Real Time Anomaly Detection in Heart Activity

Amro Ghoneim
Ismail El Sharkawy
Zeyad Ali

Our Project

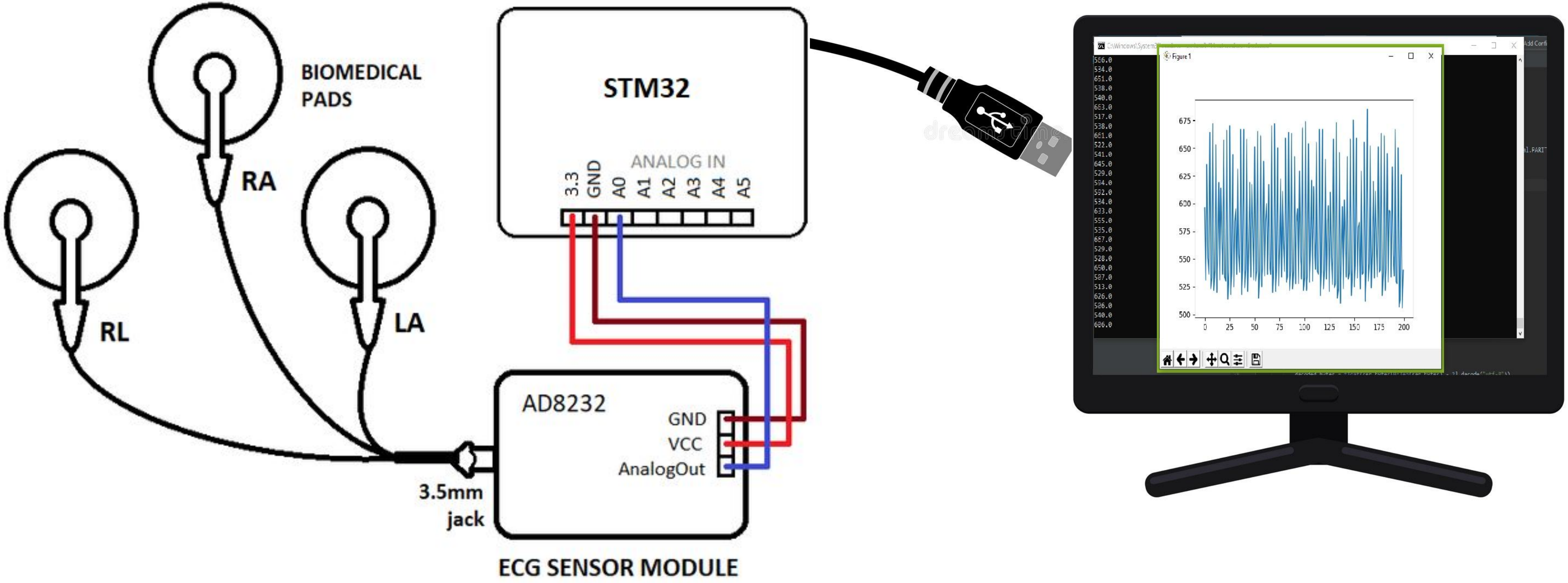
The aim of our project is detect anomalies in heartbeats constructed from an electrocardiogram (ECG) monitor, which is the recording of the electrical pulse/activity of one’s heart within the MCU. The recording of the heart beat will be displayed and the system will detect if there is an anomaly in the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

Design

- 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)

.

Design


BIOMEDICAL
PADS
RA
RL
LA
STM32
ANALOG IN
3.3
GND
A0
A1
A2
A3
A4
A5
AD8232
GND
VCC
AnalogOut
3.5mm
jack
ECG SENSOR MODULE
Figure 1

Implementation: Keil (Old)

The embedded code on the STM32 received the ADC input from the AD8232 and transmitted it through the UART. (Sampling rate unchanged)

```
while (1)
{
    // Test: Set GPIO pin high
  //HAL_GPIO_WritePin(GPIOB, GPIO_PIN_3, GPIO_PIN_SET);

  // Get ADC value
  HAL_ADC_Start(&hadc1);
  HAL_ADC_PollForConversion(&hadc1, HAL_MAX_DELAY);
  raw = HAL_ADC_GetValue(&hadc1);


  // Test: Set GPIO pin low
  //HAL_GPIO_WritePin(GPIOA, GPIO_PIN_10, GPIO_PIN_RESET);

  // Convert to string and print
  sprintf(reading, "%hu\r\n", raw);
  HAL_UART_Transmit(&huart2, (uint8_t*)reading, sizeof(reading)-5, HAL_MAX_DELAY);

  // Pretend we have to do something else for a while
  HAL_Delay(1);
}
```

Sampling

- Clock Configuration
  - Set the timer clock to 8MHz
- Timer
  - Enable TIM2 for counter
- ADC interrupt
  - Fire an interrupt every 8ms

125Hz sampling rate

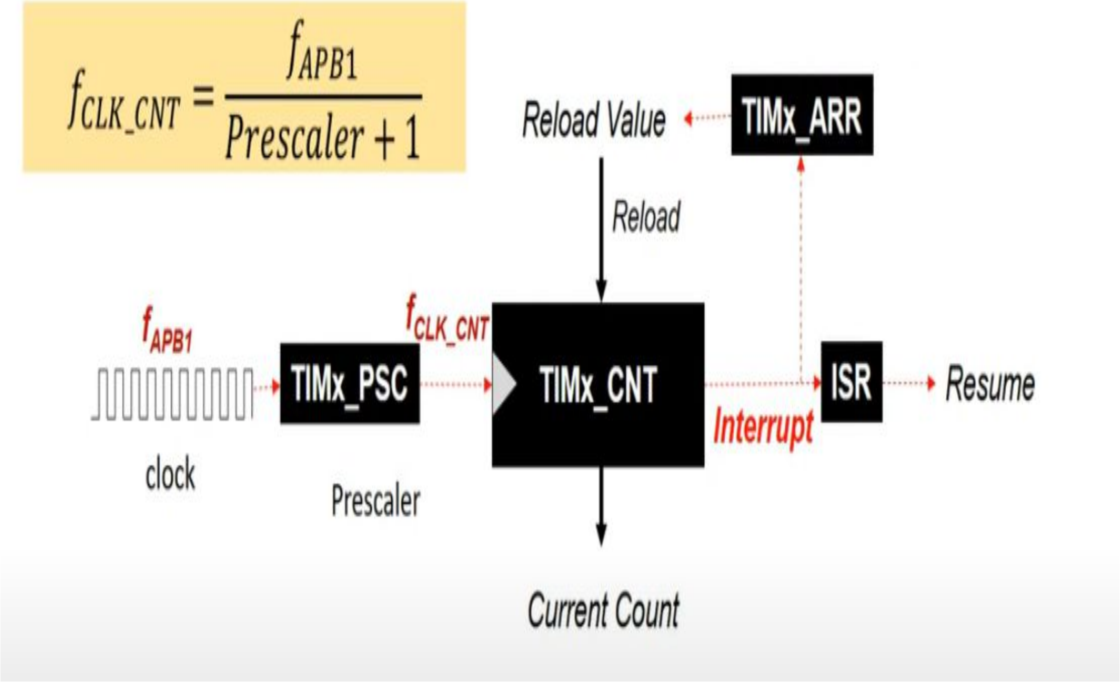
- f_{apb}= 8MHz

Prescaler= 7, clock counter would count up to 1MHz (slow)

- TIM_ARR set to value we want to count to: 7999

$$f_{Interrupt} = \frac{f_{CLK\_CNT}}{TIM\_ARR + 1}$$

$f_{interrupt}$= 1MHz/8000= 125 S/S



Implementation: Keil (New)

The embedded code on the STM32 received the ADC input by firing an ADC interrupt every 8 milliseconds (125SPS). The ADC value was read and stored each time an interrupt occurred.

```
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
}
```

Implementation: Keil (New)

After we read the ADC value and add it to the in_data buffer and then proceed to get the minimum and the maximum values for later normalization.

```
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
}
```

Implementation: Keil (New)

In the while loop the ADC interrupt is stopped when 187 samples are collected and then these samples are normalized.

```
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.

    }
}
```

Implementation: Python Application

Using python’s library Pyserial, the port of interest “COM3” was specified alongside the corresponding baudrate.

UART output is read line by line and formatted using the decode(”utf-8”) function after being parsed to output desired format.

Data is then gathered and stored in a csv file for later review.

For better visualization of errors and changes, data is then graphed in real time as shown in slide 8.

```python
ser = serial.Serial(port='COM3', baudrate=9600, bytesize=serial.EIGHTBITS, parity=serial.PARITY_NONE, timeout=2)
ser.flushInput()
plot_window = 200
y_var = np.array(np.zeros([plot_window]))

plt.ion()
fig, ax = plt.subplots()
line, = ax.plot(y_var)

if ser.isOpen():
    try:
        while 1:
            ser_bytes = ser.readline()
            decoded_bytes = float(ser_bytes[0:len(ser_bytes) - 2].decode("utf-8"))
            print(decoded_bytes)
            with open("test_data.csv", "a") as f:
                writer = csv.writer(f, delimiter=",")
                writer.writerow([time.time(), decoded_bytes])
                y_var = np.append(y_var, decoded_bytes)
                y_var = y_var[1:plot_window + 1]
                line.set_ydata(y_var)
                ax.relim()
                ax.autoscale_view()
                fig.canvas.draw()
                fig.canvas.flush_events()
    except Exception:
        print("error")
else:
```

Looks like you're using NumPy

About the Data set

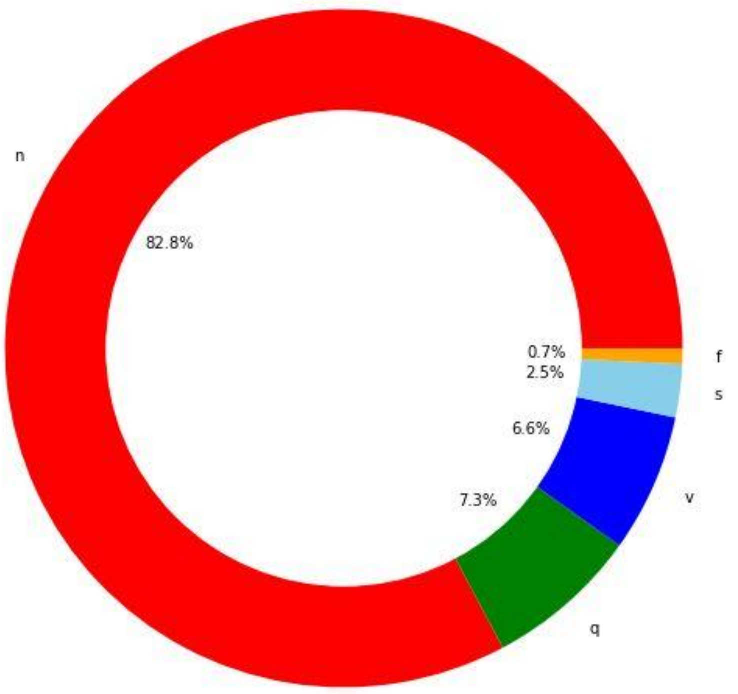
- Number of Samples: 109446
- Number of Categories: 5
- Sampling Frequency: 125Hz
- Data Source: Physionet's MIT-BIH Arrhythmia Dataset
  - Preprocessing steps described in this paper to create the data set
- Classes: ['N': 0, 'S': 1, 'V': 2, 'F': 3, 'Q': 4]
  - N is the normal class and the rest are different anomalies

Data set Class Distribution

Initial Class distribution of the data set shows highly imbalanced data !

This will affect model performance for detecting anomalies

```
0    72471
4     6431
2     5788
1     2223
3      641
Name: 187, dtype: int64
```

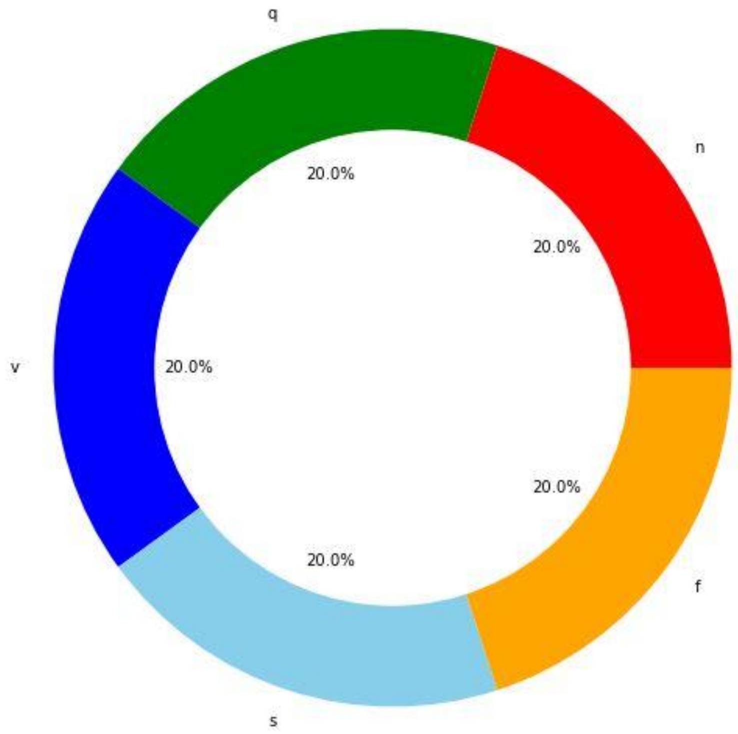


Data set resampling

Resampled the data set to have a balanced class distribution

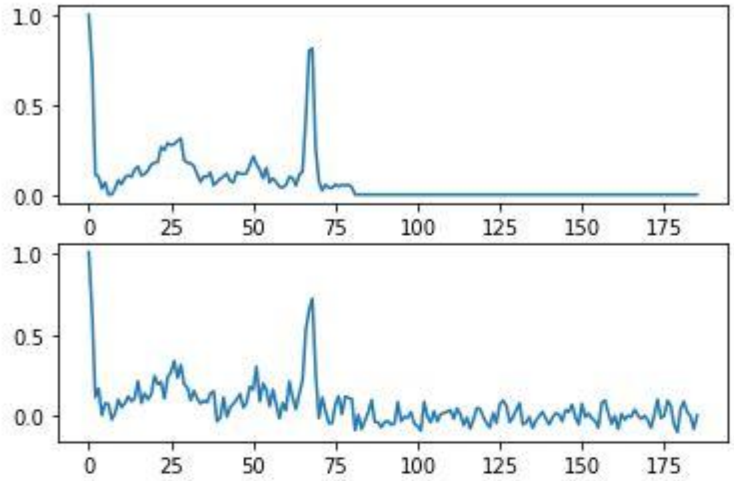
Each class now has 20,000 samples each

```
4    20000
3    20000
2    20000
1    20000
0    20000
Name: 187, dtype: int64
```



Other preprocessing techniques

Added some noise to the data to make it more generalized



Keras

- Open-source neural-network library written in Python
- Can run on top of Tensorflow, Theano and other machine learning frameworks
- User Friendly, Modular and Extensible

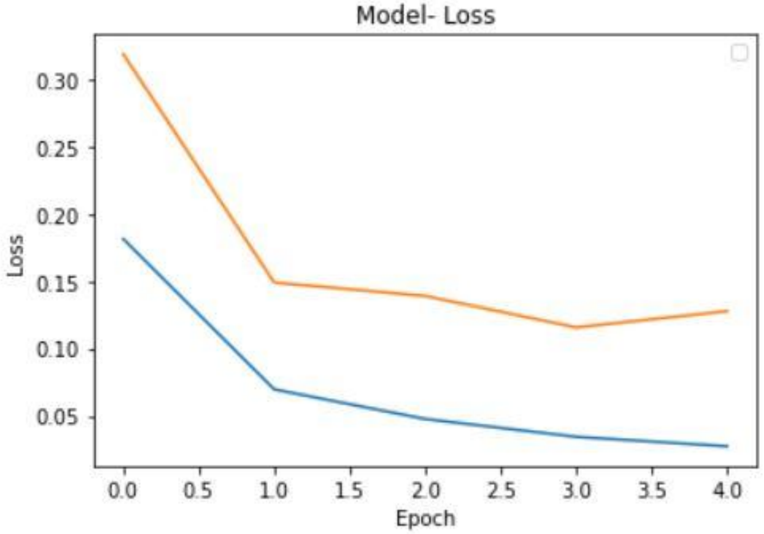
Model Architecture - initial

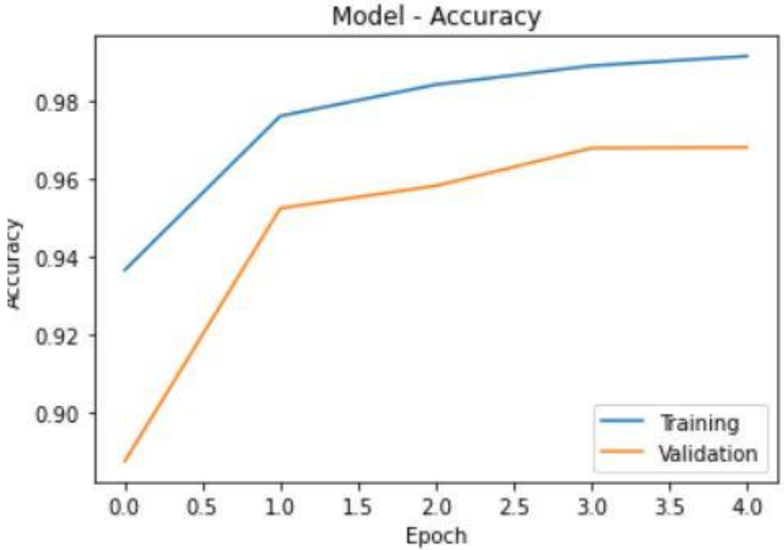
```
data_input=(X_train.shape[1],1)
inputs_cnn=Input(shape=(im_shape), name='data_input')
conv1_1=Convolution1D(64, (6), activation='relu', input_shape=im_shape)(inputs_cnn)
conv1_1=BatchNormalization()(conv1_1)
pool1=MaxPool1D(pool_size=(3), strides=(2), padding="same")(conv1_1)
conv2_1=Convolution1D(64, (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)
```

```
model = Model(inputs= inputs_cnn, outputs=main_output)
model.compile(optimizer='adam', loss='categorical_crossentropy',metrics = ['accuracy'])


callbacks = [EarlyStopping(monitor='val_loss', patience=8),
        ModelCheckpoint(filepath='CNN_model.h5', monitor='val_loss', save_best_only=True)]
```

Results - initial

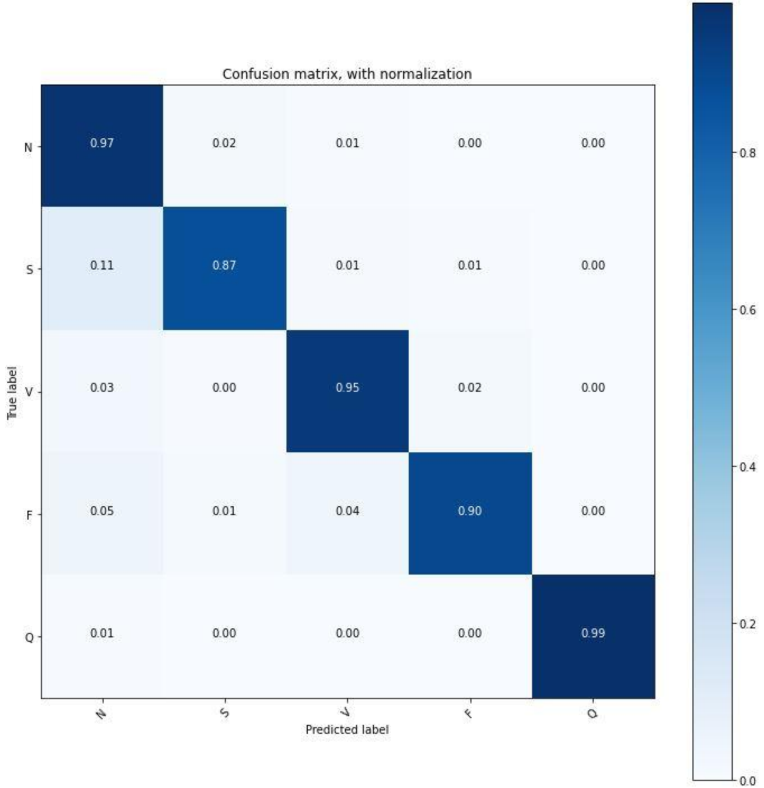
The graphs below show the training **accuracy** and **validation loss** over **5** epochs





Results - initial

- **The confusion matrix shows that the model performs on the test set with slight decrease in results for the two classes S and F**



STM32Cube.AI

- Interoperable with popular deep learning training tools (Keras)
- Compatible with many IDEs and compilers (Keil)
- Sensor and RTOS agnostic
- Allows multiple Artificial Neural Networks to be run on a single STM32 MCU
- Full support for ultra-low-power STM32 MCUs
- Allows for model compression (x4 & x8)

STM32Cube.AI

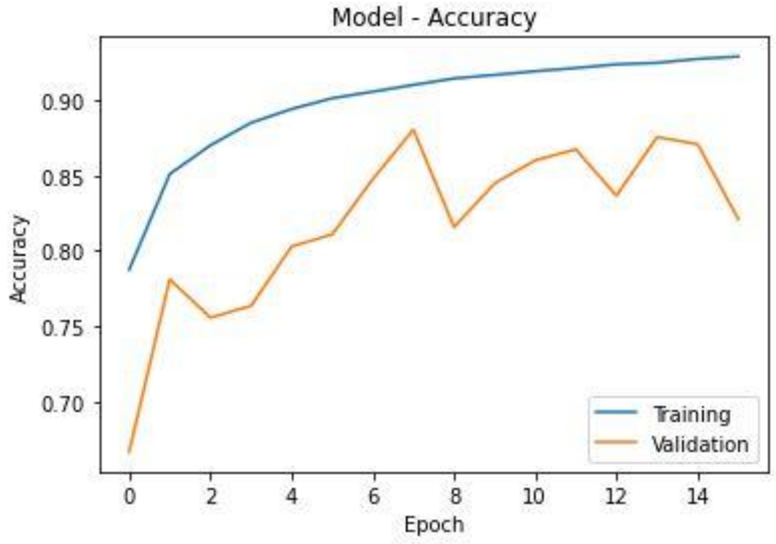
- Initial model too big for nucleo board provided, even after x8 compression
- New model implementation
    a. consumes around 6.7KB of flash after x8 compression with somewhat degraded yet comparable results

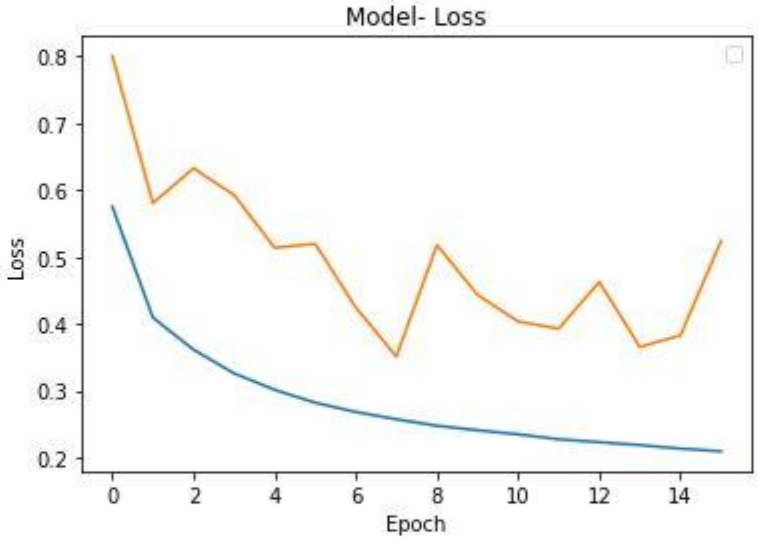
Model Architecture - final

```
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'])
```

Results - final

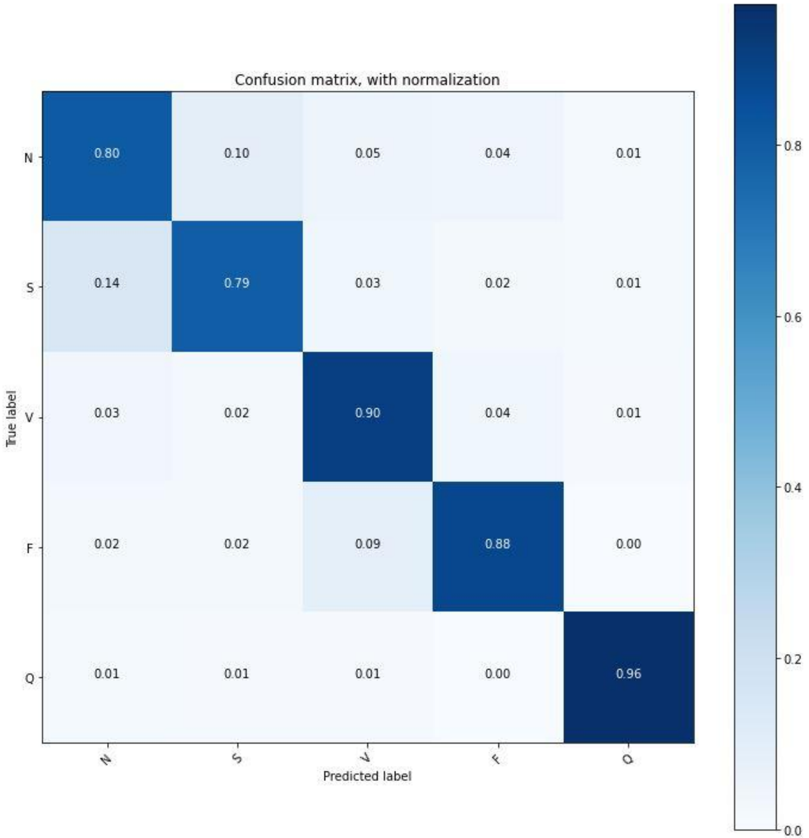
The graphs below show the training **accuracy** and **validation loss** over **20** epochs with overall accuracy of around 92%





Results - final

- **The confusion matrix shows that the new model is somewhat close to the initial model’s results**



STM32Cube.AI APIs

- **STM32Cube.AI provides an application template for its APIs for easier deployment**
  - MX_X_CUBE_AI_Init() function used to create and initialize the Neural Network

```
void MX_X_CUBE_AI_Init(void)
{
    /* USER CODE BEGIN 0 */
    /* Activation/working buffer is allocated as a static memory chunk
     * (bss section) */
    AI_ALIGNED(4)
    static ai_u8 activations[AI_NETWORK_DATA_ACTIVATIONS_SIZE];

    aiInit(activations);
    /* USER CODE END 0 */
}
```

STM32Cube.AI APIs

- **STM32Cube.AI provides an application template for its APIs for easier deployment**
    - MX_X_CUBE_AI_Process() function used to run the model and get the results

```
float MX_X_CUBE_AI_Process(const float* in_data)
{
    /* USER CODE BEGIN 1 */
    int res;

    static float out_data[5];

     /* Perform the inference */
    res = aiRun(in_data, out_data);

    return out_data[0];

    /* USER CODE END 1 */
}
```

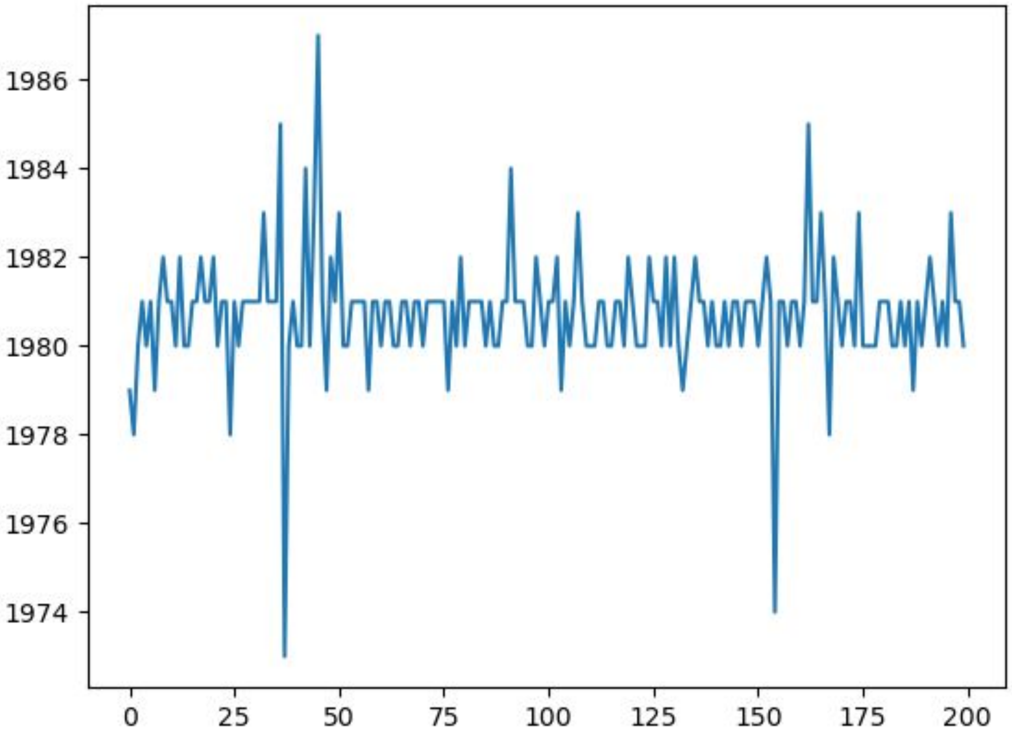
Implementation: Keil (New)

The normalized values are sent to the AI_Process API for prediction and are then returned and copied to the inference array. Based on the inference value “normal” or “abnormal” is transmitted to the UART.

```
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.

    }
}
```

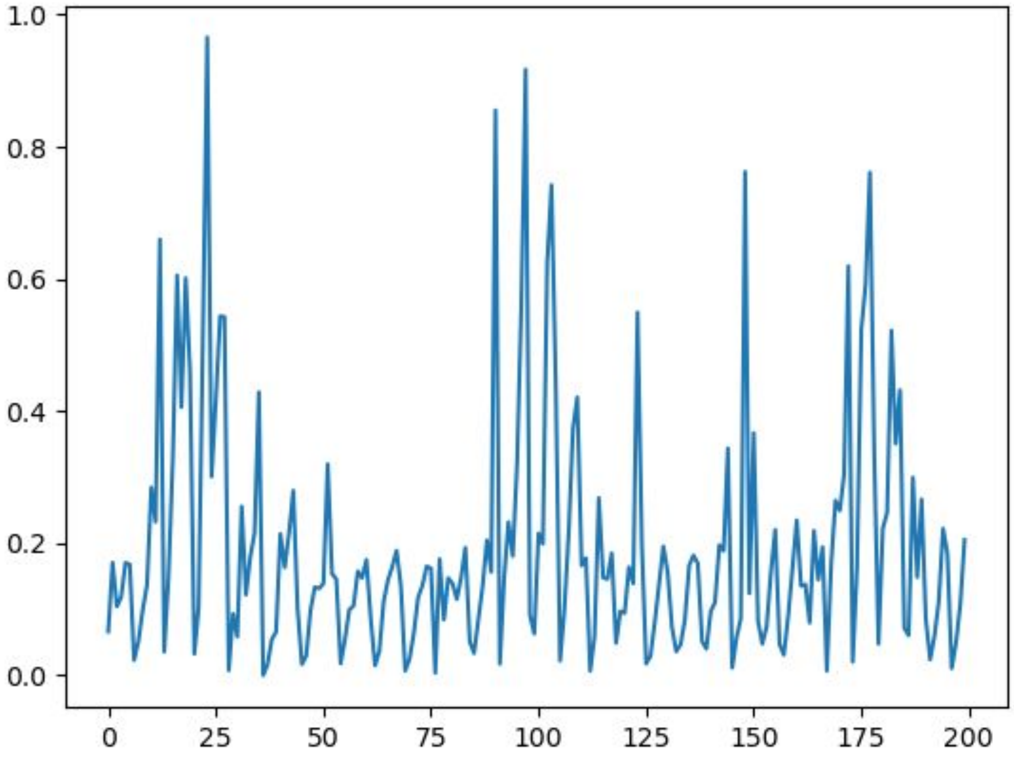
Results

ADC Signal



1986
1984
1982
1980
1978
1976
1974
0
25
50
75
100
125
150
175
200

ADC Signal Normalized



Final output Prediction:

```
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

- The model performs the inference and returns the results to be displayed.
- The results achieved are not quite accurate due to a number of limitations.
  - Data coming from the sensor is noisy.
  - Memory limitations regarding the nucleo MCU led to the usage of a highly compressed model with somewhat degraded results.
  - Limitations regarding Keil IDE did not allow for full implementation of the required preprocessing tasks that are described by the Data set's paper
- Therefore, further work regarding the preprocessing of the data is required to allow for better results.

THANK YOU!