviz
```

Installing Xilinx Vivado

Download Vivado HLx 2019.2: All OS installer Single-File and unpack it:
https://www.xilinx.com/member/forms/download/xef.html?filename=Xilinx_Vivado_2019.2_1106_2127.tar.gz

We should unpack a large archive. To see the progress of this unpacking install the **pv** utility:

```
$ sudo apt install pv
```

Then unpack:

```
$ pv Downloads/Xilinx_Vivado_2019.2_1106_2127.tar.gz | tar -xz
```

Or unpack it in blind mode:

```
~$ tar -xf Downloads/Xilinx_Vivado_2019.2_1106_2127.tar.gz
~$ sudo ./Downloads/Xilinx_Vivado_2019.2_1106_2127/xsetup
```

Press Next button to ignore the OS version warning in the welcome window and continue installation in the window with a proposition to get a newer version. We need a strict Xilinx Design Tool version 2019.2. Agree in the check boxes for Licence and Condition in the next window. Select Vivado HL WebPACK and leave all default product and installation options in the next window.

Run Vivado to test (and check/load license):

```
~$ /tools/Xilinx/Vivado/2019.2/bin/vivado
```

Exit Vivado.

Alternatively, we can install Vitis that includes both Vitis and Vivado :
Xilinx_Vitis_2019.2_1106_2127.tar.gz

Adding PYNQ-Z2 board

From the TUL manufacturer website: www.tulembedded.com/fpga/ProductsPYNQ-Z2.html load file: https://dpoauwgwqsy2x.cloudfront.net/Download/pynq-z2.zip

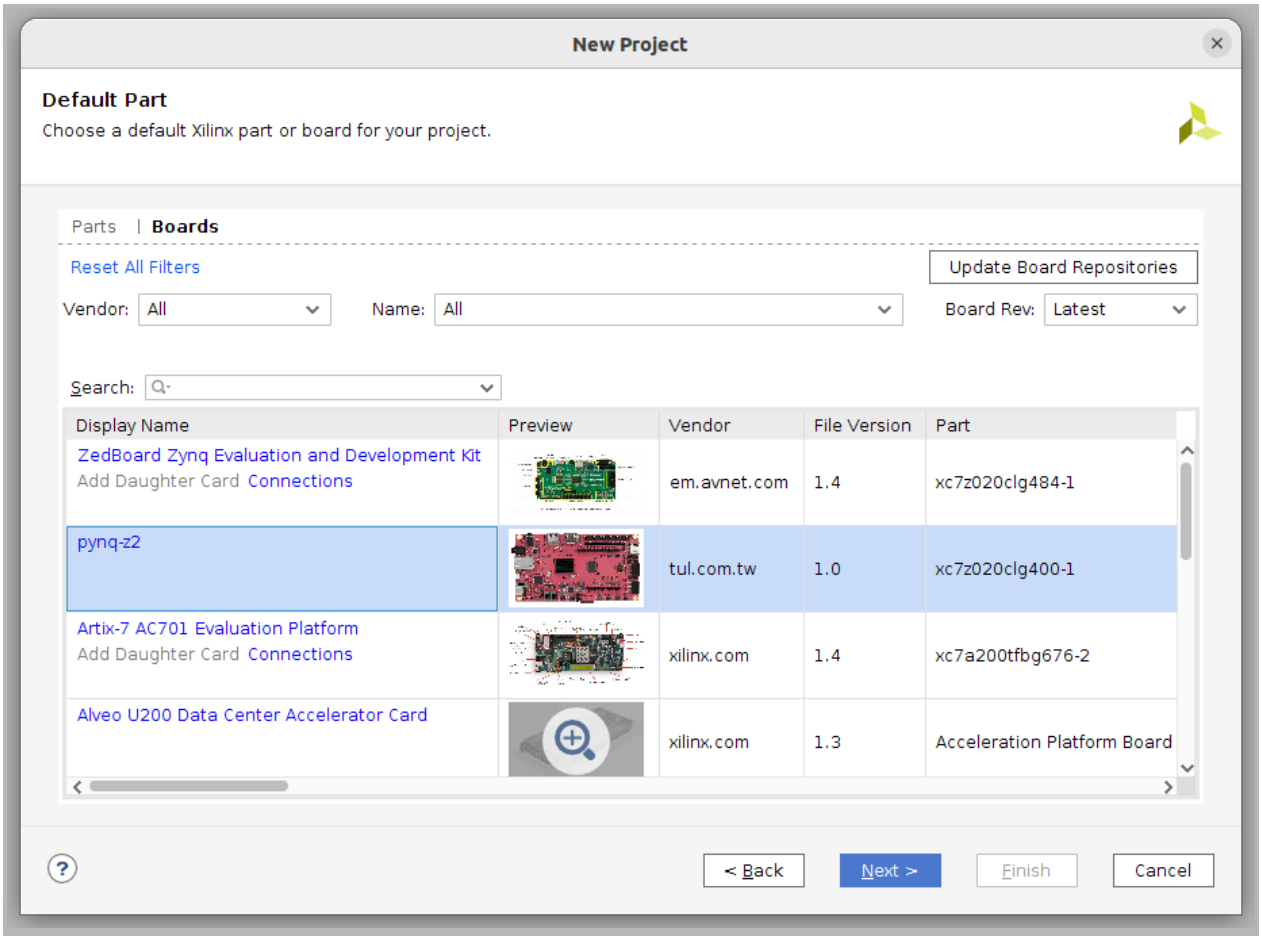
Install PYNQ-Z2 board files:

```
$ sudo cp -r <path to>pynq-z2 /tools/Xilinx/Vivado/2019.2/data/boards/board_files/
```

Once the installer has completed, it is good to double-check the installation by first running Vivado:

```
~$ source /tools/Xilinx/Vivado/2019.2/settings64.sh
~$ vivado
```

Test project with PYNQ-Z2 board



Close Vivado or open a new terminal window and launch Vivado HLS:

```
$ vivado_hls
```

The Vivado installer does not install the USB drivers required to recognize an FPGA on a Linux system, regardless of whether the option was checked or not. In order to install these drivers, navigate to the driver installer and run it:

```
$ cd /tools/Xilinx/Vivado/2019.2/data/xicom/cable_drivers/lin64/install_script/install_drivers
$ ./install_drivers
```

In order to use the USB drivers with a serial terminal, each user that will be using serial terminals must be added to the dialout group. Current user can be added to the dialout group with:

```
$ sudo adduser $USER dialout
```

Vivado HLS 2019.2 requires patching to overcome the Y2K22 problem. Download the patch zip file from the AMD-Xilinx support article for the issue here. Then unzip it into Xilinx installation directory /tools/Xilinx:

```
$ cd ~/Downloads
~/Downloads$ sudo unzip y2k22_patch-1.2.zip -d /tools/Xilinx/
```

Xilinx 2019.2 products patch process requires python version 2.7.5 or later. Also, If you are patching Xilinx 2019.x release or later you can use python installed with Xilinx tools. Install the libpython2.7-dev package then set the LD_LIBRARY_PATH variable as instructed in the patch README, and that was enough for the patch to run successfully:

```
~/Downloads$ sudo apt install libpython2.7-dev
~/Downloads$ export LD_LIBRARY_PATH=$PWD/Vivado/2019.2/tps/lnx64/python-2.7.5/lib/
```

Change directories into the Xilinx installation directory and run the patch with root privileges:

```
~/Downloads$ cd /tools/Xilinx/
/tools/Xilinx$ sudo Vivado/2019.2/tps/lnx64/python-2.7.5/bin/python2.7 y2k22_patch/patch.py
```

When successful, the output looks as such:

```
[2023-12-07] INFO: This script (version: 1.2) patches Xilinx Tools for HLS Y2k22 bug for the following release:
             2014.*, 2015.*, 2016.*, 2017.*, 2018.*, 2019.*, 2020.* and 2021.*
[2023-12-07] UPDATE: /tools/Xilinx/Vivado/2019.2/common/scripts
[2023-12-07] COPY: /tools/Xilinx/y2k22_patch/automg_patch_20220104.tcl  to /tools/Xilinx/Vivado/2019.2/common/scripts/automg_patch_20220104.tcl
```

Adding AMD/Xilinx license

You need to obtain a **Xilinx.lic** file to load it to Vivado. Of course, you need to get a license from AMD/Xilinx.

In Vivado IDE install license as the follows:

Help>Manage License>
Get License>Load License>Copy License
Manage License>View License Status

Close Vivado Licence Manager

[1] Cardiology Explained. Cardiology Explained. Ashley EA, Niebauer J. London: Remedica; 2004. Chapter 3 Conquering the ECG https://www.ncbi.nlm.nih.gov/books/NBK2214/

[2] Zheng, J., Zhang, J., Danioko, S. et al. A 12-lead electrocardiogram database for arrhythmia research covering more than 10,000 patients. Sci Data 7, 48 (2020). https://doi.org/10.1038/s41597-020-0386-x

[3] Zheng, Jianwei; Rakovski, Cyril; Danioko, Sidy; Zhang, Jianming; Yao, Hai; Hangyuan, Guo (2019). A 12-lead electrocardiogram database for arrhythmia research covering more than 10,000 patients. figshare. Collection. https://doi.org/10.6084/m9.figshare.c.4560497

[4]: Git repository for the project (private). https://git.strikersoft.com/edgeai/ecg

About the author

Alexandr Mruga - Technical Lead IoT & WSN.

Strikersoft. Isafjordsgatan 39B SE-164 40 Kista, Sweden.

Visit our website strikersoft.com