Tutorial 5

Reconocimiento de sonidos personalizados con Yamnet

Introducción

En este tutorial, crearás

Objetivos

En este tutorial aprenderás a:

- Utilizar las Web Audio API para
    - Cargar ficheros de audio y obtener su representación como forma de onda
    - Utilizar el micrófono como fuente de audio
- Obtener las características de un fichero de audio del módelo YAMNet para su clasificación
- Realizar aprendizaje por transferencias utilizando las representaciones (embeddings) generadas por YAMNet

Requisitos

- Una versión reciente de Chrome o de otro navegador actualizado.
- Tu editor favorito que se **ejecute localmente en tu máquina**, por ejemplo Visual Studio Code con la extensión Live Server, **o en la Web** con servicios como codesandbox.io o glitch.com.
- Familiaridad con JavaScript.
- Conocimientos básicos de Node.js.
- Familiaridad con las herramientas para desarrolladores de Chrome (DevTools) o de otro navegador.

Código

- Código tutorial

Tutorial

Yamnet es un modelo de aprendizaje automático para clasificar eventos de audio realizando predicciones independientes para cada uno de los 521 eventos de audio de la ontología AudioSet. Se puede utilizar como extractor de características para crear rápidamente clasificadores de audio especializados sin requerir una gran cantidad de datos etiquetados y sin tener que entrenar un modelo grande de principio a fin. La salida de representación del audio (embedding) 1024D de YAMNet se puede utilizar como las características de entrada de otro modelo que luego se puede entrenar con una pequeña cantidad de datos para una tarea específica.

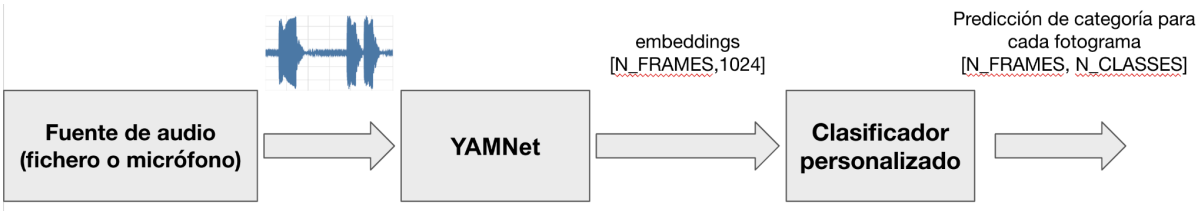
En este tutorial aprenderás cómo utilizar Yamnet para realizar aprendizaje por transferencia y entrenar un simple clasificador para reconocer cuatro clases de sonidos, entrenando el modelo en un conjunto de datos reducidos.
Para el tutorial utilizarás una versión reducida del dataset ESC-50. ESC-50 es un conjunto de datos de sonidos ambientales, con 40 clip de audio de duración de 5 segundos por cada clase o categoría de sonido ambiental. El conjunto de datos se ha organizado previamente en 5 particiones para la validación cruzada comparable, asegurándose de que los fragmentos de 5 segundos de un mismo archivo de origen estén contenidos en una sola partición. Podéis encontrar una descripción del contenido del dataset en este fichero.

En el tutorial aprenderás cómo clasificar los sonidos de bebés llorando, despertador, tirar del váter y gotas de agua. Las categorías de ESC-50 correspondientes son:

```
"crying_baby", "clock_alarm", "toilet_flush", "water_drops"
```

Cada categoría tiene un número identificador en el dataset original, pero esto no importa, ya que al utilizar un número reducido de categorías y entrenador un clasificador personalizado, volverás a asignar un identificador de tu elección a cada categoría.

La idea es la siguiente: utilizamos YAMNet para sacar los embeddings (para cada fotograma de YAMNet de 0,96 segundos) de una fuente de audio (fichero o micrófono) y utilizamos la salida de YAMNet como entrada a un clasificador personalizado que nos dará la probabilidad para cada clase (para cada fotograma). Con esta información, por ejemplo, podemos calcular la media agregada para el clip de audio.



En el tutorial utilizaremos las Web Audio API para realizar operaciones básicas sobre ficheros y flujos de audio, como cargar y reproducir un clip de audio, o grabar un clip de audio en un fichero y utilizar el micrófono como fuente de audio. La gestión del audio en la Web es un tema complejo que no trataremos en detalle en el curso: para esto podéis consultar los tutoriales en la página de MDN.

Paso 1. Esqueleto HTML

1.1. Rellenar el esqueleto de la página HTML

Empieza creando el esqueleto de una página Web. Crea un archivo y nómbralo **index.html**. El archivo contendrá el siguiente código:

```
<!DOCTYPE html>
<html lang="en">

<head>
 <title>Aprendizaje por transferencia con YAMNet</title>
 <meta charset="utf-8" />
 <meta http-equiv="X-UA-Compatible" content="IE=edge" />
 <link rel="shortcut icon" href="data:image/x-icon;," type="image/x-icon">
 <meta name="viewport" content="width=device-width, initial-scale=1,
maximum-scale=1, user-scalable=no" />

 <!-- CSS Styling -->
 <link rel="stylesheet" href="style.css" />

     <!-- TensorFlow.js library -->
 <script
     src="https://cdn.jsdelivr.net/npm/@tensorflow/tfjs@latest"
     type="text/javascript"
   ></script>
</head>

<body>
 <h1>Aprendizaje por transferencia con YAMNet</h1>

 <section>

   <div class="ui-div">
     <button class="ui-btn" id="btnCreateAndTrain">Create and
Train</button>
     <button class="ui-btn" id="btnLoadModel">Load Model</button>
     <button class="ui-btn" id="btnTestModel">Test Model</button>
   </div>
   <div class="ui-div">
     <button class="ui-btn" id="btnMicStart">Start Mic.</button>
     <button class="ui-btn" id="btnMicStop">Stop Mic.</button>
```

```
    </div>

    <div class="ui-div">
      <audio id="audioPlayer" controls="controls" src=""></audio>
    </div>
  </section>

  <script src="index.js"></script>
</body>

</html>
```

La página HTML proporcionará una interfaz para experimentar con la creación de un modelo para reconocer sonidos “personalizados” a partir de la salida de YAMNet (los embeddings). El modelo se entrenará y validará con el conjunto de sonidos de ESC-50.

En la interfaz creamos los botones para activar las siguiente funcionalidades:

1. Crear, entrenar y guardar el modelo de clasificación personalizado (para la cuatro categorías)
2. Cargar un modelo existente (que ya habíamos entrenado previamente)
3. Comprobar que el modelo funciona correctamente utilizando ficheros de audio
4. Empezar a realizar clasificaciones de los sonidos desde el micrófono
5. Terminar la tarea de clasificación y parar el micrófono.

En la interfaz también pondremos el elemento **audio** que utilizaremos para reproducir los archivos de test para el modelo de clasificación.

Paso 2. Entender la estructura de la aplicación Web

La aplicación Web del tutorial tiene la siguiente estructura de ficheros y carpetas:

```
yamnet-transfer-learning/
├─ data/                    # carpeta con el dataset
|  ├─ audio/         # carpetas con los ficheros audio
|  |  ├─ 1-12653-A-15.wav
|  |  ├─ 1-12654-A-15.wav
|  |  ├─ …
|  ├─ testMetadata.csv         # fichero con los metadatos de test
|  ├─ trainMetadata.csv       # fichero con los metadatos para entrenamiento
├─ model/                     # carpeta con el modelo para clasificar audio
|  ├─ model.json
|  ├─ model.weights.bin
├─ style.css
├─ index.html
├─ index.js
```

```
├─ notas.txt                    # fichero donde escribir vuestras observaciones
```

2.1. Dataset y sus ficheros

El dataset que tenemos a disposición es un subconjunto del dataset ESC-50, con los ficheros de audio y la información de las cuatros clases que consideramos en este tutorial. Debido a que YAMNet acepta audio con un muestreo de 16 kHz, los ficheros de audio de ESC-50 se han preprocesado para contener todos 5 segundos de audio a 16 kHz (80000 muestras).

El proyecto, en el directorio **data**, contiene dos ficheros **.csv** con información relevante sobre el dataset:

1. **fileName**: nombre del fichero en la carpeta data/audio
2. **className**: nombre de la categoría del clip de audio
3. **classNumber**: identificador de la categoría del clip de audio (número de 0 a 3)
4. **fold**: partición del dataset

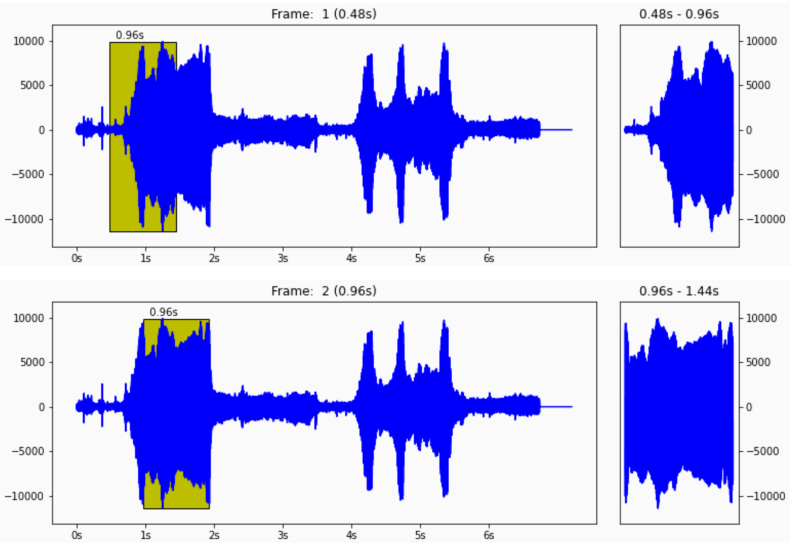
Utilizamos las particiones del 1 al 4 para el entrenamiento (80%) y la partición 5 (20%) para las pruebas. Los identificativos de las clases han sido reasignados de esta manera:

```
crying_baby -> 0
clock_alarm -> 1
toilet_flush -> 2
water_drops -> 3
```

En el directorio **data/audio** están los ficheros de audio en formato **.wav**.

2.2. Modelo

La carpeta model contiene ya el modelo preentrenado (con los datos del dataset para el entrenamiento), listo para cargarse como modelo de capas para reconocer los sonidos de las cuatro categorías. Este modelo recibe como entrada un tensor 2D [N_FRAMES, 1024] con los embeddings del modelo YAMNet y su salida será un tensor [N_FRAMES, 4] con las predicciones (probabilidad de cada una de las cuatro categorías) para cada fotograma. Por ejemplo, si hiciéramos las predicciones en 5 segundos de audio (duración de los clips del dataset), tendremos 10 fotogramas, ya que YAMNet divide el clip en fotogramas de 0,96 segundos, intercalados cada 0,48 segundos.



2.3. Aplicación

Los demás ficheros en la carpeta raíz son los ficheros clásicos de nuestra aplicación, **index.html**, **index.js** y **style.css**.

3. Empezar a desarrollar la aplicación

Podemos empezar a crear nuestra aplicación, escribiendo el código en el fichero **index.js**.

3.1. Crear la función app

Empezaremos creando la función app, donde escribiremos el código principal de la aplicación

```
(async function initApp() {

    try {
        initTFJS();
        await app();

    } catch (error) {
        console.error(error);
    }

}());

async function app() {}
```

3.2 Cargar los metadatos y el modelo YAMNet

En la función app, creamos dos array que contendrán los metadatos de nuestros dataset de entrenamiento y de prueba. También esperamos a que se cargue el modelo YAMNet, que

utilizaremos para sacar las características de los ficheros de audio y habilitar el aprendizaje por transferencia.

```
const YAMNET_MODEL_URL = "https://tfhub.dev/google/tfjs-model/yamnet/tfjs/1";

async function app() {

   const trainCsvUrl = "data/trainMetadata.csv"; // Path to your training CSV
   const testCsvUrl = "data/testMetadata.csv";  // Path to your testing CSV

   const trainDataArray = await loadCsvMetadata(trainCsvUrl);
   const testDataArray = await loadCsvMetadata(testCsvUrl);

   yamnet = await loadYamnetModel();
   console.log("YamNet model loaded");

}

async function loadCsvMetadata(csvUrl) {
   const metadata = tf.data.csv(csvUrl, {
       hasHeader: true
   });

   return await metadata.toArray();;
}
```

YAMNET_MODEL_URL será una variable global en nuestro fichero.

La función **loadCsvMetadata()** es una utilidad diseñada para leer y procesar archivos CSV en un entorno de TensorFlow.js. Se utiliza para cargar los metadatos de entrenamiento y prueba en el formato de valores separados por comas (CSV) y convertirlos en un array.

tf.data.csv() es un método de TensorFlow.js que permite cargar y procesar datos CSV. Se encarga de interpretar el contenido del archivo como un conjunto de registros. La opción hasHeader indica si la primera fila del archivo CSV contiene los nombres de las columnas (valor true), lo cual facilita identificar los datos. La línea await metadata.toArray() convierte los registros cargados del fichero CSV en un array de objetos de JavaScript. Cada objeto contiene {fileName, className, classNumber, fold}.

3.3. Habilitar los botones de la interfaz

Creamos, al principio del fichero, como variables globales, las referencias a los botones y el elemento audio que utilizaremos como reproductor.

```
const btnCreateAndTrain = document.querySelector("#btnCreateAndTrain");
const btnTestModel = document.querySelector("#btnTestModel");
```

```
const btnLoadModel = document.querySelector("#btnLoadModel");
const btnMicStart = document.querySelector("#btnMicStart");
const btnMicStop = document.querySelector("#btnMicStop");
const btnCreateSampleAudio =
document.querySelector("#btnCreateSampleAudio");



const audioPlayer = document.querySelector("#audioPlayer");
```

Ahora, en la función **app()**, podemos habilitar solamente los botones para crear y entrenar el modelo, cargar el modelo y para ver el resultado de un código de ejemplo para grabar y guardar un fichero de audio. Las demás funcionalidades se podrán utilizar únicamente después de que (i) se haya cargado un modelo para la clasificación existente o (ii) se haya creado y entrenado un modelo para la clasificación.

Creamos dos funciones para habilitar/deshabilitar botones individualmente o todos a la vez:

```
function enableButton(buttonElement, enabled) {
   buttonElement.disabled = !enabled;
}

function enableAllButtons(enabled) {
   document.querySelectorAll("button").forEach(btn => {
       btn.disabled = !enabled;
   });
}
```

Luego, en la función **app()**, añadimos el siguiente código:

```
   enableAllButtons(false);
   enableButton(btnCreateSampleAudio, true);

   yamnet = await loadYamnetModel();
   console.log("YamNet model loaded");
   enableButton(btnCreateAndTrain, true);
   enableButton(btnLoadModel, true);

   enableButton(btnTestModel, false);
   enableButton(btnMicStart, false);
   enableButton(btnMicStop, false);
```

4. Desarrollar las funcionalidades de la aplicación

Pasamos ahora a desarrollar las funcionalidades de la aplicación

4.1. Cargar el modelo preentrenado

Empezamos con la funcionalidad para cargar el modelo preentrenado existente. Asignamos al evento **click** del botón para cargar el modelo la siguiente función de callback, dentro de la función **app()**:

```
   btnLoadModel.onclick = async () => {
       model = await
loadCustomAudioClassificationModelFromFile("./model/model.json");
       enableButton(btnTestModel, true);
       enableButton(btnMicStart, true);
       enableButton(btnCreateAndTrain, false);
       enableButton(btnLoadModel, false);
   }
```

La función asíncrona **loadCustomAudioClassificationModelFromFile()** nos permitirá cargar el modelo. La definimos de la siguiente manera:

```
async function loadCustomAudioClassificationModelFromFile(url) {
   const model = await tf.loadLayersModel(url);
   model.summary();
   return model;
}
```

Ahora que hemos habilitado la funcionalidad para cargar el modelo, podemos ver el resultado en nuestra interfaz. Se nos habilitarán los botones para probar el modelo, haciendo predicciones (i) desde un fichero (utilizando los ficheros de test) y (ii) recuperando audio continuamente desde el micrófono.

Definimos también dos array globales con los nombres de las clases de sonido que vamos a clasificar:

```
const CLASSES = ["crying_baby", "clock_alarm", "toilet_flush", "water_drops"];
```

4.2. Realizar clasificaciones desde un fichero de prueba

Para realizar clasificaciones utilizando un fichero del dataset de prueba, asignamos la siguiente función al botón **btnTestModel**:

```
   btnTestModel.onclick = async () => {
       testCustomAudioClassificationModel(yamnet, model, testDataArray);
   };
```

Implementamos la función **testCustomAudioClassificationModel()**. A esta función le pasaremos el modelo YAMnet que hemos cargado previamente, el modelo para la clasificación personalizada y el dataset de test.

```
async function testCustomAudioClassificationModel(yamnet, model,
testDataArray) {
   const RANDOM = Math.floor((Math.random() * testDataArray.length));
   const testSample = testDataArray[RANDOM];
   // console.log(testSample);
   const audioData = await
getTimeDomainDataFromFile(`data/audio/${testSample.fileName}`);
   playAudio(`data/audio/${testSample.fileName}`);
   const prediction = await predict(yamnet, model, audioData);
   console.log(CLASSES[prediction]);
}
```

En esta función:

1. Seleccionamos al azar uno de los ejemplos del dataset de prueba, un elemento del array **testDataArray**. Este objeto contiene la información sobre el clip de audio, que se ha detallado en el punto 2.1.
2. Obtenemos la forma de onda del fichero de audio con la función **getTimeDomainDataFromFile().**
3. Reproducimos el sonido para así poder interpretar interactivamente el resultado de la clasificación con la función **playAudio().**
4. Realizamos la clasificación con la función **predict().**

4.2.1 Obtener la forma de onda del fichero audio

Para obtener los datos de la forma de onda (dominio del tiempo) como un array de Float32, utilizamos la siguiente función **getTimeDomainDataFromFile():**

```
async function getTimeDomainDataFromFile(url) {
   const audioContext = new AudioContext({
       sampleRate: MODEL_SAMPLE_RATE
   });
   const response = await fetch(url);
   const arrayBuffer = await response.arrayBuffer();
   const audioBuffer = await audioContext.decodeAudioData(arrayBuffer);
   audioContext.close();
   return audioBuffer.getChannelData(0);
}
```

Creamos también la siguiente variable global al principio de nuestro fichero **index.js**, donde guardaremos la frecuencia de muestreo para YAMNet en Hz (16000).

```
const MODEL_SAMPLE_RATE = 16000; // Frecuencia de muestreo para YAMNet
```

La función tendrá como parámetro la url del fichero. El objeto **AudioContext** de las Web Audio API nos permite procesar un audio guardándolo en un buffer de audio (**AudioBuffer**) con el método **decodeAudioData()**. El **AudioBuffer** decodificado se vuelve a muestrear a la frecuencia de muestreo del **AudioContext**, luego se pasa a una devolución de llamada o promesa. En este caso podríamos haber utilizado también un **OfflineAudioContext**.

Un AudioBuffer tiene propiedades como **duration**, que nos dice la duración en segundos del audio, **numberOfChannels**, un entero que representa el número de canales de audio, o **length**, un entero que representa la longitud del buffer en muestras. En nuestro caso tendremos un duración de 5 (segundos), 1 canal (mono) y longitud 80000 (16000*5).

El método **getChannelData()** devuelve un array de Float32 (Float32Array) que contiene la forma de onda (PCM) del audio. El argumento es el número del canal, en este caso 0 para el primer (y único) canal de nuestro fichero de audio.

4.2.2. Reproducir el audio

Para reproducir el fichero de audio, implementamos una simple función **playAudio()** de esta manera:

```
async function playAudio(url) {
   audioPlayer.src = url;
   audioPlayer.load();
   audioPlayer.onloadeddata = function () { audioPlayer.play(); };
}
```

La función tendrá como parámetro la url del fichero, al igual que la función anterior. Podríamos haber creado también una versión de esta función para reproducir el audio desde la forma de onda retornada por la función **getTimeDomainDataFromFile()**, pero sería más complejo. En **onloadeddata()** podemos definir la función que se llamará cuando se realiza el evento **“loadeddata”** cuando el método **load()** haya terminado de cargar los datos del audio.

4.2.3. Realizar la predicción

Por último, podemos realizar la predicción utilizando la forma de onda que acabamos de obtener del fichero y, al mismo tiempo, escuchar el contenido del fichero de audio que hemos cargado de manera aleatoria para comprobar que la clasificación es correcta. Para la predicción, implementamos una función **predict()**:

```
async function predict(yamnet, model, audioData) {
   const embeddings = await getEmbeddingsFromTimeDomainData(yamnet,
audioData);
```

```
    // embeddings.print(true);
    const results = model.predict(embeddings);
    // results.print(true)
    const meanTensor = results.mean((axis = 0));
    // meanTensor.print();
    const argMaxTensor = meanTensor.argMax(0);

    embeddings.dispose();
    results.dispose();
    meanTensor.dispose();
    return argMaxTensor.dataSync()[0];
}
```

A esta función le pasamos el modelo **yamnet**, nuestro modelo de clasificación **model** y los datos de la forma de onda **audioData**, el array de Float32 de 80000 elementos. Para obtener el resultado de la predicción tenemos primero que obtener las características (embeddings) de YAMNet y pasarlos como entrada al modelo de clasificación.

Para obtener los embeddings, implementamos una función **getEmbeddingsFromTimeDomainData()**:

```
async function getEmbeddingsFromTimeDomainData(model, audioData) {
    const waveformTensor = tf.tensor(audioData);
    const [scores, embeddings, spectrogram] = model.predict(waveformTensor);
    waveformTensor.dispose();
    return embeddings;
}
```

Creamos un tensor para dárselo como entrada a YAMNet. De la salida de YAMNet, el resultado del método **predict()**, nos quedamos con el tensor **embeddings** de forma [10,1024]. Este es el tensor que pasaremos al método **predict()** del modelo de clasificación, que sustituye la última capa de YAMNet y nos permite realizar las clasificaciones sobre las categorías personalizadas.

El resultado de la clasificación es un tensor de forma [10, 4] con la clasificación para cada uno de los 10 fotogramas de YAMNet (0,96 segundos con intervalo de 0,48 segundos en un audio de 5 segundos). Para obtener un valor final, calculamos la media agregada. Luego, utilizando tf.argMax() podemos reducir el tensor con las medias agregadas para quedarnos con la posición del valor máximo, que será la clasificación que corresponde a la posición de la clase en el array **CLASSES**.

4.2.3. Crear y guardar audio

Se ha incluido también, como ejemplo, una funcionalidad para crear y guardar un fichero de audio en formato .**wav** utilizando las Web Audio API. La función **createAudio()**, que

asignamos como función de devolución de llamada del evento **click** del botón **btnCreateSampleAudio**, tiene como parámetro los milisegundos de duración del audio que queremos crear y genera y guarda en un fichero **millis** milisegundos de audio generado aleatoriamente (ruido). El código es el siguiente:

```
    btnCreateSampleAudio.onclick = async () => {
        createAudio(5000);
    };
```

```
function createAudio(millis) {
    const audioData = new Float32Array(Array.from(
        {
            length: MODEL_SAMPLE_RATE * millis / 1e3
        }, () => Math.random() * 2 - 1));

    const wavBytes = getWavBytes(audioData.buffer);

    const blob = new Blob([wavBytes], { 'type': 'audio/wav' });
    const audioURL = window.URL.createObjectURL(blob);
    const a = document.createElement("a");
    a.href = audioURL;
    a.download = "test.wav";
    a.click();
}
```

Creamos nuestro fichero con audio al azar utilizando la misma frecuencia de muestreo de nuestro modelo (16 kHz), por lo que el array con los datos de audio contendrá **16000 * millis / 1000** elementos de tipo Float32 (80000 en nuestro caso, con audio de 5 segundos). El método Array.from() permite crear un array rellenando cada elemento con el valor retornado por una función que pasamos como segundo argumento. El primer argumento es el objeto que convertimos a array, tiene que ser un objeto *array-like*, es decir un objeto con una propiedad **length** y elementos indexados.

Los array tipados en JavaScript tiene una propiedad **buffer** que es un objeto **ArrayBuffer** que se utiliza para representar un búfer de datos binarios sin formato de longitud fija genérico.

Para poder guardar el buffer de audio como un fichero **wav**:

1. Utilizamos la función **getWavBytes()** para obtener un array de bytes en formato wav. Esta función llamará a una función **getWavHeader()** para escribir la cabecera del audio en formato wav. Las dos funciones se han adaptado, simplificandolas, de las funciones en esta entrada de StackOverlfow. La cabecera contiene información específica del formato WAV y del audio con el que estamos trabajando: 1 canal (mono), frecuencia de muestreo 16 kHz, tipo de dato Float32 (4 bytes por muestra).

Por ejemplo, si utilizamos Float32, el número del formato WAV es 3, y si utilizamos UInt16 sería 1 (https://i.stack.imgur.com/BuSmb.png).

2. El fichero WAV resultado se hará disponible a través del método estático de las Web API **URL.createObjectURL()** que permite crear una URL que se puede utilizar para hacer referencia al objeto que le pasamos como argumento. Para crear un objeto a partir de los datos WAV, utilizamos el objeto Blob de las Web API, que se utiliza para crear objetos con datos binarios.
3. La URL se puede asignar a la propiedad **href** de un elemento ancla (anchor) **a**. Podemos simular un clic en el elemento llamando al método click de HTMLElement y así activar automáticamente la descarga. El atributo **download** especifica el nombre del fichero que descargamos.

A continuación se muestra el código de las dos funciones **getWavBytes()** y **getWavHeader()**:

```
function getWavBytes(buffer) {
   const numFrames = buffer.byteLength / Float32Array.BYTES_PER_ELEMENT;
   const headerBytes = getWavHeader(numFrames);
   const wavBytes = new Uint8Array(headerBytes.length + buffer.byteLength);

   // prepend header, then add pcmBytes
   wavBytes.set(headerBytes, 0);
   wavBytes.set(new Uint8Array(buffer), headerBytes.length);

   return wavBytes;
}

function getWavHeader(numFrames) {
   const numChannels = 1;
   const bytesPerSample = 4;

   const format = 3; //Float32

   const blockAlign = numChannels * bytesPerSample;
   const byteRate = MODEL_SAMPLE_RATE * blockAlign;
   const dataSize = numFrames * blockAlign;

   const buffer = new ArrayBuffer(44);
   const dv = new DataView(buffer);

   let p = 0;

   function writeString(s) {
```

```
        for (let i = 0; i < s.length; i++) {
            dv.setUint8(p + i, s.charCodeAt(i));
        }
        p += s.length;
    }

    function writeUint32(d) {
        dv.setUint32(p, d, true);
        p += 4;
    }

    function writeUint16(d) {
        dv.setUint16(p, d, true);
        p += 2;
    }

    writeString('RIFF');              // ChunkID
    writeUint32(dataSize + 36);       // ChunkSize
    writeString('WAVE');              // Format
    writeString('fmt ');              // Subchunk1ID
    writeUint32(16);                  // Subchunk1Size
    writeUint16(format);              // AudioFormat
    writeUint16(numChannels);         // NumChannels
    writeUint32(MODEL_SAMPLE_RATE);   // SampleRate
    writeUint32(byteRate);            // ByteRate
    writeUint16(blockAlign);          // BlockAlign
    writeUint16(bytesPerSample * 8);  // BitsPerSample
    writeString('data');              // Subchunk2ID
    writeUint32(dataSize);            // Subchunk2Size

    return new Uint8Array(buffer);
}
```

5. Clasificar sonidos desde el micrófono

Hemos visto que podemos probar el modelo clasificando los audio en los ficheros del dataset de prueba que hemos creado separando utilizando la partición (fold) número 5 del resto de particiones que se utilizaron previamente para entrenar el modelo. Para esto utilizamos la función `testCustomAudioClassificationModel()`, como se ha explicado en la Sección 4.2. Ahora, realizaremos las clasificaciones de manera continua, utilizando el micrófono como fuente de audio.

5.1. Activar el micrófono y empezar la clasificación

Para empezar a clasificar los eventos sonoros desde el micrófono, implementaremos una función que asignaremos al evento **click** del botón **btnMicStart**. Esta función se encargará de:

1. Activar y obtener el flujo de audio desde el micrófono, con la misma frecuencia de muestreo que necesita el modelo.
2. Recupera fotogramas de audio y guardarlos en una cola (función **onAudioFrame()**)
3. Realizar la predicción después de un número de fotograma establecidos, llamando a la función **predict()**.

Primero, necesitamos declarar algunas constantes globales para definir los parámetros para el procesamiento audio necesario para la aplicación.

```
/*
* Parámetros para el procesamiento de audio
*/
const MODEL_SAMPLE_RATE = 16000; // Frecuencia de muestreo para YAMNet
const NUM_SECONDS = 3; // Número de segundos para el muestreo desde mic
const OVERLAP_FACTOR = 0.0; // Factor de superposición de los fotogramas
```

El código de la función, que escribimos dentro de la funciòn principal **app()**, es el siguiente:

```
   let audioContext;
   let stream;

   const timeDataQueue = [];

    btnMicStart.onclick = async () => {

       stream = await getAudioStream();
       audioContext = new AudioContext({
           latencyHint: "playback",
           sampleRate: MODEL_SAMPLE_RATE
       });

       const streamSource = audioContext.createMediaStreamSource(stream);

       await audioContext.audioWorklet.addModule("recorder.worklet.js");
       const recorder = new AudioWorkletNode(audioContext,
"recorder.worklet");
       streamSource.connect(recorder).connect(audioContext.destination);
```

```
        enableButton(btnMicStart, false);
        enableButton(btnMicStop, true);

        recorder.port.onmessage = async (e) => {
            const inputBuffer = Array.from(e.data);

            if (inputBuffer[0] === 0) return;

            timeDataQueue.push(...inputBuffer);

            const num_samples = timeDataQueue.length;
            if (num_samples >= MODEL_SAMPLE_RATE * NUM_SECONDS) {
                const audioData = new Float32Array(timeDataQueue.splice(0,
MODEL_SAMPLE_RATE * NUM_SECONDS));
                console.log(CLASSES[await predict(yamnet, model,
audioData)]);
            }
        }
    };
```

La función **getAudioStream()** permite recuperar el flujo de audio del micrófono con los parámetro necesarios:

```
async function getAudioStream(audioTrackConstraints) {
    let options = audioTrackConstraints || {};
    try {
        return await navigator.mediaDevices.getUserMedia({
            video: false,
            audio: {
                sampleRate: options.sampleRate || MODEL_SAMPLE_RATE,
                sampleSize: options.sampleSize || 16,
                channelCount: options.channelCount || 1
            }
        });
    } catch (e) {
        console.error(e);
        return null;
    }
}
```

El código de la función para recuperar los datos de audio desde el micrófono utiliza el API Worklet, una versión ligera de la API Web Workers, que representa una tarea en segundo plano que se puede crear a través de un script, que puede enviar mensajes a su creador. El API AudioWorklet, se utiliza para proporcionar secuencias de comandos de procesamiento de audio personalizadas que se ejecutan en un subproceso separado para proporcionar un procesamiento de audio de muy baja latencia. Esta API reemplaza el API deprecada ScriptProcessorNode, que tenía fallos de diseño.

Primeramente creamos un objeto AudioContext que representa el grafo para el procesamiento audio; este objeto tendrá la misma frecuencia de muestreo de 16 kHz que necesitamos para el modelo. Con el método **createMediaStreamSource()** creamos la fuente de audio que queremos manipular, pasándole el flujo de audio del micrófono creado anteriormente con nuestra la función **getAudioStream().** Con el AudioContext, creamos un nuevo **AudioWorkletNode** que realizará las operaciones en el hilo separado. Este nodo tiene asociado un **AudioWorkletProcessor** donde se define el código personalizado para el procesamiento del audio. Este código está en el fichero **recorder.worklet.js** y se muestra a continuación:

```
// https://gist.github.com/louisgv/f210a1139d955baf511ff35f58fc8db1

class RecorderProcessor extends AudioWorkletProcessor {
   bufferSize = 4096;
   _bytesWritten = 0;

   _buffer = new Float32Array(this.bufferSize);

   constructor() {
       super();
       this.initBuffer();
   }

   initBuffer() {
       this._bytesWritten = 0;
   }

   isBufferEmpty() {
       return this._bytesWritten === 0;
   }

   isBufferFull() {
       return this._bytesWritten === this.bufferSize;
   }

   process(inputs) {
       this.append(inputs[0][0]);
       return true;
   }
```

```
    append(channelData) {
        if (this.isBufferFull()) {
            this.flush();
        }

        if (!channelData) return;

        for (let i = 0; i < channelData.length; i++) {
            this._buffer[this._bytesWritten++] = channelData[i];
        }
    }

    flush() {
        this.port.postMessage(
            this._bytesWritten < this.bufferSize
                ? this._buffer.slice(0, this._bytesWritten)
                : this._buffer
        )
        this.initBuffer();
    }

}

registerProcessor("recorder.worklet", RecorderProcessor);
```

Este código almacenará en un buffer los bytes que recibe del micrófono, hasta rellenar el buffer (hemos puesto 4096 como tamaño, teniendo que ser este valor potencia de 2). Cuando el buffer esté lleno, lo vacía y envía un mensaje con el buffer al código creador, que se captura con el método asignado al evento onmessage. En esta función, cada vez que recibimos un buffer de audio, lo guardamos en un array **timeDataQueue** que utilizamos para leer un número de muestras de audio que depende del valor de segundos que deseamos. Por ejemplo, si quisiéramos activar la clasificación cada 3 segundos de audio, tendríamos que leer 16000 * 3 = 48000 muestras. Cuando la cola timeDataQueue se llena, la vaciamos y activamos la predicción del modelo.

Por lo tanto, en nuestro **AudioWorkletProcessor** personalizado (un objeto **RecorderProcessor**) vamos leyendo bytes de audio conforme vayan llegando, con una latencia mínima. Cada 4096 los pasamos al hilo principal que los va guardando en una cola. Cuando la cola llega a 48000 bytes (49152 en realidad, 12 * 4096, nos quedamos con los primeros 48000) la vaciamos y activamos el modelo de clasificación con la forma de onda contenida en la cola.

5.2. Desactivar el micrófono y parar la clasificación

Para interrumpir el proceso de clasificación desde el micrófono y desactivar el micrófono, asignamos una función de devolución de llamada al evento **onclick** del elemento **btnMicStop**. Esta función , que se muestra a continuación, cerrará el AudioContext y el flujo de audio correspondiente, vaciará la cola de datos de audio, y volverá a habilitar el botón para recuperar audio desde el micrófono en la interfaz, que se había deshabilitado previamente.

```
  btnMicStop.onclick = () => {
      if (!Boolean(audioContext) || !Boolean(stream)) return;
      audioContext.close();
      audioContext = null;

      timeDataQueue.splice(0);
      if (stream != null && stream.getTracks().length > 0) {
          stream.getTracks()[0].stop();
          enableButton(btnMicStart, true);
          enableButton(btnMicStop, false);
      }
  }
```

6. Crear un clasificador para eventos de audio personalizados

En esta sección se describirán los detalles sobre la creación de la red de neuronas para la clasificación de las cuatro clases de eventos de audio, para que sirva como base para la creación de eventos personalizados.

6.1. Preparar el proceso

En primer lugar, asignamos la siguiente función al evento onclick del botón **btnCreateAndTrain**:

```
  btnCreateAndTrain.onclick = async () => {
      enableAllButtons(false);
      model = await createAndTrainCustomAudioClassificationModel(yamnet,
trainDataArray);
      enableAllButtons(true);
      enableButton(btnCreateAndTrain, false);
      enableButton(btnLoadModel, false);
  };
```

En esta función, además de gestionar los elementos de la interfaz (botones), por ejemplo inhabilitandolos mientras dure el proceso de creación y entrenamiento del modelo, llamaremos a la función **createAndTrainCustomAudioClassificationModel()** que se

encargará de crear y entrenar el modelo de clasificación para nuestra cuatro clases, pasandole el modelo YAMNet, que utilizaremos para extraer las características de los ficheros de audio, y el dataset con los ficheros de audio de entrenamiento. A continuación se muestra el código de la función:

```
async function createAndTrainCustomAudioClassificationModel(yamnet,
trainDataArray) {

   const [inputTensor, outputAsOneHotTensor] = await
prepareTrainingData(yamnet, trainDataArray, "data/audio");
   // outputAsOneHotTensor.print(true);

   const model = createModel();
   await trainModel(model, inputTensor, outputAsOneHotTensor);
   await saveModel(model);
   return model;
}
```

```
async function prepareTrainingData(yamnet, metadata, audioBasePath) {
   const INPUT_DATA = [];
   const OUTPUT_DATA = [];

   for (const { fileName, classNumber } of metadata) {
       const audioData = await
getTimeDomainDataFromFile(`${audioBasePath}/${fileName}`);
       const embeddings = await getEmbeddingsFromTimeDomainData(yamnet,
audioData);
       const embeddingsArray = embeddings.arraySync();

       for (const embedding of embeddingsArray) {
           INPUT_DATA.push(embedding);
           OUTPUT_DATA.push(classNumber);
       }

       embeddings.dispose();
   }

   tf.util.shuffleCombo(INPUT_DATA, OUTPUT_DATA);

   const inputTensor = tf.tensor2d(INPUT_DATA);
```

```
    const outputAsOneHotTensor = tf.oneHot(tf.tensor1d(OUTPUT_DATA,
'int32'), NUM_CLASSES);

    return [inputTensor, outputAsOneHotTensor];
}
```

En la función **prepareTrainingData()**, antes de crear y entrenar el modelo, creamos dos array con los datos de entrada y salida (clasificación) para el entrenamiento. Para esto, obtenemos la forma de onda de cada uno de los ficheros del dataset de entrenamiento y su clase correspondiente. La clase la guardamos en el array **OUTPUT_DATA**, mientras que en el array **INPUT_DATA** guardaremos los embeddings obtenidos de YAMNet para cada forma de onda. Mezclamos los dos array con el método **tf.util.shuffleCombo()**, para mantener la misma relación de los índices en los dos array, y luego creamos los dos tensores de entrada y salida para el entrenamiento. El tensor de entrada será un tensor 2D [TRAIN_SAMPLES * 10, 1024]. En este caso, para cada una de las 4 clases tenemos 40 ejemplos, y 8 de ellos se han destinado a las pruebas, por lo que tenemos 32 por clase, por un total de 128 ejemplos. Para cada ejemplo tenemos 10 embeddings de 1024.

6.2. Crear el modelo

Para crear el módelo, llamamos a la función **createModel**:

```
function createModel() {
    const model = tf.sequential();

    model.add(tf.layers.dense({ dtype: 'float32', inputShape: [INPUT_SHAPE],
units: 512, activation: 'relu' }));
    model.add(tf.layers.dense({ units: NUM_CLASSES, activation: 'softmax'
}));
    model.summary();
    return model;
}
```

Esta función crea y retorna un simple modelo de dos capas completamente conectadas que clasifica (utilizando softmax) la entrada en **NUM_CLASSES** que en nuestro caso es 4. Habíamos definido esta variable global previamente al principio de nuestro fichero, junto con la variable **INPUT_SHAPE** con el número de características retornadas por YAMNet.

```
const INPUT_SHAPE = 1024;
const NUM_CLASSES = 4;
```

6.3. Entrenar el modelo

Para entrenar el modelo, llamamos a la función **trainModel()**:

```
async function trainModel(model, inputTensor, outputTensor) {
   model.compile({
       optimizer: 'adam',
       loss: 'categoricalCrossentropy',
       metrics: ['accuracy']
   });

   const params = {
       shuffle: true,
       validationSplit: 0.15,
       batchSize: 16,
       epochs: 20,
       callbacks: [new tf.CustomCallback({ onEpochEnd: logProgress })
     ]
   };

   const results = await model.fit(inputTensor, outputTensor, params);
   console.log("Average error loss: " +
Math.sqrt(results.history.loss[results.history.loss.length - 1]));
   console.log("Average validation error loss: " +
       Math.sqrt(results.history.val_loss[results.history.val_loss.length - 1]));
}
```

A esta función le pasamos el modelo y los dos array con los datos de entrenamiento de entrada (embeddings) y salida (su clasificación). Utilizamos como función de pérdida Categorical Cross-Entropy (Softmax) porque queremos clasificar en cuatro clases diferentes. Utilizaremos el optimizador adam, un algoritmo de descenso de gradiente estocástico. Utilizamos el 15% del dataset para la validación (atributo **validationSplit**). El método **tf.LayersModel.fit()** entrenará el modelo para un número fijo de iteraciones en el conjunto de datos (epochs). Ponemos 20 epochs. Con la función de callback logProgress asociada al avento onEpochEnd podemos monitorizar el resultado del proceso de entrenamiento en cada iteración. Al final del proceso de entrenamiento podemos visualizar información como el error medio y el error de validación medio.

6.4 Guardar el modelo

Para guardar el modelo descargándolo automáticamente, llamamos a la función **saveModel()**:

```
async function saveModel(model) {
   model.save('downloads://model');
}
```

Esta función llama a **model.save("downloads://model")**; al final del código de la función principal **createAndTrainCustomAudioClassificationModel()**. Esta funcionalidad nos

permitirá reutilizar el modelo preentrenado sin tener que volver a crear y entrenar cada vez que lo queremos utilizar en una aplicación. Ya hemos aprendido en la Sección 4.1 cómo cargar un modelo preentrenado.

7.(Extra) Visualizar forma de onda y espectrograma

A continuación, se introduce brevemente como implementar una funcionalidad que permita visualizar una forma de onda y su correspondiente espectrograma Mel utilizando la librería tfjs-vis de TensorFlow.js. También incluye una interfaz de usuario que permite habilitar o deshabilitar las visualizaciones de manera dinámica. tfjs-vis es una librería de visualización para TensorFlow.js que permite inspeccionar y analizar datos, modelos y métricas de entrenamiento en tiempo real, ofreciendo herramientas como gráficos de líneas, histogramas, mapas de calor, entre otros, para facilitar el desarrollo y la depuración de proyectos de aprendizaje automático en el navegador.

Primeramente, incluimos el código de la librería en nuestro fichero HTML, después de incluir el código para TensorFlow.js. Este código nos dará acceso al API de la librería tfjs-vis a través del objeto **tfvis**.

```
 <!-- TensorFlow.js library -->
 <script src="https://cdn.jsdelivr.net/npm/@tensorflow/tfjs@latest"
type="text/javascript"></script>
  <!-- TFJ-VIS library-->
 <script src="https://cdn.jsdelivr.net/npm/@tensorflow/tfjs-vis@latest"
type="text/javascript"></script>
</head>
```

7.1. Definir la interfaz de usuario

Creamos un botón para mostrar o ocultar el Visor de tfjs-vis, una ventana donde se muestran gráficos y estadísticas. Creamos también un checkbox para habilitar o deshabilitar la visualización.

```
  <div class="ui-div">
    <input type="checkbox" id="checkViz" name="checkViz" />
    <label for="checkViz">Enable visualization</label>
  </div>
  <div class="ui-div">
    <button class="ui-btn" id="btnToggleViz">Show/Hide</button>
  </div>
```

En el fichero JavaScript **index.js** escribiremos el código para mostrar/ocultar el visor y para activar/desactivar la visualización automática en cada intento de clasificación.

```
const btnToggleViz = document.querySelector("#btnToggleViz");
btnToggleViz.addEventListener("click", () => {
```

```
    tfvis.visor().toggle();
});
```

El evento click del botón **btnToggleViz** activa el método [tfvis.visor().toggle()](), que alterna la visibilidad del visor. **tfvis.visor()** es la interfaz principal para el visor, que a devuelve una instancia única de la clase [Visor]().

```
let visualizationEnabled = false;

const checkViz = document.querySelector("#checkViz");
checkViz.addEventListener("change", () => {
    visualizationEnabled = checkViz.checked;
    enableButton(btnToggleViz, visualizationEnabled);
});
```

El checkbox **checkViz** habilita o deshabilita la visualización. Si el checkbox está marcado, la variable visualizationEnabled se actualiza y habilita el botón **btnToggleViz**.

7.2. Definir la funcionalidad de visualización

La función **visualizeData()** es el núcleo del código, y realiza dos visualizaciones principales: forma de onda y espectrograma mel.

```
async function visualizeData(audioData, spectrogram, predictions) {
    // Visualize the waveform
    enableButton(btnToggleViz, true);
    tfvis.render.linechart(
        { name: 'Waveform' },
        { values: Array.from(audioData).map((y, x) => ({ x, y })) },
        { width: 600, height: 200 }
    );

    // Visualize the spectrogram
    const spectrogramArray = spectrogram.arraySync();
    console.log(spectrogram.shape);
    const xTickLabels = Array.from({ length: spectrogramArray.length }, (_,
i) => `${i}`);
    const yTickLabels = Array.from({ length: 64 }, (_, i) => `${64- i}`);
    tfvis.render.heatmap(
        { name: 'Mel Spectrogram' },
        { values: spectrogramArray, xTickLabels: xTickLabels, yTickLabels:
yTickLabels},
```

```
        { width: 600, height: 400, xLabel: 'Time frames', yLabel: 'Mel
bins', domain: [Math.min(...spectrogramArray.flat()),
Math.max(...spectrogramArray.flat())] }
    );
}
```

La función requiere los siguientes parámetros:

- **audioData**: un array de datos de audio (forma de onda) que representa las amplitudes a lo largo del tiempo.
- **spectrogram**: un tensor de dos dimensiones que contiene el espectrograma mel del audio, con fotogramas de tiempo que representan segmentos del audio (eje x), y mel bins que representan frecuencias en escala Mel (eje y)

Para la forma de onda, la función convierte el array **audioData** en un array de puntos { x, y } donde **x** es el índice (marco temporal), mientras **y** es el valor de amplitud correspondiente. El método **tfvis.render.linechart()** de tfjs-vis se utiliza para crear gráficos de líneas y requiere una serie de valores específicos para configurarse correctamente. Los parametros necesarios son tres objetos:

1. Nombre y Opciones del Gráfico
2. Valores de datos: **values** es un array de objetos que representan los puntos del gráfico. Cada objeto debe tener: **x** (valor en el eje x), **y** (valor en el eje y).
3. Opciones de configuración (opcional). Es un objeto con las siguientes propiedades, entre otras:
    a. width: Ancho del gráfico en píxeles
    b. height: Alto del gráfico en píxeles

Para la visualización del espectrograma utilizamos **tfvis.render.heatmap()**, que renderiza un mapa de calor para visualizar el espectrograma mel.Primer se convierte el tensor **spectrogram** en un array de valores 2D (**spectrogramArray**) para pasarlo al método. Se podría utilizar también el tensor directamente. El método necesita los mismo parámetros que el método **tfvis.render.linechart()**. Adicionalmente, en el objetos con los valores de datos, se necesitan dos atributos:

- xTickLabels: Representan los marcos de tiempo, numerados de 0 a N-1.
- yTickLabels: Representan los bins mel (64), invertidos para que el bin más bajo esté cerca del eje X.

En las opciones de configuración, utilizamos el atributo **domain** para ajustar el rango de colores de la representación en decibelios, calculando los valores mínimo y máximo del espectrograma.

Añadimos la llamada a la función **visualizeData()** en la función **getEmbeddingsFromTimeDomainData()** para poder visualizar la forma de onda y espectrograma en cada ejecución del modelo YAMNet para extraer las representaciones, dependiendo del valor de la variable **visualizationEnabled**.

```
async function getEmbeddingsFromTimeDomainData(model, audioData) {
   const waveformTensor = tf.tensor(audioData);
   // waveformTensor.print(true);
   const [scores, embeddings, spectrogram] = model.predict(waveformTensor);

   if (visualizationEnabled) visualizeData(audioData, spectrogram);

   waveformTensor.dispose();
   return embeddings;
}
```

Ejercicios

1. Modificar el código de la función **micStartBtn.onclick** para capturar audio teniendo en cuenta el valor del factor de superposición. Por ejemplo, capturar audio cada 3 segundos con una superposición de 1.5 segundos, es decir con un valor del factor de superposición de 0.5. En este caso, se debería activar el proceso de detección cada 1.5 en lugar de 3 y actualizar la cola de audio en consecuencia…
2. Añadir una nueva categoría desde:
    a. El dataset ESC50
    b. Datos grabados y etiquetados manualmente