Implementación de modelo de inteligencia artificial para procesamiento de audio usando una tarjeta de desarrollo 3 MINI PROYECTO

Víctor Daniel Álvarez (2181052), Juan Pablo Agudelo Barajas (2175741)

Resumen – En este trabajo se busca la implementación de un modelo de inteligencia artificial para procesamiento de audio usando una tarjeta de desarrollo, esto con fines de generar una interacción para el hogar, así se logra controlar el aire acondicionado y tres luces de la casa a través de una interfaz similar a un asistente de hogar, usando el micrófono integrado en el nano 33 ble, como el medio para ejecutar las órdenes. Así en este documento se muestra la descripción del proceso con el que se desarrolló y la jerarquía de las órdenes.

Índice de Términos – Nano 33 ble, domótica, Edge Impulse, Arduino uno.

I. INTRODUCCIÓN

En la actualidad cada vez se hacen más presente las distintas tecnologías en los hogares, ejemplos claros tenemos a los asistentes de hogar como Alexa. Y llegando a abarcar empresas tan grandes en desarrollo y construcción como las guiadas a la domótica. Un informe presentado por Camacol en el Foro Biocasa 2021 resaltó que Colombia se estaba posicionando como líder mundial en construcción sostenible tras haber logrado que más de 3 millones de metros cuadrados en el país recibieran certificación Excellence in Design For Greates Efficiencies (EDGE), eso significó 201 proyectos, de los cuales 138 son residenciales y equivalen a 51.807 unidades de vivienda. Según el portal Statista, esta es una tendencia de vivienda que viene creciendo a pasos inusitados, ya que se prevé que para 2022 el mercado de la domótica genere ingresos superiores a los US$53.000 millones a nivel mundial.

Pero esto no es posible si no fuera por la creación de sistemas computacionales e inteligencia artificial que son capaces de procesas la voz humana para la ejecución de una acción a través de una palabra clave, este método es conocido como KWS o keyword Spotting. Este es un sistema de detección de palabras clave que hace uso de un reconocimiento de voz para identificar las palabras claves y frases con una arquitectura especial en el que se procesa continuamente la voz recibida, esto en un estado de bordo por lo que no se activara el código de ejecución hasta que se escuche la palabra clave definida.

Así los avances recientes en el aprendizaje profundo han convertido a KWS en un tema candente para el reconocimiento de voz, el aprendizaje profundo ha mejorado tanto la precisión como la eficiencia de los modelos de IA, por lo tanto, KWS está disponible incluso para pequeños dispositivos electrónicos no navegadores web.

Gracias a esta portabilidad, nos podemos dar cuenta de que la aplicación de este tiempo de detección en sistemas embebidos puede resultar en una aplicación física interesante. Recordando antes que los sistemas embebidos, también llamados sistemas empotrados son sistemas operativos creados con el fin de ser controlados por microprocesadores o microcontroladores, por lo que dichos sistemas son llevados a un proceso sistematizado que cumplen una tarea especifico, siendo programados para hacer esa única acción.

Así para generar implementaciones físicas que no requieren hacer múltiples acciones, pero a su vez debe de estar continuamente funcionando, el mejor componente para dicho método es el nano 33 ble.

Después de todo este pequeño microcontrolador nos ofrece diferentes funciones como cámara, sensores de velocidad, inerciales y lo que más requerimos, un micrófono. Así el producto final de este documento es la creación de un modelo que se asemeje a un método KWS, para así aplicarlo en una imitación de "Alexa" para el encendido de la ventilación, luz de la casa, luz de la sala, luz de patio y todas las luces si el usuario lo requiere.

OBJETIVOS DEL PROYECTO

OBJETIVO GENERAL: Desarrollar una metodología para el diseño de productos en la ingeniería basándonos en los objetivos 9 y 11 de desarrollo sostenible.

- **OBJETIVOS ESPECIFICOS**

  - Investigar metodologías de diseño en la ingeniería.
  - Analizar información obtenida.
  - Realizar documento guía con la metodología.
  - Copilar y aplicar la teoría vista en un caso práctico.

II. DESCRIPCIÓN DEL PROYECTO

Como se menciono antes, se busca que este proyecto realice varias acciones a través del uso del comando de voz, pero debido a que no tenemos el medio para realizar una aplicación completa de KWS, se opto por hacer una versión que simulará las tres etapas del KWS, esta etapa será:

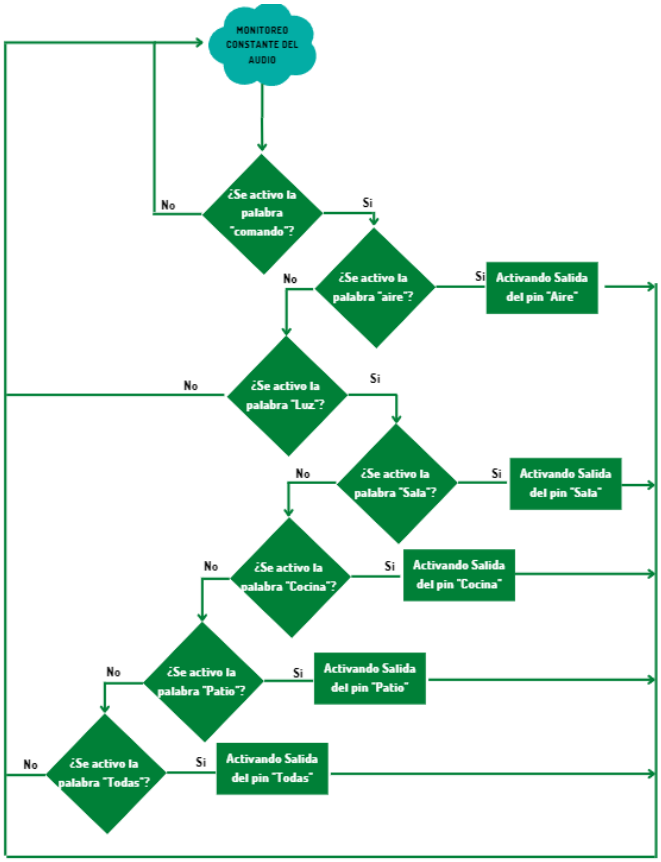
Inicio de orden: Cuando la persona diga la palabra comando, se activará una voz que indica que se encuentra activa las demás ordenes, mientras que esto no pase no se deberá activar ninguna acción, esto se puede llamar como una palabra de seguridad para el modelo.

Ordenar acción: después de la palabra clave, se tendrá que ejecutar la acción que se asocia con la palabra clave, pero el

modelo debe de reconocer si esa acción ya se realizo con anterioridad y aun se encuentra en funcionamiento dicha acción.

Apagar y repetir: Por ultimo el modelo debe de poder saber cuando todas las acciones se encuentran apagadas, así pues, gracias a esto se busca que el modelo reconozco que ya se puede repetir el ciclo y volver a solicitar la misma acción.

Así para entender de mejor manera la arquitectura del modelo, realizaremos el siguiente diagrama de bloques:



Con esto tenemos nuestra arquitectura que vamos a usar en el desarrollo del proyecto, aclarando que se busca un modelo secuencial por etapas, aunque se puede decir que es un modelo tosco al estar acondicionado, tiene la gran ventaja de poseer una mayor robustes a modelos mas libres de elecciones, por lo que el margen de error se reduce considerablemente.

INSPIRACIÓN PARA EL DESARROLLO DEL PROYECTO:

Como es sabido, el producto distintivo por excelencia en la Mecatrónica son los robots y la inteligencia artificial aplicada a la funcionalidad de estos, teniendo esto en cuenta se pensó en la domótica como la mejor forma de finalizar el curso en cuestión. La tecnología es mejorada progresivamente para hacer la vida humana más sencilla y aunque cada vez parecían haber más comodidades, no fue sino hasta que las máquinas podían hacer los procesos domésticos casi con total autonomía que se alivió una gran carga para las familias que tuviesen el poder adquisitivo que permite la compra e instalación de estos dispositivos.

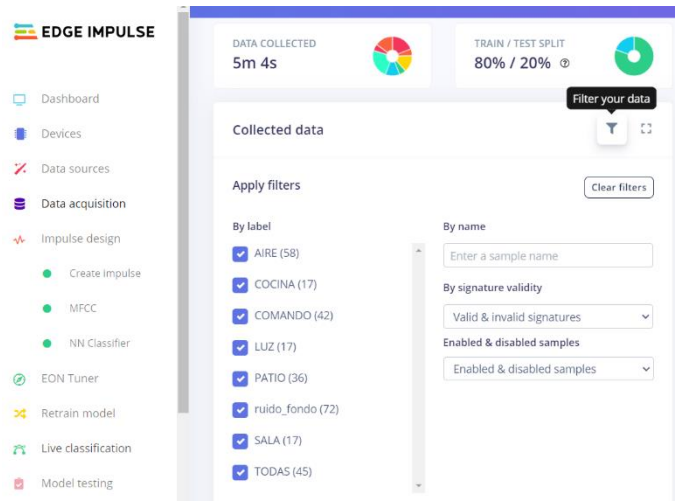
Ahora bien, debido a que la domótica está compuesta por una gigantesca variedad de controladores, sensores, actuadores, entre otros, nos inspiramos en pequeños y prácticos dispositivos como Alexa y Ok Google que usan el reconocimiento de voz para ejecutar el comando requerido por el usuario.

Adicionalmente, se obtuvo información para el desarrollo del proyecto en una página web en la que se enseñaban procesos básicos pero importantes que se usaban para la implementación, como: controlar dos LED de Arduino mediante Bluetooth, enviar información al móvil desde Arduino, controlar un LED tricolor desde el móvil a Arduino, obtener en el móvil un valor enviado por un módulo de ultrasonido. Pasando desde los componentes necesarios, hasta los emparejamientos, conexiones, drivers e implementaciones necesarias para su debido funcionamiento.

III. METODOLOGÍA

Después de realizar una breve descripción del proyecto, lo que se busca hacer y las fuentes de inspiración que se van a utilizar, procederemos a realizar tanto el código como el entrenamiento para llegar al montaje físico.

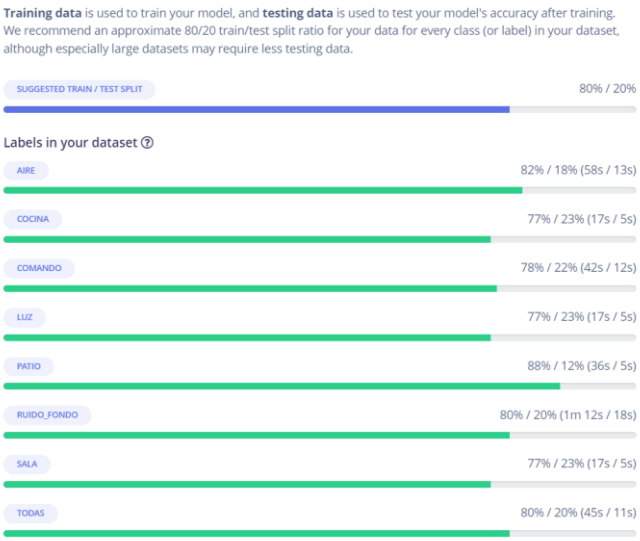
Así lo primero que hacemos es crear nuestro dataset, para esto haremos uso de Edge impulse, con esta página y con la ayuda del micrófono del nano 33 ble, comenzaremos a grabar las distintas clases, teniendo así un total de 8 clases las cuales son:



Como se observa en total tenemos 8 clases, una cosa a destacar es que no todas las clases tiene el mismo número de datos, esto se debe principalmente a que este dataSet presentado es la versión número 6, las anteriores fueron puestas a prueba evidenciando las palabras más débiles, por eso palabras como comando, todas y aire tienen mayor cantidad de muestras.

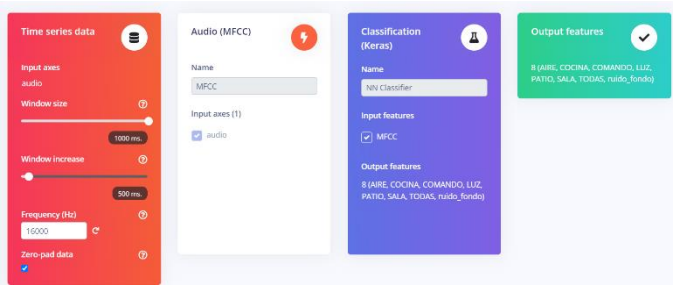
Por otro lado el ruido de fondo es la clase más amplia ya que se busca que el modelo tenga una cierta robustes, con el fin de conseguir eso el ruido de fondo está comprendido por silencio, ruido de gente hablando en el fondo y ruidos de artículos como ollas, llaves entre otros objetos cotidianos, siempre buscando que se asimile al ruido de fondo que se generaría en el uso real de esta aplicación, si desea revisar con mayor profundidad este dataSet y todo el despliegue lo invitamos a que ingrese al repositorio público de este proyecto, el cual se encontrara en el siguiente link "https://studio.edgeimpulse.com/public/155978/latest" .

Después de juntar todas las muestras y de realizar un Split en cada una de dichas muestras, distribuimos los datos para el entrenamiento y para el testeo, esto con una proporción de 80/20 como se observa a continuación:

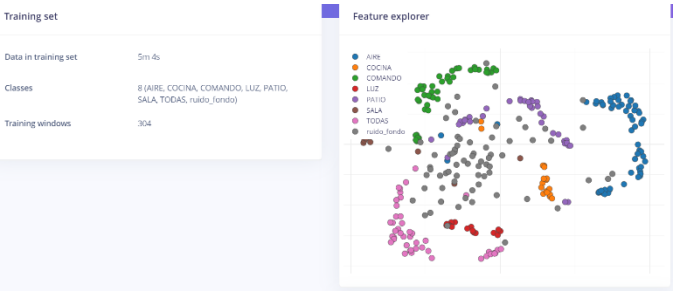


Así en cada una de las clases se buscó una proporción igual, o aproximada, con esto ya tenemos nuestros datos tanto de entrenamiento como de testeo.

Después de esto creamos el impulso, que será los parámetros que deseamos usar en nuestro modelo:

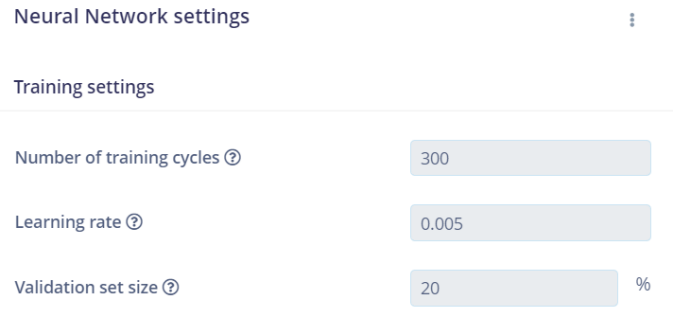


Como se observa en el impulso se usará la venta de 1 segundo que se creó en el dataset, y se crearan ventanas de 500ms para el análisis de toda la señal, posteriormente se usara el MFCC para generar el mapa de característica de los audios, se intentó usar otros modelos, pero este resulto ser el más preciso ya por último se especifica que claramente necesitamos hacer una clasificación y listo, tenemos el impulso de nuestro modelo.



Después generamos nuestro mapa de característica el cual arroja un total de 304 ventanas de entrenamiento, y como se observa en el mapa generado, se encuentra todas las clases incluso algunas muestran una tendencia de agrupación significando que los datos no están completamente dispersos, esto es una buena noticia ya que entre mejor estén agrupados los datos, hay una mayor probabilidad para que a la hora de entrenarlos tenga una mayor precisión.

Después de generar nuestro mapa de característica, nos dirigimos al entrenamiento del modelo, para esto configuramos nuestro modelo de la siguiente manera:



La cantidad de épocas fue determinada de manera empírica a

través de prueba y error llegando a la conclusión de que con alrededor de 300 épocas se llega a un modelo con una tasa de precisión superior al 90%, así nuestra arquitectura será la siguiente:

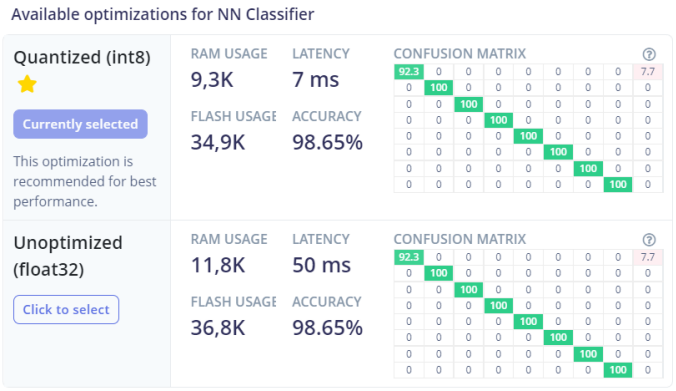
Como se observa tenemos en total 5 capas de neuronas que van aumentando hasta la última capa con un total de 32 neuronas, de igual manera se conservó una estructura similar a la que general el Edge Impulse.

Pero antes se decidió usar la convolución 1D puesto que es la más fácil para su uso, aunque con la 2D se recomienda el procesamiento de voces humanas se decidió conservar la 1D ya que genera un mayor rendimiento al copilarse el modelo. De esta forma al entrenarlo obtenemos el siguiente porcentaje de precisión:

	AIRE	COCINA	COMAND	LUZ	PATIO	SALA	TODAS	RUIDO_F
AIRE	100%	0%	0%	0%	0%	0%	0%	0%
COCINA	0%	100%	0%	0%	0%	0%	0%	0%
COMAND	0%	0%	100%	0%	0%	0%	0%	0%
LUZ	0%	0%	0%	100%	0%	0%	0%	0%
PATIO	0%	0%	0%	0%	100%	0%	0%	0%
SALA	0%	0%	0%	0%	0%	100%	0%	0%
TODAS	0%	0%	0%	0%	0%	0%	90%	10%
RUIDO_F	0%	0%	0%	0%	0%	0%	0%	100%
F1 SCORE	1.00	1.00	1.00	1.00	1.00	1.00	0.95	0.96

Como se observa tenemos un 98,4% de precisión además de que tenemos solo una clase con un problema para su clasificación la cual es la clase “Todas”, seguramente para una solución rápida, es necesario aumentar el número de muestras para esta clase, ya que se está confundiendo con el ruido de fondo. Otra explicación para esto es que el ruido de fondo usado incluye personas hablando por lo que en un punto se puede confundir el modelo, entendiendo las palabras de las personas del fondo como la categoría de Todos, pero al tener una perdida tan baja y una precisión alta, decidimos conservar este pequeño error en esta clase y continuar con el despliegue del modelo.

Por último, lo que vamos a hacer en el Edge Impulso es el despliegue de nuestro modelo, lo que significa descargar dicho modelo:



En este caso al descargar el modelo ya el error de la clase cambia y aumenta un poco la precisión, demostrando que puede haber posible conflicto con el ruido de fondo y las demás categorías ya que en ambas condiciones la clase se confundió con el ruido de fondo.
El modelo para descargar seleccionado fue el cuaternizado, ya que, aunque se espera perder algo de precisión, como se observa su latencia disminuye considerablemente al igual que el espacio de RAM que usa, esto ayuda a que el nano se encuentre libre para estar constantemente enviando la información, pero para esto es necesario que disponga de cierta cantidad de RAM y memoria, por lo que afirmamos que la mejor opción a descargar es el modelo cuaternizado.

Programación:

La programación del proyecto se divide en dos, el código para el nano y el código del Arduino uno, recordemos que el Arduino nano procesara toda la información y el Arduino uno se encargara de mover las luces y los motores puesto que tiene habilitada su salida de 5 voltios además de que puede conectarse al bluethoot del celular, para así ingresar los datos y enviar datos a la aplicación.

Código del Nano: El código usado para el nano es uno entregado por el profesor, con la diferencia que se modificó el tiempo de ventana de muestreo de 100 ms a 300 ms, permitiendo que la ventana sea mayor y no genere una cola de envió de información, por otro lado, la programación nueva principal se centra aquí:

```
void comandos_de_voz(int pred_index) {
  switch (pred_index)
  {
    case 0:     // Idle:      [0] ==> All OFF
      turn_off_leds();
      if(Comando == 1){
        if(encenderAire == 1){
         apagarAire = 0;
        }
        if(apagarAire== 0 && seleccionLuz== 0){
          digitalWrite(LEDB, LOW);
          Serial.print("\n..Recibiendo Orden de aire acondicionado..\n");
          apagarAire = 1;
          Comando = 0;
          Detener = 1;
          digitalWrite(aire, HIGH);
          delay(300);
          digitalWrite(aire, LOW);
          delay(5000);
          Serial.print("\n..Continuando muestreo de microfono..\n");
          Detener = 0;
        }
      }
      break;

    case 1:     // lift:       [1] ==> Green ON
      turn_off_leds();
```

En la función comandos de voz, se hace uso del pred_index el cual arroja no el valor si no el objeto correspondiente a dicho valor dentro de la lista de objetos en la predicción, como el modelo posee 8 objetos, significa que hay 8 posibles resultados, así para saber cuál objeto corresponda a su respectiva clase se inicia el código y se observa la terminal:

```
Predictions (DSP: 103 ms., Classification: 7 ms., Anomaly: 0 ms.):
    AIRE: 0.00000
    COCINA: 0.00000
    COMANDO: 0.00000
    LUZ: 0.96484
    PATIO: 0.01953
    SALA: 0.00000
    TODAS: 0.01562
    ruido_fondo: 0.00000
```

Así observamos la posición de la lista pred_index de cada una de las clases, con esto resulto fácil programar el switch case, donde dependiendo de cada uno de los objetos seleccionados generamos un caso concreto para cada uno, si desea conocer a mayor profundidad como se abordó cada caso, se invita a leer el código anexado en la carpeta del proyecto.

Código del Arduino Uno: El código del Arduino uno tiene la función de enviar la información que recibe del nano ble, a la aplicación de app inventor y del mismo modo se encarga de recibir la información que envía el app inventor para ser enviada nuevamente al nano 33, por lo que este Arduino es el puente de comunicación y ejecución del proyecto, no se pudo realizar esto de manera remota ya que el bluethoot ble es bastante limitado en cuanto a funciones, permitiendo solo enviar o recibir mas no puede activar los dos canales de comunicación, dificultando totalmente la ejecución de órdenes.

Aquí es donde entra el Arduino Uno, donde primero recibe la orden del nano 33, para encender el aire, uno de los 3 leds o los 3 leds a la vez, esto a través del siguiente código:

```
controlAsistente = digitalRead(encenderAsistente);
controlAire = digitalRead(encenderAire);
controlLuces = digitalRead(seleccionLuces);
controlSala = digitalRead(encenderSala);
controlCocina = digitalRead(encenderCocina);
controlPatio = digitalRead(encenderPatio);
controlTodas = digitalRead(todaslasLuces);
if (controlAsistente == HIGH) {
  int i = 1;
  Serial.print(i);
  Serial.print("\n");
  delay(300);
}
if (controlAire == HIGH) {
  int i = 2;
  Serial.print(i);
  Serial.print("\n");
  delay(300);
  digitalWrite(apagarAire, HIGH);
  seguroAire = 0;
}
if (controlSala == HIGH) {
  int i = 3;
  Serial.print(i);
  Serial.print("\n");
  delay(300);
  digitalWrite(apagarSala, HIGH);
  seguroSala = 0;
}
```

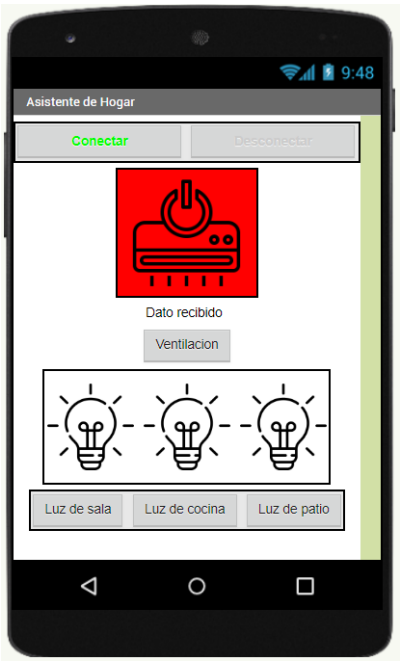
Así se está leyendo constantemente los pines asignados como entradas, con lo que cuando el nano 33 ble, genere una señal de activación este inmediatamente activara un pulso que es recibido por el uno, cuando esto ocurre esa entrada se pone alto y enciende en la salida correspondiente a la acción que se ordenó desde el nano, a la vez que envía esta información a la app inventor a través de una comunicación serial con este, pero esto debe de ocurrir a la vez, ya que el nano está censando de manera constante, y enviado información constantemente solo que no se considera los flancos de bajada, así que para mantener una coordinación, se deja un tiempo de 300 ms para que la aplicación reciba la información, este tiempo de muestreo debe de ser igual tanto en el nano como en el Arduino uno y como el app inventor, todo esto se coordina gracias a un temporizador que censa la información de entrada cada 300 ms, mismo tiempo en el que se envía nueva información desde el Nano.

Luego desde la aplicación se puede apagar las luces de manera manual, esto significa que el Arduino Uno recibe información proveniente de la conexión serial, esta lectura se hace en el siguiente código:

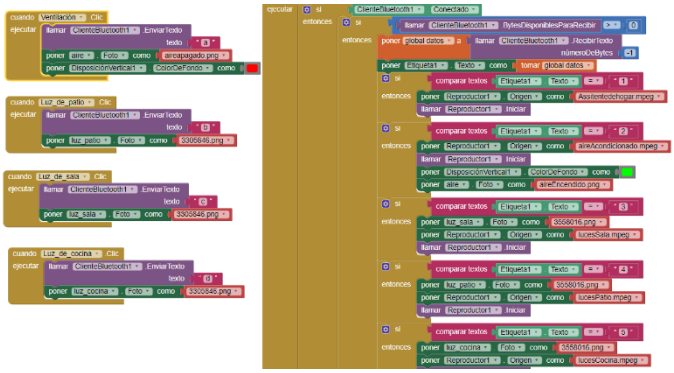
```
if( Serial.available()){
  valorblutuch = Serial.read();
  if( valorblutuch =='a'){
    digitalWrite(apagarAire, LOW);
    digitalWrite(controlAire, LOW);
    digitalWrite(desbloqueoAire, HIGH);
    delay(100);
    digitalWrite(desbloqueoAire, LOW);
    }
  if( valorblutuch =='c'){
    digitalWrite(apagarSala, LOW);
    seguroSala = 1;
    }
  if( valorblutuch =='b'){
    digitalWrite(apagarPatio, LOW);
    seguroPatio = 1;
    }
  if( valorblutuch =='d'){
    digitalWrite(apagarCocina, LOW);
    seguroCocina = 1;
    }
  }
```

Con esto aseguramos que al recibir la respectiva letra se apague la luz o el motor que corresponde a esa clase, a la vez que generamos una señal que será enviada al nano, para que este entienda que se encuentra disponible la activación de esa acción así encender el aire solo se puede hacer una vez, para volver a activar el aire es necesario que primero se apague.

App Inventor: Por último, la programación de bloques del app inventor no implica mayor complejidad, ya que solo se requiere recibir y enviar información el problema radica en que en ocasiones como se limita la aplicación a que lea cada 300 ms la información (ya que este es el tiempo que se tarda para que el nano envié la información de audio), se puede generar conflictos entre el envío de los bits y el recibir bits.
Las consecuencias de esta acción recaen en un delay para que la aplicación apague correctamente los dispositivos por lo que se tiene que oprimir varias veces el botón para que se ejecute la acción. Después de aclarar esto observamos la interfaz de la aplicación:



Como se observa la interfaz realizada es una interfaz simple pero que entrega la información que el usuario necesita, una anotación es que el texto que dice dato recibido es un método que tenemos para verificar que se haya entregado la información correcta, pero no afecta ni influencia para nada la interacción con el usuario.



Este es una pequeña parte de la programación de bloques usada para el funcionamiento de la aplicación, en el lado izquierda se encuentra él envió de la ordenes hacia el Arduino a través del bluethooth y la otra columna se encuentra la programación para recibir la información que envié el Arduino Uno a través de igual manera por el módulo bluethooth. Si desea conocer más a detalle cómo se programó la aplicación en la carpeta del proyecto se anexará él .aia para que lo cargue a su app inventor y pueda manipularlo como desee o puede instalar el apk de la aplicación que también se anexara en la carpeta del proyecto.

IV. MODELO FISICO

Para el modelo físico se compró una casa, para tener la arquitectura y tener un mejor entorno para la simulación de nuestro modelo, así lo que se agregan son las conexiones de los 3 leds y el ventilador quedando de la siguiente manera:

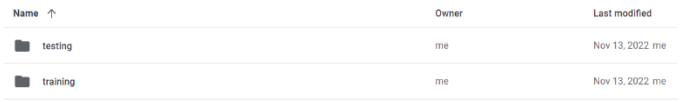
En cuanto las conexiones del Arduino nano y el Arduino UNO, se encuentran especificadas dentro del código de cada uno.

Ahora que tenemos nuestro modelo físico, es necesario realizar la prueba de fuego, y poner a funcionar el modelo, comprobando así su correcto funcionamiento, para esto se realizó un video en el que se muestra y se explica a detalle cómo funciona el programa: "**https://youtu.be/GQ2fknYMQPE**"

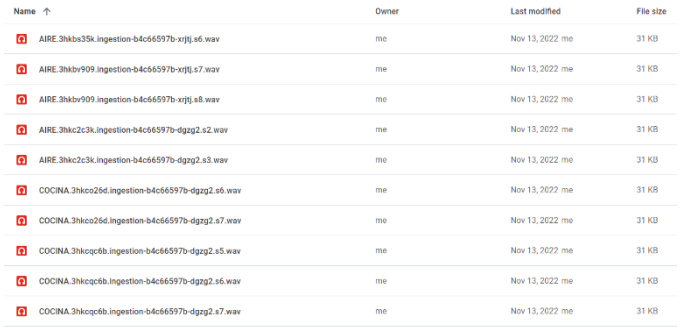
V. CONVERSION DEL MODELO EN COLAB

Después de comprobar el funcionamiento de nuestro modelo y de realizar su despliegue físico, lo único que resta es hacer su conversión a un modelo en colab, esto se hace con el fin de dar a entender que no importa si se realiza en Edge Impulse o en colab, el modelo debería de poder desplegarse, claro que el modelo en colab tendrá ciertas limitaciones, por eso llegaremos solo a una parte de este proceso.

El primer paso es agregar los datos usados en el Edge Impulse en una carpeta de drive:



Como se mencionó antes en estas carpetas debe de estar los datos usados en el entrenamiento de Edge Impulse, debe de recordar el orden en el que estos se almacenaron ya que será necesario para el entrenamiento.



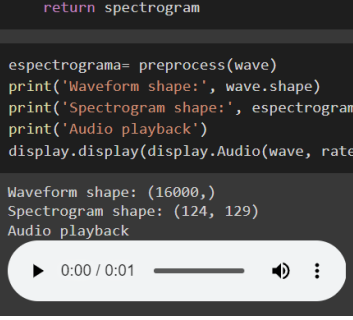
Después de acomodar los datos, comenzamos a programar en Python, primero permitiendo que la plantilla del colab tenga acceso a Google drive, después descargamos las librerías y procedemos a mostrar los datos que vamos a entrenar:

Sacamos una muestra para observar que estén correctas, en especial que sean de 1 segundo, para estar seguros verificamos que se escuche la palabra que deseamos entrenar.

```
def preprocess(wave):
    #wav = wave[:32000]
    wav = wave[:16000]
    #zero_padding = tf.zeros([48000] - tf.shape(wav), dtype=tf.float32)
    # wav = tf.concat([zero_padding, wav],0)
    spectrogram = tf.signal.stft(wav, frame_length=255, frame_step=128)
    spectrogram = tf.abs(spectrogram)
    return spectrogram
```

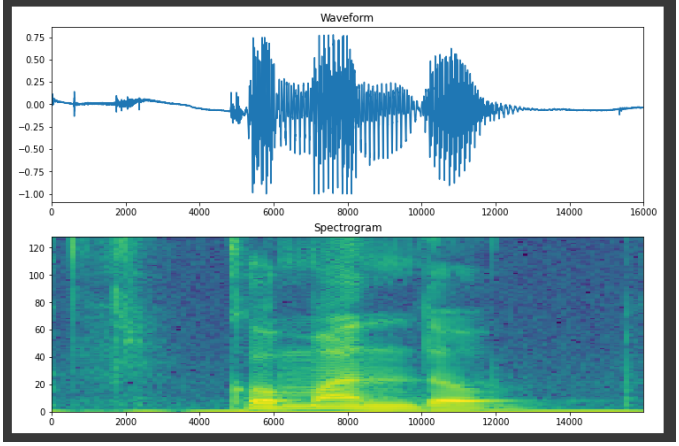
```
espectrograma= preprocess(wave)
print('Waveform shape:', wave.shape)
print('Spectrogram shape:', espectrograma.shape)
print('Audio playback')
display.display(display.Audio(wave, rate=16000))
```

```
Waveform shape: (16000,)
Spectrogram shape: (124, 129)
Audio playback
```



después de confirmar que la palabra es correcta, entendible y se encuentra en una ventana de 1 segundos, podemos comenzar a entrenar nuestro modelo, primero generando el espectrograma a cada uno de los datos entregados:

```
fig, axes = plt.subplots(2, figsize=(12, 8))
timescale = np.arange(wave.shape[0])
axes[0].plot(timescale, wave.numpy())
axes[0].set_title('Waveform')
#axes[0].set_xlim([0, 32000])
axes[0].set_xlim([0, 16000])
plot_spectrogram(espectrograma.numpy(), axes[1])
axes[1].set_title('Spectrogram')
plt.show()
```



Después de esto generamos nuestras ventanas de entrenamiento, para esto debemos de contar el número de audios que se ingresaron la carpeta de drive:

```
YtrainIni=np.zeros((163,1))
for i in range(18):
    YtrainIni[i]=0

for i in range(18,35):  #rec
    YtrainIni[i]=1  #abrir t

for i in range(35,55):
    YtrainIni[i]=2

for i in range(55,72):
    YtrainIni[i]=3

for i in range(72,89):
    YtrainIni[i]=4

for i in range(89,106):
    YtrainIni[i]=5

for i in range(106,123):
    YtrainIni[i]=6

for i in range(123,163):
    YtrainIni[i]=7

print(YtrainIni)
```

Así generamos un arreglo con el numero de clases y el numero de datos de entrenamiento que hay en cada clase, como se observa se ingresaron un total de 163 audios para este modelo en Colab. después armamos la arquitectura de nuestro modelo para nuestro entrenamiento, en este caso se hará uso de las convoluciones 2D recordemos que estas son la mejor opción para procesar audio, a la par comprobaremos cual obtiene mejores resultados si la 2D de Colab o la 1D en Edge Impulse:

```
#Definición del modelo
modelo = keras.models.Sequential()
modelo.add(keras.layers.Conv2D(8, 8, activation="relu",padding="same", input_shape=(124,129,1)))
modelo.add(keras.layers.Conv2D(16, 8, activation="relu",padding="same"))
modelo.add(keras.layers.Conv2D(32, 8, activation="relu",padding="same"))
modelo.add(keras.layers.Conv2D(64, 8, activation="relu",padding="same"))
modelo.add(keras.layers.MaxPooling2D(pool_size=2,strides=2, padding='same'))
modelo.add(keras.layers.Flatten())
modelo.add(keras.layers.Dense(8, activation = 'softmax'))
modelo.summary()

keras.utils.plot_model(modelo, to_file='model_plot3.png', show_shapes=True, show_layer_names=True
```

conv2d_input	input:	[(None, 124, 129, 1)]
InputLayer	output:	[(None, 124, 129, 1)]

↓

conv2d	input:	(None, 124, 129, 1)
Conv2D	output:	(None, 124, 129, 8)

↓

conv2d_1	input:	(None, 124, 129, 8)
Conv2D	output:	(None, 124, 129, 16)

↓

conv2d_2	input:	(None, 124, 129, 16)
Conv2D	output:	(None, 124, 129, 32)

↓

conv2d_3	input:	(None, 124, 129, 32)
Conv2D	output:	(None, 124, 129, 64)

↓

max_pooling2d	input:	(None, 124, 129, 64)
MaxPooling2D	output:	(None, 62, 65, 64)

↓

flatten	input:	(None, 62, 65, 64)
Flatten	output:	(None, 257920)

↓

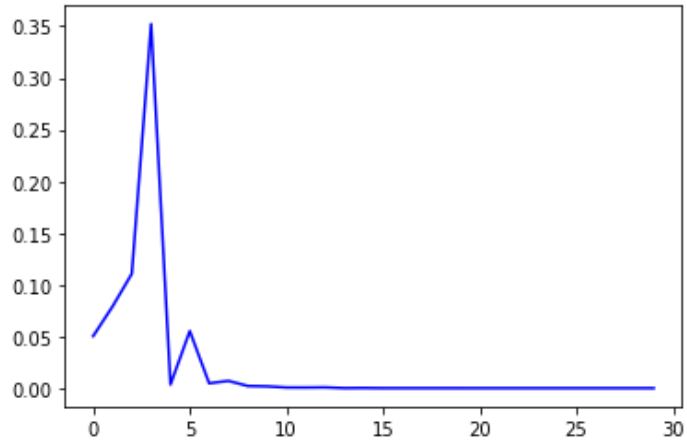
dense	input:	(None, 257920)
Dense	output:	(None, 8)

Como se observa tenemos un modelo robusto con 5 capas de neuronas, donde vamos aumentando el numero de neuronas en cada capa.

Ahora que definimos nuestro modelo, comenzamos a entrenar, para posteriormente observar la grafica de perdida generada durante el entrenamiento:

```
plt.plot(history.history["loss"],'b')
modelo.evaluate(Xtrain, Ytrain)
```

```
6/6 [==============================] - 0s 29ms/step - loss: 2.5230e-06 - accuracy: 1.0000
[2.523033344914438e-06, 1.0]
```



Al final del entrenamiento se obtuvo una precisión del 100% y una perdida de 2,52 elevado a la menos 6, como se observa en la grafica se genera un pico hasta que se estabiliza a partir de las 10 épocas, mostrando que el modelo puede llegar a tener buenos resultados.

```
YValIni=np.zeros((46,1))
for i in range(5):
    YValIni[i]=0

for i in range(5,10):
    YValIni[i]=1

for i in range(10,16):
    YValIni[i]=2

for i in range(16,21):
    YValIni[i]=3

for i in range(21,26):
    YValIni[i]=4

for i in range(26,31):
    YValIni[i]=5

for i in range(31,36):
    YValIni[i]=6

for i in range(36,46):
    YValIni[i]=7

print(YValIni)
YVal= keras.utils.to_categorical(YValIni)

print(YVal)
```

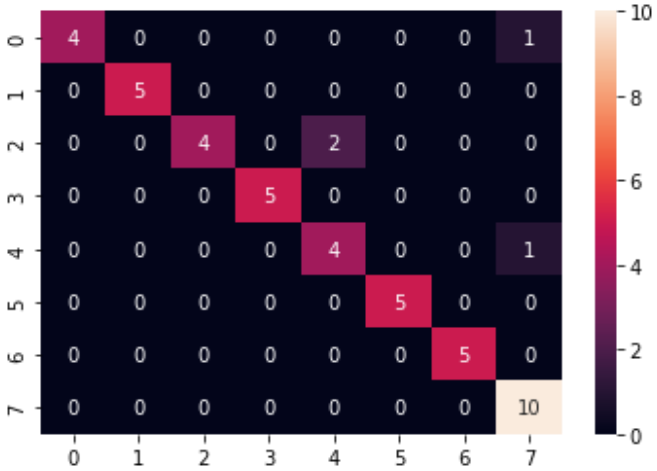
Después generamos un vector con los datos para el testeo, en esta parte de igual manera se debe de tener en cuenta el número de datos que se ingresaron en la carpeta de drive, que en nuestro caso es un total de 46 datos de prueba repartido entre las 8 clases. Después de tener nuestro vector lo que resta es hacer una predicción de los datos:

```
2/2 [==============================] - 0s 28ms/step
              precision    recall  f1-score   support

           0       1.00      0.80      0.89         5
           1       1.00      1.00      1.00         5
           2       1.00      0.67      0.80         6
           3       1.00      1.00      1.00         5
           4       0.67      0.80      0.73         5
           5       1.00      1.00      1.00         5
           6       1.00      1.00      1.00         5
           7       0.83      1.00      0.91        10

    accuracy                           0.91        46
   macro avg       0.94      0.91      0.92        46
weighted avg       0.93      0.91      0.91        46

[[ 4  0  0  0  0  0  0  1]
 [ 0  5  0  0  0  0  0  0]
 [ 0  0  4  0  2  0  0  0]
 [ 0  0  0  5  0  0  0  0]
 [ 0  0  0  0  4  0  0  1]
 [ 0  0  0  0  0  5  0  0]
 [ 0  0  0  0  0  0  5  0]
 [ 0  0  0  0  0  0  0 10]]
```



Como se observa el resultado final no es muy bueno que se diga, aunque todas las clases lo clasifica por encima del 50% tenemos una clase que esta en el 73% mostrando que se equivoco dos veces en su predicción, pero sigue estando por encima del 50% por lo que se conserva este modelo, ya conocemos que la precisión en Google Colab no llega a ser similar a la precisión obtenida en el Edge Impulse, ya para finalizar solo nos resta descargar el modelo en un formato ejecutable el cual será el .h

```
# Convierta el modelo al formato TensorFlow Lite sin cuantificación
converter = tf.lite.TFLiteConverter.from_keras_model(modelo)
tflite_model = converter.convert()


# Guardar el modelo en el disco
open("gesture_model.tflite", "wb").write(tflite_model)

import os
basic_model_size = os.path.getsize("gesture_model.tflite")
print("Model is %d bytes" % basic_model_size)
```

```
#La siguiente celda crea una matriz de bytes constantes que contiene el modelo TFlite.
#Importarlo como una pestaña con el siguiente código.

!echo "const unsigned char model[] = {" > /content/model.h
!cat gesture_model.tflite | xxd -i      >> /content/model.h
!echo "};"                              >> /content/model.h

import os
model_h_size = os.path.getsize("model.h")
print(f"Header file, model.h, is {model_h_size:,} bytes.")
print("\nAbre el panel lateral (refrecar si es necesario). Doble click en model.h para descargar el archivo.")
```

Después de desplegar el modelo en el .h realizamos el programa en colab, para esto se intenta hacer uso de dos ejemplos el de activación de micrófono y el de entrenamiento sin Edge impulse que a su vez es un ejemplo de la librería IMU.

```
#include <Arduino_LSM9DS1.h>

#include <TensorFlowLite.h>
#include <tensorflow/lite/micro/all_ops_resolver.h>
#include <tensorflow/lite/micro/micro_error_reporter.h>
#include <tensorflow/lite/micro/micro_interpreter.h>
#include <tensorflow/lite/schema/schema_generated.h>
#include <tensorflow/lite/version.h>

#include "modelo3Miniproyecto.h"

const float tamañoVentana = 2.5;
const int numSamples = 124;

int samplesRead = numSamples;

// global variables used for TensorFlow Lite (Micro)
tflite::MicroErrorReporter tflErrorReporter;

// pull in all the TFLM ops, you can remove this line and
// only pull in the TFLM ops you need, if would like to reduce
// the compiled size of the sketch.
tflite::AllOpsResolver tflOpsResolver;

const tflite::Model* tflModel = nullptr;
tflite::MicroInterpreter* tflInterpreter = nullptr;
TfLiteTensor* tflInputTensor = nullptr;
TfLiteTensor* tflOutputTensor = nullptr;
```

Primero preparamos e incluimos tanto las librerías para la lectura del modelo de tensorflow y después el modelo .h descargado del Colab.

```
constexpr int tensorArenaSize = 8 * 1024;
byte tensorArena[tensorArenaSize] __attribute__((aligned(16)));

// array to map gesture index to a name
const char* GESTURES[] = {
  "AIRE",
  "COCINA",
  "COMANDO",
  "LUZ",
  "PATIO",
  "SALA",
  "TODAS",
  "ruido_fondo"
};

#define NUM_GESTURES (sizeof(GESTURES) / sizeof(GESTURES[0]))

void setup() {
  Serial.begin(9600);
  while (!Serial);

  // initialize the IMU
  if (!IMU.begin()) {
    Serial.println("Failed to initialize IMU!");
    while (1);
  }

  // Initialize the TinyML Shield
  initializeShield();
```

```
PDM.onReceive(onPDMdata);
// Initialize PDM microphone in mono mode with 16 kHz sample rate
if (!PDM.begin(1, 16000)) {
  Serial.println("Failed to start PDM");
  while (1);
}

Serial.println("Welcome to the microphone test for the built-in microphone on the Nano 33 BLE Sense\n");
Serial.println("Use the on-shield button or send the command 'click' to start and stop an audio recording");
Serial.println("Open the Serial Plotter to view the corresponding waveform");
}
Serial.println();

// get the TFL representation of the model byte array
tflModel = tflite::GetModel(model);
if (tflModel->version() != TFLITE_SCHEMA_VERSION) {
  Serial.println("Model schema mismatch!");
  while (1);
}

// Create an interpreter to run the model
tflInterpreter = new tflite::MicroInterpreter(tflModel, tflOpsResolver, tensorArena, tensorArenaSize, &tflErrorReporter);

// Allocate memory for the model's input and output tensors
tflInterpreter->AllocateTensors();

// Get pointers for the model's input and output tensors
tflInputTensor = tflInterpreter->input(0);
tflOutputTensor = tflInterpreter->output(0);
```

Con esta primera parte aseguramos que la librería se lea correctamente al igual que descargamos lo necesario para activar el micrófono, además de incluir las clases que pensamos clasificar.

```
void loop() {
  float aX, aY, aZ, gX, gY, gZ;
  bool clicked = readShieldButton();
  if (clicked){
    record = !record;
  }

  // see if a command was sent over the serial monitor and record it if so
  String command;
  while (Serial.available()) {
    char c = Serial.read();
    if ((c != '\n') && (c != '\r')) {
      command.concat(c);
    }
    else if (c == '\r') {
      commandRecv = true;
      command.toLowerCase();
    }
  }
  while (samplesRead == numSamples) {
  // parse the command if applicable
  if (commandRecv && command == "click") {
     if (aSum >= tamañoVentana) {
        // reset the sample read count
        samplesRead = 0;
        break;
    commandRecv = false;
    record = !record;
  }
}
```

Después definimos los ejes y generamos la grabación de audio, en esta parte es importante aclarar que el audio este controlado por el botón que posee el Arduino, o enviándolo a la terminal, durante el testeo este parará automáticamente hasta llegar al tamaño de la venta definido, esto será guardado en el samplesRead, pero solo lo que agarro en el tamaño de la ventana y el resto del audio se grabará en el comand.

```
void loop() {
  float aX, aY, aZ, gX, gY, gZ;
  bool clicked = readShieldButton();
  if (clicked){
    record = !record;
  }

  // see if a command was sent over the serial monitor and record it if so
  String command;
  while (Serial.available()) {
    char c = Serial.read();
    if ((c != '\n') && (c != '\r')) {
      command.concat(c);
    }
    else if (c == '\r') {
      commandRecv = true;
      command.toLowerCase();
    }
  }
  while (samplesRead == numSamples) {
  // parse the command if applicable
  if (commandRecv && command == "click") {
     if (aSum >= tamañoVentana) {
        // reset the sample read count
        samplesRead = 0;
        break;
```

```
void loop() {
  float aX, aY, aZ, gX, gY, gZ;
  bool clicked = readShieldButton();
  if (clicked){
    record = !record;
  }

  // see if a command was sent over the serial monitor and record it if so
  String command;
  while (Serial.available()) {
    char c = Serial.read();
    if ((c != '\n') && (c != '\r')) {
      command.concat(c);
    }
    else if (c == '\r') {
      commandRecv = true;
      command.toLowerCase();
    }
  }
  while (samplesRead == numSamples) {
  // parse the command if applicable
  if (commandRecv && command == "click") {
    if (aSum >= tamanoVentana){

      samplesRead = 0;
      break;
  while (samplesRead < numSamples) {

      // normalize the IMU data between 0 to 1 and store in the model's
      // input tensor
      tflInputTensor->data.f[samplesRead * 12 + 0] = (aX + 4.0) / 8.0;
      tflInputTensor->data.f[samplesRead * 12 + 1] = (aY + 4.0) / 8.0;
      tflInputTensor->data.f[samplesRead * 12 + 2] = (aZ + 4.0) / 8.0;
      tflInputTensor->data.f[samplesRead * 12 + 3] = (aX + 100.0) / 2000.0;
      tflInputTensor->data.f[samplesRead * 12 + 4] = (aY + 100.0) / 2000.0;
      tflInputTensor->data.f[samplesRead * 12 + 5] = (aZ + 100.0) / 2000.0;
      tflInputTensor->data.f[samplesRead * 12 + 6] = (aX + 100.0) / 2000.0;
      tflInputTensor->data.f[samplesRead * 12 + 7] = (aY + 100.0) / 2000.0;
      tflInputTensor->data.f[samplesRead * 12 + 8] = (aZ + 100.0) / 2000.0;
      tflInputTensor->data.f[samplesRead * 12 + 9] = (aX + 100.0) / 2000.0;
      tflInputTensor->data.f[samplesRead * 12 + 10] = (aY + 100.0) / 2000.0;
      tflInputTensor->data.f[samplesRead * 12 + 11] = (aZ + 100.0) / 2000.0;
      tflInputTensor->data.f[samplesRead * 12 + 12] = (aX + 100.0) / 2000.0;

      samplesRead++;

      if (samplesRead == numSamples) {
        // Run inferencing
        TfLiteStatus invokeStatus = tflInterpreter->Invoke();
        if (invokeStatus != kTfLiteOk) {
          Serial.println("Invoke failed!");
          while (1);
          return;
```

Con esto lo que hacemos es comenzar a ingresar al .data de nuestro modelo una serie de valores random, esto se ingresa en los tres ejes que posee el audio x y y z, por ultimo guardamos el audio de cada una de las ventanas en una matriz, esta se guarda en el sample record, generando asi que el modelo procese la informacion y arroje una serie de valores, el problema es que no tenemos ningun tipo de medio para realizar el procedimiento por lo que solo pudimos llegar hasta aquí.

VI. CONCLUSIÓN

A manera de conclusión cabe recalcar que, con los conocimientos aprendidos en el curso, se obtuvieron las bases necesarias para implementar un mejor sistema de toma de decisiones y activación de diferentes actuadores para ejecutar modelos fundamentados en reconocimiento de movimientos, sonidos e imágenes. De la misma forma, se evidenció las diferencias en la implementación de redes neuronales con microprocesadores y sensores, el uso de aplicaciones como App Inventor y los diferentes mecanismos para cumplir una función específica. Esto nos permite una visión más objetiva y global sobre diferentes campos de aplicación en la industria partiendo desde la domótica, hasta los pequeños dispositivos que constantemente aprenden de las costumbres particulares de cada usuario, profundizando en investigaciones y desarrollo de

dispositivos que sean más eficientes con un menor consumo energético.

APÉNDICE

Los apéndices, si son necesarios, aparecen antes del reconocimiento.

REFERENCES

[1] http://kio4.com/appinventor/9bluetootharduino.htm
[2] http://kio4.com/appinventor/9B_bluetooth_arduino_basico.htm
[3] http://kio4.com/appinventor/9Bbluetootharduino.htm
[4] http://kio4.com/arduino/29Bmotordecontinua.htm