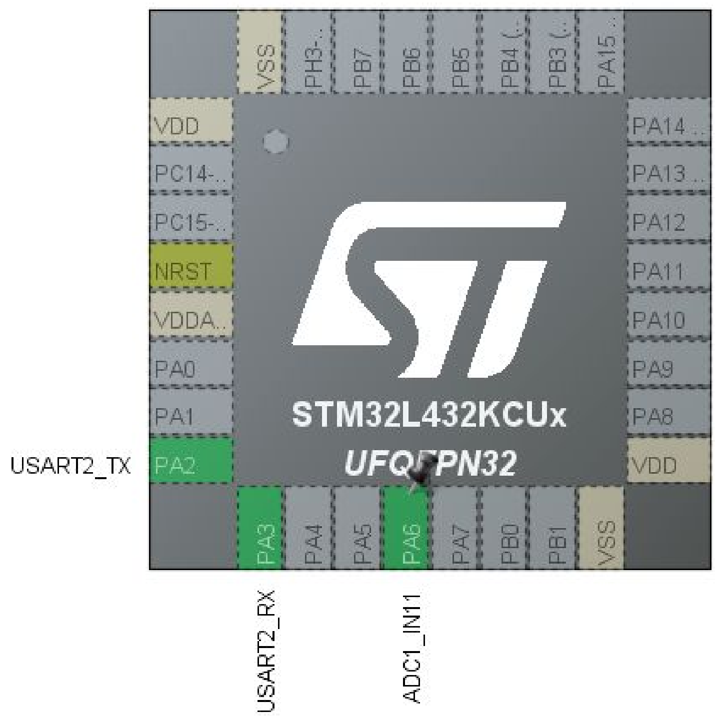
# Real-Time Heart Beat Anomaly Detection

Submitted to:
Dr. Mohamed Shalan

Submitted by:
Amro Ghoneim 900150924
Ismail ElSharkawy 900151594
Zeyad Ali 900152807

# Project Design and Description

The aim of our project is to detect anomalies in heartbeats constructed from an electrocardiogram (ECG) monitor, which is the recording of the electrical pulse/activity of one's heart on the MCU in real time. The recording of the heart beat will be displayed and the system will detect if there is an anomaly in the heart beat.




First the AD8232 sensor will be used to output the corresponding ADC signals. These signals will be received by the STM32L432 Nucleo microcontroller. Pin PA6 on the board was configured as shown in the following figure as an ADC input. This pin will receive the ADC signasl and the signals will then be used later for anomaly detection.

The final design of our system is as follows:

- The heart pulses will be detected using the AD8232 module.
- The readings detected from the AD8232 will be sent to the STM32 microcontroller on one of its ADC input pins.
- Needed Preprocessing tasks will be applied to the incoming readings

- The readings will then be given to the machine learning model deployed on the MCU for real time inference
- The readings/output will then be transmitted via UART and/or displayed on MCU(LEDs)

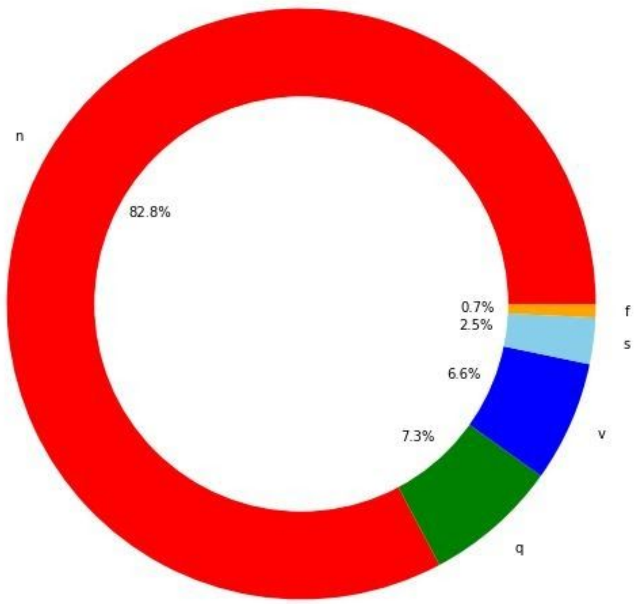
# Data set

For the purposes of this project the data set used was the ECG heartbeat Categorization Data set. It consists of two collections of heartbeat signals that are derived from two famous data sets in this domain, The MIT-BIH Arrhythmia Data set and the PTB Diagnostic Data set. For this project, we used the collection based on the MIT-BIH Data set as the number of samples are approximately 110K using 11 bit resolution and downsampled at 125 Hz. This is a good amount of samples to train a Neural Network properly. This Data set is comprised of 5 classes which are the following:

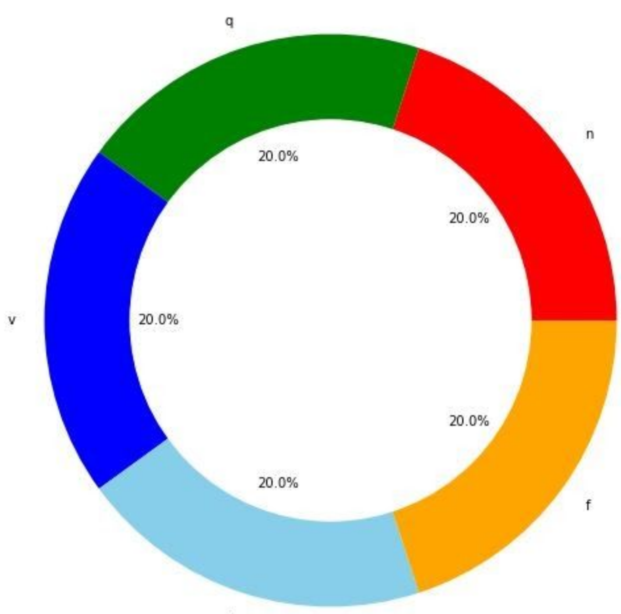
- 'N' : 0
- 'S' : 1
- 'V' : 2
- 'F' : 3
- 'Q' : 4

Where N represents the normal class and all other classes are abnormalities. The following screenshots show the class distributions in the training data before and after data augmentation:

*Before data augmentation:*




*After Data Augmentation:*



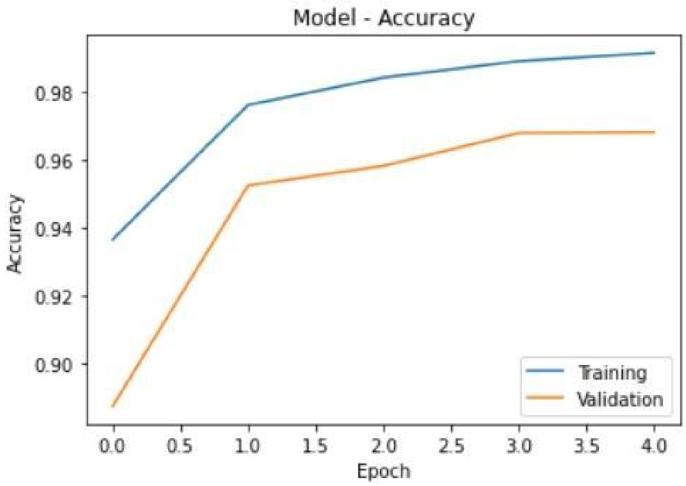
Where data augmentation restructured the data to make it more balanced, which in turn allows for better training.

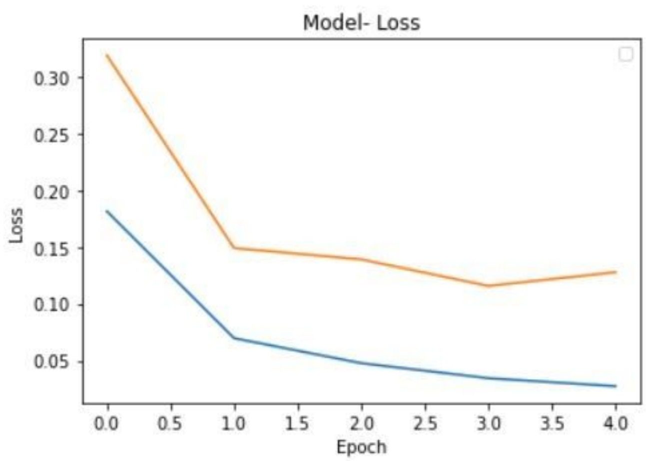
# Machine Learning Model:

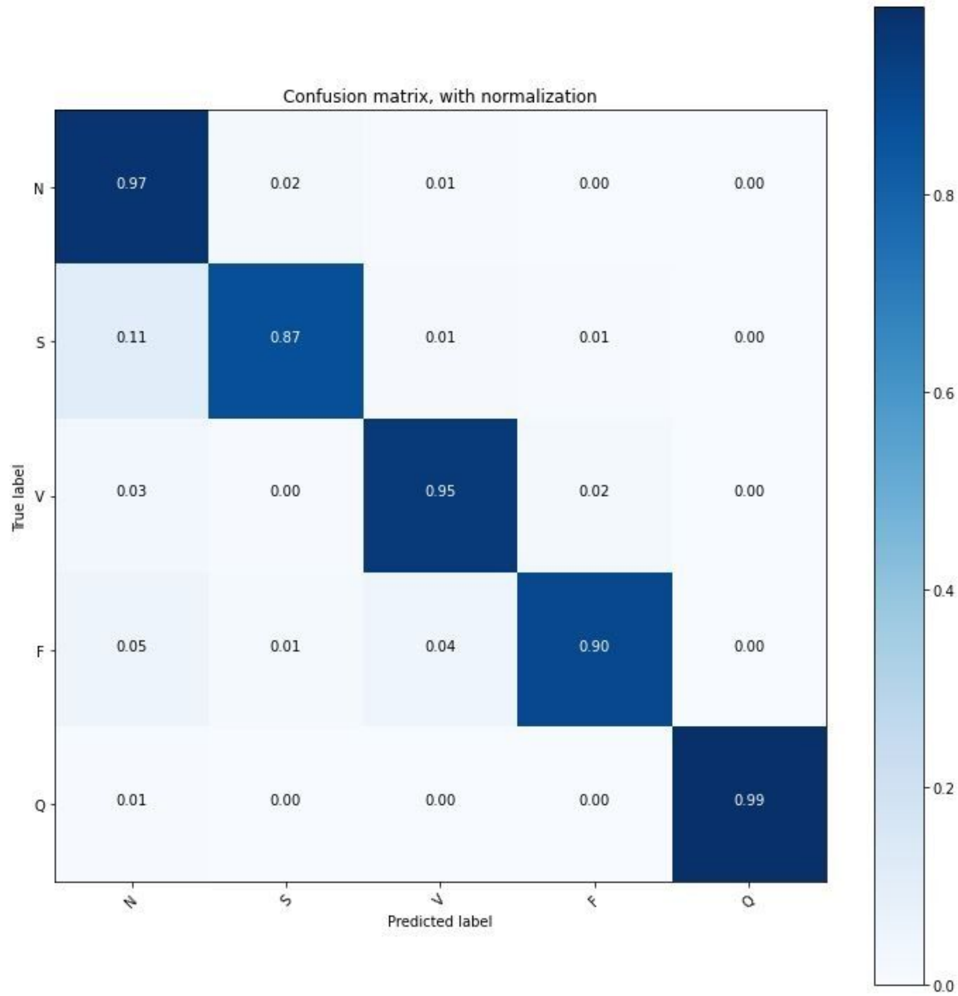
The initial machine learning model used was a 11 layer model (not including the output and input layers), consisting of 3 1-D CNN layers, 3 Max-Pooling layers and 3 Batch Normalization layers followed by 2 Dense layers (Keras Documentation provided in references) as seen in the screenshot below:

```
data_input=(X_train.shape[1],1)
inputs_cnn=Input(shape=(im_shape), name='data_input')
conv1_1=Convolution1D(64, (6), activation='relu', input_shape=im_shape)(inputs_cnn)
conv1_1=BatchNormalization()(conv1_1)
pool1=MaxPool1D(pool_size=(3), strides=(2), padding="same")(conv1_1)
conv2_1=Convolution1D(64, (3), activation='relu', input_shape=im_shape)(pool1)
conv2_1=BatchNormalization()(conv2_1)
pool2=MaxPool1D(pool_size=(2), strides=(2), padding="same")(conv2_1)
conv3_1=Convolution1D(64, (3), activation='relu', input_shape=im_shape)(pool2)
conv3_1=BatchNormalization()(conv3_1)
pool3=MaxPool1D(pool_size=(2), strides=(2), padding="same")(conv3_1)
flatten=Flatten()(pool3)
dense_end1 = Dense(64, activation='relu')(flatten)
dense_end2 = Dense(32, activation='relu')(dense_end1)
main_output = Dense(5, activation='softmax', name='main_output')(dense_end2)
```

The model, created using **python** and **keras** performed quite well after only 5 epochs of training as described by the screenshots below:


Model - Accuracy
Accuracy
Epoch
Training
Validation
0.98
0.96
0.94
0.92
0.90
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0

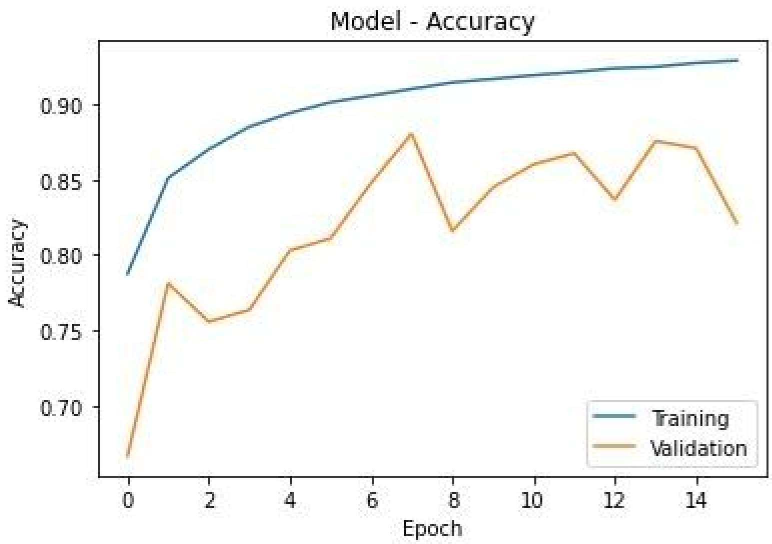

Model- Loss
Loss
Epoch
0.30
0.25
0.20
0.15
0.10
0.05
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0

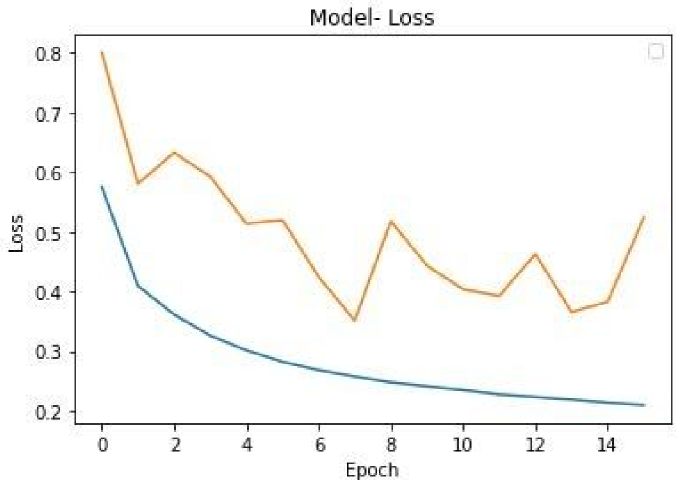


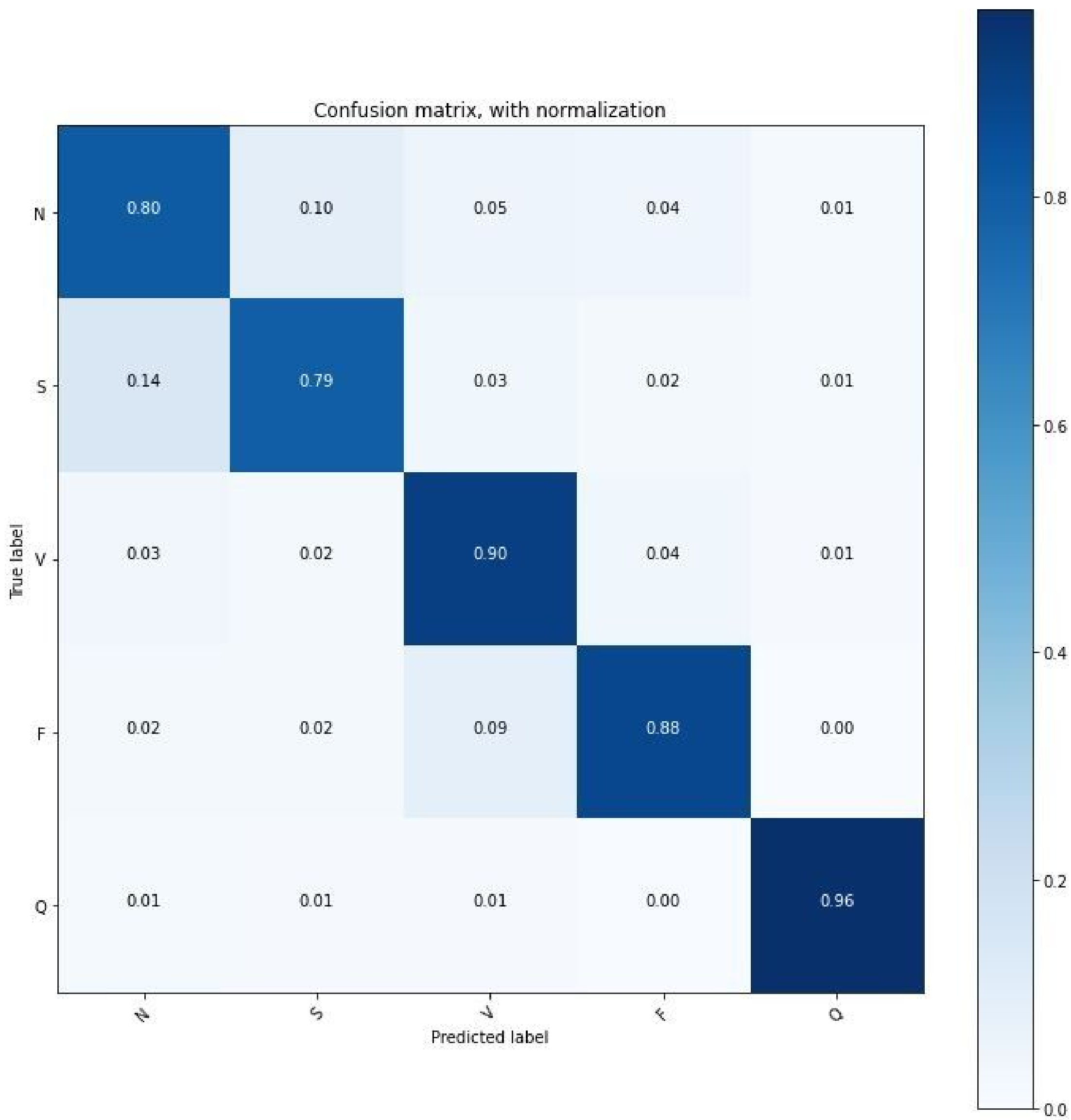

However, when using the model in tandem with STM32Cube.AI and the provided Nucleo MCU, it can be seen that the model, even after compression, is too large and complex to be used. Therefore; we decided to implement a new model that is much smaller and less complex that still proved to be quite accurate.

The new model architecture consists of only two dense layers (not including input and output layers). The screenshots below show the implementation of the machine learning model using **python** and **Keras** as well as the model's performance after 20 epochs of training:

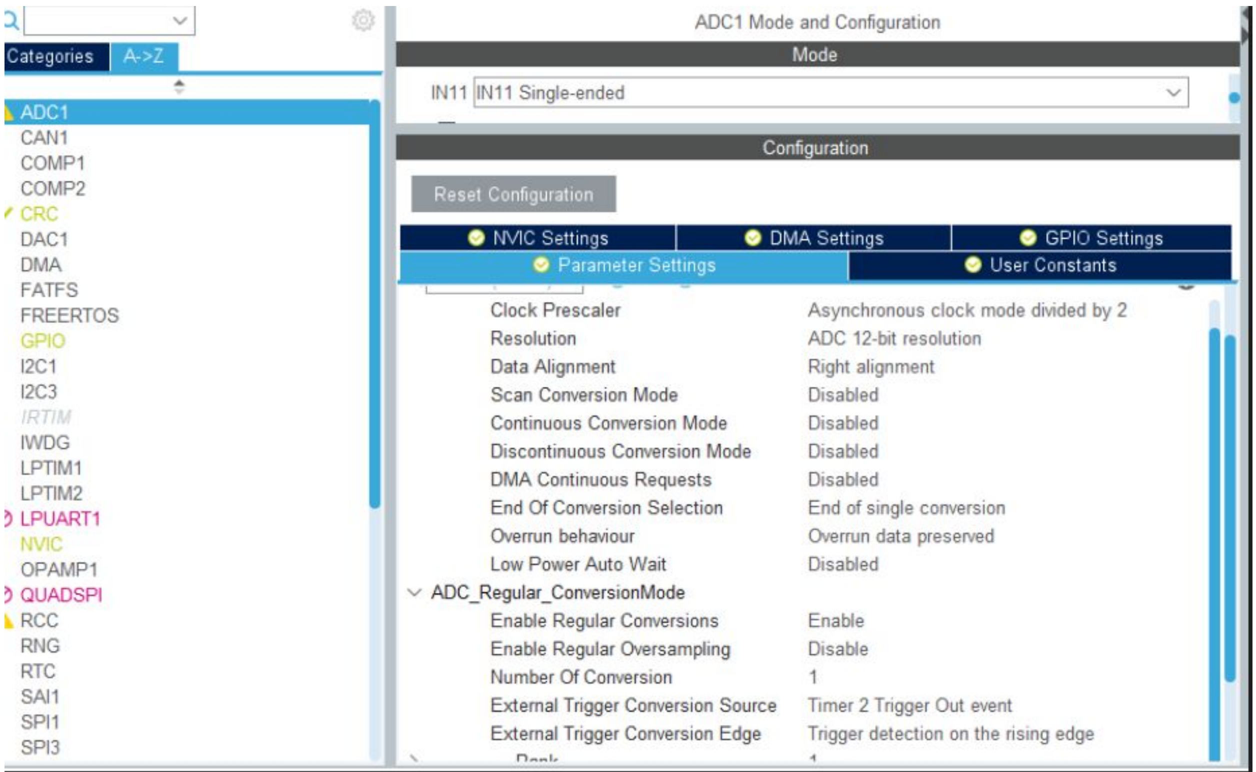
```
def network_1(X_train,y_train,X_test,y_test):

    data_input=(X_train.shape[1],1)
    inputs_dense = Input(shape = (data_input), name = 'data_input')
    dense_1 = Dense(50, activation='relu', input_shape=data_input)(inputs_dense)
    dense_2 = Dense(50, activation='relu')(dense_1)
    flat = Flatten()(dense_2)
    dense_out = Dense(5, activation='softmax', name='main_output')(flat)

    model = Model(inputs= inputs_dense, outputs=dense_out)

    model.compile(optimizer='adam', loss='categorical_crossentropy',metrics = ['accuracy'])
```



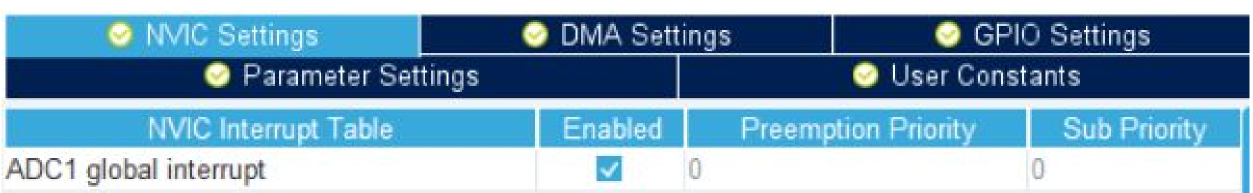

Model- Loss
Loss
Epoch
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0
2
4
6
8
10
12
14


Confusion matrix, with normalization
True label
Predicted label
N
S
V
F
Q
0.80
0.10
0.05
0.04
0.01
0.14
0.79
0.03
0.02
0.01
0.03
0.02
0.90
0.04
0.01
0.02
0.02
0.09
0.88
0.00
0.01
0.01
0.01
0.00
0.96
0.8
0.6
0.4
0.2
0.0

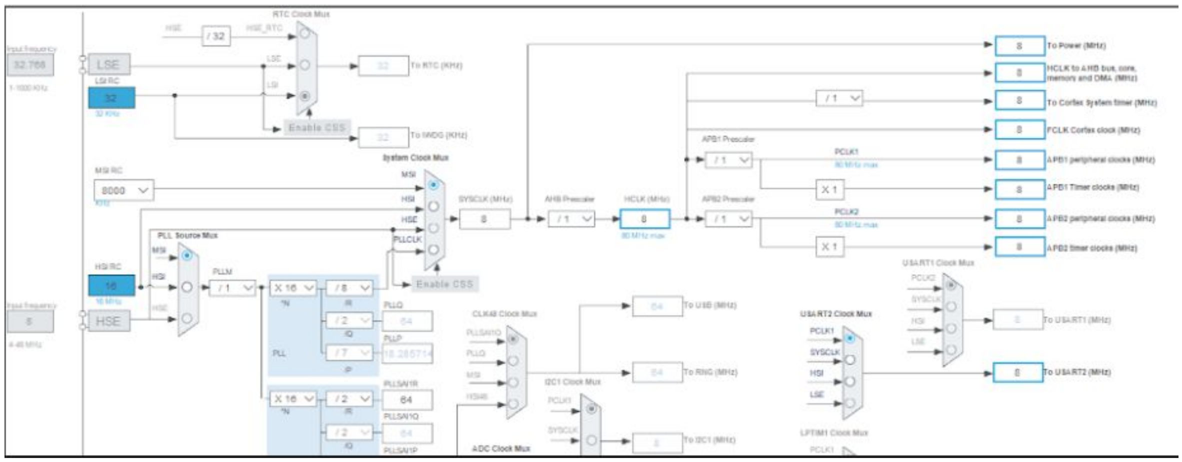
# CubeMX settings for ADC and Timer:



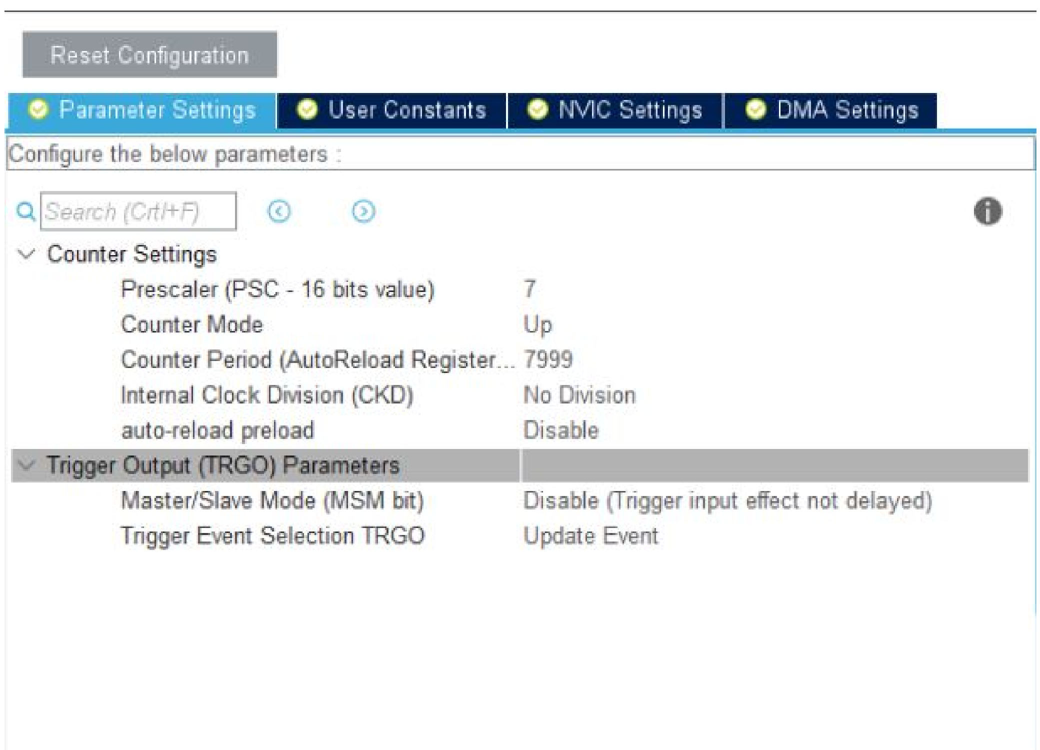

Here, we see the specifications of our used ADC. As we can see, we use a 12 bit resolution with enabled regular conversions alongside a Timer 2 trigger out event on the rising edge of our clock. (1 conversion per trigger)



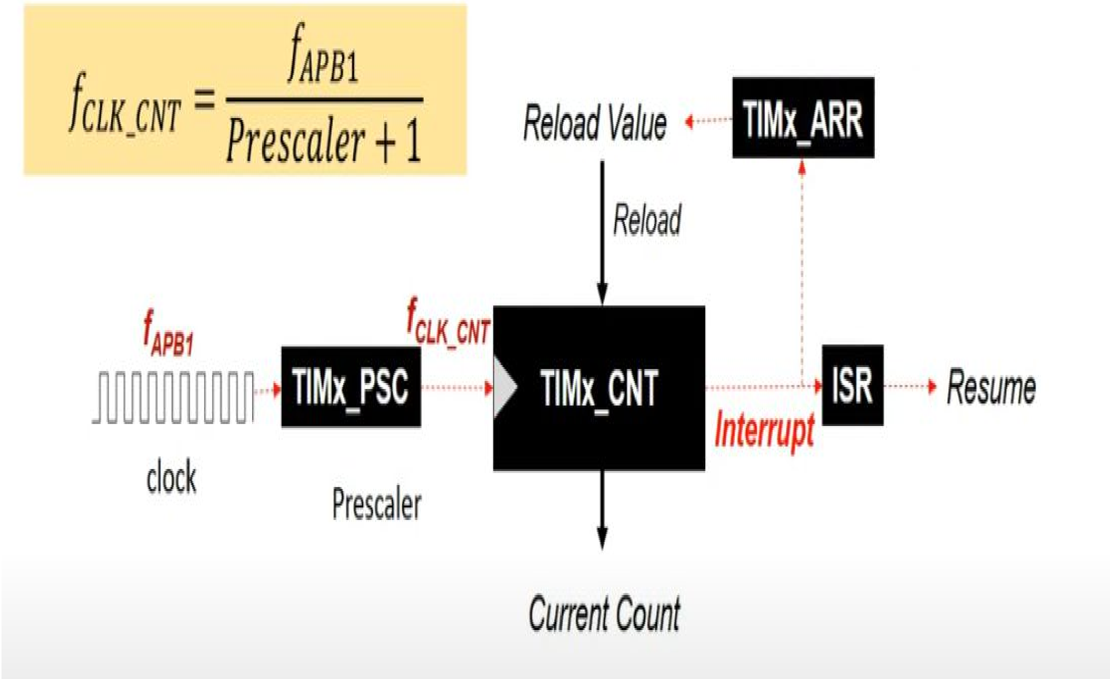

We ensure that the global interrupt is enabled so we can follow desired logic each time a conversion is made.



Here we see our clock tree, the one relevant to us is the APB1 Timer clock which ends up initially being 8MHz.



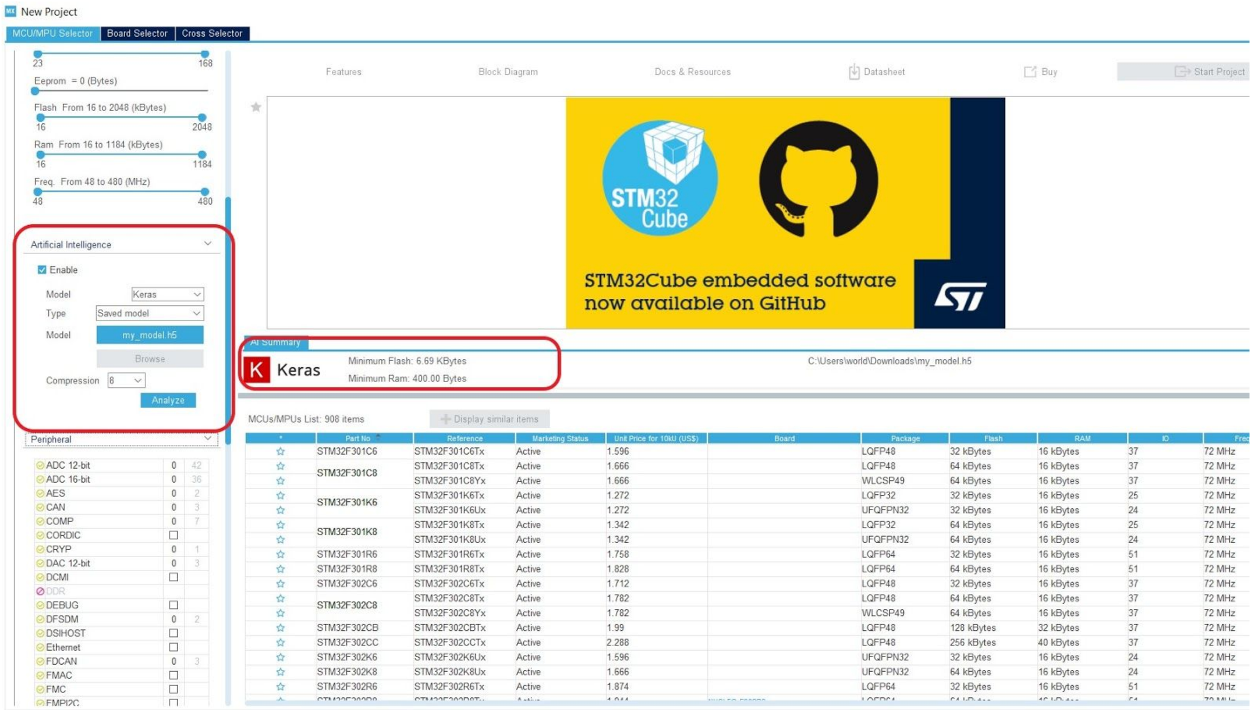

As seen in the figure above, our prescaler is set to 7, and out counter period goes up to 7999.



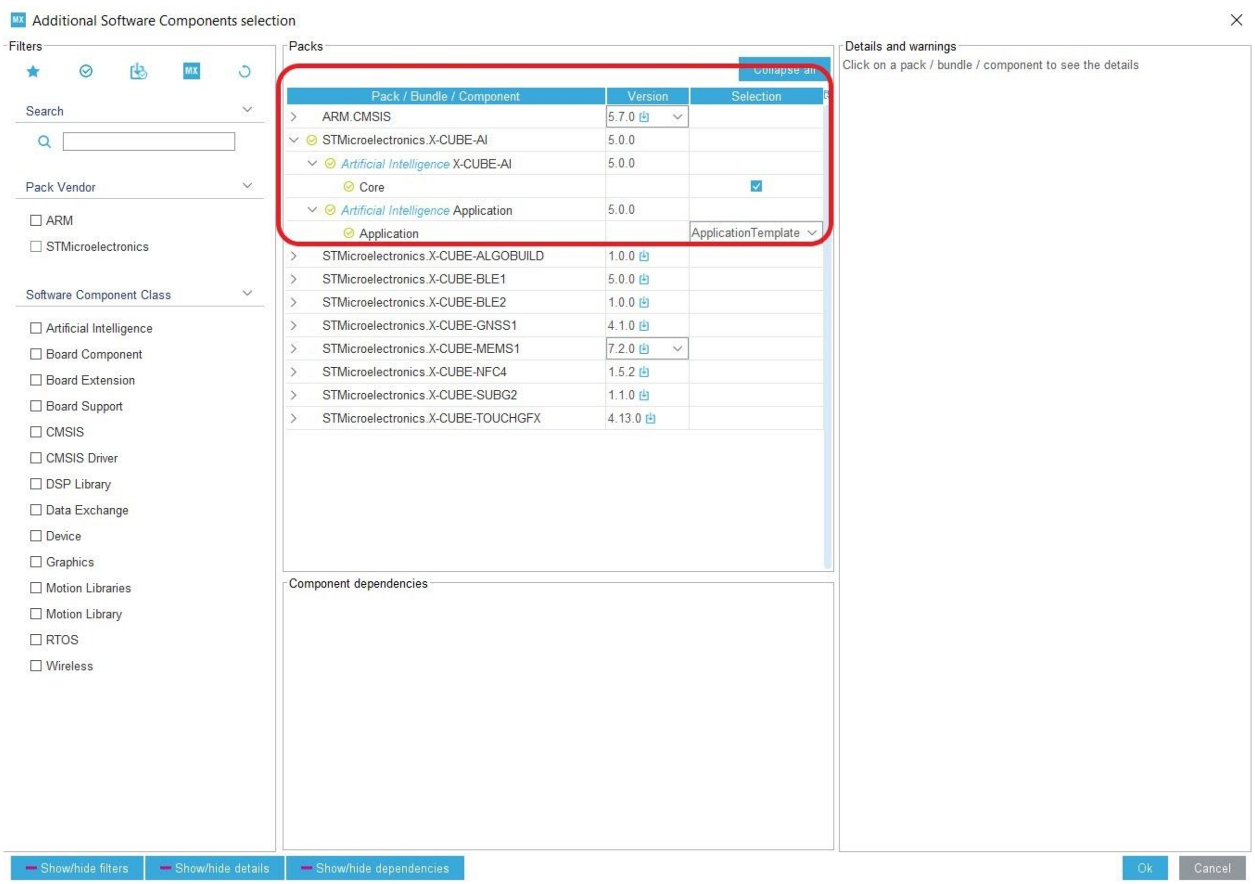

From this figure, we see how the sampling process took place according to previously mentioned assigned values. $f_{APB1}$ = 8MHz; Since the prescaler is set to 7, the $f_{CLK\_CNT}$ is 1MHz. That 1MHz used to calculate the interrupt frequency by being divided by the counter period value + 1 which results in a frequency of 125 Samples/Sec or 1 sample every 8ms.

# CubeMX settings FOR AI:

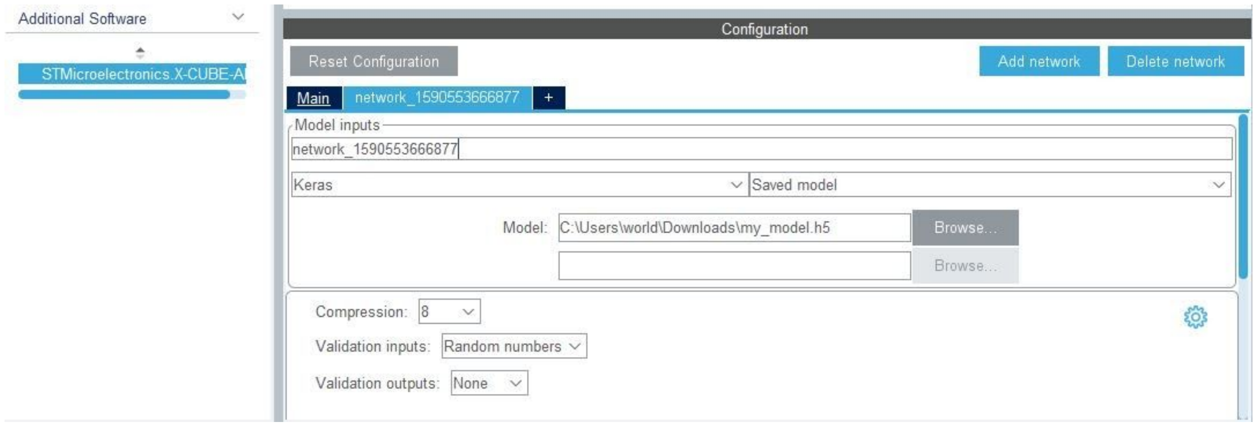
The screenshot below shows a summary of the model's size on chip as well as the model used with the compression rate specified. This initial step acts as a filter for MCUs where any MCU that cannot include AI or this specific model size will not be shown



After choosing the nucleo board, additional settings must be set as shown below:



The screenshots below show the network as included before with a random name. This name should be changed as it will reflect in the generated code. They also show some statistics and methods that can be used to see how the model fares after compression.





# Implementation(Keil/main):

```
  void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef *hadc)
{
  /* Prevent unused argument(s) compilation warning */
  UNUSED(hadc);
  /* NOTE : This function should not be modified. When the callback is needed,
            function HAL_ADC_ConvCpltCallback must be implemented in the user file.
   */
  HAL_ADC_PollForConversion(&hadc1, HAL_MAX_DELAY);
  adcval = HAL_ADC_GetValue(&hadc1); //getting the ADC value
  //adcval= ((float)adcval/4096.0) * 2048.0;
  if(count<187){  //only add the first 187 samples to the buffer in_data
        in_data[count] = adcval;
        if(in_data[count]<min)  //get the minimum value
          min=in_data[count];
        if(in_data[count]>max)  //get the maximum value
          max=in_data[count];
        count++;
      }
  sprintf(reading, "%hu", adcval); // copy the adc value to a reading buffer to display the readings if needed.
  HAL_UART_Transmit(&huart2, (uint8_t*)reading, sizeof(reading), HAL_MAX_DELAY); //transmit reading
  HAL_UART_Transmit(&huart2, (uint8_t*)newline, sizeof(newline), HAL_MAX_DELAY); //newline
}
```

In the beginning, we receive conversions from the ADC and store them in *adcval* once every 8ms so that we can reach a total of 125 samples per second (like reference paper). Our received converted values are then stored in the *in_data buffer* before we proceed to find maximum and minimum of each 187 values interval and incrementing the counter.

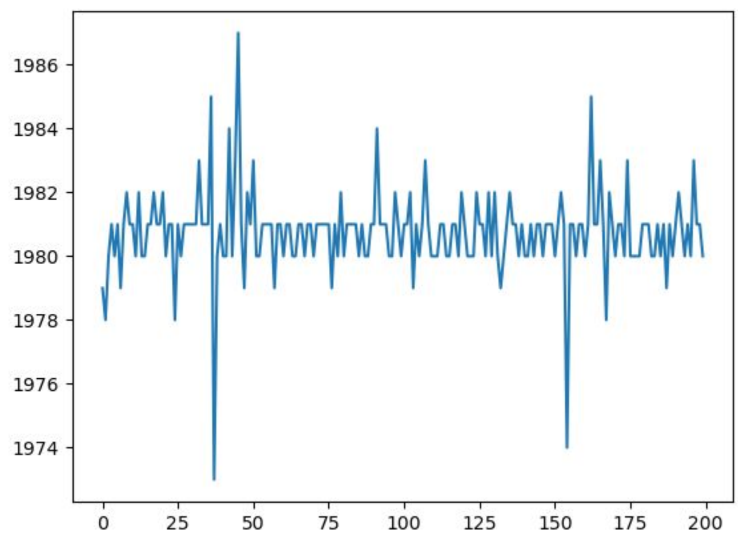
```
while (1)
{
  if(count>=187){  //if Counter exceeded 187 samples
    HAL_ADC_Stop_IT(&hadc1); //stop ADC interrupts so no more values are read
      for (int j=0; j<187;j++){
        norm_in_data[j]= (in_data[j]-min)/(max-min); //normailize the signals in in_data buffer
      }
      test = MX_X_CUBE_AI_Process(norm_in_data); //use the AI_Process to get the prediction
      if(!isnan(test))
      {sprintf(inference, "%f\r\n", test); //copy the probability to inference array
      //HAL_UART_Transmit(&huart2, (uint8_t*)inference, sizeof(inference), HAL_MAX_DELAY);
      if((inference[2]=='9') || (inference[0]=='1')) //check if the probability is >=0.9
        HAL_UART_Transmit(&huart2, (uint8_t*)normal, sizeof(normal), HAL_MAX_DELAY); //trannsmit normal
      else
        HAL_UART_Transmit(&huart2, (uint8_t*)abnormal, sizeof(abnormal), HAL_MAX_DELAY); //transmit abnormal
    }
      count =0; //reset counter
      min=999999; //reset minimum
      max=-1; //reset maximum
      HAL_ADC_Start_IT(&hadc1); //start ADC interrupt again.

    }
}
```

After the collection of 187 samples, interrupts are stopped and we proceed to normalize all collected samples before they are sent to the *MX_X_CUBE_AI_Process* function which returns the prediction. We make sure the returned value isn't null and transmit prediction as either *Normal* or *Abnormal* depending on a threshold of 0.9 plus. At the end of any 187 value interval, values get reset and the interrupts are started again.

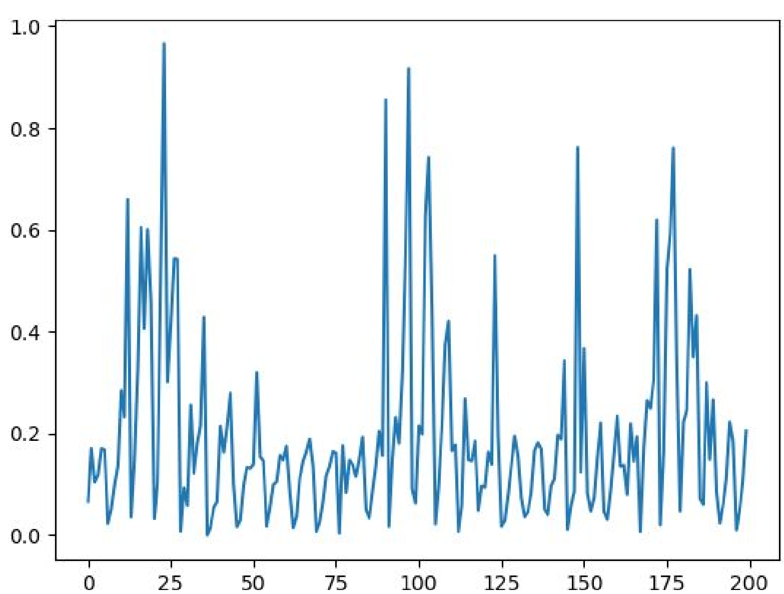
# **Results**:

The screenshots below show the output signal as well as the normalized output signal that is fed into the machine learning model for around 200 samples.

# ADC Signal:




# Normalized ADC Signal:



The screenshot below shows a sample of the output predictions in real time after feeding the normalized data into the model.

Predictions:

```
Normal Good Job
Normal Good Job
Normal Good Job
Normal Good Job
Normal Good Job
Normal Good Job
Normal Good Job
Normal Good Job
Normal Good Job
Normal Good Job
Normal Good Job
Abnormal
Normal Good Job
Normal Good Job
Normal Good Job
Normal Good Job
Normal Good Job
Normal Good Job
Normal Good Job
Abnormal
Normal Good Job
Normal Good Job
Normal Good Job
```

# Comments on the results:

We were able to gather the data from the ECG sensor in accordance with the required sampling rate (125 Hz) and the sampling resolution (11 bits). We were able to do some preprocessing tasks such as normalizing the data before feeding it into the machine learning model. The model performs the inference and returns the results to be displayed. The results achieved are not quite accurate, however, due to a number of limitations. The first one is the data coming from the sensor is quite noisy as can be seen in the screenshots provided earlier. In addition, memory limitations regarding the nucleo MCU and the Keil IDE led to the usage of a highly compressed model with somewhat degraded results as well as not being able to complete the required preprocessing tasks that are described by the Data set's paper (provided in the references). Therefore, further work regarding the preprocessing of the data is required to allow for better results.

# References:


1. https://www.physionet.org/physiobank/database/mitdb/
2. https://keras.io/api/
3. https://www.kaggle.com/shayanfazeli/heartbeat
4. https://www.st.com/resource/en/user_manual/dm00570145-getting-started-with-xcubeai-expansion-package-for-artificial-intelligence-ai-stmicroelectronics.pdf
5. https://www.st.com/content/st_com/en/stm32-ann.html