



**Tecnológico
de Monterrey**

Instituto Tecnológico y de Estudios Superiores de Monterrey

Campus Querétaro

TC3007C Inteligencia artificial avanzada para la ciencia de datos II

Módulo 2 Análisis y Reporte sobre el desempeño del modelo.

Profesores:

Benjamín Valdés Aguirre

Carlos Alberto Dorantes Dosamantes

José Antonio Cantoral Ceballos

Ismael Solis Moreno

Eduardo Daniel Juárez Pineda

Presenta:

José Emiliano Riosmena Castañón – A01704245

Fecha:

Lunes, 11 de noviembre del 2024

Índice

Introducción.....	3
Descripción del Dataset.....	4
Uso de ETL (Extract – Transform – Load)	6
Implementación de Red Neuronal.....	8
Construcción del modelo CNN	8
Entrenamiento del modelo	10
Análisis de Resultados	12
Resultados R^2	12
Pérdida.....	13
Desempeño en Testing	13
Predicciones del modelo	15
Conclusiones.....	17
Referencias	18

Introducción

En este ensayo, hablaremos sobre el proceso seguido para construir un modelo CNN para identificar si un acorde es mayor o menor. Por eso, en este análisis se presentará una discusión sobre el conjunto de datos utilizado, el uso de las técnicas y algoritmos vistos en clase para el desarrollo del modelo, y los desafíos presentados durante la implementación de este último.

Por último, veremos los resultados del desempeño de nuestro modelo, analizando su nivel de precisión, su capacidad de identificar la tónica en diferentes acordes en el conjunto de prueba, y en datos ajenos al dataset. De este modo nos permitirá concluir si el modelo es efectivo para cumplir su objetivo.



Descripción del Dataset

Para entender cómo funcionara el modelo, vamos a definir lo que es un acorde, y entender lo que significa un acorde mayor o menor. En una guitarra, un acorde es un conjunto de tres o más notas que se tocan al mismo tiempo. Esto genera uno de los principales componentes de la música, la armonía. En la guitarra, esto se produce apretando ciertas cuerdas en ciertos trastes, y tocando varias cuerdas al mismo tiempo, y produce un sonido más completo que una sola nota. En la música, los acordes se clasifican según su calidad, esta describe las notas específicas que lo componen y cómo se relacionan entre sí. Cada calidad transmite una sensación distintiva, lo que permite expresar diferentes emociones o “colores” musicales.

En el caso específico de este proyecto, estamos identificando únicamente mayores y menores. La diferencia entre un acorde mayor y un acorde menor está en la tercera nota que lo compone, conocida como la tercera del acorde.

- **Acorde Mayor:** Tiene una tercera mayor, es decir, 4 semitonos arriba de la raíz. Su sonido se percibe como alegre, brillante o abierto. Y se escriben de la misma forma que su raíz. Por ejemplo, el acorde de Do mayor se escribiría como Do, o en su notación en inglés como C.
- **Acorde Menor:** Tiene una tercera menor, es decir, 3 semitonos arriba de la raíz. Su sonido se percibe como melancólico, triste u oscuro. Se escriben añadiendo la letra “m” minúscula al lado de la raíz. Por ejemplo, el acorde de Do menor se escribiría como Dom, o en su notación en inglés como Cm.

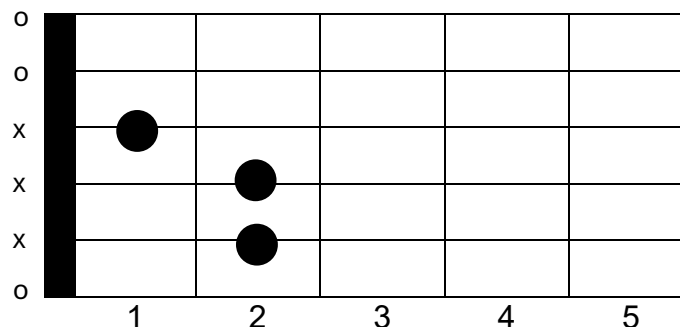


Figura 1.0. Ejemplo de acorde mayor. Mi o E.

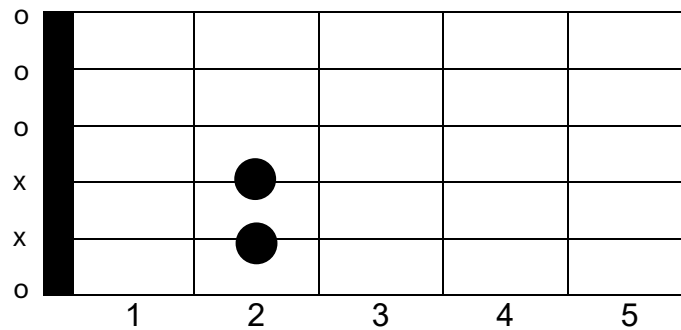


Figura 1.1. Ejemplo de acorde menor. Mim o Em.

Entendiendo esto, vamos a ver los datos que hemos utilizado para este modelo. El dataset es un conjunto de archivos de audio, los cuales contienen una mezcla de diferentes acordes reproducidos ya sea en guitarra acústica o eléctrica. Los archivos vienen clasificados en dos subcarpetas: mayor y menor, las cuales representan a los acordes mayores y menores.

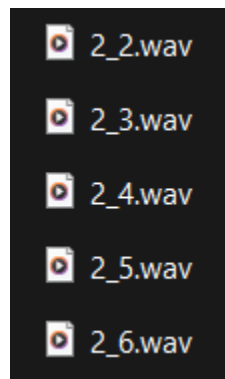


Figura 1.2. Muestra del dataset utilizado.

Como hemos mencionado al principio de este documento, el objetivo de este proyecto es generar un modelo que sea capaz de identificar si un acorde dado es mayor o menor.

Uso de ETL (Extract – Transform – Load)

Para la extracción de los datos, en este dataset particular, los dos datos venían en un formato decente para su utilización. Pues, los archivos de audio ya vienen clasificados en su diferente tónica. Por lo que solo es necesario almacenarlos en nuestras variables X y Y, siendo X para cada uno de los archivos de audio, y Y su predicción si es mayor o menor.

```
def load_dataset(data_dir):  
    """  
    This function loads the dataset from the specified directory.  
    The dataset is expected to have two subdirectories: 'major' and 'minor'.  
    Each subdirectory should contain audio files corresponding to major and  
    minor chords, respectively.  
  
    Parameters:  
    - data_dir (str): path to the dataset directory  
    - Returns: X (np.ndarray), y (np.ndarray), filenames (list)  
    """  
    X, y, filenames = [], [], []  
    for label in ['major', 'minor']:  
        folder = os.path.join(data_dir, label)  
        for filename in os.listdir(folder):  
            if filename.endswith('.wav'):  
                audio_path = os.path.join(folder, filename)  
                spectrogram = audio_to_spectrogram(audio_path)  
                X.append(spectrogram)  
                y.append(0 if label == 'major' else 1)  
                filenames.append(filename)  
  
    return np.array(X), np.array(y), filenames
```

Figura 2.0. Función para cargar el dataset en arreglos para X y Y.

Pero antes de continuar, es importante que aclaremos algo muy importante. Al principio del modelo mencionamos que trabajaríamos en un CNN, pero esta última esta diseñada para analizar datos en forma de imágenes. Y nuestro conjunto de datos son archivos de audio. Entonces podríamos preguntarnos ¿cómo vamos a hacer un CNN si nuestros datos no están en formato de imágenes? Para ello debemos transformar el audio en una representación visual, un espectrograma. Un espectrograma es una imagen que muestra cómo cambian las frecuencias de audio en el tiempo. Y podemos utilizar un CNN para analizar estos espectrogramas ya como imágenes, y extraer patrones y características relevantes para la clasificación o el reconocimiento.

En la Figura 2.0 podemos observar la llamada a una función “audio_to_spectrogram” que, cómo su nombre sugiere, convierte los archivos de audio en espectrogramas, permitiendo al modelo trabajar sobre estas imágenes.

Esta función recibe cuatro parámetros:

1. audio_path: La ruta al archivo del audio a procesar.
2. n_fft: El tamaño de la ventana del FFT (Fast Fourier Transform), esta define el nivel de detalle frecuencial del espectrograma.
3. hop_length: El número de muestras entre ventanas sucesivas en la FFT, determina el solapamiento entre frames.
4. fixed_size: El tamaño deseado del espectrograma en alto y ancho.

Utilizamos la librería de librosa para cargar el archivo de audio, calcular el espectrograma en escala de mel, el cual es un tipo de espectrograma adaptado a cómo el oído humano percibe el sonido, y convertir la escala de potencia a decibelios para hacerla más interpretable. Por último, simplemente redimensionamos el espectrograma al tamaño fijo establecido en la función.

```
def audio_to_spectrogram(audio_path, n_fft=2048, hop_length=512, fixed_size=(128, 128)):
    """
    This function reads an audio file and computes its mel spectrogram.
    The mel spectrogram is then converted to decibels and resized to a fixed size.

    Parameters:
    - audio_path (str): path to the audio file
    - n_fft (int): length of the FFT window
    - hop_length (int): number of samples between successive frames
    - fixed_size (tuple): desired size of the spectrogram
    - Returns: resized mel spectrogram (np.ndarray)
    """
    # Load the audio file and compute its mel spectrogram
    y, sr = librosa.load(audio_path, sr=None)
    spectrogram = librosa.feature.melspectrogram(y=y, sr=sr, n_fft=n_fft, hop_length=hop_length)
    log_spectrogram = librosa.power_to_db(spectrogram, ref=np.max)

    # Resize the spectrogram to the fixed size
    if log_spectrogram.shape[1] < fixed_size[1]:
        pad_width = fixed_size[1] - log_spectrogram.shape[1]
        log_spectrogram = np.pad(log_spectrogram, ((0, 0), (0, pad_width)), mode='constant')

    return log_spectrogram[:fixed_size[0], :fixed_size[1]]
```

Figura 2.1. Función para convertir un audio a espectrograma.

En la función original para cargar el dataset, se manda a llamar esta función para cada archivo, las convertimos a espectrogramas, y las almacenamos en nuestro arreglo de X. Para el arreglo Y, guardamos la etiqueta de cada archivo, es decir, si es mayor o menor. Y con eso, ya tenemos nuestro dataset cargado y listo, por lo que ya podemos construir nuestro modelo.

Implementación de Red Neuronal

Llegamos a la implementación de nuestro modelo, cómo hemos mencionado en los puntos anteriores, vamos a trabajar con un CNN. Anteriormente mencionábamos el proceso de convertir los archivos de audio a espectrogramas para poder así trabajar con imágenes que nuestro CNN pueda procesar.

Construcción del modelo CNN

El modelo esta construido de la siguiente forma:

```
# Build the CNN model
model = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(128, 128, 1)),
    BatchNormalization(),
    MaxPooling2D((2, 2)),
    Dropout(0.3),

    Conv2D(64, (3, 3), activation='relu'),
    BatchNormalization(),
    MaxPooling2D((2, 2)),
    Dropout(0.3),

    Conv2D(128, (3, 3), activation='relu'),
    BatchNormalization(),
    MaxPooling2D((2, 2)),
    Dropout(0.4),

    Flatten(),
    Dense(128, activation='relu'),
    BatchNormalization(),
    Dropout(0.5),

    Dense(2, activation='softmax')
])
```

Figura 3.0. Estructura del modelo CNN

El modelo utiliza tres bloques convolucionales para extraer las características de nuestras imágenes, luego conecta estas características a capas densas que realizan la clasificación final. Echemos un vistazo a cada capa:

1. La primera capa convolucional es de 32 filtros, cada uno con tamaño de 3x3. Utilizamos relu como función de activación, introduce no linealidad, y nos permite que la red pueda aprender patrones complejos. Y por último definimos que las imágenes de entrada son de tamaño 128x128 y en escala de grises.

- a. **Batch Normalization:** Normaliza la salida de la capa anterior, estabilizando y acelerando el entrenamiento.
 - b. **Max Pooling:** Reducimos la resolución de las características de entrada a la mitad, eso nos ayuda a reducir el tamaño de la imagen y captura características más generales.
 - c. **Dropout:** Desconectamos al azar el 30% de las neuronas durante el entrenamiento para evitar overfitting y mejorar la capacidad de generalización.
2. En la segunda capa, utilizamos 64 filtros de tamaño 3x3. Esto nos permite extraer características más complejas y detalladas de las imágenes de entrada, como bordes y texturas de mayor nivel. Volvemos a utilizar relu como función de activación para ayudar a la red a aprender representaciones más complejas.
- a. **Batch Normalization:** Volvemos a utilizar esta función para normalizar las salidas de la capa para que tengan una media cercana a 0 y una desviación estándar cercana a uno, de este modo, podemos estabilizar y acelerar el entrenamiento.
 - b. **Max Pooling:** Al igual que la capa anterior, volvemos a utilizar esta función para reducir las dimensiones de las características a la mitad, de ese modo podemos conservar la información más relevante, y reducimos el número de parámetros, mejorando la eficiencia de la red.
 - c. **Dropout:** Del mismo modo, desactivamos el 30% de las neuronas de esta capa durante cada iteración del entrenamiento. Como mencionábamos, esto nos ayudará a prevenir el overfitting al forzar que la red aprenda patrones robustos sin depender de neuronas específicas.
3. En la tercera capa utilizamos 128 filtros con tamaño de 3x3. De este modo, la red profundiza aún más en la extracción de características, logrando capturar patrones aún más complejos que en las capas anteriores. Al igual que las capas anteriores, utilizamos relu para mantener la no linealidad en las representaciones.

- a. **Batch Normalization:** Volvemos a utilizar esta función para normalizar la salida de la capa, de ese modo aseguramos que los valores estén bien distribuidos y facilitando la optimización y la estabilidad.
 - b. **Max Pooling:** Nuevamente, nos ayuda a reducir a la mitad, la resolución de las características, extrayendo patrones más generales, y resumiendo la información clave.
 - c. **Dropout:** En esta ocasión, estamos desactivando un 40% de las neuronas en el entrenamiento. Estamos usando este valor ya que la red está en etapas donde tiene características de alto nivel, por lo tanto, queremos asegurarnos de prevenir el overfitting.
4. En nuestra capa final, estamos usando Flatten para convertir la salida 2D en un vector 1D, permitiéndonos conectar con la capa densa.
- a. **Dense:** La capa densa esta completamente conectada con 128 neuronas. En este punto se realiza la clasificación en función de las características aprendidas.
 - b. **Batch Normalization:** Como hemos mencionado anteriormente, utilizamos esta función para normalizar la salida para asegurar que los valores estén distribuidos de forma correcta y mejorar la optimización y estabilidad del modelo.
 - c. **Dropout:** En este caso desconectamos el 50% de las neuronas para reducir el riesgo de un overfitting.
 - d. **Dense (2):** La capa de salida tiene 2 neuronas, una para cada clase (mayor o menor). Utilizamos softmax para convertir los valores en probabilidades, permitiendo obtener la clase final.

Entrenamiento del modelo

Además del modelo definido anteriormente. Hemos implementado algunas optimizaciones para ayudar al modelo a tener un mejor rendimiento y estabilidad, y evitar el overfitting.

1. **Optimizador:** Estamos utilizando AdamW, le asignamos un learning rate inicial de $1e-5$. Además, estamos definiendo un weight decay de 0.001 para la regularización.

```
# Create an optimizer
optimizer = AdamW(learning_rate=0.001, weight_decay=1e-5)
```

Figura 3.1. Optimizador AdamW para el modelo.

2. **Checkpoint para el mejor modelo:** Este nos permite guardar el resultado de los modelos en cada epoch, en cada uno de los pasos del entrenamiento, el callback analiza la precisión en validation en el epoch actual, y si el resultado es mejor que el epoch anterior, entonces el modelo se guarda y nos permitirá utilizar ese modelo guardado para las predicciones.

```
# Create a callback to save the best model during training
checkpoint_callback = ModelCheckpoint(
    'models/temp_best_model.keras',
    monitor='val_accuracy',
    mode='max',
    save_best_only=True,
    verbose=1
)
```

Figura 3.2. Callback para guardar el mejor modelo durante el entrenamiento.

3. **Callback para reducir el learning rate:** Utilizamos una instancia de ReduceLROnPlateau, como su nombre sugiere, reducimos el learning rate durante el entrenamiento cuando el modelo deja de mejorar, lo que puede ayudar a salir de mínimos locales y mejorar el rendimiento en general.

```
# Create a callback to reduce the learning rate when the validation loss plateaus
lr_reduction = ReduceLROnPlateau(monitor='val_loss', patience=3, factor=0.5, min_lr=1e-6)
```

Figura 3.3. Callback para reducir el learning rate.

Una vez que hemos definido nuestras optimizaciones, el siguiente paso es entrenar el modelo, analizar los resultados y tratar de realizar predicciones con nuestro modelo. En nuestro caso, veremos si nuestro modelo es capaz de identificar si un acorde es mayor o menor.

Análisis de Resultados

Para el entrenamiento del modelo hemos utilizado un total de 50 epochs, y un batch_size de 32, el cual establece el tamaño de la cantidad de muestras que se procesan antes de actualizar los parámetros del modelo. Con ese tamaño definido, el modelo ajustará los pesos cada vez que haya procesado 32 muestras.

Echemos un vistazo a los resultados.

Resultados R^2

```
Epoch 43/50 -- 0s 156ms/step - accuracy: 0.9768 - loss: 0.0418  
13/13  
Epoch 43: val_accuracy improved from 0.92000 to 0.94000, saving model to models/temp_best_model.keras  
13/13 -- 2s 179ms/step - accuracy: 0.9767 - loss: 0.0425 - val_accuracy: 0.9400 - val_loss: 0.1773 - learning_rate: 6.2500e-05
```

Figura 4.0. Epoch del mejor modelo adquirido durante el entrenamiento

1. **Train R^2 → 97.67%:** La precisión en training nos dice que el modelo clasifica correctamente si un acorde es mayor o menor en aproximadamente 97.67% de los casos.
2. **Validation R^2 → 94%:** La precisión en validation nos dice que tenemos un buen indicador de cómo el modelo podría funcionar con datos nuevos, en este caso, nos dice que el modelo podría ser efectivo para acordes nuevos e identificar si su tónica es mayor o menor.

La diferencia entre ambas precisiones es pequeña. Esto es un resultado positivo, ya que una diferencia más alta nos indicaría que el modelo tiene overfitting. Y en nuestro modelo, la diferencia es pequeña como para sugerir que el modelo está generalizando bien, sin memorizar en exceso los datos de entrenamiento.

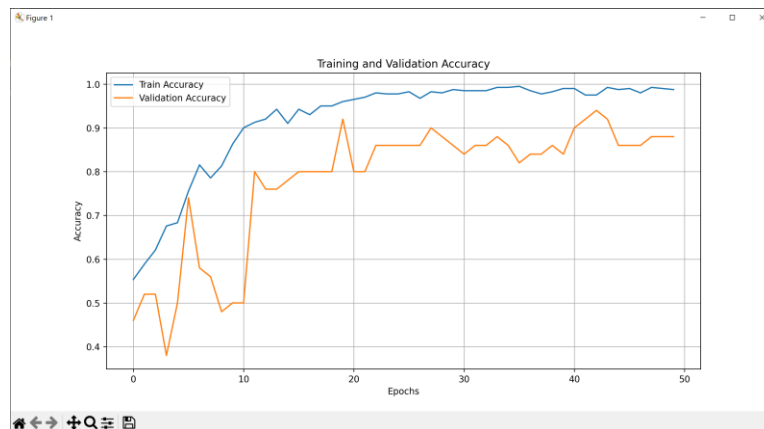


Figura 4.1. Visualización de la precisión en Training y Validation

Pérdida

1. **Training** → **0.0425**: Esta pérdida es una medida de cuán bien el modelo se ajusta a los datos, en este caso la pérdida es muy baja, lo que nos indica que tenemos un buen ajuste en el modelo.
2. **Validation** → **0.1773**: Este valor es más alto que la pérdida en training, lo cual es normal. En los modelos bien ajustados, se espera que la pérdida de validación sea un poco más alta que la de entrenamiento, ya que el modelo está expuesto a datos que no ha visto antes.

La diferencia entre training y validation sugiere que el modelo está funcionando bien, aunque esta diferencia nos dice que el modelo tiene un pequeño margen para mejorar su capacidad de generalización. Pero en general, nos indica que el problema de overfitting es mínimo.

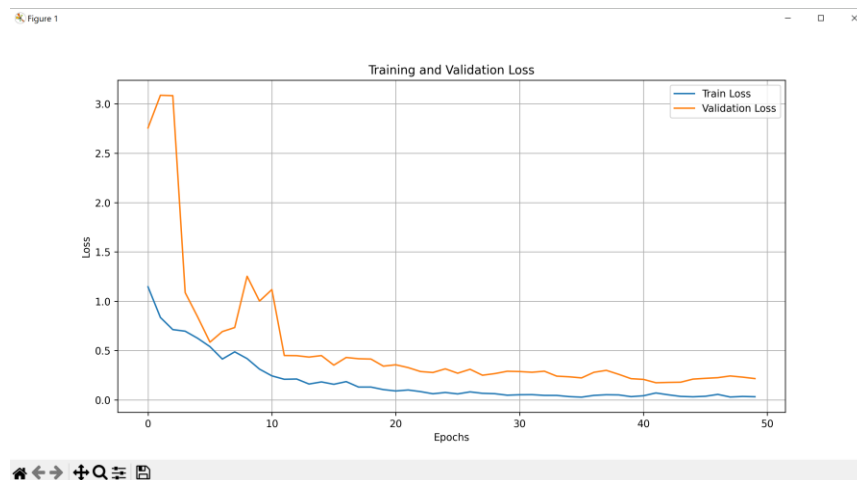


Figura 4.2. Visualización de la pérdida en Training y Validation

Desempeño en Testing

```
2/2 ————— 0s 22ms/step - accuracy: 0.9269 - loss: 0.3465
2/2 ————— 0s 102ms/step
Test Accuracy: 92.16%
```

Figura 4.3. Desempeño del modelo en Testing

Adicional al análisis del desempeño en Training y Validation, hemos separado parte del dataset en Testing para probar y realizar predicciones rápidas. Al evaluar nuestro modelo, obtuvimos una precisión del 92.16%, con una pérdida de 0.3465. Lo que nos permite decir que el modelo en general tiene un buen desempeño, la

pérdida es un poco mayor a la pérdida en Validation, sin embargo, la diferencia es muy pequeña, por lo que nos permite decir que el modelo aún tiene buen ajuste, y es lo suficientemente apto para determinar la tónica mayor o menor de un acorde.

```
Predictions:
Chord: 3_2.wav - Prediction: minor
Chord: 4_5.wav - Prediction: major
Chord: 7_52.wav - Prediction: minor
Chord: 4_19.wav - Prediction: major
Chord: 5_25.wav - Prediction: major
Chord: 5_12.wav - Prediction: major
Chord: 9_74.wav - Prediction: major
Chord: 3_7.wav - Prediction: major
Chord: 1_7.wav - Prediction: major
Chord: 1_26.wav - Prediction: minor
```

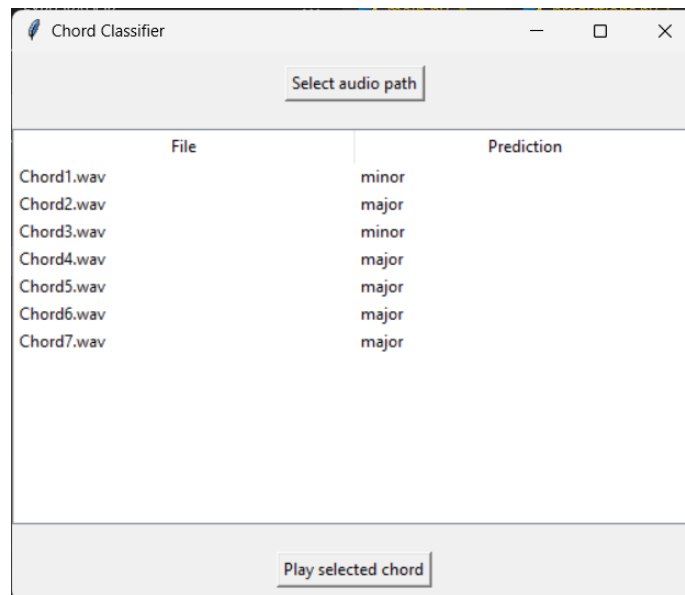
Figura 4.4. Ejemplo de predicciones en Testing

Adicional al modelo, hemos implementado que pueda realizar predicciones en el conjunto de Testing del dataset original, en general podemos ver que tiene un buen desempeño y en la mayoría de los acordes presentados en la Figura 4.4 ha obtenido un resultado correcto. Sin embargo, hay que aclarar algo importante, en el dataset, tanto en la subcarpeta mayor como en menor, hay casos donde el archivo se llama exactamente igual que en el subconjunto de la tónica contraria, por lo que algunos de estos resultados se mantienen ambiguos, pues no sabemos a cuál de estos archivos está realizando la predicción.

Para poder determinar la predicción sin ambigüedad y determinar si se equivoca o no, vamos a realizar las predicciones por aparte. En la siguiente sección veremos a detalle cómo hemos realizado el código para las predicciones, y veremos los resultados finales.

Predicciones del modelo

En la sección anterior vimos la capacidad del modelo para realizar predicciones en el conjunto de Testing, sin embargo, pudimos notar que los resultados de esas predicciones son ambiguos y no nos permiten llegar a un resultado completo. Para eso, hemos implementado un código por aparte para poder determinar con mayor exactitud las predicciones del modelo. Para ello hemos construido una interfaz, en la que estamos utilizando un pequeño conjunto con datos ajenos al dataset original. En la interfaz se nos permite la opción de escuchar el acorde, y adicionalmente incluye la predicción, de ese modo, nos permite a nosotros como humanos escuchar el acorde y por cuenta propia determinar si es mayor o menor, y comparar con el resultado de la predicción del modelo.



The screenshot shows a window titled "Chord Classifier" with a "Select audio path" button at the top. Below is a table with two columns: "File" and "Prediction". The table lists seven audio files and their corresponding predicted chord types. At the bottom of the window is a "Play selected chord" button.

File	Prediction
Chord1.wav	minor
Chord2.wav	major
Chord3.wav	minor
Chord4.wav	major
Chord5.wav	major
Chord6.wav	major
Chord7.wav	major

Figura 5.0. Visualización gráfica de las predicciones

En este ejemplo podemos ver a mejor detalle las predicciones del modelo, cómo mencionábamos anteriormente, podemos escuchar el acorde que queramos y a simple oído podemos determinar si es mayor o menor y compararlo con la predicción del modelo.

Después de realizar esta comparación podemos determinar con mayor detalle que el modelo aún tiene áreas de mejora en la que podría mejorar su desempeño, pues algunas de estas predicciones son incorrectas. Esto puede ser por algunas razones

diferentes, puede ser que cómo mencionábamos anteriormente, el modelo aún tiene un poco de margen de mejora, y podríamos intentar mejorar su desempeño para obtener resultados más precisos, o puede ser que cómo los acordes utilizados para este nuevo conjunto de prueba, al ser diferentes que los del conjunto original, realiza predicciones diferentes debido a cómo se ve su espectrograma, por lo que quizás sea necesario entrenar el modelo con más datos, en este caso más ejemplos de acordes.

Conclusiones

Recapitulando lo que hemos hablado en este reporte, hemos visto un poco sobre teoría de los acordes y cómo nosotros como personas podemos identificar la tónica mayor o menor de un acorde. Hemos visto las técnicas y herramientas utilizadas en la construcción del modelo. Visualizamos los resultados del modelo Training, Validation y Testing. Y hemos visto la capacidad del modelo para predecir la tónica mayor o menor de un acorde.

Pudimos observar que el modelo tiene áreas de oportunidad en las predicciones, por lo que podemos ajustar aún más el modelo para tratar de obtener mejores resultados. Pero en general pudimos observar que el modelo tiene un buen desempeño para una tarea complicada, pues al trabajar con archivos de audio en un CNN, tuvimos que convertir en espectrogramas estos archivos para que el modelo pudiera trabajar con estas imágenes, y un espectrograma a simple vista es complicado para analizar, por lo que haber obtenido un desempeño como el que vimos en el modelo, nos da un resultado positivo y nos permite concluir que el modelo es lo suficientemente apto para predecir la tónica mayor o menor de un acorde.

Referencias

Анатолий Михайлин (2021). *Major VS Minor guitar chords*. Kaggle.

<https://www.kaggle.com/datasets/mehanat96/major-vs-minor-guitar-chords>

Vinci F. (2021). *GUITAR CHORDS V3*. Kaggle.

<https://www.kaggle.com/datasets/fabianavinci/guitar-chords-v3>

Hugging Face (s.f.). *Introducción a los datos de audio*.

https://huggingface.co/learn/audio-course/es/chapter1/audio_data

IBM (s.f.). *¿Qué son las redes neuronales convolucionales?*

<https://www.ibm.com/mx-es/topics/convolutional-neural-networks>