

PROCESAMIENTO DE SEÑALES UNIDIMENSIONALES CON MODELOS AUTO REGRESIVOS, CONVOLUCIÓN 1D, CONVOLUCIÓN 2D USANDO EL ESPECTROGRAMA Y REDES RECURRENTES

Diego Iván Perea Montealegre, Carlos Ivan Osorio Moreno, Gabriel Jeannot Viaña

Resumen : En este taller se realiza la creación y evaluación de múltiples modelos de Deep Learning para abordar tareas específicas, incluyendo predicción de series temporales, clasificación de movimiento y clasificación de audio, con un enfoque en la implementación utilizando TensorFlow-Keras y la verificación de modelos en dispositivos móviles mediante Edge Impulse.

INTRODUCCIÓN

El procesamiento de señales unidimensionales es un campo fascinante que se ocupa de la manipulación y análisis de señales, que son representaciones matemáticas de fenómenos físicos. Los modelos autorregresivos (AR)[1] son una herramienta poderosa en este ámbito, ya que permiten predecir valores futuros de una señal basándose en sus valores pasados.

La convolución 1D es una operación matemática fundamental en el procesamiento de señales. Permite combinar dos señales para producir una tercera, proporcionando una forma de ‘mezclar’ información en el dominio del tiempo o del espacio. Por otro lado, la convolución 2D es una extensión natural de la convolución 1D al caso bidimensional. Es especialmente útil en el procesamiento de imágenes, donde las señales son funciones de dos variables espaciales.[2]

El espectrograma es una representación visual de la frecuencia y la intensidad de una señal a

lo largo del tiempo. Es una herramienta invaluable para analizar señales no estacionarias, es decir, señales cuyas propiedades estadísticas cambian con el tiempo.

Las redes neuronales recurrentes (RNN) son un tipo de modelo de aprendizaje profundo que es particularmente adecuado para el procesamiento de secuencias temporales, como las señales unidimensionales. Las RNN tienen la capacidad única de ‘recordar’ información del pasado, lo que las hace muy efectivas para tareas como la predicción de series temporales y el reconocimiento de patrones en datos secuenciales.

METODOLOGÍA

Para trabajar el punto número 1 donde se solicita seleccionar una serie temporal de cualquier repositorio y realizar algunas precisiones entrenando tres modelos basados en redes neuronales profundas puntualmente MLP, CNN y LSTM. Como datos para realizar estas predicciones se tomó un conjunto de datos sobre el número de pasajeros aéreos por fechas, el cual cuenta con dos columnas (Months, #passengers).

Estas son las librerías utilizadas para el desarrollo del proyecto:

```
[ ] import numpy as np
import tensorflow as tf
import pandas as pd
from tensorflow import keras
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.datasets import load_diabetes
import matplotlib.pyplot as plt
```

En este punto, se realiza la carga del dataset Airpassengers.csv y un proceso de normalización para los datos, adicionalmente se realiza una conversión de datos.

```
❶ # Cargar datos 'Airpassenger.csv'
data = pd.read_csv('AirPassengers.csv')
print(data.to_string())

# Columnas Dataset
data.columns = ['Month', 'Passengers']

# Convertir la columna 'Month' a un formato de fecha
data['Month'] = pd.to_datetime(data['Month'], format='%y-%m')

# Convertir las fechas en números enteros
data['Month'] = pd.to_numeric(data['Month'])

# Normalizar los números
data['Month'] = (data['Month'] - data['Month'].min()) / (data['Month'].max() - data['Month'].min())
```

Durante este punto se realiza la división de los datos de entrenamiento y de prueba.

```
[ ] # Dividir datos en conjuntos de entrenamiento y prueba
X = data['Month']
y = data['Passengers']

X_train, X_test, y_train, y_test = train_test_split(X[:-1], y[1:], test_size=0.2, random_state=42)

❷ # tipo NumPy
X_train = X_train.values.reshape(-1, 1)
y_train = y_train.values.reshape(-1, 1)
```

En este punto se realiza la construcción de un modelo de red multi capas profundas MLP.

```
❸ model_mlp = keras.Sequential([
    keras.layers.Dense(128, activation='relu', input_shape=(1,)),
    keras.layers.Dense(64, activation='relu'),
    keras.layers.Dense(32, activation='relu'),
    keras.layers.Dense(16, activation='relu'),
    keras.layers.Dense(1)
])
model_mlp.compile(optimizer='adam', loss='mean_squared_error')
# Entrenar el modelo MLP
model_mlp.fit(X_train, y_train, epochs=100, verbose=1)
```

Para evaluar la calidad de un modelo de regresión, es apropiado utilizar métricas como el error cuadrático medio (MSE), el coeficiente de determinación (R^2). Estas métricas y visualizaciones son más relevantes para entender qué tan bien está funcionando tu modelo de regresión.

```
❹ # Evaluar el modelo en el conjunto de prueba
X_test = X_test.to_numpy().reshape(-1, 1)
y_pred_mlp = model_mlp.predict(X_test)

mse_mlp = mean_squared_error(y_test, y_pred_mlp)
r2_mlp = r2_score(y_test, y_pred_mlp)

print("Modelo MLP - error cuadrático medio (MSE):", mse_mlp)
print("Modelo MLP - coeficiente de determinación ( $R^2$ ):", r2_mlp)
```

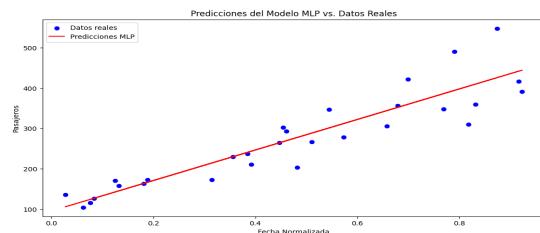
De acuerdo a los datos R^2 de 0.8332, indica que el modelo tiene un buen poder predictivo,

ya que explica más del 83% de la variabilidad en los datos de salida. Esto sugiere que el modelo MLP es efectivo para predecir los valores de salida basados en las características de entrada.

Por otro lado el Error cuadrático medio es 2139, el cuál se logró reducir usando más capas y aumentando las épocas pero podría mejorar aumentando más datos.

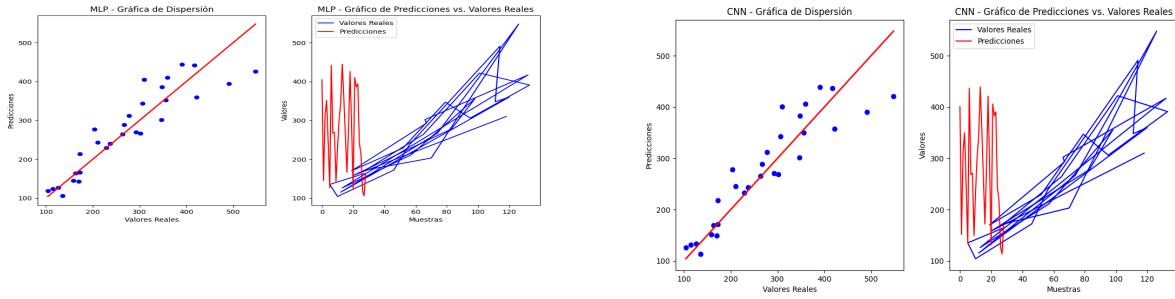
```
1/1 [=====] - 0s 249ms/step
Modelo MLP - error cuadrático medio (MSE): 2139.000763847505
Modelo MLP - coeficiente de determinación ( $R^2$ ): 0.8331976175697168
```

En la siguiente gráfica, se puede evidenciar que la línea roja representa las predicciones hechas por el modelo MLP y los puntos azules representan los datos reales. Lo ideal es que la línea roja esté lo más cerca posible de los puntos azules, lo que indicaría que las predicciones del modelo se alinean bien con los datos reales.



En nuestra gráfica de la izquierda, el caso ideal es que los puntos estén cerca de la línea roja diagonal, lo que indica que las predicciones son cercanas a los valores reales. Si los puntos se alejan de esta línea, indica que las predicciones no son tan precisas.

Pero para la gráfica de la derecha la idea es que las líneas azules (predicciones) sigan el patrón de las líneas rojas (valores reales). Si las líneas azules y rojas están cerca, indica que las predicciones son precisas. Si hay mucha divergencia entre las líneas, indica que las predicciones no son precisas en esas muestras.



Ahora en este punto vamos a realizar un entrenamiento utilizando los mismo datos pero cambiando a una convolución 1D

```
( ) model_cnn = keras.Sequential([
    keras.layers.Conv1D(128, kernel_size=1, activation='relu', input_shape=X_train_cnn.shape[1], 1),
    keras.layers.MaxPooling1D(pool_size=1),
    keras.layers.Conv1D(64, kernel_size=1, activation='relu'),
    keras.layers.MaxPooling1D(pool_size=1),
    keras.layers.Flatten(),
    keras.layers.Dense(64, activation='relu', kernel_regularizer=keras.regularizers.l2(0.01)),
    keras.layers.Dense(1)
])
model_cnn.compile(optimizer='adam', loss='mean_squared_error')

# Entrenar el modelo CNN
model_cnn.fit(X_train_cnn, y_train_cnn, epochs=100, verbose=1)
```

Al mirar estos resultados podemos evidenciar que son muy parecidos al modelo MLP con la diferencia que nuestro error cuadrático es de unos puntos mayor.

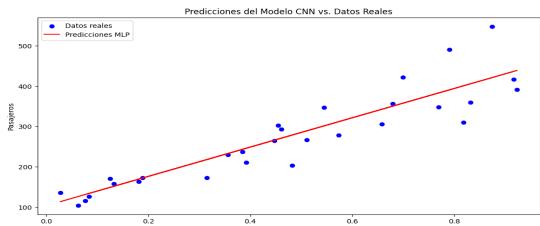
```
( ) # Evaluar el modelo en el conjunto de prueba
X_test_cnn = X_test.reshape(-1, 1)
y_pred_cnn = model_cnn.predict(X_test_cnn)

mse_cnn = mean_squared_error(y_test, y_pred_cnn)
r2_cnn = r2_score(y_test, y_pred_cnn)

print("Modelo cnn - error cuadrático medio (MSE):", mse_cnn)
print("Modelo cnn - coeficiente de determinación (R^2):", r2_cnn)
```

1/1 [=====] - 0s 131ms/step
 Modelo cnn - error cuadrático medio (MSE): 2141.8094605449364
 Modelo cnn - coeficiente de determinación (R^2): 0.832978591326915

Las siguientes gráficas tiene mucha relación en el modelo anterior, con la diferencia que los puntos están más cerca de la línea en el modelo MLP.



En este punto, se puede observar el entrenamiento de un modelo LSTM usando solo 3 capas.

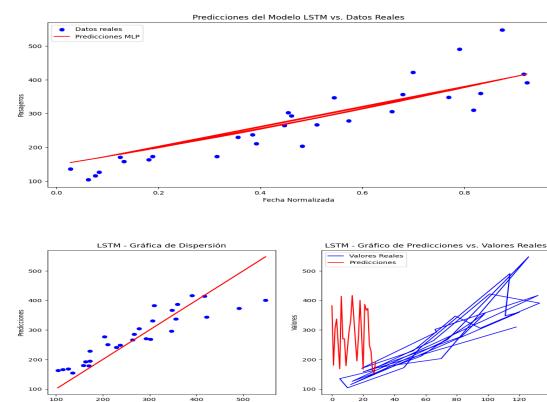
```
( ) # Crear un modelo basado en redes recurrentes (LSTM)
model_rnn = keras.Sequential([
    keras.layers.LSTM(64, activation='relu', return_sequences=True, input_shape=(1, 1)),
    keras.layers.LSTM(32, activation='relu'),
    keras.layers.Dense(1)
])
model_rnn.compile(optimizer='adam', loss='mean_squared_error')
# model_rnn.fit(X_train.reshape(-1, 1, 1), y_train, epochs=100, verbose=0)
model_rnn.fit(X_train_LSTM, y_train_LSTM, epochs=100, verbose=1)
```

Para este modelo, se puede evidenciar que los resultados van desmejorando en comparación con los modelos anteriores.

```
( ) # Evaluar el modelo en el conjunto de prueba
X_test_rnn = X_test.reshape(-1, 1)
y_pred_rnn = model_rnn.predict(X_test_rnn)

mse_rnn = mean_squared_error(y_test, y_pred_rnn)
r2_rnn = r2_score(y_test, y_pred_rnn)

print("Modelo rnn - error cuadrático medio (MSE):", mse_rnn)
print("Modelo rnn - coeficiente de determinación (R^2):", r2_rnn)
1/1 [=====] - 0s 283ms/step
Modelo rnn - error cuadrático medio (MSE): 2668.878488561479
Modelo rnn - coeficiente de determinación (R^2): 0.796555871608947
```



Para concluir, el modelo MLP tiene el mejor desempeño, ya que explica aproximadamente el 83.3% de la variación en los datos. Sin embargo, hay espacio para mejorar, ya que no todas las predicciones están perfectamente alineadas con los datos reales.

Empezando el segundo punto, se seleccionó cinco categorías, las cuales están relacionadas al boxeo, debido a que se proyectó a detectar estos movimientos particulares para entrenar en este deporte. Tenemos las siguientes categorías:

Normal: Brazos y manos a lado del torso relajadas.



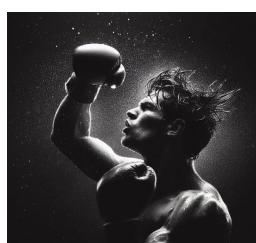
Defense: Brazos en modo defensa en donde los puños están por debajo de la mandíbula, a nivel del pecho.



Punch: Golpe de puño al frente.



PunchUp: Golpe de puño que viene desde el torso hasta arriba de la cabeza.



DHead(Defense Head): Defensa de ambos puños en la zona de la cara y cabeza.



ANÁLISIS Y RESULTADOS

En la clasificación de movimiento, se trabajó con 15600 datos con los tres canales (x,y,z), por el cual se utilizaron 104 ventanas. Se etiquetaron y se realizaron los modelos de una red multicapa profunda, otro en convolución 1D y el otro en redes recurrentes.

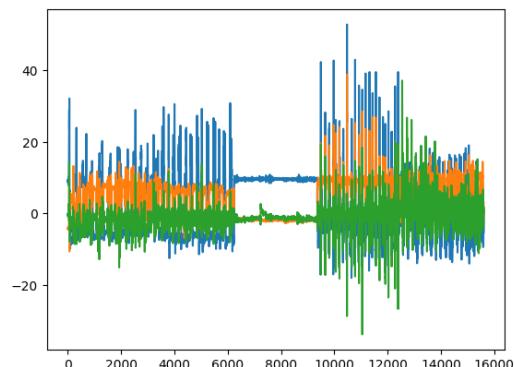


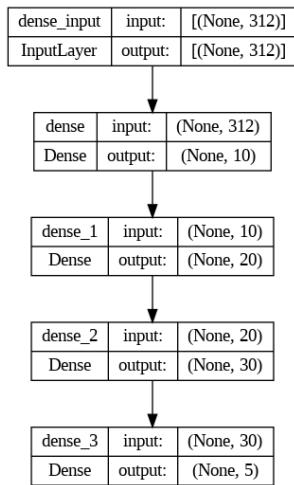
Fig visualización de los datos

Primero, realizando MLP se dio el siguiente modelo y arquitectura:

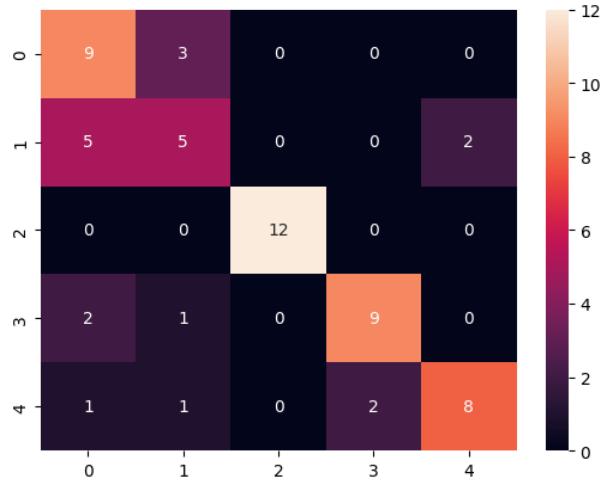
```
#Definición del modelo
modelo = Sequential()
modelo.add(Dense(10, input_shape=(312,), activation='relu'))
modelo.add(Dense(20, activation='relu'))
modelo.add(Dense(30, activation='relu'))
modelo.add(Dense(5, activation = 'softmax'))

modelo.summary()

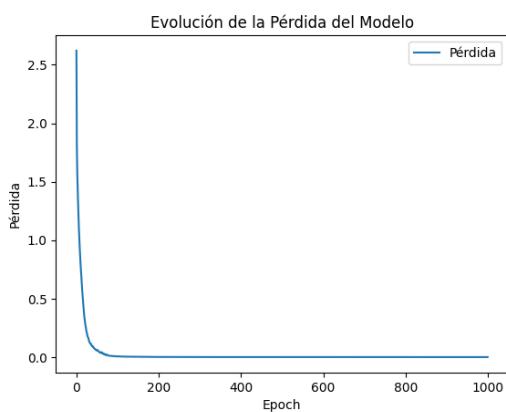
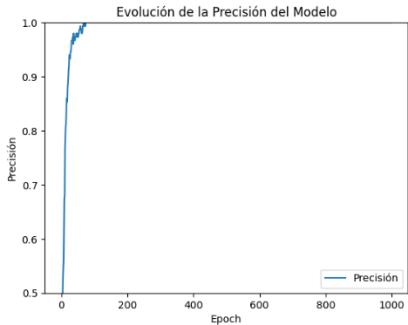
keras.utils.plot_model(modelo, to_file='model_plot3.png', show_shapes=True, show_layer_names=True)
```



	precision	recall	f1-score	support
0	0.53	0.75	0.62	12
1	0.50	0.42	0.45	12
2	1.00	1.00	1.00	12
3	0.82	0.75	0.78	12
4	0.80	0.67	0.73	12
accuracy			0.72	60
macro avg	0.73	0.72	0.72	60
weighted avg	0.73	0.72	0.72	60



En donde se entrenó con 1000 épocas, dando el siguiente resultado:



Su reporte y matriz de confusión , en donde sus etiquetas eran Dhead (0),Defense(1) , Normal(2), Punch(3) y PunchUp(4):

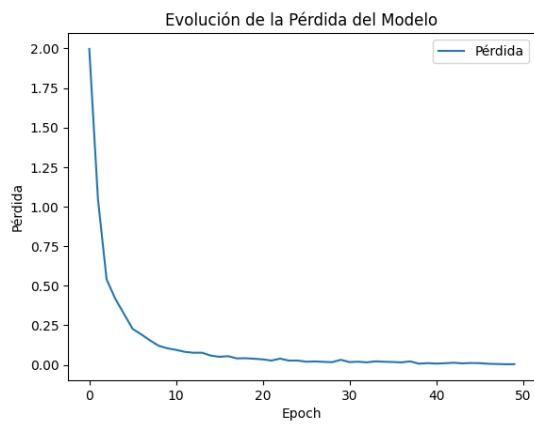
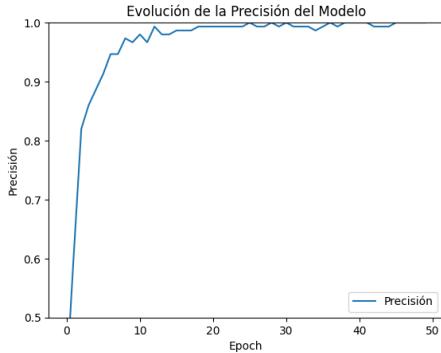
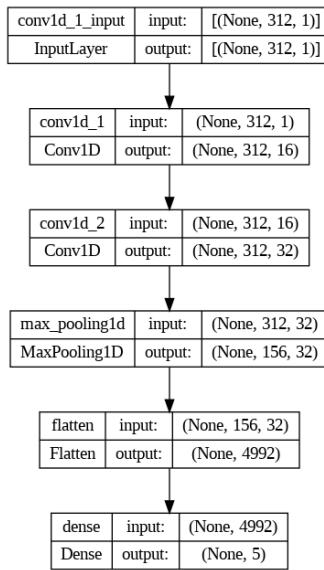
Se puede observar que el accuracy es de buena calidad, ya que tiene un 72% , y en la matriz de confusión acierta en la mayoría de los casos, y evita equivocarse con una pequeña cantidad.

Realizando Convolución 1D, es el mismo proceso pero cambiando el modelo:

```
#Definición del modelo
modelo = Sequential()
modelo.add(Conv1D(16, 3, activation="relu", padding="same", input_shape=(312,1)))
modelo.add(Conv1D(32, 3, activation="relu", padding="same"))
modelo.add(MaxPooling1D(pool_size=2, strides=2, padding='same'))
modelo.add(Flatten())
modelo.add(Dense(5, activation='softmax'))

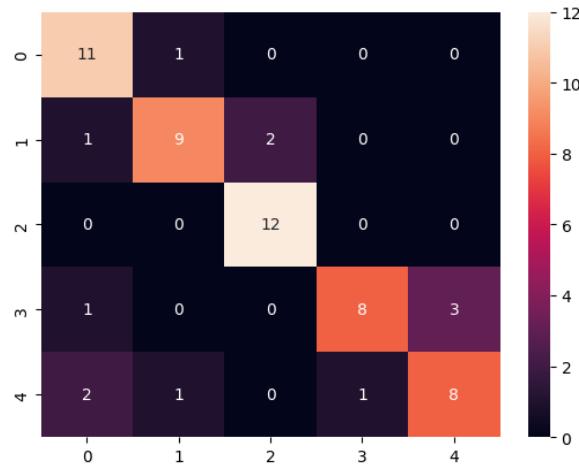
modelo.summary()

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



Su reporte y matriz de confusión, en donde sus etiquetas eran Dhead (0), Defense(1), Normal(2), Punch(3) y PunchUp(4):

	precision	recall	f1-score	support
0	0.73	0.92	0.81	12
1	0.82	0.75	0.78	12
2	0.86	1.00	0.92	12
3	0.89	0.67	0.76	12
4	0.73	0.67	0.70	12
accuracy			0.80	60
macro avg	0.80	0.80	0.80	60
weighted avg	0.80	0.80	0.80	60



Se considera este modelo de Convolución 1D mejor que el MLP, debido a que este tiene un 80% de accuracy y su matriz evidencia la capacidad de acertar con mucha eficacia y una cantidad mínima de errores.

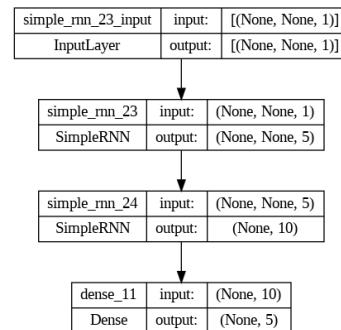
Realizando Redes Recurrentes, es el mismo proceso, pero cambiando el modelo:

```

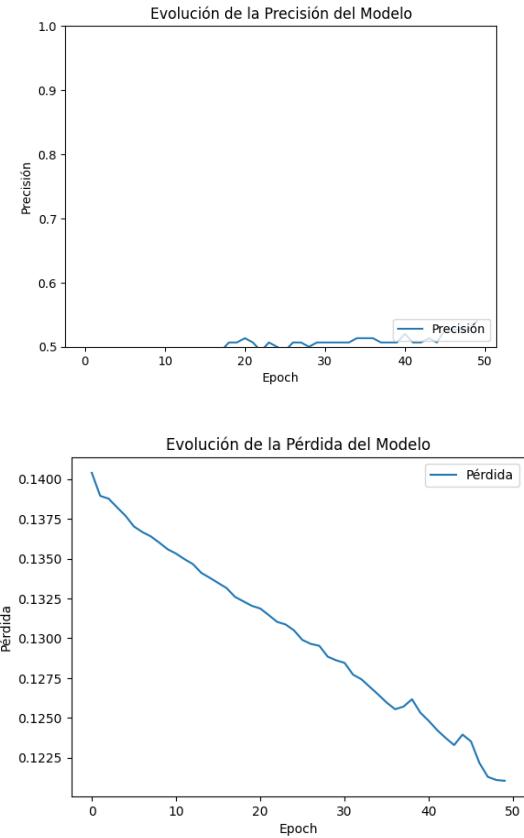
#Definición del modelo
modelo = Sequential()
modelo.add(SimpleRNN(5,return_sequences=True,input_shape=[None ,1])) # tres canales
modelo.add(SimpleRNN(10))
modelo.add(Dense(5))#clases
modelo.summary()

keras.utils.plot_model(modelo, to_file='model_plotRecurrents.png', show_shapes=True, show_layer_names=True)

```



Se entrena con 50 épocas y usando como métrica MSE:



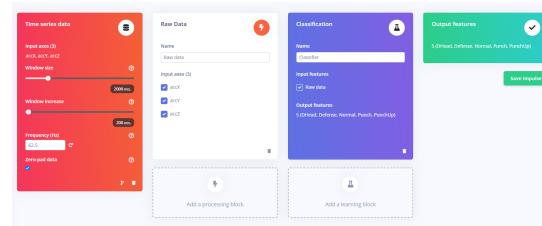
Su reporte y matriz de confusión , en donde sus etiquetas eran Dhead (0),Defense(1) , Normal(2) ,Punch(3) y PunchUp(4):

	precision	recall	f1-score	support
0	0.42	0.67	0.52	12
1	0.30	0.25	0.27	12
2	0.92	1.00	0.96	12
3	0.44	0.33	0.38	12
4	0.33	0.25	0.29	12
accuracy			0.50	60
macro avg	0.48	0.50	0.48	60
weighted avg	0.48	0.50	0.48	60

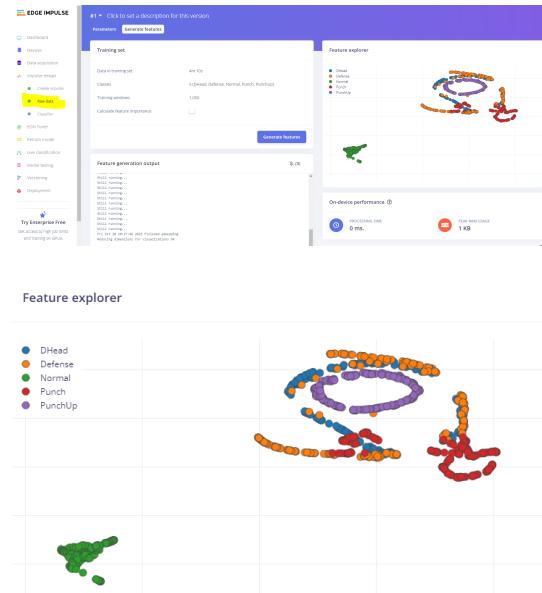


Este modelo es considerado fuerte en su entrenamiento, pero en este caso dio peor en comparación con los dos modelos anteriores, debido a que su accuracy es de 50% y en su matriz de confusión se ve reflejado con la equivocación de varias veces en su clasificación de cada movimiento.

Usando solamente Edge Impulse, se decidió tomar la parte de MLP, para eso se usó este esquema:



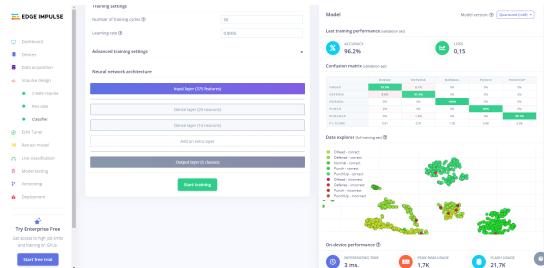
En la parte de Raw data se seleccionó todo por defecto y se guardaron los cambios, obteniendo:



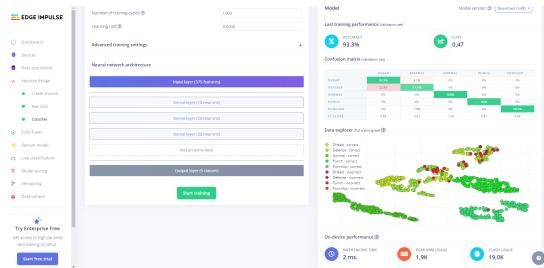
Por lo que, al visualizar las clases, algunas tenían un poco las zonas muy cerca, pero no se combinaban y se podían diferenciar con certeza.

Tomando el modelo por defecto que tiene Edge Impulse de 20 neuronas densas y 10 neuronas densas, se cambió a 50 épocas, por

lo que el resultado fue positivo, con accuracy de 96.2% y loss del 0.15. Comparando los tres modelos realizados con TensorFlow-Keras, este modelo hecho en Edge Impulse resulta mejor. (se puede ver en: <https://studio.edgeimpulse.com/public/296436/latest>)



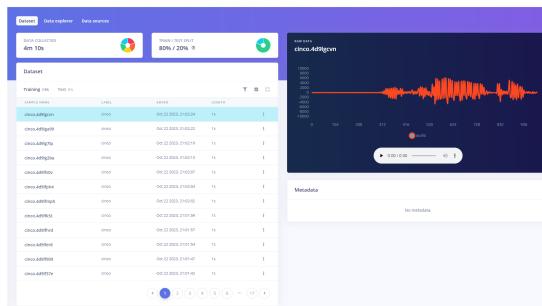
Pero para tener una comparación más eficiente, se replicó el modelo de keras MLP hacia el de Edge impulse:



El modelo arrojó un resultado evidentemente mejor que los tres modelos anteriores de TensorFlow-Keras, ya que tiene un 93.3% de accuracy y 0.47 de loss, pero un poco peor comparándolo con el más simple como el anterior de Edge impulse por defecto. Hay que tener en cuenta que el de solo 20 y 10 neuronas, solamente utilizó 50 épocas y que el de la réplica utilizó 1000 épocas.

Para el tercer punto, en primera medida, se determina la necesidad de crear un dataset adecuado a este ejercicio. Se opta por el uso de la herramienta de Edge impulse para grabar, a través de la opción de “Data acquisition”, cinco distintas categorías, siendo “uno”, “dos”, “tres”, “cuatro” y “cinco” respectivamente. El objetivo de la

aplicación es ofrecer una solución más general para clasificar cinco distintas categorías, en este caso, enfocadas en los números, siendo posible implementarlo a un software para comandar una condición específica, como puede ser “abrir la puerta uno”, “encender la luz dos”, entre otras. El hardware utilizado es el computador para implementar los desarrollos de los modelos en línea, y el dispositivo móvil para la recolección de datos y testeo de modelos. A continuación se visualiza una la herramienta utilizada:



Dando como resultado una carpeta llamada “training” con 199 muestras, y otra carpeta llamada “testing” con 51 muestras disponibles. A continuación, se comienza con el código.

```
▶ from google.colab import drive
drive.mount('/content/gdrive')
↳ Mounted at /content/gdrive

[ ] # Se cargan las diferentes dependencias necesarias
import tensorflow as tf
from tensorflow import keras
import pandas as pd
import numpy as np
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
import matplotlib as mpl

from IPython import display
```

Se procede a realizar la implementación de un clasificador de audio mediante el método de Convolución 2D, a través de la plataforma de TensorFlow-Keras en Google Colab. Como primera medida, se hace la conexión de

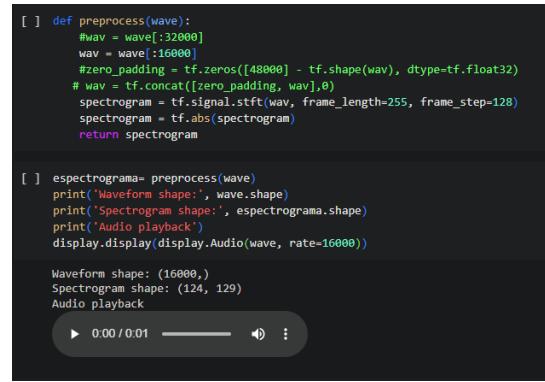
Google Drive para el posterior uso del dataset guardado en esta herramienta, y se realiza la importación de las librerías necesarias para este proyecto.



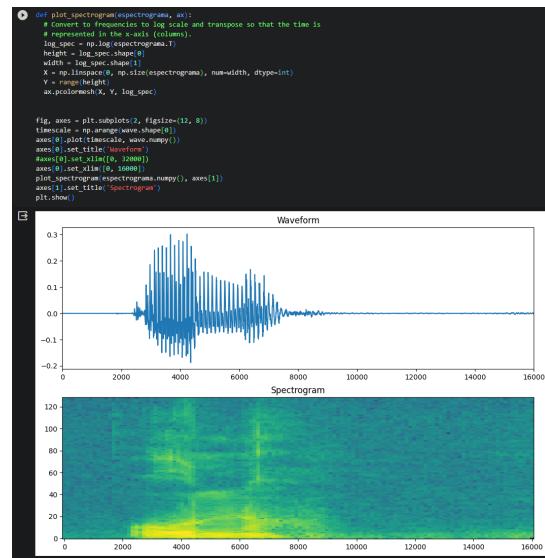
La función `load_wav_16k_mono` ha sido diseñada para simplificar y estandarizar la carga y el procesamiento de archivos de audio en formato WAV. Su propósito principal es adaptar archivos de audio a un formato más manejable y uniforme para su posterior análisis y procesamiento. Al cargar un archivo de audio, la función transforma el audio a un solo canal (mono) y ajusta su frecuencia de muestreo a 16kHz. Estos ajustes son cruciales para garantizar la consistencia en la calidad y la representación del audio, lo cual es esencial al trabajar con redes neuronales y otros algoritmos de aprendizaje automático.

Como ejemplo práctico del uso de esta función, se muestra un segmento donde se carga un archivo de audio específico desde una ruta en Google Drive. Una vez cargado y procesado, se visualiza la forma de onda del audio. Esta visualización es una herramienta esencial para comprender las características temporales del sonido, permitiendo a los investigadores y desarrolladores identificar

patrones, anomalías o características específicas en los datos de audio.



La función `preprocess` transforma una forma de onda de audio en un espectrograma, una representación gráfica que muestra cómo varía la amplitud de una señal con el tiempo y la frecuencia. Utilizando la transformada de corto plazo de Fourier (STFT), la señal se descompone en sus componentes de frecuencia, permitiendo visualizar la energía en diferentes bandas de frecuencia a lo largo del tiempo. Para ilustrar su uso, se procesa una forma de onda, mostrando luego sus dimensiones y ofreciendo una reproducción auditiva de la señal procesada.



La función presentada, plot_spectrogram, tiene el objetivo de visualizar tanto una forma de onda de audio como su respectivo espectrograma. La forma de onda muestra las fluctuaciones de amplitud de la señal a lo largo del tiempo, mientras que el espectrograma representa la intensidad de las frecuencias de la señal en función del tiempo. Para el espectrograma, se realiza una transformación a una escala logarítmica, permitiendo una mejor distinción de las frecuencias bajas. Ambas visualizaciones se generan en subtramas separadas, con la forma de onda en la parte superior y el espectrograma en la inferior. Las gráficas facilitan una comprensión más clara de cómo una señal de audio se distribuye tanto en el tiempo como en las frecuencias, por lo que se realiza un plot con el Waveform del audio y su respectivo espectrograma.

```
[ ] # Se cargan los diferentes archivos *.json que se van a usar en el proceso
# de entrenamiento
import os
directory="/content/gdrive/MyDrive/numbers_dataset/training/"
files = os.listdir(directory)
files.sort()
cantidadFiles = len(files)
# Variable donde se almacenaron los datos leidos de los archivos *.json
#Datos=np.zeros((cantidadFiles*624,3))
Datos=np.zeros((cantidadFiles,124,129))
i = 0
for file in files:
    Rutafile=directory + file
    wave = load_wav_16k_mono(Rutafile)
    espectrograma= preprocess(wave)
    Datos[i,:,:]=espectrograma
    i+=1
print(Datos.shape)
Xtrain=tf.expand_dims(Datos, axis=3)
print(Xtrain.shape)

(199, 124, 129)
(199, 124, 129, 1)
```

El código proporcionado está diseñado para cargar y procesar archivos de audio desde un directorio específico con el propósito de entrenamiento. Utilizando funciones del sistema operativo, se enumeran y ordenan los archivos, luego se inicializa una matriz basada en la cantidad de archivos y el tamaño del espectrograma. Cada archivo se procesa para obtener un espectrograma, que se almacena en la matriz. Tras procesar todos los

archivos, la matriz se reestructura para ser compatible con modelos de aprendizaje profundo, y finalmente se imprimen las dimensiones de las matrices procesadas para verificación, dando como resultado (199, 124, 129, 1).

```
[ ] #cinco: 39
#cuatro: 43 = 82
#dos: 37 = 119
#tres: 39 = 158
#uno: 41 = 199
#Total: 199

YtrainIni=np.zeros((199,1))
for i in range(39):
    YtrainIni[i]=4

for i in range(39,82):
    YtrainIni[i]=3

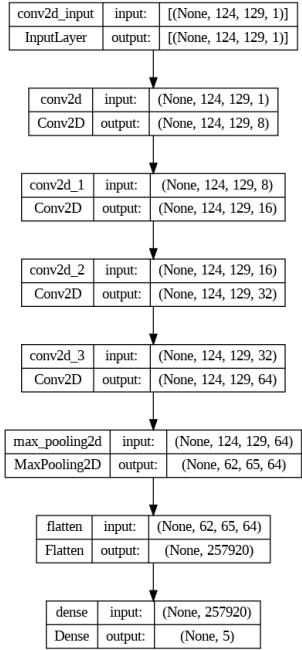
for i in range(82,119):
    YtrainIni[i]=1

for i in range(119,158):
    YtrainIni[i]=2

for i in range(158,199):
    YtrainIni[i]=0

print(YtrainIni)
```

En este código, se hace la definición de YtrainIni haciendo una separación manual de los audios. Para esto, se revisó la carpeta original donde se ubican los audios y se separó la variable en distintos rangos definidos de manera coherente con la cantidad de audios por categoría. Posteriormente a eso, se realiza el código Ytrain= keras.utils.to_categorical(YtrainIni) con el objetivo de darle formato para la clasificación de las distintas categorías.



Se realiza la definición de la arquitectura de una red neuronal convolucional (CNN) para el procesamiento de los datos. Comienza con una capa de entrada que acepta datos en la forma (None, 124, 129, 1). Luego, hay cuatro capas convolucionales consecutivas (Conv2D) que aumentan gradualmente en profundidad desde 8 a 64 canales. Tras estas capas, se introduce una capa de max pooling (MaxPooling2D) que reduce las dimensiones espaciales a la mitad. Después de esta reducción, los datos se aplanan (Flatten) en un vector largo, y finalmente, pasan a través de una capa densa (Dense) que produce 5 valores de salida, representando las cinco categorías de este proyecto.

Posteriormente, se realiza el entrenamiento con el siguiente código

```

[ ] ➜ modelo.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
history = modelo.fit(Xtrain,Ytrain, epochs=30, batch_size=None)

Epoch 2/30
7/7 [=====] - 17s 2s/step - loss: 0.6686 - accuracy: 0.8643
Epoch 3/30
7/7 [=====] - 16s 2s/step - loss: 0.2015 - accuracy: 0.9598
Epoch 4/30
7/7 [=====] - 17s 2s/step - loss: 0.0365 - accuracy: 0.9899
Epoch 5/30
7/7 [=====] - 16s 2s/step - loss: 0.0117 - accuracy: 0.9950
Epoch 6/30
7/7 [=====] - 16s 2s/step - loss: 0.0024 - accuracy: 1.0000
Epoch 7/30
7/7 [=====] - 16s 2s/step - loss: 0.0412 - accuracy: 0.9899

```

Obteniendo como resultado los siguientes valores graficados:



Después de esto, se repite el proceso de obtención de datos (en este caso de testeo), procesamiento usando el proceso de Convolución 2D y definición de la variable XVal y YVal para la evaluación. Se realiza dicha evaluación, obteniendo como resultado lo siguiente:

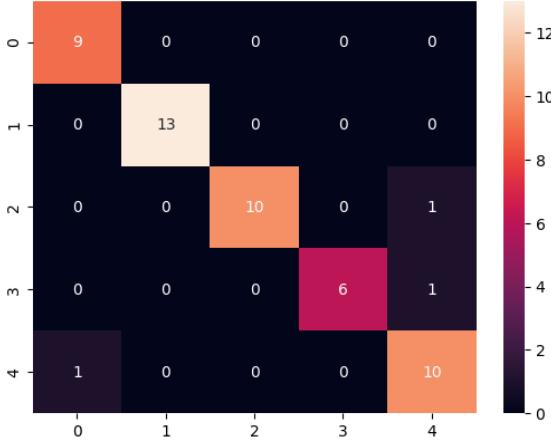
```

[ ] ➜ modelo.evaluate(XVal, YVal)

2/2 [=====] - 1s 377ms/step - loss: 2.4139 - accuracy: 0.9412
[2.413900375366211, 0.9411764740943909]

```

Finalmente, se grafica la matriz de confusión para una mejor comprensión de los resultados, verificando que se obtuvo una precisión excelente para cada categoría,



```
[1]: # Configuraciones iniciales
directory = "/content/gdrive/MyDrive/numbers_dataset/training/"
files = os.listdir(directory)
files.sort()
cantidadFiles = len(files)

# Tamaño de las características MFCC + delta + delta2
feature_size = 40 + 40 + 40 # MFCC + Delta + Delta2
Datos = np.zeros((cantidadFiles, feature_size))

for i, file in enumerate(files):
    # Cargar archivo
    file_name = os.path.join(directory, file)
    x, Fs = librosa.load(file_name, mono=True, sr=16000)

    # Calcular MFCC
    mfccs = librosa.feature.mfcc(y=np.float32(x), sr=Fs, n_fft=512, hop_length=256, n_mfcc=40)
    mfccs_sc = np.mean(mfccs.T, axis=0)

    # Calcular derivadas primera y segunda
    mfcc_delta = librosa.feature.delta(mfccs_sc)
    mfcc_delta2 = librosa.feature.delta(mfccs_sc, order=2)

    # Consolidar características
    Datos[i] = np.hstack((mfccs_sc, mfcc_delta, mfcc_delta2))

Xtrain = Datos
print(Xtrain.shape)
(199, 128)
```

Para el siguiente modelo basado en redes neuronales profundas, se define una estructura basada en MFCC en la plataforma TensorFlow-Keras manejada en Google Colab, con el objetivo de poder comparar ambos resultados.

```
[ ] from google.colab import drive
drive.mount('/content/gdrive')
Mounted at /content/gdrive

[ ] # Se cargan las diferentes dependencias necesarias
import tensorflow as tf
from tensorflow import keras
import pandas as pd
import numpy as np
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
import matplotlib as mpl
import librosa
import os

from IPython import display
```

En primera medida, se realiza la conexión a Google Drive y las importaciones pertinentes para este proyecto.

Este código realiza la extracción de características de audio a partir de archivos almacenados en el directorio definido para almacenar el dataset previamente existente. Inicialmente, se obtiene una lista de todos los archivos presentes en el directorio, que luego es ordenada alfabéticamente. Se define el tamaño total de las características a extraer, que consiste en coeficientes MFCC y sus primeras y segundas derivadas. Para cada archivo, se carga el contenido de audio y se calculan los coeficientes MFCC. Posteriormente, se determinan las primeras y segundas derivadas de estos coeficientes. Estas características se consolidan en un único vector, que es almacenado en una matriz. Al final del proceso, se tiene una matriz Xtrain que contiene las características de todos los archivos de audio, y se imprime su forma para verificar la correcta consolidación de datos.

```
[ ] #cinco: 39
#cuatro: 43 = 82
#dos: 37 = 119
#tres: 39 = 158
#uno: 41 = 199
#Total: 199

YtrainIni=np.zeros((199,1))
for i in range(39):
    YtrainIni[i]=4

for i in range(39,82):
    YtrainIni[i]=3

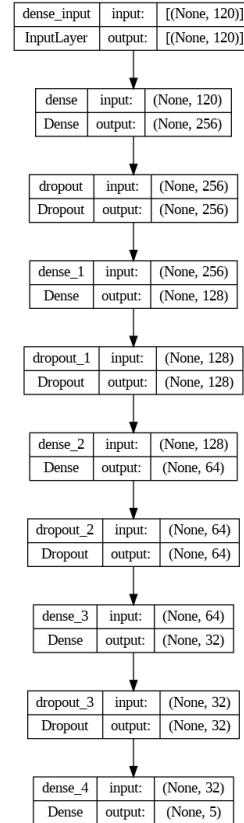
for i in range(82,119):
    YtrainIni[i]=1

for i in range(119,158):
    YtrainIni[i]=2

for i in range(158,199):
    YtrainIni[i]=0

print(YtrainIni)
```

Al igual que en el proyecto anterior, se hace la definición de YtrainIni haciendo una separación manual de los audios. Para esto, se revisó la carpeta original donde se ubican los audios y se separó la variable en distintos rangos definidos de manera coherente con la cantidad de audios por categoría. Posteriormente a eso, se realiza el código Ytrain= keras.utils.to_categorical(YtrainIni) con el objetivo de darle formato para la clasificación de las distintas categorías.



Se define el modelo a utilizar. Se ilustra la construcción de un modelo de red neuronal artificial utilizando la biblioteca TensorFlow Keras. El modelo, denominado model_MFCC, es de tipo secuencial y se compone de cuatro capas ocultas y una capa de salida. La primera capa oculta tiene 256 neuronas con una función de activación ReLU y regularización L2. La segunda, tercera y cuarta capas ocultas cuentan con 128, 64 y 32 neuronas respectivamente, todas con activación ReLU y regularización L2. Cada una de estas capas está seguida por una capa de "Dropout" con una tasa del 20% para prevenir el sobreajuste. La capa de salida tiene 5 neuronas, que corresponden a las cinco categorías a predecir, y utiliza una función de activación softmax. Finalmente, se muestra un resumen del modelo a través de la imagen anterior.

```
[ ] # Compilación del modelo
model_MFCC.compile(optimizer='Adam', loss='categorical_crossentropy', metrics=['accuracy'])

[ ] from tensorflow.keras.callbacks import EarlyStopping
# Definición del callback EarlyStopping
early_stopping = EarlyStopping(monitor='val_loss', patience=100, verbose=1, restore_best_weights=True)

# Entrenamiento del modelo
history = model_MFCC.fit(Xtrain, Ytrain, epochs=100, batch_size=32, validation_split=0.2, callbacks=[early_stopping], verbose=1)

# Al final de cada validación imprime
loss, acc = history.history['val_loss'][1:-1], history.history['val_accuracy'][1:-1]
print('loss: {} - Accuracy: {}'.format(loss, acc))
```

El fragmento de código muestra la compilación y entrenamiento del modelo. Este se compila utilizando el optimizador 'Adam', una función de pérdida de entropía cruzada categórica y se rastrea su precisión durante el entrenamiento. Se introduce un mecanismo de "detención temprana" que monitorea la pérdida en los datos de validación, deteniendo el entrenamiento si no hay mejoras tras 100 épocas y restaurando los mejores pesos obtenidos. El modelo se entrena con un conjunto de datos, Xtrain y Ytrain, a lo largo de 100 épocas, usando un tamaño de lote de 32 y reservando el 20% de los datos para validación. Al final, se imprime la última pérdida y precisión registrada en el proceso de validación.

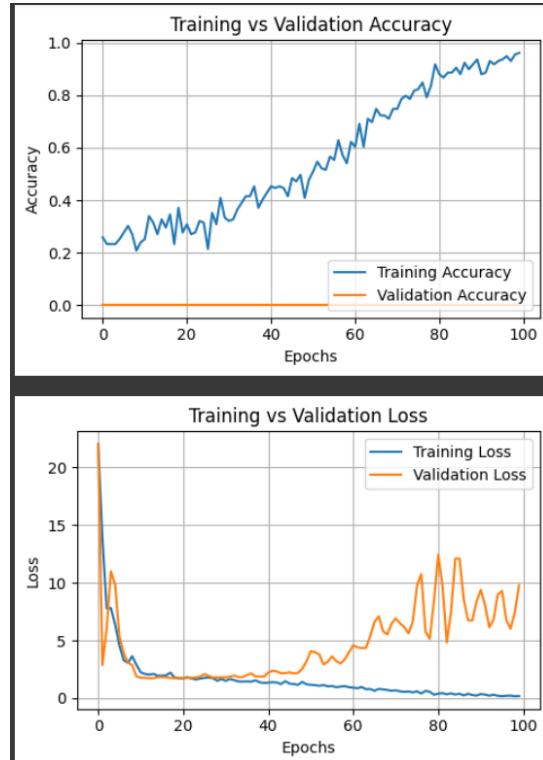
```
Epoch 95/100
S/5 [=====] - 0s 15ms/step - loss: 0.2095 - accuracy: 0.9308 - val_loss: 0.9986 - val_accuracy: 0.0000e+00
Epoch 96/100
S/5 [=====] - 0s 15ms/step - loss: 0.1752 - accuracy: 0.9371 - val_loss: 0.2946 - val_accuracy: 0.0000e+00
Epoch 97/100
S/5 [=====] - 0s 17ms/step - loss: 0.2085 - accuracy: 0.9459 - val_loss: 0.7856 - val_accuracy: 0.0000e+00
S/5 [=====] - 0s 16ms/step - loss: 0.2215 - accuracy: 0.9398 - val_loss: 0.0288 - val_accuracy: 0.0000e+00
Epoch 98/100
S/5 [=====] - 0s 16ms/step - loss: 0.1689 - accuracy: 0.9568 - val_loss: 0.4363 - val_accuracy: 0.0000e+00
Epoch 99/100
S/5 [=====] - 0s 16ms/step - loss: 0.1689 - accuracy: 0.9568 - val_loss: 0.4363 - val_accuracy: 0.0000e+00
Epoch 100/100
S/5 [=====] - 0s 10ms/step - loss: 0.1868 - accuracy: 0.9623 - val_loss: 0.7776 - val_accuracy: 0.0000e+00
loss: 0.777612686357227 - Accuracy: 0.0
```

La imagen muestra la salida de las últimas 6 épocas (de 95 a 100) de un proceso de entrenamiento de un modelo de machine learning. Durante estas épocas, el modelo parece tener un rendimiento bastante bueno en el conjunto de entrenamiento, con una precisión superior al 93%. Sin embargo, hay un problema evidente en la validación: la función de pérdida en los datos de validación (val_loss) es muy alta y, lo más preocupante, la precisión en los datos de validación (val_accuracy) es de 0.0 en todas las épocas mostradas. Al final del entrenamiento, se presenta la pérdida y precisión finales en la validación, confirmando una precisión de 0.0.

Esto sugiere un grave problema de sobreajuste, donde el modelo se desempeña bien en los datos de entrenamiento, pero falla por completo en los datos de validación.

Después de esto, se repite el proceso de obtención de datos (en este caso de testeo), procesamiento usando MFCC y definición de la variable XVal y YVal para la evaluación. Se realiza dicha evaluación, obteniendo como resultado lo siguiente:

```
model_MFCC.evaluate(XVal, YVal)
2/2 [=====] - 0s 8ms/step - loss: 1.8117 - accuracy: 0.8039
[1.811654806137085, 0.8039215803146362]
```

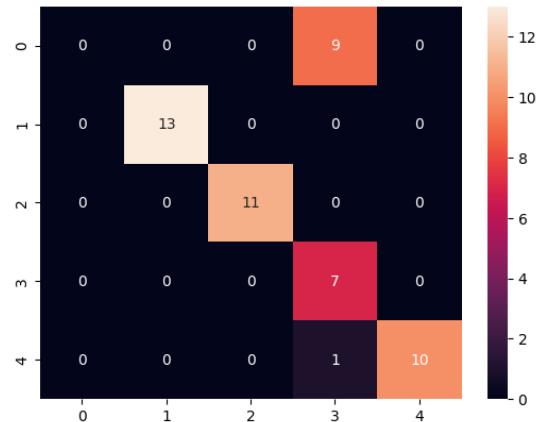


Las gráficas presentadas muestran la evolución de la precisión y la pérdida del modelo durante el entrenamiento y la

validación a lo largo de 100 épocas. En la gráfica superior, vemos que la precisión de entrenamiento aumenta constantemente, acercándose a 1.0 hacia el final de las épocas, lo que indica un buen rendimiento en el conjunto de entrenamiento. Sin embargo, la precisión de validación permanece prácticamente en 0 a lo largo de todas las épocas, lo que sugiere que el modelo no está generalizando bien y presenta un grave problema de sobreajuste, lo cual se entiende al recordar que el dataset consiste en la voz de una sola persona.

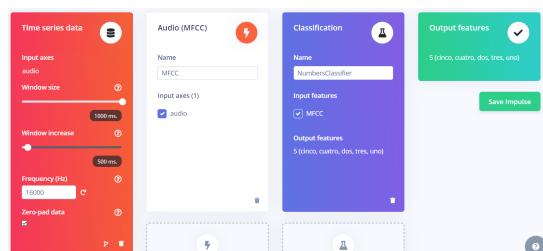
La gráfica inferior muestra las pérdidas de entrenamiento y validación a lo largo del tiempo. La pérdida de entrenamiento disminuye continuamente, lo que es consistente con la creciente precisión de entrenamiento. Por otro lado, la pérdida de validación experimenta picos significativos y no muestra una tendencia clara de disminución. Aunque hay momentos en que la pérdida de validación disminuye, en general, permanece alta, lo que refuerza la idea de que el modelo no está desempeñándose bien en el conjunto de validación.

En conjunto, estas gráficas resaltan un claro caso de sobreajuste, donde el modelo se adapta muy bien a los datos de entrenamiento pero falla en generalizar a datos nuevos o no vistos, como es el conjunto de validación. Es recomendable revisar el modelo, posiblemente introducir regularización, y asegurarse de que los datos de entrenamiento y validación estén adecuadamente preparados y distribuidos.

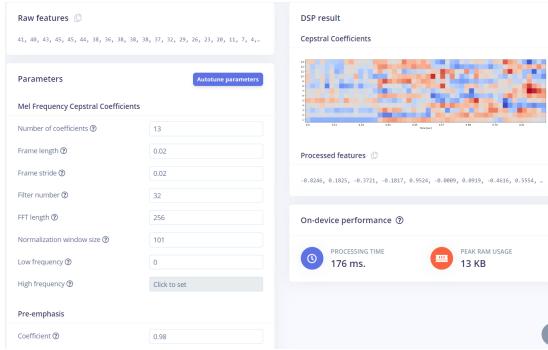


Finalmente, la matriz de confusión demuestra un desempeño positivo para todas las categorías, exceptuando la inicial, donde las predicciones parecen ser tomadas como la categoría tres, en vez de la correcta que es la cero.

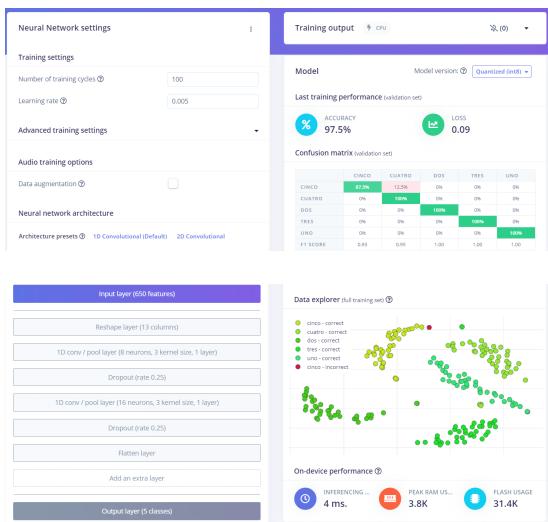
Pasando ahora al siguiente ejercicio, se replica el entrenamiento del modelo de MFCC en Edge Impulse, con el objetivo de obtener mejores resultados al visto anteriormente. Se comparte aquí el acceso al proyecto:
<https://studio.edgeimpulse.com/public/297130/latest>. A continuación se visualizan los esquemas utilizados:



En los parámetros de MFCC, se define la siguiente estructura:



Aquí se puede notar una configuración común, predefinida, en el área de los MFCC, donde el número de coeficientes suele ser 13. Los demás parámetros también se dejaron predefinidos, realizando así el entrenamiento con los siguientes ajustes y resultados:



En la arquitectura de red neuronal mostrada, se inicia con una capa de entrada que acepta 650 características. Esta entrada se redimensiona en una capa de remodelación con 13 columnas. Luego, sigue una serie de capas convolucionales unidimensionales (1D) intercaladas con capas de abandono (dropout) para evitar el sobreajuste. La primera capa convolucional 1D utiliza 8 neuronas y un tamaño de núcleo de 3, seguida por un

abandono del 0.25. La segunda capa convolucional 1D se configura con 16 neuronas y un tamaño de núcleo similar, seguida nuevamente por un abandono del 0.25. Tras estas capas, la información se aplana en una capa de aplanamiento, con una opción para añadir una capa adicional si se requiere.

Finalmente, se tiene una capa de salida que categoriza las entradas en 5 clases diferentes. A la derecha, se visualiza un explorador de datos que muestra el desempeño del modelo en el conjunto de entrenamiento completo, distinguiendo entre predicciones correctas e incorrectas para cinco categorías: 'cinco', 'cuatro', 'dos', 'tres', y 'uno'. Las métricas de rendimiento en el dispositivo indican un tiempo de inferencia extremadamente rápido de 4 ms, con un uso de RAM pico de 3.8K y un consumo de memoria flash de 31.4K.

El modelo en cuestión dio resultados más positivos que el realizado en Google Colab a través de TensorFlow-Keras. Al realizar las pruebas en tiempo real, su precisión es excepcionalmente alta, logrando tener un buen desempeño. Se verifica mediante el celular esta afirmación, pudiéndose ver en el video relacionado a este punto en cuestión.

En conclusión a este ejercicio, resulta importante resaltar que, el desarrollo de los tres modelos, tanto con TensorFlow-Keras como con la plataforma de Edge Impulse, suponen un ejercicio enriquecedor para entender las dinámicas explícitas en la recolección y organización de datos, preprocesamiento, desarrollo de la arquitectura, entrenamiento y testeo de estos.

La realización de este ejercicio permitió comparar la usabilidad de un entorno más enfocado al desarrollador como lo es Google Colab con sus distintas librerías, y para un entorno enfocado en usuarios con menor experiencia en esta área, permitiendo mayor campo para estos, como es el caso de Edge Impulse. El manejo de datos y entendimiento

de conceptos es fundamental para realizar un trabajo exitoso.

BIBLIOGRAFÍA

- [1] G. Monte, “Representación simplificada de señales unidimensionales y bidimensionales para la extracción de información real,” Tesis de doctorado, Universidad Nacional de Mar del Plata, 2022. [En línea]. Disponible: <http://rinfofi.fimdp.edu.ar/bitstream/handle/123456789/653/GMonte-TD-Ee-2022.pdf?sequence=1>.
- [2] I. Bosch Roig, J. Gosálbez Castillo, R. Miralles Ricós, y L. Vergara Domínguez, “Señales y Sistemas: teoría y problemas,” Universitat Politècnica de València, 2022. [En línea]. Disponible: https://gdocu.upv.es/alfresco/service/api/node/content/workspace/SpacesStore/5a1651eb-f46e-4752-995a-125a77bf045e/TOC_0377_04_01.pdf?guest=true.
- [3] Wikipedia, “Redes neuronales recurrentes,” 2023. [En línea]. Disponible: https://es.wikipedia.org/wiki/Redes_neuronales_recurrentes.
- [4] IBM, “¿Qué son las redes neuronales recurrentes?,” 2023. [En línea]. Disponible: <https://www.ibm.com/es-es/topics/recurrent-neural-networks>.