4. Hardware AI <Nizar>

The task of Hardware AI is to provide a model to predict the required gestures during the game. With Ultra96's Zynq processing system, we can make use of the FPGA to program an ML model to predict either a Shield, Reload or Grenade. This section explores the ways that the model was implemented, in terms of training, FPGA implementation and optimizations, and also the challenges faced when implementing the model.

4.1. Ultra96 Synthesis and simulation setup

4.1.1. Section 4.1.1 Synthesis

There were multiple ways that the synthesis of the AI model can be done on Ultra96, The initial implementation was to synthesise the model using a HDL design on vivado. This however would be more complicated and sub-optimal. This is because optimizations has to be made manually and HDL code logic is harder to implement as compared to higher level languages like C++ or Python.

Another option was hls4ml. This tool is easier to use as compared to vivado HDL as the tool will implement the Python scripts onto the FPGA interface. However, the result may not be entirely accurate as the conversion may not be done properly.

Hence the last option, vivado HLS was the best option for our case. We only need to implement a C++ script for the feedward of the model, and then vivado HLS will synthesise the model into an IP Package that can be used in generating the block diagram and bitstream for the FPGA. Vivado HLS also allows for code optimization. This allows us to reduce the latency when predicting.

Simulation Setup

The initial report suggested that Vitis could be used to set up the FPGA. However, when implementing we found that this was not possible as we cannot access the FPGA remotely due to the restrictions by NUS's firewall.

An alternative to programming the bitstream onto the FPGA would be using Pynq's library. This implementation allows us to 'bypass' the firewall issue as we can access the Ultra96's jupyter notebook through SSH tunnelling and subsequently create the necessary python scripts to implement the bitstream onto Ultra96's FPGA. The scripts can then be used by External Communications to integrate the AI model with the visualizer.

Python Productivity for Zynq, also known as Pynq, offers us a platform to integrate our generated bitstream from vivado onto the Zynq's processing system using python scripts. It offers a myriad of libraries, of which, one of them is called Overlay that allows for bitstream implementation on the FPGA. We only need to provide the bitstream, the hardware handover file and the block design file. Pynq will then program the FPGA according to the block diagram that is used to generate the bitstream, and the associating IP in the block diagrams can then be accessed just by calling its name.

For example, we used a DMA in our implementation and the block diagram names the DMA as axi_dma_0 for player 1 and axi_dma_1 for player 2. This can be seen in the images below.

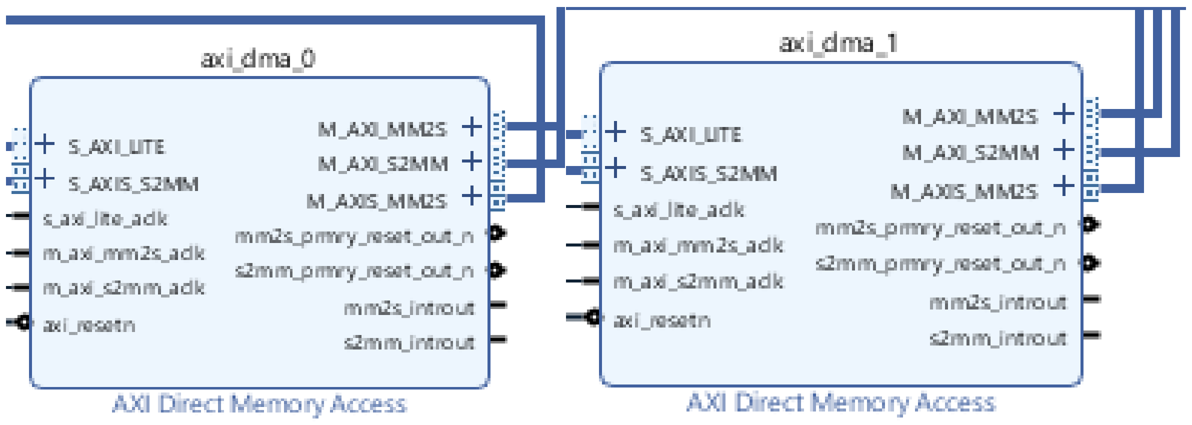
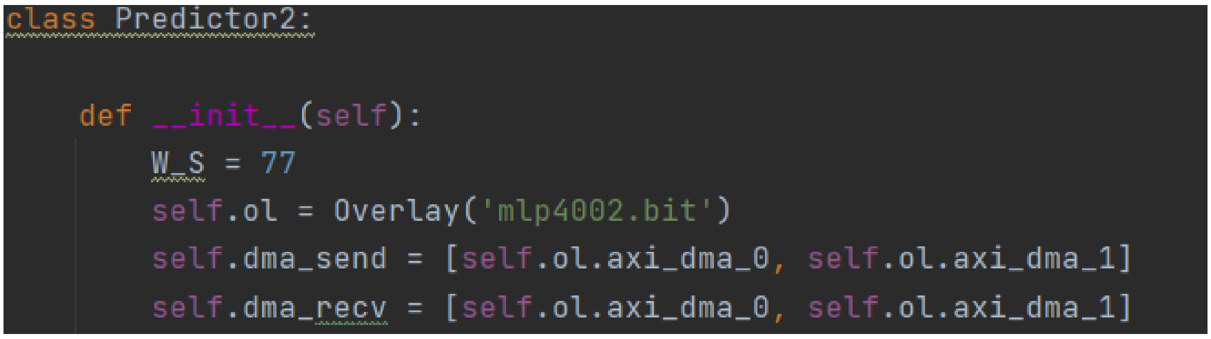


Figure 26. Block IP of the DMA used in the Predictor's Block Design.

This can then be accessed in Pynq as follows:



```
class Predictor2:

    def __init__(self):
        W_S = 77
        self.ol = Overlay('mlp4002.bit')
        self.dma_send = [self.ol.axi_dma_0, self.ol.axi_dma_1]
        self.dma_recv = [self.ol.axi_dma_0, self.ol.axi_dma_1]
```

Figure 27. Code Snippet of the Predictor Class's Constructor.

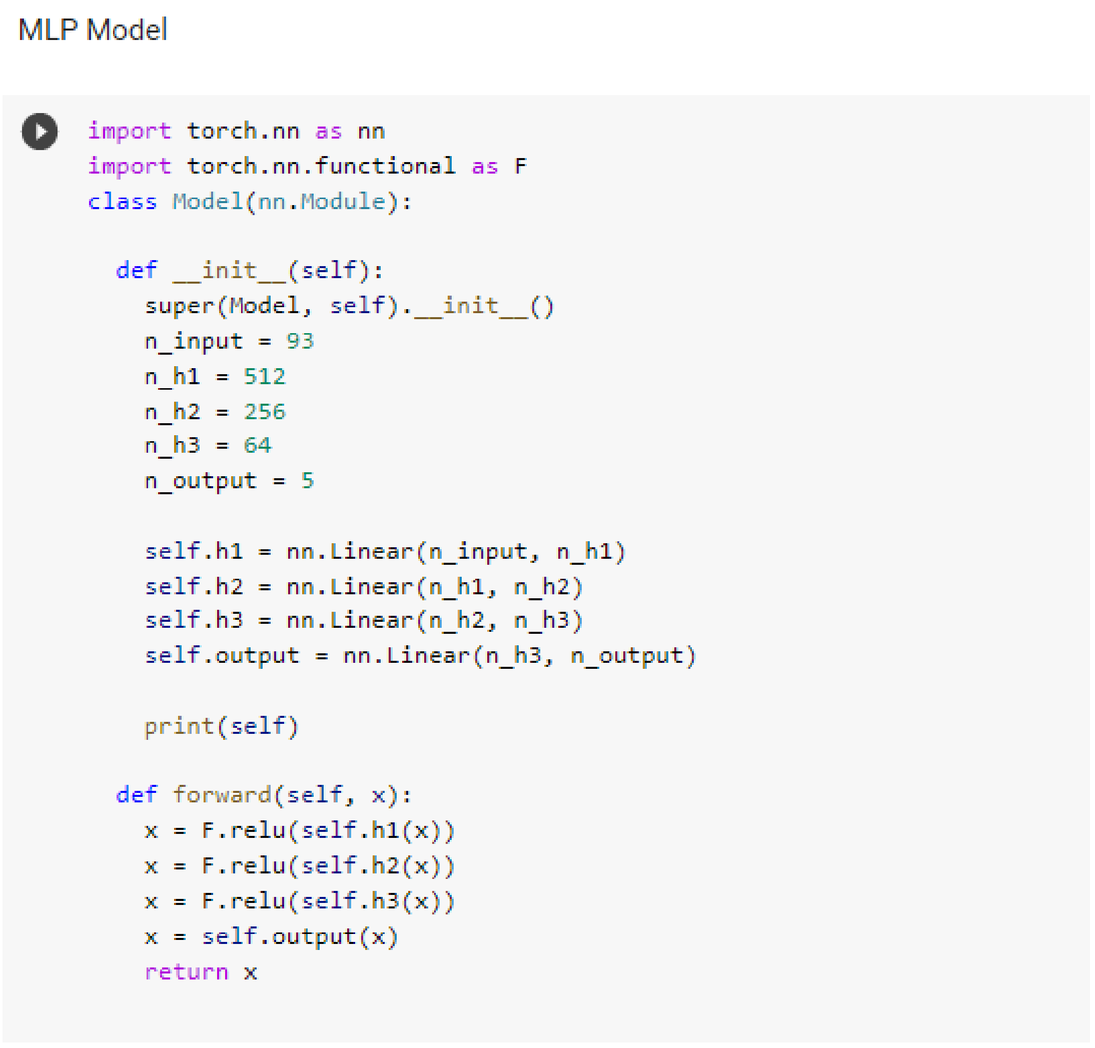
The overlay can be called using its constructor. The path to the .bit file is set as an argument for the overlay's constructor. As the overlay will recursively check for the hardware handover and block design file in the folder, the other two files must also be placed in the same directory. The DMA is then accessed by calling the respective DMA's name as seen in the image above by setting dma_send and dma_rcv to an array of two DMAs. The two variables, dma_send and dma_rcv, act as the input and output lines for the DMA, where the features for the prediction can be sent through the dma_send line and the prediction can be accessed from the dma_rcv line. More details on the Predictor will be explained in the section for realisation of the model onto the FPGA board.

4.2. Neural network

4.2.1. Initial Neural Network

In the initial report, the proposed model was CNN. However, upon embarking on the design, we realise that using CNN may not be the optimal solution. CNN requires the dataset to have spatial information. However the data that we will be receiving in real time will be the 6 attributes , acceleration_x, acceleration_y, acceleration_z, gyro_x, gyro_y, gyro_z , instantaneously. To represent these data spatially, we would have to keep a 2D image of the data which would require more computational effort. Furthermore, CNN would require convolutional layers before using a dense layer which could use up more resources in the FPGA when the model is being realised, as compared to other models like MLP or DNN where only the dense layers are required.

The next approach was to implement a Multi-Layer Perceptron, which acts as a dense layer. Comparing this with CNN, we would require certain features to be extracted beforehand. This is because CNN has the convolutional layers to do the feature extraction for us, hence, with the absence of the convolutional layers, we have to preprocess the incoming data to extract certain key features before putting it into the model. Upon data collection, and visualisation of data, we realised we would require an MLP of 3 hidden layers and one output layer. The model can be seen below.


MLP Model

```
import torch.nn as nn
import torch.nn.functional as F
class Model(nn.Module):

  def __init__(self):
    super(Model, self).__init__()
    n_input = 93
    n_h1 = 512
    n_h2 = 256
    n_h3 = 64
    n_output = 5

    self.h1 = nn.Linear(n_input, n_h1)
    self.h2 = nn.Linear(n_h1, n_h2)
    self.h3 = nn.Linear(n_h2, n_h3)
    self.output = nn.Linear(n_h3, n_output)

    print(self)

  def forward(self, x):
    x = F.relu(self.h1(x))
    x = F.relu(self.h2(x))
    x = F.relu(self.h3(x))
    x = self.output(x)
    return x
```

Figure 28. MLP model’s class.

The sizes of the hidden layers are 512, 256, and 64 respectively. The output layer is of size 5 as we would be predicting either a Shield, Grenade, Reload, Logout or No Gestures. Hence the total number of neurons is 837. For each layer, we would implement a RELU. The reason for RELU is because it can be easily implemented in the FPGA by only reassigning the negative discharge of the perceptron as 0 [14]. The output layer does not have any activation function, as we would want to reduce the computational latency. Since the prediction can be obtained by finding the argmax of the output layer's result, then no further activation function is required.

4.2.2. Training and Testing

The data collected for the design was only from our team members. We planned to collect the data from more individuals but due to time constraints, we were not able to do so. For the data collection phase, we collected the data using the Bluepy integrated script provided by Internal Communications. This allows for the data to be collected using bluetooth, which allows for more representative actions as compared to the real-time situation. The raw values of the attributes are scaled down to signed 8 bits, so that the computational power needed by the FPGA can be reduced.

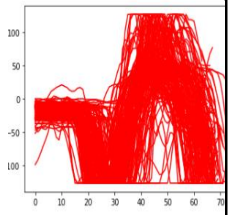
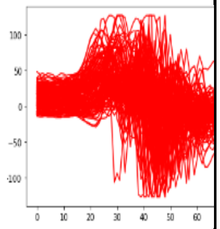
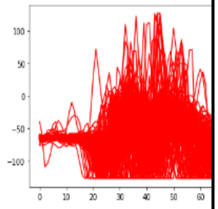
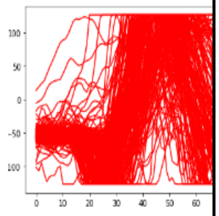
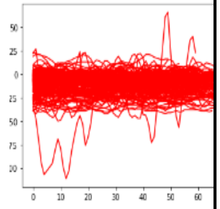
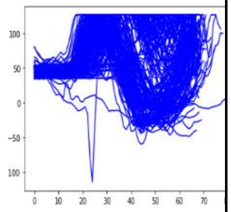
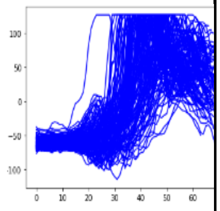
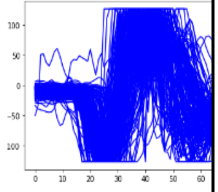
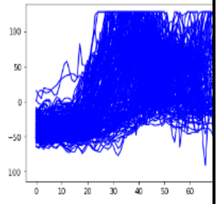
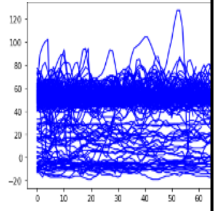
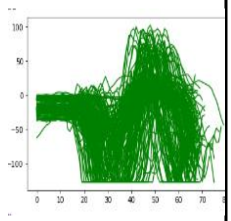
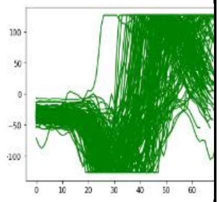
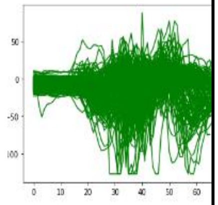
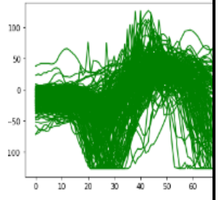
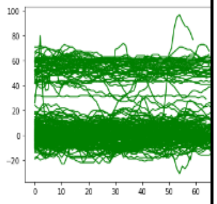
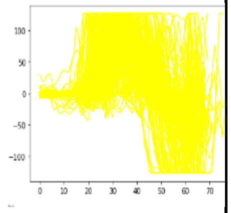
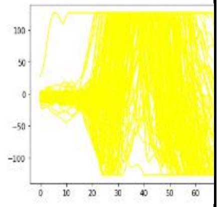
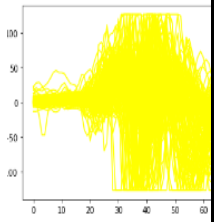
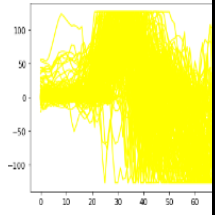
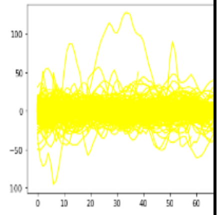
The data is collected in a one second window, with each action having a specific starting gesture. The individuals would then have to complete the gestures in that window, and the data is then extracted into a text file. From the text files, we can then visualise the data to check for any visible patterns. This would allow for better features extraction as we can see any visible correlations, or minimum or maximum outputs. The visualisations can be found on Figure 29.

From the visualisation, we can see that there are specific peaks, namely that most attributes usually hit a value of 127 or -127. There is also a pattern where certain positive peaks usually come before the negative peaks. Some of the attributes are also negatively correlated. All of which are strong features to determine an action. We conducted a feature extraction using statistical variables. The features are minimum value, maximum value, average value, median value, mean absolute deviation, standard deviation, interquartile range, entropy, root mean square, range, average magnitude and median absolute deviation of frequencies obtained through FFT, time difference between positive and negative peaks and correlation between every two distinct attributes. These bring a total of 93 features, where the correlation between every two distinct attributes add up to 15 features, while the other 13 statistical variables on each attribute add up to 78 features. These 93 features then make up the input layer as seen in the previous figure.

Before training, we conducted a correlation check of each feature of each datapoint in the dataset with the output label. We realised that there is a high correlation with most features and an exceptionally strong correlation between the output labels and features extracted from acceleration_x. A table on the correlation of the features with the output labels is found on Figure 30.

The dataset is then trained using SGD with batch size 1 and 50 epochs, with 90% of the dataset as training set while the rest as test set. The reason for a very small batch size is because we collected a small size of dataset of about 150 data points per gesture. A small batch size allows the model to be more sensitive to the pattern of the training set. However, this also means that the model will be sensitive to the noise of the training set, in which we reduced this by setting a very low learning rate of 0.000008 and a momentum of 0.9 to allow for a steady convergence of the model's target concept. We implemented a scheduler for the learning rate to reduce at the next epoch when the learning rate meets a plateau. This prevents overfitting as it ensures that the learning rate is reduced when validation loss does not decrease. The entire training is done on 10 K-fold to ensure that the model is not overfitting and that it is exposed to the most optimal training and test set. After training, we picked the K-fold with the least validation loss,and extracted the model's weight from there.

Since the weights in Pytorch are in floats, we would have to convert the weights to integer for easier computation in the FPGA. The weights from Pytorch are scaled by 1000 and then converted into integers by rounding it up to its nearest integer, The weights are then saved and copied over to the HLS code. We trained the dataset for both models on only 3 teammates. For testing, we then tested it on the other two team members. The model showed a fairly good prediction for the other two teammates who are not part of the training, hence showing an absence of overfitting.

	Shield	Grenade	Reload	Logout	Non_Gesture
acc_x					
acc_y					
acc_z					
gyro_x					

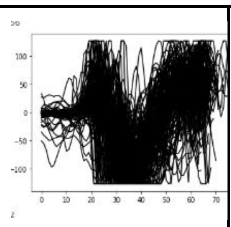
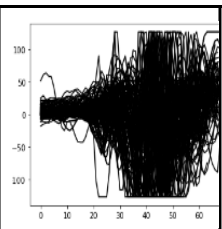
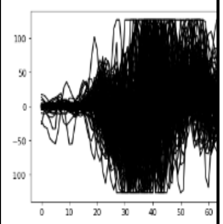
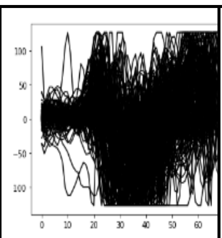
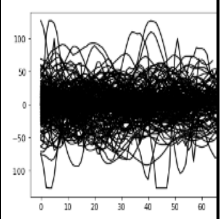
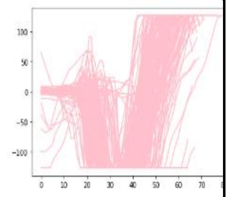
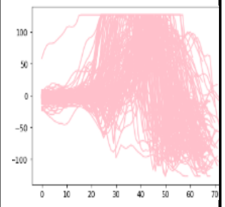
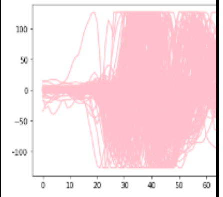
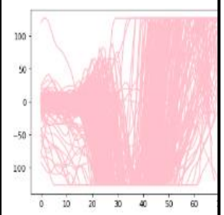
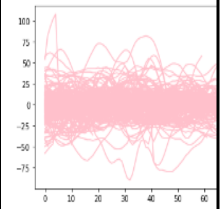
gyro_y					
gyro_z					

Figure 29. Visualisation of datasets based on attributes and actions.

min ax	-0.656466	entropy gy	-0.178313
max ax	0.574839	rms gy	0.357953
mean ax	-0.184144	range_gy	0.494008
median ax	-0.551588	avg magf gy	0.465693
mad ax	0.690788	madf gy	0.448533
sd ax	0.680181	tdiff gy	0.117354
iqr ax	0.634598	min gz	-0.670152
entropy ax	-0.499144	max gz	0.616115
rms ax	0.780757	mean gz	-0.154732
range_ax	0.692766	median gz	-0.246736
avg magf ax	0.721205	mad gz	0.602388
madf ax	0.703360	sd gz	0.627525
tdiff ax	0.154476	iqr gz	0.486880
min ay	-0.683610	entropy gz	0.186566
max ay	0.214595	rms gz	0.649608
mean ay	-0.580172	range_gz	0.682678
median ay	-0.478794	avg magf gz	0.672620
mad ay	0.565862	madf gz	0.676959
sd ay	0.575342	tdiff gz	0.128734
iqr ay	0.547271	corr ax ay	0.492050
entropy ay	-0.531043	corr ax az	0.407982
rms ay	0.009848	corr ax gx	0.176024
range_ay	0.588428	corr ax gy	-0.094221
avg magf ay	0.508606	corr ax gz	0.118484
madf ay	0.507982	corr ay az	0.507900
tdiff ay	0.504794	corr ay gx	0.060878
min az	-0.394895	corr ay gy	-0.170004
max az	0.219627	corr ay gz	-0.048806
mean az	-0.166159	corr az gx	-0.154101
median az	-0.234314	corr az gy	-0.065912
mad az	0.246654	corr az gz	-0.048586
sd az	0.266111	corr gx gy	-0.062654
iqr az	0.176832	corr gx gz	-0.342900
entropy az	-0.559118	corr gy gz	0.148125

```
rms az          0.087313
range_az        0.360820
avg magf az     0.265147
madf az         0.215290
tdiff az        0.065898
min gx         -0.464207
max gx          0.567771
mean gx         0.110417
median gx       0.191422
mad gx          0.385936
sd gx           0.429176
iqr gx          0.230918
entropy gx     -0.367842
rms gx          0.432051
range_gx        0.553383
avg magf gx     0.495605
madf gx         0.493668
tdiff gx       -0.285065
min gy         -0.494789
max gy          0.413546
mean gy         0.030145
median gy       0.092823
mad gy          0.387435
sd gy           0.431149
iqr gy          0.200439
```

Figure 30. Correlation between output labels and extracted features of each attribute.

209	0	0	0	0
0	156	0	0	0
0	0	143	0	0
0	0	0	156	0
0	0	0	0	195

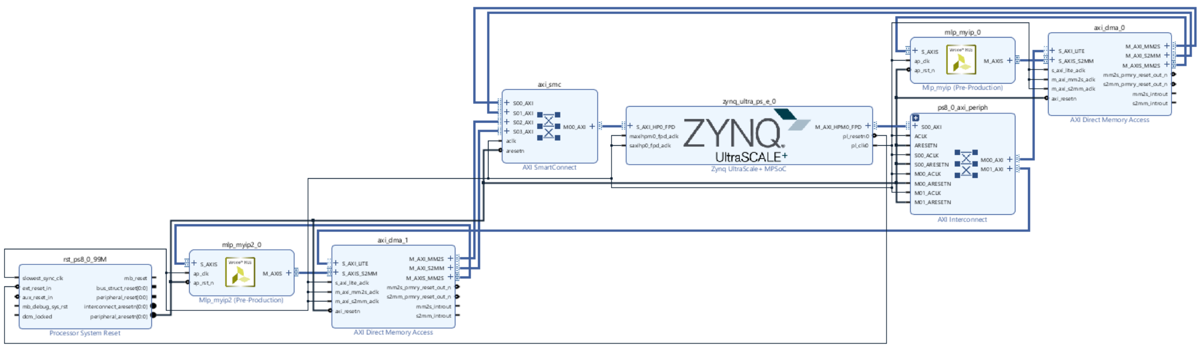
Figure 31. Confusion Matrix of Software Training (Output By Target).

Link for verification of training:
https://colab.research.google.com/drive/1H_TlOTmgkc8pPtO7jLFNKIWrgBq-iocP#scrollTo=-2Kc1fe2FCtN

4.3. Realisation of model on the FPGA board

4.3.1. Block Diagram

The block diagram can be seen on page 36. We used 2 DMAs for each model. The DMA allows us to send in input from the python scripts and to read the prediction from the model. The DMA are connected to the Zynq's processing system using a smart interconnect, which allows for optimal connection and switching of lines. We used two models because the IMU's sensitivity is largely different for each player, hence a differently trained model is required for each player. Therefore we came up with two IP packages for the HLS models and used 2 DMA to read from and write to the model when the IMU sends in data.



4.3.2. Feedforward implementation

The FPGA implementation only requires a feedforward because the model has already been trained, and no backpropagation is required to reinforce learning. We only require the prediction from the model and computation of validation loss is not possible during live sessions, hence, only feed forward is required.

The feedforward equation is as follows:

$$Z = Wx + b$$

As seen in the equation above,W is the weight matrix, x is the input matrix for the layer, and b is the bias matrix. Since there exists 3 hidden layers and one output layer, there will be 4 weight matrices needed to compute Z for each layer and 4 bias matrices. The computation for the prediction is done by the following steps.

Feed Forward Algorithm:

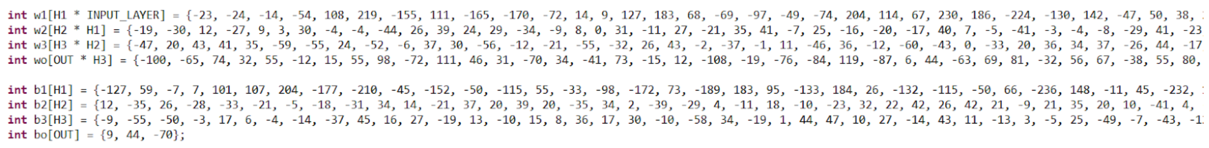
1. Compute Z1 : Z1 = W1 * x1 + b1, W1 is of size 512 (hidden layer 1 size) by 18 (input layer size), x1 is the 18 input features and b1 is the bias matrix for hidden layer 1 and is of the size 512 by 1. Therefore the resulting matrix Z1 is 512 by 1.
2. Compute ReLU for Z1: ReLU function is f(x) = max(0, x). Therefore, for each value of z in Z1, if z < 0 assign to 0 else keep the value that it has.

3. Compute Zj and ReLU: Zj = Wj * Zj-1 + bj, for j = 2 to 4 where 4 is the output layer. Wj is the weight , bj is the bias for the respective layers and Zj-1 is the output of the previous layers. Apply the same ReLU logic for each element of Zj, for j =2 and j =3, before feeding inputs to the next layer.
4. At j = 4, when Z4 is computed, pick the argmax of the array. The prediction is the argmax of the array.

4.3.3. Ultra96 Synthesis

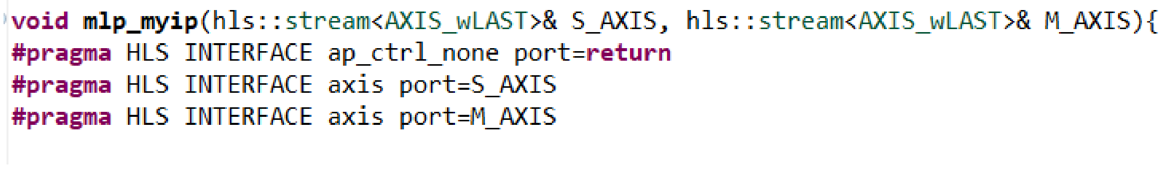
The following would be the steps taken to create a HLS synthesis of the model, followed by setting it up on Ultra96.

- Initialise Weights:



```
int w1[H1 * INPUT_LAYER] = {-23, -24, -14, -54, 108, 219, -155, 111, -165, -170, -72, 14, 9, 127, 183, 68, -69, -97, -49, -74, 204, 114, 67, 230, 186, -224, -130, 142, -47, 50, 38,
int w2[H2 * H1] = {-19, -30, 12, -27, 9, 3, 30, -4, -4, -44, 26, 39, 24, 29, -34, -9, 8, 0, 31, -11, 27, -21, 35, 41, -7, 25, -16, -20, -17, 40, 7, -5, -41, -3, -4, -8, -29, 41, -23
int w3[H3 * H2] = {-47, 20, 43, 41, 35, -59, -55, 24, -52, -6, 37, 30, -56, -12, -21, -55, -32, 26, 43, -2, -37, -1, 11, -46, 36, -12, -60, -43, 0, -33, 20, 36, 34, 37, -26, 44, -17
int wo[OUT * H3] = {-100, -65, 74, 32, 55, -12, 15, 55, 98, -72, 111, 46, 31, -70, 34, -41, 73, -15, 12, -108, -19, -76, -84, 119, -87, 6, 44, -63, 69, 81, -32, 56, 67, -38, 55, 80,

int b1[H1] = {-127, 59, -7, 7, 101, 107, 204, -177, -210, -45, -152, -50, -115, 55, -33, -98, -172, 73, -189, 183, 95, -133, 184, 26, -132, -115, -50, 66, -236, 148, -11, 45, -232,
int b2[H2] = {12, -35, 26, -28, -33, -21, -5, -18, -31, 34, 14, -21, 37, 20, 39, 20, -35, 34, 2, -39, -29, 4, -11, 18, -10, -23, 32, 22, 42, 26, 42, 21, -9, 21, 35, 20, 10, -41, 4,
int b3[H3] = {-9, -55, -50, -3, 17, 6, -4, -14, -37, 45, 16, 27, -19, 13, -10, 15, 8, 36, 17, 30, -10, -58, 34, -19, 1, 44, 47, 10, 27, -14, 43, 11, -13, 3, -5, 25, -49, -7, -43, -1
int bo[OUT] = {9, 44, -70};
```

- Define pragmas for interface to allow AXI communications:

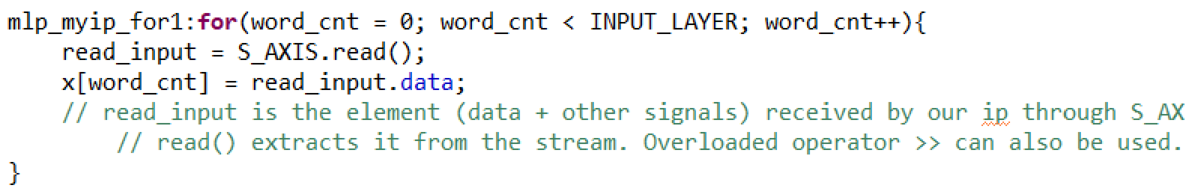


```
void mlp_myip(hls::stream<AXIS_wLAST>& S_AXIS, hls::stream<AXIS_wLAST>& M_AXIS){
#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE axis port=S_AXIS
#pragma HLS INTERFACE axis port=M_AXIS
```

- Define scale used to convert floating points of weights and biases into integers:

```
int scale = 1000;
```

- Read data through S_AXIS into an input array. **Note: S_AXIS last is not set to 1 because both Masters and Slaves know the input size. Only need to assign data in the AXI line into the array



```
mlp_myip_for1:for(word_cnt = 0; word_cnt < INPUT_LAYER; word_cnt++){
    read_input = S_AXIS.read();
    x[word_cnt] = read_input.data;
    // read_input is the element (data + other signals) received by our ip through S_AX
       // read() extracts it from the stream. Overloaded operator >> can also be used.
}
```

- Implement feed forward by doing matrix multiplication of weights with input using row major order, and adding in bias at the end of every row. The resulting sum would then be scaled down by the scale factor. Note: Row for w1 is the length of input vector, therefore when deriving modulo of word count from input layer size, if the value is lesser than input layer by 1, the word count is at the end of the row in the weight matrix.

```
int z1[H1] = {0};
int j = 0;
mlp_myip_for2:for(word_cnt = 0; word_cnt < H1*INPUT_LAYER; word_cnt++) {

    z1[j] = z1[j] +  w1[word_cnt] * x[word_cnt%INPUT_LAYER];
    if (word_cnt % INPUT_LAYER == INPUT_LAYER - 1) {
        z1[j] = (z1[j] + b1[j]) < 0 ? 0 :  ((z1[j] + b1[j])/scale);
        j = j + 1;
    }
}
```

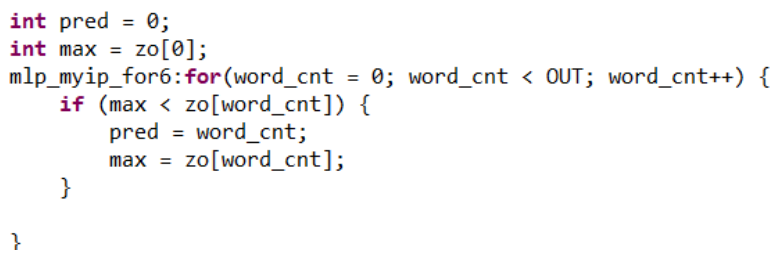
- Implement feed forward for respective layers. Note that the input value for each layer is the resultant matrix from previous layer

```
int z2[H2] ={0};
j = 0;
mlp_myip_for3:for(word_cnt = 0; word_cnt < H2*H1; word_cnt++) {

    z2[j] = z2[j] +  w2[word_cnt] * z1[word_cnt%H1];
    if (word_cnt % H1 == H1 - 1) {
        z2[j] = (z2[j] + b2[j]) < 0 ? 0 :  ((z2[j] + b2[j])/scale);
        j = j + 1;
    }
}

int z3[H3] = {0};
j = 0;
mlp_myip_for4:for(word_cnt = 0; word_cnt < H3*H2; word_cnt++) {

    z3[j] = z3[j] +  w3[word_cnt] * z2[word_cnt%H2];
    if (word_cnt % H2 == H2 - 1) {
        z3[j] = (z3[j] + b3[j]) < 0 ? 0 :  ((z3[j] + b3[j])/scale);
        j = j + 1;
    }
}
```

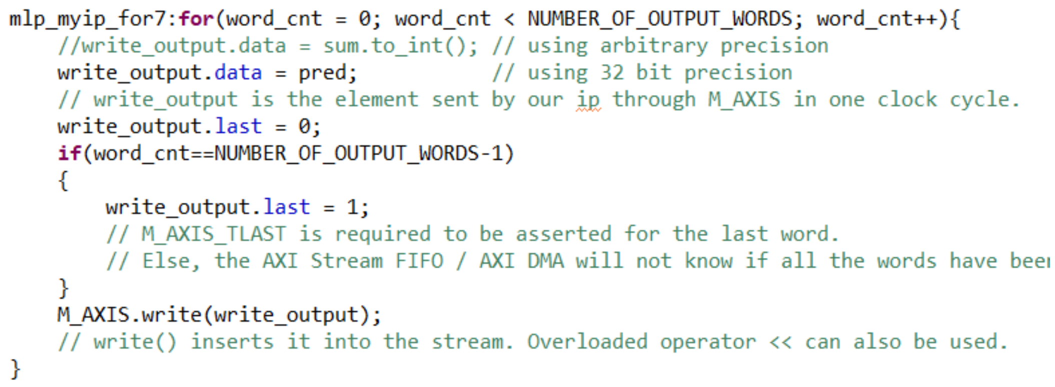
- Implement feed forward for the output layer. Note that no ReLU is used for the output layer.

```
int zo[OUT] = {0};
j = 0;
mlp_myip_for5:for(word_cnt = 0; word_cnt < OUT*H3; word_cnt++) {
    zo[j] = zo[j] +  wo[word_cnt] * z3[word_cnt%H3];
    if (word_cnt % H3 == H3 - 1) {
        zo[j] = zo[j] + bo[j];
        j = j + 1;
    }
}
```

- Obtain prediction by finding the argmax of the output array


```
int pred = 0;
int max = zo[0];
mlp_myip_for6:for(word_cnt = 0; word_cnt < OUT; word_cnt++) {
    if (max < zo[word_cnt]) {
        pred = word_cnt;
        max = zo[word_cnt];
    }

}
```

- Write output to the M_AXIS line. Note that the last bit is set to 1 to indicate that the last output is written into the M_AXIS line so that DMA need not wait for any further output


```
mlp_myip_for7:for(word_cnt = 0; word_cnt < NUMBER_OF_OUTPUT_WORDS; word_cnt++){
    //write_output.data = sum.to_int(); // using arbitrary precision
    write_output.data = pred;           // using 32 bit precision
    // write_output is the element sent by our ip through M_AXIS in one clock cycle.
    write_output.last = 0;
    if(word_cnt==NUMBER_OF_OUTPUT_WORDS-1)
    {
        write_output.last = 1;
        // M_AXIS_TLAST is required to be asserted for the last word.
        // Else, the AXI Stream FIFO / AXI DMA will not know if all the words have bee
    }
    M_AXIS.write(write_output);
    // write() inserts it into the stream. Overloaded operator << can also be used.
}
```

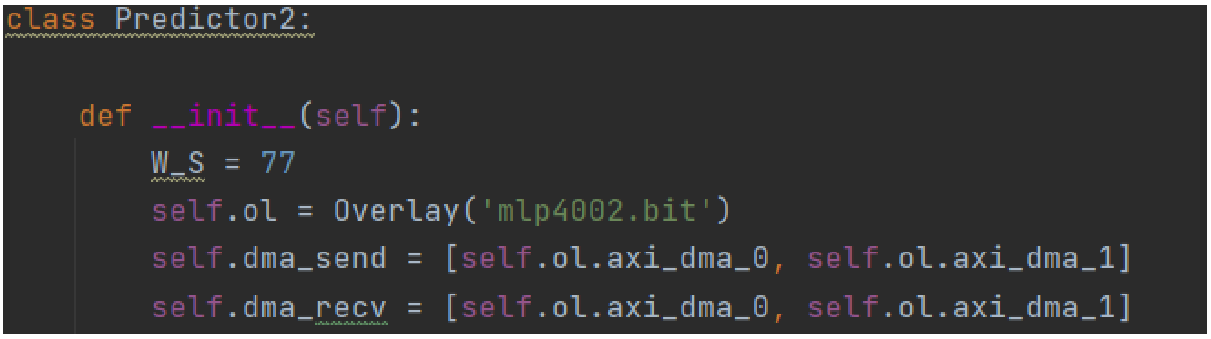
- Synthesise and export the HLS into the solution folder in Vivado HLS
- Resulting HLS code will appear in the solution folder.

4.3.4. Importing IP Core into Vivado

The HLS code can be imported into the block diagram as an IP package. This can be done by adding the solution folder that was generated in the previous step into vivado's IP repo. Ultra96 Zynq's IP has to be imported inside as well, which can be retrieved from the board repository in Vivado if it cannot be found by default. With the necessary IPs imported, we are only left with adding in the DMA as a communication interface. When the block diagram is ready, generate bitstream and export the bitstream file (.bit), hardware handover (.hwh) and block diagram file (.tcl) to Ultra96. Ensure that the files are in the same directory.

4.3.5. Ultra96 Setup

With the files above, we can use Pynq's library to create an overlay and program the FPGA in the ultra96 with bit file.

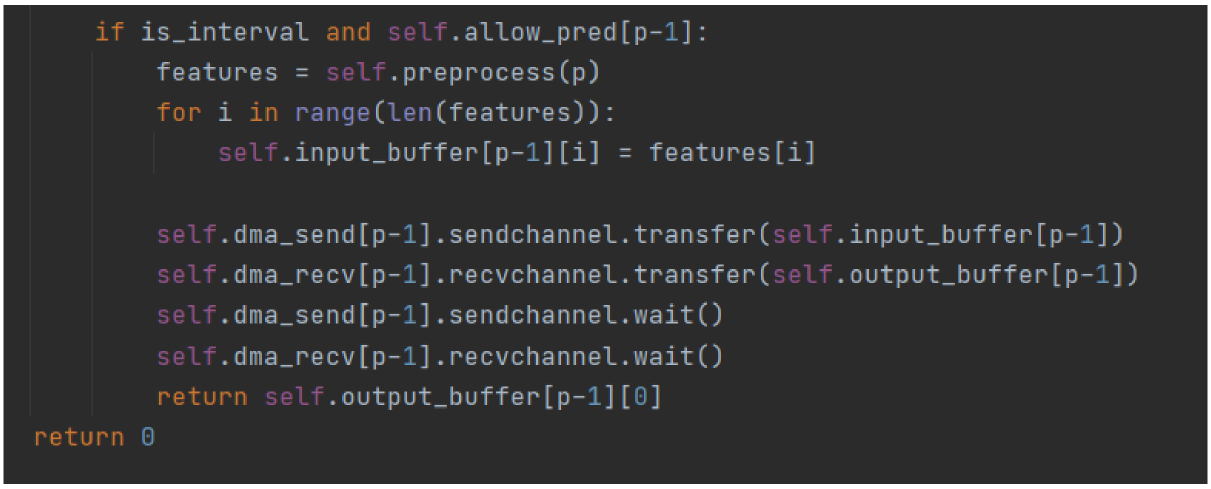


```python
class Predictor2:

    def __init__(self):
        W_S = 77
        self.ol = Overlay('mlp4002.bit')
        self.dma_send = [self.ol.axi_dma_0, self.ol.axi_dma_1]
        self.dma_recv = [self.ol.axi_dma_0, self.ol.axi_dma_1]
```

Figure 32. Code Snippet of the Predictor Class's Constructor.

The DMA can be accessed by calling the names used in the block diagram. We have a dma_send and dma_recv as different lines for writing and reading from the DMA. The features are sent through dma_send while the predictions are read through dma_recv.

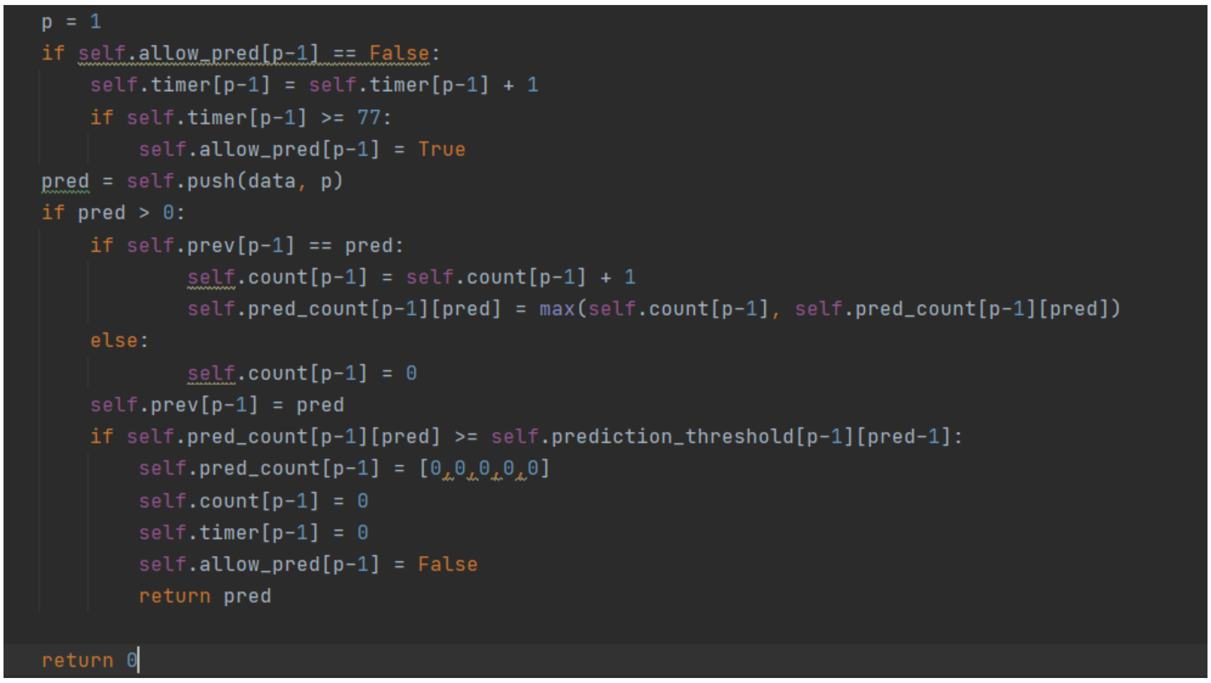
We also created input and output buffers to store the input and output data that is to be sent or received from the DMA. When we are sending the input to the DMA, we will write the input into the input buffer, and subsequently read it from the output buffer. This can be seen in the image below.



```python
    if is_interval and self.allow_pred[p-1]:
        features = self.preprocess(p)
        for i in range(len(features)):
            self.input_buffer[p-1][i] = features[i]

        self.dma_send[p-1].sendchannel.transfer(self.input_buffer[p-1])
        self.dma_recv[p-1].recvchannel.transfer(self.output_buffer[p-1])
        self.dma_send[p-1].sendchannel.wait()
        self.dma_recv[p-1].recvchannel.wait()
        return self.output_buffer[p-1][0]
return 0
```

Figure 33. Code Snippet of the Predictor Class's Push To DMA.

We set an interval for the prediction to be retrieved from the model. The interval value is 7. This means that at every 7th incoming data from External Communications, the model will output a value for prediction. This is then stored in another buffer to calculate the accumulation of consecutive predictions. If the number of consecutive predictions meets the threshold for a certain gesture, the model will then send that prediction to External Communications. This can be seen in the image below, for the prediction of Player 1's Gesture.

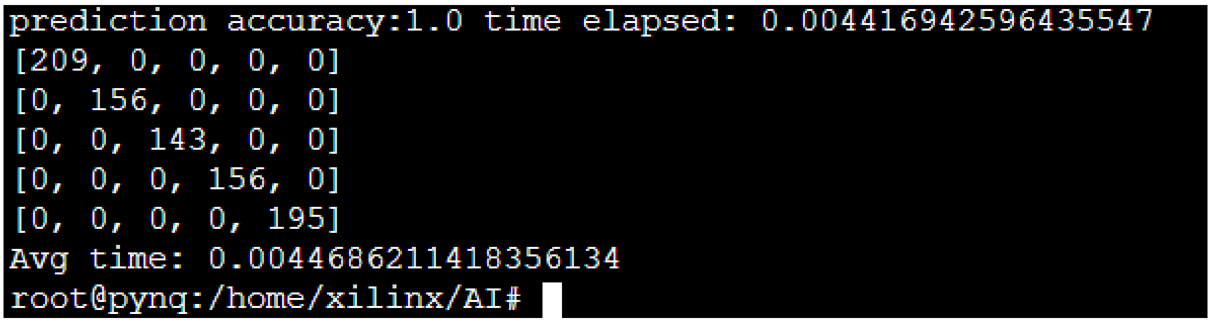


```
p = 1
if self.allow_pred[p-1] == False:
    self.timer[p-1] = self.timer[p-1] + 1
    if self.timer[p-1] >= 77:
        self.allow_pred[p-1] = True
pred = self.push(data, p)
if pred > 0:
    if self.prev[p-1] == pred:
            self.count[p-1] = self.count[p-1] + 1
            self.pred_count[p-1][pred] = max(self.count[p-1], self.pred_count[p-1][pred])
    else:
            self.count[p-1] = 0
    self.prev[p-1] = pred
    if self.pred_count[p-1][pred] >= self.prediction_threshold[p-1][pred-1]:
        self.pred_count[p-1] = [0,0,0,0,0]
        self.count[p-1] = 0
        self.timer[p-1] = 0
        self.allow_pred[p-1] = False
        return pred

return 0
```

Figure 34. Code Snippet of the Predictor Class's Send To DMA.

Note that in this function, we set allow_pred to False when prediction is sent to External Communications. This is to block off any predictions for one second. When the timer reaches 77, which is equivalent to 1 second given a data rate of 77Hz, allow_pred is then set to true to allow for subsequent predictions.

We also have a testbench, predictor.py, created for the FPGA implementation of the models. This is done by using the features that we used for training, and pushing in each row of features into the predictor's push to dma function. A confusion matrix was then obtained.



```
prediction accuracy:1.0 time elapsed: 0.004416942596435547
[209, 0, 0, 0, 0]
[0, 156, 0, 0, 0]
[0, 0, 143, 0, 0]
[0, 0, 0, 156, 0]
[0, 0, 0, 0, 195]
Avg time: 0.0044686211418356134
root@pynq:/home/xilinx/AI#
```

Figure 35. Terminal Output of Testbench Script Predictor.py.

As seen, both matrices are the same for hardware and software.

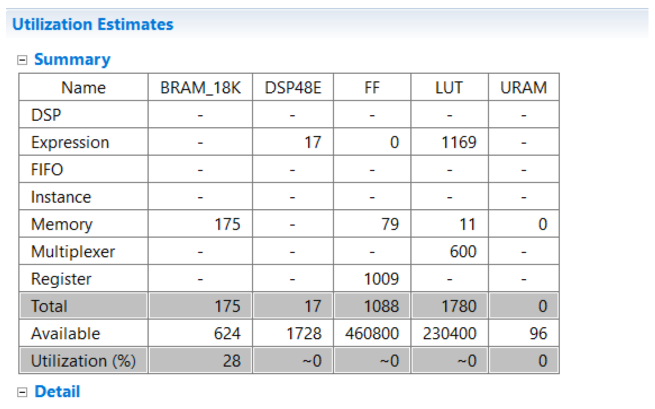
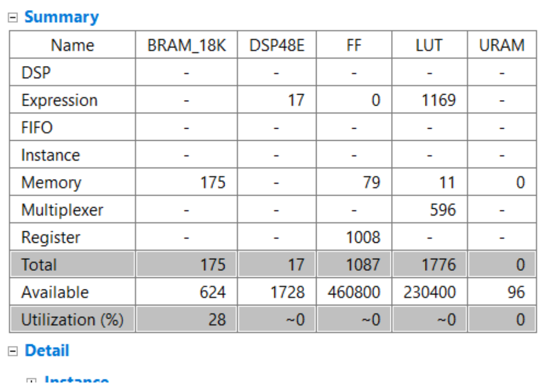
4.4. Design's timing, power and area with the given dataset.

4.4.1. Timing

The timing has been reduced by using optimizations such as DMA, and predicting in intervals. This allows the coprocessor to be used only when needed (in terms of intervals) and does not use up the coprocessor's computation efforts in remembering the inputs in terms of DMA. The timing is also optimised by using integer computation instead of floating points. This trades off accuracy at the expense of timing. However, the confusion matrix did not differ much from the software training, hence this optimization did not affect the accuracy as much. The latency for prediction is 0.004468 seconds. However, when taking into account real-time predictions, and latency with respect to data coming from Internal Communications, and External communications sending to Eval Server, the time taken was on average 4 seconds, inclusive of human reaction time.

4.4.2. Area

The area of the design could not be optimised as the model was large and most of the BRAM was used even without any hardware code optimization. With code optimization such as loop unrolling or pipelining, the BRAM utilisation can increase up to more than 50% per model, which in total would exceed the available resources.

Model 1	Model 2
	

Model 1

Utilization Estimates

⊟ **Summary**

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	17	0	1169	-
FIFO	-	-	-	-	-
Instance	-	-	-	-	-
Memory	175	-	79	11	0
Multiplexer	-	-	-	600	-
Register	-	-	1009	-	-
Total	175	17	1088	1780	0
Available	624	1728	460800	230400	96
Utilization (%)	28	~0	~0	~0	0

⊟ **Detail**

Model 2

⊟ **Summary**

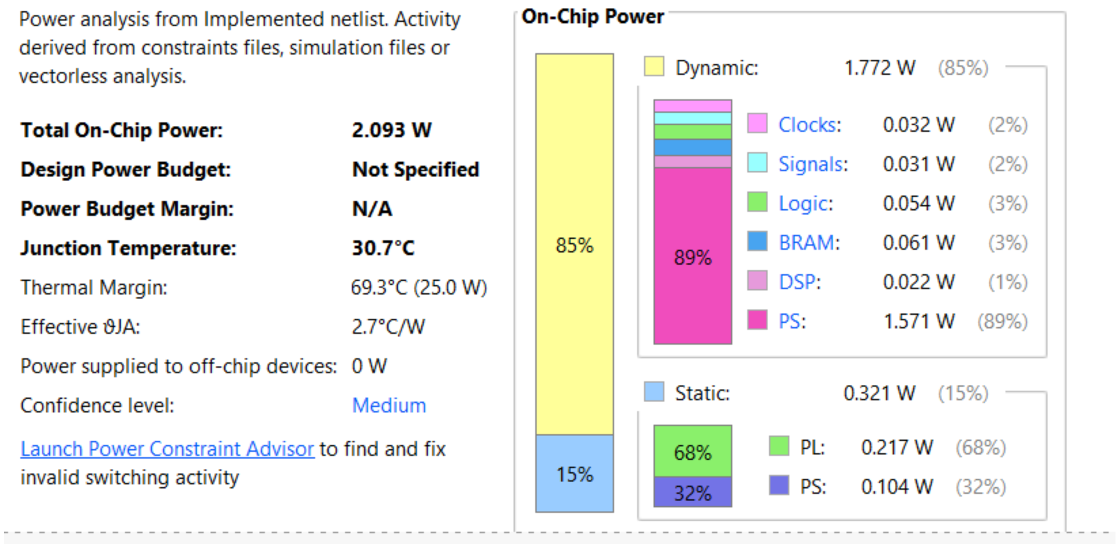
Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	17	0	1169	-
FIFO	-	-	-	-	-
Instance	-	-	-	-	-
Memory	175	-	79	11	0
Multiplexer	-	-	-	596	-
Register	-	-	1008	-	-
Total	175	17	1087	1776	0
Available	624	1728	460800	230400	96
Utilization (%)	28	~0	~0	~0	0

⊟ **Detail**

Figure 36. Hardware Resources Utilisation.

The total BRAM utilisation for both models is 56% without any hardware code optimization.

4.4.3. Power



Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**2.093 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**30.7°C**
Thermal Margin:	69.3°C (25.0 W)
Effective ϑJA:	2.7°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Medium

Launch Power Constraint Advisor to find and fix invalid switching activity

On-Chip Power

🟨	Dynamic:	1.772 W	(85%)
🟪	Clocks:	0.032 W	(2%)
🟦	Signals:	0.031 W	(2%)
🟩	Logic:	0.054 W	(3%)
🟦	BRAM:	0.061 W	(3%)
🟪	DSP:	0.022 W	(1%)
🟥	PS:	1.571 W	(89%)
🟦	Static:	0.321 W	(15%)
🟩	PL:	0.217 W	(68%)
🟦	PS:	0.104 W	(32%)

Figure 37. Power Utilisation Report from Vivado

The power is used up mainly dynamically, of which, most of it is used up by the PS logic. Statically, most of the power is used up by the PL logic. The dynamic power is 1.772 W which is the amount of power used when the design is switching between High and Low in the datapath and clock values. The total static power is 0.321W which means this is the minimum amount of power the design needs in order to operate, even when no action is done. The power consumption was optimised by ensuring that the model is only used at every 0.1s or rather at every 7th incoming data. This means that the prediction is made 11 times in 1 second, given a data rate of 77 incoming data per second. Therefore prediction is made at 0.142% of the entire live session. Hence the dynamic power is used at every 0.1s, making the total power 2.093 W at 0.142% of the live session while the remaining time, the power consumption is at 0.321W.