Universidad de la República
Facultad de Ingeniería
Instituto de Computación

PROYECTO DE GRADO

Generación de música y letra utilizando aprendizaje profundo

Autores:
Santiago Pérez Toledo
Fernando Medina Suárez
Sergio Bonilla Morales

Tutores:
Dr. Guillermo Moncecchi
MSc. Diego Garat

Agosto, 2017
Montevideo, Uruguay

"Those who can imagine anything, can create the impossible."

Alan Turing

Resumen

Crear música es una forma de expresar sentimientos y otras emociones que se entiende solamente el humano es capaz de hacer. Si bien hay trabajos que intentan demostrar que una máquina puede componer música de forma automática, pocos lo hacen con música y letra en conjunto. Esto último es el objetivo de este proyecto.

Con el fin de generar música y letra, se construye un corpus que será fuente de conocimiento para un algoritmo de aprendizaje profundo, el cual se implementa utilizando redes neuronales LSTM de entre una y hasta nueve capas. También se usa un segundo corpus de referencia, de la universidad de Nottingham, como forma de comparación al anterior.

Se evalúa el algoritmo empíricamente mediante un Test de Turing Musical utilizando canciones generadas en base a los dos corpus. Considerando las canciones generadas por máquina, sólo el 50 % de las personas pudieron identificar que fueron compuestas automáticamente. Vistos estos resultados, se puede afirmar que aprendizaje profundo, y en particular las redes LSTM, parecen ser una buena estrategia para atacar la temática en cuestión.

Palabras clave: Generación de Música, Aprendizaje Automático, Aprendizaje Profundo, Test de Turing Musical.

Agradecimientos

Dedicamos este trabajo a nuestras familias y amigos que siempre nos apoyaron a lo largo del proyecto. Agradecemos a todas las personas que se tomaron el tiempo de participar en el Test de Turing Musical y en particular a nuestros tutores Diego Garat y Guillermo Moncecchi, que nos han guiado durante todo el proceso y nos han ayudado siempre con excelente disposición y fluidez.

Índice general

Capítulo 1

Introducción

La música es una de las expresiones artísticas más antiguas de la humanidad. Como tal desempeña un papel importante en la vida de los seres humanos [24]. Crear música, y arte en general, es una forma de expresar sensaciones, sentimientos, ideas y otras emociones que se entiende solamente el humano es capaz de hacer. Pero hoy en día, con el avance de muchas tecnologías y metodologías de aprendizaje automático, la creación de música podría ser una tarea no solamente del hombre. ¿Será capaz una máquina de componer música y letra? Esta pregunta es uno de los motivadores de este proyecto de grado.

En la actualidad, muchas de las grandes empresas de tecnología e investigación, así como varias universidades alrededor del mundo, están aplicando tecnologías en áreas tales como el procesamiento de imagen [14] y video [16], el procesamiento de lenguaje natural [6], reconocimiento facial [31], planificación de movimiento en robots [7], reconocimiento de voz [10], entre otras, obteniendo muy buenos resultados.

El objetivo del proyecto es la creación de un algoritmo para la generación de música y letra utilizando aprendizaje profundo. Esta técnica es una subárea de aprendizaje automático que utiliza redes neuronales. Mediante varias capas de procesamiento intenta modelar abstracciones de alto nivel en los datos, utilizando estructuras basadas en transformaciones no lineales

múltiples [23].

En cualquier proyecto relacionado al área de aprendizaje automático es necesario contar con un conjunto de datos para el aprendizaje del algoritmo. En particular se quiere un corpus de canciones para entrenar un modelo que luego se pueda utilizar para la generación de música y letra. También es importante el estudio de formatos que permitan representar ambas estructuras de datos.

Los corpus utilizados están escritos en un formato de música que en definitiva es texto plano, donde se representan notas musicales y las relaciones entre sí. Con lo cual, el problema de generación de música se transforma en un problema de generación de texto.

La idea general que se utiliza para implementar la generación es la clasificación de tiras de texto. Dada una tira de caracteres, se busca predecir el siguiente carácter, utilizando un clasificador supervisado. Para generar, se toma una semilla inicial y mediante la clasificación se obtiene lo que sería el siguiente carácter. Repitiendo este proceso se generan tantos caracteres como se deseen.

Al no existir una medida numérica o una escala en donde poder ubicar los resultados obtenidos, y para a su vez evaluar la música generada de forma más objetiva, se decide realizar una variación del Test de Turing. En este test, el participante se enfrenta a distintas canciones y debe determinar si fueron compuestas por una máquina o por humanos.

1.1. Estructura del documento

El presente documento aborda todo el proceso realizado en el marco del proyecto de grado, incluyendo las áreas de desarrollo tecnológico e investigación relacionadas a la música y letra.

En el capítulo 2, se hace una introducción a aprendizaje profundo y se presentan una serie de trabajos relacionados al área. En el capítulo 3 se presenta la investigación asociada a la

representación de música, los formatos asociados y la construcción del corpus. En el capítulo 4 se desarrolla la solución planteada al problema; se presentan además distintos recursos disponibles que sirven de ayuda para la construcción del algoritmo. En el capítulo 5 se define la metodología de evaluación y se realiza un análisis de los resultados obtenidos, incluyendo el test de Turing Musical. Por último, en el capítulo 6, se realizan conclusiones y se notan trabajos a futuro, indicando puntos de mejora y extensión.

Capítulo 2

Aprendizaje profundo

El objetivo del proyecto es la creación de un algoritmo para la generación de música y letra utilizando aprendizaje profundo. Resulta interesante enfocar la solución al problema utilizando esta técnica, que se implementa con redes neuronales artificiales [23].

Las redes neuronales se caracterizan principalmente por adquirir conocimiento a través de la experiencia [18], lo cual es aplicable al problema planteado. Se quiere generar canciones aprendiendo de un conjunto previo que se le da al algoritmo con el objetivo de obtener esa experiencia. Aprendizaje profundo es una técnica que utiliza redes neuronales, conteniendo múltiples capas ocultas no lineales, convirtiendo estas redes en modelos muy expresivos ya que pueden aprender relaciones complejas entre sus entradas y salidas. Tienen también como consecuencia el crear buenas representaciones de datos en las capas más bajas [23] y con esto tener buenas características para el modelo generado.

Como ejemplo, una aplicación interesante es la asistencia en la conducción de automóviles. En la publicación *An Empirical Evaluation of Deep Learning on Highway Driving* [13] se detalla cómo entrenan redes neuronales convolucionales para la detección de carriles y otros automóviles. El entrenamiento lo llevan a cabo mediante procesamiento de imágenes y videos.

En el campo de generación de imágenes se encuentra Deep Dream [21]. Este proyecto, parte de

una red neuronal convolucional de 22 capas, que se utiliza para clasificar imágenes. Esta red fue desarrollada para el concurso *ImageNet Large-Scale Visual Recognition Challenge 2014* [28] y es luego utilizada como un generador.

En este trabajo se desea hacer uso de aprendizaje supervisado adaptado para la generación. Es decir, entrenar un modelo con ejemplos de canciones que, de alguna manera, estén anotadas, y utilizar la clasificación como forma de generación. Con este enfoque, existen varios proyectos interesantes que logran buenos resultados.

En este capítulo se presentan formalmente los conceptos básicos relacionados a aprendizaje profundo y luego se hace un resumen de los proyectos similares donde se utiliza esta técnica.

2.1. Conceptos básicos

Aprendizaje profundo es una subárea de aprendizaje automático (*machine learning*) que intenta modelar abstracciones de alto nivel en los datos mediante varias capas de procesamiento, utilizando estructuras basadas en transformaciones no lineales múltiples [23]. Se utiliza especialmente cuando la función objetivo es realmente complicada de modelar o depende de un número muy alto de parámetros como para ser modelada a mano. Este tipo de métodos se implementa utilizando redes neuronales artificiales [18], una familia de modelos que viene de los años 60 y que, a partir de nuevos algoritmos, mayor capacidad de procesamiento y mayor cantidad de datos, han resurgido en los últimos años.

2.1.1. Redes neuronales artificiales

Una neurona artificial es una función matemática que toma una o mas entradas a las que aplica una transformación que produce una salida. La transformación usual toma las entradas, realiza una suma ponderada de éstas y luego se pasa a través de una función no lineal.

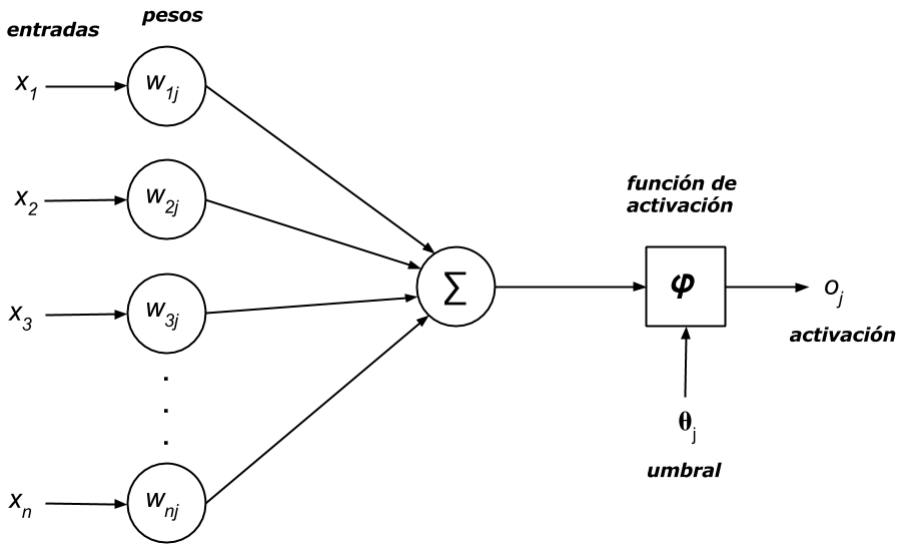

Figura 2.1: Neurona artificial

$$o_j = \varphi(\sum_{i=0}^{n} w_{ij} x_i)$$

Gráficamente esto se puede ver en la figura 2.1.

Las características de la función de activación φ son:

- **Monótona creciente:** la magnitud de la salida aumenta a medida que aumenta la magnitud de la entrada.
- **Continua:** pequeños cambios en la entrada producen pequeños cambios en la salida.
- **Diferenciable:** se puede calcular su derivada de manera eficiente.

Una red neuronal artificial estándar consiste en una cantidad de neuronas interconectadas en las que cada una produce una secuencia de activaciones. Las neuronas de entrada son activadas en base a cambios en el contexto, las demás son activadas a través de conexiones ponderadas a partir de las activaciones de las neuronas previas.

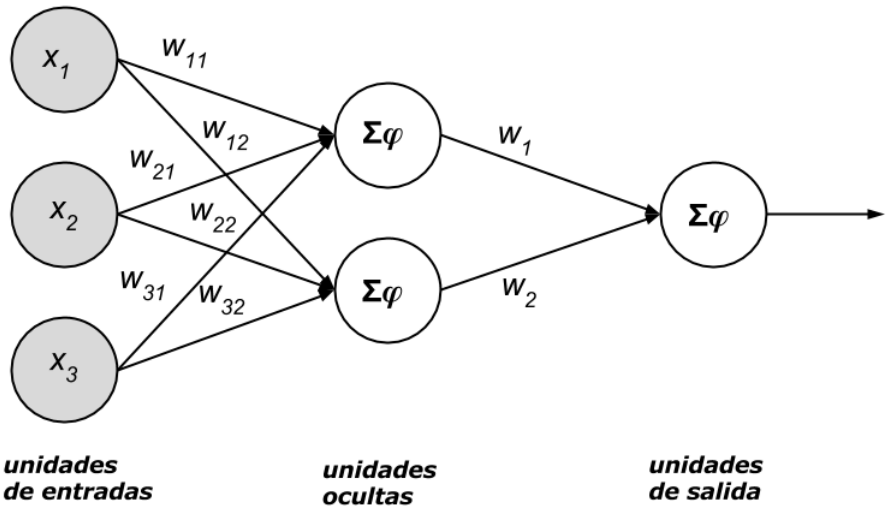

Figura 2.2: Red neuronal artificial Feedforward

2.1.2. Entrenamiento

Todo lo visto hasta ahora sobre las redes neuronales cobra sentido si se pueden cambiar, adaptar y mejorar dinámicamente las conexiones entre neuronas, para de ese modo obtener la salida esperada para una entrada en particular. Es aquí donde entra en juego el aprendizaje automático. El aprendizaje consiste en encontrar los pesos adecuados en cada neurona para un determinado ejemplo de entrada.

La idea general que hay detrás de este proceso es elegir valores aleatorios de forma de inicializar los pesos. Luego iterativamente se alimenta la red con un ejemplo de entrenamiento y al finalizar los pesos se ajustan apropiadamente si la salida es diferente al resultado esperado. Esto último, se repite tantas veces como sea necesario hasta que la salida para cada ejemplo de entrenamiento sea la correcta. Dependiendo del problema y de cómo las neuronas están conectadas, el resultado esperado puede requerir muchas etapas de procesamiento.

Existen varios algoritmos que permiten realizar el ajuste de los pesos para corregir la salida de una neurona, entre los más destacados tenemos: regla del perceptron, método del gradiente descendente y método del gradiente descendente estocástico [19] El algoritmo más utilizado para ajustar los pesos de una red neuronal de muchas capas, es la generalización del método del gradiente descendente, el algoritmo de backpropagation [20]. Se puede ver un pseudocódigo

del algoritmo en el cuadro 2.1.

- Para cada ejemplo de entrenamiento $\langle x, t \rangle$:
  - Propagar la entrada hacia adelante a través de toda la red:
    1. Introducir la instancia x en la red y calcular la salida o_u de cada unidad u en la red.
  - Propagar el error hacia atrás a través de la red:
    2. Por cada unidad de salida k, calcular el error δ_k:

$$\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k)$$

    3. Por cada unidad oculta h, calcular el error δ_h:

$$\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in outputs} w_{kh}\delta_k$$

    4. Actualizar los pesos w_{ji}:

$$w_{ji} \leftarrow w_{ji} + \alpha\delta_j x_{ji}$$

Cuadro 2.1: Pseudocódigo del algoritmo de backpropagation

Con el fin de simplificar y entender la idea por detrás del algoritmo backpropagation, es que se lo describe con una sola capa oculta. Dado que se usa aprendizaje profundo en el contexto del proyecto, se utilizan combinaciones de varias capas en profundidad.

2.1.3. RNN y LSTM

A pesar de la gran performance que las redes neuronales han mostrado para resolver muchos problemas, tienen una gran limitante: no comprenden una secuencia, esto es, saber cómo un estado es afectado por otro estado previo. Dicho esto y teniendo en cuenta que lo que se busca es clasificar tiras de caracteres, como se muestra en el capítulo 4, es que nace la necesidad de estudiar una alternativa a las redes neuronales tradicionales, las RNN (*Recurrent Neural Network*, en español redes neuronales recurrentes) [15], en particular las redes neuronales recurrentes LSTM (*Long Short-Term Memory*, en español redes de gran memoria de corto plazo) [12].

La primer idea de las RNN es apilar una o más capas ocultas, donde cada capa oculta tiene la entrada correspondiente al paso actual y además tiene otra entrada correspondiente a la salida

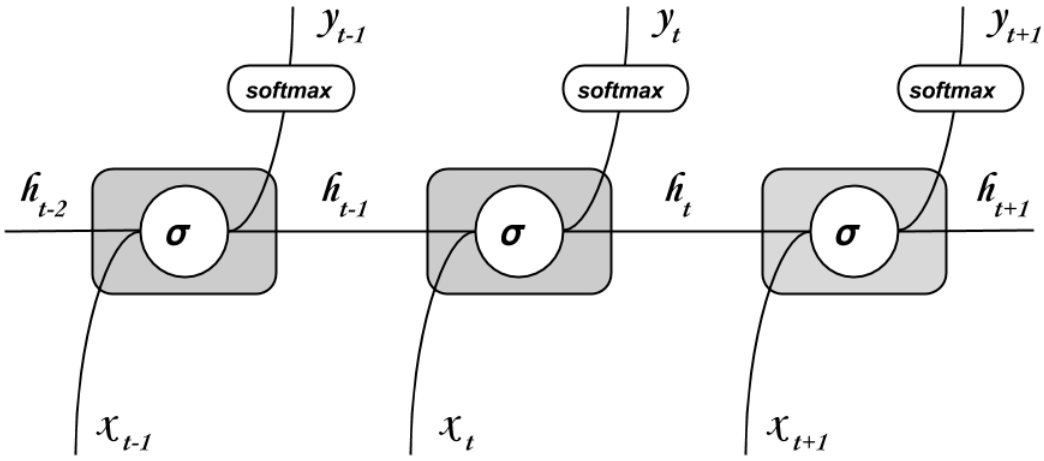

Figura 2.3: Neurona de una red RNN en el transcurso del tiempo

en el paso previo (ver figura 2.3).

$$h_t = f(W_{xh}x_t + W_{hh}h_{t-1})$$

La salida hacia la capa superior es computada usando solo la capa oculta asociada. Y la función de activación usada es *softmax* que permite hacer una transformación de la salida para que se comprenda en el intervalo (0, 1).

$$y_t = softmax(W_{hy}h_t)$$

Con lo cual, si cada capa oculta tiene además la entrada computada en el instante de tiempo anterior, este nuevo tipo de red ahora puede tener la capacidad de "recordar". Pero no puede recordar durante un tiempo largo debido al problema llamado gradiente desvanecido (*vanishing gradient*) [11]. Este fenómeno hace que, si los pesos durante el entrenamiento disminuyen a valores menores que uno, el gradiente tienda a hacerse más pequeño a medida que se retrocede a través de las capas ocultas. Esto significa que las neuronas en las capas anteriores aprenden mucho más lento que las neuronas en capas posteriores. En cambio, si durante el entrenamiento los pesos aumentan a valores mayores que uno, el gradiente aumenta exponencialmente y ocurre

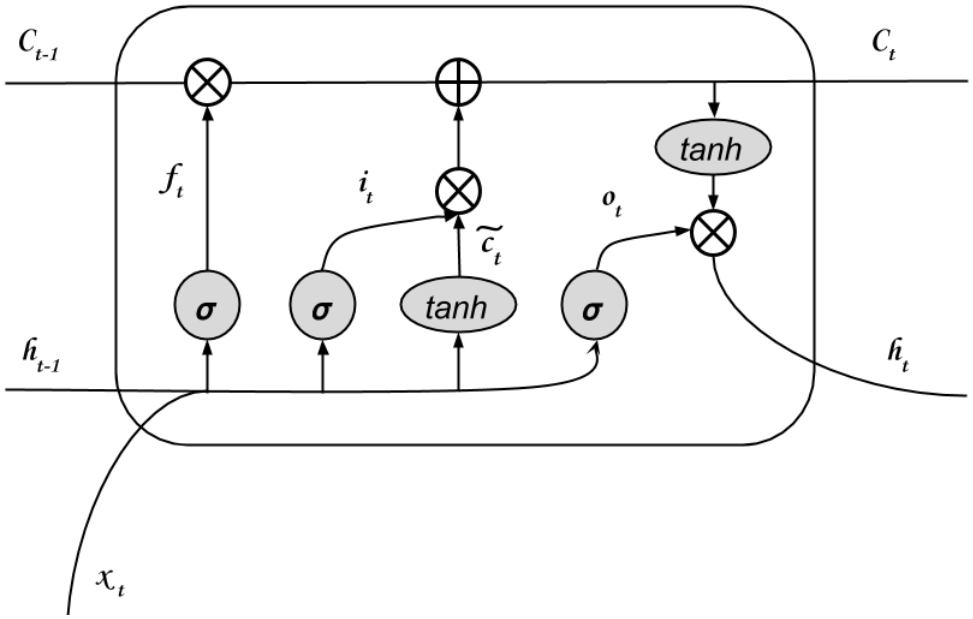

Figura 2.4: Estado de una neurona de una red LSTM

lo opuesto. Es por esto que aparece su sucesor, las redes LSTM.

La clave de las redes LSTMs es el manejo del estado de una neurona. En la figura 2.4 esto se representa con la línea horizontal que pasa por la parte superior. C_{t-1} indica el estado previo mientras que en C_t el estado actual. El estado de una neurona es como una cinta transportadora. Se ejecuta directamente en toda la cadena, con sólo algunas pequeñas interacciones lineales. Esto permite que la información fluya a lo largo sin cambios.

Analizando cómo trabajan estas neuronas, el primer paso es decidir qué información se va a quitar del estado de la neurona. Esto se hace mediante una capa *sigmoide* llamada "*forget gate layer*". Se computa h_{t-1} y x_t, y se genera un número entre 0 y 1 para cada número en el estado de la neurona C_{t-1}. Un 1 indica que se debe mantener completamente el estado, mientras que un 0 indica que se debe deshacer completamente.

$$f_t = \sigma(W_f[h_{t-1}, xt] + b_f)$$

El siguiente paso es decidir qué nueva información se debe almacenar en el estado de la neurona. Esto tiene dos partes. En primer lugar, una capa *sigmoide* llamada "*input gate layer*", la cual

decide qué valores actualizar. Luego, una capa tanh que crea un vector de nuevos valores candidatos, $\tilde{C}_t$, que pueden ser añadidos al estado. Más adelante, se combinan estos dos para crear una actualización del estado.

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

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

Para actualizar el estado antiguo de la neurona, C_{t-1}, por el nuevo estado C_t, se tienen que combinar los resultados de los pasos anteriores, que es donde en definitiva se decide qué hacer. Para eso se multiplica el estado viejo por f_t y luego se añade $i_t * C_t$. Intuitivamente, esto hace que la red LSTM sea capaz de quedarse con la información necesaria.

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Finalmente, se calcula la salida, la cual se basa en el estado de la neurona, pero filtrada. Primero, se ejecuta una capa *sigmoide* que decide qué partes del estado de la neurona formara parte de la salida, esto se representa en o_t. Luego, se pasa el estado de la neurona a través de la función tanh (para dejar los valores entre -1 y 1) y se lo multiplica por o_t.

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

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

2.2. Regularización *Dropout*

Las redes neuronales profundas contienen múltiples capas ocultas no lineales y esto las convierte en modelos muy expresivos, ya que pueden aprender relaciones muy complicadas entre sus entradas y salidas. De todos modos, existe un problema serio en las redes neuronales artificales que es el sobreajuste a los datos de entrenamiento (*overfitting*), el cual se produce cuando un modelo se ajusta demasiado a las características del corpus en el que fue entrenado. Las redes grandes, como las que se utilizan en aprendizaje profundo, también sufren esta problemática.

Es por ello que surge la técnica *dropout*, la cual es una regularización de los modelos de redes neuronales profundas propuesta por Srivastava en su artículo *Dropout: A Simple Way to Prevent Neural Networks from Overfitting* [27]. La técnica de *Dropout* consiste en seleccionar aleatoriamente neuronas que serán temporalmente ignoradas en el entrenamiento: su contribución para la activación de neuronas es removida en el pasaje hacia adelante y cualquier actualización de pesos no se aplica en la neurona en el paso hacia atrás.

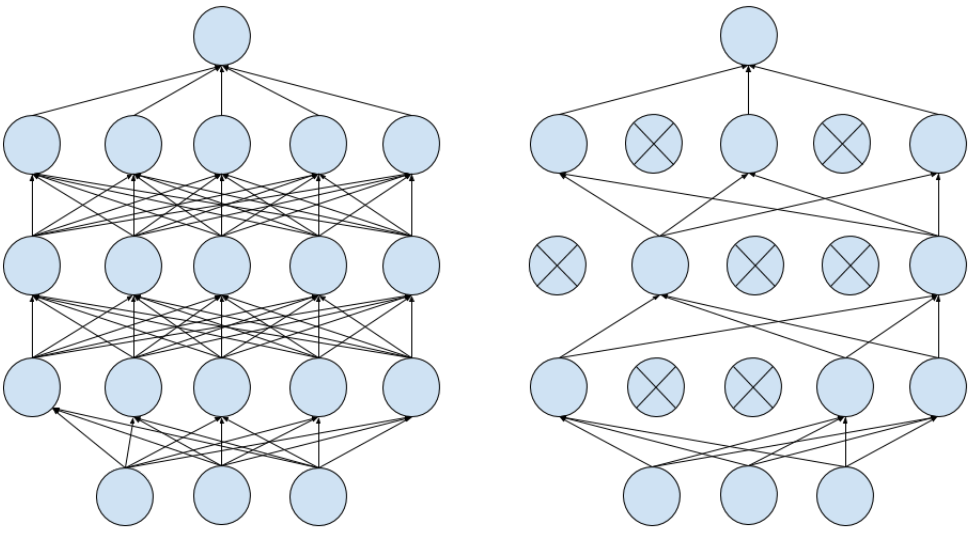

Figura 2.5: Como actúa Dropout en el entrenamiento de una red neuronal. Las neuronas marcadas con una cruz son las ignoradas por el algoritmo.

La realización de este proceso durante el entrenamiento hace que otras neuronas tengan que intervenir y manejar la representación necesaria para hacer las predicciones de las neuronas ignoradas. Con esto se obtienen múltiples representaciones internas que son aprendidas por la red. Al final, se logra que la red sea menos sensible a los pesos específicos de las neuronas. Esto da como resultado una red con una mejor generalización y además es menos probable que imite los datos de entrada.

En principio, la elección de las neuronas que se ignoran es totalmente aleatorio. En el caso más simple, cada unidad es retenida con una probabilidad fija p independiente a las demás unidades.

2.3. Proyectos similares

La mayoría de las publicaciones que realizan la aplicación de algoritmos de aprendizaje profundo sobre música lo hacen sobre un género en particular. Básicamente se entrenan modelos sobre corpus de música Jazz [17] o música Folk [32]. Al restringir a un único género, se parte de un corpus muy interrelacionado en donde existen patrones identificables, haciendo más simple, al menos en teoría, la generación. Se puede destacar que la particularidad de los géneros antes mencionados está bien marcada con resultados bastante interesantes. En algunos casos la música generada se aproxima muy bien al género del corpus en el que fue entrenado el algoritmo, con melodías que dejan marcado a cuál género pertenecen [22].

En general el formato de audio elegido por los investigadores como base para sus proyectos es el MIDI, un formato que esencialmente es un protocolo de comunicación para computadoras e instrumentos musicales electrónicos. Es un lenguaje estándar que les da soporte para comunicarse y permite la composición de música digital. Algo en común también es que se toma como base el MIDI pero se utilizan formatos intermedios, como ABC. Este es un formato que permite de una forma sencilla y estructurada la creación de melodías musicales. Las melodías en ABC son escritas en texto plano. Estos formatos se describen en mayor profundidad en el capítulo 3.

El proyecto *"SMUG"* [25] fue llevado a cabo por tres integrantes de la IT University of Copenhagen y uno de la New York University. Se plantean la generación de música y letras, tomando como punto de partida publicaciones científicas, y lo implementan en base a cadenas de Markov utilizando un algoritmo genético. Los resultados musicales son presentados únicamente como partituras, y los autores se muestran conformes con los resultados, tanto por parte de letra como de música. Un ejemplo de las letras obtenidas se puede ver en el cuadro 2.2.

1. Infinite redefine the game
2. Changes , differ , game could given place are
3. Complete traits example , pcg
4. Can well-known modification game depending
5. University place , took gaps "strategical generation"?
6. Would redefine landscapes me-me-memory
7. Based user actions mixed-initiative currently
8. Changes a actions randomly perhaps

Cuadro 2.2: Ejemplo de letras generadas en el proyecto SMUG

Otro de los proyectos interesantes donde se usa aprendizaje profundo para componer música es el que desarrolla Daniel Johnson [4], usando redes neuronales recurrentes y MIDI como formato musical. Johnson construye la arquitectura de su red de forma que lo generado cumpla ciertas propiedades, tales como ser invariante en nota, esto es, que las relaciones entre las notas sean más importantes que su posición en la escala. Esto permitiría que la red identifique una relación, por ejemplo, para una misma canción en tonalidades distintas. La principal salida de este algoritmo es la probabilidad de que una nota en particular sea elegida para ser tocada. Este proyecto es desarrollado en Python, utilizando el framework de aprendizaje profundo Theano y el formato de audio MIDI.

Capítulo 3

Corpus

La correcta elección o construcción de un corpus forma parte importante de cualquier proyecto relacionado al área de aprendizaje automático. Para este trabajo se desea un conjunto de canciones con la finalidad de entrenar un modelo de red neuronal que luego se pueda utilizar para la generación de música y letra. Además, dada esta particularidad del problema, también es una tarea importante el estudio de formatos que permitan representar ambas estructuras de datos.

Debido principalmente a la escasez de recursos y de colecciones de canciones en alguno de los formatos elegidos (solamente se encuentra un corpus de folk) se decide la construcción de uno que se adapte completamente al problema.

En este capítulo se presentan los dos formatos de música elegidos. Asimismo se muestran las modificaciones que se le hacen al corpus de folk en base a las necesidades del proyecto y se detalla el proceso de construcción del corpus final sobre canciones de rock.

3.1. Formatos de música

El uso de aprendizaje profundo como paradigma para implementar un algoritmo que componga música requiere en primera instancia especificar cuál es la entrada. Los formatos no sólo forman parte de la entrada, sino que también de la salida. Esto último es muy significativo ya que son el resultado palpable de lo desarrollado y lo que permite al final evaluar el algoritmo.

Con respecto al formato que represente el audio y la letra de una canción, se decide utilizar MIDI y ABC. En principio MIDI es un formato muy interesante, ya que es usado en proyectos similares, como se mostró en el capítulo 2. A su vez posee la ventaja de ser muy popular y por ende existen bases de datos libres, como por ejemplo Midi World[1] que posee miles de canciones de los más variados géneros; solamente de rock hay 2380. Esto es interesante ya que se hace viable la construcción de un corpus a ser utilizado por el algoritmo.

Sin embargo, los archivos MIDI se representan de forma binaria y procesarlos resulta un tanto más complicado. Por este motivo surge la posibilidad de utilizar el formato ABC que tiene la ventaja de representar a las canciones en texto plano, permitiendo que el procesamiento sea menos costoso. Por otro lado, hace posible el análisis de la salida del algoritmo de una forma más clara, pudiendo detectar partes de generaciones potencialmente iguales al corpus. Como contrapartida, no hay grandes bases de datos de donde se puedan obtener canciones en este formato.

Tampoco existe una base de datos en MIDI ni en ABC suficientemente grande que contenga letra en sus canciones, si bien los formatos lo soportan. Por este motivo y en pro de utilizar las ventajas de ambos formatos, se opta por obtener las canciones MIDI sin letras y posteriormente se realiza un proceso de conversión a ABC aprovechando su simple formato para el procesamiento, utilizándolo como entrada al algoritmo. Luego, para solventar la falencia de las letras, se crea un *crawler*[2] que descargue las letras y las incorpore en las canciones. Sobre éste se profundiza en las siguientes subsecciones.

[1] `http://www.midiworld.com/`

[2] Script que navega, de forma automática, por sitios web para obtener datos que sean de interés.

A continuación se hace una introducción a ambos formatos y se muestra brevemente cómo se codifican.

3.1.1. MIDI

El término MIDI[3] proviene de Musical Instrument Digital Interface y esencialmente es un protocolo de comunicación para computadoras e instrumentos musicales electrónicos. Es un lenguaje estándar que les da soporte para comunicarse y permite la composición de música digital.

Los datos MIDI son una lista de eventos o mensajes que forman instrucciones para que un dispositivo electrónico (instrumento musical, tarjeta de sonido, celular, etc) sea capaz de generar cierto sonido. Se componen principalmente de la información que describe qué notas musicales se van a reproducir, en qué momento y los instrumentos que se van a utilizar. Aunque los archivos no almacenan sonidos, como otros formatos digitales de audio existentes hoy día (mp3. ACC, WMA, etc), pueden ser considerados como música digital. La gran ventaja de estos archivos es que son muy pequeños y pueden ser reproducidos en cualquier dispositivo que soporte el protocolo. Es también un formato muy utilizado, y existen grandes colecciones de archivos disponibles para descarga legal gratuita.

El formato MIDI es usado en proyectos similares, pero su procesamiento es más complejo que el ABC. Los archivos se estructuran en trozos de bytes [3] donde los primeros 4 son el tipo, los siguientes 4 el largo y los restantes bytes (variables, depende del largo) los datos. En nuestra solución se toma archivos MIDI como base, debido a la gran cantidad y variedad que existen en la web, pero se los transforma a ABC para su posterior procesamiento.

[3] `https://www.midi.org/`

3.1.2. ABC

ABC[4] es un formato que permite de una forma sencilla y estructurada la creación de melodías musicales. Las melodías en ABC son escritas en texto plano; un único archivo puede contener varias canciones. Típicamente una melodía en ABC consiste de dos grandes partes, el cabezal y las notas.

Cada melodía puede contener muchos campos en su cabezal, por lo que existen muchos identificadores para poder definirlos [1], pero generalmente los que suelen aparecer más y tiene mayor importancia son X, M, y K. El cabezal es un bloque en el que se especifica toda la información que no sean las notas musicales. Se compone de campos de información y configuración, donde cada campo se introduce en una línea diferente. Al comienzo de cada línea se introduce una letra, que hace referencia a la información que allí se define, seguida de dos puntos. Por ejemplo "T:Back in Black" contiene al Título de la melodía y "C: AC/DC" al Compositor.

M y K definen la métrica y el tono de la melodía respectivamente, por lo que una posible melodía podría definir la métrica en 6/8 y el tono en D (re en el sistema de notación musical anglosajón). X define un número de referencia para la melodía, esto se utiliza para identificar una melodía dentro de un archivo, ya que se puede definir más de una en un mismo archivo de texto ABC. Si bien no existe un orden para los identificadores, es recomendable comenzar el cabezal con el número de referencia X y terminarlo con el tono K, solamente por un tema de fácil lectura.

Las notas están definidas por el sistema de notación musical anglosajón donde se nombran las notas desde la letra A hasta la G. A su vez existen diversos modificadores para representar octavas inferiores y superiores, notas en sostenido o bemol, representar silencios, longitud de notas, entre otros. En el cuadro 3.1 se muestra un ejemplo del formato y en la figura 3.1 su melodía correspondiente.

[4] `http://abcnotation.com/`

```
X:1
T:Dusty Miller, The Binny's Jig
M:3/4
K:G
B>cd BAG|FA Ac BA|B>cd BAG|DG GB AG:|
Bdd gfg|aA Ac BA|Bdd gfa|gG GB AG:|
BG G/2G/2G BG|FA Ac BA|BG G/2G/2G BG|DG GB AG:|
```

Cuadro 3.1: Ejemplo de una canción representada con el formato ABC

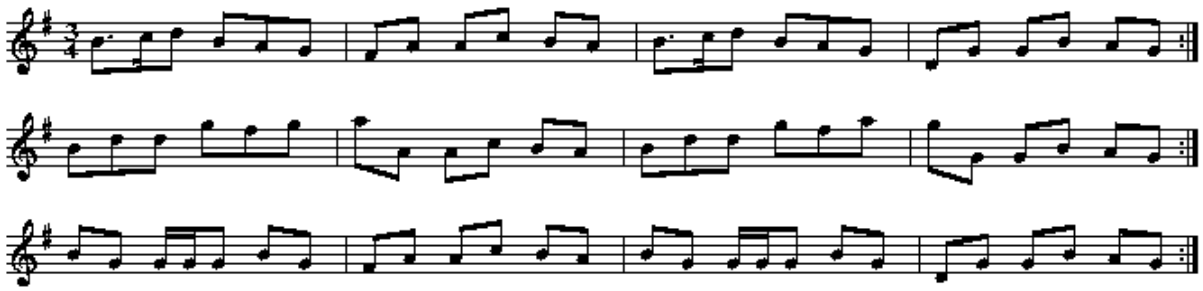

Figura 3.1: Melodía correspondiente a la canción representada en el cuadro 3.1

El cuadro 3.1 es un ejemplo de una posible entrada al algoritmo. Asimismo puede ser un ejemplo de salida. En definitiva este tipo de estructura es lo que se quiere generar en el modelo que se entrena.

3.2. Aproximación al corpus

En base a la estructura necesaria y a los formatos, se indaga sobre las posibles formas de obtener un corpus. Se investiga y evalúa la oferta de conjuntos de música con estas características como así también la posibilidad de construir uno propio.

Si bien la búsqueda de conjuntos de música en ABC que se realiza no es exhaustiva, pretende encontrar un corpus que permita hacer un estudio interesante sobre la generación de música y letra. Lamentablemente no se logra encontrar uno que cumpla los requerimientos de estructura vistos previamente, es decir, que sean temas musicales en ABC y que tengan sus correspondientes letras.

En esta sección se describe el único corpus en ABC encontrado y la forma con la que se fueron construyendo algunos prototipos de evaluación, partiendo de la obtención de música en MIDI, hasta llegar a un conjunto final de canciones propio, creado para este proyecto.

3.2.1. Nottingham con letras

Este corpus, llamado *Nottingham Music Database* [9], fue creado por Eric Foxley, un profesor británico que trabajó durante muchos años en la Universidad de Nottingham en las áreas de Matemáticas y Computación y como investigador en el área de la música estudiando y analizando formas de almacenamiento, así como temas relacionados con el diseño de lenguajes y sistemas para la descripción de música y partituras en una computadora. Posee alrededor de 1000 temas musicales de folk (británico y americano) provenientes en su mayoría de la *Fred Folks Ceilidh Band*, una banda que ha estado activa por más de 50 años en Nottingham y sus alrededores, componiendo música cuyo destino era específicamente el baile. Entre los temas que lo componen, se encuentran variaciones del género folk británico y americano, como ser *Jibs, Waltzes, Hornpipes, etc.*

El corpus está constituido por temas musicales puramente instrumentales, por lo que no poseen letra asociada. Un ejemplo de melodía del corpus de *Nottingham* se puede ver en el cuadro 3.2.

```
X: 1
T:A and D
% Nottingham Music Database
S:EF
M:4/4
K:A
M:6/8
P:A
f|"A"ecc c2f|"A"ecc c2f|"A"ecc c2f|"Bm"BcB "E7"B2f|
"A"ecc c2f|"A"ecc c2c/2d/2|"D"efe "E7"dcB| [1"A"Ace a2:|
 [2"A"Ace ag=g||\
K:D
P:B
"D"f2f Fdd|"D"AFA f2e/2f/2|"G"g2g ecd|"Em"efd "A7"cBA|
"D"f^ef dcd|"D"AFA f=ef|"G"gfg "A7"ABc |1"D"d3 d2e:|2"D"d3 d2||
```

Cuadro 3.2: Ejemplo de una canción del corpus de Nottingham

Entonces se decide incorporar pequeños textos que simulen la letra sobre cada una de las canciones. Las letras fueron obtenidas mediante un *crawler*, que se implementa para adquirirlas desde sitios web[5].

Las letras son agregadas en el campo que ABC destina, luego del prefijo *w:* . Se realiza, además, una serie de ajustes. Por un lado se agrega un tag al principio de la canción y otro al final, que pretende dejar más clara la estructura de una canción, ya que todas forman parte de un mismo archivo.

También se hace una normalización de los caracteres del cabezal y de las letras a minúsculas, para que la cantidad de clases que el algoritmo necesite para clasificar sea la menor posible. El resultado de este proceso se puede ver en el cuadro 3.3.

```
[begin]
X:1
T:a and d
M:4/4
K:A
M:6/8
P:A
f|"A"ecc c2f|"A"ecc c2f|"A"ecc c2f|"Bm"BcB "E7"B2f|
"A"ecc c2f|"A"ecc c2c/2d/2|"D"efe "E7"dcB| [1"A"Ace a2:|
 [2"A"Ace ag=g||\
K:D
P:B
"D"f2f Fdd|"D"AFA f2e/2f/2|"G"g2g ecd|"Em"efd "A7"cBA|
"D"f^ef dcd|"D"AFA f=ef|"G"gfg "A7"ABc |1"D"d3 d2e:|2"D"d3 d2||
w: they looked me up and down a bit
w: and turned to each other
w: i say
w: i don't like cricket oh no
w: i love it
w: i don't like cricket no no
[end]
```

Cuadro 3.3: Ejemplo de canción del corpus de Nottigham con letras

[5] www.azlyrics.com,www.songlyrics.com

3.2.2. Corpus de rock: primera versión

Se trata de encontrar temas musicales en ABC que puedan ser el punto de partida para la construcción de un corpus. Sin embargo, en las búsquedas realizadas no se encuentra una gran cantidad de temas musicales en formato ABC. Esto en principio impide construir un corpus en el que sea viable entrenar un algoritmo de aprendizaje profundo. Por otro lado, sí existe un formato de música (MIDI) en el que se encuentra una cantidad suficiente de temas musicales como para armar un corpus, y que a su vez pueden ser convertidos a ABC con una aplicación de consola muy simple llamada MIDI2ABC[6].

Se decide obtener canciones que sean del género rock, debido a que es el único que posee una cantidad significativa de canciones, unas 2380. Cualquier otro género posee 1080 (como el género pop) o menos. Con este preámbulo se construyen dos *crawlers*. Uno permite descargar de internet archivos MIDI y el otro en base al nombre de la canción descarga la letra correspondiente. Antes de poder unir cada canción con su letra se debe convertir los MIDI a ABC con el conversor comentado anteriormente. Para ello se crea un proceso que convierte al archivo MIDI con su duración y tamaños originales, sin ningún tipo de modificación al formato. En la figura 3.2 puede verse un diagrama de este proceso.

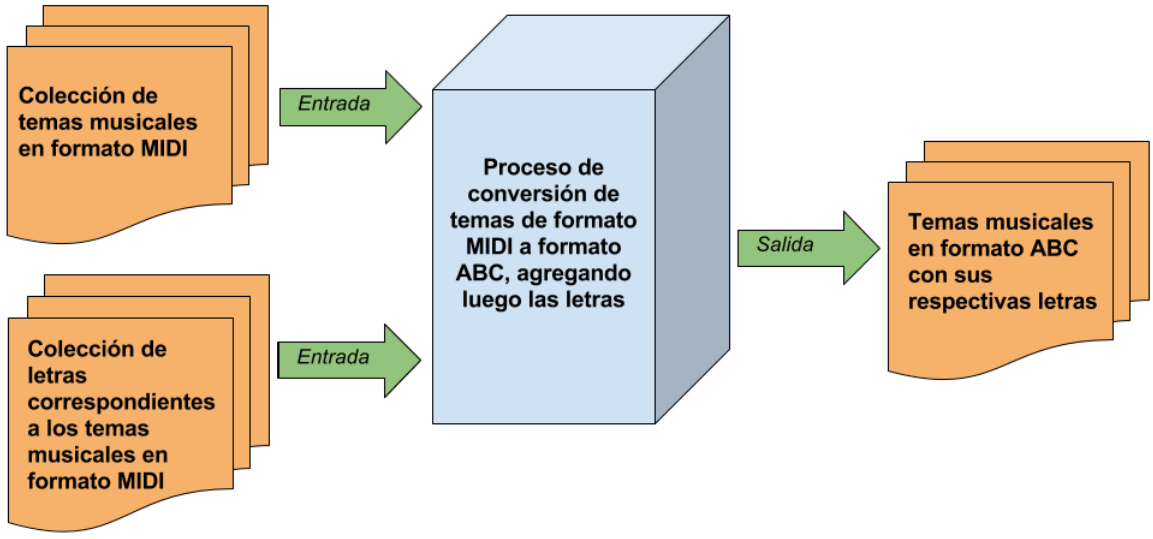

Figura 3.2: Proceso de conversión de los temas musicales.

[6] http://abc.sourceforge.net/abcMIDI/

Con el objetivo de generar una entrada uniforme al algoritmo se decide tener todas las canciones en un mismo archivo de texto, donde se tiene música en ABC seguido de su correspondiente letra. Para esto se construye un nuevo *script*, que une al archivo ABC de música con su correspondiente letra.

El resultado final de este preprocesamiento se representa en el cuadro 3.4.

```
<BEGIN>
X: 1
T: from AC_DC_-_Dirty_Deeds_Done_Dirt_Cheap.mid
M: 4/4
L: 1/8
Q:1/4=140
K:C %0 sharps
V:1
%Rhythm
%%MIDI program 29
%%MIDI program 29
[^G,6-E,6-B,,6-E,,6-] [^G,E,B,,E,,][=G,-D,-G,,-]| \
[G,D,G,,][^G,6E,6B,,6E,,6][A,-E,-A,,-]| \
[A,E,A,,][^G,6E,6B,,6E,,6][D-A,-D,-A,,-]| \
[DA,D,A,,][^G,6-E,6-B,,6-E,,6-][^G,E,B,,E,,]|
[^G,6-E,6-B,,6-E,,6-] [^G,E,B,,E,,][=G,-D,-G,,-]| \
[G,D,G,,][^G,6E,6B,,6E,,6][A,-E,-A,,-]| \
[A,E,A,,][^G,6E,6B,,6E,,6][D-A,-D,-A,,-]| \
[DA,D,A,,][^G,6-E,6-B,,6-E,,6-][^G,E,B,,E,,]|
                    .
                    .
                    .
w: I'm happy to be
w: Your back door man
w:
w: Dirty deeds done dirt cheap
w: Dirty deeds and they're done dirt cheap
w:
w: Concrete shoes, cyanide, tnt
w: Done dirt cheap
w: Neckties, contracts, high voltage
w: Done dirt cheap
<END>
```

Cuadro 3.4: Ejemplo de una canción luego de realizado el proceso de conversión

Dos aclaraciones importantes: desde el punto de vista del tamaño y la complejidad en la sección de notas, el anterior es un ejemplo representativo del corpus. Por otro lado, por un tema de espacio, el ejemplo mostrado está recortado. Aquí se muestran 30 líneas de las 509 del ejemplo original.

Si bien la idea de construir este corpus es sumamente interesante, ya que se parte de una cantidad de canciones de un determinado género en MIDI, se posee un conversor para transformarlos a ABC, y se cuenta también con las letras correspondientes, hay un detalle no menor que no se tuvo en cuenta: la decisión por omisión de convertir los MIDI en bruto, esto es, sin ningún tipo de preprocesamiento. Esto trae consigo dos grandes problemas, que fueron luego las causas de que este resultado sea inutilizable. Por un lado, la conversión genera archivos ABC sumamente largos debido a que los MIDI son de 3 minutos en promedio, llevando a que los tiempos de entrenamiento de la red sean altos. Por otro lado, debido a que los MIDI poseen más de un instrumento (en promedio 6), la estructura del ABC resultante queda sumamente compleja, dificultando el aprendizaje por parte del algoritmo. El uso de varios instrumentos en una canción plantea además otra complejidad: la armonía y sincronización entre ellos.

Debido a estos problemas, se opta entonces por dejar de lado al corpus construido. Igualmente el proceso deja lecciones aprendidas que se utilizan para las posteriores construcciones.

Teniendo en cuenta los factores de falla del proceso anterior, se trata de construir otro corpus a partir de canciones de juegos de consola en formato MIDI, de los cuales existe una gran cantidad en la web y en teoría poseen menos instrumentos y su duración es menor a un minuto en promedio. Al convertir los archivos MIDI a ABC, si bien se nota una leve mejoría en cuanto a tamaño en bruto, al final, los MIDI no son tan cortos en promedio, ni contienen una menor cantidad de instrumentos, con lo que básicamente se llega al mismo problema que en el caso anterior.

3.3. Corpus de rock

Los dos factores más importantes que hacen que los corpus anteriores no sean de utilidad son la larga duración en tiempo y la cantidad de pistas (instrumentos) que poseen sus canciones. Por eso es que se construye un nuevo corpus teniendo como base la primera versión del corpus de rock antes descrito. El objetivo principal es lidiar con esos dos factores tratando de minimizarlos

tanto como sea posible.

Si bien acortar las canciones en tiempo y reducir la cantidad de instrumentos parece una tarea medianamente sencilla y automatizable, recortar duraciones o quitar canales de las canciones de forma automática, puede dar como resultado porciones que no sean las más valiosas de cada canción.

Es claro que automatizar el recorte de las canciones lograría obtener el corpus en un tiempo menor, pero, con chances de que no sea de la calidad esperada. Por otro lado, editar las canciones manualmente es un trabajo lento y sumamente engorroso, pero que genera un corpus con una calidad sonora, al menos, mejor que el anterior. Poniendo ambas opciones en la balanza es que se decide editar manualmente los archivos MIDI, en pro de obtener un corpus de mayor calidad. La edición se hace con la ayuda de un software libre llamado MidiEditor [7] el cual permite mediante una interfaz gráfica ver y editar los canales que componen el MIDI, como así también poder cortarlos en duración.

Se editan un total de 866 canciones y se procede a convertirlas a ABC, que es el formato que el algoritmo va a procesar. Lleva entre 5 y 8 minutos la edición por cada tema, lo cual implica entre 72 y 115 horas hombre la edición de todo el corpus. Una canción completa del corpus puede verse en el cuadro 3.5.

Se configura un nuevo *crawler* para obtener las letras de las canciones. Luego de obtenidas las letras se hace el mismo procedimiento que en los corpus anteriores juntando las letras y las canciones. De las letras de cada canción, se utilizan solamente 6 líneas. Esto se hace para disminuir el tamaño total del archivo de texto final, uno de los problemas de versiones anteriores. El mismo ejemplo mostrado anteriormente pero con las letras ya añadidas se puede ver en el cuadro 3.6.

Luego de la serie de transformaciones que se le realizan a las canciones, se debe hacer un postprocesamiento para dejar lo más prolijo posible el ABC y tener una cantidad mínima de

[7] `http://midieditor.sourceforge.net/`

```
X: 1
T: led zeppelin - babe i'm gonna leave you
M: 4/4
L: 1/8
Q:1/4=140
K:C
%%MIDI program 24
[E,A,,-] [CA,,-][E-A,,] [EA,,-][E,A,,-] [A,A,,-][C-A,,]| \
[CG,,-][E,G,,-] [G,G,,-][D-G,,] [DG,,-][E,G,,-] [G,G,,-][C-G,,]| \
[C^F,,-][D,^F,,-] [A,^F,,-][D-^F,,] [D^F,,-][D,^F,,-] [A,^F,,-][C-^F,,]| \
[C=F,,-][F,F,,-] [A,F,,-][D-F,,] [DE,,-][E,E,,-] [^G,E,,-][B,-E,,]|
[B,A,,-][E,A,,-] [A,A,,-][=G-A,,] [GA,,-][E,A,,-] [A,A,,-][E-A,,]| \
                    .
                    .
                    .
[EG,,-][E,G,,-] [G,G,,-][G-G,,] [GG,,-][E,G,,-] [G,G,,-][E-G,,]| \
[E^F,,-][D,^F,,-] [A,^F,,-][D-^F,,] [D^F,,-][D,^F,,-] [A,^F,,-][C-^F,,]| \
[C=F,,-][F,F,,-] [A,F,,-][C-F,,] [CE,,-][E,E,,-] [^G,E,,-][B,-E,,]|
[B,A,,-][E,A,,-] [A,A,,-][=G-A,,] [GA,,-][E,A,,-] [A,A,,-][E-A,,]| \
[EG,,-][E,G,,-] [G,G,,-][G-G,,] [GG,,-][E,G,,-] [G,G,,-][E-G,,]| \
[E^F,,-][D,^F,,-] [A,^F,,-][D-^F,,] [D^F,,-][D,^F,,-] [A,^F,,-][C-^F,,]| \
```

Cuadro 3.5: Ejemplo de una canción del corpus en ABC.

```
X: 1
T: led zeppelin - babe i'm gonna leave you
M: 4/4
L: 1/8
Q:1/4=140
K:C
%%MIDI program 24
[E,A,,-] [CA,,-][E-A,,] [EA,,-][E,A,,-] [A,A,,-][C-A,,]| \
[CG,,-][E,G,,-] [G,G,,-][D-G,,] [DG,,-][E,G,,-] [G,G,,-][C-G,,]| \
[C^F,,-][D,^F,,-] [A,^F,,-][D-^F,,] [D^F,,-][D,^F,,-] [A,^F,,-][C-^F,,]| \
[C=F,,-][F,F,,-] [A,F,,-][D-F,,] [DE,,-][E,E,,-] [^G,E,,-][B,-E,,]|
[B,A,,-][E,A,,-] [A,A,,-][=G-A,,] [GA,,-][E,A,,-] [A,A,,-][E-A,,]| \
                    .
                    .
                    .
[EG,,-][E,G,,-] [G,G,,-][G-G,,] [GG,,-][E,G,,-] [G,G,,-][E-G,,]| \
[E^F,,-][D,^F,,-] [A,^F,,-][D-^F,,] [D^F,,-][D,^F,,-] [A,^F,,-][C-^F,,]| \
[C=F,,-][F,F,,-] [A,F,,-][C-F,,] [CE,,-][E,E,,-] [^G,E,,-][B,-E,,]|
[B,A,,-][E,A,,-] [A,A,,-][=G-A,,] [GA,,-][E,A,,-] [A,A,,-][E-A,,]| \
[EG,,-][E,G,,-] [G,G,,-][G-G,,] [GG,,-][E,G,,-] [G,G,,-][E-G,,]| \
[E^F,,-][D,^F,,-] [A,^F,,-][D-^F,,] [D^F,,-][D,^F,,-] [A,^F,,-][C-^F,,]| \
w: babe, baby, baby, i'm gonna leave you.
w: i said baby, you know i'm gonna leave you.
w: i'll leave you when the summertime,
w: leave you when the summer comes a-rollin',
w: leave you when the summer comes along.
w: babe, babe, babe, babe, babe, babe, baby, mmm, baby
```

Cuadro 3.6: Ejemplo de una canción en ABC con letras

caracteres distintos que mejoren la performance al momento de entrenar la red neuronal. Se hace un procesamiento tanto con las canciones como con las letras asociadas.

En primer lugar las conversiones realizadas con MIDI2ABC generan una directiva que representa al instrumento asociado llamada *midi program x*, donde x es un valor numérico que identifica al instrumento. Por errores de conversión, en muchos casos se generan más de una directiva por canción, en lugar de una, que sería lo correcto debido a que las canciones poseen un solo instrumento. Además la conversión agrega metadata sobre el autor, compositor y otra información en forma de comentarios que tampoco son de importancia para el procesamiento.

Luego de la obtención de las letras a través del *crawler* creado, se hace una normalización de los caracteres y se transforman las letras a minúsculas para que la cantidad de clases que el algoritmo necesite para clasificar sea la menor posible.

Capítulo 4

Solución

Como se mencionó en el Capítulo 2, las redes neuronales tradicionales no poseen la capacidad de saber cómo un estado es afectado por un estado previo, pero hay un tipo de red que sí, las RNN. Las RNN son redes donde las neuronas poseen bucles, permitiendo que la información sea almacenada. Esto hace que este tipo de redes sean sumamente atractivas para trabajos de generación dentro de un dominio dado, debido a que el uso de memoria permite abordar problemas donde existen dependencias entre estados.

Durante la fase de *back-propagation*, en las RNN tradicionales, la señal del gradiente puede ser multiplicada un número grande de veces por la matriz de pesos asociada con las conexiones entre las neuronas de la capa oculta. Esto significa que la magnitud de los pesos puede tener un fuerte impacto en el proceso de aprendizaje. Si los pesos de la matriz son pequeños el aprendizaje puede ser muy bajo por lo que la convergencia puede tardar mucho o incluso detenerse. Por otro lado, si los pesos de la matriz son altos, puede ocurrir que el aprendizaje no llegue a converger. Estos problemas son básicamente la motivación para la creación del modelo LSTM.

Muchos de los trabajos relacionados al área, más específicamente en la generación de texto, utilizan redes LSTM donde se comprueba que dan buenos resultados. Daniel Johnson [4] en su proyecto expone una mirada hacia otros trabajos del área marcando que las capas LSTM

son atractivas para el objetivo planteado. Douglas y Jürgen [5] concluyen en su investigación que su algoritmo basado en LSTM ha logrado aprender la estructura de un formato musical y en base a ello componer nuevas piezas en ese formato de forma exitosa. Andrej Karpathy [15] en su publicación *The Unreasonable Effectiveness of Recurrent Neural Networks* afirma que las redes LSTM en la práctica funcionan un poco mejor que las tradicionales, debido a su ecuación de actualización más poderosa y algunas dinámicas de *backpropagation* más atractivas.

El objetivo de este proyecto es la construcción de un algoritmo que logre componer música y letra utilizando aprendizaje profundo. Concretamente, dada una colección de temas musicales, entrenar un modelo y con éste generar nuevos temas musicales. Como se detalla en el Capítulo 3, el corpus está construido utilizando un formato de música que en definitiva es texto plano, con lo cual el problema de generación de música se transforma en un problema de generación de texto, pero no cualquier texto, sino texto con una estructura bien marcada por la notación ABC, que representa notas musicales y las relaciones entre sí.

En lo que sigue de este capítulo se describe la estrategia utilizada para la generación y se detallan todos los pasos y decisiones tomadas en la de etapa de entrenamiento.

4.1. Generando con aprendizaje profundo

Considerando un problema clásico con aprendizaje supervisado, un corpus posee elementos de entrenamiento, los cuales tienen una etiqueta que lo clasifica en una determinada clase. En este proyecto se utiliza este enfoque, ya que se usan elementos etiquetados, pero desde un punto de vista distinto. Dado un ejemplo de entrenamiento, que en este caso es una tira de caracteres, el objetivo del generador que se entrena es predecir el siguiente carácter, basándose en el aprendizaje adquirido.

Relacionando lo anterior a nuestro problema y teniendo en cuenta que el corpus con el que se trabaja en este proyecto posee canciones representadas en texto plano, se decide fraccionar a

cada canción en partes iguales, donde cada una es utilizada como instancia de entrenamiento. Fraccionar las canciones en partes de igual largo es consecuencia de que los tipos de redes que se utilizan poseen entradas de largo fijo.

Sea $y = x_1, x_2, ..., x_n$ un elemento de entrenamiento donde cada $x_i, i : 1, ..., n$ representa al carácter en la posición i, se realiza una representación vectorial de cada elemento de entrenamiento para proveerle la entrada a la red. Al querer hacer uso de un aprendizaje supervisado se necesitan tener los elementos *etiquetados*: se toma como etiqueta al carácter x_{n+1} como la etiqueta del elemento y antes definido.

Tomemos como ejemplo la siguiente fracción de texto extraída del corpus de rock, correspondiente a una canción de Bruce Springsteen:

```
w: i was bruised and battered
```

El texto se divide en tiras de caracteres de tamaño n, tomemos *n=10* para este ejemplo. El primer elemento de entrenamiento es *"w: i was b"*, por lo tanto el siguiente carácter, "*r*", es la predicción esperada.

Con esta estructura luego de tener un modelo entrenado por el algoritmo es que se realiza la generación de música y letra en formato ABC. Básicamente lo que se hace en el método de generación del algoritmo es hacer predicciones secuenciales, dada una semilla primaria. Esto es: consideremos la semilla $s = x_1, x_2, ..., x_n$, (como se ve, s posee la secuencia de caracteres que conforman un elemento sin su etiqueta, pues esto es lo que se quiere predecir por el algoritmo). La semilla s es la entrada del método de generación; en base a esto y a lo aprendido por el modelo se predice cual es la *etiqueta* del elemento, que no es más que un carácter dentro del abecedario posible de caracteres. Éste luego es colocado a continuación de la secuencia s, siendo el carácter x_{n+1}. La salida hasta el momento es la tira $x_1, x_2, ..., x_n, x_{n+1}$.

Este proceso se repite, teniendo en cuenta que en la siguiente iteración la nueva semilla que el algoritmo toma como entrada es $s_1 = x_2, x_3, ..., x_{n+1}$. En resumen, luego de cada predicción se modifica la semilla, haciendo un desplazamiento hacia la derecha de largo 1. Los caracteres

se van guardando en el orden en que fueron generados y luego de finalizar las generaciones se tiene en texto plano la (o las) canciones generadas.

4.2. Arquitectura de la red

La arquitectura inicial de la red, tiene una capa LSTM con una cantidad determinada de neuronas y un vector de entrada de dimensión igual al largo de un elemento de entrenamiento por el tamaño del abecedario. El abecedario es la cantidad de caracteres distintos que aparecen en el corpus o, visto de otra manera, la cantidad de posibles etiquetas. Esto se representa en la figura 4.1.

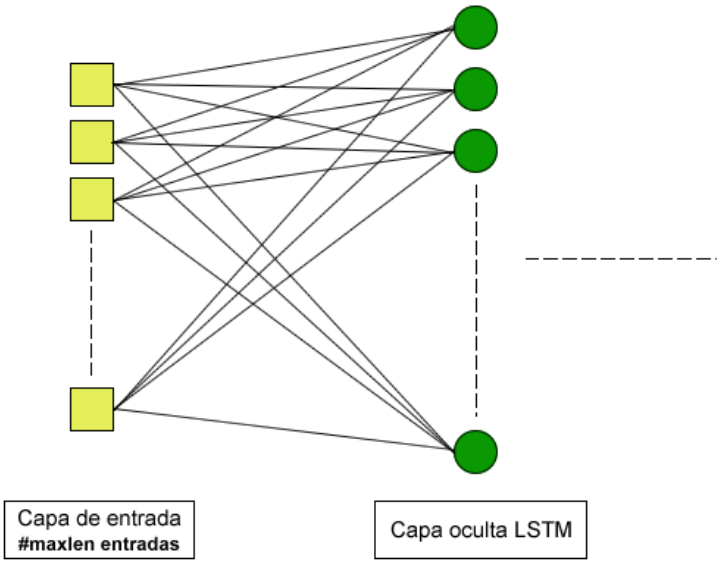

Figura 4.1: Vista parcial de la topología de red base. Una capa de entradas y una capa oculta LSTM

Luego se aplica un *dropout* con tasa **d** (ver figura 4.2). Esto se lleva a cabo ignorando ciertas neuronas durante el entrenamiento, fijando aleatoriamente una fracción **d** de entradas a 0 en cada actualización durante el tiempo de entrenamiento. Esta técnica es utilizada para tratar de prevenir el sobreajuste, uno de los factores que hacen que un algoritmo tenga baja performance.

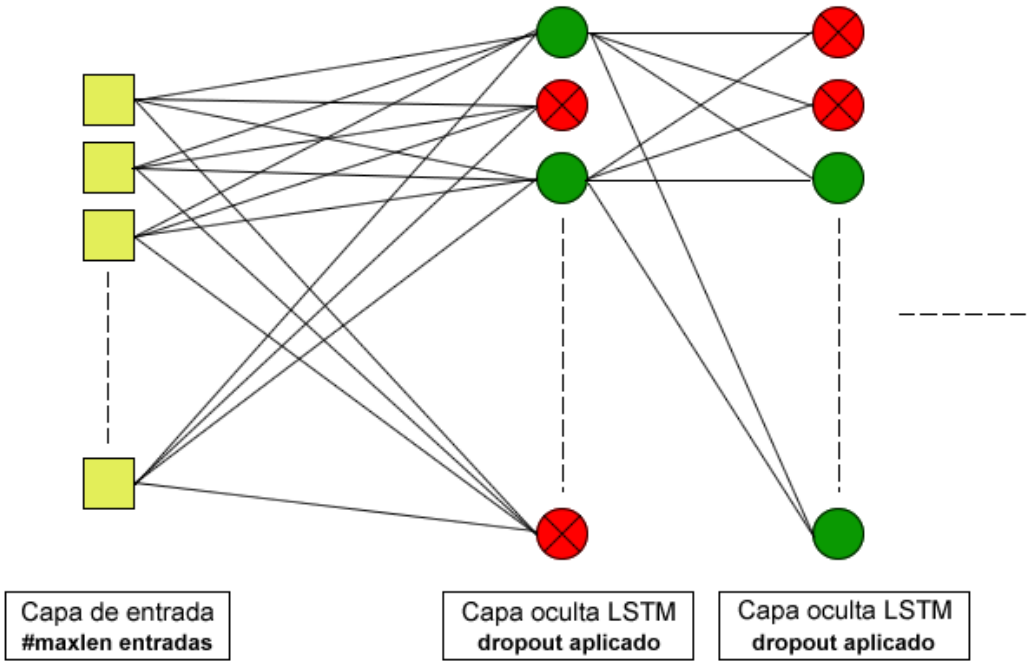

Figura 4.2: Vista parcial de la topología de red base. Una capa de entradas, una capa oculta LSTM a la cual se le aplicó el dropout

El modelo posee varias capas LSTM utilizando *dropout*. Luego se prueban variaciones, agregando en profundidad más de estas capas. Finalmente se tiene una capa densa con una cantidad de neuronas igual al tamaño del abecedario y una función de activación *softmax*. *Softmax* permite hacer una transformación de la salida para que se comprenda en el intervalo (0, 1).

En la figura 4.3 se puede ver la arquitectura final base de la red neuronal. A lo largo del proceso de entrenamiento y de obtención de los hiperparámetros se estudian algunas variaciones que consisten en el agregado de capas LSTM con *dropout* en profundidad y en el agregado de neuronas. Con la realización de varias pruebas, se utilizan entre una y nueve capas, y entre 128 y 768 neuronas.

En las secciones posteriores se muestran las formas de entrenamiento y el ajuste de hiperparámetros, como así también las diferentes pruebas realizadas.

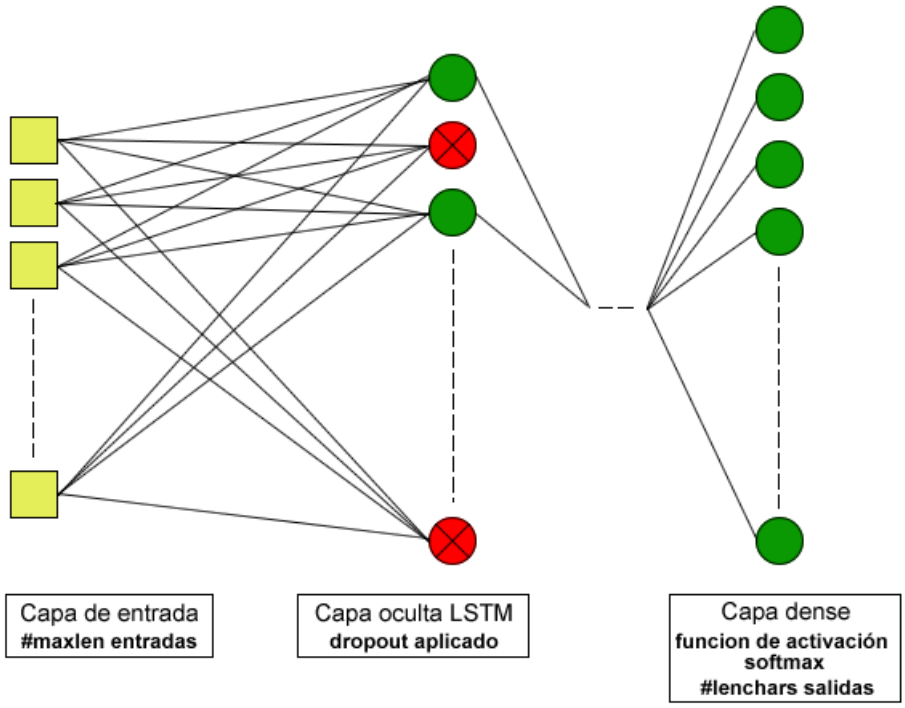

Figura 4.3: Topología de red base. Una capa de entradas, capas ocultas LSTM con la aplicación de dropout y una capa de salida densa con función de activación softmax

4.3. Entrenamiento

Luego de preparar los datos y elegir la configuración de la red neuronal, se comienza con el entrenamiento del modelo. Esto es, obtener los coeficientes a partir de los datos y sus respectivas etiquetas. Para implementar esto, se utiliza Keras [2], una biblioteca modular escrita en Python especialmente diseñada para redes neuronales. Keras utiliza Theano [29] para la definición, optimización y evaluación de expresiones matemáticas.

En la función de entrenamiento, además de los datos y etiquetas, se puede indicar la cantidad de épocas, esto es, cuantas veces serán usados los datos para ajustar los pesos de la red. También se puede fijar cada cuántos ejemplos de entrenamiento se actualiza el gradiente.

Otro parámetro interesante es la *tasa de entrenamiento*, que indica cuán rápido la red neuronal "aprende", es decir, cuán rápido ajusta sus pesos con valores nuevos. Una tasa alta deriva en correcciones grandes, por lo que puede que no se converja al valor óptimo, al ajustarse por demás. Una tasa muy baja, en cambio, lleva a correcciones muy pequeñas que pueden hacer

que, en términos prácticos, nunca se ajusten lo suficiente para alcanzar el valor óptimo.

Unas de las decisiones más importantes que se toma en esta etapa, es determinar qué métrica usar para considerar si luego de transcurrida una época hay una mejora en el entrenamiento. La métrica elegida es el *train accuracy*[1], que nos indica cuán bien se adapta el modelo a los datos de entrenamiento. No se toma en cuenta el *validation accuracy*[2] ya que no nos dice nada sobre la calidad de la música que puede generar el modelo, solo indica cuán bueno es el modelo en predecir salidas sobre datos de entrada que nunca ha visto.

La decisión anterior tiene como consecuencia el riesgo de tener sobreajuste a los datos de entrenamiento. Esto se podría haber mitigado utilizando técnicas de validación cruzada, que permite que los resultados sean independientes de los datos de entrenamiento y evaluación, o *held-out* que, mediante la partición del corpus en dos conjuntos, uno utilizado para el entrenamiento y otro para la validación, minimiza el sobreajuste del modelo a los datos de entrenamiento.

Los tiempos de entrenamiento en pruebas iniciales con una red de hasta 6 capas ocultas varían hasta alcanzar las 46 horas (ver sección 4.4). Por otra parte, el uso de validación cruzada aumenta aún más los tiempos de entrenamiento, con lo cual hace que el uso de esta técnica sea inviable. Dado que el corpus de rock contiene 866 muestras de canciones, realizar un *held-out* tiene como consecuencia disminuir los datos para el entrenamiento, limitando las posibilidades de aprendizaje. De todos modos, se realiza un ajuste analizando las generaciones obtenidas de los modelos, ya que dadas las características del problema, evaluar objetivamente la calidad musical es algo difícil de lograr.

4.3.1. Mejoras realizadas a la etapa de entrenamiento

El entrenamiento de una red neuronal es la etapa que consume más tiempo, sobre todo si se tiene en cuenta que la red en este caso tiene muchas capas y que no se cuenta con un hardware

[1] Precisión con la que el modelo se ajusta a los datos de entrenamiento.

[2] Porcentaje de instancias clasificadas correctamente por el modelo tomando datos de un conjunto de validación.

óptimo para ejecutar algoritmos que requieren un gran poder de cómputo. Es aquí donde se decide usar *callbacks* para introducir algunas mejoras a la función de entrenamiento y de ese modo acortar estos tiempos. Los *callbacks* son una lista de funciones que pueden ser aplicadas en diferentes etapas durante el entrenamiento, sin tener que parar el proceso y volver a empezar. Esto nos permite tener más control sobre esta etapa.

En este caso se decidió monitorear el *train accuracy* al término de cada época. Si no se obtiene una mejora, se reduce la *tasa de entrenamiento*. Con esto lo que se intenta lograr es bajar la velocidad de aprendizaje de las neuronas y de ese modo obtener un entrenamiento más preciso. Este proceso se hace hasta llegar a un valor de *tasa de entrenamiento* mínimo determinado.

En paralelo a la reducción de la tasa de entrenamiento, se usa *early stopping* para parar el entrenamiento en casos donde no hay una mejora en el *train accuracy* durante dos épocas. Esto permite detectar el máximo valor al que puede llegar el *train accuracy*.

4.4. Ajuste de hiperparámetros

Luego de tomadas las decisiones en la etapa de entrenamiento, queda definir los hiperparámetros involucrados en el algoritmo y en la red neuronal, para de ese modo obtener uno o varios modelos candidatos a ser evaluados en la próxima etapa.

Como no existe un procedimiento a seguir, ni valores establecidos como los mejores para los distintos hiperparámetros, se opta por determinarlos empíricamente. Dado que la cantidad tanto de variables como de sus valores es alta, no es factible probar todas las combinaciones posibles. Es por esto que se decide ajustarlos de forma independiente. Como las variables no poseen valores acotados, se dividen en rangos de valores y se seleccionan representantes de cada uno.

Para los hiperparámetros que no poseen valores fijos (ver cuadro 4.1), los valores iniciales elegidos se determinaron en ejecuciones previas con alguno de los corpus intermedios. Dado

- ***cantidad de iteraciones*:** indica cuantas veces se invoca la función de entrenamiento. Su valor se fija en 1 y 10.
- ***cantidad de épocas*:** indica la cantidad de épocas con las que se invoca la función de entrenamiento. Su valor se fija en 100, aunque puede ser menor debido al *early stopping*.
- ***tasa de entrenamiento mínima*:** indica el valor mínimo al que puede llegar la *tasa de entrenamiento* cada vez que se actualiza. Su valor se fija en 0,0001.
- ***factor de reducción*:** es el factor por el cual se reduce *tasa de entrenamiento*, una vez que se detecta que el *train accurracy* no mejoró. Su valor se fija en 0,3.
- ***épsilon*:** valor mínimo que debe mejorar el *train accurracy* de una época a otra. Su valor se fija en 0.0005.
- ***tamaño de ventana*:** indica el tamaño de los ejemplos de entrenamiento.
- ***tasa de entrenamiento*:** indica cuán rápido la red neuronal "aprende".
- ***tamaño de lote*:** indica cada cuantos ejemplos de entrenamiento se actualiza el gradiente.
- ***pasos*:** indica la granularidad con la que se toman los ejemplos de entrenamiento, esto es, cada cuantos caracteres se elige un nuevo ejemplo de entrenamiento.

Cuadro 4.1: Lista de hiperparámetros

que el proceso de construcción del corpus de rock es una tarea que toma mucho tiempo, se prueba al algoritmo en paralelo con una representación del corpus más pequeña, asumiendo que los valores obtenidos son luego extrapolables al corpus de rock. Esas ejecuciones se pueden encontrar en el apéndice A.

Las primeras ejecuciones se realizaron con una red pequeña de una sola capa. Esto es debido a que se decide primero probar a variar algunos hiperparámetros y de ese modo, con una red chica, acortar los tiempos de entrenamiento. Con esto, se toma el riesgo de asumir que los valores obtenidos de los hiperparámetros se pueden extrapolar luego a redes más grandes y también a otro corpus (para más detalles ver apéndice B).

Luego se comienza a probar otras redes, con más capas y más neuronas por capa. Desde ahí en adelante casi todos los hiperparámetros comienzan a tomar un valor fijo (para más detalles ver el apéndice C). En primera instancia se toma la decisión de elegir esos valores de forma objetiva, tomando como métrica el *train accurracy*, asumiendo el riesgo de que los modelos representados por esos hiperparámetros estén sobreajustados. Luego, para refinar los valores de estos hiperparámetros, de forma subjetiva, se comienza a analizar distintas generaciones

obtenidas a partir de los modelos conseguidos. La idea general es obtener varias generaciones a partir de distintas semillas, para luego escuchar la música y leer las letras. Dado que quienes realizan el análisis no son letrados en música, la decisión de qué modelo es mejor no es una decisión experta.

Para terminar el análisis de los valores obtenidos, queda por ver algunas ejecuciones que destacan sobre otras. Estas son las cinco donde el *train accurracy* está por arriba del 0.9 (ver tabla 4.1 y tabla 4.2). En los hiperparámetros, la gran diferencia se encuentra en *pasos* (excepto en la ejecución 17 de la tabla 4.1, donde cambia también el *tamaño de ventana*) donde se prueba con valores más altos, comparado a los demás donde siempre se usa el valor 3. En un principio se podría esperar que los modelos generados aquí sean grandes candidatos a generar música y letra de una calidad aceptable. Pero, analizando algunas generaciones obtenidas con estos modelos se puede apreciar que ocurre todo lo contrario. Esto se podría deber a la presencia de sobreajuste, ya que comparado a otras ejecuciones, en proporción, la granularidad y cantidad de ejemplos de entrenamiento es mucho menor. De esta manera se está consiguiendo una peor generalización de los datos de entrada.

#	*cant. iter.*	*tam. lote*	*tam. ventana*	*tasa de e.*	*pasos*	*acc*	*loss*	*tiempo*
15	10	512	40	0,005	40	0.9997	0.0110	0:25 hs
16	10	512	40	0,005	20	0.9991	0.0156	0:55 hs
17	10	512	195	0,005	195	0.9577	0.2800	0:58 hs

Tabla 4.1: Ajuste de hiperparámetros para una red con una capa LSTM de 128 neuronas y una capa densa con activación Softmax

#	*cant. iter.*	*tam. lote*	*tam. ventana*	*tasa de e.*	*pasos*	*acc*	*loss*	*tiempo*
26	1	512	40	0,002	40	0.9554	0.1559	0:55 hs
27	1	512	40	0,002	20	0.9469	0.1748	2:15 hs

Tabla 4.2: Ajuste de hiperparámetros para una red con 6 capas LSTM de 256 neuronas, más dropout 0.2 en cada LSTM y una capa densa con activación Softmax

En el cuadro 4.2 se muestra un ejemplo con algunas líneas de las letras obtenidas con estos

modelos y en la figura 4.4 una partitura de la música generada donde se puede ver como se repite varias veces la misma nota sin ningún sentido.

```
w: w3 y3u3362En: 2a4e't'M-p4eth odow ast sn ptes cow lao d wagn eo whally
w: 44T: 1he meo't-tocl
he lhinssh,s d(ve dasd
w: der
he ink,my ur coaal lesarse we moay seod logll
k, oi peplgt' troreey
we ci'm mot do glocs mf then shet
ve low you docreln
w: lyyduM: 4/8
```

Cuadro 4.2: Ejemplo de generación de letras en modelo con sobreajuste

Figura 4.4: Melodía en modelos con sobreajuste

En el proceso de análisis de las generaciones, se puede notar que para redes más profundas, a partir de 6 capas LSTM, todos los modelos logran generar cada vez mejor la estructura presentada en el corpus (ver tablas 4.3, 4.4 y 4.5 que muestran estas ejecuciones). Ahora, en ningún caso se obtiene una generación de música de una calidad superior. Algunos modelos logran componer música con pasajes destacables, pero no de forma completa; lo mismo ocurre con las letras. En el cuadro 4.3 se puede ver un ABC de una generación obtenida a partir de un modelo con 6 capas LSTM.

#	*cant. iter.*	*tam. lote*	*tam. ventana*	*tasa de e.*	*pasos*	*acc*	*loss*	*tiempo*
20	1	512	40	0,005	3	0.8468	0.4638	38:17 hs

Tabla 4.3: Ajuste de hiperparámetros para una red con 3 capas LSTM de 768 neuronas, más dropout 0.2 en cada LSTM y una capa densa con activación Softmax

#	cant. iter.	tam. lote	tam. ventana	tasa de e.	pasos	acc	loss	tiempo
21	1	512	40	0,005	3	0.7724	0.7249	11:16 hs
22	1	512	40	0,002	3	0.8485	0.4598	17:35 hs
23	1	512	100	0,002	3	0.8483	0.4616	46:05 hs

Tabla 4.4: Ajuste de hiperparámetros para una red con 6 capas LSTM de 256 neuronas, más dropout 0.2 en cada LSTM y una capa densa con activación Softmax

#	cant. iter.	tam. lote	tam. ventana	tasa de e.	pasos	acc	loss	tiempo
28	1	512	40	0,002	3	0.8756	0.3887	41:00 hs

Tabla 4.5: Ajuste de hiperparámetros para una red con 9 capas LSTM de 512 neuronas, más dropout 0.2 en cada LSTM y una capa densa con activación Softmax

```
X: 1
T: from old hind - day crange the love
M: 4/4
L: 1/8
Q:1/4=1e4
K:G
%%MIDI program 27
A,E A,/2E2E2DB,/2-
D/2=B,/2C/2B,/2 DB/2B,xB,/2 xB,
CD CB, D/2EE3/2x/2[f3/2d3/2B3/2]x/2[c-A-E-]
x[G/2D/2B,/2]x3/2 F,/2x/2[E2C2G,2]E,
w: you know the chas ran,
w: cras
w: she heul take will got i come don't smit for sumhoor
w: everybind the yor machin' go
w: and leads ain't tomd the world up
```

Cuadro 4.3: ABC generado con el corpus de rock

Como última instancia, y dado que gran parte de los resultados conseguidos en las distintas generaciones realizadas son un poco decepcionantes, se decide entrenar dos modelos, uno con el corpus de Nottingham (tabla 4.6) y otro con el corpus de Nottingham con letras (tabla 4.7). Los valores de los hiperpárametros se eligen en base a las ejecuciones anteriores para el corpus

de rock. Se entrena una red de 6 capas y 256 neuronas, y no una más grande, para obtener resultados rápidamente.

#	*cant. iter.*	*tam. lote*	*tam. ventana*	*tasa de e.*	*pasos*	*acc*	*loss*	*tiempo*
31	1	512	40	0,002	3	0.8913	0.3207	3:40 hs

Tabla 4.6: Entrenamiento para una red con 6 capas LSTM de 256 neuronas, más dropout 0.2 en cada LSTM y una capa densa con activación Softmax

#	*cant. iter.*	*tam. lote*	*tam. ventana*	*tasa de e.*	*pasos*	*acc*	*loss*	*tiempo*
32	1	512	40	0,002	3	0.7945	0.6216	4:35 hs

Tabla 4.7: Entrenamiento para una red con 6 capas LSTM de 256 neuronas, más dropout 0.2 en cada LSTM y una capa densa con activación Softmax

Nuevamente, analizando la música generada obtenida a partir de las ejecuciones de las tablas 4.6 y 4.7, se logra apreciar que los resultados conseguidos en el apartado de la música son muy superiores a los obtenidos con el corpus de rock. Esta diferencia se le puede adjudicar principalmente al corpus, no al algoritmo o al proceso de ajuste de los hiperparámetros. En cuanto a las letras, en cambio, se conservan los pobres resultados.

La partitura en la figura 4.5 corresponde a un tema musical generado con el corpus de Nottingham.

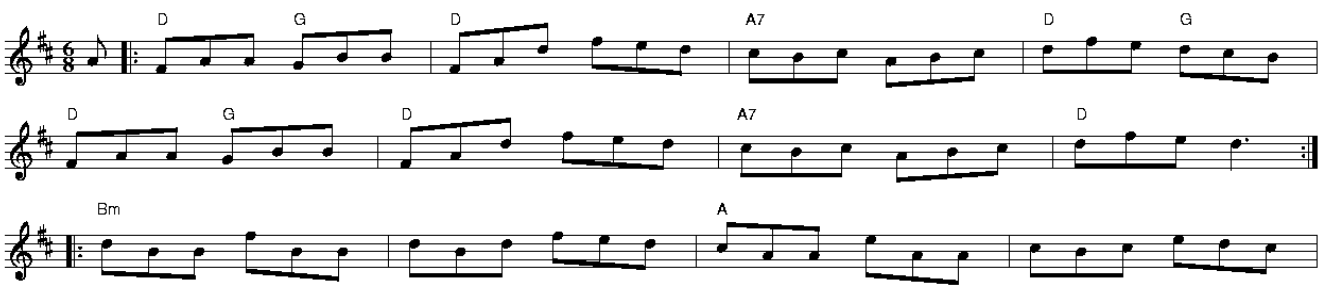

Figura 4.5: Melodía en modelos entrenados con el corpus de Nottingham

Capítulo 5

Evaluación

La evaluación del algoritmo construido, en conjunto con la creación de un corpus de música, forman parte de una de las etapas más importantes del proyecto. Es aquí donde se puede apreciar si los mecanismos usados y decisiones tomadas previamente han sido de utilidad para cumplir con los objetivos planteados.

Para la evaluación se utilizan generaciones de modelos entrenados con dos conjuntos de canciones: el corpus creado durante el proyecto (al que de ahora en más nos referiremos como *corpus de rock*) y otros con el corpus de Nottingham.

Se decide no incorporar las letras generadas al test, debido a que su calidad no es buena. Es muy fácil notar que no las creó un humano, por lo que de evaluarse en conjunto con la música podrían sesgar los resultados del test. Los resultados de las letras generadas son analizados en la sección 5.1.

Para poder evaluar de forma más objetiva la música generada, se decide realizar una variación del test de Turing, en el que se usan canciones de los corpus y una selección de música generada por algunos modelos entrenados. Esto se detalla en la sección 5.2, donde también se analizan los resultados obtenidos.

5.1. Análisis de las letras generadas

La finalidad del proyecto es la generación de música y letras utilizando aprendizaje profundo. Durante todo el proyecto se busca lograr este objetivo. Como fue descrito en capítulos anteriores, se realizan varios pasos para que los corpus incluyan letras, por más que música y letras no estén sincronizadas.

Las letras generadas no fueron incluidas en ningún momento durante la evaluación por la sencilla razón de que a simple vista pueden diferenciarse de las que generaría un humano.

```
the ler in the funksting fetpling sfilk.
the will your she won't have love and sprung, yeah
to my all nike gone
when i call to gar do you could,
huve that was hings come to me
no i cin pace, sirging to ginl,
```

Cuadro 5.1: Ejemplo de letra generada

En el cuadro 5.1 se ve un ejemplo en el cual, si bien hay palabras que se logran reproducir de buena forma, muchas son inexistentes y no se ve ningún tipo de coherencia, aunque sí, con algo de imaginación, un poco de estructura.

Las razones por las que la generación de letras no es exitosa son varias. Una de las principales es la escasa cantidad de letras que se utilizan durante el entrenamiento. Cada canción cuenta solamente con 6 líneas de letras, unas 5186 en total. Si bien parece bastante información, cada línea suele ser corta, con un promedio de 34 caracteres, incluyendo espacios y puntuación. A esto se le debe sumar que la totalidad del texto es inconexo, habiendo además secciones de música y letra de forma salteada, dificultando el aprendizaje. La principal razón por la que no se aumenta la cantidad de letras por canción son los largos tiempos de entrenamiento.

Se considera realizar entrenamientos por separado para buscar mejores resultados, entrenando redes distintas para música y letra. Esto se descarta, ya que se aleja del objetivo del proyecto, que es generar música y letra en conjunto.

5.2. Evaluación de la música

La evaluación de la música generada es difícil de realizar de forma objetiva. No existe una medida numérica o una escala en donde poder ubicar los resultados obtenidos. Al tratarse de música, no es simple establecer que lo que se genera es bueno, malo o regular, sin que gustos o subjetividades entren en juego. La música generada, además, depende en gran medida del corpus utilizado para el entrenamiento, por lo que requiere de un contexto.

Para lidiar con esto y aun así lograr una evaluación de resultados lo más objetiva posible, se decide realizar un Test de Turing musical, el cual cuenta con canciones generadas por modelos entrenados en este proyecto, tomando como base el corpus de rock y el de Nottingham.

5.2.1. Qué es un test de Turing

El test de Turing, ideado por Alan Turing en 1950, es un test para probar las habilidades de una máquina de mostrar un comportamiento equivalente o indistinguible del de un humano. Turing, en "Computing Machinery and Intelligence" [30], replantea la pregunta "¿Pueden las máquinas pensar?" al proponer la superación de lo que hoy conocemos como el test de Turing.

El test, originalmente llamado "imitation game", consiste en que un juez intente distinguir cuál de dos interlocutores es humano y cuál una máquina. Para esto, el juez entabla una conversación con ambos, de forma independiente, mediante una interfaz (habla de texto escrito a máquina) que aísle a ambos de características fácilmente distinguibles, como la voz. De superarse este test, se daría una condición necesaria de comportamiento inteligente.

Turing predijo que para el año 2000, las máquinas podrían pasar el test a un nivel de sofisticación razonable. En particular, que el juez promedio no sería capaz de identificar a una máquina correctamente más del 70 % del tiempo, luego de una conversación de 5 minutos [26].

5.2.2. Test de Turing musical

Formalmente, el test planteado no es el clásico Test definido por Turing, ya que aquí no hay una interacción juez-máquina. Además hay diferencias en cuanto a los jueces, debido a que en este test hay jueces que saben de música y jueces que no, ya que se intenta evaluar si el algoritmo supera el test con el público en general.

El test se realiza durante una semana, a través de una encuesta en línea. Además del propio test, se solicita completar algunos datos para poder realizar un mejor análisis: edad y género, y conocimiento sobre teoría de la música, con "Nada", "Poco", "Regular" y "Mucho" como opciones.

El test se divide en dos secciones iguales, cada una en base a un conjunto de canciones distinto. La primera se basa en el corpus de rock y la segunda en el de Nottingham. Cada sección brinda un texto introductorio al corpus y una canción extraída de él, marcada como tal, a modo de muestra sobre lo que escucharían a continuación.

En cada sección se presentan 6 canciones para que sean escuchadas, sobre las cuales se solicita responder la siguiente pregunta: *¿Quién cree que compuso esta canción?*. Las opciones de respuesta son las siguientes:

- *Seguro fue una máquina*
- *Creo que fue una máquina*
- *Creo que fue un humano*
- *Seguro fue un humano*

Si bien no se explicita en el test, en cada sección hay tres canciones pertenecientes al corpus y tres generadas de forma automática. El orden de las canciones se define de forma aleatoria.

Al generar con el corpus de rock, se suelen encontrar varios momentos en los que la calidad de

lo generado es buena, pero estos suelen ser cortos y muchas veces rodeados de cambios muy bruscos de ritmo o saltos de escala fuertes, que hacen notorio el hecho de no ser compuestas por un humano. Esto impide hacer una selección aleatoria dentro de una generación larga. Por eso la elección de las canciones generadas para utilizar en el test tiene criterios distintos para cada corpus.

La decisión que se toma, para el corpus de rock, es realizar recortes pero de forma manual, manteniendo siempre la integridad de lo generado. Es decir, quitando partes de lo generado al principio y al final únicamente, sin quitar, corregir o agregar nada en ninguna posición dentro de lo seleccionado. Se realiza una primera preselección de 34 canciones generadas a partir del corpus de rock, todas de entre 10 y 25 segundos, tomadas de generaciones realizadas con 10 modelos distintos. En todas ellas se ve algo interesante en lo generado. De esa preselección, se eligen tres muestras finales para realizar el test, todas provenientes de modelos distintos.

Las canciones elegidas en base a generaciones del corpus de Nottingham son todas obtenidas de una única generación corta, de una longitud unas 10 veces menor a lo generado con el corpus de rock. El recorte también fue realizado a mano, pero en este caso con el principal objetivo de mantener el test con una duración corta y similar a la sección anterior.

Tanto las canciones generadas como las muestras del corpus que se incluyen en el test, son creadas a partir de instrumentos sintetizados, por lo que ninguna canción es interpretada por un instrumento real.

5.2.3. Sobre los jueces

Se obtienen respuestas de 206 jueces distintos. Las características generales de los jueces, de acuerdo a las respuestas brindadas para las preguntas mencionadas anteriormente se pueden ver en las figuras 5.1, 5.2 y 5.3.

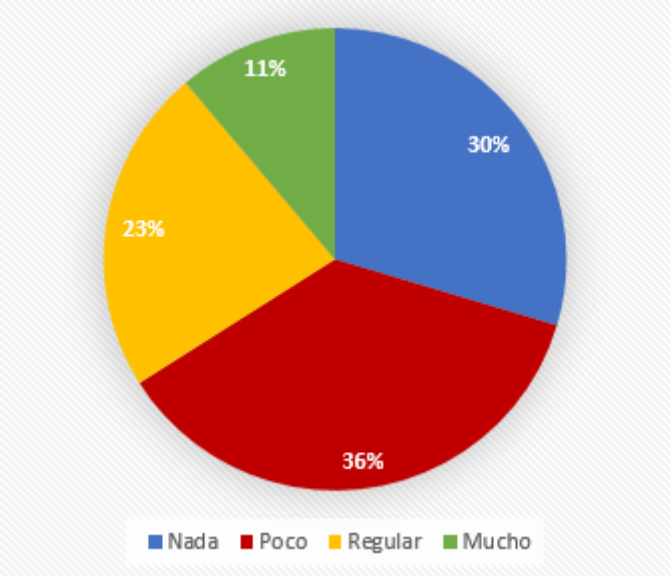

Figura 5.1: Distribución de jueces según conocimiento musical

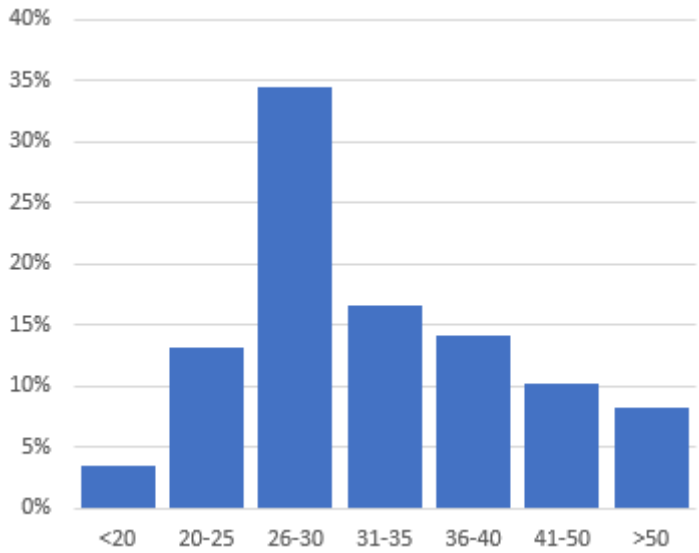

Figura 5.2: Distribución de jueces según rangos etarios

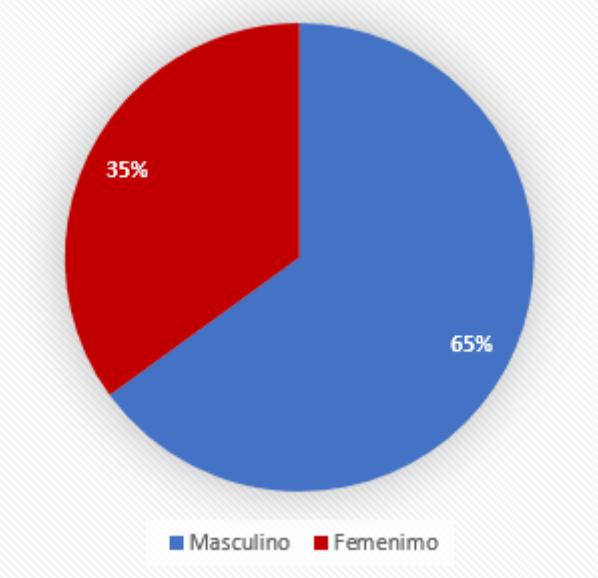

Figura 5.3: Distribución de jueces según género

5.2.4. Análisis sobre la totalidad de las respuestas

Habiendo 206 jueces y 12 preguntas (respecto a canciones) respondidas por cada uno, hay un total de 2472 respuestas. Se puede ver su distribución en la figura 5.4.

Se nota en general una leve inclinación (54 %) hacia etiquetar canciones como compuestas por humanos, y mayor seguridad al afirmar que es un humano respecto a afirmar que es una máquina. Las respuestas con poca seguridad son mayoría (un 66 %).

En promedio, cada juez acertó 6.49 preguntas de las 12 propuestas. Todos los jueces lograron identificar correctamente al menos 2 canciones (como máquina o humano). Ninguno logró identificarlas correctamente a todas.

El porcentaje de jueces para cada número de respuestas correctas se distribuye como se ve en la figura 5.5.

Restringiendo el análisis a las preguntas sobre canciones compuestas por máquina, el acierto es de un 50 %, 6 preguntas en las 12 planteadas. Todos los jueces detecten al menos una canción correctamente como máquina, 11 de ellos etiquetan correctamente a las 6.

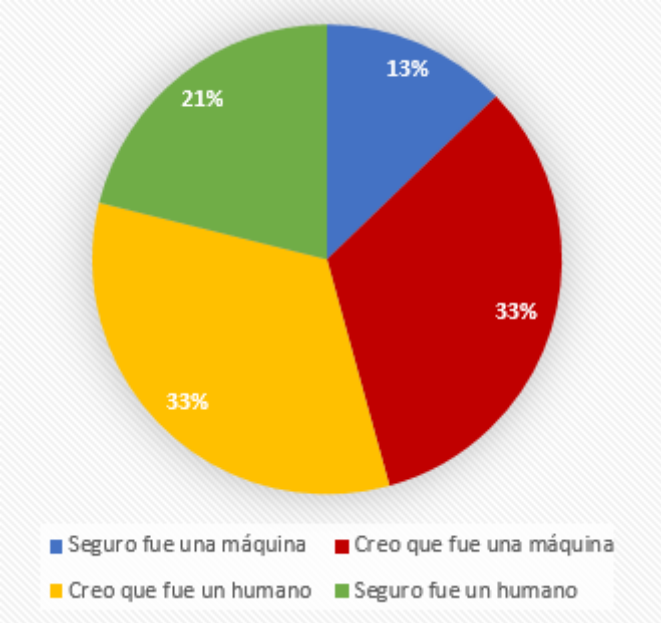

Figura 5.4: Distribución de las 2472 respuestas entre las 4 opciones dadas

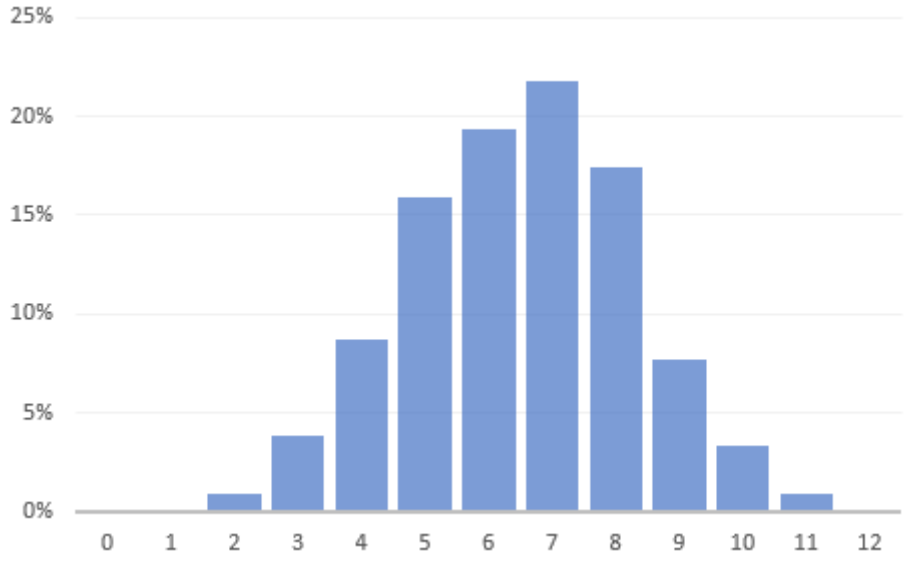

Figura 5.5: Porcentaje de jueces para cada número de aciertos en total

El porcentaje de jueces para cada número de respuestas correctas se distribuyeron como en la figura 5.6.

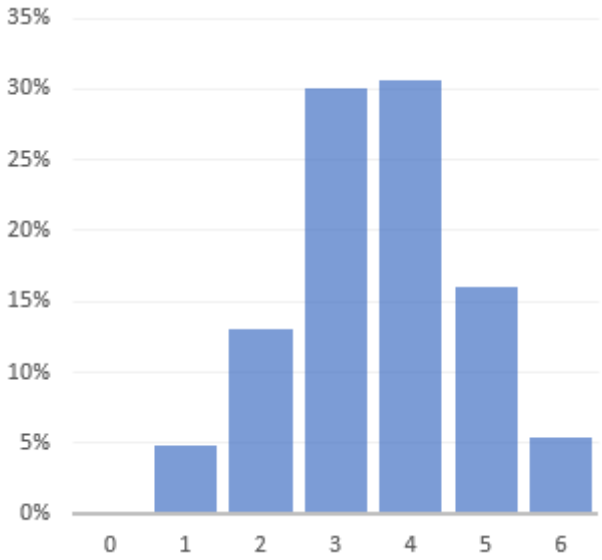

Figura 5.6: Porcentaje de jueces para cada número de aciertos en canciones de máquina

La matriz de confusión para la totalidad de las respuestas dadas, se muestra en la tabla 5.1.

		Respuesta obtenida	
		Humano	Máquina
Realidad	Humano	722	514
	Máquina	620	616

Tabla 5.1: Matriz de confusión para respuestas.

Se puede notar que las canciones generadas por máquina dan más lugar a confusión, al momento de etiquetar, que las generadas por humano. También se ve que es más frecuente la distinción correcta de un humano que de una máquina.

En la tabla 5.2, se puede ver un resumen de la totalidad de los resultados más importantes.

	% de acierto
Totalidad de preguntas	54 %
Canciones compuestas por máquina	50 %
Canciones compuestas por humanos	58 %
Canciones en el corpus de rock	54 %
Canción #1, corpus de rock (compuesta por máquina)	68 %
Canción #2, corpus de rock (compuesta por máquina)	43 %
Canción #4, corpus de rock (compuesta por máquina)	53 %
Canciones en el corpus de rock compuestas por máquina	55 %
Canciones en el corpus de rock compuestas por humano	54 %
Canciones en el corpus de Nottingham	54 %
Canción #1, corpus de Nottingham (compuesta por máquina)	40 %
Canción #4, corpus de Nottingham (compuesta por máquina)	41 %
Canción #6, corpus de Nottingham (compuesta por máquina)	53 %
Canciones en el corpus de Nottingham compuestas por máquina	45 %
Canciones en el corpus de Nottingham compuestas por humano	63 %

Tabla 5.2: Resumen de los resultados del Test de Turing Musical.

5.2.5. Análisis sobre canciones generadas automáticamente

Este análisis es el más relevante para el proyecto, ya que se evalúan las canciones que son generadas y cómo son las respuestas para este tipo de canciones, sin importar las que se seleccionan del corpus. De las 1236 respuestas correspondientes a estas canciones, vemos la distribución en la Figura 5.7.

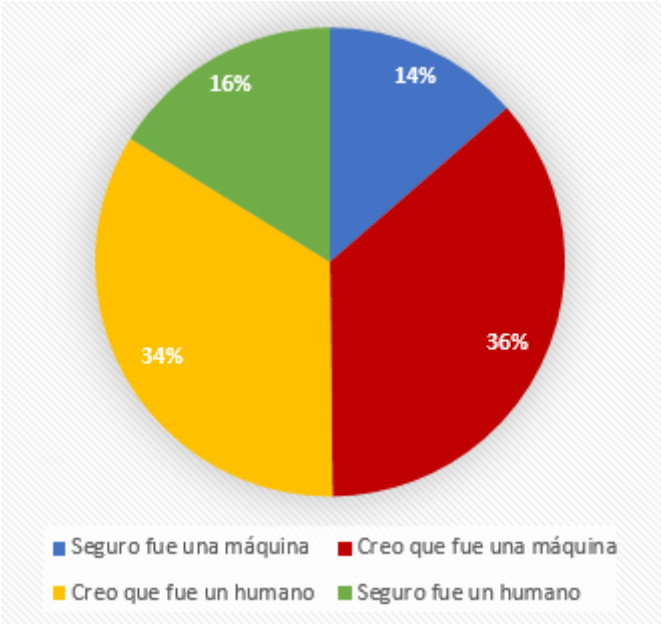

Figura 5.7: Distribución de respuestas sobre canciones generadas

Se nota en general una división al 50 % al elegir una canción como compuesta por humanos y máquinas. Es prácticamente un empate incluso al analizar considerando un solo nivel de seguridad. Hay poca variación al afirmar que un tema es humano o máquina con seguridad (con un 2 % favoreciendo la afirmación "humano") al igual que al afirmarlo con un grado de duda (con un 2 % favoreciendo el etiquetado como "máquina"). Dentro de las canciones de máquina, las respuestas con duda son abrumadora mayoría, llegando al 70 %.

5.2.5.1. Según conocimiento musical

Las respuestas son consistentes a través de los cuatro grupos. Los aciertos al etiquetar una canción como máquina varían entre un 49 % y un 52 %, siendo los de mayor conocimiento musical los que más aciertan. Si bien hay algunas excepciones, en general se ve una cierta linealidad entre aciertos y nivel de conocimiento en ambas secciones del test.

Hay mayor seguridad al etiquetar una canción como compuesta automáticamente al haber mayor conocimiento musical, pero también hay un porcentaje más alto de respuestas erróneas con un mayor grado de confianza.

Las respuestas con duda (*creo que es una máquina* y *creo que es un humano*) se dan en un mayor porcentaje entre los jueces de menor conocimiento musical (71 % en "nada", 73 % en "poco") y disminuyen entre los de mayor conocimiento (69 % en "regular", 59 % en "mucho").

5.2.5.2. Según grupos etarios

Agrupando a los jueces según su edad, se ven resultados homogéneos. Los jueces mayores a 50 años son quienes tienen el menor porcentaje de aciertos, llegando apenas al 42 %, aunque ese grupo no alcanza a ser el 10 % del total de los jueces. Los grupos etarios que no llegan a la media son los de los extremos, estos son los menores a 20 y los mayores de 50 años. Y el grupo que tuvo el mayor porcentaje de aciertos es el comprendido entre 31 y 35 años, solamente 3 % por encima de la media. La distribución se puede ver en la figura 5.8.

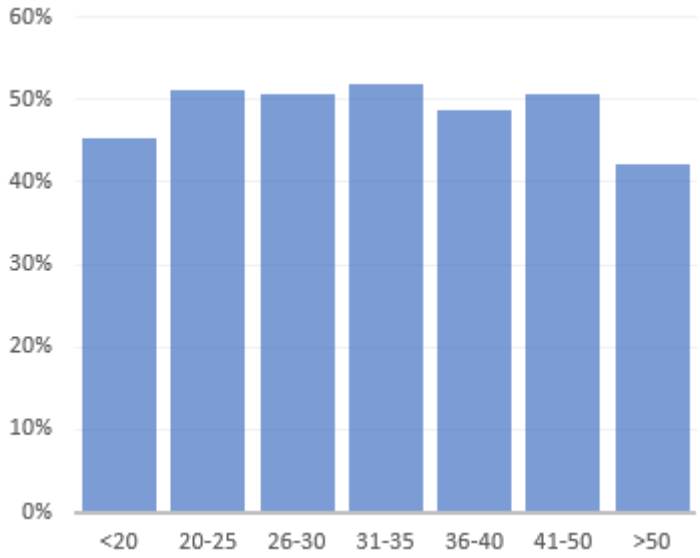

Figura 5.8: Porcentaje de acierto según grupos etarios

Si este mismo análisis se hace poniendo foco en la generalidad de las canciones, se ve una foto similar, pero con aumento en aciertos para todos los grupos, variando entre un 2 % y un 9 %.

5.2.5.3. Diferencias entre los corpus

Comparando los resultados obtenidos por los jueces en una sección y otra del test, algunas diferencias son notorias. Como se puede ver en el cuadro 5.2, para el corpus de Nottingham el porcentaje de acierto en canciones generadas por máquina (45 %) es considerablemente menor que en el corpus de rock (55 %).

La primera canción de la sección de las generadas a partir del corpus de rock, es la que peor resultados obtuvo: es catalogada correctamente por un 68 % de los jueces. La que logró engañar a la mayor cantidad de jueces, por el contrario, se generó a partir del corpus de Nottingham: sólo un 40 % de los jueces logran categorizarla correctamente.

5.2.6. Acuerdo entre jueces

Resulta interesante analizar cuál es el grado de acuerdo que existe entre los jueces al categorizar una canción entre las 4 opciones posibles. Para esto se utilizó la medida kappa de Fleiss.

La medida kappa de Fleiss [8] es un indicador del nivel de acuerdo sobre una clasificación, en contraste al que sucedería por azar. Si hay acuerdo total, el valor es máximo y será 1. Si el nivel de acuerdo es el mismo que el esperable por el azar, el valor será 0. Si el nivel de acuerdo es peor, será menor que 0.

Se define de la siguiente manera:

$$\kappa = \frac{\tilde{P} - \tilde{P}_e}{1 - \tilde{P}_e}$$

Sea N el total de canciones calificadas, n la cantidad de respuestas obtenidas y k el número de categorías. Sea n_{ij} el número de jueces que asignaron a la canción i en la categoría j.

El primer paso en el cálculo es obtener p_j, la proporción en la cual la categoría j fue utilizada

con respecto a la totalidad:

$$p_j = \frac{1}{Nn} \sum_{i=1}^{N} n_{ij}, 1 = \sum_{j=1}^{k} p_j$$

Luego se calcula P_i, el grado de acuerdo para la canción i:

$$P_i = \frac{1}{n(n-1)} \sum_{j=1}^{k} n_{ij}(n_{ij} - 1)$$

Una vez obtenido P_i, calculamos a $\tilde{P}$ como el promedio de los P_i:

$$\tilde{P} = \frac{1}{N} \sum_{i=1}^{N} P_i$$

Y obtenemos $\tilde{P}_e$ de la siguiente forma:

$$\tilde{P}_e = \sum_{j=1}^{k} {p_j}^2$$

Para calcular la medida kappa de Fleiss se agrupan las cuatro categorías en dos que representan la elección de máquina o humano.

La medida kappa obtenida es de 0.056. Es un número muy bajo y cercano al cero, siendo coherente con los resultados encontrados, ya que no hay una polarización en las respuestas, sino un reparto relativamente equitativo entre las opciones dadas.

Capítulo 6

Conclusiones

El presente proyecto se concentra en estudiar la generación de música y letra con aprendizaje profundo. Se comienza por la construcción de un corpus con características que lo hagan útil para el objetivo planteado, utilizando herramientas que facilitan esta tarea. Además, se define una estrategia de generación, como así también una estrategia de evaluación del desempeño del algoritmo de forma empírica.

Al momento de iniciar el proyecto algunas pocas investigaciones se interesaban sobre el tema, pero en el transcurso fueron apareciendo nuevos aportes al área, lo cual indica un claro interés de la comunidad científica. Pocos de los trabajos similares que se nombran en este documento apunta a la generación de música y letra a la vez.

Se construyen tres corpus a lo largo del proyecto de los cuales solamente uno es utilizado por el algoritmo final presentado. Los primeros dos son versiones que permiten identificar qué características se tienen que mejorar y cuáles mantener, para perfeccionar el aprendizaje. A su vez, se utiliza un cuarto corpus de referencia, construido por la Universidad de Nottingham.

Como forma de evaluar el trabajo, al no encontrarse medidas objetivas para evaluar el desempeño del algoritmo, se opta por hacer un *Test de Turing Musical*. Se comprueba que en general las personas evaluadas por el test están confundidas en cuanto a quién creó la música. Esto es

un resultado positivo para el proyecto, pero se debe tener en cuenta el contexto en el que se realiza la evaluación: las canciones compuestas por humanos son también representadas a través de archivos MIDI que tienen un procesado previo, poseen un solo instrumento y una duración de entre 30 y 45 segundos. Además, ninguna canción es interpretada por un instrumento real, sino generadas por un sintetizador.

La evaluación mediante el test de Turing musical demuestra un mejor desempeño para los modelos entrenados con el corpus de *Nottingham*, lo que confirma la suposición de que el corpus construido es sin dudas mejorable. Los resultados expuestos en el capítulo 5 avalan lo antedicho.

Se considera como principal línea de trabajo a futuro la construcción o mejora del corpus propuesto, con una cantidad mayor de canciones, pero sobre todo, con una mayor homogeneidad en ellas. En principio, las canciones utilizadas son de género rock, pero hay que tener en cuenta que el género es muy heterogéneo y que los MIDI obtenidos pueden haber sido etiquetados equivocadamente. En algunos casos se encontraron canciones que claramente no pertenecían al género, entorpeciendo el aprendizaje del algoritmo. La posibilidad de que las canciones utilizadas tengan más de un instrumento plantea un desafío mayor, que podría lograr que el resultado sonoro sea más rico.

Aplicar técnicas como held-out o validación cruzada, como forma de mitigar el riesgo de sobreajuste asumido, es otra de las mejoras que se le puede hacer al proyecto. Como se explicó en la sección 4.3, aquí no se aplican estas técnicas debido a los altos tiempos de entrenamiento de los modelos y al reducido tamaño de los corpus.

Si bien los resultados obtenidos en el test no son excepcionales, la utilización de aprendizaje profundo y redes *LSTM* como técnica de generación de música parecen ser un buen enfoque para abordar la temática. Se estima que las mejoras en el corpus construido repercutirán positivamente en los resultados obtenidos.

Apéndice A

Primer ajuste de hiperparámetros

En el presente anexo se detalla las primeras ejecuciones realizadas para ajustar algunos hiperparámetros.

En todos los casos presentados, se usa una red con 1 capa LSTM de 128 neuronas más una capa Dense con activación Softmax. Además, los hiperparámetros *tamaño de ventana*, *pasos* y *tasa de entrenamiento mínima* valen 40, 3 y 0.0001 respectivamente.

En las ejecuciones de las tablas A.1 y A.2 no se realiza ninguna mejora en el entrenamiento (ver 4.3.1). La tasa de entrenamiento usada es 0.01.

cant.iter.	***cant.épocas***	***tam. lote***	***acc***	***loss***	***tiempo***
10	20	128	0.8108	0.5825	50:54 hs
10	20	32	0.6593	1.1485	53:00 hs

Tabla A.1: Ajuste de hiperparámetros tomando como entrada el corpus de Notingham

cant. iter.	*cant. épocas*	*tam. lote*	*acc*	*loss*	*tiempo*
10	20	128	0.6957	0.9905	11:00 hs
10	20	256	0.7378	0.8305	53:00 hs
10	20	4096	0.7609	0.743	60:00 hs

Tabla A.2: Ajuste de hiperparámetros tomando como entrada el corpus de Notingham con letras

cant. iter.	*cant. épocas*	*tam. lote*	*t.e.*	*factor red.*	*epsilon*	*acc*	*loss*	*tiempo*
10	20	256	0.05	0.5	0.002	0.912	0.329	15:55 hs
10	20	1024	0.01	0.5	0.002	0.472	1.845	23:15 hs
5	40	256	0.01	0.5	0.002	0.912	0.329	15:55 hs
2	100	256	0.01	0.5	0.002	0.998	0.0219	6:27 hs
10	20	4096	0.01	0.5	0.002	0.4992	1.6778	6:27 hs

Tabla A.3: Ajuste de hiperparámetros tomando como entrada 200 canciones del corpus de Nottingham con letras

cant. iter.	*cant. épocas*	*tam. lote*	*t.e.*	*factor red.*	*epsilon*	*acc*	*loss*	*tiempo*
5	100	256	0.05	0.3	0.0005	0.6107	1.2681	42:30 hs
5	100	512	0.05	0.3	0.0005	0.6657	1.0639	35:00 hs
5	100	1024	0.05	0.3	0.0005	0.5602	1.4213	20:57 hs
10	100	512	0.03	0.3	0.0005	0.8143	0.6006	3:40 hs
1	100	512	0.03	0.3	0.002	0.7793	0.7793	26:50 hs

Tabla A.4: Ajuste de hiperparámetros tomando como entrada el corpus de Nottingham con letras con un procesado previo

El procesado previo realizado al corpus de Nottingham con letras, implica quitar caracteres especiales para disminuir el abecedario.

cant. iter.	*cant. épocas*	*tam. lote*	*t.e.*	*factor red.*	*epsilon*	*acc*	*loss*	*tiempo*
2	100	256	0.01	0.5	0.002	0.9906	0.0427	19:25 hs

Tabla A.5: Ajuste de hiperparámetros tomando como entrada el corpus de melodías de juegos con letras

Apéndice B

Ejecuciones para una red de una capa

#	*cant. iter.*	*tam. lote*	*tam. ventana*	*tasa de e.*	*pasos*	*acc*	*loss*	*tiempo*
1	10	512	40	0,005	3	0,8801	0.4016	9:40 hs
2	10	256	40	0,005	3	0.8784	0.4044	12:00 hs
3	10	512	40	0,01	3	0.8764	0.4093	10:18 hs
4	10	512	40	0,015	3	0.8484	0.4879	9:20 hs
5	10	256	40	0,01	3	0.8455	0.5005	9:40 hs
6	10	512	40	0,02	3	0.8263	0.5509	7:37 hs
7	10	256	40	0,015	3	0.8084	0.6137	6:00 hs
8	10	128	40	0,01	3	0.7812	0.7160	19:58 hs
9	10	256	40	0,02	3	0.7256	0.8906	4:30 hs
10	10	128	40	0,015	3	0.6899	1.0440	4:57 hs
11	1	512	195	0,005	3	0.8765	0.4097	18:20 hs
12	1	256	195	0,005	3	0.8698	0.4295	21:40 hs
13	1	512	195	0,01	3	0.8568	0.4648	17:20 hs
14	1	256	195	0,01	3	0.8480	0.4944	19:30 hs
15	10	512	40	0,005	40	0.9997	0.0110	0:25 hs
16	10	512	40	0,005	20	0.9991	0.0156	0:55 hs
17	10	512	195	0,005	195	0.9577	0.2800	0:58 hs

Tabla B.1: Ajuste de hiperparámetros para una red con una capa LSTM de 128 neuronas y una capa densa con activación Softmax

Apéndice C

Ejecuciones para redes de varias capas

#	*cant. iter.*	*tam. lote*	*tam. ventana*	*tasa de e.*	*pasos*	*acc*	*loss*	*tiempo*
18	1	512	40	0,005	3	0.8117	0.6546	5:23 hs

Tabla C.1: Ajuste de hiperparámetros para una red con 3 capas LSTM de 256 neuronas, más dropout 0.2 en cada LSTM y una capa densa con activación Softmax

#	*cant. iter.*	*tam. lote*	*tam. ventana*	*tasa de e.*	*pasos*	*acc*	*loss*	*tiempo*
19	1	512	40	0,005	3	0.8082	0.6067	13:00 hs

Tabla C.2: Ajuste de hiperparámetros para una red con 3 capas LSTM de 512 neuronas, más dropout 0.2 en cada LSTM y una capa densa con activación Softmax

#	*cant. iter.*	*tam. lote*	*tam. ventana*	*tasa de e.*	*pasos*	*acc*	*loss*	*tiempo*
20	1	512	40	0,005	3	0.8468	0.4638	38:17 hs

Tabla C.3: Ajuste de hiperparámetros para una red con 3 capas LSTM de 768 neuronas, más dropout 0.2 en cada LSTM y una capa densa con activación Softmax

#	cant. iter.	tam. lote	tam. ventana	tasa de e.	pasos	acc	loss	tiempo
21	1	512	40	0,005	3	0.7724	0.7249	11:16 hs
22	1	512	40	0,002	3	0.8485	0.4598	17:35 hs
23	1	512	100	0,002	3	0.8483	0.4616	46:05 hs

Tabla C.4: Ajuste de hiperparámetros para una red con 6 capas LSTM de 256 neuronas, más dropout 0.2 en cada LSTM y una capa densa con activación Softmax

#	cant. iter.	tam. lote	tam. ventana	tasa de e.	pasos	acc	loss	tiempo
24	1	512	40	0,005	3	0.6932	0.9904	17:40 hs
25	1	512	15	0,002	3	0.7782	0.6791	11:00 hs

Tabla C.5: Ajuste de hiperparámetros para una red con 9 capas LSTM de 256 neuronas, más dropout 0.2 en cada LSTM y una capa densa con activación Softmax

#	cant. iter.	tam. lote	tam. ventana	tasa de e.	pasos	acc	loss	tiempo
28	1	512	40	0,002	3	0.8756	0.3887	41:00 hs

Tabla C.6: Ajuste de hiperparámetros para una red con 9 capas LSTM de 512 neuronas, más dropout 0.2 en cada LSTM y una capa densa con activación Softmax

#	cant. iter.	tam. lote	tam. ventana	tasa de e.	pasos	acc	loss	tiempo
29	1	512	40	0,005	3	0.1586	2.9730	15:00 hs
30	1	512	40	0,005	3	0.1275	3.2878	5:40 hs

Tabla C.7: Ajuste de hiperparámetros para una red con 12 capas LSTM de 256 neuronas, más dropout 0.2 en cada LSTM y una capa densa con activación Softmax

La tabla C.8 corresponde a una ejecución realizada con el corpus de Nottingham y la C.9 corresponde a una ejecución realizada con el corpus de Nottigham con letras.

#	cant. iter.	tam. lote	tam. ventana	tasa de e.	pasos	acc	loss	tiempo
31	1	512	40	0,002	3	0.8913	0.3207	3:40 hs

Tabla C.8: Entrenamiento para una red con 6 capas LSTM de 256 neuronas, más dropout 0.2 en cada LSTM y una capa densa con activación Softmax

#	*cant. iter.*	*tam. lote*	*tam. ventana*	*tasa de e.*	*pasos*	*acc*	*loss*	*tiempo*
32	1	512	40	0,002	3	0.7945	0.6216	4:35 hs

Tabla C.9: Entrenamiento para una red con 6 capas LSTM de 256 neuronas, más dropout 0.2 en cada LSTM y una capa densa con activación Softmax

Bibliografía

[1] ABC. *ABC notación: definición de campos*, (2016). Disponible en: `http://abcnotation.com/wiki/abc:standard:v2.1#information_fields` (accedido en Agosto de 2017).

[2] Chollet, F., et al. Keras. `https://github.com/fchollet/keras`, 2015.

[3] Craig, S. S. *Standard MIDI File Structure*. Stanford University, California, USA, (2005). Disponible en: `https://ccrma.stanford.edu/~craig/14q/midifile/MidiFileFormat.html` (accedido en Agosto de 2017).

[4] Daniel, J. *Composing music with recurrent neural networks*, (2015). Disponible en: `http://www.hexahedria.com/2015/08/03/composing-music-with-recurrent-neural-networks/` (accedido en Agosto de 2017).

[5] Douglas Eck, J. S. *A First Look at Music Composition using LSTM Recurrent Neural Networks*. Technical Report No. IDSIA-07-02. IDSIA / USI-SUPSI. Instituto Dalle Molle di studi sull' intelligenza artificiale. Galleria 2 CH-6900 Manno, Switzerland, (2002).

[6] Du, T., and Shanker, V. K. *Deep Learning for Natural Language Processing*. Department of Computer and Information Sciences, University of Delaware, Newark, DE 19711, (2013).

[7] Finn, C., Tan, X. Y., Duan, Y., Darrell, T., Levine, S., and Abbeel, P. *Deep Spatial Autoencoders for Visuomotor Learning*. University of California, Berkeley,, (2015).

[8] Fleiss, J. L. *Measuring nominal scale agreement among many raters.* Psychological bulletin 76.5, pág. 378., (1971).

[9] Foxley, E. *Nottingham Music Database.* University of Nottingham, (2001). Disponible en: `http://www.chezfred.org.uk/University/music/index.htm` (accedido en Agosto de 2017).

[10] Graves, A., Mohamed, A.-r., and Hinton, G. *SPEECH RECOGNITION WITH DEEP RECURRENT NEURAL NETWORKS.* Pusblisher: Department of Computer Science, University of Toronto, (March 22, 2013).

[11] Hochreiter, S. *Recurrent Neural Net and Vanishing Gradient.* Institut für Informatik Technische Universität München D-80290 München, Germany. International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems 6(2):107- 116, (1998).

[12] Hochreiter, S., and Schmidhuber, J. *LONG SHORT-TERM MEMORY.* Neural Computation 9(8):1735-1780. Fakultat fur Informatik Technische Universitat Munchen 80290 Munchen. IDSIA Corso Elvezia 36 6900 Lugano Switzerland Germany, (1997).

[13] Huval, B., Wang, T., Tandon, S., Kiske, J., Song, W., Pazhayampallil, J., Andriluka, M., Rajpurkar, P., Migimatsu, T., Cheng-Yue, R., Mujica, F., Coates, A., and Ng, A. Y. *An Empirical Evaluation of Deep Learning on Highway Driving.* Stanford University, (2015).

[14] Johnson, J., Karpathy, A., and Fei-Fei, L. *Densecap: Fully convolutional localization networks for dense captioning.* Presented at CVPR 2016 (oral). Department of Computer Science, Stanford University, (2015).

[15] Karpathy, A. *The Unreasonable Effectiveness of Recurrent Neural Networks*, (2015). Disponible en: `http://karpathy.github.io/2015/05/21/rnn-effectiveness/` (accedido en Agosto de 2017).

[16] Karpathy, A., Toderici, G., Shetty, S., Leung, T., Sukthankar, R., and Fei-Fei, L. *Large-scale video classification with convolutional neural networks.* In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition. Computer Science Department, Stanford University* (2014).

[17] Kim, J.-S. *Using Keras and Theano for deep learning driven jazz generation.* (2016). Disponible en: `https://deepjazz.io/`(accedido en Agosto de 2017).

[18] Mitchell, T. M. *Machine Learning. Artificial Neural Networks; pag 81.* Pusblisher: McGraw-Hill Science/Engineering/Math, (March 1, 1997). 432 pages. ISBN: 0070428077.

[19] Mitchell, T. M. *Machine Learning. Perceptrons, Gradient Descent and Stochastic Gradient Descent; pags 86, 89 and 92.* Pusblisher: McGraw-Hill Science/Engineering/Math, (March 1, 1997). 432 pages. ISBN: 0070428077.

[20] Mitchell, T. M. *Machine Learning. The Backpropagation Algorithm, pags 97-98.* Pusblisher: McGraw-Hill Science/Engineering/Math, (March 1, 1997). 432 pages. ISBN: 0070428077.

[21] Mordvintsev, A., Olah, C., and Tyka, M. *Inceptionism: Going Deeper into Neural Networks.* (2015). Disponible en: `https://research.googleblog.com/2015/06/inceptionism-going-deeper-into-neural.html` (accedido en Agosto de 2017).

[22] Nayebi, A., and Vitelli, M. *GRUV: Algorithmic Music Generation using Recurrent Neural Networks.* Stanford University. Stanford, California, (2015). Disponible en: `http://cs224d.stanford.edu/reports/NayebiAran.pdf` (accedido en Agosto de 2017).

[23] Schmidhuber, J. *Deep Learning in Neural Networks: An Overview.* Technical Report IDSIA-03-14 (88 pages, 888 references). The Swiss AI Lab IDSIA, Istituto Dalle Molle di Studi sull'Intelligenza Artificiale, University of Lugano and SUPSI, Galleria 2, 6928 Manno-Lugano Switzerland, (8 October 2014).

[24] Schulkin, J., and Raglan, G. B. *The evolution of music and human social capability.* Department of Neuroscience, Georgetown University, Washington, DC, USA; Department of Research, American College of Obstetricians and Gynecologists, Washington, DC, USA, (2014).

[25] Scirea, M., Barros, G., Shaker, N., and Togelius, J. *SMUG: Scientific Music Generator.* Center for Computer Games Research IT University of Copenhagen, Denmark and Department of Computer Science and Engineering New York University, NY, USA, (2015).

[26] Shieber, S. M. *Lessons from a Restricted Turing Test.* Aiken Computation Laboratory, Division of Applied Sciences, Research in Computing Technology, Harvard University, (April 15, 1993).

[27] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. *Dropout: A Simple Way to Prevent Neural Networks from Overfitting.* Journal of Machine Learning Research 15 (2014) 1929-1958. Department of Computer Science, University of Toronto, 10 Kings College Road, Rm 3302 Toronto, Ontario, M5S 3G4, Canada.

[28] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. *Going deeper with convolutions.* University of North Carolina, Chapel Hill and University of Michigan, (Setiembre 2014).

[29] Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints abs/1605.02688* (2016).

[30] Turing, A. M. *Computing machinery and intelligence.* A. M. Turing (1950) Computing Machinery and Intelligence. Mind 49: 433-460.

[31] Zhang, N., Paluri, M., Taigman, Y., Fergus, R., and Bourdev, L. *Beyond frontal faces: Improving person recognition using multiple cues.* In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2015), IEEE, pp. 4804–4813.

[32] Zimmerman, Y. *Music Language Modeling with Recurrent Neural Networks*, (2015). Disponible en: `http://yoavz.com/music_rnn/` (accedido en Agosto de 2017).