E.T.S. de Ingeniería Industrial,
Informática y de Telecomunicación

Análisis de Sentimientos en Armonías y Melodías mediante Deep Learning

Grado en Ingeniería Informática

Trabajo Fin de Grado

Alumno: Xabier Dendarieta Sarries

Tutor: Mikel Galar Idoate

Pamplona, 21 de marzo de 2019

upna Universidad Pública de Navarra
Nafarroako Unibertsitate Publikoa

Resumen

Este proyecto consiste en determinar la capacidad de las redes neuronales profundas (Deep Learning) para analizar el sentimiento que transmite la música. La música es algo que transmite sentimientos de una forma compleja y por tanto es difícil de tratar.

Para realizar este trabajo, se transformará la señal de audio a una representación basada en frecuencias (espectrograma) que será utilizado por las redes neuronales para clasificar cada fragmento de audio en uno de los sentimientos establecidos.

Estudiamos primero las clases de sentimientos a considerar, posteriormente creamos una base de datos propia sobre la que aplicar los algoritmos de Deep Learning.

Dividimos el problema en dos partes: clasificación de la energía que transmite una canción; y clasificación del placer que transmite. Realizamos pruebas con varios tipos de redes neuronales recurrentes, obteniendo buenos resultados en la clasificación de la energía y malos resultados en la clasificación del placer. También probamos una red convolucional para poder comparar los diferentes tipos de red.

Creemos que el mayor problema reside en la base de datos creada, tanto en su tamaño como en su consistencia. La tecnología como tal, resulta prometedora a pesar de la mayoría de los resultados.

Palabras clave

Deep Learning, Redes Neuronales Recurrentes, análisis de sentimientos, música, emociones.

Contenido

1 Introducción

Vivimos en una época en la que estamos rodeados de música. Cada año surgen y se olvidan cientos de autores. Cada año cambian los temas populares. Cada año se generan grandes cantidades de ingresos con esto. La industria está a rebosar. Pero, a pesar de todo, sigue habiendo música muy diferente: diferentes estilos, diferentes composiciones, diferentes sentimientos transmitidos.

El análisis de sentimientos es algo que lleva practicándose en computación varios años atrás. Sin embargo, esto suele hacerse sobre texto. Los ejemplos más conocidos son el interpretar opiniones sobre productos en los comentarios de las páginas de ventas o incluso en redes sociales. Suele ofrecer buenos resultados. Sin embargo, puede resultar interesante poder analizar el nivel de felicidad o tristeza que puede transmitir una canción, más allá del texto empleado en esta. El nivel de tensión o calma que puede generar en quien la escucha.

Vivimos en una época en la que la tecnología crece cada vez más rápido. Y en esta época hace ya algunos años que surgió el Machine Learning y con él las Redes Neuronales Profundas, rebautizadas como Deep Learning. Cuando éstas surgieron, hace ya un par de décadas, la tecnología para procesar estas arquitecturas no era suficiente. Sin embargo, a estas alturas somos capaces de crear arquitecturas profundas y de conseguir que aprendan a realizar tareas complejas de forma mucho más sencilla.

Con este proyecto se pretende ver hasta qué punto las Redes Neuronales Recurrentes son capaces de aprender a distinguir patrones en audio, que no tengan que ver con las palabras como tal. Nuestro lenguaje puede cambiar el significado según algo tan sutil como la entonación de algunas palabras, por lo que, si se consigue aprender algo tan delicado como lo agradable que resulta una canción, comprobamos que el Deep Learning es capaz de extraer información del sonido, y no únicamente de las palabras dichas. A parte, puede ser útil poder clasificar la música de forma automática en función de lo que transmita.

Si bien ya hay plataformas que hacen una distinción parecida, estas suelen llevar consigo el género de la canción. Pero, en nuestro caso, creemos que la clasificación por sentimientos tiene que ser independiente al género que se le pueda atribuir a cada canción en concreto. Por ejemplo, no debería importarnos si hablamos de una cumbia o una polka si ambas nos transmiten felicidad y ganas de bailar. Ante la falta de investigaciones al respecto hemos decidido ser nosotros los creadores de tales arquitecturas de Deep Learning que sean capaces de tal clasificación.

El principal objetivo del proyecto será crear un modelo de Redes Neuronales que clasifique la música en cuatro sentimientos básicos: Estrés, Felicidad, Calma y Tristeza. Si bien, estos sentimientos no son perfectamente separables, nos centraremos en diferenciarlos de un modo diferente. Se clasificará la música según su energía y según su agradabilidad. Al combinar ambos parámetros tendremos las cuatro clases mencionadas anteriormente. Explicamos este aspecto en profundidad en el capítulo siguiente. Para llegar a conseguir esto, necesitaremos plantear antes los pasos a seguir.

En primer lugar, será necesario disponer de una base de datos de música etiquetada de forma en la que podamos extraer las etiquetas requeridas, o, directamente, que ya tenga esas

etiquetas. Empezaremos buscando información sobre el tema propuesto y con ello buscando datasets ya preparados en la web. En caso de no encontrar nada que podamos utilizar pasaremos a crear nuestra propia base de datos y de ella nuestro dataset a medida. La idea de todo esto es conseguir un dataset de clips de audio, de varios segundos, descompuestos en frecuencias (mediante el espectrograma), y que cada uno de los clips tenga asociado una etiqueta.

Por supuesto, las etiquetas tendrán que seguir cierto patrón. Un modelo de clasificación de emociones que tendremos que decidir e implementar más adelante. De este modo, tendremos una buena forma de clasificar los sentimientos más allá de las típicas etiquetas de *bueno* y *malo*, *positivo* y *negativo* o *alegre* y *triste* que son tan vistas para el análisis de sentimientos tradicional en texto. Ya que, para nuestro caso, el uso de dos únicas etiquetas como estas podría resultar muy pobre.

A continuación, procederemos a la creación de los modelos de Deep Learning que clasifiquen el dataset obtenido anteriormente. Es esperado que esto sea algo costoso, ya que al comenzar a tratar los datos de cero habrá que hacer una exploración amplia de los parámetros a utilizar con diferentes valores y de diferentes arquitecturas. Es decir, no va a ser algo inmediato, habrá muchas posibilidades a probar.

Como es un problema que utiliza datos secuenciales, nos centraremos en el uso de las Redes Neuronales Recurrentes, ya que estas nos permiten el análisis de secuencias de una forma cómoda y ligera (no suelen tener cantidades inmensas de parámetros, como las Redes Convolucionales). Probaremos también el uso de las celdas de memoria en las citadas redes recurrentes, que probablemente mejoren mucho el aprendizaje con respecto a las que no tengan memoria.

En definitiva, el objetivo principal de este trabajo es crear un modelo basado en Deep Learning, que consiga interpretar los sentimientos primarios que transmite la música. Empleando para ello el audio únicamente, sin entrar en artistas, géneros, letras, y demás metainformación disponible dentro de una canción.

Para ello, necesitamos cumplir los siguientes objetivos particulares:

1. **Base de datos**
   Necesitaremos buscar una base de datos online y etiquetarla según nuestras necesidades (es un etiquetado muy específico, no habrá nada parecido por la web), y en caso de no encontrar nada, buscar música gratuita (sin copyright), realizar nuestra base de datos y etiquetarla. De la base de datos que consigamos deberemos extraer un dataset preprocesado con el que trabajar.

2. **Aprendizaje del modelo**
   Utilizaremos el dataset para poner a aprender varios modelos, con diferentes hiperparámetros y arquitecturas, para ver cómo se comporta cada uno y cuando tengamos alguno con un aprendizaje más estable nos centraremos en él/ellos. Hay muchos hiperparámetros y muchas variantes a probar.

2 Análisis de sentimientos

A lo largo de los años ha habido diversos estudios tratando de buscar una solución al problema de clasificación de la música. Han sido exploradas diferentes formas de enfocar el problema y diferentes técnicas y algoritmos para resolverlo.

Al principio, el problema se resumía en la utilización de las etiquetas como el género o el autor para determinar más o menos como podía ser la canción [1]. Pero en seguida aparecieron estudios tratando el audio de otra forma. Se empezó a probar obteniendo características rítmicas y tonales de las canciones [2]. En algún estudio se propuso el uso exclusivo de las características tonales, porque, según los autores, las rítmicas no aportaban valor al sentimiento generado [3]. Las características se obtienen mediante el uso de MFCC (Mel-Frequency Cepsal Coefficient), y el procesado de algunas de estas.

La transformación a frecuencias (como con MFCC) es algo que parece natural. El oído humano actúa de la misma forma: divide el sonido en frecuencias mediante la cóclea (o caracol, situado en el oído interno) y cada una de las frecuencias es recogida por una conexión neuronal diferente [4]. Por tanto, si nosotros transformamos el audio en crudo (la señal de amplitud) a su espectrograma (la transformación a frecuencias), tendremos una mejor representación de lo que el oído humano percibe y, por tanto, la red neuronal actuará de una forma más similar a la humana.

También hay quienes han enfocado el problema desde el lado de las letras, y han estudiado y creado clasificadores de sentimientos para tratar las letras de las canciones [5]. En nuestro caso, esto no es de especial interés. Incluso, un gran porcentaje de los clips que trataremos lo más probable es que ni si quiera tengan letra, que sean instrumentales.

Son típicos los usos de algoritmos como el RAkEL (Random k-Labelsets) y el MLkNN (Multi-Label k-Nearest Neighbors), estando presentes en la gran mayoría de los artículos al respecto [3]. Y en menor medida, pero también habitual, se ven las SVM (Support Vector Machines). Hay incluso algunos intentos de utilizar el Perceptrón Multicapa [1], pero sin llegar a un gran número de capas ya que, por aquel entonces, no había tanta potencia de procesamiento. El uso de fragmentos de canciones de longitud idéntica y pequeña es común puesto que simplifica el procesamiento de estos y mejora el manejo del dataset, ya que, de este modo tiene unas dimensiones fijas [2].

Pero las redes neuronales han adquirido una fuerza mayor estos últimos años, por la mejora teórica de los modelos, por su optimización en la práctica y por la mejora en la potencia de procesamiento de las maquinas. Con esto, ya somos capaces de utilizar el Deep Learning a un nivel mucho más potente, y presenta una alternativa prometedora ante los estudios anteriores.

En cuanto a la clasificación de sentimientos, podríamos plantear una clasificación por etiquetas de forma que el clasificador nos indique para cada ejemplo con qué certeza es de cada clase. Pero nosotros vamos a intentar buscar otro modo de clasificar, puesto que, a la hora de hablar de sentimientos, tenemos que estos no tienen por qué ser perfectamente separables, sin correlación, y un único ejemplo no debe clasificarse en dos clases inversamente proporcionales a la vez. Por tanto, los modelos que clasifiquen según un espacio (probablemente bidimensional) en el que se distribuyan las emociones parecen una opción más ajustada a nuestro caso.

Hay varios modelos dimensionales desarrollados desde hace algo más de un siglo (el primero: Wilhelm Max, 1897), los cuales pueden ser muy útiles a la hora de enfrentar nuestro problema. Se han desarrollado desde entonces modelos variados, llegando incluso a haber modelos tridimensionales. En concreto, los modelos bidimensionales de Thayer [6] y Russell [7] parecen una buena elección, por su simplicidad e intuitividad. En la literatura actual, estos son los más usados, por no decir los únicos.

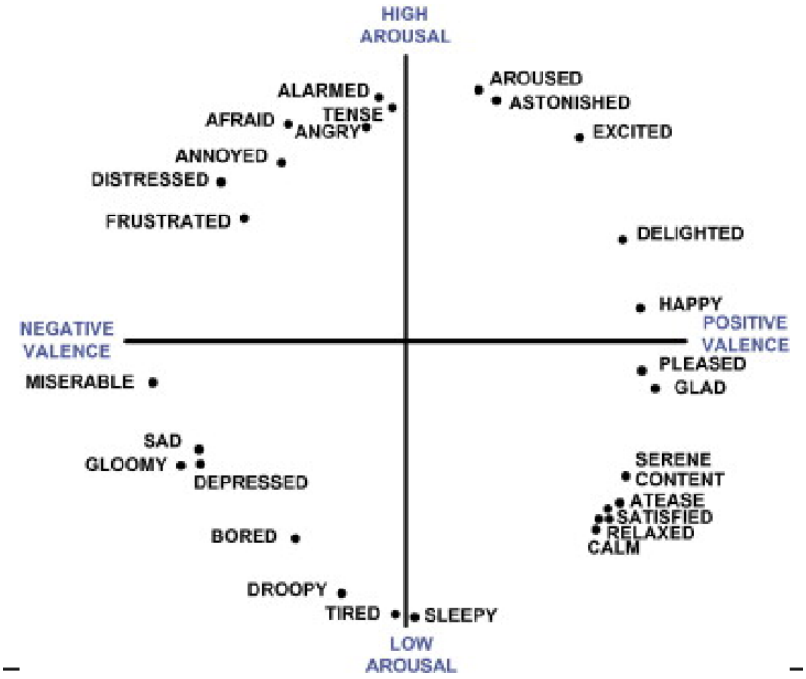

Figura 2-1 Modelo de emociones de Russell. Fuente.

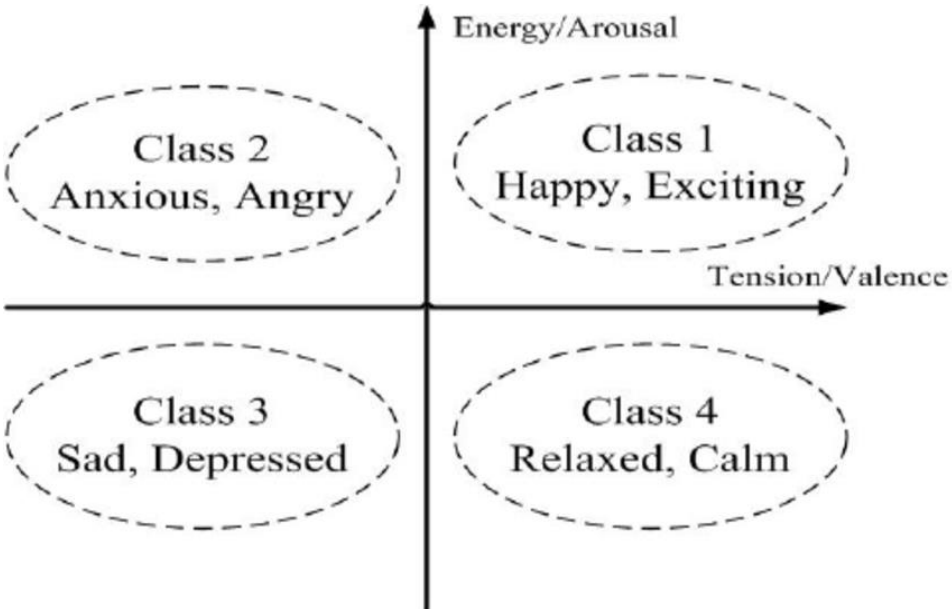

Figura 2-2: Modelo de emociones de Thayer. Fuente.

El modelo de Russell es anterior al de Thayer, pero contempla, quizá, demasiadas emociones, por lo que a la hora de clasificar complicarían la tarea. Dicho lo cual, el modelo de Thayer parece la mejor opción sobre la mesa. Está basado en las dimensiones Arousal (Energía) y Valence (Tensión), como veremos a continuación.

2.1 Modelo de emociones

El modelo desarrollado por Thayer está basado en las dimensiones Arousal y Valence, que al ser bidimensional nos da facilidad para definir varias emociones. De esta forma son necesarios únicamente dos números para definir cuatro clústeres de sentimientos, y por tanto cuatro sentimientos primarios, de los que deriva el resto.

Primero está la dimensión Arousal, que podemos traducirlo como Energía y mide la fuerza o el ritmo con el que se siente la emoción. Sería el grado de estimulación que provoca el audio.

Y segundo tenemos Valence, podríamos traducirlo como Tensión y mide la intensidad con la que se siente la emoción. Sería el grado de agradabilidad (o en negativo tristeza) que provoca el audio.

Combinando ambos valores, se puede obtener un abanico muy amplio de emociones diferentes (como es el caso del modelo de Russell), pero nosotros nos quedaremos con las emociones que representan cada cuadrante del grafico bidimensional: Enfado (Anger), Felicidad (Happiness), Tristeza (Sadness) y Calma (Calm).

En la literatura actual, se da el caso de que Felicidad y Tristeza aparecen siempre que se utilizan modelos de este tipo, pero Enfado y Calma varían entre otras opciones muy similares, como Ansiedad y Relax [5]. Pero, al fin y al cabo, todas estas etiquetas engloban las mismas clases de sentimientos. Hay que remarcar el hecho de que estamos englobando muchas emociones en cuatro, por tanto, podemos hablar como cada una de ellas como la representante del clúster. Es decir, cuando digamos que una canción transmite calma, realmente no tiene porqué ser calma tal y como la concebimos, sino alguna de las emociones que este cuadrante engloba, como serenidad, ganas de dormir o relax.

El modelo que vamos a utilizar está basado en el de Thayer, sin embargo, me he tomado la libertad de cambiar algunos términos, para que todo tenga más sentido a la hora de etiquetar y clasificar la música. De este modo los ejes se llamarán: Energía (Activo-Pasivo), sustituyendo a Arousal; y Placer (Positivo-Negativo), sustituyendo a Valence. Los cuadrantes pasan a ser llamados Estrés (sustituyendo a Enfado), Felicidad, Calma y Tristeza. De este modo tendríamos un modelo en el que el primer eje (Energía, en la posición del eje Y tradicional) representaría la intensidad con la que se siente la emoción representada en el segundo eje (Placer, en la posición del eje X tradicional).

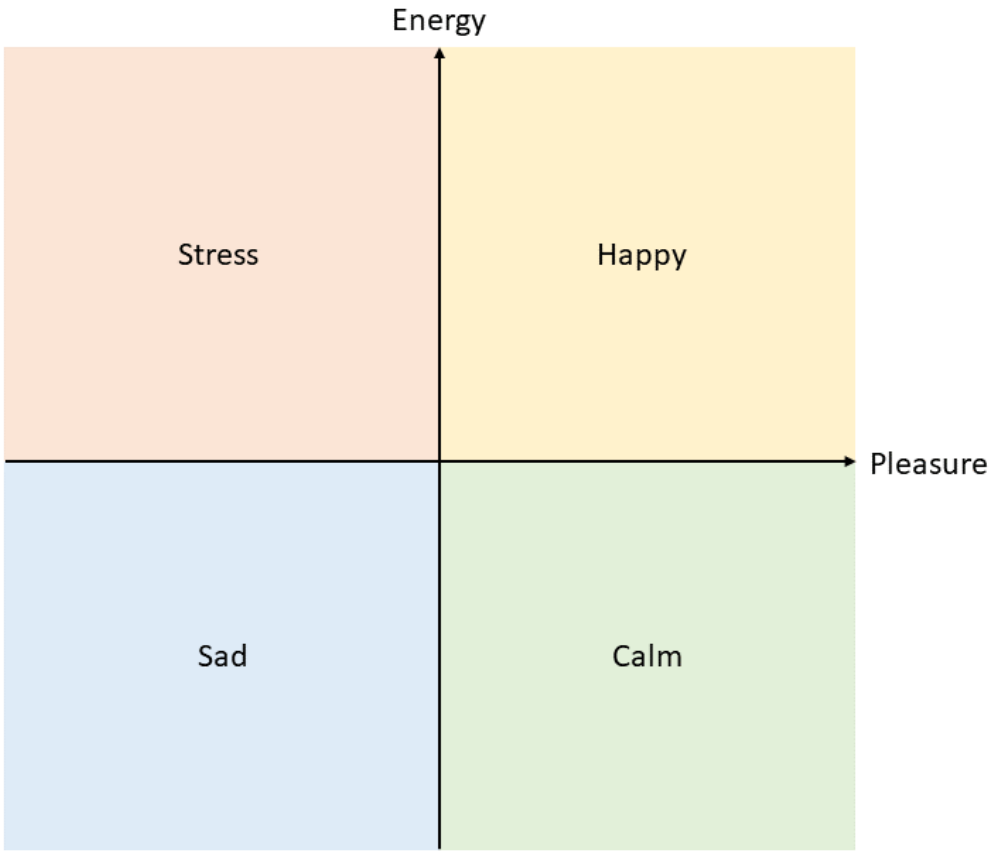

Figura 2-3: Modelo propio sobre el que trabajaremos.

Con esto, tenemos un modelo de clasificación de sentimientos que parece que va a encajar con nuestro problema de una forma más natural que las etiquetas tradicionales, que no expresarían relación entre ellas.

3 Deep Learning

3.1 Introducción

La unidad básica de la que partimos para crear una Red Neuronal Artificial es el perceptrón [8], que representa a una neurona. El perceptrón está estructurado de forma que tiene varios valores de entrada y un único valor de salida.

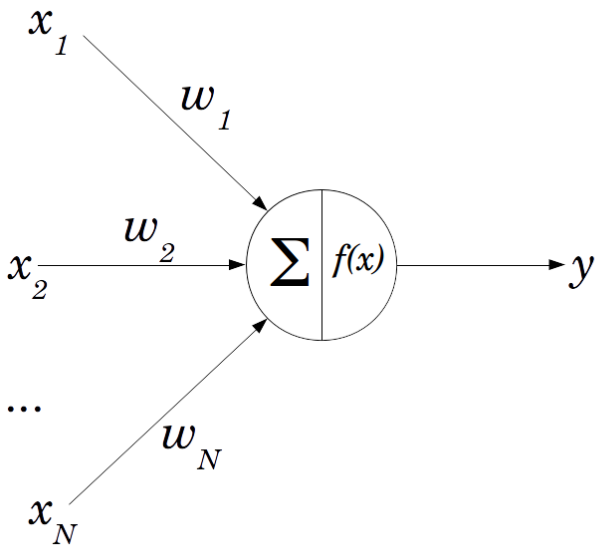

Figura 3-1: Representación del Perceptrón. Fuente.

Su funcionamiento se basa en la combinación de la Regresión Lineal con una función de activación. Es decir, hace una suma ponderada de todos los valores de entrada y utiliza una función (por lo general no lineal) para distorsionar el resultado, obteniendo así el valor de salida.

En la figura 3-1 no está contemplado el termino de bias, pero es algo necesario. Este es un valor que se suma de igual manera que los valores de entrada, y sirve para elevar el resultado de la suma de los valores de entrada, o disminuirlo en caso de ser negativo. Podríamos decir que es una especie de x_0 que siempre toma el valor de +1, aunque a la hora de hacer las implementaciones modernas, por comodidad, se trata por separado. Por tanto, la ecuación final del perceptrón puede expresarse de la siguiente manera:

$$f\left(\sum_{i=1}^{N} w_i \cdot x_i + b\right) = y$$

Ecuación 3-1: Ecuación final del Perceptrón

Por comodidad, utilizaremos una representación matricial en vez del sumatorio, para representar todas las neuronas de una capa a la vez.

Pero este modelo por sí solo lo único que puede hacer es una clasificación lineal, en otras palabras, solo es capaz de distinguir dos clases separadas mediante una línea. Por esto, se empezaron a concatenar perceptrones en una estructura por capas, obteniendo modelos más complejos, que, a su vez, aumentaron las posibilidades de clasificación. Con esto, ya somos capaces de crear clasificaciones mucho más flexibles y complicadas. Aquí es donde introducir la

no linealidad gana más sentido, ya que la combinación de funciones lineales da otra lineal, pero al introducir dicha no linealidad logramos tener modelos no lineales complejos.

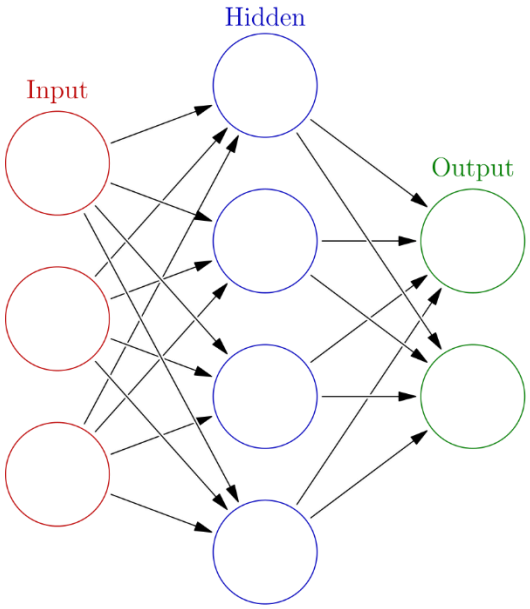

Figura 3-2: Red Neuronal de tres capas. Fuente.

Nótese que gracias a estas estructuras ahora podemos obtener más de un valor de salida, permitiendo así la obtención de resultados más complejos, como clasificaciones multiclase.

En la estructura de una red neuronal diferenciamos tres tipos de capas: la capa de entrada, situada a la izquierda y con la función de recoger los valores de entrada a la red; la capa de salida, situada a la derecha y es la encargada de obtener los valores de salida definitivos; y las capas ocultas, que son todas las que están entre las dos anteriores. Las neuronas conectan cada una con todas las de la siguiente capa, generando así un alto número de conexiones entre cada capa y su siguiente.

La idea tras esta estructura es propagar los valores de salida de cada neurona de una capa hasta las de la siguiente, de forma que los datos entren por la capa de entrada, se propaguen y transformen por las capas ocultas y acaben saliendo los resultados por la capa de salida. Cada una de estas conexiones, al igual que con los valores de entrada y los valores de bias (hay un bias en cada neurona) tiene su propio peso para ponderar la suma que mencionábamos en el perceptrón, por lo que los resultados que obtengamos de la red dependen directamente de lo *buenos* que sean tales pesos.

Se dice que una red neuronal es más profunda cuantas más capas ocultas tiene, de ahí que el termino Deep Learning, literalmente traducido como aprendizaje profundo, apareciera para denominar estos modelos.

Para que todo este modelo pueda funcionar correctamente hace falta un algoritmo de aprendizaje que permita recalibrar la red en función de los fallos que comete. Este algoritmo es denominado Back Propagation o propagación hacia atrás, y consiste en el uso de una función de

coste (a elegir) que derivamos para obtener el gradiente que nos guía hacia una disposición de pesos mejor por cada fallo que cometemos. No vamos a ver esto más en detalle, hoy en día las librerías hacen todo esto de manera automática.

Sin embrago, sí que es importante la elección de la función de coste que elijamos, que será la que nos permita saber cómo de bueno (o malo) es el modelo con los pesos actuales y será la que le dirá a la librería como realizar la propagación hacia atrás.

Por otro lado, también tenemos distintas funciones de activación a elegir [9]. Cada una de ellas es recomendable utilizarla en ciertas ocasiones.

Sigmoide

La función sigmoide, también representado como σ, es utilizada sobre todo para las capas de salida, ya que nos devuelve valores en el rango (0, 1). Se define de la siguiente manera:

$$f(z) = \frac{1}{1 - e^{-z}}$$

Ecuación 3-2: Función sigmoide

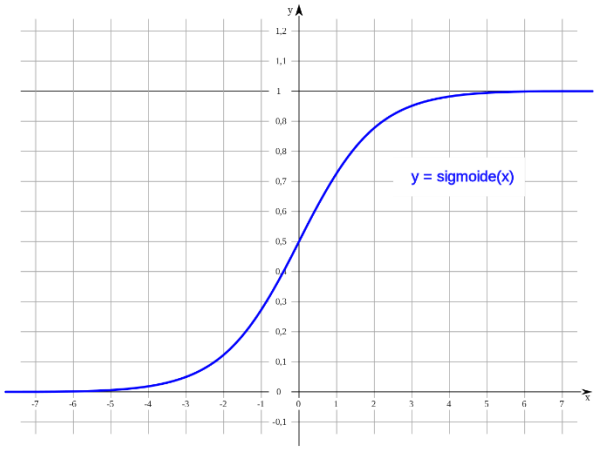

Figura 3-3: Representación gráfica de la sigmoide. Fuente.

Tanh

La función de tangente hiperbólica, o tanh para abreviar, es una función de activación que se recomienda utilizar en las capas ocultas, y permite resultados entre -1 y 1, por lo que en ocasiones puede ser útil también en las capas de salida. Puede definirse de la siguiente forma:

$$f(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

Ecuación 3-3: Función tanh simplificada

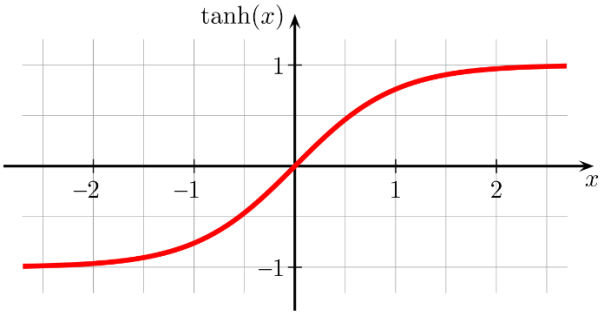

Figura 3-4: Representación gráfica de la tanh. Fuente.

ReLU

La Rectified Linear Unit, o ReLU, es recomendada para su uso en las capas ocultas de la red. Se define como:

$$f(z) = max(0, z)$$

Ecuación 3-4: Función ReLU

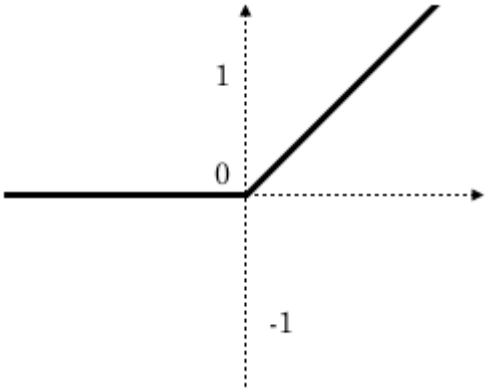

Figura 3-5: Representación gráfica de la ReLU. Fuente.

Softmax

Esta última función es utilizada en casos con varias neuronas de salida para obtener una clasificación multiclase. Provoca que la salida sea una distribución de probabilidad donde la salida de todas las neuronas suma 1.

3.2 Redes Recurrentes

Visto todo esto, centrémonos en redes más específicas. Puesto que un archivo de audio se define como una secuencia de bytes, nuestro problema requiere el análisis de una secuencia de

frecuencias. Por tanto, el modelo a utilizar para resolverlo debería estar especializado en secuencias.

Vamos a centrar nuestro trabajo en las redes neuronales recurrentes (RNN), que precisamente son las que están especializadas en el análisis de secuencias. Este tipo de modelos están estructurados de forma que la capa recurrente utiliza información de cada resultado para obtener el siguiente. De este modo, la secuencia se analiza por fragmentos y nosotros solo tenemos que entrenar una capa [10].

Así podemos analizar las secuencias de una forma natural y no importa el tamaño de la secuencia, porque la arquitectura no limita las unidades de la secuencia (X_t) de la entrada y por tanto las secuencias pueden tener cualquier longitud.

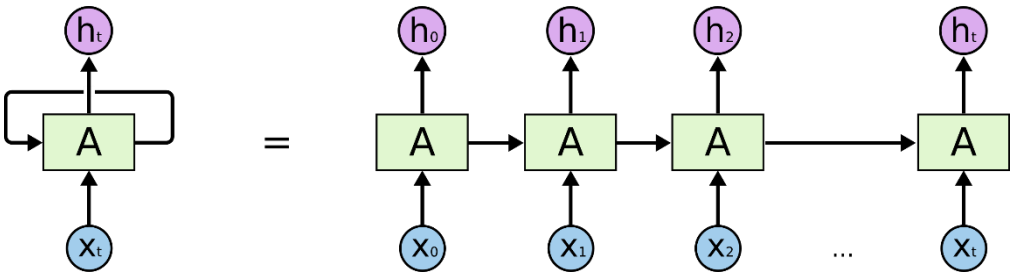

Figura 3-6: Diagrama de RNN resumido (izquierda), diagrama de RNN desenrollado (derecha). Fuente.

El bloque A de la figura anterior representa una capa recurrente y en el diagrama resumido (izquierda) podemos ver como conecta consigo misma. Hoy en día, hay varios modelos de RNN que suelen utilizarse, pero nosotros nos vamos a centrar en los tres más conocidos. Cada uno de estos tiene una estructura interna diferente y por tanto un funcionamiento diferente.

SimpleRNN

El primer tipo de RNN que vamos a ver es el más básico, apodado Simple RNN. Consiste en una única activación, normalmente tanh, que utiliza de entrada la salida del paso anterior y la entrada de la secuencia actual.

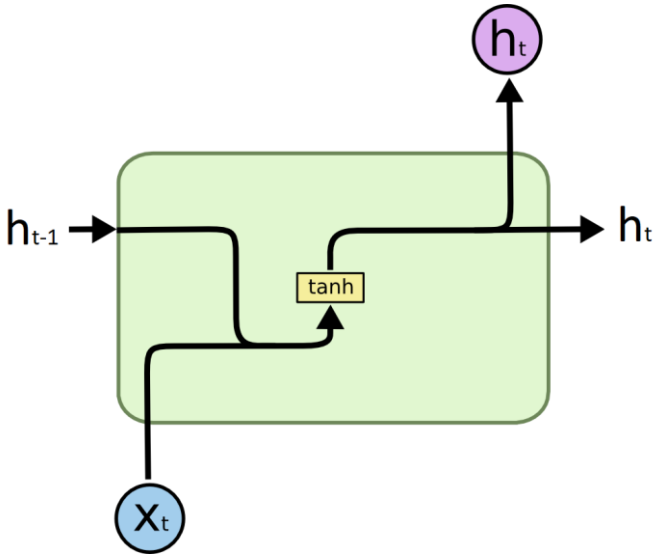

Figura 3-7: Representación de una capa SimpleRNN. Fuente.

En la Figura 3-7 podemos ver la estructura que conlleva este tipo de capa, donde podemos observar más en detalle lo que el bloque A mencionado en el apartado anterior sería. La flecha que sale por la derecha conectaría con la entrada de la izquierda, de modo que en el instante t+1 el resultado de t se empleara junto con X_{t+1} para calcular el nuevo h_{t+1}. Esto lo representamos en forma de ecuaciones para ver cómo se calculan los resultados y los pasos intermedios en todo momento.

En el caso de la SimpleRNN tenemos que la representación en ecuaciones sería la misma que para las redes neuronales estándares, puesto que no hemos añadido nada más por ahora. Tenemos que introducir la notación para las RNN que utilizaremos de ahora en adelante:

- W: es la matriz de pesos que se aplica para ponderar la suma.
- b: es el vector de pesos del bias.
- X_t: es la entrada en el instante t de la secuencia.
- h_t: es la salida en el instante t de la secuencia.

Visto esto, la ecuación antes mencionada queda de la siguiente forma (como este tipo de capa no es complejo, no son necesarias más de una ecuación) [11]:

$$h_t = tanh(W \cdot [h_{t-1}, X_t] + b)$$

Ecuación 3-5: Cálculo directo de la salida.

LSTM

La Long Short-Term Memory, o LSTM, es una capa compleja que introdujo varias novedades a las redes recurrentes [12]. Es el tipo de capa recurrente más pesada que vamos a utilizar, pero lo más importante a mencionar sobre ella es que introdujo el concepto de memoria.

La LSTM hace uso de una matriz de valores, llamados celdas de memoria. Estas son capaces de retener la información aprendida conforme va avanzando la secuencia. De esta forma, permite a la red unir y analizar conceptos separados por largas distancias en la secuencia de entrada. A diferencia de la SimpleRNN, la cual a la larga se olvida de lo que ha ido pasando por la secuencia y se queda solo con los momentos más recientes.

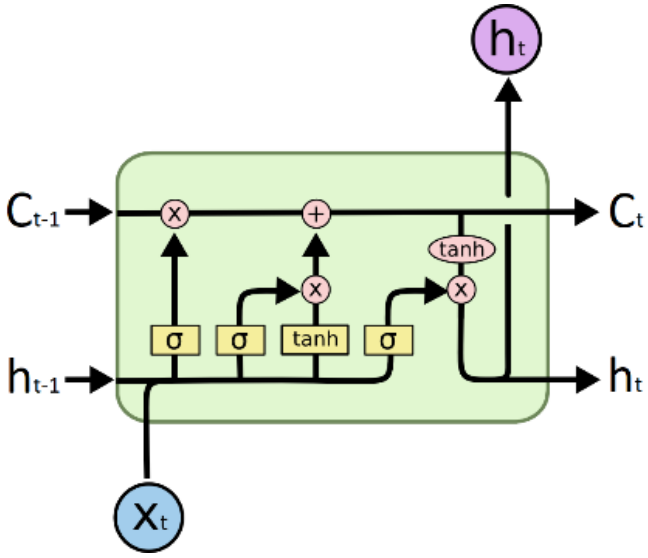

Figura 3-8: Representación de una LSTM. Fuente.

Como podemos ver en la representación anterior, tenemos como nuevo la matriz C_t, que es la línea superior. Vemos que se conserva la activación tanh igual que en la SimpleRNN, pero en este caso la salida de esta activación se trata diferente. Las funciones sigmoides se utilizan a modo de puerta permitiendo pasar solo parte de la información, ya que devuelven valores en el rango (0, 1), así podemos decidir qué partes recordar y qué partes olvidar.

$$f_t = \sigma\left(W_f \cdot [h_{t-1}, X_t] + b_f\right)$$

Ecuación 3-6: Puerta forget.

$$i_t = \sigma(W_i \cdot [h_{t-1}, X_t] + b_i)$$

Ecuación 3-7: Puerta input.

$$o_t = \sigma(W_o \cdot [h_{t-1}, X_t] + b_o)$$

Ecuación 3-8: Puerta output.

$$\hat{C}_t = tanh(W_C \cdot [h_{t-1}, X_t] + b_C)$$

Ecuación 3-9: Cálculo de la mueva memoria no definitiva.

Con esto tenemos las tres sigmoides que podemos ver en la figura anterior (en orden) y la tanh. La puerta de forget y la de input se combinan para crear la memoria usando la memoria anterior y la nueva memoria calculada con la tanh. Y, por último, calculamos la salida filtrando la memoria definitiva obtenida con la puerta de salida.

$$C_t = f_t \cdot C_{t-1} + i_t \cdot \hat{C}_t$$

Ecuación 3-10: Cálculo de la nueva memoria definitiva.

$$h_t = o_t \cdot tanh(C_t)$$

Ecuación 3-11: Cálculo de la salida.

GRU

Por último, la Gated Recurrent Unit, o GRU, está basada en la LSTM pero es una propuesta algo más simple pero igualmente prometedora [13].

La GRU también tiene un mecanismo similar a la LSTM para memorizar información. Sin embargo, esta vez no disponemos de celdas de memoria como tal, sino que utiliza la propia salida para memorizar en todo momento, permitiendo en cada paso recordar u olvidar algunos valores o parte de ellos.

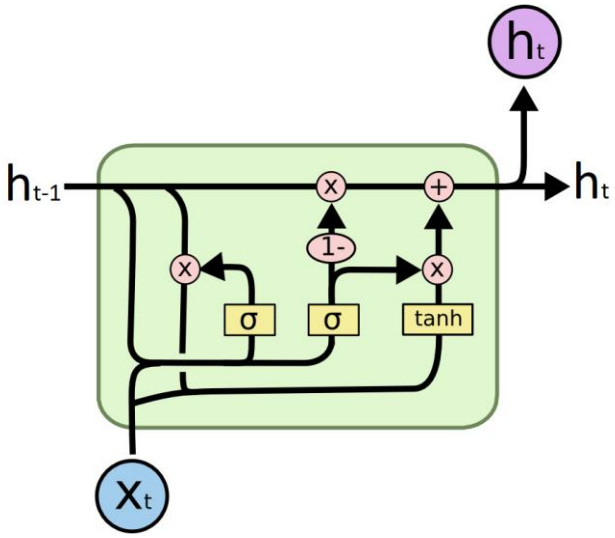

Figura 3-9: Representación de una GRU. Fuente.

A primera vista, podemos apreciar que aquí no hay tantas puertas sigmoide ni activaciones tanh. En este caso tenemos dos puertas, que actuaran de forma parecida a las de la LSTM.

$$r_t = \sigma(W_r \cdot [h_{t-1}, X_t] + b_r)$$

Ecuación 3-12: Puerta reset.

$$u_t = \sigma(W_u \cdot [h_{t-1}, X_t] + b_u)$$

Ecuación 3-13: Puerta update.

Una vez tenemos ambas puertas (en orden, también), calculamos la salida combinándolas de forma que la puerta de reset actúe sobre la salida del momento t-1 antes de pasar por la activación tanh, y la puerta de update actúe sobre la combinación de la salida de t-1 y el

resultado de la tanh. El resultado de la mencionada tanh funciona como la memoria no definitiva que veíamos en la LSTM.

$$\hat{\mathrm{h}}_t = tanh(W_h \cdot [r_t \cdot h_{t-1}, X_t] + b_h)$$

Ecuación 3-14: Cálculo de la salida no definitiva

$$h_t = u_t \cdot \hat{\mathrm{h}}_t + (1 - u_t) \cdot h_{t-1}$$

Ecuación 3-15: Cálculo de la salida definitiva

3.3 Herramientas

Para implementar este proyecto, hemos decidido utilizar Python como lenguaje principal, y, de hecho, como único lenguaje, ya que hay librerías suficientes para poder implementar todo lo que nos hace falta. En concreto haremos uso de la versión 3.6 de Python, que es, de las últimas versiones, la que nos ofrece la mayor compatibilidad con las librerías a utilizar. Cabe mencionar también, que haremos uso de su versión x64 puesto que la versión x32 tiene limitación de memoria RAM a 4GB y en más de una ocasión vamos a necesitar superar ese límite.

Para programar las tareas más relativamente sencillas, haremos uso de las siguientes librerías:

- Numpy, para el manejo de estructuras de datos no básicas (listas, vectores, matrices) de una forma liviana [14].
- Pandas, para el manejo de estructuras de datos complejas (DataFrame), y poder guardarlas en ficheros compatibles con Excel (CSV) [15].
- Matplotlib.pyplot, para dibujar gráficos y poder guardarlos además de verlos [16].
- PyDub, para el manejo general de los archivos de audio [17].
- WAVE, para la obtención de datos en crudo de los archivos de audio [18].
- PyGame, para la reproducción de los clips a la hora de etiquetarlos [19].

Para la implementación de las redes neuronales desde un alto nivel, vamos a hacer uso de la librería Keras [20]. Pero esta no puede funcionar por sí misma, sino que actúa como interfaz entre otra librería de más bajo nivel y el usuario permitiéndole generar modelos muy complejos en pocas líneas de código y por tanto en poco tiempo. Estas librerías de bajo nivel pueden ser TensorFlow, CNTK o Theano, cada una de las cuales tiene su particular funcionamiento. Pero Keras no varía su interfaz para el usuario en función de la librería que haya por debajo, por lo que nosotros no debemos preocuparnos por cual usar. Usaremos TensorFlow, que es la librería de código abierto desarrollada por Google Brain exclusivamente para el manejo de redes neuronales.

Para la ejecución de todo esto, usaremos Anaconda, como gestor de entornos virtuales, para las ejecuciones locales (portátil Lenovo i7, Nvidia GTX 960M, 8GB RAM). Además, trae ya instalados Keras, TensorFlow y algunas de las librerías mencionadas antes para las tareas menores.

Sin embargo, habrá ejecuciones que por mayor comodidad (y porque el equipo local no sea capaz) las haremos mediante Google Colaboratory. Mediante su versión de los Notebooks de Jupyter nos deja a nuestra disposición equipos más especializados para tareas de Deep Learning (más RAM, GPUs, etc.).

4 Creación de una Base de Datos

4.1 Búsqueda preliminar

En primer lugar, tenemos que buscar una base de datos de música que se adapte bien a nuestro problema. Es decir, necesitamos música variada, de diversos géneros, estilos, instrumentos, etc. En la web disponemos de algún que otro dataset, ya preprocesado, pero con etiquetas que no nos sirven. Viendo los artículos que hay respecto al tema de nuestro proyecto podemos ver que los datasets son escasos. Todos los artículos, al menos los que tratan la música como audio, trabajan sobre las mismas bases de datos.

La más importante al respecto, por el número de citas, es la llamada *Emotions from Mulan* [2], [21], que contiene canciones de 7 géneros diferentes (clásica, reggae, Rock, pop, hip-hop, techno, y jazz). Para crear esta base de datos se recopilaron canciones de 233 álbumes diferentes, cogiendo tres canciones de cada uno de ellos. Tras esto se eliminaron los 30 segundos iniciales de cada canción y se trocearon en fragmentos de 30 segundos. Se guardaron en archivos de 16-bit a 22050 Hz y en un único canal (mono).

De aquí extrajeron las características de cada archivo, creando el dataset de los datos de entrada. Extrajeron características rítmicas mediante el uso de los golpes por minuto (BPM) y características tonales mediante el uso de Mel Frequency Cepstral Coefficients (MFCC) y de la transformada de Fourier (Short-Term Fourier Transform, FFT), obteniendo unos pocos valores de cada uno de estos métodos. Se pueden obtener más detalles en [2].

Respecto al el etiquetado de los datos, este no se hizo sobre ningún modelo dimensional de los que mencionábamos en el capítulo 2. En este caso se utilizaron seis etiquetas.

Puesto que el preprocesado de este dataset es demasiado pesado, resume mucho la información para nuestro objetivo, y las etiquetas planteadas no son las que estamos buscando, decidimos que la mejor opción parece crear nuestra propia base de datos y extraer el dataset desde ella como creamos conveniente.

4.2 Búsqueda de música

Planteamos entonces la creación de nuestra base de datos desde cero. Pero con esto nos topamos con un pequeño problema: el copyright. Como necesitaremos descargar los archivos y conservarlos en disco al menos durante lo que dure el proyecto, necesitamos obtener dichos archivos de forma limpia. No podemos usar Spotify, ni ninguna plataforma por el estilo. Es decir, necesitamos música de descarga gratuita y que no nos de problemas con los derechos de autor. Vamos a intentar hacer uso de canciones únicamente con permiso para usarlas, por lo que necesitamos una amplia colección de música sin copyright.

Buscando por la web música libre de copyright, encontramos a Kevin MacLeod, un compositor y productor con una larga carrera cuya música comparte de forma gratuita en su web personal [22]. En mi caso, yo ya conocía a este autor por otros proyectos personales que realicé hace tiempo. Es un autor de renombre, que incluso tiene ficha en IMDb [23] porque su música ha sido utilizada en diversas películas y cortometrajes.

A la hora de hacer uso de su trabajo, él únicamente pide reconocimiento. Puesto que nosotros vamos a utilizar una gran cantidad de sus canciones, casi en su totalidad, no podemos citar todas y cada una de las canciones utilizadas, al menos citamos su web personal. En cualquier caso, queremos agradecerle que cuando le contactáramos respondiera que apoyaba el proyecto y esperaba los resultados con mucho entusiasmo.

Dicho esto, disponemos de más de 1300 canciones de diversos géneros, estilos, instrumentación, y demás características que podemos utilizar. Así generamos un amplio abanico de posibilidades para crear nuestra base de datos.

4.3 Creación de la Base de Datos

Una vez tenemos las canciones en nuestro PC, procedemos a crear scripts en Python que nos ayuden a ir creando la base de datos.

Nosotros vamos a utilizar una longitud de clip de 10 segundos. Para ello primero vamos a hacer mediante un script (*lister.py*) un listado de los clips que podemos hacer del total de canciones, con su respectiva información (archivo, origen y fin). Lo guardaremos en un archivo CSV para poder utilizar la información más adelante. Obtenemos un total de 23125 clips posibles.

El siguiente paso a realizar es la selección de algunos de los fragmentos (*selector.py*). Como somos nosotros quienes vamos a etiquetar manualmente, dicha cantidad de clips es absurdamente alta. Por tanto, seleccionaremos 1000 fragmentos al azar y esos serán los que etiquetemos.

Después, vamos a recortar los clips seleccionados (*slicer.py*). Para ello haremos uso de la librería PyDub, que entre otras cosas permite seleccionar partes del archivo y guardarlo como uno nuevo. Este paso es algo más pesado que los demás, por el hecho de que el guardar a disco es más costoso. Guardaremos los fragmentos utilizando de nombre para el archivo el índice que el selector les haya dado y rellenando con ceros a la izquierda hasta llegar a cinco dígitos (ej. 00354.mp3).

Una vez llegados a este punto, es hora de preocuparnos por el formato de audio. Las canciones de MacLeod están en MP3 originariamente. Nosotros para hacer uso de la librería WAVE necesitaremos que estén en formato WAV. Pero, por otra parte, hemos detectado que no todas las canciones tienen el mismo ratio de muestreo (sample rate). Los ratios habituales son 44100 Hz y 48000 Hz, y en las canciones que tenemos vemos que predomina el primero respecto al segundo. Vamos a hacer uso de un programa de terceros, llamado FormatFactory [24], para convertir todos los archivos resultantes a WAV con un ratio de muestreo de 44100 Hz.

Con todo esto, ya podemos ponernos a clasificar cada fragmento (*classifier.py*). Para ello vamos a utilizar la librería matplotlib.pyplot, de modo que dibujaremos los cuadrantes de nuestro modelo de emociones. Iremos escuchando cada canción (con la librería PyGame) y clickaremos donde queramos poner la etiqueta. Al clickar el punto, el script guardará en forma de coordenadas (x: placer, y: energía) y empezará a escucharse la siguiente canción. Aquí generaremos el CSV final con toda la información lista para crear el dataset. Las coordenadas se codificarán en el rango (-1, 1).

4.4 El Dataset

Para crear un conjunto de datos final a utilizar en una red neuronal necesitamos elegir una codificación apropiada. No vamos a utilizar la onda de audio tal y como está en el fichero, porque no parece natural. Son demasiadas muestras por segundo y un único valor de entrada por instante de tiempo parece algo escaso.

Como comentábamos en el Capítulo 2, el oído humano separa las frecuencias de la onda justo antes de enviar los impulsos nerviosos al cerebro. Por esto, parece natural el uso de técnicas como la transformada de Fourier para codificar nuestros datos de entrada. Y nosotros utilizaremos el espectrograma que viene implementado en la librería matplotlib.

Es espectrograma es una visualización de qué frecuencias están activas en una onda sonora para cada momento. Cogen un color más vivo las frecuencias más intensas y uno más apagado las que en ese momento no forman parte de la onda.

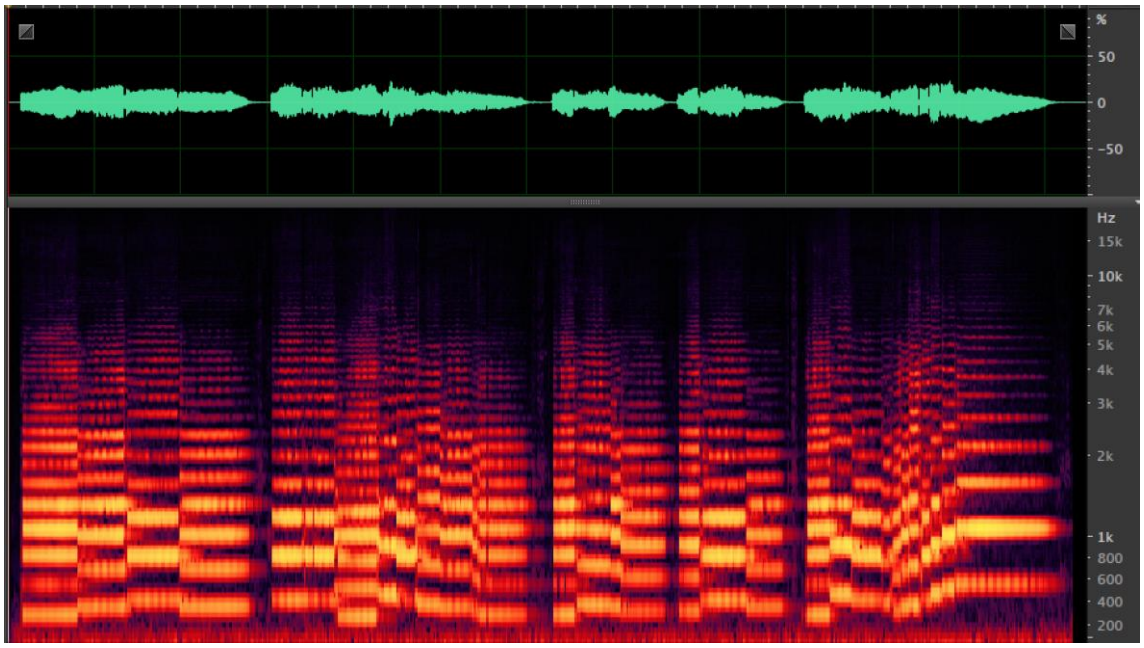

Figura 4-1: Señal de amplitud (arriba) y espectrograma generado (abajo) obtenidos en Adobe Audition. Fuente.

Como podemos apreciar en la Figura 4-1, el espectrograma ofrece mucha más información que la onda en crudo. Y en espectrogramas de música, como este, podemos ver que el tono va variando. Dicho de otra forma, con el espectrograma somos capaces de ver que hay movimiento en la melodía y cómo es este movimiento. Nosotros utilizaremos la función de matplotlib specgram para obtener el espectrograma.

Para acabar, tenemos un último script que genera nuestro conjunto de datos final (*dataset.py*). En él generamos un espectrograma para cada clip en formato WAV, a los que llamaremos ejemplos. Obtenemos la información para crear el espectrograma con WAVE y le pasamos esa información a la función de matplotlib. Los espectrogramas obtenidos son matrices de tamaño

timesteps x frecuencias, en nuestro caso 1721 x 129. Originalmente eran más valores en la dimensión de timesteps, pero decidimos coger uno de cada dos para reducir a la mitad todos los datos, y con ello el peso en disco y el tiempo de procesamiento. A la vez que los espectrogramas, que se guardan en *X.npy*, generamos los datos de salida utilizando los valores de energía y placer que hemos guardado al clasificar, que guardamos en *Y.npy*. Por si lo necesitamos más adelante, también hemos guardado en *Y_tag.npy* las etiquetas que corresponden a cada cuadrante.

4.5 Re-etiquetado

Con esto ya tenemos el dataset inicial completo. Sin embargo, en el Capítulo 5 veremos que el dataset podría mejorarse y conseguir que sea algo menos problemático para clasificar. Hemos realizado tres re-etiquetados del dataset con este fin. Dando como resultado cuatro datasets diferentes que van mejorando del primero al cuarto.

En primer lugar, tenemos el dataset original. Recordemos que el eje vertical representa la energía y el eje horizontal representa el placer.

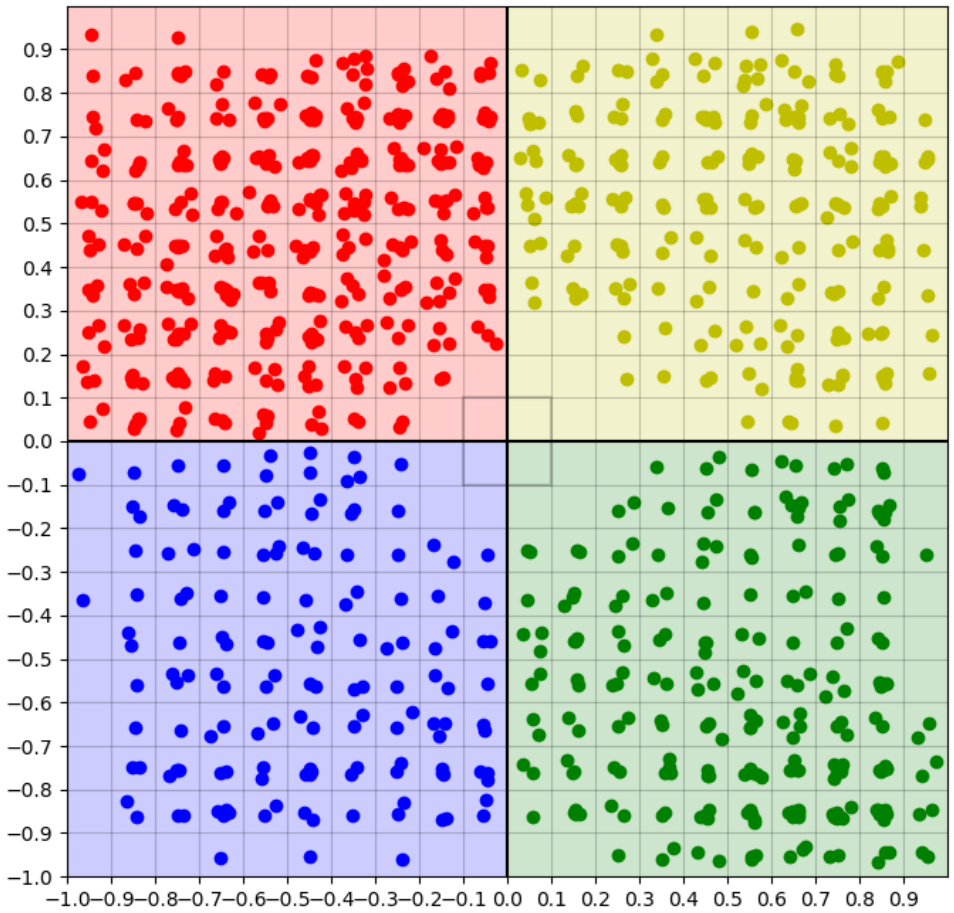

Figura 4-2: Primera versión del dataset

Tras detectar que el dataset debía ser mejorado re-etiquetamos los datos que más estábamos fallando en el eje de la energía, un total de 105 ejemplos. Después del cambio podemos apreciar diferencias visuales en la distribución de los puntos. Podemos observar cómo algunos puntos se han desplazado de arriba hacia abajo. Del cuadrante de Stress (en rojo) parecen haber bajado algunos al de Sad (en azul); y del cuadrante de Happy (en amarillo) parecen haber bajado al de Calm (en verde).

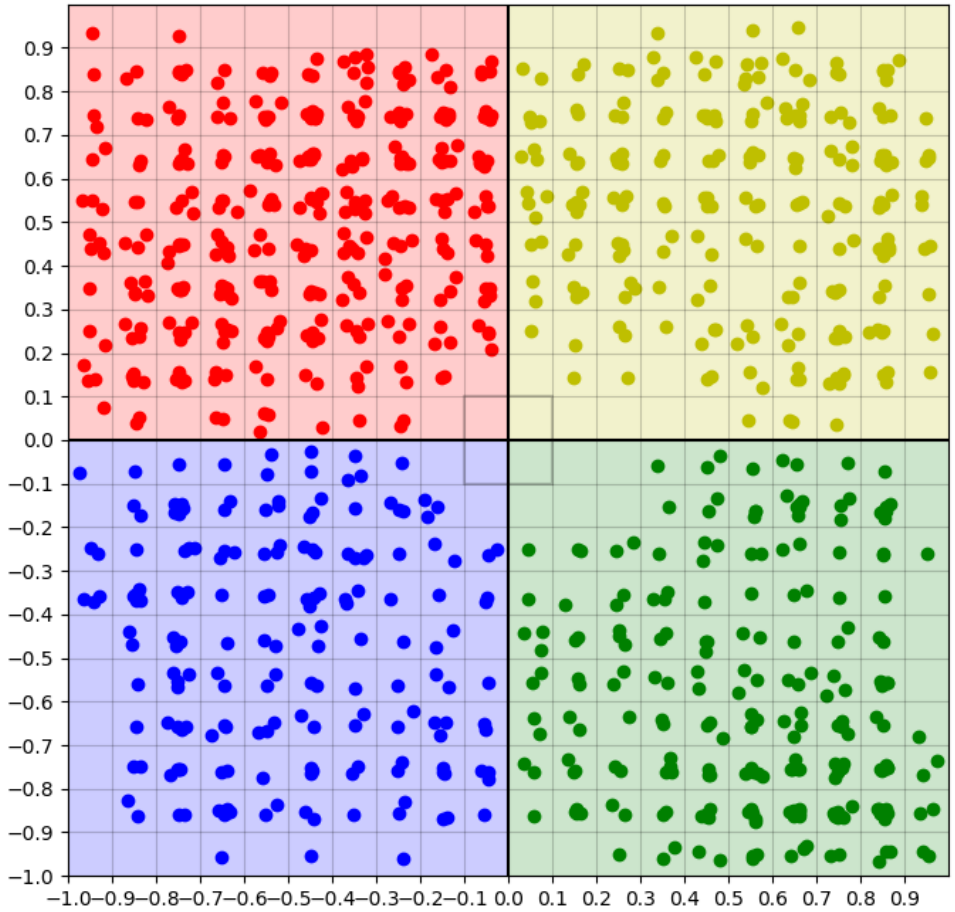

Figura 4-3: Segunda versión del dataset

Luego llega el turno del segundo eje, el eje del placer. Esta vez tuvimos que re-etiquetar 196 ejemplos, siguiendo el mismo criterio de los fallos frecuentes. Observamos que en este caso tenemos un desplazamiento de los puntos de los cuadrantes de la derecha a los de la izquierda, aunque es más notorio en los cuadrantes de energía negativa.

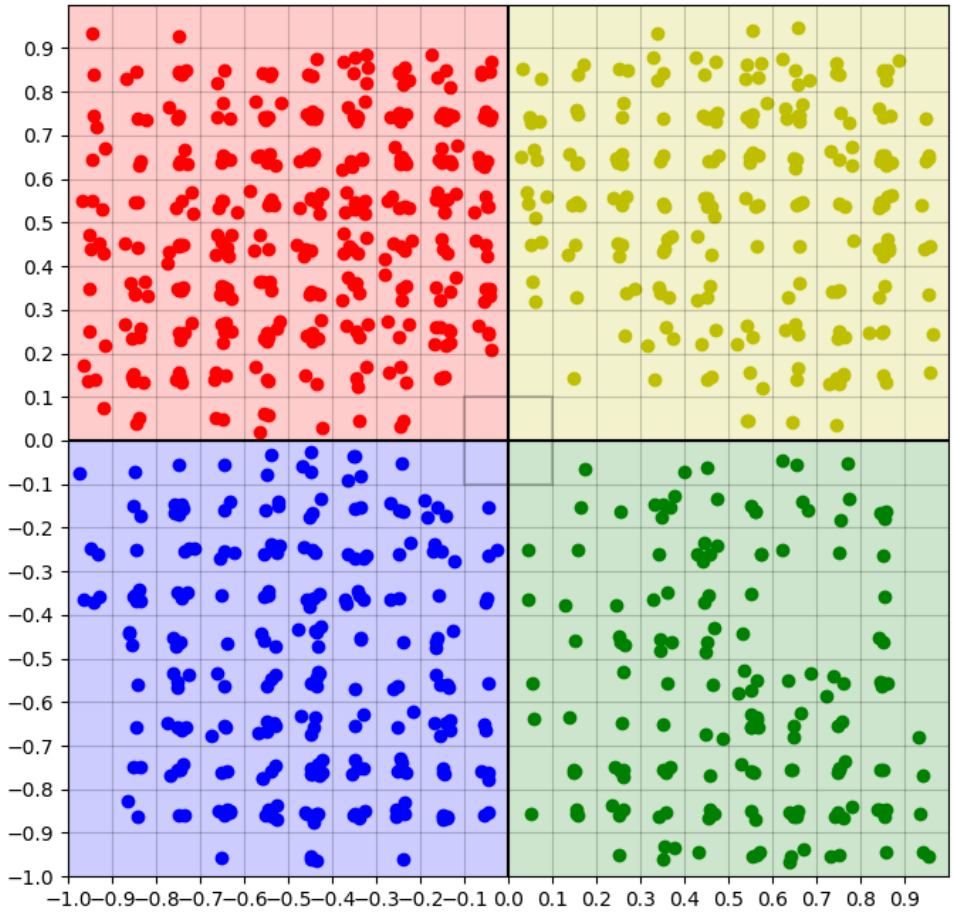

Figura 4-4: Tercera versión del dataset

Por último, ya que el segundo eje sigue dando problemas, hemos ideado una forma de re-etiquetarlo de manera que la subjetividad quede mucho más limitada que en las veces anteriores. Y esta vez lo aplicamos a todo el dataset, para que sea homogéneo en el estilo de la clasificación.

El eje del placer indica lo agradable que resulta una melodía o una armonía al oído del oyente. Dentro de la música, esto está estudiado y podemos saber que los acordes mayores son alegres, mientras que los menores son tristes y que las disonancias nos resultan desagradables. No vamos a entrar en detalle a este tema, pero es necesario comentarlo ya que siguiendo estas normas el dataset cambia drásticamente.

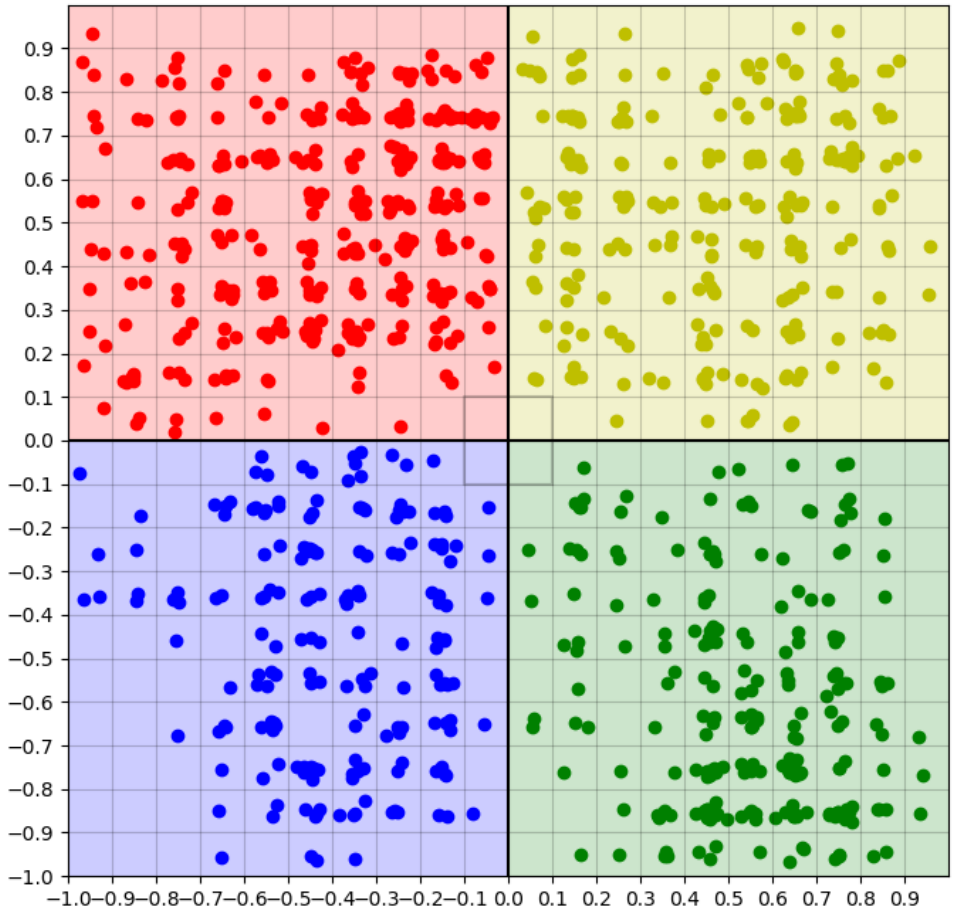

Figura 4-5: Cuarta versión del dataset

Tras este último re-etiquetado, podemos apreciar unos enormes cambios en la distribución de los puntos. Sin embargo, si nos paramos a pensarlo, esta distribución parece mucho más natural que las anteriores. En una base de datos limitada a 1000 clips, de un único autor, pero con géneros y estilos variados, lo lógico y natural seria que haya huecos en distintas zonas del plano y cúmulos en otras partes. Estos huecos y cúmulos podríamos interpretarlos como el estilo de composición del autor que tenemos, o simplemente como la música que representa sentimientos parecidos entre sí, cosa que nos beneficia ampliamente.

5 Sistema de Análisis de Sentimientos basado en Deep Learning

En este capítulo trataremos las pruebas realizadas y sus respectivos resultados. Para ello tendremos que ver antes qué arquitecturas vamos a crear y cómo, pasando por las opciones que Keras pone a nuestro alcance para la implementación de dichas arquitecturas.

5.1 Exploración de los parámetros

Keras ofrece un amplio rango de posibilidades a la hora de implementar modelos de redes neuronales. Incluye en su librería gran cantidad de tipos de capas, tanto simples como complejas. Incluye gran cantidad de funciones de activación, de coste, métricas, reguladores y demás implementaciones que pueden resultar de gran ayuda.

A continuación presentamos las opciones que Keras pone a nuestro alcance, las cuales pueden encontrarse junto con más información en la web de Keras [20]:

Funciones de activación

En Keras tenemos, entre otras, las funciones de activación vistas en el Capítulo 3. Pero para nuestro caso no haremos uso de la ReLU, de modo que únicamente usaremos la sigmoide, la tanh y la softmax.

La ReLU no es necesaria porque se recomienda usar en las capas ocultas y nosotros vamos a usar como capas ocultas las recurrentes, que tienen como función de activación recomendada la tanh. Respecto a la capa de salida, utilizaremos la sigmoide o la softmax dependiendo si la clasificación debe ser binaria (sigmoide) o multiclase (softmax).

Optimizadores

Para que la red neuronal aprenda necesitamos un método de optimización. En Keras tenemos diferentes optimizadores implementados, pero si miramos en su web en la sección de optimizadores, vemos que sugieren el uso del RMSprop (Root-Mean-Square) para redes recurrentes. Esto se debe a que, a la hora de utilizar el gradiente para ajustar los pesos de los parámetros, este método va haciendo una media al vuelo del gradiente al cuadrado y la utiliza para dividir el gradiente por la raíz de tal media y así actualizar los pesos de una forma más sutil [25].

En un instante t, para los parámetros W, cuyo gradiente es dW tenemos la siguiente ecuación:

$$MeanSquare(W,t) = \rho \cdot MeanSquare(W,t-1) + (1-\rho) \cdot dW^2$$

Ecuación 5-1: Cálculo de la media cuadrada de la matriz de pesos.

El valor de rho (ρ) más común es ρ = 0.9. Podemos elegirlo nosotros, pero ese es el valor por defecto en Keras y el recomendado en [25]. Con ρ = 0.9 tenemos un buen balance entre la media que estamos procesando y el valor a añadir a la misma.

Con esto, solo quedaría dividir el gradiente por la raíz de MeanSquare (Ecuación 5-1) y ya tendríamos el suavizado necesario para que el optimizador funcione mejor. Lo mismo haríamos para los parámetros de bias, utilizando su gradiente (db) para el cálculo de MeanSquare.

$$dW_{suavizado} = \frac{dW}{\sqrt{MeanSquare(W,t)}}$$

Ecuación 5-2: Obtención del suavizado para los pesos.

$$db_{suavizado} = \frac{db}{\sqrt{MeanSquare(b,t)}}$$

Ecuación 5-3: Obtención del suavizado para el bias.

El suavizado lo aplicamos en la actualización de pesos del mismo modo que se aplica el gradiente original.

$$W_{nuevo} = W - \alpha \cdot dW_{suavizado}$$

Ecuación 5-4: Actualización de los pesos.

$$b_{nuevo} = b - \alpha \cdot db_{suavizado}$$

Ecuación 5-5: Actualización del bias.

Los optimizadores tienen un parámetro clave, el learning rate o ratio de aprendizaje (α). Mediante este parámetro vamos a ser capaces de darle una determinada velocidad al aprendizaje de forma que se mueva más rápido o más lento hacia la mejor solución. El reto está en encontrar la velocidad que mejor haga converger a la solución, ni muy rápido ni muy lento.

Funciones de coste

Las funciones de coste son utilizadas por el optimizador para calcular el gradiente y poder así buscar la mejor solución. Son funciones en las que buscamos mínimos siempre, de modo que, si necesitamos maximizar una función, tendremos que utilizar como coste su inversa.

Dichas funciones de coste son clave para la resolución de la tarea ya que la propia función es la que define el problema. Dicho de otra forma, la función de coste es la que va a imponer las condiciones para la búsqueda de la solución. De modo que es vital elegir la función correcta siempre que queramos entrenar un modelo de redes neuronales.

En Keras tenemos gran diversidad de funciones. Aunque nosotros, utilizaremos las más típicas que se emplean para clasificación:

- Binary crossentropy: para clasificación binaria.

$$Loss(y,h) = -\big(y \cdot log(h) + (1-y) \cdot log(1-h)\big)$$

- Categorical crossentropy: para clasificación multiclase.

$$Loss(y,h) = -\sum_{c=1}^{C} y_c \cdot log(h_c)$$

Métricas

Para utilizar como métricas podemos hacer uso de cualquier función de coste. A parte de esto Keras nos provee de otras funciones que pueden ser interesantes para ver el progreso de nuestro modelo. En nuestro caso, únicamente veremos las dos funciones de precisión: Binary accuracy para clasificación binaria y Categorical accuracy para clasificación multiclase. Simplemente consisten en contar los aciertos de cada clase y ver en total cuantos hay respecto al número de ejemplos. En el caso de la clasificación binaria pueden utilizarse cualquiera de las dos funciones citadas, pero la Binary accuracy está optimizada para este caso.

Métodos de Regularización

El optimizador puede provocar situaciones en las que el modelo se ajusta demasiado a los datos de entrenamiento provocando un empeoramiento general con datos fuera de este conjunto. En ese caso decimos que el modelo tiene sobreaprendizaje, u overfitting, el cual lo asociamos a un sesgo, o bias, notable en el modelo.

Con esto en mente, aparece la regularización. Es una forma de hacer que nuestro modelo no llegue a conclusiones muy complejas permitiendo generalizar lo aprendido a los datos externos de forma más sencilla. El problema es que, si la regularización es demasiado fuerte, puede provocar el efecto contrario, impidiendo que el modelo llegue a aprender nada del conjunto de entrenamiento, lo que conocemos como underfitting. Keras implementa la regularización dentro de cada capa de forma que únicamente necesita que le digamos qué función de penalización queremos que use.

Nosotros no haremos uso de esta funcionalidad puesto que a día de hoy la regularización ha dejado de aplicarse de este modo. Hoy en día para este mismo fin principalmente se utilizan las técnicas de Batch Normalization y de Dropout, que funcionan mucho mejor en la práctica.

A la hora de entrenar el modelo, es mucho más rápido ir por bloques, en vez de ir ejemplo a ejemplo. A cada uno de estos bloques les llamamos batch, lote en inglés. Sabiendo esto, la técnica del Batch Normalization simplemente consiste en hacer la normalización estadística para los datos de cada batch. De esta forma cuantos más ejemplos tengamos en el batch, mayor será su efecto.

La técnica de Dropout consiste en seleccionar un porcentaje (parámetro rate) de neuronas de una capa e ignorarlas. Esto únicamente se hace a la hora de entrenar el modelo. Internamente hablando, lo que hacemos es multiplicar los pesos de las seleccionadas por cero, de modo que no aparezcan en la entrada de la siguiente capa. Es una manera de simplificar el modelo, obligándolo además a tener cierta redundancia. De esta manera, forzamos al modelo a que no confíe plenamente en lo que diga una neurona por separado, necesitando la información de varias para concluir sus resultados, y con ello evitar el sobreaprendizaje.

Capas

Junto con todo esto, en Keras también tenemos las implementaciones de los diferentes tipos de capas que utilizaremos:

- Dense: es la capa básica de una red neuronal, conecta todas las entradas con todas las neuronas que tiene y devuelve el resultado de estas. A veces también se le llama Fully Connected, pero en Keras aparece como Dense. Parámetros: Units, el número de neuronas; activation, la función de activación que necesitemos.
- SimpleRNN: implementación de la RNN más sencilla, la Simple RNN. Parámetros: units, el número de neuronas.
- LSTM: implementación de la Long Short-Term Memory. Mismo parámetro que la SimpleRNN.
- GRU: implementación de la Gated Recurrent Unit. Mismo parámetro que la SimpleRNN y la LSTM.

Las capas tienen más parámetros para su uso general, pero nosotros no vamos a necesitar prácticamente ningún otro. A excepción del parámetro return sequences de las capas recurrentes, que nos va a permitir concatenar las capas recurrentes.

Las capas recurrentes en Keras están implementadas de forma que por defecto devuelven un único valor al terminar la secuencia, por lo que, si queremos poner dos capas recurrentes seguidas, tenemos que activar el retorno de secuencias (return sequences) en todas las capas anteriores a la última, para que devuelvan un valor por timestep.

En primer lugar, consideramos el problema al completo, mediante una clasificación multiclase, e hicimos varias pruebas para hacernos una idea de cómo reaccionaba cada parámetro. En dichas pruebas pudimos elegir algunos valores para los citados parámetros que parecían mejorar el aprendizaje de los modelos. Sin embargo, los resultados obtenidos están lejos de ser buenos. Por esta razón acabamos decidiendo que sería mejor tratar el problema analizando cada eje por separado, pudiendo ver así si tenemos la misma dificultad en ambos o el problema se origina en únicamente un eje.

Dicho lo anterior, hemos decidido que los parámetros a utilizar van a ser los siguientes, ya que parecen ser valores que en nuestro caso funcionan de forma estable en el entrenamiento:

- Learning rate = {0.002, 0.003}. Por lo general 0.002 pero si queremos hacer alguna prueba concreta y acelerarla un poco 0.003.
- Dropout rate = {0.3, 0.5, 0.8}. empezaremos con 0.5 y según cómo reaccionen los modelos lo moveremos.
- Batch size = {32, 48}. No tenemos datos como para coger batches más grandes y más pequeños no ofrecen tan buenos resultados.
- Units = {32, 64, 128}. Meteremos por lo general más neuronas en la primera capa. Usaremos una o dos capas mayormente, más de eso es muy costos de entrenar.

- Epochs (iteraciones en el entrenamiento) = 50. A las 50 iteraciones los modelos ya han bajado mucho el loss y se han ralentizado hace rato.

En las citadas pruebas hemos observado que el dataset presenta dificultades para ser clasificado correctamente, por lo que procederemos a analizar cada eje por separado. Es decir, primero trataremos de clasificar las canciones como energéticas o no (eje de Energía), y luego pasaremos a tratar de clasificar si nos resultan agradables o no (eje de Placer).

5.2 Creación de arquitecturas

Así pues, procedemos a crear un script que nos facilite implementar las arquitecturas que necesitemos (architectures.py). De este modo, con llamar a una función del script tendríamos el modelo ya preparado para entrenar.

El script creará arquitecturas de la siguiente forma:

- BatchNormalization
- bloques
    - Capa de RNN
    - BatchNormalization
    - Dropout
- Dense

De este modo, podremos decidir el número de capas recurrentes que queremos meter indicando cuántas unidades tendrá cada capa. El script generará tantas capas como valores tenga el vector de unidades. También podremos decidir qué tipo de capa recurrente usará el modelo, pero todas deberán ser del mismo tipo. Será necesario indicarle al script el tamaño de la entrada, en nuestro caso, el número de frecuencias que contienen los espectrogramas.

La Dense del final tendrá tantas neuronas como salidas necesite el modelo. Para analizar un único eje, usaremos una única neurona con activación sigmoide (clasificación binaria), y para clasificación multiclase, pondremos cuatro neuronas con activación softmax.

También seremos capaces de decidir el dropout rate que utilizaran todas las capas Dropout, pero en este caso también deben ser siempre el mismo para todas las capas.

Como último apunte sobre la creación automática de modelos, este script generará el nombre del modelo en base al tipo de capa recurrente que esté usando, al número de capas que vaya a generarle, al número de salidas que le incorpore y al dropout rate que le asigne.

5.3 Pruebas en el eje de Energía

De los dos ejes propuestos, el eje de Energía es que da la impresión de ser más sencillo. La energía es al fin y al cabo la intensidad con la que sentimos el tipo de sentimiento transmitido mediante el otro eje. Por eso, parece lógico que su clasificación resulte más sencilla que la del eje de Placer. Empezaremos pues a tratar el eje de Energía en primer lugar.

Con todo esto, preparamos unas pruebas automáticas (locales) que combinen los parámetros que le indicamos con varios valores cada uno. Por ahora solo vamos a intentar clasificar los ejemplos según el eje de Energía. Dividiremos los 1000 ejemplos del dataset en 950 para train y 50 para test. Es decir, entrenaremos los modelos con 950 ejemplos y lo evaluaremos con 50.

Junto con los tests hemos implementado un contador de tiempo para saber cuánto tarda cada modelo y tenerlo en cuenta a la hora de evaluarlos. Hemos fijado algunos parámetros de la siguiente forma: learning rate = 0.002, dropout = 0.5, epochs = 50. *Nota: bs = batch size.*

test	name	units	bs	training_time	train_loss	train_acc	test_loss	test_acc
3	GRU_l1_o1_d0.5	[32]	32	23m 50s	0.3869	0.8179	0.2929	0.9
9	GRU_l1_o1_d0.5	[64]	32	27m 10s	0.3523	0.8463	0.3007	0.9
17	LSTM_l2_o1_d0.5	[64, 32]	48	55m 38s	0.3528	0.8274	0.3528	0.86
5	LSTM_l1_o1_d0.5	[32]	48	23m 06s	0.4161	0.7811	0.3812	0.84
6	GRU_l1_o1_d0.5	[32]	48	21m 04s	0.355	0.84	0.3852	0.84
15	GRU_l2_o1_d0.5	[64, 32]	32	48m 48s	0.3825	0.8379	0.4043	0.84
24	GRU_l2_o1_d0.5	[128, 64]	48	03h 23m 33s	0.224	0.8663	0.5096	0.84
10	SRNN_l1_o1_d0.5	[64]	48	13m 51s	0.3945	0.8105	0.4204	0.82
12	GRU_l1_o1_d0.5	[64]	48	26m 44s	0.2224	0.8989	0.5752	0.82
22	SRNN_l2_o1_d0.5	[128, 64]	48	51m 19s	0.5928	0.7284	0.4235	0.82
23	LSTM_l2_o1_d0.5	[128, 64]	48	04h 45m 50s	0.2731	0.8842	0.5127	0.82
4	SRNN_l1_o1_d0.5	[32]	48	11m 40s	0.4475	0.7811	0.4542	0.8
8	LSTM_l1_o1_d0.5	[64]	32	30m 20s	0.3649	0.8242	0.4185	0.8
18	GRU_l2_o1_d0.5	[64, 32]	48	50m 53s	0.3547	0.84	0.3542	0.8
19	SRNN_l2_o1_d0.5	[128, 64]	32	45m 28s	0.7177	0.6611	0.5442	0.76
20	LSTM_l2_o1_d0.5	[128, 64]	32	01h 56m 59s	0.3099	0.8463	0.4633	0.76
13	SRNN_l2_o1_d0.5	[64, 32]	32	25m 56s	1.0043	0.6537	0.8514	0.74
2	LSTM_l1_o1_d0.5	[32]	32	27m 21s	0.4472	0.7032	0.4285	0.72
21	GRU_l2_o1_d0.5	[128, 64]	32	01h 41m 56s	0.3222	0.84	0.7686	0.72
7	SRNN_l1_o1_d0.5	[64]	32	14m 30s	0.5344	0.6926	0.5187	0.7
16	SRNN_l2_o1_d0.5	[64, 32]	48	24m 41s	0.4552	0.7579	0.5196	0.7
14	LSTM_l2_o1_d0.5	[64, 32]	32	57m 41s	0.4022	0.7789	0.5769	0.66
1	SRNN_l1_o1_d0.5	[32]	32	12m 38s	0.6417	0.6547	0.6447	0.64
11	LSTM_l1_o1_d0.5	[64]	48	29m 29s	0.4402	0.7032	0.6891	0.6

Tabla 5-1: Resultados de la primera tanda de pruebas ordenados por test_acc.

En la Tabla 5-1 observamos que las capas con métodos de memoria implementada funcionan mejor. Es decir, la LSTM y la GRU por lo general tienen mejores resultados que la SimpleRNN. Podemos ver también, que los mejores resultados los hemos obtenido con arquitecturas más simples generalmente. Vamos a quedarnos, entonces, con los cuatro modelos que mejor funcionan, que deberían ser los cuatro primeros de la tabla. Sin embargo, vemos también que el cuarto puesto está empatado en test_acc, por lo que debemos elegir uno de ellos. Elegiremos el test número 6, porque en train_acc tiene un valor superior al resto. Podríamos elegir el test

24 en su lugar, que es todavía mejor, pero las más de tres horas de entrenamiento, no compensan. Por tanto, tenemos que los mejores modelos son los de los tests 3 (GRU [32]), 9 (GRU [64]), 17 (LSTM [64, 32]) y 15 (GRU [64, 32]). Estos test tienen arquitecturas de GRU y LSTM con un número bastante reducido de unidades.

Vistos estos resultados, decidimos intentar aumentar la precisión en test. Para esto, vamos a utilizar los cuatro modelos que mejor funcionan, mencionados anteriormente, para no tener que volver a probar con todas las arquitecturas y ahorrar así algo de tiempo. Planteamos la posibilidad de que el etiquetado sea incorrecto y la red neuronal clasifique este eje mejorando el dataset. Esto bajaría su precisión, pero realmente es bueno que ocurra. Para comprobarlo vamos a re-etiquetar parte del eje de Energía, sin tocar los valores de y ver si notamos mejoría.

Procedemos al primer re-etiquetado del dataset (véase Capítulo 4) en el que cambiaremos los valores de energía en los ejemplos que al menos tres de los cuatro mejores tests fallen al clasificarlos, que son un total de 105 clips. Mientras re-etiquetamos el dataset vemos que muchos de los ejemplos que la red neuronal esta "fallando" en realidad son ejemplos que nosotros teníamos mal clasificados, por lo que podemos ver claramente que la red está aprendiendo a clasificar el eje de la energía mejor que nosotros mismos.

Volvemos a probar el dataset repitiendo únicamente los cuatro tests antes mencionados. Los tests generan un nuevo identificador, pero tienen la misma arquitectura que antes. Esta vez en vez de dividir el dataset en 950/50 para train/test, lo dividiremos de forma que queden 900 en el conjunto de train y 100 en el de test. *Nota: dr = dropout rate.*

test	name	units	dr	bs	training_time	train_loss	train_acc	test_loss	test_acc
25	GRU_l1_o1_d0.5	[64]	0.5	32	28m 06s	0.132	0.95	0.441	0.8
26	GRU_l1_o1_d0.5	[32]	0.5	32	23m 30s	0.2806	0.8867	0.3515	0.86
27	LSTM_l2_o1_d0.5	[64, 32]	0.5	48	52m 36s	0.7356	0.7011	0.7854	0.7
28	GRU_l2_o1_d0.5	[64, 32]	0.5	32	50m 38s	0.1636	0.9367	0.4377	0.84

Tabla 5-2: Resultados de la segunda tanda de pruebas.

Podemos ver que los resultados empeoran en test, pero mejoran en train por lo que deducimos que tenemos un caso de overfitting. Por eso, planteamos aumentar la regularización, concretamente, vamos a aumentar el dropout rate, de 0.5 a 0.8.

test	name	units	dr	bs	training_time	train_loss	train_acc	test_loss	test_acc
29	GRU_l1_o1_d0.8	[64]	0.8	32	26m 22s	0.3474	0.8356	0.4195	0.82
30	GRU_l1_o1_d0.8	[32]	0.8	32	23m 08s	0.2991	0.8678	0.328	0.89
31	LSTM_l2_o1_d0.8	[64, 32]	0.8	48	50m 26s	0.413	0.8411	0.4044	0.86
32	GRU_l2_o1_d0.8	[64, 32]	0.8	32	49m 19s	0.286	0.8933	0.4508	0.82

Tabla 5-3: Resultados de la tercera tanda de pruebas.

Esta vez observamos una mejora general, ya que las precisiones de train y de test se ven más equilibradas. Tenemos en el test 30 (arquitectura del test 3) una precisión en test del 89% por lo que podemos decir que es un buen resultado, teniendo en cuenta la subjetividad implícita en el problema.

Los resultados obtenidos después de haber aumentado el conjunto de test no son directamente comparables a los obtenidos antes de esto. Si bien podemos hacernos una idea, tras el cambio el conjunto de train ha disminuido y el conjunto de test ha aumentado, y con ello ha cambiado ligeramente la forma en la que aprenden los modelos.

A continuación, vamos a ver los gráficos generados por dicho test (30). Tenemos un descenso del loss (Figura 5-1) que se estabiliza sobre la iteración (epoch) 50 y que sigue bajando lentamente. Y los gráficos de la clasificación en train (Figura 5-2) y test (Figura 5-3) nos muestran como la gran mayoría de fallos se concentran en la zona cercana a la frontera, los extremos los clasifica bien en general. Tenemos en rojo los clasificados como energéticos y en verde los que no.

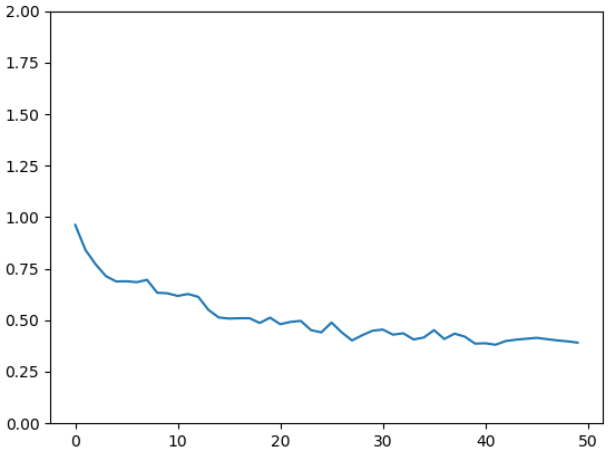

Figura 5-1: Test 30. Bajada del loss durante el entrenamiento.

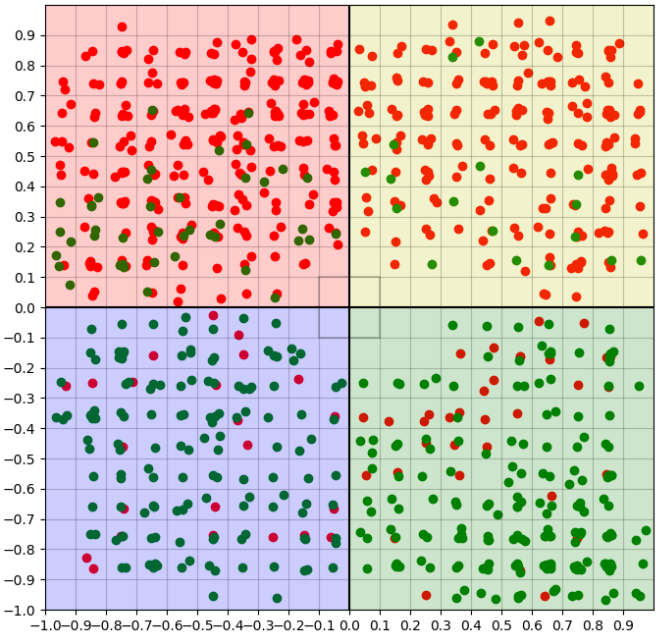

Figura 5-2: Test 30. Clasificación del conjunto de train.

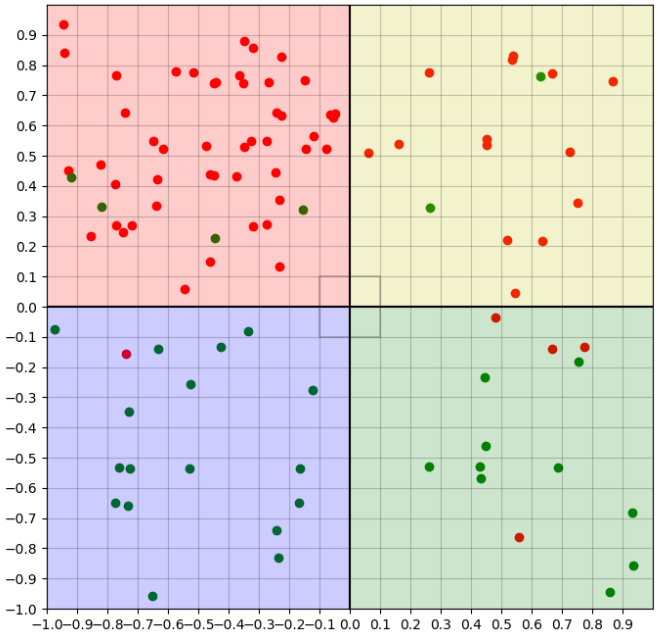

Figura 5-3: Test 30. Clasificación del conjunto de test.

Con todo esto, podemos concluir que tenemos un modelo capaz de clasificar la música según lo energética que sea. Por lo que tenemos la mitad de nuestro clasificador final terminado.

5.4 Pruebas en el eje de Placer

Ahora nos toca clasificar el segundo eje, el eje de Placer. El procedimiento va a ser el mismo que en el eje anterior, pero esta vez no vamos a probar las capas recurrentes simples (SimpleRNN). Esto se debe a que en el eje anterior hemos tenido un número relativamente alto de pruebas realizadas y algunas de ellas llevan unas cuantas horas de procesamiento. Quitando las SimpleRNN tenemos varias horas de pruebas menos, por lo que nuestro PC no tiene que trabajar tanto tiempo, y ya sabemos que estas no van a ser muy relevantes por su comportamiento en el primer eje.

Además, se espera que este eje sea más difícil. Así como el eje de Energía era un sentimiento primario, este eje actúa sobre las armonías de forma directa, por lo que podemos presuponer que es complejo.

Al igual que antes, dividimos el dataset en los conjuntos de entrenamiento (900 ejemplos) y de testeo (100 ejemplos). Hemos fijado algunos parámetros de la siguiente forma: learning rate = 0.002, dropout = 0.3, epochs = 50.

test	name	units	bs	training_time	train_loss	train_acc	test_loss	test_acc
48	GRU_l2_o1_d0.3	[128, 64]	48	02h 58m 37s	0.2427	0.8711	1.1488	0.59
36	GRU_l1_o1_d0.3	[32]	48	20m 08s	0.5025	0.69	0.832	0.58
39	LSTM_l1_o1_d0.3	[64]	48	27m 52s	0.4956	0.7278	1.1488	0.58
41	LSTM_l2_o1_d0.3	[64, 32]	32	56m 40s	0.3871	0.7744	1.0242	0.58
33	LSTM_l1_o1_d0.3	[32]	32	26m 48s	0.4659	0.7244	0.8745	0.57
44	GRU_l2_o1_d0.3	[64, 32]	48	48m 38s	0.4299	0.7311	0.9655	0.57
45	LSTM_l2_o1_d0.3	[128, 64]	32	01h 50m 41s	0.4901	0.7389	1.1746	0.57
34	GRU_l1_o1_d0.3	[32]	32	23m 27s	0.515	0.6933	0.8696	0.56
40	GRU_l1_o1_d0.3	[64]	48	25m 14s	0.4757	0.7222	1.0704	0.56
47	LSTM_l2_o1_d0.3	[128, 64]	48	04h 10m 14s	0.3032	0.8244	1.1148	0.56
46	GRU_l2_o1_d0.3	[128, 64]	32	01h 33m 31s	0.309	0.8289	1.3246	0.55
38	GRU_l1_o1_d0.3	[64]	32	26m 08s	0.4549	0.7433	0.9611	0.48
42	GRU_l2_o1_d0.3	[64, 32]	32	49m 54s	0.4243	0.7578	0.8785	0.47
35	LSTM_l1_o1_d0.3	[32]	48	22m 00s	0.4662	0.7244	1.0781	0.46
37	LSTM_l1_o1_d0.3	[64]	32	29m 13s	0.4078	0.7611	1.0753	0.46
43	LSTM_l2_o1_d0.3	[64, 32]	48	54m 03s	0.3973	0.7722	1.5087	0.45

Tabla 5-4: Resultados de la cuarta tanda de pruebas ordenados por test_acc.

En el caso de este eje, podemos observar claramente cómo los resultados son muchísimo peores, con una enorme diferencia respecto al eje anterior. Podemos ver como en train llega a conseguir extraer algún patrón, es decir, a aprender algo, pero muy poco. Y podemos ver como en test no hay avance alguno. Una precisión cercana al 50% es equivalente a clasificar a ciegas, por lo que afirmamos que son malos resultados.

Estos resultados eran esperados, puesto que este eje por definición ya implica una complejidad mucho más alta que el primer eje y con ello la necesidad de más datos para ser capaz de generalizar la información fuera del conjunto de entrenamiento.

Como nosotros no disponemos de un dataset mayor, vamos a intentar mejorar el dataset actual, procurando eliminar la subjetividad subyacente, únicamente teniendo en cuenta el eje actual.

Procedemos pues a re-etiquetar el dataset. En este caso los ejemplos que fallen las cuatro mejores arquitecturas. Estas arquitecturas van a ser las de los tests 36, 39, 41 y 44. La del test 48 no vamos a utilizarla porque tarda demasiado tiempo en entrenar y la del test 33 porque ofrece peores resultados en train que la del test 44.

En la Tabla 5-5 podemos ver que los resultados, aunque ligeramente mejores, no alcanzan a ser buenos resultados en test. Por esta razón, decidimos que necesitamos ver qué pasa durante el entrenamiento más en detalle.

test	name	units	dr	bs	training_time	train_loss	train_acc	test_loss	test_acc
49	LSTM_l2_o1_d0.3	[64, 32]	0.3	32	01h 13m 08s	0.3525	0.8411	1.1169	0.65
50	LSTM_l1_o1_d0.3	[64]	0.3	48	28m 34s	0.3135	0.8644	1.2256	0.53
51	GRU_l1_o1_d0.3	[32]	0.3	48	20m 47s	0.4443	0.78	0.93	0.61
52	GRU_l2_o1_d0.3	[64, 32]	0.3	48	45m 13s	0.3488	0.8422	0.9998	0.57

Tabla 5-5: Resultados de la quinta tanda de pruebas.

Con este fin, vamos a añadir al entrenamiento un conjunto de validación, abreviado como val o dev (development). Este conjunto se utilizará en cada época del entrenamiento para calcular el coste del modelo, pero no influirá en la corrección de los pesos, de modo que tenemos una forma de ver el comportamiento externo al conjunto de train durante ese entrenamiento. Este comportamiento se verá reflejado en los gráficos de coste (loss).

Así pues, el conjunto de entrenamiento queda reducido un 20%. De los 900 ejemplos que tenemos asignados para el entrenamiento, 720 ejemplos van para train y 180 para val, durante el entrenamiento. Para evaluar el modelo una vez este haya entrenado las épocas que hayamos determinado, los conjuntos de train y test seguirán siendo 900 y 100.

Con los nuevos conjuntos listos, probamos las cuatro mejores arquitecturas del segundo eje de nuevo. En la Tabla 5-6 podemos apreciar los resultados obtenidos.

test	name	units	dr	bs	training_time	train_loss	train_acc	test_loss	test_acc
53	LSTM_l2_o1_d0.3	[64, 32]	0.3	32	48m 42s	0.4432	0.8156	1.2345	0.6
54	LSTM_l1_o1_d0.3	[64]	0.3	48	23m 50s	0.4353	0.8211	1.4448	0.5
55	GRU_l1_o1_d0.3	[32]	0.3	48	17m 47s	0.5202	0.7378	1.0687	0.51
56	GRU_l2_o1_d0.3	[64, 32]	0.3	48	39m 06s	0.5706	0.6433	1.2233	0.44

Tabla 5-6: Resultados de la sexta tanda de pruebas.

Al añadir el conjunto de validación no podemos apreciar ninguna mejora, ya que siguen siendo los mismos datos en las mismas arquitecturas y con los mismos parámetros. En todo caso notamos un ligero empeoramiento, por haber reducido el tamaño del conjunto de entrenamiento.

Sin embargo, este nuevo conjunto de validación nos permite tener unos gráficos de coste (loss) más detallados y poder ver el comportamiento de los datos de validación conforme se va entrenando.

En la Figura 5-4 queda claro que los comportamientos de train y val no van acordes. Esto nos indica que el modelo no es capaz de generalizar lo aprendido, que encuentra patrones en train, pero no consigue generalizarlos fuera de este conjunto.

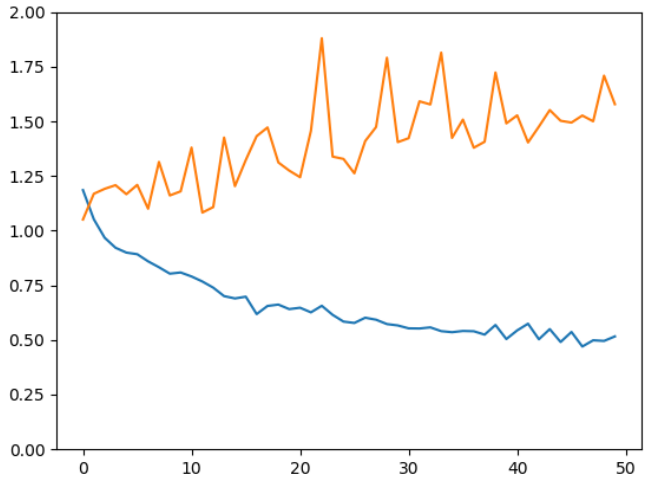

Figura 5-4: Test 53. Gráfico de loss. Train en azul, val en naranja.

Llegados a este punto, podemos decir que el dataset en el eje de Placer no puede clasificarse con las arquitecturas que proponemos. La causa más importante de esto es la falta de datos, únicamente tenemos 1000 clips etiquetados. La posibilidad de que el dataset no sea coherente en algunos ejemplos y eso cause una inconsistencia general consideramos que es alta, por lo que los 1000 ejemplos que tenemos podrían no estar rindiendo como es debido.

Por tanto, vamos a proponer un último re-etiquetado, que vemos más en detalle en el Capítulo 4. Este re-etiquetado afecta a todo el dataset, a los 1000 ejemplos, y esperamos que elimine la subjetividad de una forma mucho más drástica que la anterior.

En esta última tabla (Tabla 5-7) podemos observar cómo los resultados de las cuatro arquitecturas de los tests 36, 39, 41 y 44 mencionadas al principio de esta sección, no consiguen aprender, a pesar de haber probado diferentes valores de dropout rate.

test	name	units	dr	bs	training_time	train_loss	train_acc	test_loss	test_acc
57	LSTM_l2_o1_d0.5	[64, 32]	0.5	32	48m 39s	0.4719	0.7944	1.2387	0.64
58	LSTM_l1_o1_d0.5	[64]	0.5	48	23m 45s	0.5082	0.8011	1.2891	0.64
59	GRU_l1_o1_d0.5	[32]	0.5	48	17m 40s	0.5166	0.7533	0.991	0.55
60	GRU_l2_o1_d0.5	[64, 32]	0.5	48	39m 12s	0.7196	0.7244	1.2043	0.63
61	LSTM_l2_o1_d0.8	[64, 32]	0.8	32	50m 08s	0.5622	0.7189	0.8384	0.54
62	LSTM_l1_o1_d0.8	[64]	0.8	48	25m 00s	0.509	0.7756	1.3614	0.54
63	GRU_l1_o1_d0.8	[32]	0.8	48	17m 53s	0.6498	0.5444	1.0398	0.33
64	GRU_l2_o1_d0.8	[64, 32]	0.8	48	39m 56s	0.6183	0.6867	0.9315	0.48
65	LSTM_l2_o1_d0.3	[64, 32]	0.3	32	56m 30s	0.5322	0.7056	0.9296	0.5
66	LSTM_l1_o1_d0.3	[64]	0.3	48	26m 04s	0.5338	0.7322	1.1605	0.46
67	GRU_l1_o1_d0.3	[32]	0.3	48	20m 01s	0.5805	0.6589	0.9879	0.47
68	GRU_l2_o1_d0.3	[64, 32]	0.3	48	43m 49s	0.6093	0.6689	1.2229	0.5

Tabla 5-7: Resultados de la séptima tanda de pruebas.

Tras todos estos resultados, podemos concluir que el segundo eje resulta muy complejo para nuestras arquitecturas recurrentes. Complejo a la hora de generalizar la información aprendida, puesto que en train hemos llegado a conseguir un 80% de acierto, que, pese a no ser ideal, no es mal resultado.

5.5 Convolucional1D en RNN

Con el fin de mejorar los resultados en la clasificación del segundo eje, hemos decidido incluir en nuestros modelos recurrentes una primera capa convolucional en una dimensión que preceda a todas las capas que estábamos utilizando anteriormente, a modo de capa de entrada.

Las redes convolucionales, originalmente diseñadas para tratamiento de imagen, son redes que aplican filtros de tamaño a especificar a matrices de datos de más de dos dimensiones. Los filtros se componen parámetros que la red aprende del mismo modo que los pesos de las redes clásicas o las recurrentes que hemos visto, pero el número de operaciones es generalmente muy elevado en las capas de convolución. A pesar de necesitar menos parámetros, el elevado número de operaciones hace a las redes convolucionales pesadas en ejecución.

Para nuestro caso, como vamos a tratar una secuencia y no una imagen, vamos a utilizar la convolución 1D, es decir, en vez de tratar matrices bidimensionales con varios canales, tratará matrices unidimensionales con varios canales. Concretamente, la dimensión será la dimensión temporal por la que avanza la secuencia y los canales serán las frecuencias que tenemos en cada timestep. Vamos a utilizar, por tanto, un número de filtros igual al número de frecuencias que tratamos, para que la salida de la convolución mantenga el tamaño de entrada, y un tamaño de filtro de 32 unidades, para que tenga en cuenta 32 timesteps en cada convolución.

El resto de los parámetros siguen siendo igual que en las pruebas anteriores. Seguimos fijando el número de epochs a 50. En este caso, hemos probado la mejor arquitectura para cada eje.

En la Tabla 5-8 podemos apreciar los resultados de estas pruebas. Los tests 69 y 70 son del eje de Placer, mientras que los tests 71 y 72 son del eje de Energía. El primer test para cada eje ha sido realizado sin la mencionada capa de convolución 1D, mientras que los siguientes sí que la incluyen. Queda decir que el dropout rate y el batch size para el eje de Placer los hemos reducido con la esperanza de evitar todo el sobreaprendizaje posible y aumentado ligeramente el learning rate. *Nota: ax = eje, conv = convolucional 1D.*

test	name	units	ax	conv	lr	dr	bs	training_time	train_loss	train_acc	test_loss	test_acc
69	GRU_l1_o1_d0.3	[64]	2	no	0.003	0.3	16	39m 28s	0.5363	0.7089	0.8722	0.48
70	GRU_l1_o1_d0.3	[64]	2	yes	0.003	0.3	16	01h 27m 50s	0.3685	0.8678	1.7659	0.37
71	GRU_l1_o1_d0.8	[32]	1	no	0.002	0.8	32	23m 37s	0.33	0.8633	0.3052	0.88
72	GRU_l1_o1_d0.8	[32]	1	yes	0.002	0.8	32	01h 12m 51s	0.2115	0.9222	0.4812	0.76

Tabla 5-8: Resultados de la octava tanda de pruebas.

Si bien en la tabla de resultados no somos capaces de apreciar gran diferencia entre los modelos con convolucional y los modelos sin ella, a excepción de un aumento de overfitting en ambos ejes (mejora en train, pero empeora en test), podemos ver cambios en el comportamiento del coste de validación y del coste de entrenamiento mediante las Figuras 5-5, 5-6, 5-7 y 5-8.

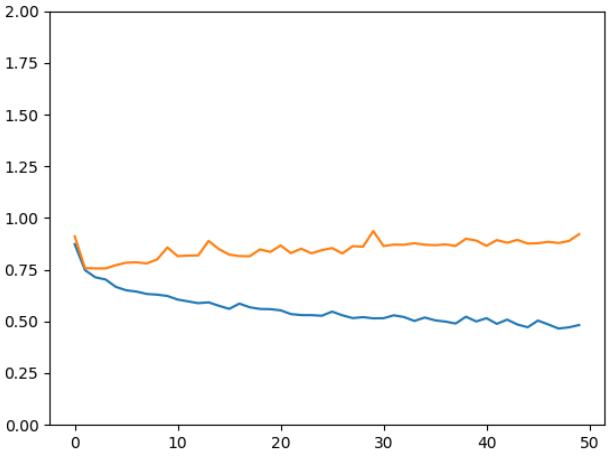

Figura 5-5: Test 69. Gráfico del loss. Eje de Placer.

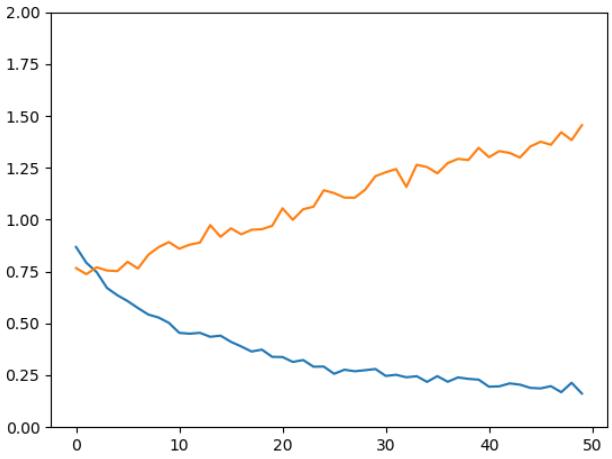

Figura 5-6: Test 70. Gráfico del loss. Eje de Placer con convolucional 1D.

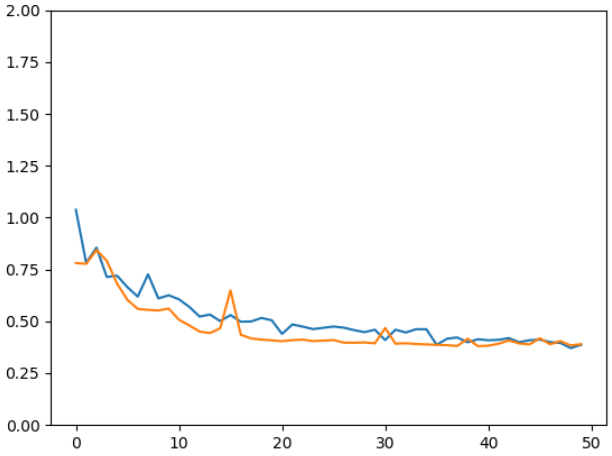

Figura 5-7: Test 71. Gráfico del loss. Eje de Energía.

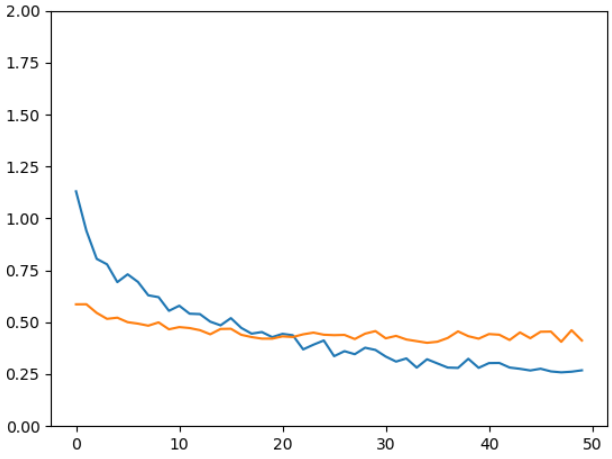

Figura 5-8: Test 72. Gráfico del loss. Eje de Energía con convolucional 1D.

Con todo esto, vemos que la adición de la capa convolucional añade un ligero overfitting en ambos ejes por lo que seguimos sin tener un buen clasificador para el eje de Placer y, además, tenemos como mejor alternativa para el eje de Energía el modelo obtenido sin añadir la capa convolucional.

Vamos a ver los gráficos de clasificación del test 70 para observar visualmente cómo clasifica el modelo, a pesar de no ofrecer buenos resultados, y así poder apreciar el sobreentrenamiento.

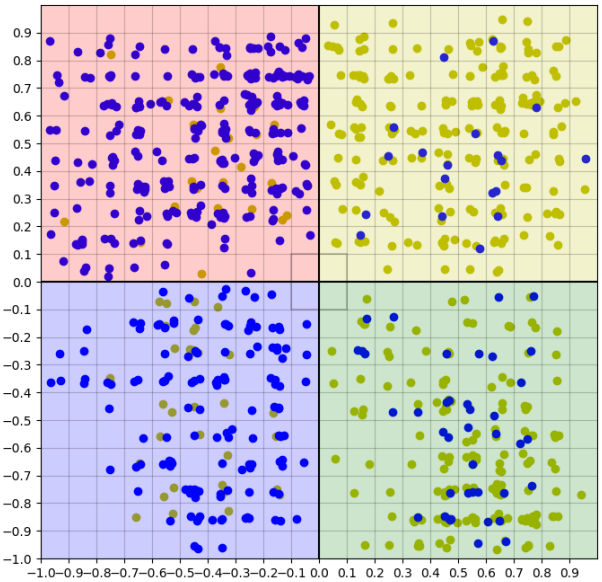

Figura 5-9: Test 70. Conjunto de train.

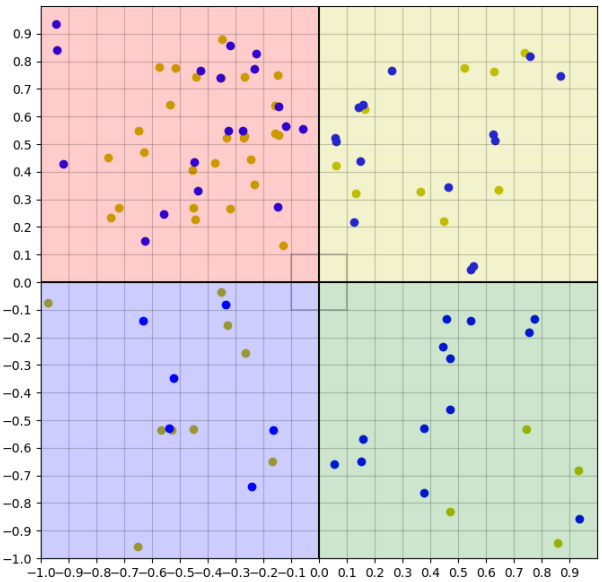

Figura 5-10: Test 70. Conjunto de test.

5.6 Convolucional2D en CNN

Ante los resultados obtenidos, sobre todo a causa del segundo eje, hemos decidido probar una red neuronal convolucional clásica clasificando directamente las cuatro clases, por si de este modo vemos que puede mejorar.

Las redes convolucionales, como ya hemos comentado antes, son mucho más pesadas de ejecutar que las recurrentes debido a su alto número de operaciones a realizar. Por tanto, no vamos a ejecutar estas en local. Para esta tarea vamos a movernos a Google Colaboratory.

Vamos a aplicar la red convolucional directamente sobre las matrices que tenemos en el dataset, que pueden interpretarse como imágenes sin color. Si bien lo ideal habría sido guardar los espectrogramas originales en formato PNG o similares, por cómo están codificados actualmente no creemos que esto suponga problemas. Nuestro dataset está compuesto por espectrogramas de dos dimensiones (tiempo x frecuencias).

En este punto, hay que introducir las capas de Conv2D, MaxPooling y Flatten de las que disponemos en Keras:

- Conv2D: es la implementación clásica de red convolucional, orientada mayormente a imagen, que necesita como entrada una matriz tridimensional de la forma anchura x altura x canales. En nuestro caso tenemos un único canal, porque en cada posición tenemos un único color. Puede compararse con las imágenes en escala de grises, realmente no nos importan los colores en RGB (Red-Blue-Green) ni ningún otro formato similar.
- MaxPooling: es la implementación del Pooling por máximos. Esta es una técnica para reducir el tamaño de una capa convolucional que coge varios valores adyacentes y los fusiona mediante una función, en este caso utiliza el máximo.
- Flatten: esta capa simplemente reduce todas las dimensiones a una, pasando los datos a vector para poder utilizarlo como entrada en la capa Dense.

Probamos una red convolucional con la arquitectura mostrada en la Figura 5-11 con Adam como optimizador. Adam es una versión avanzada del RMSprop en la que se añade otra técnica parecida llamada momentum, que mejora el rendimiento para algunas tareas.

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 1721, 129, 1)	0
conv2d_1 (Conv2D)	(None, 858, 127, 8)	176
max_pooling2d_1 (MaxPooling2	(None, 429, 63, 8)	0
conv2d_2 (Conv2D)	(None, 212, 61, 16)	2704
max_pooling2d_2 (MaxPooling2	(None, 106, 30, 16)	0
conv2d_3 (Conv2D)	(None, 50, 28, 32)	10784
max_pooling2d_3 (MaxPooling2	(None, 25, 14, 32)	0
conv2d_4 (Conv2D)	(None, 10, 12, 64)	43072
max_pooling2d_4 (MaxPooling2	(None, 5, 6, 64)	0
flatten_1 (Flatten)	(None, 1920)	0
dropout_1 (Dropout)	(None, 1920)	0
dense_1 (Dense)	(None, 4)	7684

Figura 5-11: Arquitectura del modelo convolucional en Keras. Visto desde Colaboratory.

Tras el entrenamiento durante 100 iteraciones de dicha red convolucional, obtenemos resultados similares a los que obtuvimos con las redes recurrentes. Es decir, seguimos sin poder generalizar la información. Queda destacar que al ejecutar el entrenamiento en Colaboratory accedemos a una maquina con una potencia muy superior a la que estamos utilizando. Por tanto, el tiempo de ejecución se disminuye notablemente y todavía más en el caso de las convolucionales, que respecto a las recurrentes son mucho más paralelizables y por tanto rápidas de ejecutar en ciertas maquinas (GPUs, TPUs).

test	name	optim.	training_time	epochs	train_loss	train_acc	test_loss	test_acc
73	Conv2D_o4	Adam	03m 46s	100	0.8217	0.6778	1.4182	0.48

Tabla 5-9: Resultados de la prueba de convolucional2D.

Podemos ver en las Figuras 5-12, 5-13 y 5-14 cómo en train consigue aprender algo (muy poco), pero no llega a ser capaz de generalizarlo para val ni para test. Aunque no son unos resultados buenos, podemos decir que está tecnología podría llegar a realizar bien esta tarea si tuviéramos más datos y después refináramos los parámetros debidamente, del mismo modo que con los modelos recurrentes.

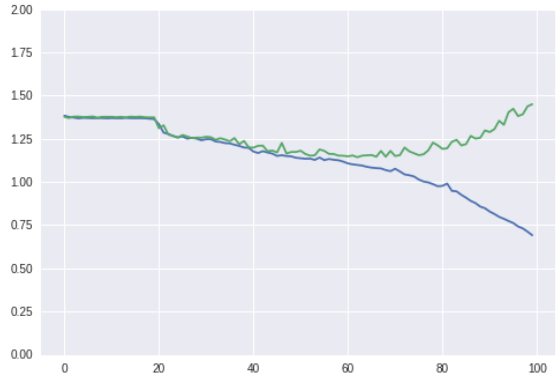

Figura 5-12: Prueba de convolucional2D. Loss de train en azul, loss de val en verde.

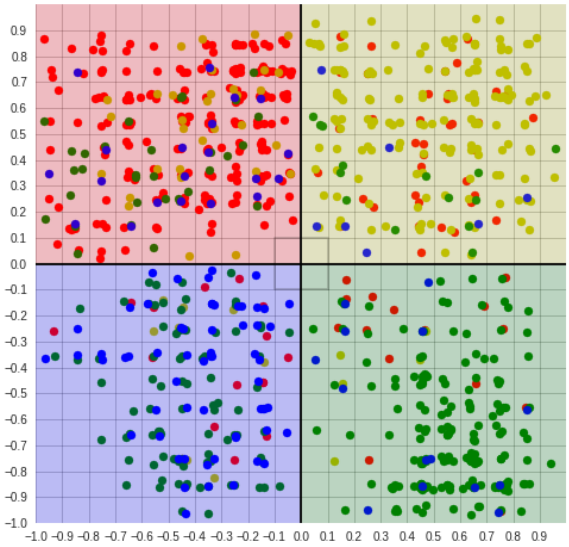

Figura 5-13: Prueba de convolucional2D. Conjunto de train.

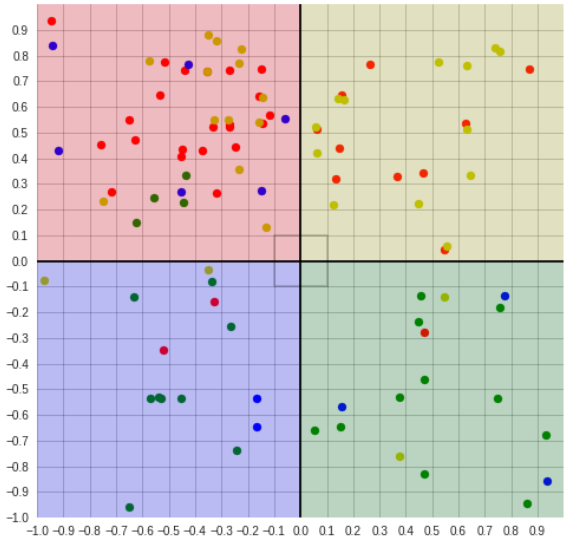

Figura 5-14: Prueba de convolucional2D. Conjunto de test.

6 Conclusiones y líneas futuras

Recapitulando, hemos sido capaces de conseguir un clasificador para analizar la energía que transmite la música y no hemos logrado el clasificador para analizar el nivel de confortabilidad que esta puede transmitir. Esto se debe a la complejidad inherente de cada una de estas partes. Podemos verlo de la siguiente manera: el nivel de energía es un sentimiento mucho más primario que el nivel de confortabilidad.

Esto podemos afrontarlo aumentando el número de datos sobre el que trabajamos y la calidad de los mismos. Hasta que esto no se consiga, la tarea será demasiado complicada para realizarla mediante los modelos que proponemos, los recurrentes.

Hemos observado que las redes neuronales con memoria implementada son muy eficaces en la práctica, en comparación con las que no tienen dicha memoria. Con la implementación de la memoria ganamos precisión a la hora de retener información durante más tiempo a lo largo de las secuencias de datos de entrada, por lo que ganamos productividad en la mayoría de las tareas.

Respecto al modelo convolucional, resulta una alternativa prometedora. Parece que, a pesar de ser más pesado a la hora del entrenamiento, este tipo de modelos tienen un potencial más elevado, puesto que con menos retoques de parámetros hemos conseguido unos resultados similares. Su utilidad, dependerá de si compensa su entrenamiento respecto a la ligereza de los modelos recurrentes aquí propuestos.

Mirando hacia el futuro, proponemos la creación de un nuevo y mejor dataset. Para ello sería conveniente tratar con varias escuelas de música o conservatorios y que ellos se encargaran del etiquetado en condiciones de un dataset mucho más grande que el nuestro. Pongamos como ejemplo, del orden de un millón de ejemplos.

Una buena práctica sería que cada ejemplo de ese gran dataset estuviese etiquetado varias veces, por personas diferentes, y haciendo una media de los valores obtenidos eliminaríamos el problema de la subjetividad en gran medida.

Los modelos recurrentes con memoria para analizar sentimientos en la música parecen prometedores, pese a que no hayamos logrado durante este proyecto resultados destacables. Somos capaces de ver que detectan patrones en el conjunto de entrenamiento, por lo que todo indica que si obtuviéramos un dataset grande bien etiquetado se conseguirían grandes avances mediante estos modelos. Todo apunta a que con más investigación en este campo se conseguirían grandes resultados.

Por último, proponemos continuar con la línea de investigación de las redes convolucionales orientadas a imagen (2D) para el análisis de sentimientos en música. Las redes neuronales convolucionales tienen un enorme potencial y son usadas habitualmente para la realización de muchas tareas. Nosotros hemos realizado una única prueba con estas para tener una ligera idea de si funcionarían y creemos que con algo de investigación también podrían obtenerse muy buenos resultados por esta vía.

Bibliografía

[1] F. Colace and L. Casaburi, “An Approach for Sentiment Classification of Music,” vol. 2, no. Iceis, pp. 421–426, 2016.

[2] K. Trohidis, G. Kalliris, G. Tsoumakas, and I. Vlahavas, “Multi-Label Classification of Music Into Emotions,” *ISMIR 2008 – Sess. 3a – Content-Based Retrieval, Categ. Similarity 1*, vol. 3a, pp. 325–330, 2008.

[3] L. M. Gómez and M. N. Cáceres, “Trends in Cyber-Physical Multi-Agent Systems. The PAAMS Collection - 15th International Conference, PAAMS 2017,” vol. 619, pp. 3–6, 2018.

[4] C. Pérez, “Sonido y audicion,” pp. 1–22, 2003.

[5] E. Çano and M. Morisio, “MoodyLyrics: A sentiment annotated lyrics dataset,” *ACM Int. Conf. Proceeding Ser.*, vol. Part F1278, pp. 118–124, 2017.

[6] R. E. Thayer, *The Biopsychology of Mood and Arousal.* Oxford University Press, 1990.

[7] J. A. Russell, “A circumplex model of affect.,” *J. Pers. Soc. Psychol.*, vol. 39, no. 6, pp. 1161–1178, 1980.

[8] F. Rosenblatt, “The perceptron: A probabilistic model for information storage and organization in the brain.,” *Psychol. Rev.*, vol. 65, no. 6, pp. 386–408, 1958.

[9] “Activation functions - Shallow neural networks | Coursera.” [Online]. Available: https://www.coursera.org/lecture/neural-networks-deep-learning/activation-functions-4dDC1. [Accessed: 10-Mar-2019].

[10] “Recurrent Neural Network Model - Recurrent Neural Networks | Coursera.” [Online]. Available: https://www.coursera.org/lecture/nlp-sequence-models/recurrent-neural-network-model-ftkzt. [Accessed: 10-Mar-2019].

[11] “Gated Recurrent Unit (GRU) - Recurrent Neural Networks | Coursera.” [Online]. Available: https://www.coursera.org/lecture/nlp-sequence-models/gated-recurrent-unit-gru-agZiL. [Accessed: 10-Mar-2019].

[12] S. Hochreiter and J. Schmidhuber, “Long Short-Term Memory,” *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997.

[13] K. Cho *et al.*, “Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation,” Jun. 2014.

[14] “NumPy Reference — NumPy v1.16 Manual.” [Online]. Available: https://docs.scipy.org/doc/numpy/reference/. [Accessed: 11-Mar-2019].

[15] “API Reference — pandas 0.24.1 documentation.” [Online]. Available: https://pandas.pydata.org/pandas-docs/stable/reference/index.html. [Accessed: 11-Mar-2019].

[16] “The Matplotlib API — Matplotlib 3.0.3 documentation.” [Online]. Available: https://matplotlib.org/api/index.html. [Accessed: 11-Mar-2019].

[17] “pydub/API.markdown at master · jiaaro/pydub · GitHub.” [Online]. Available: https://github.com/jiaaro/pydub/blob/master/API.markdown. [Accessed: 11-Mar-2019].

[18] “audiotools/spectrogram_matplotlib.py at master · cgoldberg/audiotools · GitHub.” [Online]. Available: https://github.com/cgoldberg/audiotools/blob/master/visualization/spectrogram_matplotlib.py. [Accessed: 11-Mar-2019].

[19] “pygame.mixer — Pygame v1.9.5.dev0 documentation.” [Online]. Available: https://www.pygame.org/docs/ref/mixer.html. [Accessed: 11-Mar-2019].

[20] “Home - Keras Documentation.” [Online]. Available: https://keras.io/. [Accessed: 11-Mar-2019].

[21] “multilabel datasets.” [Online]. Available: http://mulan.sourceforge.net/datasets-mlc.html. [Accessed: 12-Mar-2019].

[22] “incompetech – Music and also Graph Paper.” [Online]. Available: https://incompetech.com/. [Accessed: 12-Mar-2019].

[23] “Kevin MacLeod - IMDb.” [Online]. Available: https://www.imdb.com/name/nm1930572/. [Accessed: 12-Mar-2019].

[24] “Format Factory es un convertidor multifuncional de medios de comunicación.” [Online]. Available: http://www.pcfreetime.com/formatfactory/index.php. [Accessed: 13-Mar-2019].

[25] R. F. Lyon, “Neural Networks for Machine Learning,” *Hum. Mach. Hear.*, pp. 419–440, 2017.