GALLOGLY COLLEGE OF ENGINEERING
The UNIVERSITY of OKLAHOMA

SNN-Based Gesture Recognition on the Arduino Portenta H7: STM32 HAL-Enabled

Introduction

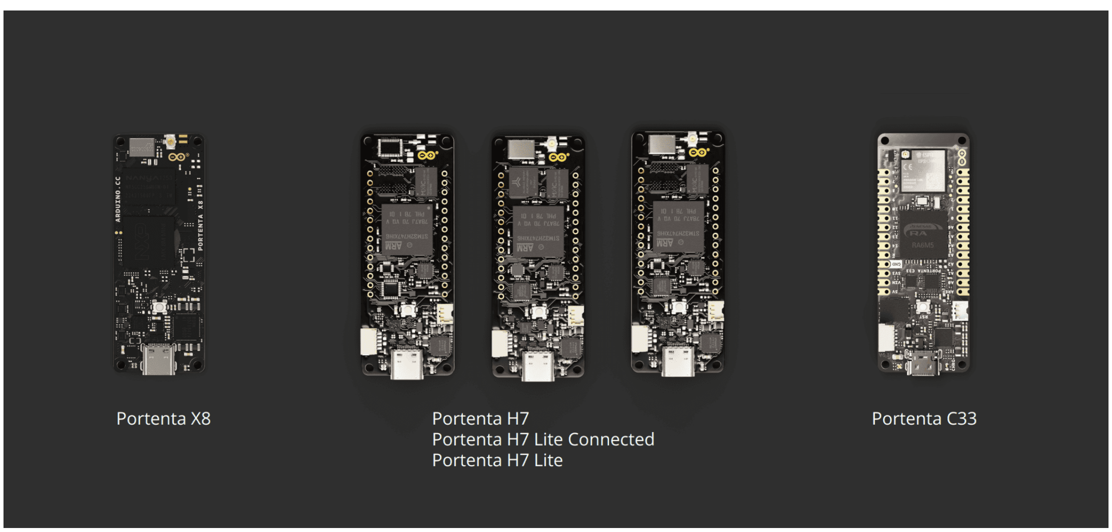
SNN based Artificial Intelligences serve to further streamline applications of artificial intelligence in endpoint devices. To accomplish this, we must focus on the individual design points that are considered when setting up a project idea and executing upon the project. In this manual, we will be focusing on the following learning points:

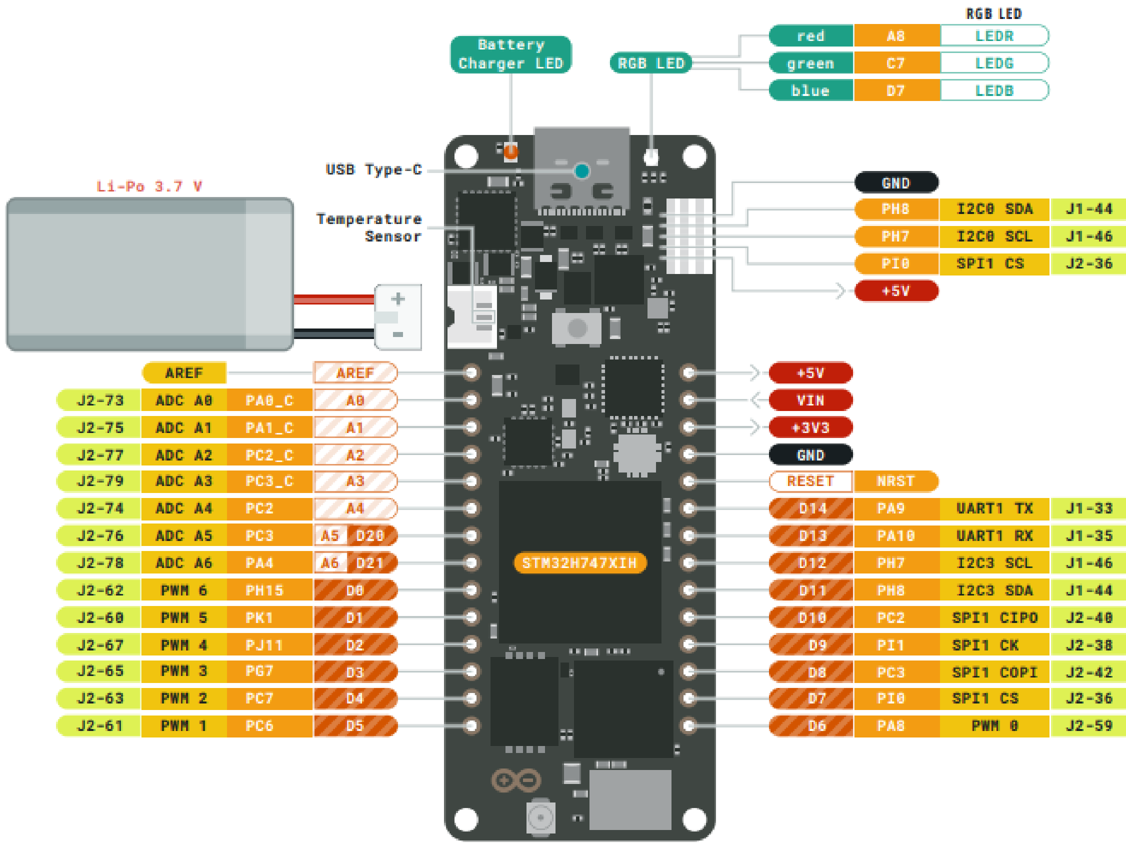
1. SNN Tuning
2. Dataset Preparation
3. Signal Processing w/ Radially Invariant Fourier Transformation
4. SNN Training Mechanisms
5. Network Classification via Hybridization

In this project, you will be implementing a hand gesture recognition system on the edge device known as the Arduino Portenta H7. It's uniquely designed dual processor architecture includes an H7 Core, H4 Core, and 2 mb Flash storage. This enables us to pre-train increasingly complex neural networks before implementing on field deployments without concern of network capacity. By the end of the project, your SNN will be capable of detecting any number of gestures predefined; but for the sake of this manual, we will focus on 2 pre-collected datasets.

Hardware Description

Portenta H7



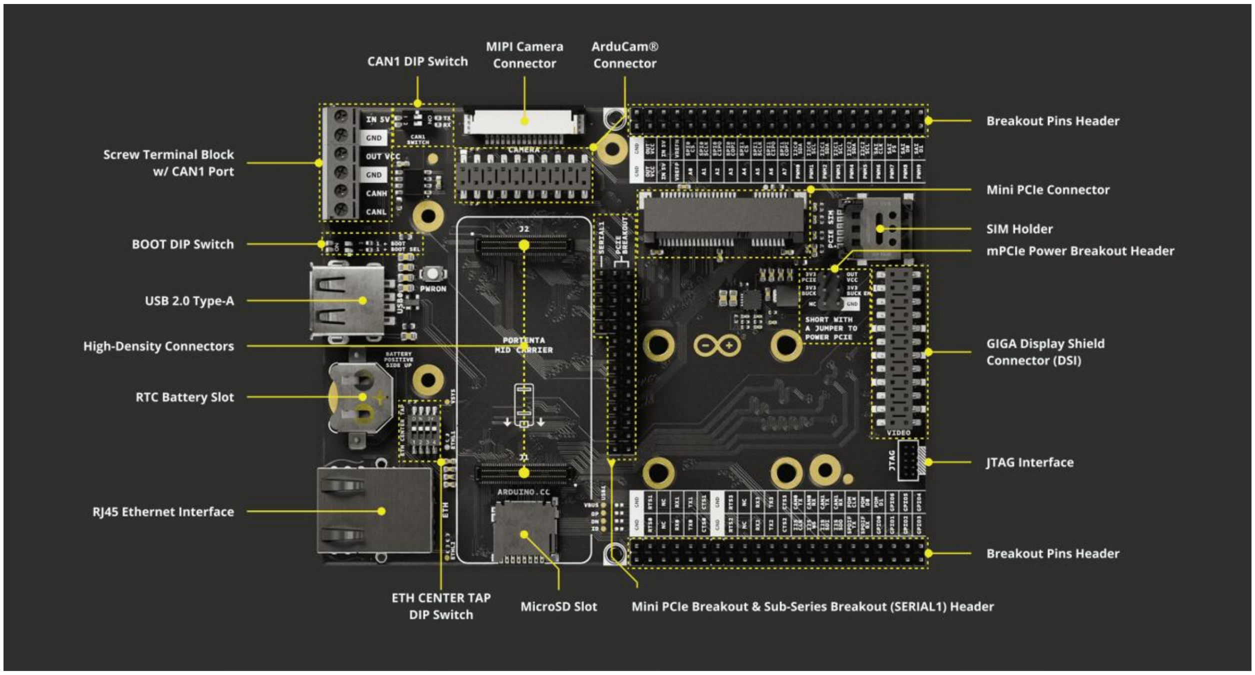


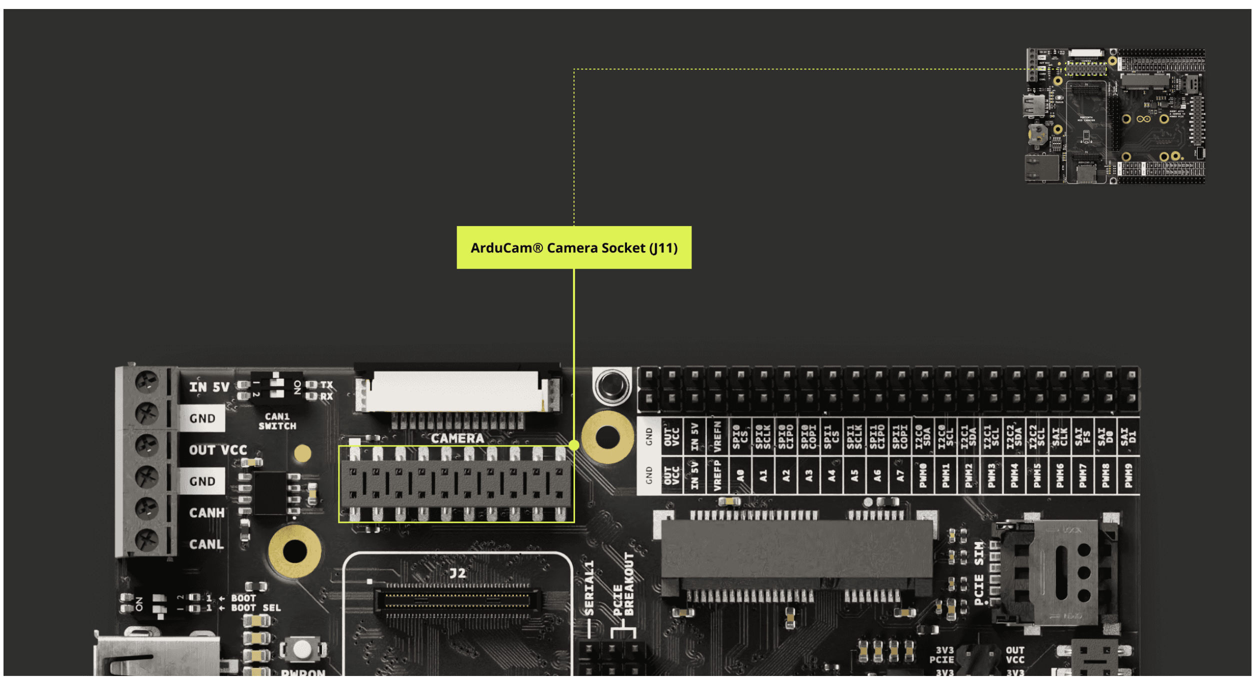
Parameter	Specification
Microcontroller	STM32H747XI dual-core: Cortex-M7 @ 480 MHz, Cortex-M4 @ 240 MHz
Memory	8 MB SDRAM, 16 MB QSPI Flash
Operating Voltage	3.3 V
Power Supply	5 V via USB-C or VIN; supports single-cell Li-Po battery (3.7 V, ≥700 mAh)
Current Consumption	2.95 µA in standby mode (Backup SRAM OFF, RTC/LSE ON)
Connectivity	Wi-Fi 802.11b/g/n (65 Mbps), Bluetooth 5.1 (Classic & BLE), 10/100 Ethernet
USB Interface	USB-C: Host/Device, DisplayPort out, High/Full Speed, Power Delivery
Display Interface	MIPI DSI host & MIPI D-PHY (Portenta H7 only)
Graphics Accelerator	Chrom-ART Accelerator™, JPEG encoder/decoder
Analog Inputs (ADC)	3× ADCs, up to 36 channels, 16-bit resolution, up to 3.6 MSPS
Analog Outputs (DAC)	2× 12-bit DACs (1 MHz), one accessible via A6 pin
Timers	22 timers and watchdogs
UART Ports	4× UARTs (2 with flow control)
Secure Element	ECC608 or NXP SE050C2 (Common Criteria EAL 6+), depending on variant
Operating Temperature	−40 °C to +85 °C (−40 °F to +185 °F)
Form Factor	Arduino MKR-compatible; includes two 80-pin high-density connectors
Expansion Interfaces	MKR headers, high-density connectors, ESLOV connector
Camera Interface	8-bit parallel, up to 80 MHz

The Portenta is a highly versatile microcontroller! The primary features we will be using are as follows:

1. M7 Core with 2 MB of Flash Allocated
2. Bank 0 80-Pin High Density Connector (Mid Carrier DVI Port)
3. M7 Hardware Abstraction Layer (Configuring Custom GPIO Clocks)

Portenta Mid Carrier





Pin Number	**Silkscreen Pin**	**Power Net**	**Portenta Standard Pin**	**High-Density Pin**
1	VCC	+3V3 Portenta (Out)	VCC	J2-23, J2-34, J2-43, J2-69
2	GND	Ground	GND	J1-22, J1-31, J1-42, J1-47, J1-54, J2-24, J2-33, J2-44, J2-57, J2-70
3	SCL0		I2C0_SCL	J1-46
4	SDA0		I2C0_SDA	J1-44
5	VSYNC		CAM_VS_CK_P	J2-18
6	HREF		CAM_HS	J2-22
7	PCLK		CAM_CK_CK_N	J2-20
8	XCLK		PWM_0	J2-59
9	DOUT7		CAM_D7_D3_P	J2-2
10	DOUT6		CAM_D6_D3_N	J2-4
11	DOUT5		CAM_D5_D2_P	J2-6
12	DOUT4		CAM_D4_D2_N	J2-8
13	DOUT3		CAM_D3_D1_P	J2-10
14	DOUT2		CAM_D2_D1_N	J2-12
15	DOUT1		CAM_D1_D0_P	J2-14
16	DOUT0		CAM_D0_D0_N	J2-16
17	PWRENABLE		GPIO_3	J2-52
18	PWDN		GPIO_4	J2-54
19	PWRENABLE		GPIO_3	J2-52
20	PWDN		GPIO_4	J2-54

The ArduCam socket (J11) socket will be the interface we are utilizing to address our camera. Note that HAL is utilized to drive XCLK to anywhere between 18-42 Mhz for the camera register latching.

<u>*OV7670*</u>

Parameter	Specification
Sensor Type	CMOS image sensor
Sensor Model	OV7670
Resolution	VGA (640 × 480 pixels)
Pixel Size	3.6 µm × 3.6 µm
Image Array Size	656 × 488 pixels (active: 640 × 480 pixels)
Operating Voltage	2.5 V to 3.0 V (Digital I/O: typically 3.3 V)
Frame Rate	Up to 30 fps (VGA mode), higher in QVGA mode
Output Format	YUV/YCbCr422, RGB565/555/444, GRB 4:2:2, Raw RGB
Interface	SCCB (Serial Camera Control Bus, similar to I²C)
Clock Frequency (XCLK)	10 MHz to 24 MHz (typical 12 MHz to 16 MHz)
Sensitivity	1.3 V/(Lux-sec)
Signal-to-Noise Ratio (SNR)	46 dB
Lens Mount	Fixed lens (typically standard M6 or integrated)
Exposure Control	Automatic exposure and gain control
White Balance	Automatic white balance control
Power Consumption	Approx. 60 mW at 15 fps
Operating Temperature	-30 °C to +70 °C

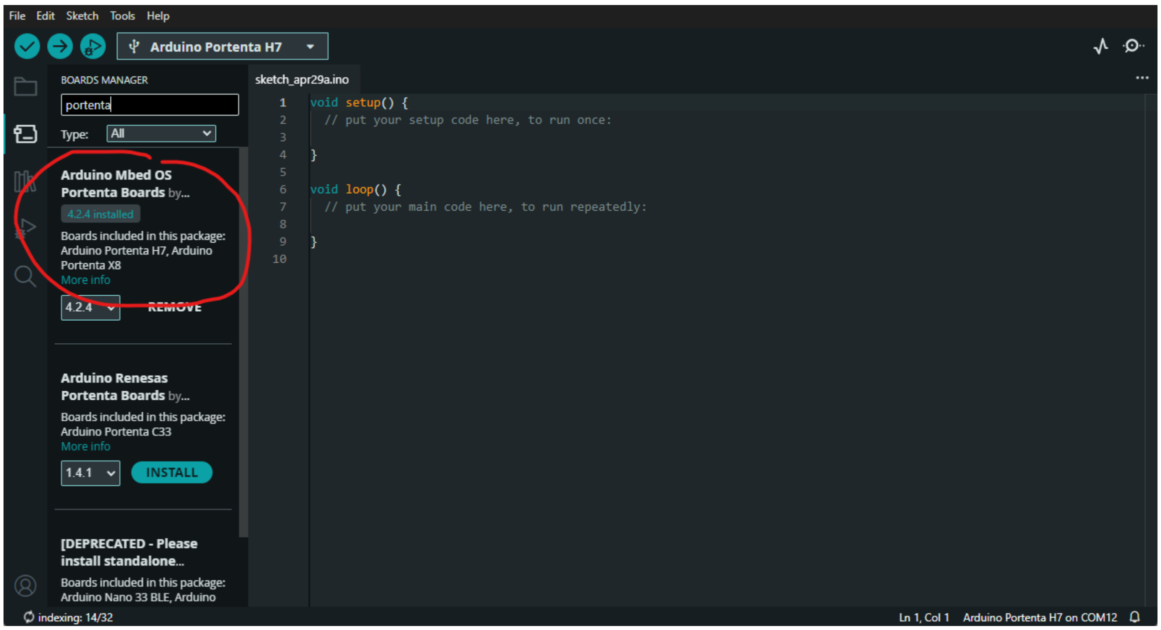
For this project, we will be using the OV7670 in the following configuration:

1. QVGA (320x240)
2. RGB565
3. 30 FPS

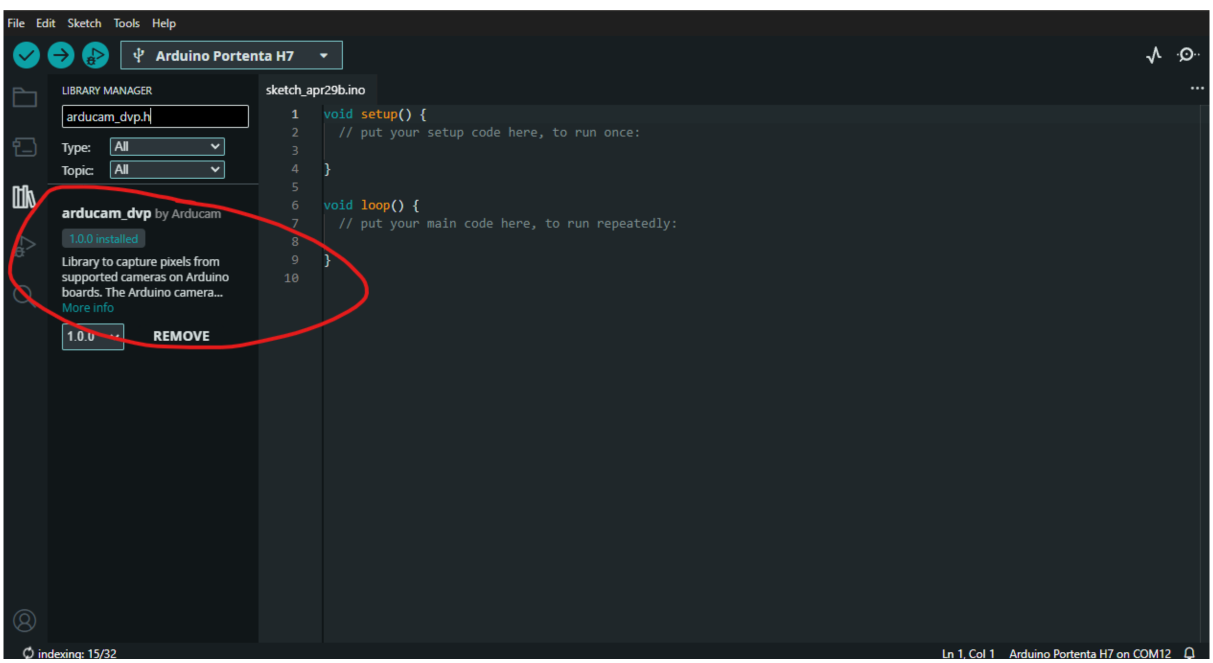
Setting Up the Portenta

The Portenta H7 is most easily programmed using the standard Arduino IDE as it provides the standard libraries for flashing the H7 as well as associated libraries for the camera and HAL.

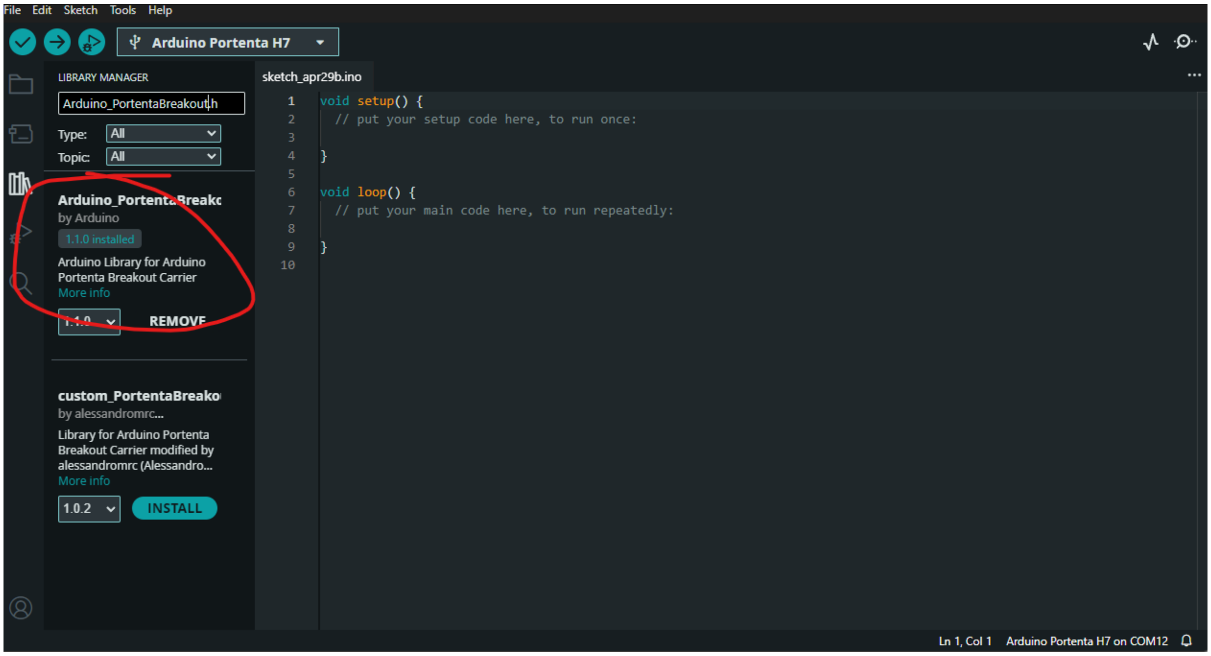
1. Download Arduino IDE
    a. https://www.arduino.cc/en/software/
2. Install Project Libraries
    a. Mbed OS for Portenta H7



    b. Arducam_dvp.h



c. Arduino_PortentaBreakout.h

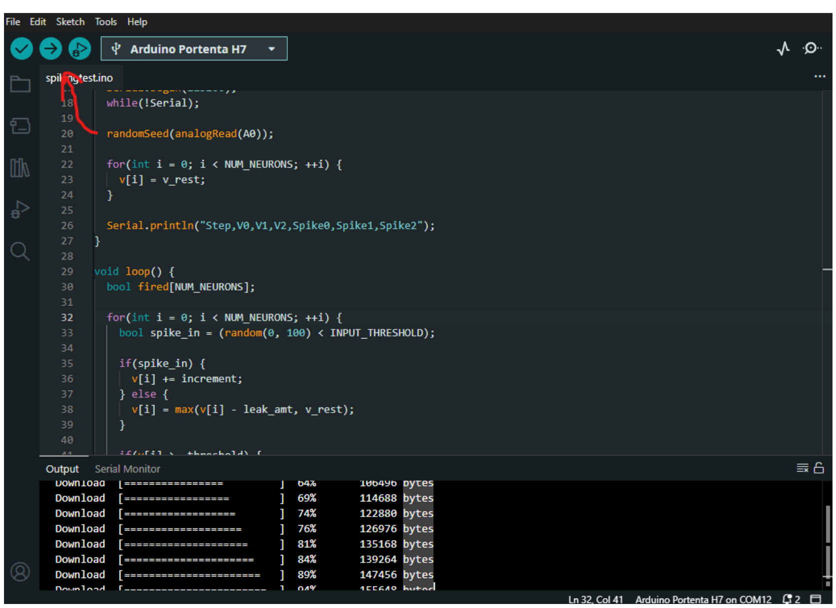


That's all of the headers we need! Arducam provides us the necessary register addresses and logic to address our OV7670 sensor. However, it is not capable of driving the breakout boards GPIO for the necessary clock register (XCLK). That is addressed using HAL and the Breakout header files. You will see this explained more thoroughly in the code, however for now we will focus on getting everything set up.

Next, we will explore an example SNN on the Portenta to validate operational ability. At this point you should ensure you have pulled the associated repository off GitHub for this project

→ https://github.com/INQUIRELAB/SNNGesturesPortentaH7

3. Load your first sketch to Portenta!
    a. Access the sketch folder in the directory .\spiking test



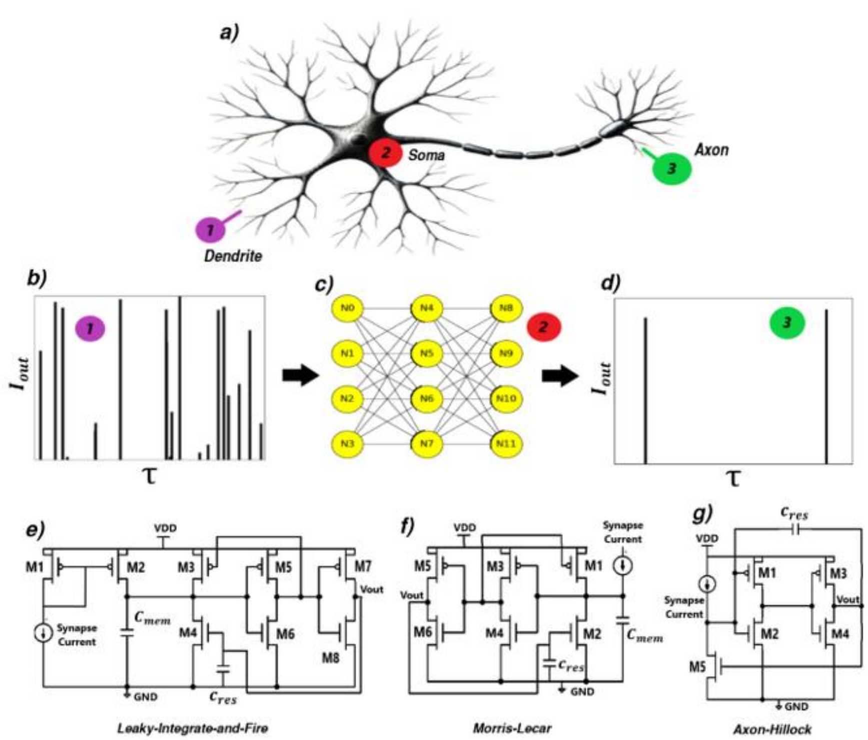
You might see a dfu suffix error. This is normal, msobed is actually designed for the X7 with onboard linux capabilities. This won't affect our translation of logic on the M7 Core we are writing to though.

b. Do you have results?

```
22:50:50.504 -> 3170,0.200,0.993,0.298,0,0,0
22:50:50.582 -> 3171,0.300,0.000,0.398,0,1,0
22:50:50.721 -> 3172,0.400,0.100,0.397,0,0,0
22:50:50.815 -> 3173,0.399,0.099,0.396,0,0,0
22:50:50.909 -> 3174,0.398,0.098,0.395,0,0,0
22:50:51.002 -> 3175,0.397,0.198,0.495,0,0,0
22:50:51.079 -> 3176,0.396,0.298,0.494,0,0,0
22:50:51.204 -> 3177,0.395,0.398,0.493,0,0,0
22:50:51.296 -> 3178,0.394,0.397,0.492,0,0,0
22:50:51.381 -> 3179,0.494,0.396,0.491,0,0,0
22:50:51.513 -> 3180,0.493,0.395,0.490,0,0,0
22:50:51.605 -> 3181,0.492,0.394,0.489,0,0,0
22:50:51.680 -> 3182,0.491,0.49
```

System Clock	Time Stamp	Neuron 1	Neuron 2	Neuron 3	N1 Spikes	N2 Spikes	N3 Spikes

If you see this screen we are in the money. Let's discuss what those results actually mean though. If your new to Spiking Neural Networks, they are composed of biologically inspired neurons that perform L-I-F. Also known as leak, integrate, and fire. Similar to cortical neurons, we seek a behavioral pattern to which the neurons become excited from pulses of current asserted by preceding neurons. However, if the activity is too low, we don't want to spike. This is what allows us to extract features about the temporal information of datasets.



The image above contains some great graphics for explaining the architecture of LIF neurons alongside some examples of transistor-based structures. Figures b-d take it a step further by focusing on how input spikes translate to output spikes via the three-layer neural network in shown in c.

Setting Up an Anaconda Environment

For some of the code in this program you must install anaconda to your pc. I recommend miniconda as it runs much faster when installing new packages.

Download anaconda → https://www.anaconda.com/download

You will need to install some packages to get this code off the ground → **“pip install opencv-python numpy matplotlib numba pyserial”**

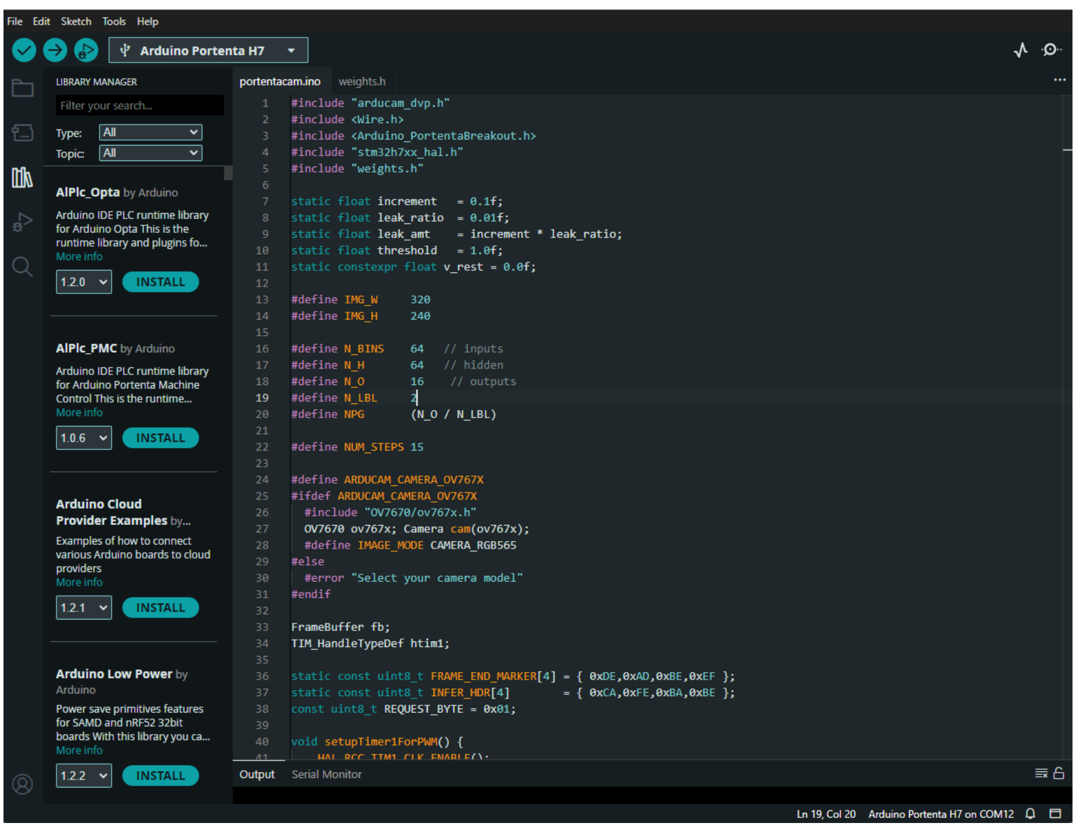
And we also need a specific version of another package → **“pip install mediapipe==0.10.13”**

In the next three sections, I will focus on one primary required step that uses the pretrained model in the repository. The other two sections include optional steps for collecting your own data, and training on that data.

Setting Up Your Inquire Labs SNN Camera Hardware and Loading the Model

Provided with this project is a pre-made solution to get straight to loading the model onto the Portenta and calling on it from the python interface provided.

You will need to access the following Arduino sketch → .\portentacam\portentacam.ino



Our code is broken up into a couple of blocks.

1. SNN Hyperparameters
2. Load weights from weights.h (.\portentacam\weights.h)
3. HAL (Hardware Abstraction Layer) configuration for XCLK at ~30 Mhz
4. Arducam configuration logic
5. SNN time step analysis
6. Serial interface

Snn Hyperparameters

```
static float increment   = 0.1f;
static float leak_ratio  = 0.01f;
static float leak_amt    = increment * leak_ratio;
static float threshold   = 1.0f;
static constexpr float v_rest = 0.0f;

#define IMG_W     320
#define IMG_H     240

#define N_BINS    64   // inputs
#define N_H       64   // hidden
#define N_O       16    // outputs
#define N_LBL     2
#define NPG       (N_O / N_LBL)

#define NUM_STEPS 15
```

These are the recommended configurations for the pre-trained weights provided. If you train a different set of weights, you may have to modify some of the parameters. N-Bins and N-H will always match. The binning is performed on the Radially Invariant Fourier Analysis of the hands detected. Output layer neurons are dependent on the sizing set by the weights. Exploring with the increment and leak ratio is not necessary as they can be fine tuned through commands sent by the host device.

Load Weights

```
portentacam.ino   weights.h

// Auto-generated by prepweights.py
#pragma once

// Hidden layer size
#define N_H 128
// Output layer size
#define N_O 32

// W1: flat, one weight per hidden neuron (length = N_H)
const float W1_data[N_H] = {
  1.000000f,  4.380000f,  30.000000f,  11.110000f,  30.000000f,  27.459999f,  30
  30.000000f,  9.630000f,  30.000000f,  22.840000f,  30.000000f,  9.440000f,  30
  0.140000f,  15.130000f,  30.000000f,  1.770000f,  30.000000f,  24.500000f,  30
  30.000000f,  12.650000f,  30.000000f,  6.630000f,  30.000000f,  25.510000f,  3
  30.000000f,  23.950001f,  30.000000f,  26.320000f,  30.000000f,  4.200000f,  3
  30.000000f,  13.630000f,  1.170000f,  26.750000f,  30.000000f,  4.650000f,  30
  2.040000f,  25.870001f,  30.000000f,  7.950000f,  30.000000f,  16.559999f,  1.
```

```
int infer_and_groups(const float x[N_BINS], int grp[N_LBL]) {
  static bool seeded = false;
  if(!seeded){
    randomSeed(micros());
    seeded = true;
  }

  float v_h[N_H], v_o[N_O];
  int   os_h[N_H] = {0}, os_o[N_O] = {0};
```

HAL XCLK Configuration

```
void setupTimer1ForPWM() {
  __HAL_RCC_TIM1_CLK_ENABLE();
  __HAL_RCC_GPIOA_CLK_ENABLE();
  GPIO_InitTypeDef g = {};
  g.Pin       = GPIO_PIN_8;
  g.Mode      = GPIO_MODE_AF_PP;
  g.Pull      = GPIO_NOPULL;
  g.Speed     = GPIO_SPEED_FREQ_VERY_HIGH;
  g.Alternate = GPIO_AF1_TIM1;
  HAL_GPIO_Init(GPIOA, &g);

  htim1.Instance = TIM1;
  htim1.Init.Prescaler     = 1;
  htim1.Init.CounterMode   = TIM_COUNTERMODE_UP;
  htim1.Init.Period        = 6;
  htim1.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
  HAL_TIM_PWM_Init(&htim1);

  TIM_OC_InitTypeDef s = {};
  s.OCMode     = TIM_OCMODE_PWM1;
  s.Pulse      = 4;  // 50% duty @ 30 MHz
  s.OCPolarity = TIM_OCPOLARITY_HIGH;
  s.OCFastMode = TIM_OCFAST_ENABLE;
  HAL_TIM_PWM_ConfigChannel(&htim1, &s, TIM_CHANNEL_1);
  HAL_TIM_PWM_Start(&htim1, TIM_CHANNEL_1);
}

void blinkLED(uint32_t cnt=0xFFFFFFFF, uint32_t d=50) {
  while(cnt--) {
    digitalWrite(LED_BUILTIN, LOW);  delay(d);
    digitalWrite(LED_BUILTIN, HIGH); delay(d);
  }
}
```

This code is necessary for the camera, and is unique to this repo. We found a way to get the portenta DVI interface to work with any camera that has the standard dvi port interface. Using TIM, we address the PWM1 channel with a 30 Mhz clock frequency. This does not need to be changed. The traces on the breakout board do not allow for higher frequencies due to low SNR.

Arducam Configuration Logic

```
#define ARDUCAM_CAMERA_OV767X
#ifdef ARDUCAM_CAMERA_OV767X
  #include "OV7670/ov767x.h"
  OV7670 ov767x; Camera cam(ov767x);
  #define IMAGE_MODE CAMERA_RGB565
#else
  #error "Select your camera model"
#endif
```

SNN Time Step Analysis

```
for(int t = 0; t < NUM_STEPS; ++t) {
  for(int p = 0; p < N_BINS; ++p) {
    bool spike_in = (random(0,10000) < x[p] * 10000.0

    if(!spike_in) {
      for(int h=0; h<N_H; ++h)
        v_h[h] = fmaxf(v_h[h] - leak_amt, v_rest);
    } else {
      int gs   = N_H / N_BINS;
      int rem = N_H - gs * N_BINS;
      int extra   = (p < rem) ? 1 : 0;
      int start_h = p*gs + min(p,rem);
      int end_h   = start_h + gs + extra;
      for(int h=start_h; h<end_h; ++h)
        v_h[h] += increment * W1ptr[h];
    }

    bool fired_h[N_H];
    bool any_h = false;
    for(int h=0; h<N_H; ++h) {
      if(v_h[h] >= threshold) {
        fired_h[h] = true;
        os_h[h]   += 1;
        v_h[h]     = v_rest;
        any_h      = true;
      } else {
        fired_h[h] = false;
      }
    }

    if(any_h) {
      for(int o=0; o<N_O; ++o) {
        float acc = 0.0f;
        for(int h=0; h<N_H; ++h)
          if(fired_h[h])
            acc += W2ptr[o * N_H + h];
        v_o[o] += increment * acc;
      }
    } else {
      for(int o=0; o<N_O; ++o)
        v_o[o] = fmaxf(v_o[o] - leak_amt, v_rest);
    }
```

There are two things going on here. First we generate our spikes based on the Radially Invariant Fourier Analysis. This analysis is performed on the host side vs. the image due to the serial buffer on the Portenta only supporting enough bandwidth for holding the image data once. An FFT produces a number of datapoints similar to the size of the image. The Radially Invariant Fourier Analysis is performed on our data because it allows for the spectral behavior of hand gestures to be independent of the orientation of your hand.

Once the coefficients are processed on the host side, they are parsed back into the SNN running on the Arduino. These coefficients are then correlated to respective input neurons that are driven by a spike train stochastically generated by a probability matrix. The probability of each input neuron's spikes being generated is related by the amplitude of their respective a_k coefficient from the FFT.

Serial Interface

```
      increment  = inc;
      leak_ratio = leak;
      leak_amt   = increment * leak_ratio;
      Serial.println("OK");
    } else Serial.println("ERR");
    return;
  }
  if(c == 'B'){
    String line = Serial.readStringUntil('\n');
    float feats[N_BINS];
    char buf[256];
    line.toCharArray(buf, sizeof(buf));
    int idx = 0;
    char *tok = strtok(buf + 5, ",");
    while(tok && idx < N_BINS){
      feats[idx++] = atof(tok);
      tok = strtok(NULL, ",");
    }
    int grp[N_LBL];
    int lbl = infer_and_groups(feats, grp);

    Serial.print("GROUPS:");
    Serial.print(grp[0]);
    Serial.print(',');
    Serial.println(grp[1]);

    Serial.write(INFER_HDR, sizeof(INFER_HDR));
    Serial.write((uint8_t*)&lbl, 1);
    Serial.flush();
    return;
  }
}

if(Serial.read() != REQUEST_BYTE){
  blinkLED(1,10);
  return;
}

if(cam.grabFrame(fb, 3000) == 0){
  Serial.write(fb.getBuffer(), cam.frameSize());
  Serial.write(FRAME_END_MARKER, sizeof(FRAME_END_MARKER));
} else {
```

The serial interface contains the unique signatures that our python program knows to look for, in order to find the camera as well as send commands to it. We later discuss how these commands are set up.

It's now time to upload the code to your Portenta! Send it over, and once that's done perform the following steps to collect information about what your camera is seeing.

For the next steps we will launch the camera read logic and discuss how to set up the hyperparameters necessary to get similar results to what is seen experimentally on the pre-trained model.

Open the following python file → .\cameraread2.py

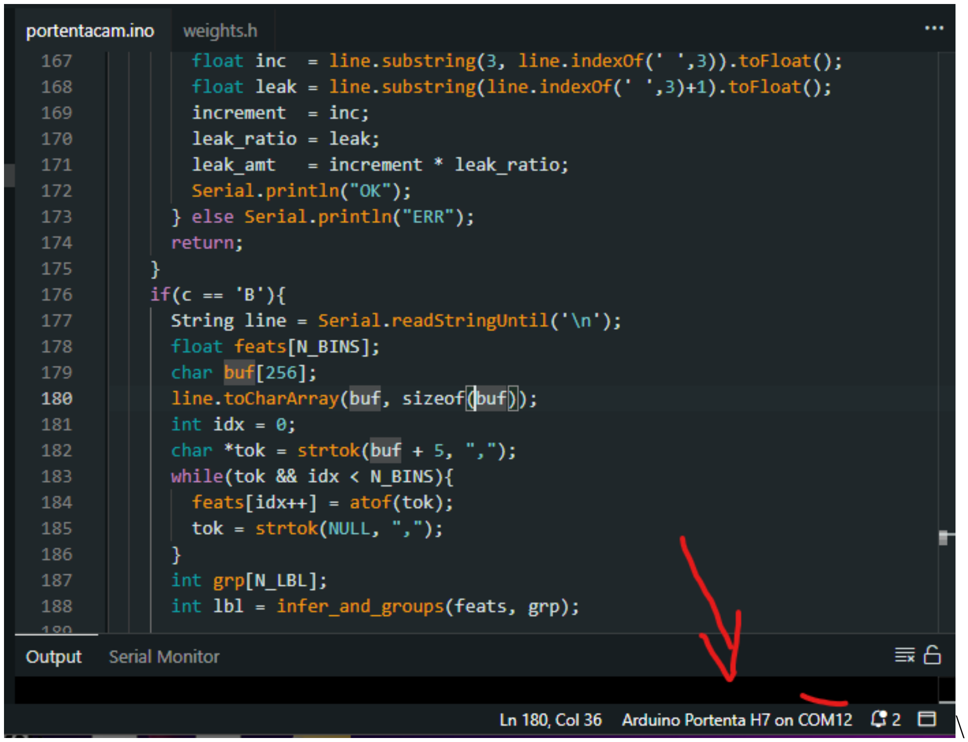
The only thing we may need to modify is the number of inputs parameter:

```
INPUTS = 64  # must match your Arduino (FFT Bins)
```

And you may also consider changing the Fourier descriptor temporal dampening if your testing the camera in a particularly noisy environment:

```
# Buffers & indices
buf_g0, buf_g1 = deque(maxlen=5), deque(maxlen=5)
bins_buffer    = deque(maxlen=1)
start, end     = INPUTS//4, INPUTS*3//4
```

Once your code is correct, we need to identify which communication port your Arduino Portenta H7 is on. This can be done in Arduino IDE.



portentacam.ino   weights.h

```
            float inc  = line.substring(3, line.indexOf(' ',3)).toFloat();
            float leak = line.substring(line.indexOf(' ',3)+1).toFloat();
            increment  = inc;
            leak_ratio = leak;
            leak_amt   = increment * leak_ratio;
            Serial.println("OK");
        } else Serial.println("ERR");
        return;
    }
    if(c == 'B'){
        String line = Serial.readStringUntil('\n');
        float feats[N_BINS];
        char buf[256];
        line.toCharArray(buf, sizeof(buf));
        int idx = 0;
        char *tok = strtok(buf + 5, ",");
        while(tok && idx < N_BINS){
            feats[idx++] = atof(tok);
            tok = strtok(NULL, ",");
        }
        int grp[N_LBL];
        int lbl = infer_and_groups(feats, grp);
```

Output   Serial Monitor

Ln 180, Col 36   Arduino Portenta H7 on COM12   2

Once you verify the COM port in use, update the following logic in cameraread2.py

```
PORT         = 'COM12'
BAUDRATE     = 12500000
WIDTH        = 320
HEIGHT       = 240
FRAME_SZ     = WIDTH * HEIGHT * 2
END_MARKER   = b'\xDE\xAD\xBE\xEF'
REQUEST_BYTE = b'\x01'
INFER_HDR    = b'\xCA\xFE\xBA\xBE'
```

Now you can launch the camera! While calling on the program in Anaconda, consider the following flags you can pass.

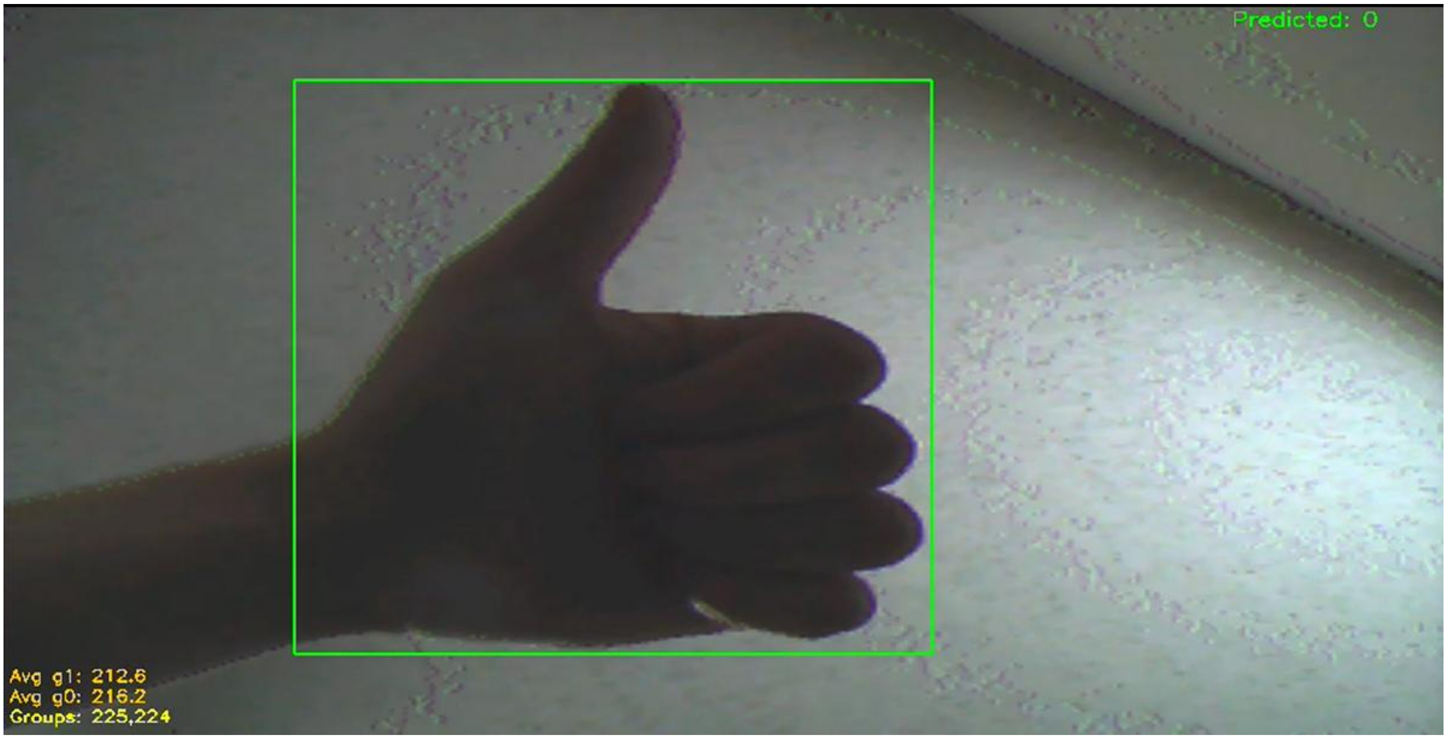
- --inc # (changes spiking current)
- --leak # (changes membrane leakage rate)
- --amp-mid # (amplifies mid range frequencies of the FFT analysis)
- --alpha # (changes contrast of video)
- --beta # (changes brightness of video)
- --video-only (don't perform inference if you want to record for training data)

I recommend the following python call based on the pre-trained models working dynamics.

- python cameraread2.py –inc 0.078 –leak 0.01 –amp-mid 1.3

After running that line of code, you should see the following.

<u>Label 0 (Thumbs Up)</u>



<u>Label 1 (Surfer Dude)</u>

At this point you have successfully set up the gesture recognition. Further features can be added with additional training and data collection. That will be discussed in the following sections.

Data Collection

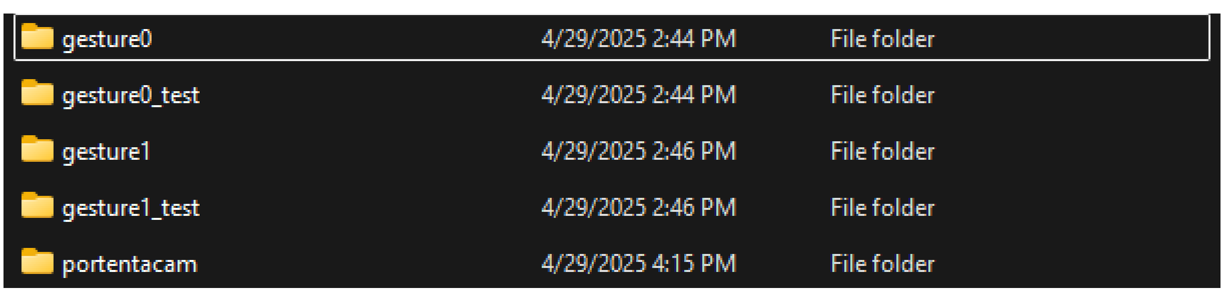
In whatever environment you wish to record gestures, you may do so utilizing the same cameraread.py with the following call: → python cameraread2.py –video-only

In this video only mode, you can use your PCs built in screen recorders to capture footage of whatever gesture you are wanting to train on. After this I have included some built in python scripts to process the data.

Make the following call to break your footage down into individual images that are normalized to the resolution of the camera → python videotoimage.py –start-index 0 "video file name" "target directory"

Next you need to package the images into folders of the project directory in the following format.

Gesture	Folder Name
0	gesture0
0 Test	gesture0_test
# (1, 2, 3)	gesture#
# (1, 2, 3)	gesture#_test



Be sure to split your images to have >=50% of them in the test folder. Now we are ready to explore training!

Training

Training on your gesture data can be daunting, but with patience and careful tuning you will succeed. The training parameters we use in pbtalgo.py are as follows:

Parameter	Value	Description
Network Structure		
n_hidden	64	Number of hidden neurons
n_output	16	Number of output neurons
num_labels	2	Number of gesture classes
neurons_per_group	8	Output neurons per label (n_output / num_labels)
Neuron Dynamics		
increment	0.1	Voltage increase on input spike
threshold	1.0	Firing threshold
v_rest	0.0	Resting potential
leak_ratio	0.01	Leak as fraction of increment
leak_amt	0.001	Absolute leak per time step (increment × leak_ratio)
STDP Parameters		
eta_w2	0.0001	Learning rate for W2
target_true	200	Target output spikes for correct label
target_false	5	Target output spikes for incorrect labels
punish_factor	35	Punishment scaling for incorrect spikes
w_min, w_max	0, 30	Min and max bounds for W2 weights
Training Settings		
epochs	500	Total training epochs
exploit_every	4	Exploitation interval in PBT
n_elites	6	Number of elite models in PBT
Input Encoding		
inputs	64	Number of input features (same as n_hidden)
num_bins	64	Number of bins for spike input
W1 Init Parameters		
eta_w1	0.05	Learning rate for W1 pretraining
W1_min_init	0.0	Initial min for W1
W1_max_init	30	Initial max for W1
Other Settings		
sensitivity_init	0	Initial sensitivity
data_root	"./"	Dataset root directory

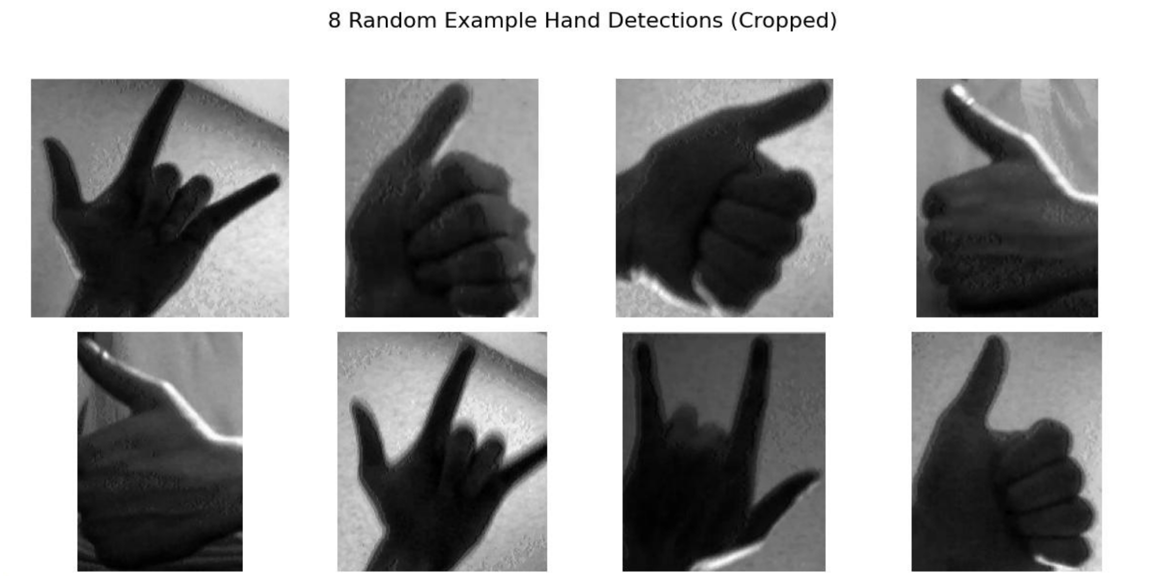
All of these settings have unique features but I want you to focus on the target firing rates, punishment factor, and the network size. You may also consider playing with the leakage and increment rates. There is no distinct arrangement of settings that work well for any given set of data, so you will have to explore what works well. The above parameters is what was used to pre-train the provided model.

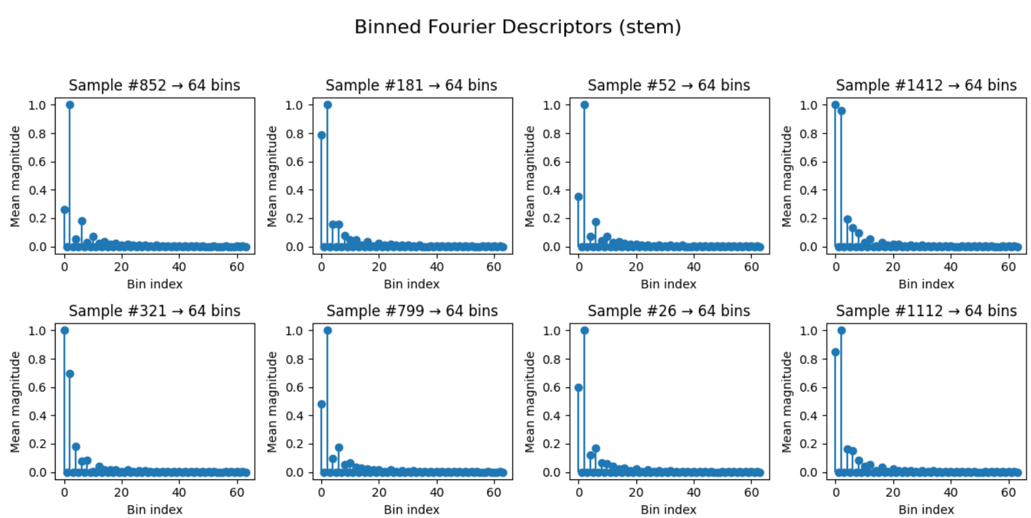
To run the program, you will want to use the following system call.

- ➔ python pbtalgo.py –w1-file w164.csv –w2-file w264.csv –steps 100

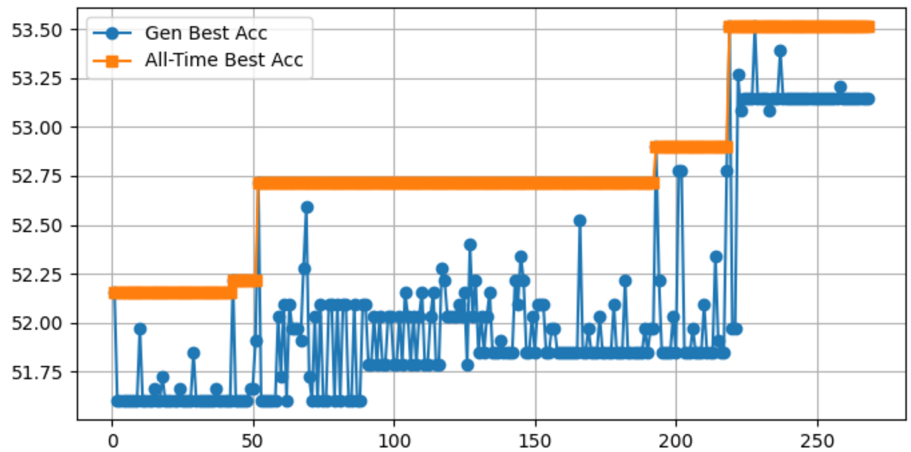
Note that some of the flags are dependent on your network size. The naming convention I like to use is w(layer)(num of neurons).csv. I addition, I have found 100 steps to provide plenty of temporal information for training. Note though, that your number of steps does influence the leakage and increments and could have an effect on accuracy. This program will run, display some example images, save the weights incrementally, and package w1 and w2 for further processing. Furthermore, you should know that the program uses caching for subsequent runs. If you change your datasets you will need to delete the .pak file. Once w1 is trained, a w1 file is saved to where the next time the program runs; w1 training will not have to be performed again.

Example images:





```
W1 sample 591: Δmin 0.000000, Δmax 0.050000, Δavg 0.000038
W1 sample 1116: Δmin 0.000000, Δmax 0.050000, Δavg 0.000061
W1 sample 987: Δmin 0.000000, Δmax 0.050000, Δavg 0.000039
W1 sample 592: Δmin 0.000000, Δmax 0.050000, Δavg 0.000044
W1 sample 1117: Δmin 0.000000, Δmax 0.050000, Δavg 0.000079
W1 sample 988: Δmin 0.000000, Δmax 0.050000, Δavg 0.000071
W1 sample 593: Δmin 0.000000, Δmax 0.050000, Δavg 0.000048
W1 sample 1118: Δmin 0.000000, Δmax 0.050000, Δavg 0.000051
  After epoch 2: W1 Δavg 15.257673
 W1 epoch 3/10
```



After your training is complete, we need to convert the weights to a C compatible header file. To do this you must use the following

➔ python prepweights.py

Depending on the naming convention of your weights you will need to update the code in that file to point to your new weight files.

Lastly, you simply move the new weights file into the Arduino project, update the Arduino code to reflect your network size; and upload! Note that you will need to change the inputs parameter in cameraread2.py to reflect your network size. → Your off to the races!

Conclusion

By the end of this project you will have explored SNN based gesture recognition with an emphasis on implementation within edge devices such as the Portenta H7. We also discussed some of the factors regarding how this program works, and what optimizations can be made. Teaching your SNN should prove to be a fun and engaging experience!