



CAMPUS
DE EXCELENCIA
INTERNACIONAL



POLITÉCNICA

"Ingeniamos el futuro"

Graduado en Ingeniería Informática

Universidad Politécnica de Madrid

Escuela Técnica Superior de
Ingenieros Informáticos

TRABAJO FIN DE GRADO

Generación automática de música mediante
tecnologías de aprendizaje profundo

Autor: Juan Carlos García Torrecilla

Director: Emilio Serrano Fernández

MADRID, ENERO 2018

Índice

Índice de Figuras	iv
Índice de Ecuaciones	vi
Índice de Tablas.....	vi
Resumen	viii
Abstract.....	ix
1. Introducción y objetivos	1
1.1. Introducción	1
1.2. Objetivos	4
2. Desarrollo	5
2.1. Background	5
2.1.1. Representación musical	5
2.1.2. Notación ABC para música	8
2.1.3. Aprendizaje automático	11
2.1.4. Redes neuronales artificiales	12
2.1.5. Redes neuronales recurrentes	18
2.1.6. Codificación One-Hot y función de activación softmax	21
2.1.7. Métricas de calidad	22
2.1.8. Keras y Tensorflow	24
2.2. Trabajos relacionados	26
2.3. Dataset	28
2.4. Entorno de trabajo.....	29
2.5. Metodología	30
2.6. Primera iteración.....	33
2.6.1. Preproceso	33
2.6.2. Modelado.....	37
2.6.3. Evaluación	40
2.6.4. Elaboración de aplicación web	52
2.7. Segunda iteración.....	68
2.7.1. Preproceso	68
2.7.2. Modelado.....	70
2.7.3. Evaluación	72

2.7.4.	Elaboración de aplicación web	77
3.	Resultados y conclusiones	90
4.	Bibliografía.....	93

Índice de Figuras

Figura 1 - Claves musicales.....	5
Figura 2 - Escala musical.....	5
Figura 3 - Figuras musicales alteradas	6
Figura 4 - Escala en Mi mayor (con y sin armadura)	6
Figura 5 - Equivalencia entre notas	6
Figura 6 - Notas con puntillo y su equivalencia en notas ligadas.....	7
Figura 7 – Tresillo	7
Figura 8 - Código ABC y representación	9
Figura 9 - Adornos musicales	10
Figura 10 - Diagrama básico de una neurona	13
Figura 11 - Neurona de McCulloch-Pitts (1943).....	13
Figura 12 - Estructura de una red neuronal artificial.....	14
Figura 13 - Función step del perceptrón simple con umbral $\theta = 0$	15
Figura 14 - Función logística.....	15
Figura 15 - Función tangente hiperbólica.....	16
Figura 16 - Rectifier Linear Unit.....	16
Figura 17 - Red neuronal recurrente.....	18
Figura 18 - Estructura de red LSTM	19
Figura 19 - Célula LSTM	20
Figura 20 - Célula GRU	20
Figura 21 - Desarrollo iterativo	30
Figura 22 - Desarrollo incremental.....	30
Figura 23 - Ejemplo de scrum	31
Figura 24 - Diagrama de proceso CRISP-DM.....	31
Figura 25 - Equivalencia de notas con diferente duración	33
Figura 26 - Equivalencia de notas con diferente armadura de tonalidad.....	34
Figura 27 - Equivalencia de multiplicadores	34
Figura 28 - Ejemplo de preproceso de clave	35
Figura 29 - Equivalencia de alteraciones.....	36
Figura 30 - Estructura de red inicial (Iteración 1)	37
Figura 31 - Diagrama para un epoch	39
Figura 32 - Entrenamiento SimpleRNN 1-layer: 16 neuronas, 16 muestras/bloque	41
Figura 33 - Entrenamiento SimpleRNN 1-layer: 16 neuronas, 32 muestras/bloque	41
Figura 34 - Entrenamiento SimpleRNN 1-layer: 16 neuronas, 64 muestras/bloque	42
Figura 35 - Entrenamiento SimpleRNN 1-layer: 16 neuronas, 128 muestras/bloque	42
Figura 36 - Entrenamiento SimpleRNN 1-layer: 32 neuronas, 16 muestras/bloque	43
Figura 37 - Entrenamiento SimpleRNN 1-layer: 32 neuronas, 32 muestras/bloque	43
Figura 38 - Entrenamiento SimpleRNN 1-layer: 32 neuronas, 64 muestras/bloque	44
Figura 39 - Entrenamiento SimpleRNN 1-layer: 32 neuronas, 128 muestras/bloque	44
Figura 40 - Entrenamiento SimpleRNN 1-layer: 64 neuronas, 16 muestras/bloque	45
Figura 41 - Entrenamiento SimpleRNN 1-layer: 64 neuronas, 32 muestras/bloque	45

Figura 42 - Entrenamiento SimpleRNN 1-layer: 64 neuronas, 64 muestras/bloque	46
Figura 43 - Entrenamiento SimpleRNN 1-layer: 64 neuronas, 128 muestras/bloque	46
Figura 44 - Entrenamiento SimpleRNN 1-layer: 128 neuronas, 16 muestras/bloque	47
Figura 45 - Entrenamiento SimpleRNN 1-layer: 128 neuronas, 32 muestras/bloque	47
Figura 46 - Entrenamiento SimpleRNN 1-layer: 128 neuronas, 64 muestras/bloque	48
Figura 47 - Entrenamiento SimpleRNN 1-layer: 128 neuronas, 128 muestras/bloque ..	48
Figura 48 - Límite de overfitting	49
Figura 49 - Mejora con 64 neuronas y 64 muestras por bloque	50
Figura 50 - Mejora con 64 neuronas y 128 muestras por bloque	50
Figura 51 - Diagrama de contexto (iteración 1)	54
Figura 52 - Acontecimiento 1	55
Figura 53 - Acontecimiento 2	55
Figura 54 - Acontecimiento 3	56
Figura 55 - Acontecimiento 1 detallado	56
Figura 56 - Acontecimiento 2 detallado	57
Figura 57 - Acontecimiento 3 detallado	57
Figura 58 - Diagrama de flujo de datos (iteración 1)	58
Figura 59 - Interfaz en PC (iteración 1)	59
Figura 60 - Interfaz en Smartphone (iteración 1)	59
Figura 61 - Estructura de directorios de Jinja2	60
Figura 62 - Componentes MVC	61
Figura 63 - Arquitectura MVC con rutas	61
Figura 64 - Plantilla base	62
Figura 65 - Diagrama UML de las plantillas de la vista (iteración 1)	63
Figura 66 - Captura de index.html en PC (iteración 1)	65
Figura 67 - Captura de generar.html en PC con datos (iteración 1)	65
Figura 68 - Captura de index.html en SmartPhone (iteración 1)	66
Figura 69 - Captura de generar.html en SmartPhone (iteración 1)	66
Figura 70 - Visualización de composición en SmartPhone (iteración 1)	67
Figura 71 - Posiciones de dropout en la red	71
Figura 72 - Comparativa de dropout	72
Figura 73 - Comparativa de dropout (ampliado)	73
Figura 74 - Comparativa SimpleRNN-LSTM-GRU (loss y accuracy)	74
Figura 75 - Comparativa SimpleRNN-LSTM-GRU (precision, recall y f-measure)	74
Figura 76 - Comparativa SimpleRNN-LSTM-GRU (loss y accuracy ampliado)	75
Figura 77 - Comparativa SimpleRNN-LSTM-GRU (precision, recall y f-measure ampliado)	75
Figura 78 - Incremento de diferencia en los resultados LSTM vs GRU	76
Figura 79 - Diagrama de contexto (iteración 2)	78
Figura 80 - Acontecimiento 4	78
Figura 81 - Acontecimiento 4 detallado	79
Figura 82 - Diagrama de flujo de datos (iteración 2)	80
Figura 83 - Interfaz en PC: Generación (iteración 2)	81

Figura 84 - Interfaz en Smartphone: Generación (iteración 2).....	81
Figura 85 - Interfaz en PC: Exploración (iteración 2)	82
Figura 86 - Interfaz en Smartphone: Exploración (iteración 2).....	82
Figura 87 - Diagrama UML de las plantillas de la vista (iteración 2)	83
Figura 88 - Captura de index.html en PC (iteración 2)	85
Figura 89 - Captura de generar.html en PC (iteración 2)	85
Figura 90 - Captura de ventana modal en PC (iteración 2)	86
Figura 91 - Visualización de composición en PC (iteración 2).....	86
Figura 92 - Captura de control de errores en PC (iteración 2)	87
Figura 93 - Captura de explorar.html en PC (iteración 2)	87
Figura 94 - Captura de generar.html en SmartPhone (iteración 2).....	88
Figura 95 - Detalle de spinner de carga en SmartPhone (iteración 2).....	88
Figura 96 - Captura de explorar.html en SmartPhone (iteración 2)	89

Índice de Ecuaciones

Ecuación 1 - Teorema de Bayes	12
Ecuación 2 - Función de propagación (McCulloch-Pitts)	14
Ecuación 3 – Función step.....	15
Ecuación 4 - Función logística	15
Ecuación 5 - Función tangente hiperbólica	16
Ecuación 6 – Función ReLU	16
Ecuación 7 - Ecuaciones de LSTM	20
Ecuación 8 - Ecuaciones de GRU	21
Ecuación 9 - Función softmax	22
Ecuación 10 - Cálculo de la exactitud	23
Ecuación 11 - Cálculo de la precisión	23
Ecuación 12 - Cálculo de la sensibilidad.....	23
Ecuación 13 - Cálculo de F-measure	24
Ecuación 14 - Cálculo de especificidad.....	24
Ecuación 15 - Cálculo del ratio de falsos positivos.....	24
Ecuación 16 - Relación iteraciones-muestras-batch size.....	38
Ecuación 17 - Parámetros por capa SimpleRNN.....	51
Ecuación 18 - Parámetros por capa Dense	51
Ecuación 19 - Parámetros por red (iteración 1).....	51

Índice de Tablas

Tabla 1 - Octavas en notación ABC	8
Tabla 2 - Ejemplo de codificación One-Hot.....	21
Tabla 3 – Matriz de confusión.....	23
Tabla 4 - Matriz de trazabilidad (iteración 1).....	58
Tabla 5 - Pruebas de integración (iteración 1).....	64
Tabla 6 - Influencia de adornos en archivos MIDI.....	69

Tabla 7 - Matriz de trazabilidad (iteración 2).....	79
Tabla 8 - Pruebas de integración (iteración 2).....	84

Resumen

Este trabajo presenta el desarrollo de un modelo capaz de generar y completar composiciones musicales de forma automática mediante algoritmos generativos de aprendizaje automático, como son las redes neuronales recurrentes.

A lo largo del documento se estudian y comparan diferentes estructuras de red neuronal –principalmente estructuras recurrentes y estructuras que implementan células de memoria, debido a sus notables resultados–, y se discute la representación de los datos de entrada, así como los aspectos de diseño para la creación de un modelo capaz de generar composiciones. El documento también presenta el diseño e implementación de una interfaz web que de acceso a este servicio.

Se trata pues, de una herramienta para asistir a creativos y artistas, tanto profesionales como aficionados, en el proceso creativo. Si bien ya existen proyectos que tratan de componer música procesada como texto, ninguno realiza un preproceso sobre los datos de entrada. Esto implica que la red debe aprender la sintaxis del código al completo, la cual contiene múltiples términos dependientes entre sí, siendo dichos términos en ocasiones muy distantes. El aprendizaje de toda esta sintaxis conlleva un entrenamiento muy intenso, lo que produce sobreajuste (*overfitting*) en muchos casos.

Habitualmente, si se detiene el entrenamiento antes del *overfitting*, no se produce código sintácticamente correcto. Y si se alcanza una sintaxis correcta, se produce *overfitting*. Para solventar este problema, se realiza un preproceso de datos que elimina las dependencias más complejas. Además, se toman las notas (“=A,” “~”, “=b”...) como elementos temporales, en lugar de los símbolos (“=”, “A”, “b”...). Así, el contenido musical se abstrae de la sintaxis.

Como resultado de esto, la salida de la red devolverá probabilidades para cada nota, y no para cada símbolo. Lo cual nos permite implementar una nueva funcionalidad que genere diferentes composiciones con los mismos datos de entrada. Esta funcionalidad selecciona la nota aleatoriamente en base a la distribución de probabilidad dada (como si de un dado ponderado se tratase). Así, se provee al modelo de un factor creativo del que carecen el resto de modelos.

A todo esto se le suma una interfaz web para facilitar el uso de la herramienta y ponerla al alcance de usuarios sin conocimientos técnicos.

En los resultados y evaluaciones de este proyecto se presta especial atención a la detección de *overfitting*, puesto que uno de los objetivos esenciales que persigue este proyecto es la generación de contenido creativo. Por ende, completamente nuevo.

A la finalización de este proyecto, los resultados alcanzan el 37% de exactitud en el conjunto de validación, sin que se produzca sobreajuste en el modelo.

Abstract

This work presents the development of a model capable of generating and completing musical compositions automatically through generative algorithms of machine learning, such as recurrent neural networks.

Throughout the document different neuronal network structures –mainly recurrent structures and structures that implement memory cells, due to their remarkable results– are studied and compared, and the representation of the input data is discussed, as well as the design aspects for the creation of a model capable of generating compositions. The document also presents the design and implementation of a web interface to access this service.

It is therefore a tool to assist creatives and artists, both professionals and amateurs, in the creative process. While there are already projects that try to compose music processed as text, none performs a preprocess on the input data. This implies that the network must learn the syntax of the code completely which contains multiple terms that depend on each other, these terms being sometimes very distant. Learning all this syntax involves a very intense training, which produces overfitting in many cases.

Usually, if training is stopped before overfitting, syntactically correct code is not produced. And if a correct syntax is reached, overfitting occurs. To solve this problem, a preprocessing of data is carried out that eliminates the most complex dependencies. In addition, notes (“=A,” “~”, “=b”...) are taken as temporary elements, instead of symbols (“=”, “A”, “b”...). Thus, the musical content is abstracted from the syntax.

As a result of this, the output of the network will return probabilities for each note, and not for each symbol. Which allows us to implement a new functionality that generates different compositions with the same input data. This functionality selects the note randomly based on the given probability distribution (like a weighted dice). Thus, the model is provided with a creative factor that the other models lack.

To all this we add a web interface to facilitate the use of the tool and make it available to users without technical knowledge.

In the results and evaluations of this project, special attention is paid to the overfitting detection, since one of the essential objectives pursued by this project is the generation of creative content. Therefore, completely new.

On finishing this project, the results reach 37% accuracy in the validation set, without overfitting the model.

1. INTRODUCCIÓN Y OBJETIVOS

1.1. Introducción

Con el avance de la tecnología, cada vez son más las tareas que podemos delegar en las máquinas, haciéndonos la vida más cómoda. Hoy en día, las máquinas llevan a cabo tareas y procesos complejos que hace años eran impensables para una máquina, llegando incluso a imitar comportamientos humanos. Por supuesto, superar y mejorar estos avances es cada vez más complicado. Pero, ¿existe un límite que marque qué puede o no puede llegar a hacer una máquina?

Cuando nos encontramos ante un problema cuya solución viene dada por una regla universal, como son problemas puramente lógicos o matemáticos (sumas, restas, operaciones lógicas), el proceso de implementación en una máquina es completamente trivial. Se conoce el mecanismo (algoritmo) para obtener el resultado, y únicamente ha de realizarse una implementación mediante hardware o software.

En cambio, cuando nos encontramos ante problemas cuya respuesta no viene dada por una regla, la complejidad aumenta. El paso de voz a texto o texto a voz, la distinción del género de una voz, la identificación de la especie de una planta o el reconocimiento de caras son problemas que, aunque los humanos podamos resolver de forma más o menos sencilla, no tienen una forma sencilla de ser llevados a lenguaje máquina ya que no existe una regla exacta que indique siempre cuál es la respuesta correcta. Es decir, no hay un algoritmo conocido que nos indique cómo hacer dicho proceso.

Se trata de aprender conceptos que no tienen una definición clara. ¿Qué hace que una cara sea una cara o que una voz sea masculina o femenina?

El aprendizaje automático nos proporciona mecanismos con los cuales enseñar a una máquina mediante inducción del conocimiento basado en ejemplos. Gracias a los algoritmos de aprendizaje automático, la máquina puede aprender, generalizar un concepto, y llevar a cabo dichos procesos que, hasta hace relativamente poco, se consideraban propios únicamente de algunos seres vivos.

Así, podemos dotar a una máquina de mecanismos que imiten procesos cerebrales y resolver problemas de percepción computacional como los comentados anteriormente. Los cuales, no olvidemos, son problemas cuya respuesta no viene definida por una regla universal; pero existe cierto consenso social en el concepto que queremos aprender (las voces femeninas son más agudas, las masculinas más graves, las caras tienen dos ojos, nariz y boca, etc.). Por lo que podemos inducir ese conocimiento mediante ejemplos.

Si trasladamos esto a la creación de música, de cuadros o esculturas, el problema cambia. Ya no estamos ante una cuestión de clasificación o percepción computacional, si no que existe un componente creativo y artístico fundamental, el cual asociamos únicamente a los humanos.

Hemos enseñado a las máquinas a identificar plantas, animales, caras, voces, a diagnosticar enfermedades, a traducir textos... Pero, ¿son capaces las máquinas de crear de forma artística?

Programar un algoritmo para generar notas musicales, es sencillo. Programarlo para que genere notas musicales que sigan un patrón melódico acorde al concepto humano de “belleza musical” es mucho más complicado. Si en los ejemplos anteriores existía un consenso social que determinaba qué es una voz femenina o masculina, o qué diferencia a un perro de un gato, en este caso no la hay, ya que la belleza es subjetiva. Es decir, no existe una respuesta completamente correcta o incorrecta.

Esto añade un grado más de dificultad a la hora de modelar los datos, ya que tratamos de modelar un concepto subjetivo. Por ello, generar contenido artístico y creativo, supone un reto para el aprendizaje automático y la inteligencia artificial, el cual abordaremos a lo largo del documento.

Algunos proyectos han tratado de abordar este problema desde diferentes enfoques, produciendo software que requiere de conocimientos técnicos elevados para su uso, o que directamente no se pone a disposición de la comunidad de usuarios. En casi todos, tampoco se publican métricas de calidad que permitan la comparación de rendimiento entre proyectos.

En este trabajo se presenta el estudio y comparación de diferentes algoritmos utilizados para modelar el problema de la composición musical como series temporales. Por medio de este proyecto se establece, como una referencia más, la exactitud obtenida, cercana al 40 %. Que, a pesar de obtener resultados inferiores en comparación a los pocos proyectos que publican su exactitud, previene el *overfitting* en sus composiciones generadas. Se elabora también una herramienta web *responsive* que facilita el uso del sistema de forma gráfica para personas sin conocimientos técnicos. Además, se dota a esta herramienta web de un segundo modo de selección de notas aleatorio basado en la distribución probabilística de las mismas, que se propone como solución al problema que presentan los modelos generativos en tareas de creatividad. En las que, estos modelos generativos, para una misma secuencia de entrada, producen siempre la misma salida, limitando la funcionalidad en lo que respecta a tareas creativas.

A lo largo de las siguientes secciones, en la sección Background se hace un recorrido por los antecedentes, tratando los conocimientos previos necesarios para comprender las ideas expuestas en este documento, tanto en lo que respecta al ámbito musical (Representación musical y Notación ABC para música) como lo referente al aprendizaje automático (2.1.3 y siguientes). Seguidamente, se expone una revisión de los Trabajos relacionados, atendiendo a la visión y el enfoque que dan al problema, así como sus métricas de calidad, estructuras neuronales y su complejidad de uso. En la sección Dataset se describe el *dataset* empleado para el entrenamiento de los modelos neuronales, seguido por el Entorno de trabajo, donde se detallan las características del entorno de desarrollo del proyecto. Posteriormente, la sección Metodología especifica de qué manera se

realizará el proceso de investigación de datos y de desarrollo de *software*. Finalmente, las secciones posteriores (Primera iteración y Segunda iteración) describen el desarrollo del proyecto a lo largo de las iteraciones descritas en Metodología, separando las fases de Preproceso, Modelado y Evaluación para la parte correspondiente al proceso de investigación de datos; y Elaboración de aplicación web para la parte correspondiente a desarrollo de *software*, que a su vez se divide en las fases clásicas de ingeniería de *software* Análisis de requisitos, Diseño de la aplicación, Codificación y Pruebas. Para terminar, en Resultados y conclusiones se resumen los objetivos logrados y se discuten los resultados obtenidos en relación a la motivación inicial planteada en la Introducción, y se plantean vías de investigación futuras aún no exploradas en este documento.

1.2. Objetivos

Ya sea generando retratos a partir de siluetas, transformando pinturas a otros estilos o diseñando arquitecturas al estilo de Gaudí, la inteligencia artificial tiene cada vez más presencia en el mundo del arte –a pesar de ser una visión realmente joven, del 2015 con el trabajo “*A Neural Algorithm of Artistic Style*” [1]–, y constituye un complemento con gran potencial para la comunidad artística.

Lo que se pretende en este proyecto es poner al alcance de esta comunidad, el potencial de la inteligencia artificial de una forma sencilla y sin conocimientos de programación o inteligencia artificial. El objetivo es desarrollar una herramienta de generación automática de música, que proporcione asistencia a creativos y artistas en la composición musical, mediante técnicas de aprendizaje automático como son las redes neuronales recurrentes.

Se trata de crear un sistema que provea al creativo de un apoyo complementario, capaz de completar melodías musicales, de una forma sencilla y al alcance de usuarios sin grandes conocimientos informáticos. De forma que, dada una secuencia de notas, el modelo sea capaz de generar las notas sucesivas de forma melódica.

Puesto que la música, a pesar de tener un gran componente matemático, tiene también un componente creativo, se busca generar diferentes composiciones para una misma secuencia de notas iniciales, de forma que no siempre se genere la misma composición, pero sin dejar esta elección al completo azar.

Para lograrlo, es necesario cumplir otros objetivos específicos:

- Efectuar un proceso de limpieza, preprocesamiento y codificación de datos musicales, de forma que sea interpretable y procesable por el sistema. Esto debe hacerse conservando sus características melódicas.
- Diseñar e implementar una estructura de red neuronal.
- Entrenar el modelo correspondiente con los datos previamente codificados.
- Evaluar el modelo entrenado.
- Desarrollar una aplicación web que permita hacer uso del modelo.

2. DESARROLLO

2.1. Background

2.1.1. Representación musical

El sonido se caracteriza por 4 elementos: altura, duración, intensidad y timbre.

La altura nos permite diferenciar sonidos graves y agudos, y depende de la frecuencia del sonido. Esta determina el tono de una nota. En notación musical se representa mediante el pentagrama, clave, notas y alteraciones de esas notas. La clave actúa de referencia en el pentagrama, indicando que la línea sobre la que se encuentra corresponde con una nota concreta.



Figura 1 - Claves musicales
(Fuente: Wikimedia commons¹)

De tal forma que la clave de sol en segunda (1, en la imagen), indicaría que la segunda línea corresponde a la nota sol. De igual manera funcionarían la clave de do en tercera (2), do en cuarta (3) y fa en cuarta (4). Y, por supuesto, se puede transformar la representación de una clave a otra.

Las notas de la escala musical (Do, Re, Mi, Fa, Sol, La, Si)², se separan 1 tono entre ellas, a excepción de los pares Mi-Fa y Si-Do, que distan 1 semitono entre sí.

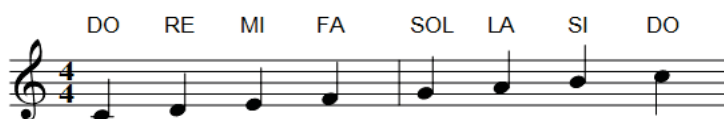


Figura 2 - Escala musical
(Fuente: Elaboración propia)

Las alteraciones “sostenido” (#) y “bemol” (b) permiten indicar variaciones de un semitono arriba o abajo respectivamente. La alteración “becuadro” (♮) anula las alteraciones anteriores que haya sobre una nota. Cuando una nota es alterada en medio de

¹ Figura recuperada de: https://commons.wikimedia.org/wiki/File:4_Common_clefs.png

² En la notación anglosajona, se utilizan respectivamente las letras: C, D, E, F, G, A, B.

una partitura, la alteración persiste sobre esa nota hasta que haya otra alteración o hasta el fin del compás.

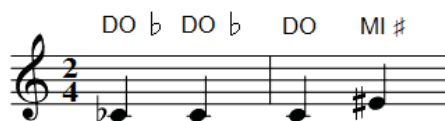


Figura 3 - Figuras musicales alteradas
(Fuente: Elaboración propia)

También pueden indicarse alteraciones al inicio del pentagrama (armadura de clave), con el objetivo de situar el pentagrama en una tonalidad específica. En ese caso, la alteración persiste durante todo el pentagrama, hasta que aparezca una nueva clave (con diferente armadura) o una alteración accidental.



Figura 4 - Escala en Mi mayor (con y sin armadura)
(Fuente: Elaboración propia)

El segundo componente del sonido es la duración, es decir, cómo de largos son los sonidos. Esto se representa en música principalmente mediante el tempo, las figuras y compases. El tempo se indica en pulsaciones por minuto (ppm, o *bpm* en inglés), y establece la velocidad a la que se debe tocar una nota (y, por extensión, las demás). De forma que, $\text{♩}=120$ indica que debe tocarse de forma que suenen 120 negras por minuto.

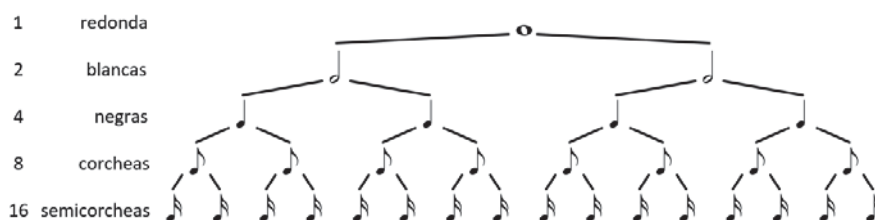
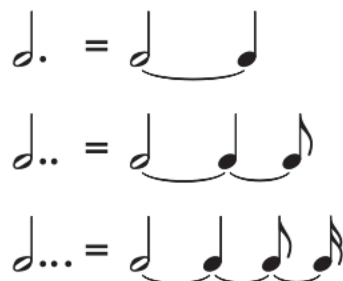


Figura 5 - Equivalencia entre notas
(Fuente: Elaboración propia)

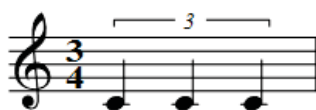
Una vez establecido el tempo, las demás figuras tendrán una duración relativa según la Figura 5 - Equivalencia entre notas. También es posible alterar la duración de las

figuras mediante símbolos, como los “puntillos”, que son puntos que se colocan a la derecha de la nota para añadirle la mitad de su duración; o ligaduras de prolongación, que suma la duración de dos notas como si fueran una sola.



*Figura 6 - Notas con puntillo y su equivalencia en notas ligadas
(Fuente: Dominio público³)*

Otra alteración en la duración de las notas se produce mediante tresillos (y similares, cinquillo, seisillo...). Esto produce que se toquen tres notas, en el tiempo correspondiente a dos notas. Por lo que, si tenemos un tresillo compuesto por tres negras, se tocarán en el tiempo correspondiente a dos negras.



*Figura 7 – Tresillo
(Fuente: Elaboración propia)*

Por otra parte, los compases actúan como métricas dividiendo un pentagrama en unidades de tiempo definidas, al inicio del pentagrama, mediante una fracción, con el objetivo de definir partes acentuadas y sin acento. El numerador indica el número de tiempos que habrá en el compás, y el denominador la unidad de medida. De tal manera que $\frac{4}{4}$ indica que cada compás deberá tener el equivalente a 4 unidades de $\frac{1}{4}$ (negras), como pueden ser 1 redonda, 2 blancas, 4 negras, 1 blanca y 2 negras, etc.

El resto de componentes del sonido (intensidad y timbre) no son especialmente relevantes en el caso que nos ocupa. La intensidad corresponde a la amplitud de la onda y nos indica la fuerza o volumen del sonido. El timbre tiene que ver con el instrumento utilizado, que nos permite diferenciar un violín de un piano.

³ Figura recuperada de: https://commons.wikimedia.org/wiki/File:Dotted_notes3.svg

2.1.2. Notación ABC para música

En este proyecto, la música está definida mediante la notación ABC, un sistema de notación musical basada en texto. Los archivos ABC se componen de una o más melodías (*tunes*). Estos están compuestos por una cabecera y contenido, sin líneas en blanco a lo largo de la melodía.

2.1.2.1. Cabecera ABC

La cabecera se compone de campos (en su mayoría opcionales), indicados cada uno en una línea. Cada campo se indica con una letra, siendo los más relevantes los listados a continuación:

- X: Número de referencia. Sirve para distinguir melodías, es obligatorio e indica el inicio de una melodía.
- T: Título de la melodía. Siempre figura como segundo campo (obligatorio).
- Q: Tempo. Opcional, por defecto: 1/4=120 (120 negras por minuto).
- M: Métrica o medida de pentagramas. Opcional, por defecto: $\frac{4}{4}$.
- L: Duración de notas. Opcional, por defecto 1/8 (corchea).
- K: Armadura de la clave. Siempre es el último campo (obligatorio).

2.1.2.2. Cuerpo ABC

El cuerpo de la melodía está compuesto por secuencias de notas representadas mediante letras. La notación ABC permite representar 4 octavas de notas, que se corresponden con las octavas 3, 4, 5 y 6 en el sistema científico internacional. Las notas se indican mediante las letras del sistema anglosajón de la forma siguiente:

	Octava 3	Octava 4	Octava 5	Octava 6
DO	C,	C	c	c'
RE	D,	D	d	d'
MI	E,	E	e	e'
FA	F,	F	f	f'
SOL	G,	G	g	g'
LA	A,	A	a	a'
SI	B,	B	b	b'

Tabla 1 - Octavas en notación ABC

En notación ABC no es necesario indicar una clave, ya que las notas llevan de forma implícita la octava a la que pertenecen (la representación queda a decisión del software que grafique el pentagrama). Sí se puede establecer la armadura de la clave mediante el campo de la cabecera “K” arriba indicado. Las alteraciones accidentales se indican delante de cada nota mediante los símbolos acento circunflejo “^” (sostenido \sharp), guion bajo “_” (bemol \flat) y signo igual “=” (becuadro \natural), y su funcionamiento es el mismo que en un pentagrama.

Los cambios de compás se indican mediante una pleca “[]”. La duración de las notas, tal cual aparecen en la Tabla 1 - Octavas en notación ABC, es la que se indique en el

campo “L” de la cabecera ya explicado, siendo su valor una fracción de redonda. Es decir, 1/2 será blanca, 1/4 será negra, 1/8 será corchea, etc. Si no se indica en la cabecera, el valor por defecto es 1/8 (corchea).

Para indicar otras duraciones, se puede añadir una fracción a la nota, que actúa como multiplicador. No existe notación para indicar el símbolo “puntillo”, ya que este cambio de duración se expresa mediante el multiplicador equivalente a la duración que exprese la nota. En cambio, sí existen ligaduras de prolongación, indicadas con el símbolo guion “-” posterior a la primera nota ligada. Los dosillos, tresillos, etc. se indican mediante la estructura “(n)” delante de la primera nota que forme parte grupo, donde “n” es el número de notas incluidas.

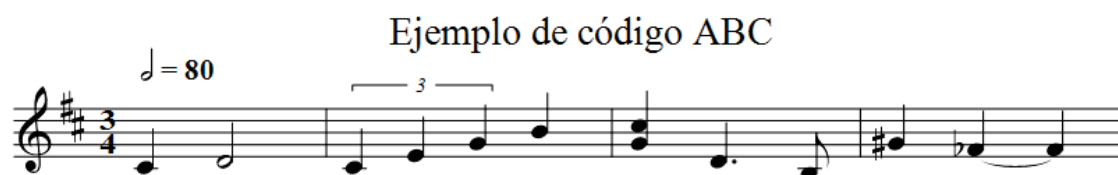
También es posible indicar acordes de acompañamiento. Para ello, basta con poner el nombre del acorde entre comillas dobles (“Am”), y los acordes se irán repitiendo a lo largo de los compases, en función de la métrica utilizada.

Finalmente, para representar acordes (varias notas tocadas en el mismo instante), simplemente se agrupan las notas dentro de corchetes.

Existe mucha más sintaxis en la notación ABC, como anotaciones de texto para indicar la letra, múltiples campos de información y metainformación en la cabecera, ligaduras de unión, símbolos de repetición de pentagramas, etc.

En la Figura 8 - Código ABC y representación se muestra un ejemplo de código y su representación en pentagrama, que ilustra la notación descrita.

```
X:1
T:Ejemplo de código ABC
L:1/4
M:3/4
Q:1/2=80
K:Bm
C D2 | (3 C E G B | [Gc] D3/2 B,/2 | ^G _F- F |
```



*Figura 8 - Código ABC y representación
(Fuente: Elaboración propia)*

2.1.2.3. Adornos

Los adornos, decoraciones u ornamentos (*decorations, ornaments*) son florituras que sirven para decorar la línea en la que aparecen, sin ser necesarias para llevar la línea general de la melodía.

En notación ABC, se indican mediante palabras clave o símbolos entre signos de cierre de exclamación (!), antes de la nota correspondiente. Los adornos más habituales tienen abreviaturas como se indica a continuación:

- **!roll!** (abreviatura: ~). *Roll* irlandés, es un adorno en el que la nota es precedida por una breve nota de tono superior, y sucedida por una breve nota de tono inferior.
- **!fermata!** (abreviatura: H). Punto de reposo, alarga la duración de la nota suspendiendo el pulso (es decir, no afecta al recuento de notas del pentagrama). La duración se alarga a decisión del intérprete, siendo habitual considerar el doble de duración de la nota a la que afecta.
- **!accent!** (abreviatura: L). Indica que la nota debe tocarse con énfasis inicial y disminuir con rapidez.
- **!lowermordent!** (abreviatura: M). La nota debe tocarse con una única y rápida alternancia con la nota de tono inferior.
- **!uppermordent!** (abreviatura: P). Igual que el anterior, pero con la nota superior.
- **!coda!** (abreviatura: O). Símbolo usado junto con anotaciones de texto que indican repeticiones. Habitualmente, desde el inicio hasta el símbolo.
- **!segno!** (abreviatura: S). Símbolo usado junto con anotaciones de texto que indican repeticiones. Habitualmente, desde el símbolo hasta el final, o hasta el símbolo anterior.
- **!trill!** (abreviatura: T). Indica alternancia rápida entre dos notas adyacentes, a un semitono o tono de distancia.
- **!upbow!** (abreviatura: u). Indicaciones de interpretación para instrumentos de cuerda frotada (violín, viola, violonchelo y contrabajo).
- **!downbow!** (abreviatura: v). Igual que el anterior.

El símbolo tilde “~” también puede ser utilizado para crear macros, de forma que al poner el símbolo antes de una nota, se sustituya por un conjunto de notas. Además, se utiliza el punto “.” delante de una nota para indicar lo que se denomina “*staccato*”. Esto acorta el sonido, el cual se continúa con una breve pausa.

```
X:1
T:Decorations
K:C
!roll!G2 !fermata!G2 !accent!G2 !lowermordent!G2 | !uppermordent!G2 !coda!G2 !segno!G2 !trill!G2 | !upbow!G2 !downbow!G2 .G2 ||
w:Roll Fermata Accent Lowermordent Uppermordent Coda Segno Trill Upbow Downbow Staccato
```

Figura 9 - Adornos musicales
(Fuente: Elaboración propia)

2.1.3. Aprendizaje automático

El aprendizaje automático (en inglés, “*Machine Learning*”) es un campo de las ciencias de la computación y una de las principales ramas de la inteligencia artificial, que tiene como objetivo el desarrollo de técnicas y mecanismos que proporcionen a un sistema la habilidad de aprender sin estar explícitamente programado para ello [2] .

En este caso, “aprender” significa identificar patrones complejos en grandes cantidades de datos. Esto es, partiendo de un conjunto de casos de ejemplo (al que llamamos “*dataset* de entrenamiento”), ser capaz de identificar los patrones generales (y no particulares) que siguen dichos ejemplos. Hablamos de patrones generales, ya que se persigue generalizar ese comportamiento, de forma que el sistema pueda reproducir los resultados tanto para casos incluidos en los ejemplos (conocidos a priori), como para casos nuevos (previamente desconocidos). Se trata de un proceso de inducción del conocimiento [3].

“No estar explícitamente programados para ello” quiere decir que es posible aplicar dicho procedimiento a múltiples casos diferentes, sin necesidad de cambiar el algoritmo. En lugar de programar un algoritmo (explícitamente) para reproducir un comportamiento o conocimiento concreto cada vez, el mismo algoritmo reproduce diferentes comportamientos en función de los datos con los que se haya entrenado.

Los algoritmos de aprendizaje automático pueden agruparse en dos clases principales dependiendo del objetivo del algoritmo y los datos de entrada que maneje [3, 4]:

- Aprendizaje supervisado: donde se proporciona un conjunto de ejemplos con una clase, categoría o valor resultante esperado, al que llamamos “etiqueta”. Estos algoritmos tratan de identificar patrones comunes a cada categoría, con el objetivo de generalizar ese conocimiento en una regla o conjunto de reglas, que predigan las etiquetas de nuevos datos.
- Aprendizaje no supervisado: donde los datos no están etiquetados. El algoritmo trata de reconocer patrones comunes en los datos para agruparlos (lo que se conoce como “*clustering*”), con el fin de identificar conjuntos de datos con características comunes, y etiquetar los nuevos datos en las clases creadas por el algoritmo.

Existen muchos enfoques diferentes, tanto en aprendizaje supervisado como no supervisado, tales como árboles de decisión, reglas de asociación, redes bayesianas... Dentro del aprendizaje supervisado, podemos agrupar los paradigmas de aprendizaje en discriminativos y generativos.

Los modelos discriminativos aprenden directamente la probabilidad condicional $p(y|x)$ para un dato “ x ” y una etiqueta “ y ” (en modelos probabilísticos), o un mapeo directo de la entrada “ x ” a la etiqueta “ y ” (no probabilísticos), es decir, aprenden a categorizar, modelan la dependencia de la variable “ y ” en función de la variable “ x ”. En cambio, los modelos generativos aprenden la distribución conjunta $p(x, y)$, es decir,

modelan cómo se estructuran y distribuyen los datos (y, por tanto, cómo se generan), y pueden hacer predicciones $p(y|x)$ mediante el teorema de Bayes [5].

$$P(A, B) = P(A|B) \cdot P(B) = P(B|A) \cdot P(A)$$

Ecuación 1 - Teorema de Bayes

Por lo general, la curva de error de los modelos discriminativos presenta una asíntota inferior a la de los modelos generativos. En cambio, estos últimos convergen mucho más rápido. Por tanto, cuando el número de datos es reducido, los algoritmos generativos se comportan mejor, y viceversa [6].

En este trabajo vamos a centrarnos en las redes neuronales artificiales que, como veremos, son algoritmos generativos muy potentes que, entre sus múltiples usos, nos van a permitir generar contenido (música en este caso) gracias a su capacidad para aprender la estructura y distribución de datos, y al aprendizaje profundo.

El aprendizaje profundo (en inglés, *Deep Learning*) constituye un subconjunto dentro del aprendizaje automático. En el ámbito de las redes neuronales, está principalmente caracterizado por la inclusión de múltiples capas de neuronas que añaden profundidad a la red neuronal. Esto, como se verá posteriormente, le permite a la red neuronal aprender características a niveles de abstracción mucho más altos que con redes neuronales tradicionales [7].

2.1.4. Redes neuronales artificiales

Las redes neuronales artificiales son estructuras con forma de red compuestas por neuronas artificiales, cuyo funcionamiento está inspirado en el cerebro animal [8].

Las neuronas del cerebro animal reciben impulsos nerviosos por las dendritas y generan un potencial de acción en el axón. De igual manera, las neuronas artificiales reciben un conjunto de entradas numéricas en un vector, imitan el comportamiento de una neurona mediante cálculos matemáticos a los que llamamos “función de propagación” (habitualmente una suma ponderada), y aplica una “función de activación”⁴ al resultado para devolver una salida. Un ejemplo de ello es el primero modelo teórico de neurona artificial planteado por Warren S. McCulloch y Walter H. Pitts en 1943 [9], aunque existen otras muchas y diversas implementaciones de neuronas artificiales.

⁴ Algunos autores utilizan indistintamente “función de propagación” y “función de activación” (utilizando en inglés “*transfer function*” y “*activation function*”) para referirse al conjunto completo. No hay un consenso al respecto. En este caso, utilizamos un término para cada parte.

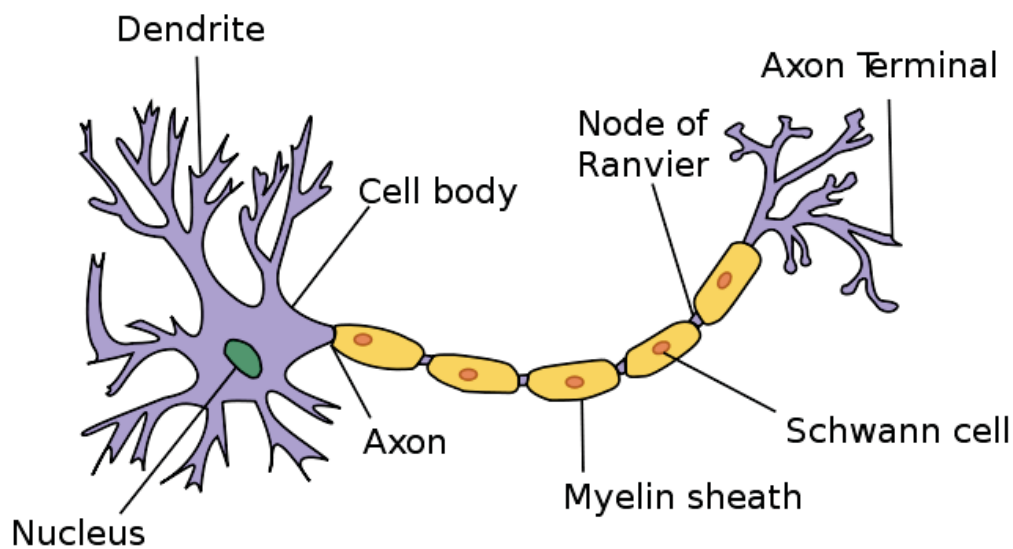


Figura 10 - Diagrama básico de una neurona
(Fuente: Wikimedia Commons⁵)

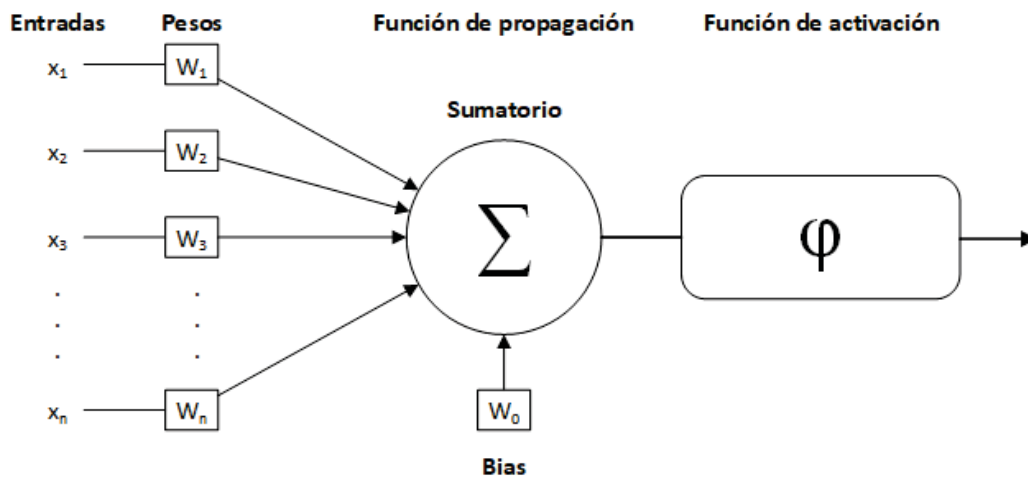


Figura 11 - Neurona de McCulloch-Pitts (1943)
(Elaboración propia)

Estas estructuras se combinan entre sí para formar redes neuronales artificiales que pueden tener múltiples capas y múltiples neuronas por capa. Habitualmente se distingue la capa de entrada, que no es una capa de neuronas como tal, sino que presenta una posición para cada uno de los datos de entrada; la capa de salida, cuyo número de neuronas determinará el número de datos de salida que obtendremos; y las capas intermedias, llamadas capas ocultas [10].

⁵ Figura recuperada de: <https://commons.wikimedia.org/wiki/File:Neuron.svg>

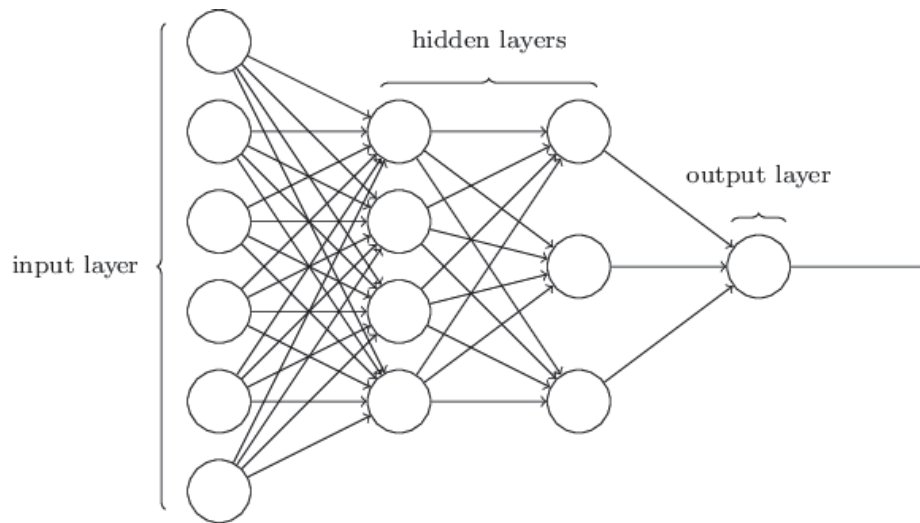


Figura 12 - Estructura de una red neuronal artificial
(Fuente: *Neural Networks and Deep Learning* by Michael Nielsen [10])

En 1958, Frank Rosenblatt implementa el perceptrón [11], basado en el modelo teórico de McCulloch y Pitts (y la regla de aprendizaje Hebbiana) cuya función de activación consistía en un umbral que disparaba la señal (Ecuación 3 – Función step). En 1969, Minsky [12] demostró que el perceptrón simple sólo podía resolver problemas linealmente separables.

2.1.4.1. Linealidad y funciones de activación

La linealidad de una neurona depende de su función de propagación y su función de activación. Si ambas son lineales, el resultado de su composición es otra función lineal, y esto aplica a toda una red. Esto se traduce en una gran limitación en las capacidades de la neurona, ya que solo puede discriminar datos linealmente separables. De igual manera, se disipa el concepto de “profundidad” en una red, ya que la consecución de capas de neuronas lineales resulta en una composición lineal, cuyo resultado es una función lineal equivalente.

En muchos casos, es habitual que la función de propagación sea una función lineal, como la que aparece en la Figura 11 - Neurona de McCulloch-Pitts (1943), que responde a la ecuación:

$$f_p(x) = W_0 + W_1x_1 + W_2x_2 + \cdots + W_nx_n$$

Ecuación 2 - Función de propagación (McCulloch-Pitts)

Por ello, las funciones de activación, que tradicionalmente establecían un umbral a partir del cual se activaba la salida, hoy son habitualmente no lineales y su objetivo de introducir no-linealidad al sistema es igual o más importante que disparar la salida.

Actualmente, las funciones de activación cuyo uso está más extendido son la función logística, tanh (tangente hiperbólica) y ReLU (Rectified Linear Unit).

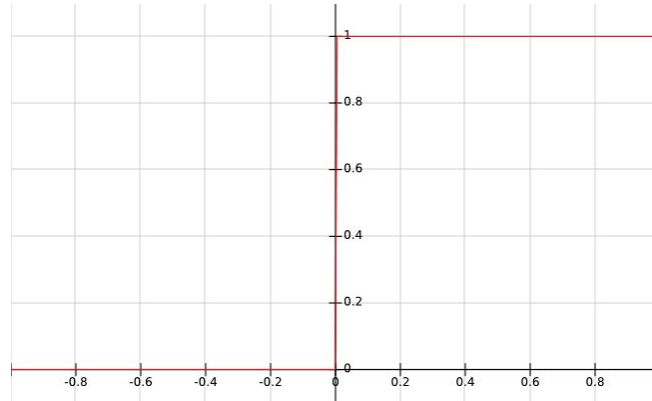


Figura 13 - Función step del perceptrón simple con umbral $\theta = 0$
(Fuente: Elaboración propia)

$$f(x) = \begin{cases} 1, & x \geq \theta \\ 0, & x < \theta \end{cases}$$

Ecuación 3 – Función step

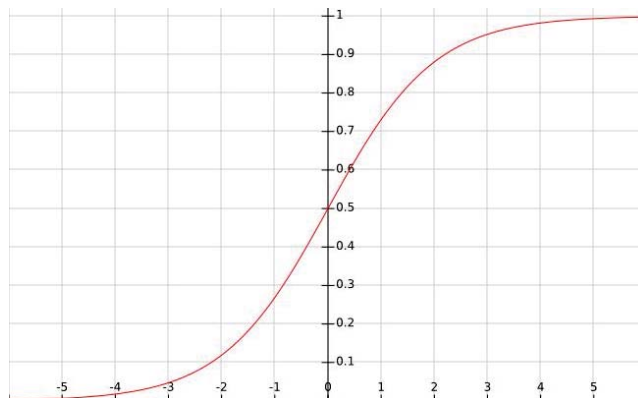


Figura 14 - Función logística
(Fuente: Elaboración propia)

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Ecuación 4 - Función logística

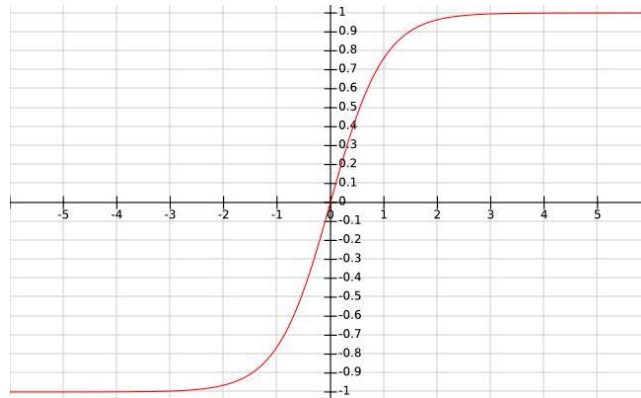


Figura 15 - Función tangente hiperbólica
(Fuente: Elaboración propia)

$$\tanh(x) = \frac{\sinh(x)}{\cosh(x)} = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{e^{2x} - 1}{e^{2x} + 1} = \frac{1 - e^{-2x}}{1 + e^{-2x}}$$

Ecuación 5 - Función tangente hiperbólica

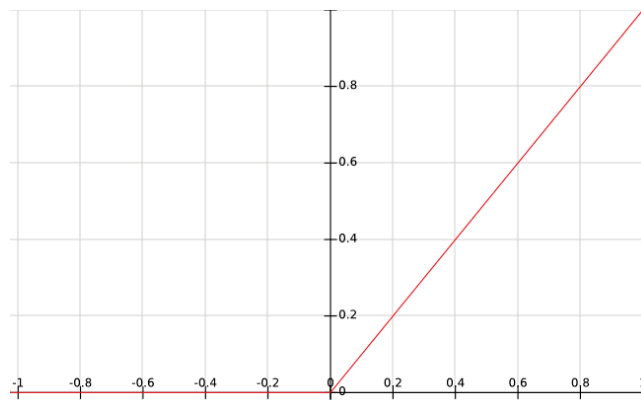


Figura 16 - Rectifier Linear Unit
(Fuente: Elaboración propia)

$$f(x) = \max(0, x)$$

Ecuación 6 - Función ReLU

En aprendizaje automático, a menudo se nombra a la función logística como función sigmoide, aunque en realidad este término denomina al conjunto de funciones que

describen una curva en forma de “S”, entre las que también se encuentra la tangente hiperbólica. Por lo que, en adelante, se nombrará como sigmoide.

Como se puede ver en la Figura 13 - Función step del perceptrón simple con umbral $\theta = 0$, el hecho de que la salida se limite a 0 ó 1 hace que un pequeño cambio en los pesos o el *bias* de la función de propagación pueda cambiar el resultado completamente, ya que no tenemos valores intermedios. Y las funciones lineales, como se ha visto, no son efectivas.

2.1.4.2. Funciones de activación y entrenamiento de redes neuronales

Actualmente, para el entrenamiento de redes neuronales, se utiliza la propagación hacia atrás (en inglés, *backpropagation*) [13, 14]. Una vez la entrada se ha propagado hacia la salida (propagación hacia delante o *forward propagation*) y se ha obtenido un resultado, se mide el error entre el resultado obtenido y el esperado, mediante una función de coste (*loss*). El error se propaga hacia atrás calculando las derivadas parciales de cada neurona [15], que se utilizan para corregir los pesos neuronales, mediante un algoritmo que minimice la función de coste (típicamente, un algoritmo derivado del gradiente descendente, que corrige los pesos de forma proporcional al gradiente de la función de coste para el peso actual).

La función sigmoide ganó popularidad como función de activación, debido a que no es lineal, a su salida acotada en el rango (0,1) y a la facilidad para calcular su derivada (necesaria para el algoritmo de aprendizaje comentado). La función tangente hiperbólica sucedió a la sigmoide ya que, al ser simétrica (centrada en cero), converge a mayor velocidad [14, 16].

Lo que ocurre con ambas funciones es el conocido “*Vanishing gradient problem*” [17]. Ambas funciones mapean un amplio dominio de la función $(-\infty, +\infty)$ a una salida en el rango (0,1), donde la mayor parte de la entrada se centra en las proximidades de las asíntotas de la función. Esto supone que, en las regiones próximas a las asíntotas, un gran cambio en el valor de entrada se refleje como un pequeño cambio en la salida, lo que se traduce en un gradiente pequeño.

Cuando se enlazan múltiples capas, el problema se agrava, ya que el mapeo del dominio $(-\infty, +\infty)$ en el rango de salida (0,1) producido en la primera capa, es mapeado de nuevo en la siguiente capa, y así sucesivamente. Por lo que, un gran cambio en la entrada, resultara en una variación ínfima en la salida. Así, durante la propagación hacia atrás el gradiente se va desvaneciendo y las neuronas de las primeras capas no aprenden.

Es por esto, que la función ReLU ha ganado popularidad en los últimos años. Se trata de una función sencilla de implementar y de derivar⁶, ha demostrado ser más robusta ante el problema del desvanecimiento del gradiente y, en consecuencia, permite entrenar redes a mayor velocidad [18]. Por otro lado, las neuronas que implementan una función de activación ReLU pueden alcanzar una condición en la que la función de propagación de

⁶ Aunque no es diferenciable en el punto $x = 0$, se toma 0 como derivada en ese punto.

siempre resultados negativos (habitualmente por el *bias*), y la salida de la función de activación sea 0, igual que su gradiente, por lo que la neurona no aprende y se dice que “muere”. Por esto, recientemente ha aparecido una variación denominada “*Leaky ReLU*” que se comporta mejor en estos casos, además de otras funciones similares como ELU (*Exponential Linear Unit*) o SELU (*Scaled Exponential Linear Unit*). Este tipo de funciones, son especialmente útiles en las capas ocultas de redes neuronales profundas debido a que éstas son las más afectadas por el *vanishing gradient problem*, de igual manera que las redes neuronales recurrentes.

2.1.5. Redes neuronales recurrentes

Las redes neuronales recurrentes son estructuras de red con bucles que realimentan la entrada. Es decir, cada neurona no solo recibe los datos de entrada de las neuronas de la capa anterior (o de la entrada de la red, en el caso de la primera capa); también recibe los datos generados por esa misma neurona en la iteración anterior. De esta manera, se implementa un mecanismo de memoria a corto plazo, devolviendo una salida, no solo en función de la entrada actual, sino de los valores que le preceden, por lo que su estructura es ideal para modelar series temporales.

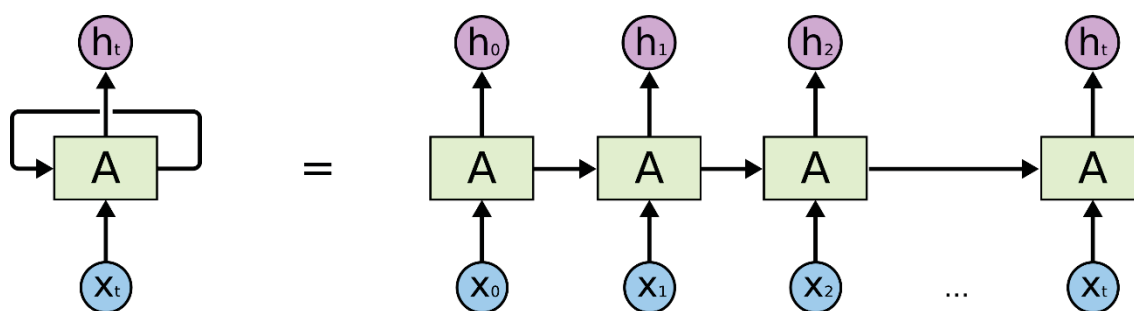


Figura 17 - Red neuronal recurrente
(Fuente: *Understanding LSTM Networks* by Christopher Olah⁷)

Debido a su estructura, las redes recurrentes se entrenan mediante *Backpropagation Through Time* (BPTT) [19], que no es más que una propagación hacia atrás adaptada a las redes recurrentes. La principal diferencia, es que se toman secuencias completas como muestras de entrenamiento; por lo cual, si tenemos en cuenta que los datos generados dependen de los datos anteriores, al calcular el gradiente del error para un parámetro en un momento dado será necesario calcular las derivadas parciales de los elementos que preceden al dato de la iteración actual. Así, el número de derivadas se incrementa y, si tenemos en cuenta que la derivada de la función sigmoide devuelve un valor entre 0 y $\frac{1}{4}$, y *tanh* entre 0 y 1, el gradiente puede en muchos casos tender a 0 rápidamente.

⁷ Figura recuperada de: <http://colah.github.io/posts/2015-08-Understanding-LSTMs>

Por ello, las redes neuronales recurrentes son especialmente difíciles de entrenar y también pueden sufrir el desvanecimiento del gradiente. No obstante es habitual utilizar funciones de activación *tanh*.

Esto es debido a que, aunque *ReLU* mejora el problema del desvanecimiento de gradiente, existe otro problema que afecta a ambas funciones, conocido como “*Exploding gradient problem*”, en el que los pesos de las primeras capas (o los primeros elementos temporales, en redes recurrentes) se saturan alcanzando valores muy altos, haciendo que las neuronas no aprendan [17, 20]. Este problema, que es especialmente característico en redes neuronales recurrentes, empeora cuando se utiliza *ReLU*, ya que su salida no está acotada como lo están las funciones sigmoide y tangente hiperbólica.

Para solucionar estos problemas, Sepp Hochreiter y Jürgen Schmidhuber introdujeron en 1997 [21, 22] las redes LSTM (*Long Short-Term Memory*). Un tipo de red recurrente compuesta por módulos LSTM que implementan mecanismos de *gating* para dotar a la red de memoria a largo plazo, y solucionan los problemas de gradiente comentados.

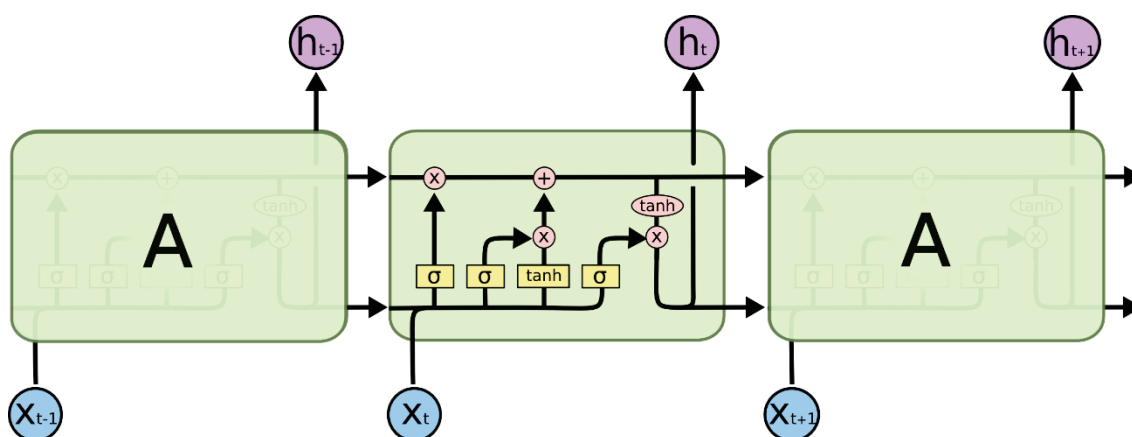


Figura 18 - Estructura de red LSTM
(Fuente: Understanding LSTM networks by Christopher Olah⁸)

Esta memoria a largo plazo se implementa mediante el estado de la unidad LSTM, que aparece representada en el diagrama mediante la línea horizontal superior (*C*), en la Figura 19 - Célula LSTM. Sobre este estado se multiplica la salida del primer sigmoide (f_t), que decide qué datos queremos borrar. Posteriormente, se suma el resultado de otras dos puertas (sigmoide i_t y tangente hiperbólica \tilde{C}_t) que añaden información nueva al estado. Este nuevo estado se pasa a la siguiente unidad, y es el que se utiliza para filtrar o_t y producir la salida h_t .

⁸ Figura recuperada de: <http://colah.github.io/posts/2015-08-Understanding-LSTMs>

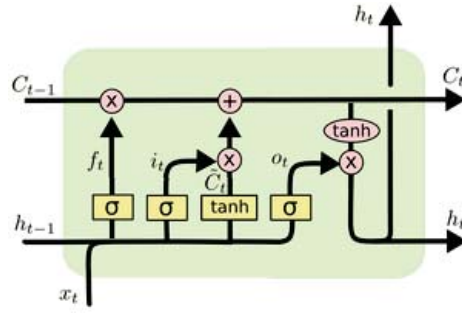


Figura 19 - Célula LSTM
(Fuente: Understanding LSTM networks by Christopher Olah⁹)

Esto se presenta matemáticamente en la Ecuación 7 - Ecuaciones de LSTM, siendo f_t la información a olvidar (*forget gate*), i_t la información a añadir (*input gate*), \tilde{C}_t el nuevo contenido de la memoria que pasa por i_t , C_t la celda de memoria y o_t la puerta de salida (*output gate*) que controla qué cantidad de la memoria se expone en la salida (h_t):

$$\begin{aligned}
 f_t &= \sigma(W_f \cdot x_t + U_f \cdot h_{t-1} + b_f) \\
 i_t &= \sigma(W_i \cdot x_t + U_i \cdot h_{t-1} + b_i) \\
 \tilde{C}_t &= \tanh(W_C \cdot x_t + U_C \cdot h_{t-1} + b_C) \\
 C_t &= f_t * C_{t-1} + i_t * \tilde{C}_t \\
 o_t &= \sigma(W_o \cdot x_t + U_o \cdot h_{t-1} + b_o) \\
 h_t &= o_t * \tanh(C_t)
 \end{aligned}$$

Ecuación 7 - Ecuaciones de LSTM

Posteriormente, en 2014, Kyunghyun Cho, Bart van Merriënboer, Dzmitry Bahdanau y Yoshua Bengio propusieron una estructura similar a LSTM llamado GRU (*Gated Recurrent Unit*) [23, 22].

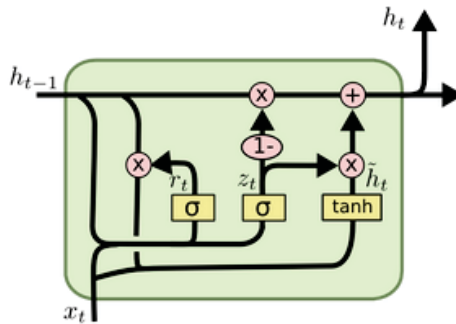


Figura 20 - Célula GRU
(Fuente: Understanding LSTM networks by Christopher Olah⁹)

⁹ Figura recuperada de: <http://colah.github.io/posts/2015-08-Understanding-LSTMs>

GRU combina la puerta de entrada y la de actualización del estado en una sola. Además, combina el estado de la célula (C en LSTM) y el estado interno (h en LSTM) en uno solo.

En este caso, z_t constituye la puerta de actualización (*update gate*), que decide qué cambios se realizan en el estado interno, r_t corresponde a la puerta de reinicio (*reset gate*) que permite “olvidar” información del estado interno para el cómputo de \tilde{h}_t , siendo \tilde{h}_t el candidato de activación, de forma que su contenido se le añade a h_t tras pasar por la puerta de actualización z_t .

$$\begin{aligned} z_t &= \sigma(W_z \cdot x_t + U_z \cdot h_{t-1}) \\ r_t &= \sigma(W_r \cdot x_t + U_r \cdot h_{t-1}) \\ \tilde{h}_t &= \tanh(W \cdot x_t + U \cdot (r_t \odot h_{t-1})) \\ h_t &= (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t \end{aligned}$$

Ecuación 8 - Ecuaciones de GRU

En este caso, no existe puerta de salida, por lo que se expone todo el contenido del estado interno.

2.1.6. Codificación One-Hot y función de activación softmax

La mayor parte de modelos de aprendizaje no manejan variables categóricas (gato, perro, caballo, ratón...), por lo que las variables categóricas son transformadas en variables numéricas (gato, 1; perro, 2; caballo, 3; ratón, 4). Esto conlleva de forma intrínseca una relación de orden entre las variables, que han pasado de categóricas a ordinales. Es decir: *gato* < *perro* < *caballo* < *ratón*.

Como esta relación no existe inicialmente, tampoco es deseable por lo que habitualmente se transforma en un vector binario mediante la codificación *One-Hot*.

La codificación *One-Hot* consiste en representar una variable categórica en un vector binario. Este vector tendrá una longitud igual al número de valores que pueda adoptar la variable categórica, de forma que cada posición corresponda a un valor.

Gato	Perro	Caballo	Ratón
1	0	0	0
0	1	0	0
0	0	1	0
0	0	0	1

Tabla 2 - Ejemplo de codificación One-Hot

Esto es especialmente útil para el uso de la función *softmax*.

$$S(y_i) = \frac{e^{y_i}}{\sum_j e^{y_j}}$$

Ecuación 9 - Función softmax

La función *softmax* es una generalización de la función logística que comprime un vector con valores reales a un vector de misma dimensión cuyos valores se encuentran en el rango $[0,1]$. Siguiendo con el ejemplo, si tenemos una red neuronal con 4 salidas (una para cada etiqueta: gato, perro, caballo, ratón), esta función nos transforma los valores del vector en probabilidades, de forma que la suma de los valores del vector resultante sea 1. Por ello, es habitualmente utilizada en la última capa de las redes neuronales.

La codificación *One-Hot* es adecuada para casos en los que se utiliza *softmax*, ya que representa los datos de igual manera, donde la variable correcta indica 1 (100%) y el resto 0 (0%).

2.1.7. Métricas de calidad

Las métricas de calidad nos permiten evaluar el rendimiento y los resultados de los modelos de aprendizaje. Si bien, entrenar y evaluar una red neuronal es complicado, especialmente redes recurrentes, para el caso que nos ocupa, la evaluación es especialmente compleja, pues no hay una forma canónica de evaluar la calidad de la música.

Esto implica que, con las métricas de las que disponemos, no necesariamente producirá mejor música un modelo cuyas métricas sean mayores; si no que, dicho modelo, se ajustará más a la distribución de los datos. Por lo general, es de esperar que cuanto más se ajuste el modelo a la distribución de los datos, mejor música producirá o, por lo menos, más se asemejará a la música empleada para el entrenamiento del modelo. Pero no necesariamente tiene que ser así, ya que la calidad musical no se ve directamente reflejada en las métricas, y recordemos que el gusto musical es completamente subjetivo.

Esto también tiene especial relación con lo que reflejan las métricas, que en ocasiones puede ser engañoso si no interpretamos correctamente las mismas.

El error o coste (*loss*), mencionado en secciones anteriores, es habitualmente utilizado para entrenar los modelos mediante la propagación hacia atrás. Es una manera de calcular el error cometido en las predicciones, y hacer correcciones en los valores. Existen múltiples funciones habitualmente utilizadas como función *loss*, como puede ser MSE (*Mean Squared Error*), MAE (*Mean Absolute Error*), *Hinge*, *CrossEntropy*, etc.

Es especialmente preciso aclarar la diferencia entre exactitud (*accuracy*), precisión (*precisión*) y sensibilidad (*recall*).

Considerando un problema de dos clases como ejemplo, tenemos 4 parámetros a partir de los resultados:

- TP (*True Positive*): Muestras positivas que han sido clasificadas positivas.
- FP (*False Positive*): Muestras negativas que han sido clasificadas positivas.
- TN (*True Negative*): Muestras negativas que han sido clasificadas negativas.
- FN (*False Negative*): Muestras positivas que han sido clasificadas negativas.

Esto generalmente se expresa mediante una matriz de confusión, donde se disponen las clases originales en un eje y las clases predichas por el modelo en otro, distribuyéndose las muestras según su clase original y su clasificación.

	Clase predicha: Sí	Clase predicha: No
Clase original: Sí	<i>True Positive</i>	<i>False Negative</i>
Clase original: No	<i>False Positive</i>	<i>True Negative</i>

Tabla 3 – Matriz de confusión

Con estos parámetros, la exactitud (*accuracy*) se calcularía como el cociente entre las respuestas correctas entre el total de respuestas. La precisión (*precisión*) se calcula para cada una de las clases, y se corresponde con el número de muestras correctamente clasificadas en una clase entre todas las clasificadas en dicha clase. La sensibilidad (*recall*) también se calcula para cada clase, correspondiendo al número de muestras correctamente clasificadas en una clase respecto del total de muestras que correspondían originalmente a dicha clase.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

Ecuación 10 - Cálculo de la exactitud

$$Precision = \frac{TP}{TP + FP}$$

Ecuación 11 - Cálculo de la precisión

$$Recall = \frac{TP}{TP + FN}$$

Ecuación 12 - Cálculo de la sensibilidad

De esta manera, si nuestro modelo clasifica casi todas las muestras en la clase negativa, tendremos alta precisión (debido a los pocos falsos positivos), pero no por ello

nuestro modelo estará funcionando correctamente, ya que la sensibilidad será muy baja debido a la gran cantidad de positivos clasificados como negativos (falsos negativos).

Como combinación de *precision* y *recall*, tenemos el *F-measure*, que proporciona un valor balanceado entre precisión y sensibilidad:

$$F_1 = 2 \cdot \frac{1}{\frac{1}{recall} + \frac{1}{precision}} = 2 \cdot \frac{precision \cdot recall}{precision + recall}$$

Ecuación 13 - Cálculo de F-measure

Por otro lado, es común utilizar la curva ROC. Se trata de una gráfica que presenta la sensibilidad (ratio de verdaderos positivos) frente a la especificidad TNR (ratio de verdaderos negativos o *True Negative Ratio*), o frente al ratio de falsos positivos FPR (*False Positive Ratio*).

$$TNR = \frac{TN}{TN + FP} = 1 - FPR$$

Ecuación 14 - Cálculo de especificidad

$$FPR = \frac{FP}{TN + FP} = 1 - TNR$$

Ecuación 15 - Cálculo del ratio de falsos positivos

Una de las métricas habituales de calidad, es calcular el área bajo la curva o AUC (*Area Under Curve*), siendo mejor cuanto mayor es el área.

2.1.8. Keras y Tensorflow

Actualmente existen varios *frameworks* de aprendizaje automático como Tensorflow, Theano, PyTorch o CNTK.

Para este trabajo, se va a utilizar Tensorflow¹⁰. Se trata de una librería desarrollada por Google que permite la elaboración de modelos de aprendizaje automático mediante grafos computacionales. En estos grafos, los nodos son tensores (matrices) y las aristas representan la aplicación de una transformación sobre los tensores.

¹⁰ https://www.tensorflow.org/api_docs/

Esto permite establecer una serie de operaciones, y mandar el grafo en su conjunto para ser ejecutado en el procesador, de forma similar a como ejecuta las operaciones individuales la librería *numpy*, pero ejecutándolas todas en conjunto. Además, proporciona la posibilidad de emplear tarjetas gráficas mediante librerías CUDA para acelerar los cálculos.

Para facilitar el uso de la librería se empleará Keras¹¹. Una API *Python* que funciona sobre Tensorflow, Theano o CNTK, y provee implementación para múltiples estructuras de red habitualmente utilizadas.

¹¹ <https://keras.io/>

2.2. Trabajos relacionados

Uno de los proyectos más conocidos en los últimos años en lo que a generación de música con aprendizaje automático se refiere es BachBot [24, 25]. Se trata de un proyecto de investigación de la Universidad de Cambridge que utiliza autocodificadores (*autoencoders*) con LSTM para generar y armonizar corales al estilo de Bach. Se utiliza el valor *loss* como métrica de evaluación y comparación de resultados, además de hacer experimentos en los que una persona debe tratar de distinguir la música original de la producida por el modelo. Otro modelo de generación de corales de Bach es DeepBach propuesto por [26, 27], en el que utilizan un algoritmo basado en el muestreo de Gibbs para generar muestras de acuerdo a la distribución de probabilidad de los datos. También evalúan la calidad del modelo con un test online con oyentes humanos.

Ambos proyectos utilizan el *dataset* “Bach Chorales” y son capaces de generar música polifónica.

Probablemente uno de los proyectos más importantes en generación musical a día de hoy sea Magenta [28]. Se trata de un proyecto de Google que ofrece un conjunto de modelos pre-entrenados, y la posibilidad de entrenar los modelos con un *dataset*. Entre sus modelos podemos encontrar tanto síntesis de audio WAV (basado en WaveNet, otro proyecto de Google), como generación basada en archivos MIDI. En varios de sus modelos aplican *language modeling* para la generación musical, como son:

- Drums RNN [29]: genera *tracks* de batería mediante un LSTM. Modela los *tracks* como secuencias de eventos.
- Melody RNN [30]: genera melodía musical mediante LSTM. Utiliza la codificación *one-hot* para representar la entrada de datos.
- Polyphony RNN [31]: genera música polifónica mediante LSTM. Representa los datos mediante palabras clave que indican el inicio, fin de unidad temporal y fin de *track*. Está inspirado en BachBot.
- Performance RNN [32]: genera música polifónica mediante LSTM. Es similar a Polyphony RNN, pero representa los datos como una secuencia de eventos MIDI (inicio de nota, fin de nota, avance de tiempo y cambio de velocidad).
- Pianoroll RNN-NADE [33]: genera música polifónica mediante LSTM combinación con NADE [34] (*Neural Autoregressive Distribution Estimator*). Representa la música en forma pianoroll con vectores binarios.

En todos ellos se utiliza una estructura codificador-decodificador (*encoder-decoder*, denominada Seq2Seq). No hay información acerca de métricas de calidad en los modelos. Como en casos anteriores, sus autores reconocen que evaluar el contenido de un modelo generativo, especialmente en el área artística, es complejo. Por ello, ponen sus herramientas en manos de artistas y músicos.

También encontramos proyectos que generan audio manipulando los datos como audio en crudo (WAV). Ejemplos de ello son WaveNet [35] de Google, que utiliza redes neuronales convolucionales, tanto para generar música como reconocer y generar

fonemas; o GRUV [36, 37], que utiliza una combinación de LSTM y GRU (*Gated Recurrent Unit*). No se mencionan métricas de calidad, más allá del *loss* en el caso de GRUV.

A principios de año, Daniel Johnson presentó en EvoMusArt 2017 un nuevo modelo recurrente inspirado en las redes neuronales convolucionales [38] utilizado en su proyecto [39], que trata de ser invariante a las transposiciones. En su modelo, combina LSTM con RBM (*Restricted Boltzmann Machine*). La representación de los datos, se hace mediante un vector en el que se codifican las características de cada nota. En sus resultados, presenta el logaritmo de la verosimilitud (*log-likelihood*) como métrica para comparar modelos.

Otra publicación, por parte de la Universidad de Stanford [40, 41], analiza la generación musical utilizando la notación ABC, codificando los datos carácter a carácter, en el que obtienen buenos resultados con arquitectura Seq2Seq (codificador-decodificador) utilizando LSTM y GRU. Por el contrario, GAN (*Generative Adversarial Network*) y CBOW (*Continuous Bag Of Words*) producen resultados no reproducibles ya que no fueron capaces de aprender la estructura sintáctica. Los resultados se expresan en función de la exactitud (*accuracy*), obteniendo en el modelo elegido un 65,5%.

Tanto estas como otras diversas publicaciones coinciden en el notable resultado de las redes recurrentes como LSTM y GRU, habitualmente en estructuras Seq2Seq, así como en el uso de RBM y NADE (esta última menos común).

Es más común manipular música en formato texto o MIDI que en WAV, ya que es necesario un entrenamiento más intensivo, y tiene un mayor coste computacional, siendo necesarios varios minutos para entrenar un segundo de audio en el caso de WaveNet.

Los experimentos basados concretamente en texto, son tratados sin preproceso ni transformación previa, únicamente la codificación de letras a números, por lo que la red neuronal debe aprender la sintaxis utilizada. Esto requiere un mayor número de iteraciones de entrenamiento, lo que en algunos casos resulta en sobreajuste, haciendo que las composiciones generadas sean, en ocasiones, muy parecidas a las de entrenamiento.

En cuanto a proyectos se refiere, es habitual presentar muestras de audio generado, pero solo en algunos casos se pone a disposición el sistema para ser utilizado. Los que sí lo hacen, requieren conocimientos técnicos que no todo el mundo posee.

Por ello, este trabajo trata de preprocesar y simplificar los datos de entrada, con el objetivo de facilitar el aprendizaje y evitar el sobreajuste. Así como elaborar una interfaz web que permita hacer un uso sencillo del modelo, sin necesidad de conocimientos técnicos.

En cuanto a resultados, la mayoría presentan comparativas de los resultados *loss* como justificación del modelo elegido. Sin embargo, estos valores dependen de muchas

características como el *dataset* empleado o la función de optimización, por lo que sólo son significativos para comparar, por ejemplo, el conjunto de entrenamiento y el de validación en un mismo modelo. Por ello, no se pueden utilizar como métricas de referencia para otros casos. Lo mismo ocurre con el logaritmo de verosimilitud. En el último proyecto mencionado se emplea la exactitud como métrica, la cual proporciona una referencia que, como se ha visto en Métricas de calidad, pueden llevar a confusión si no son interpretadas adecuadamente.

2.3. Dataset

Para el desarrollo de este proyecto se va a utilizar el *dataset* “*Nottingham Music Database*”¹², en notación ABC. Éste se compone de 1037 composiciones populares de las islas británicas (especialmente de Irlanda). Estas pertenecen a diversos géneros como *Jig*, *Hornpipe*, *Polka*, *Reel* o *Waltz* entre otros. En estos géneros musicales, por lo general, son comunes las métricas 6/8 (común en *Jigs*), 2/4 y 4/4 (habituales en *Reels*) y 9/8 (*Slip Jigs*).

Inicialmente el repositorio era mantenido por Eric Foxley, de la Universidad de Nottingham, pero actualmente está a cargo de James Allwright.

¹² <http://abc.sourceforge.net/NMD>

2.4. Entorno de trabajo

Para el desarrollo de este proyecto se ha empleado una máquina con CPU Intel Core i7-2600K, 8 GB de memoria RAM DDR3, y GPU NVIDIA GeForce GTX 1060, que a su vez cuenta con 6 GB de RAM GDDR5.

El sistema empleado para el desarrollo de este proyecto es un Debian 9 Stretch versión 4.9.51-1. El proyecto se ha desarrollado empleando la versión 3.5.3 de Python¹³, que a su vez hace uso de las librerías scikit-learn¹⁴ 0.19.1 y NumPy¹⁵ 1.13.1 para cálculos y manipulación de matrices, y Matplotlib¹⁶ 2.0.2 para la generación de gráficos. Para la elaboración de la aplicación web se emplea la versión 0.12.2 de Flask¹⁷.

En cuanto a los *frameworks* de aprendizaje automático mencionados en la sección Keras y Tensorflow, la versión de Keras¹⁸ al inicio de este proyecto era 2.0.8, la cual se actualizó posteriormente a la versión 2.1.2 para hacer uso de nuevas implementaciones de red recurrente (CuDNNLSTM¹⁹ y CuDNNGRU²⁰) implementadas a lo largo de los meses de Octubre y Noviembre, que mejoran la velocidad de entrenamiento utilizando TensorFlow como *backend* y librerías CUDA. En cuanto a TensorFlow²¹, se emplea la versión 1.4.0-rc0 compilada desde código fuente. Se decidió compilar desde código fuente para habilitar el uso de instrucciones SSE4.1, SSE4.2 y AVX, que soporta el procesador de la máquina de desarrollo, y que no vienen habilitadas en los binarios precompilados. En la compilación, también se habilitó el uso de GPU para el entrenamiento de las redes neuronales. En la máquina de desarrollo se hace uso de CUDA Toolkit²² 8.0 (V8.0.61) (versión recomendada²³ por los desarrolladores de TensorFlow) y la librería CuDNN²⁴ v7.0.3 de NVIDIA.

El uso de GPUs para el entrenamiento de redes neuronales permite disminuir el tiempo de entrenamiento en algunos casos. Las GPUs están preparadas para realizar cálculos con grandes matrices y, mientras que las CPUs están optimizadas en latencia y cuentan con pocos núcleos (*cores*) muy rápidos, las GPUs están optimizadas en ancho de banda y cuentan con miles de *cores* más lentos. Esto resulta en un mejor rendimiento en GPU cuando se realizan operaciones con matrices muy grandes u operaciones que pueden ser paralelizadas como las convoluciones. Por otro lado, un código secuencial se ejecutará generalmente más rápido en CPU.

¹³ <https://docs.python.org/3.5/>

¹⁴ <http://scikit-learn.org/stable/documentation.html>

¹⁵ <https://docs.scipy.org/doc/>

¹⁶ <https://matplotlib.org/2.0.2/index.html>

¹⁷ <http://flask.pocoo.org/docs/0.12/>

¹⁸ <https://keras.io/>

¹⁹ <https://keras.io/layers/recurrent/#cudnnlstm>

²⁰ <https://keras.io/layers/recurrent/#cudnngru>

²¹ https://www.tensorflow.org/api_docs/

²² <https://developer.nvidia.com/cuda-toolkit>

²³ https://www.tensorflow.org/install/install_sources#optional_install_tensorflow_for_gpu_prerequisites

²⁴ <https://developer.nvidia.com/cudnn>

2.5. Metodología

Durante el desarrollo de este proyecto se va a seguir un marco de desarrollo iterativo e incremental, aplicando una metodología ágil [42] de tipo SCRUM [43], mediante *sprints* o iteraciones en las que se realizarán entregas regulares de una versión del producto funcional en los periodos de tiempo establecidos. Realizándose en cada *sprint* un conjunto de mejoras y ampliaciones sobre la versión producida en el *sprint* anterior.

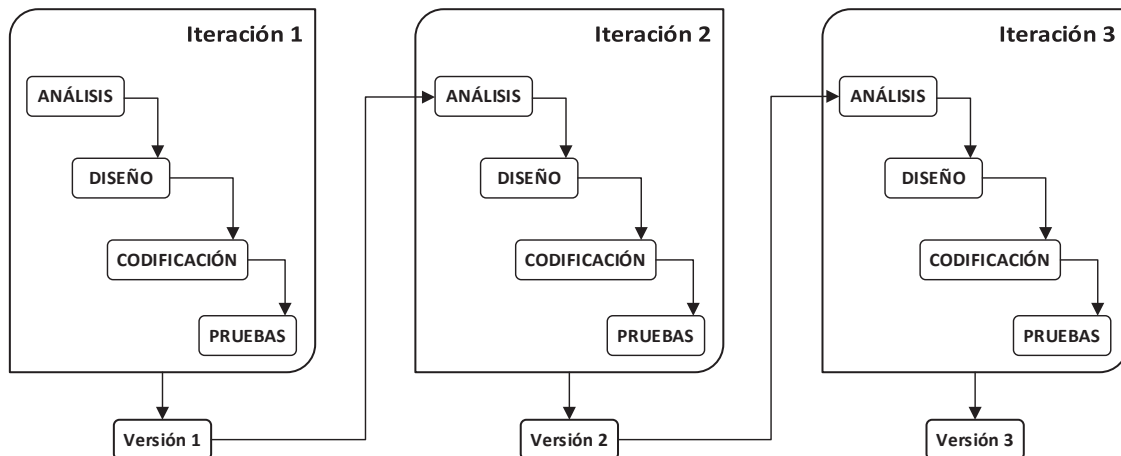


Figura 21 - Desarrollo iterativo
(Fuente: Elaboración propia)

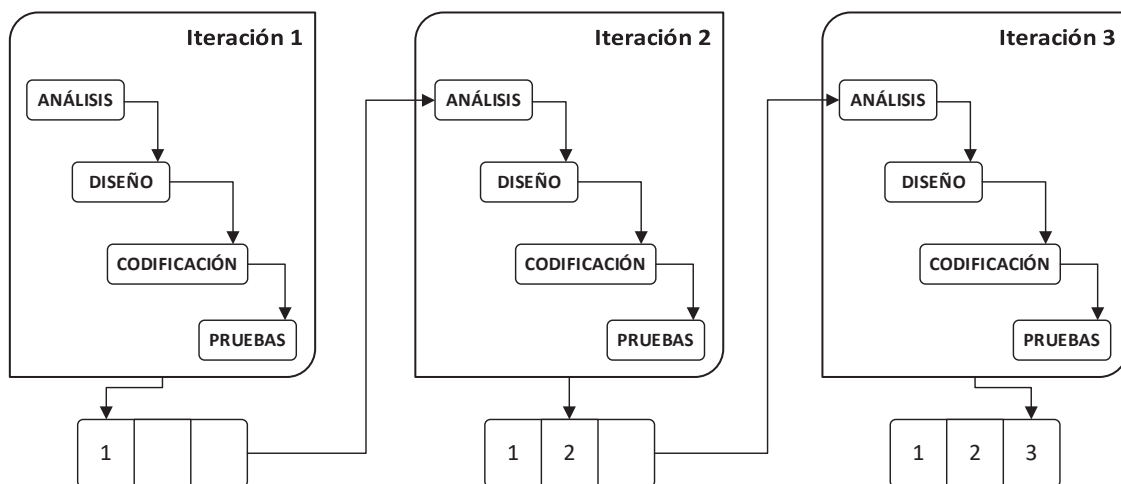


Figura 22 - Desarrollo incremental
(Fuente: Elaboración propia)

SCRUM es especialmente útil cuando se requiere un prototipado y despliegue rápido, y cuando no hay una imagen clara del producto final y se esperan cambios en los requisitos, ya que la metodología ágil presenta más agilidad para adaptarse a los cambios.

Por otro lado, requiere una alta participación por parte del cliente (que en el caso que nos ocupa, es del 100%).

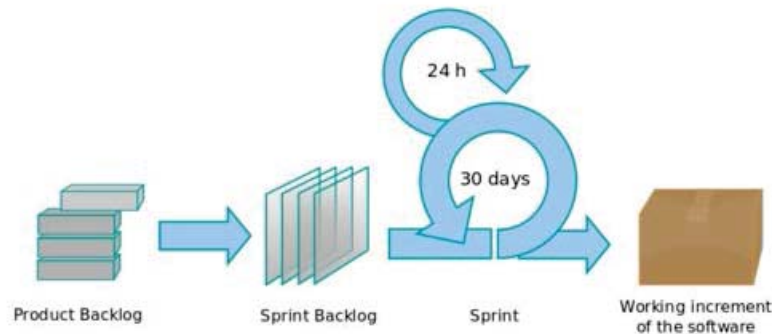


Figura 23 - Ejemplo de scrum
(Fuente: Forecast²⁵)

Durante el primer *subsprint* se desarrollará la parte correspondiente al modelo de datos del sistema. Para este desarrollo, se adaptará la metodología CRISP-DM [44, 45] (*Cross Industry Standard Process for Data Mining*). Se trata de un estándar que describe el ciclo de vida típicamente utilizado en proyectos de análisis y minería de datos. Se compone de 6 fases:

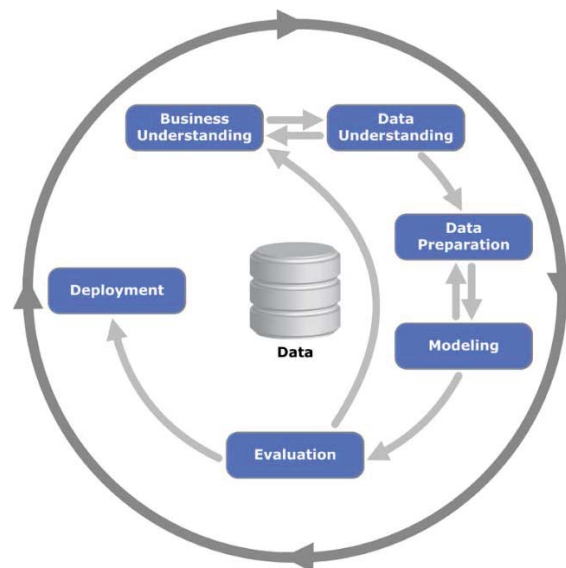


Figura 24 - Diagrama de proceso CRISP-DM
(Fuente: Wikimedia Commons²⁶)

²⁵ Figura recuperada de: <https://goo.gl/3K9bVy>

²⁶ Figura recuperada de: https://commons.wikimedia.org/wiki/File:CRISP-DM_Process_Diagram.png

- Business Understanding. Consiste en la comprensión de los objetivos del proyecto para transformarlo en un problema de minería de datos.
- Data Understanding. Durante esta fase se estudian y comprenden los datos y sus características.
- Data Preparation. Se realiza un análisis de datos y selección de características. Esta fase corresponderá al preproceso de los datos.
- Modeling. Esta fase comprende la aplicación de técnicas de modelado de datos y el ajuste de sus hiperparámetros. En nuestro caso, el diseño y elaboración de estructuras de red y su entrenamiento con diferentes hiperparámetros.
- Evaluation. Evaluación y comparación de los resultados obtenidos en la fase anterior.
- Deployment. Despliegue del modelo una vez finalizado.

Debido a las características propias del problema, se hará especial énfasis en las fases de preparación, modelado y evaluación.

El segundo *subsprint*, corresponde al diseño y desarrollo de la aplicación web que permita hacer uso del modelo. Este *subsprint* se compondrá de las fases típicas del desarrollo de software (análisis, diseño, codificación y pruebas), asumiendo que éstas son impredecibles y pueden cambiar, por lo que no son necesariamente lineales, como ocurre en el desarrollo en cascada.

2.6. Primera iteración

2.6.1. Preproceso

Como punto de partida inicial en esta iteración, se pretende reducir la música a la mínima expresión, dejando únicamente notas para hacer un análisis de las series temporales y su generación con redes neuronales recurrentes, y elaborar un prototipo inicial sobre el que hacer mejorar y ampliaciones.

Como se explicó en Representación musical, el código ABC se compone de una cabecera y un cuerpo con las notas musicales. En la cabecera podemos encontrar el número de *tune* (campo X), el título (campo T), y el *key signature* (campo K).

El campo X y el campo T son simple información que no influye en el contenido musical. El campo K, por el contrario, nos va a condicionar la interpretación de las notas, ya que establece la armadura de clave.

Otros campos que podemos encontrar habitualmente son la métrica (campo M), que establece las notas por compás, y la longitud por defecto (campo L), que nos indica el tipo de nota que se utiliza como referencia en la composición. Este último, aunque no es obligatorio (por defecto, 1/8 que equivale a notas corcheas), sí nos va a influir en la interpretación del código musical. En cambio, el campo M, afecta únicamente a la generación de acordes de acompañamiento, y a la representación del pentagrama. Es decir, dejando a un lado el acompañamiento, la melodía producida es la misma independientemente del valor del campo M.

Por ello, el campo K y el campo L nos condicionan la representación, haciendo que tengamos múltiples representaciones para el mismo código. Esto se puede ver en la Figura 25 - Equivalencia de notas con diferente duración, donde ambos códigos con diferente campo L resultan en la misma composición. Lo mismo ocurre en la Figura 26 - Equivalencia de notas con diferente armadura de tonalidad, en la que la armadura “K:Gm” introduce bemoles para las notas “si” y “mi”, alterando la nota “mi” que aparece en la composición 2, mientras que la composición 1 no tiene alteraciones en la armadura, y pero el sonido producido es el mismo, ya que la tiene indicada explícitamente en la nota “mi”.

X:1	X:2
T:Corcheas	T:Negras
L:1/8	L:1/4
K:C	K:C
C2 D2 E2 F2 	C D E F




Figura 25 - Equivalencia de notas con diferente duración

(Fuente: Elaboración propia)

X:1
T:Tonalidad C
L:1/4
K:C
C D _E F |



X:2
T:Tonalidad Gm
L:1/4
K:Gm
C D E F |



Figura 26 - Equivalencia de notas con diferente armadura de tonalidad
(Fuente: Elaboración propia)

Para simplificar estas múltiples representaciones, las primeras operaciones de preproceso consisten en transformar el *dataset* completo a una única representación. Por simplicidad, se ha decidido que el valor utilizado en el campo L sea 1/8, puesto que es el valor por defecto, y el más centrado en cuanto a multiplicadores se refiere, si consideramos notas desde la redonda hasta la semifusa. Siendo para la corchea, multiplicadores desde 8 hasta 1/8, mientras que para la negra van desde 4 hasta 1/16, como aparece en la Figura 27 - Equivalencia de multiplicadores. Esta transformación se realiza en el *script* *l-len_changer.py*²⁷.

X:1
T:Multiplicadores negra
L:1/4
K:C
C4 C2 C C/2 C/4 C/8 C/16



X:2
T:Multiplicadores corchea
L:1/8
K:C
C8 C4 C2 C C/2 C/4 C/8



Figura 27 - Equivalencia de multiplicadores
(Fuente: Elaboración propia)

²⁷ https://github.com/Tuxt/AutoScore/blob/master/01_preproceso/l-len_changer.py

A continuación, en el *script 2-key_changer.py*²⁸ se ha realizado la transformación de todas las composiciones del *dataset* a la tonalidad C (campo K), ya que no tiene alteraciones en la armadura, lo que lleva a la representación explícita de alteraciones en la propia nota. Esto simplifica la dificultad del problema eliminando dependencias, ya que las notas representan un sonido (tono) en sí mismas, sin depender de las alteraciones previas en el pentagrama, ni de alteraciones en la armadura, que pueden ser en ocasiones dependencias muy largas.

Esto nos interesa ya que, supongamos una composición cuya armadura tiene bemol en la nota “mi”, y la nota del instante temporal $t = 700$ es una nota “mi”, la nota será “mi bemol”. Tendremos una dependencia en los datos con una distancia de 700 unidades de tiempo (denominadas *timesteps*). Se trata de una dependencia muy larga. La nota será codificada como “mi” (natural), y la red debe aprender la dependencia existente entre la nota y la armadura, que convierte la nota en “mi bemol”. Si la red no es capaz de aprender dicha dependencia, será interpretada como “mi” (natural), aprendiendo entonces de forma incorrecta.

Puesto que las alteraciones de una nota en un compás afectan a las notas iguales en ese mismo compás, el *script* indica en cada nota la alteración explícitamente, incluso cuando la nota es natural (se indica un becuadro, que anula cualquier alteración). Esto se traduce en una notación mucho más densa en la escritura (lo cual no afectará al resultado, pues estas notas se codificarán en números posteriormente), pero menos redundante.

```

X:1
T:Clave original
L:1/8
K:D
C2 D2 E2 F2 | _G2 A2 B2 c2 |

X:2
T:Preproceso de clave
L:1/8
K:C
^C2 =D2 =E2 ^F2 | _G2 =A2 =B2 ^c2 |

```

Figura 28 - Ejemplo de preproceso de clave
(Fuente: Elaboración propia)

Como aparece en la Figura 28 - Ejemplo de preproceso de clave, se elimina la armadura de clave, que indica alteraciones sostenido para las notas “do” y “fa”, y se añaden las alteraciones en las propias notas (C2, F2 y c2). Las notas con alteraciones accidentales (sol “G2” en el ejemplo) mantienen su alteración propia. Las notas sin alteración propia, ni de armadura, ni de otra nota previa en el mismo compás, se marcan con un becuadro, para indicar explícitamente que no tienen alteración y evitar confusión o redundancia. Esto nos permite expresar las notas de forma independiente, no solo de la armadura del pentagrama, sino también de las alteraciones en notas previas del mismo compás.

²⁸ https://github.com/Tuxt/AutoScore/blob/master/01_preproceso/2-key_changer.py

El objetivo de este proyecto, inicialmente abarca la generación de melodías musicales, sin entrar en acompañamientos. Por ello, el *script 2-key_changer.py*²⁹ también elimina los acordes de acompañamiento, las líneas de comentarios y directivas midi, así como las directivas que indican Adornos, ya que en primera instancia sólo nos interesan las notas para esta primera iteración.

Por esa misma razón, también se eliminan los símbolos tilde (~) y las ligaduras (-) en el *script 3-del_ties_and_tildes.py*³⁰.

Posteriormente, en el *script 4-data_extractor.py*³¹, se procesan las composiciones eliminando cabeceras y otros símbolos. El *script* genera únicamente las notas de cada composición de forma secuencial, produciendo cada composición en una sola línea.

A pesar de las simplificaciones realizadas anteriormente para evitar representaciones redundantes, aún sigue existiendo redundancia debido a las alteraciones. Como se explicó en Representación musical, cada nota dista un tono de sus adyacentes, a excepción de mi-fa y si-do, que distan un semitono entre sí; y mientras que un sostenido aumenta un semitono, un bemol decrementa un semitono. De esta manera, “do sostenido” será igual que “re bemol”, y “mi sostenido” será igual que “fa” (natural).

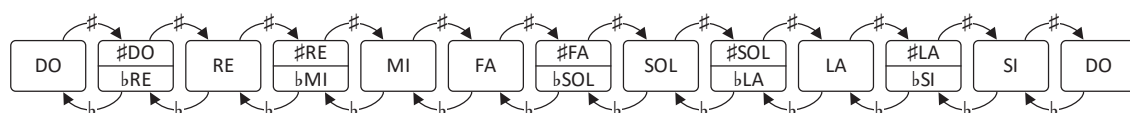


Figura 29 - Equivalencia de alteraciones
(Fuente: Elaboración propia)

El código del archivo *5-note_simplifier.py*³², trata esta problemática transformando todos los bemoles a su representación equivalente, de forma que sólo tengamos notas naturales (con becuadro), o sostenidas.

Finalmente, el archivo *6-recode.py*³³ tiene un simple *script* para separar o eliminar los multiplicadores de las notas. Como se comentó al principio de la sección, únicamente vamos a utilizar las notas (sin multiplicador) para analizar el comportamiento de las redes neuronales con secuencias temporales.

Adicionalmente, se elaboraron los *script 7-compare.py*³⁴ y *7-summary_notes.py*³⁵, para analizar el diccionario de palabras (notas), las ocurrencias de cada palabra, el porcentaje, etc.

²⁹ https://github.com/Tuxt/AutoScore/blob/master/01_preproceso/2-key_changer.py

³⁰ https://github.com/Tuxt/AutoScore/blob/master/01_preproceso/3-del_ties_and_tildes.py

³¹ https://github.com/Tuxt/AutoScore/blob/master/01_preproceso/4-data_extractor.py

³² https://github.com/Tuxt/AutoScore/blob/master/01_preproceso/5-note_simplifier.py

³³ https://github.com/Tuxt/AutoScore/blob/master/01_preproceso/6-recode.py

³⁴ https://github.com/Tuxt/AutoScore/blob/master/01_preproceso/7-compare.py

³⁵ https://github.com/Tuxt/AutoScore/blob/master/01_preproceso/7-summary_notes.py

2.6.2. Modelado

Como punto de partida, y para tener datos de referencia, en esta primera iteración se va a evaluar el rendimiento y precisión en redes neuronales recurrentes simples.

Como se explicó en Redes neuronales artificiales, es recomendable (en la mayoría de casos) utilizar una función de activación tangente hiperbólica (*tanh*) frente a ReLU para las capas recurrentes, debido a que, aunque ReLU tiene menor tendencia a desvanecer el gradiente (*vanishing gradient problem*), se comporta peor ante explosiones de gradiente (*exploding gradient problem*) ya que su salida no está acotada. Por ello, emplearemos una activación tangente hiperbólica en esta primera batería de pruebas. En cuanto al número de unidades (neuronas) a utilizar, ésta será una de las variables que analizaremos.

A esta capa le añadiremos una capa *dropout* al 20%, que no es más que una capa que descarta una pequeña cantidad de muestras en cada iteración del entrenamiento para prevenir el *overfitting*.

Finalmente, tendremos una capa *fully-connected* como capa de salida. Ésta es una simple capa cuyas neuronas implementan una función de propagación lineal y todas están conectadas con todas las neuronas de la capa previa. En Keras, a esta capa se le nombra *Dense layer*.

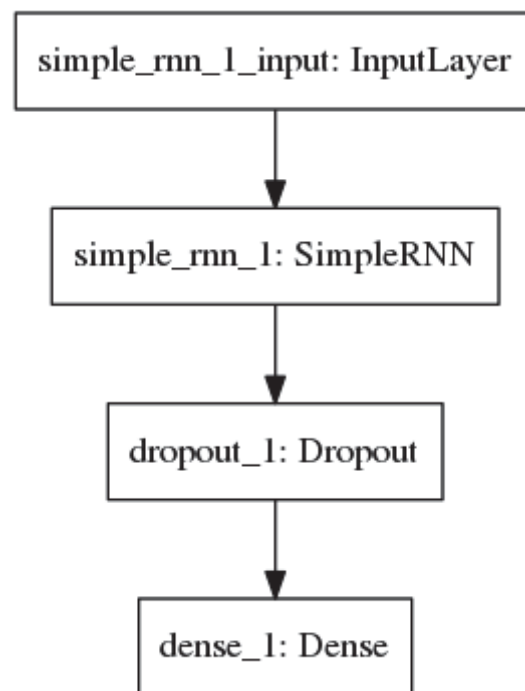


Figura 30 - Estructura de red inicial (Iteración 1)
(Fuente: Elaborado con Keras)

Con el objetivo de generar diferentes composiciones para una misma secuencia, y dotar así al sistema de un componente creativo, se van a seleccionar las notas en función

de la probabilidad. Por ello, necesitamos obtener en la salida las probabilidades de cada nota, por lo que la última capa tendrá una función de activación *softmax*. Además, la capa contendrá 33 unidades, puesto que es el número de notas obtenidas tras el preproceso realizado en la sección anterior y, por tanto, el número de salidas necesarias.

Se emplearán 10 *timesteps*. Estos son los pasos temporales anteriores que recibirá la red para generar un nuevo dato. Se ha decidido que sean 10 teniendo en cuenta que los datos son notas, y no letras, por lo que proporcionar una gran cantidad requerirá que el usuario aporte una gran composición inicial para poder generar nuevas notas. Mientras que, al utilizar 10 *timesteps*, es suficiente con proporcionar 10 notas para generar nuevo contenido.

Como se ha comentado anteriormente, uno de los hiperparámetros que compararemos será el número de unidades en la capa recurrente. Éste es un hiperparámetro crucial tanto en el entrenamiento como en el funcionamiento del sistema.

Un número pequeño de neuronas limitará la capacidad de aprendizaje y abstracción, actuando como cuello de botella. Por lo que, por mucha profundidad que tenga nuestra red, el aprendizaje de la misma será bastante pobre.

Por otro lado, un gran número de neuronas se traduce en un gran incremento en nuestro número de parámetros, por lo que el entrenamiento será mucho más costoso y lento. Además, el aumento de neuronas por capa, en muchos casos, puede resultar contraproducente. El aumento de la red en amplitud (más neuronas por capa) mejora los resultados hasta cierto punto, encontrando limitaciones que serían más fácilmente salvables si aumentáramos la red en profundidad (más capas) [46]. Mientras que una red de gran amplitud es buena memorizando, una red con profundidad es buena generalizando. La clave es encontrar el equilibrio entre amplitud y profundidad [47].

Otra de las características que variaremos es el tamaño de bloque (llamado *batch size*). Si recordamos el entrenamiento mediante el algoritmo de gradiente descendente, durante la propagación hacia atrás se estima el gradiente de la función *loss* y se actualizan los pesos de las neuronas de la red. Esto se realiza múltiples veces, y nos referimos a cada una de esas veces como “iteración”, en la cual utilizamos un bloque (*batch*) con B muestras de entrenamiento. Por otro lado, llamamos *epoch* al conjunto de iteraciones de entrenamiento necesarias para usar el *dataset* completo (compuesto por N muestras). De forma que, el número de iteraciones correspondientes a 1 *epoch* será:

$$n = \frac{N}{B}$$

Ecuación 16 - Relación iteraciones-muestras-batch size

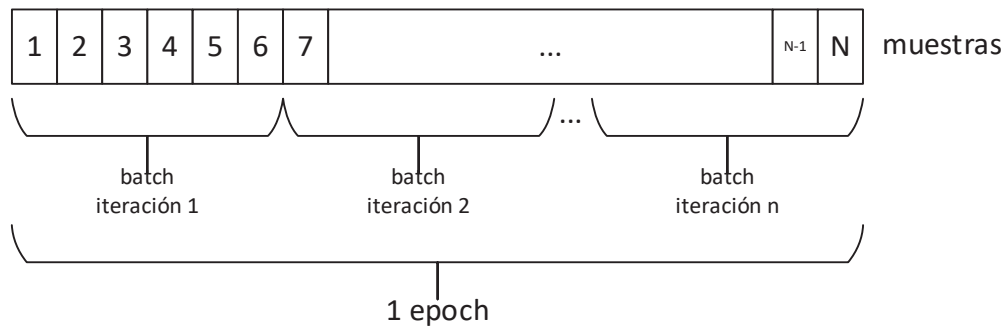


Figura 31 - Diagrama para un epoch
(Fuente: Elaboración propia)

Pues bien, el entrenamiento por gradiente descendente se puede realizar de 3 modos:

- Modo estocástico³⁶ ($B = 1$): El tamaño de bloque es de una sola muestra. Es decir, se entrena en cada iteración con una única muestra, y se producen N iteraciones ($n = N$) en 1 *epoch*, tantas como muestras hay.
- Modo bloque o *batch* ($B = N$): El tamaño del bloque es igual al del *dataset* completo. Por tanto, 1 *epoch* se compone de 1 *batch*, que abarca todas las muestras, y se realiza una sola iteración ($n = 1$) por *epoch*.
- Modo *mini-batch* ($1 < B < N$): El bloque contiene varias muestras, pero no abarca el *dataset* completo. El número de iteraciones para 1 *epoch* vendrá determinado por la Ecuación 16 - Relación iteraciones-muestras-batch size.

El modo *batch* solo resulta viable cuando manejamos *datasets* pequeños, ya que para cada actualización debe calcular el promedio de todas las muestras del bloque (que se compone del *dataset* completo), lo que consume una gran cantidad de memoria. Por otro lado, su mayor inconveniente es que, al calcular el promedio sobre todo el conjunto de datos, el gradiente siempre tiende al mínimo de forma óptima, lo cual es adecuado para optimizar una superficie convexa, pero tiene a atascarse (especialmente en puntos de ensilladura). Además, su entrenamiento es especialmente lento debido a que, como se ha mencionado, tras procesar todo el conjunto de datos, se realiza una única actualización.

En cambio, el modo estocástico realiza una actualización por cada muestra, no por el promedio de un conjunto de muestras. Esto hace que el entrenamiento sea más rápido, pero que el gradiente sea más ruidoso, lo cual hace que no tienda al mínimo de forma óptima y sea terriblemente ineficiente, pero este ruido ayuda a que el algoritmo salga de mínimos locales en los que se queda atascado [48].

³⁶ Se hace referencia al gradiente descendente estocástico (SGD) puro, en el que se utiliza una única muestra. El algoritmo SGD tiene más implicaciones, aparte de utilizar una única muestra por iteración, como la inicialización de los pesos o el muestreo estocástico. Por lo que, algunas aplicaciones implementan SGD con mini-batches, pero sigue siendo SGD. El modo batch solo implica la inicialización de los pesos.

El modo *mini-batch* pretende encontrar un equilibrio entre estos dos modos, de forma que el gradiente tenga parte de ruido que le permita salir de mínimos locales y puntos de ensilladura, pero con la mayor convergencia que le proporciona realizar actualizaciones en base al promedio de un conjunto de datos, en lugar de un único dato en cada iteración.

Por ello, también se experimenta con diferentes tamaños de bloque con el objetivo de encontrar un valor óptimo.

Para realizar estas pruebas, se ha elaborado un *script* parametrizado *test001_SimpleRNN.py*³⁷ que automatiza el proceso de entrenamiento y evaluación para los parámetros indicados. Tales como el *dataset*, el número de *timesteps*, unidades en la capa recurrente, porcentaje de *dropout*, función de cálculo *loss*, tamaño de bloque, etc.

Se han realizado pruebas con 16, 32, 64 y 128 neuronas en la capa recurrente, así como 16, 32, 64 y 128 muestras por bloque. Las métricas obtenidas para evaluar los modelos corresponden al valor *loss* y la precisión obtenida en cada *epoch*, realizándose 100 *epochs*. El *script* de entrenamiento guarda el estado interno de la red en cada *epoch*, siempre y cuando el valor *loss* mejore respecto a anteriores *epochs*. Como función de cálculo de error *loss* se va a utilizar la entropía cruzada.

El archivo *datamanager.py*³⁸ provee funciones para cargar y preparar los datos, así como generar los diccionarios para convertir las notas en números y viceversa. Este archivo se encarga de leer el *dataset* y realizar el troceo del mismo en secuencias según el número de *timesteps*. De forma que, si tenemos una composición $[x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9]$ y utilizamos 5 *timesteps*, se generarán las muestras:

$$x = \begin{bmatrix} [x_1, x_2, x_3, x_4, x_5] \\ [x_2, x_3, x_4, x_5, x_6] \\ [x_3, x_4, x_5, x_6, x_7] \\ [x_4, x_5, x_6, x_7, x_8] \end{bmatrix} \quad y = \begin{bmatrix} x_6 \\ x_7 \\ x_8 \\ x_9 \end{bmatrix}$$

Posteriormente, los valores de *y* son transformados a codificación *One-Hot*. Las muestras se dividen en tres conjuntos: entrenamiento (60%), validación (20%) y test (20%).

2.6.3. Evaluación

Cada una de las pruebas se entrena con el *dataset* de entrenamiento, y para cada *epoch* se evalúa tanto con el conjunto de entrenamiento como con el de validación. Esto nos permite comparar los resultados obtenidos con datos conocidos por el modelo (entrenamiento) y con datos no conocidos previamente (validación), lo que nos permitirá comprobar si el modelo está generalizando correctamente o está produciendo *overfitting*.

³⁷ <https://goo.gl/ENVH3J>

³⁸ https://github.com/Tuxt/AutoScore/blob/master/02_modelos/datamanager.py

En base a estos resultados podemos decidir los mejores hiperparámetros para el modelo. Y finalmente evaluar el rendimiento general con los datos de test.

Tras realizar 100 *epochs* de entrenamiento para cada una de las pruebas, obtenemos los resultados mostrados a continuación.

2.6.3.1. 16 Unidades

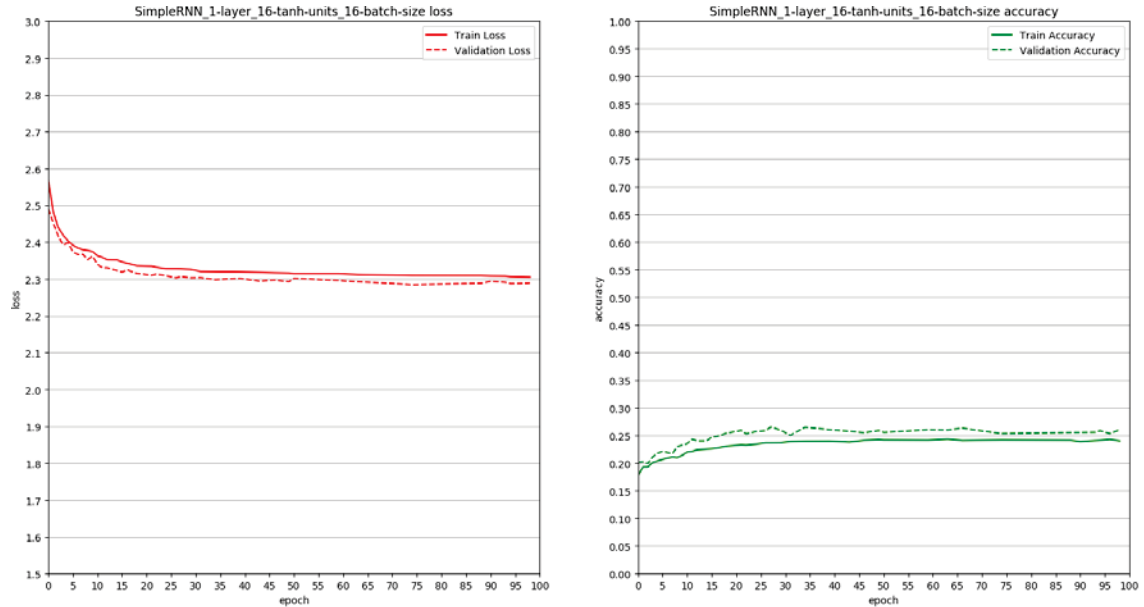


Figura 32 - Entrenamiento SimpleRNN 1-layer: 16 neuronas, 16 muestras/bloque

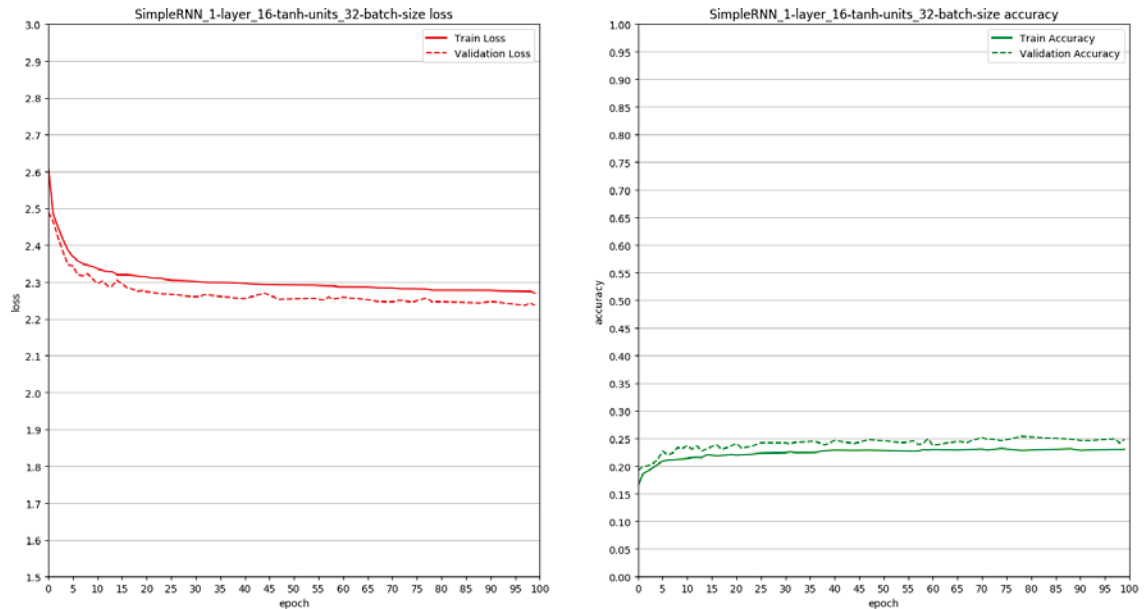


Figura 33 - Entrenamiento SimpleRNN 1-layer: 16 neuronas, 32 muestras/bloque

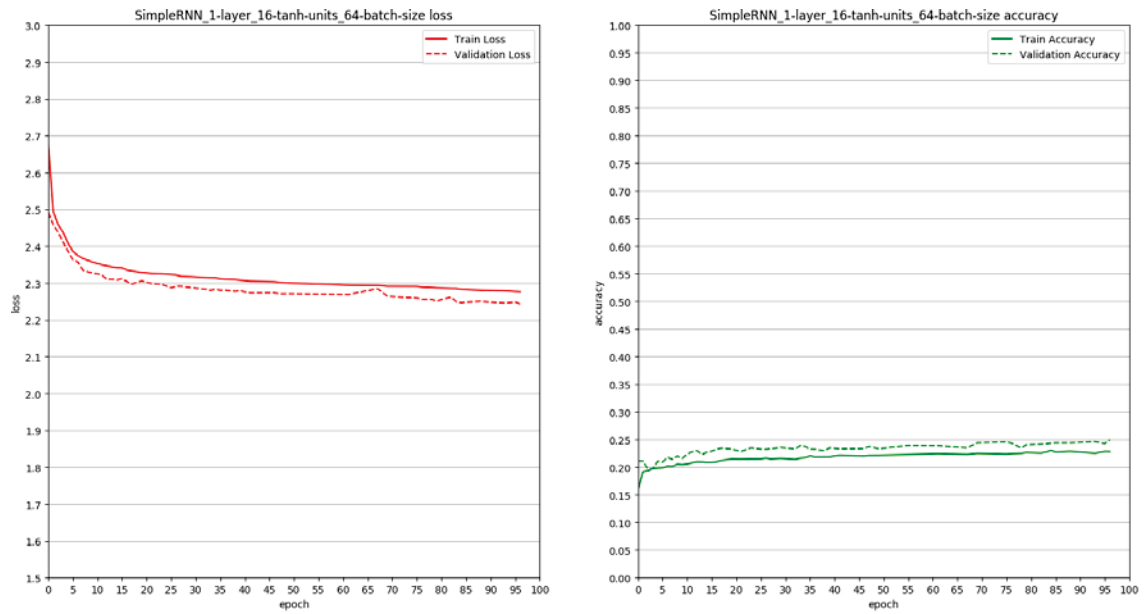


Figura 34 - Entrenamiento SimpleRNN 1-layer: 16 neuronas, 64 muestras/bloque

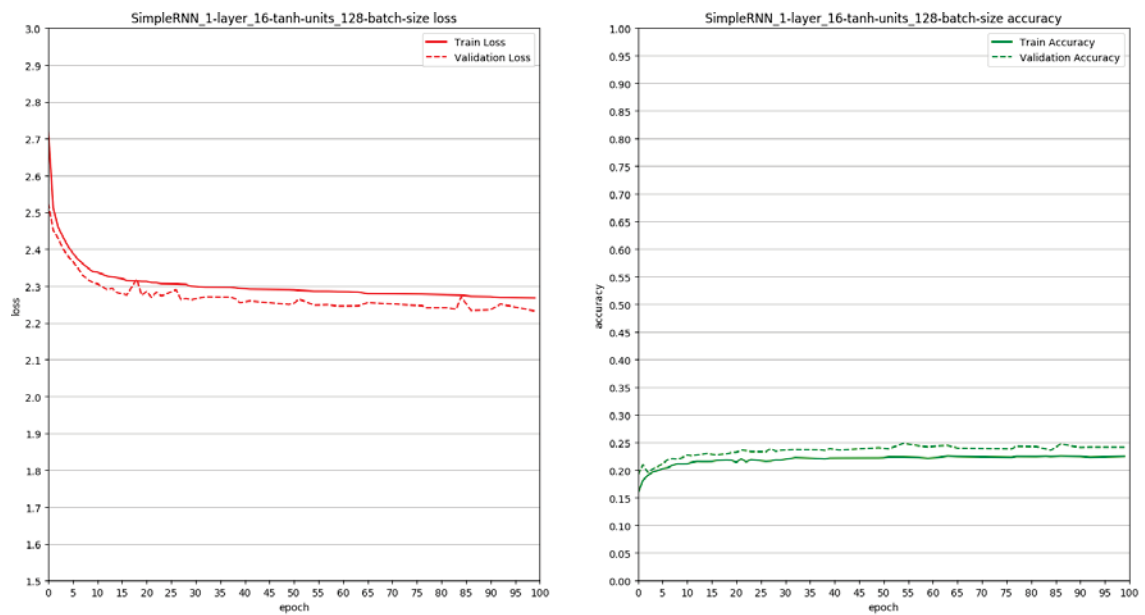


Figura 35 - Entrenamiento SimpleRNN 1-layer: 16 neuronas, 128 muestras/bloque

Los resultados para 16 unidades neuronales son similares para todos los tamaños de bloque evaluados, e inferiores al resto de resultados, por lo que parece indicar que el escaso número de neuronas limita el aprendizaje.

2.6.3.2. 32 unidades

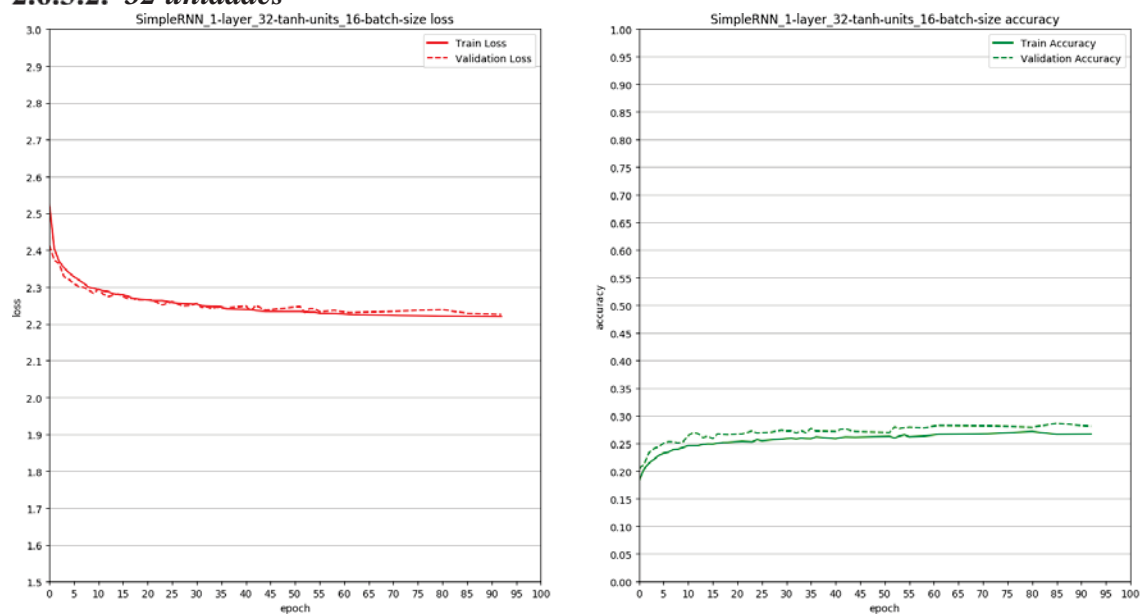


Figura 36 - Entrenamiento SimpleRNN 1-layer: 32 neuronas, 16 muestras/bloque

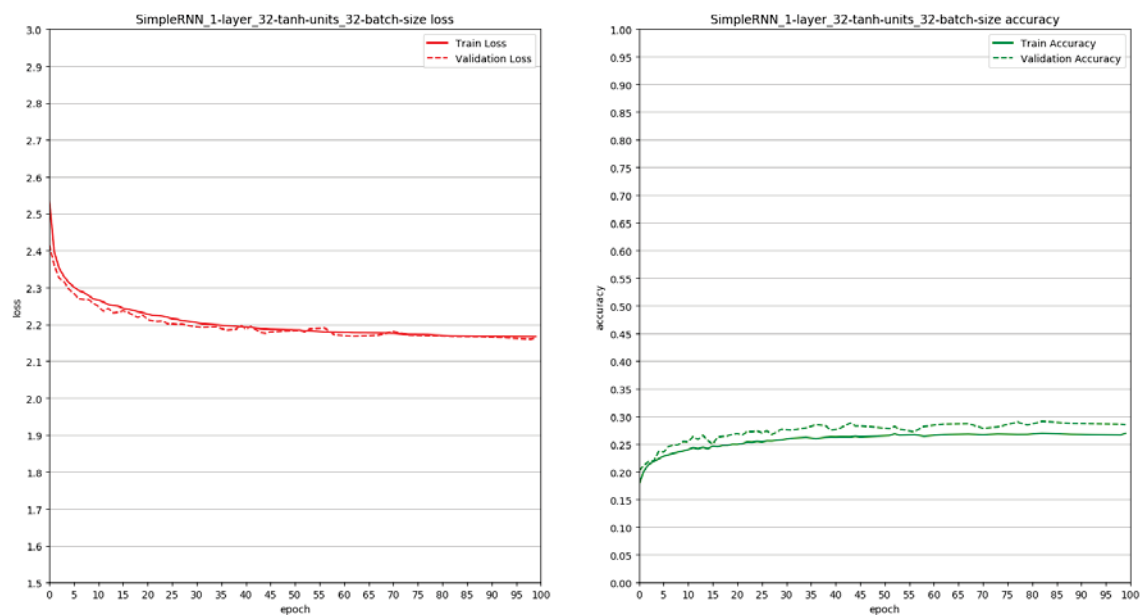


Figura 37 - Entrenamiento SimpleRNN 1-layer: 32 neuronas, 32 muestras/bloque

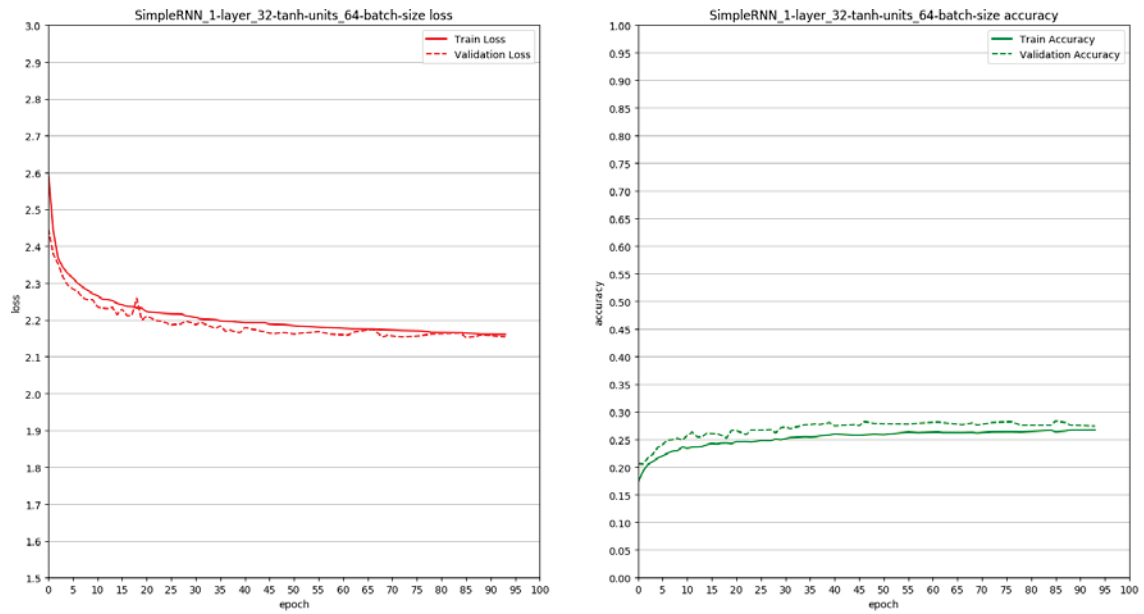


Figura 38 - Entrenamiento SimpleRNN 1-layer: 32 neuronas, 64 muestras/bloque

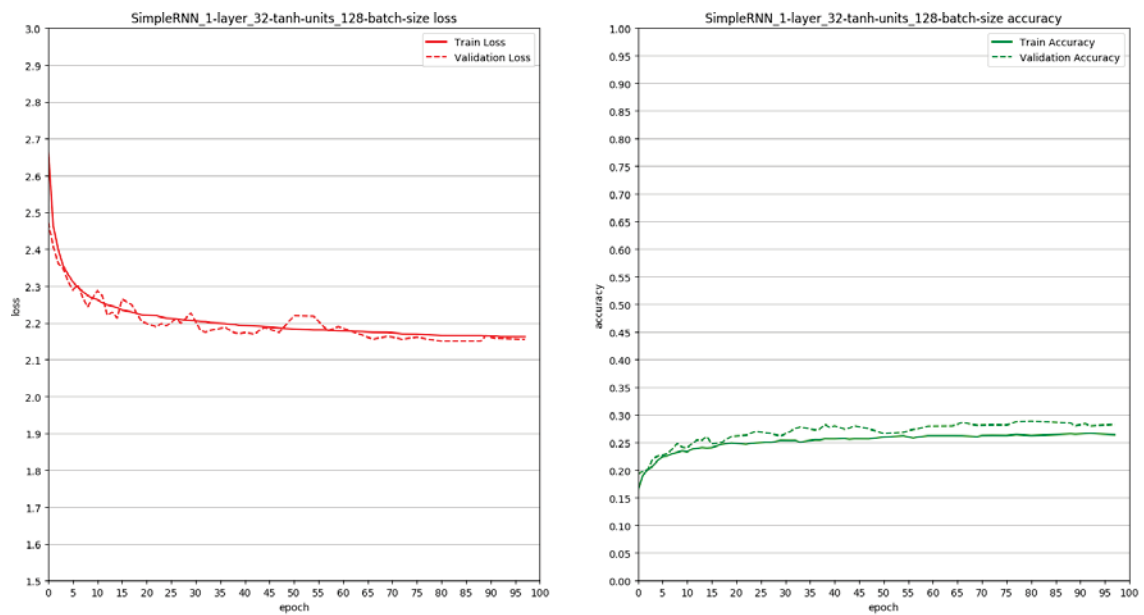


Figura 39 - Entrenamiento SimpleRNN 1-layer: 32 neuronas, 128 muestras/bloque

Para 32 unidades neuronales, los resultados son mejores en relación a los anteriores, mejorando la precisión aproximadamente un 5%. Los resultados *loss* obtenidos con bloques de 16 muestras son considerablemente peores en este conjunto de pruebas, aunque esto no se ve reflejado en el porcentaje de precisión. Se puede ver como el final de la gráfica está incompleto debido a que en los últimos *epochs* no se han guardado datos

ya que el valor *loss* no mejoraba los resultados anteriores. Aunque es un resultado común, el hecho de que se produzcan múltiples *epochs* seguidos sin mejora podría indicar que se ha alcanzado un mínimo, aunque en este caso, al ser 7 *epochs* no podemos asegurarlo. En cualquier caso, el algoritmo parece comportarse mejor con bloques de mayor tamaño.

2.6.3.3. 64 unidades

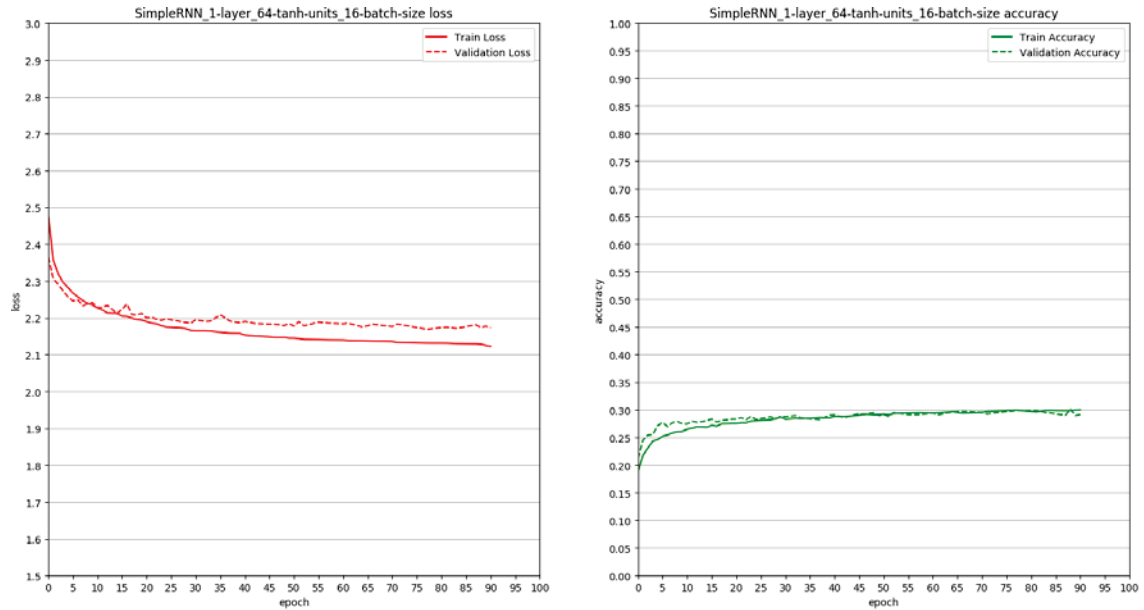


Figura 40 - Entrenamiento SimpleRNN 1-layer: 64 neuronas, 16 muestras/bloque

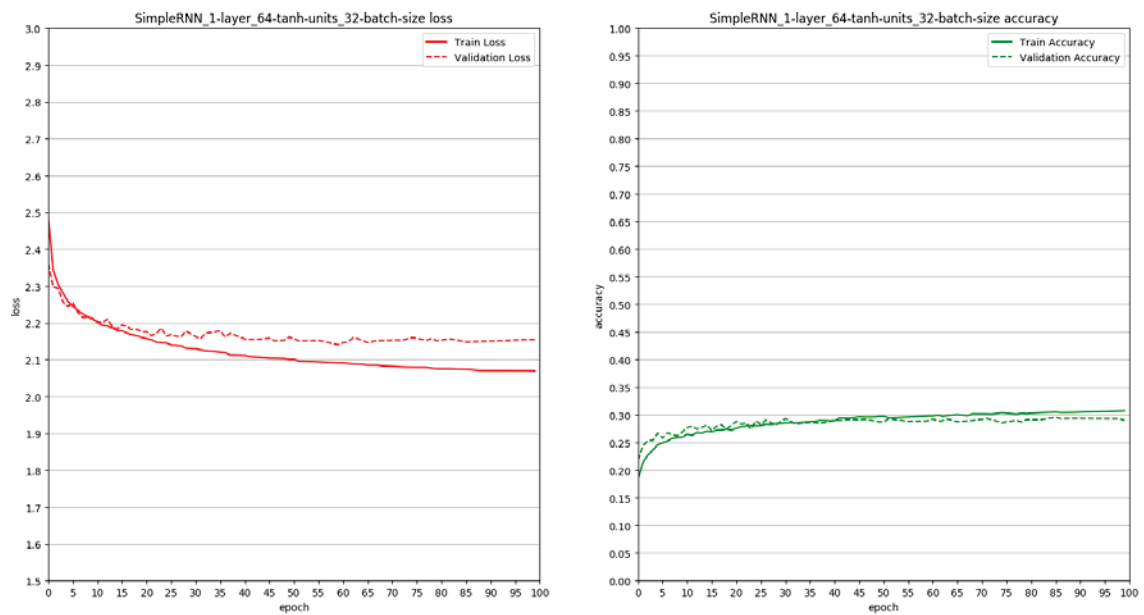


Figura 41 - Entrenamiento SimpleRNN 1-layer: 64 neuronas, 32 muestras/bloque

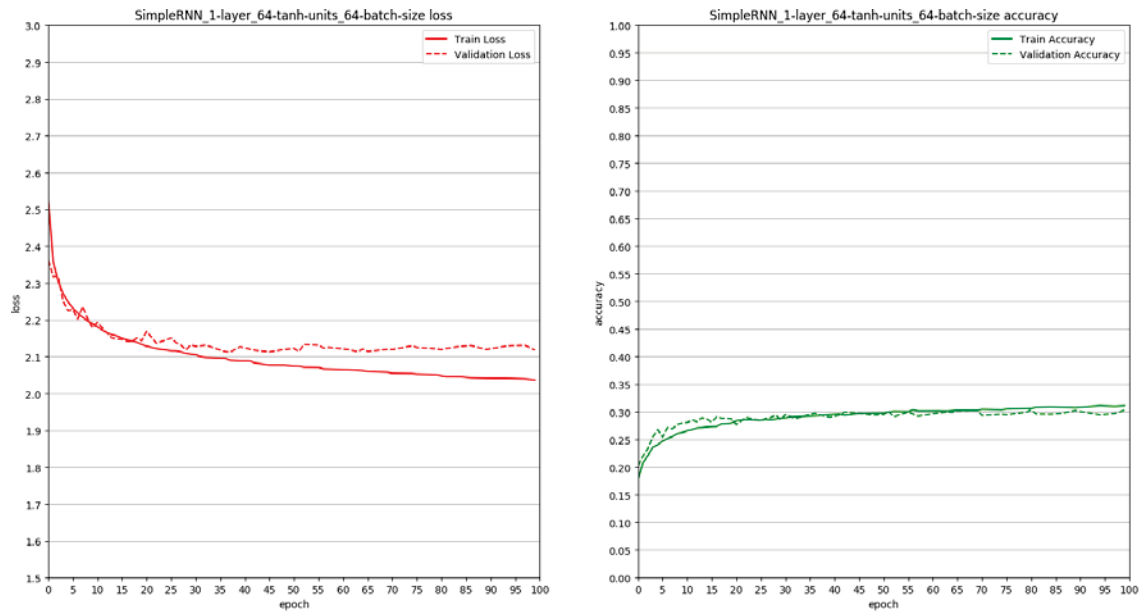


Figura 42 - Entrenamiento SimpleRNN 1-layer: 64 neuronas, 64 muestras/bloque

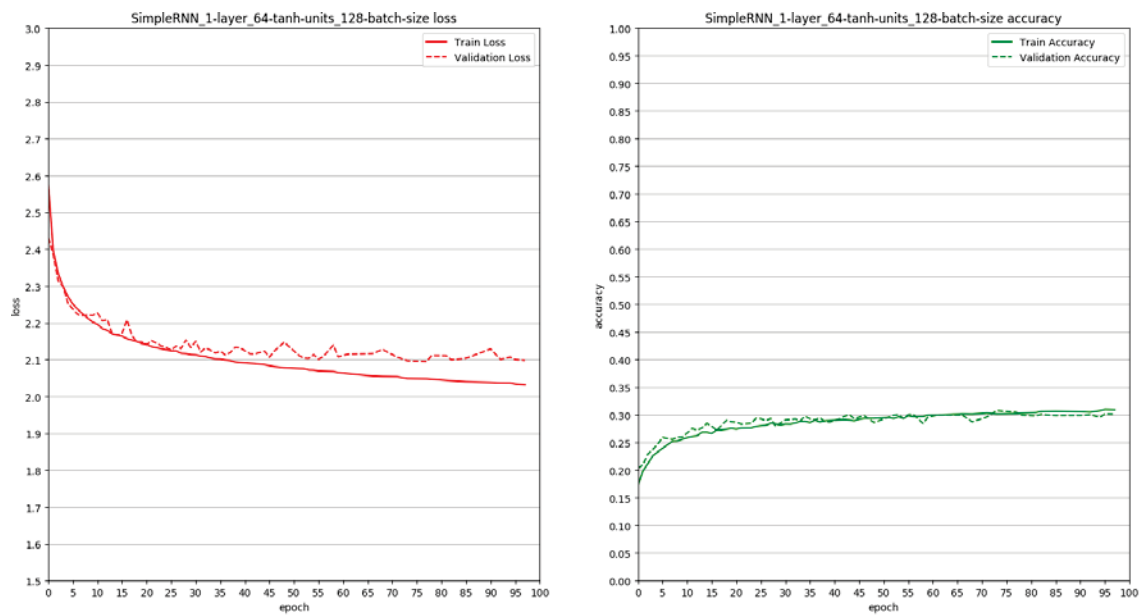


Figura 43 - Entrenamiento SimpleRNN 1-layer: 64 neuronas, 128 muestras/bloque

Lo mismo ocurre con los resultados obtenidos en las pruebas de 64 unidades neuronales. Los valores *loss* obtenidos con bloques de 16 muestras son mayores. Las pruebas con 32, 64 y 128 muestras por bloque son similares y ligeramente mejores que las obtenidas en la red de 32 neuronas. En estas tres pruebas se obtiene una precisión del 30% tanto en el conjunto de entrenamiento como en el de validación. Por otro lado, el

valor *loss* es ligeramente mejor en las pruebas de 64 y 128 muestras por bloque, mientras que las de 32 muestras por bloque tienen mayor *loss*. En las gráficas de *loss*, el trazo correspondiente a los datos de validación se separa ligeramente del *loss* correspondiente a los datos de entrenamiento, lo que indica que comienza a producirse *overfitting*.

2.6.3.4. 128 unidades

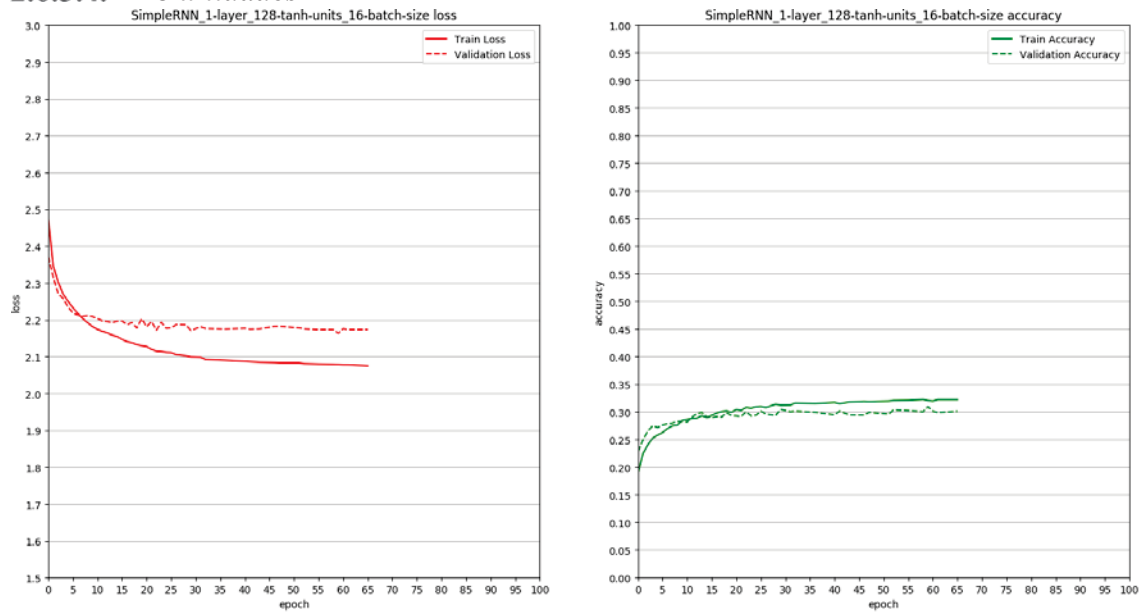


Figura 44 - Entrenamiento SimpleRNN 1-layer: 128 neuronas, 16 muestras/bloque

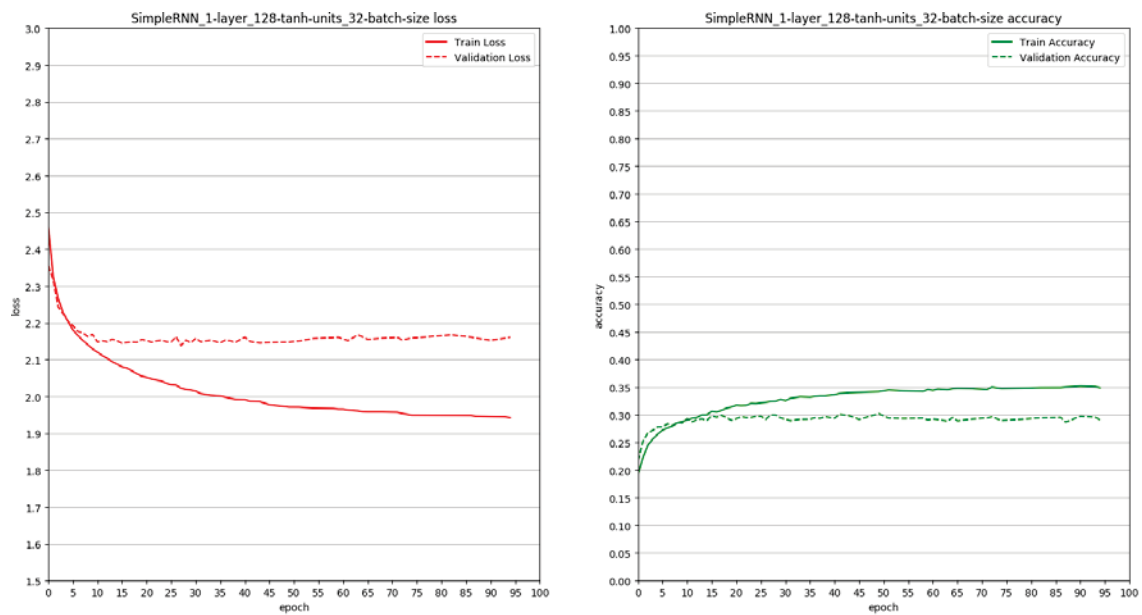


Figura 45 - Entrenamiento SimpleRNN 1-layer: 128 neuronas, 32 muestras/bloque

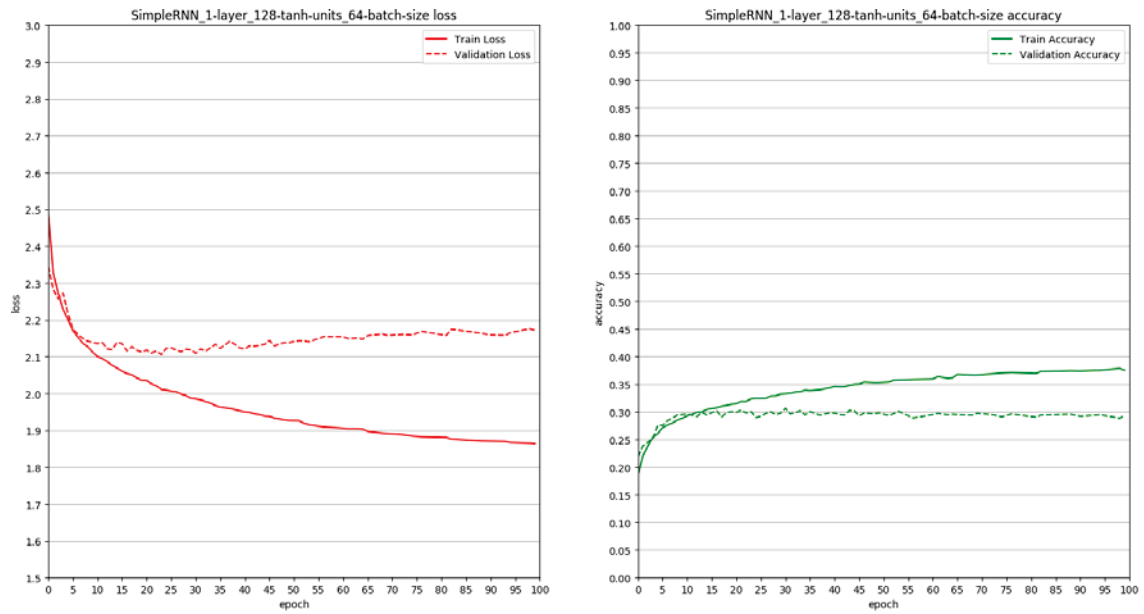


Figura 46 - Entrenamiento SimpleRNN 1-layer: 128 neuronas, 64 muestras/bloque

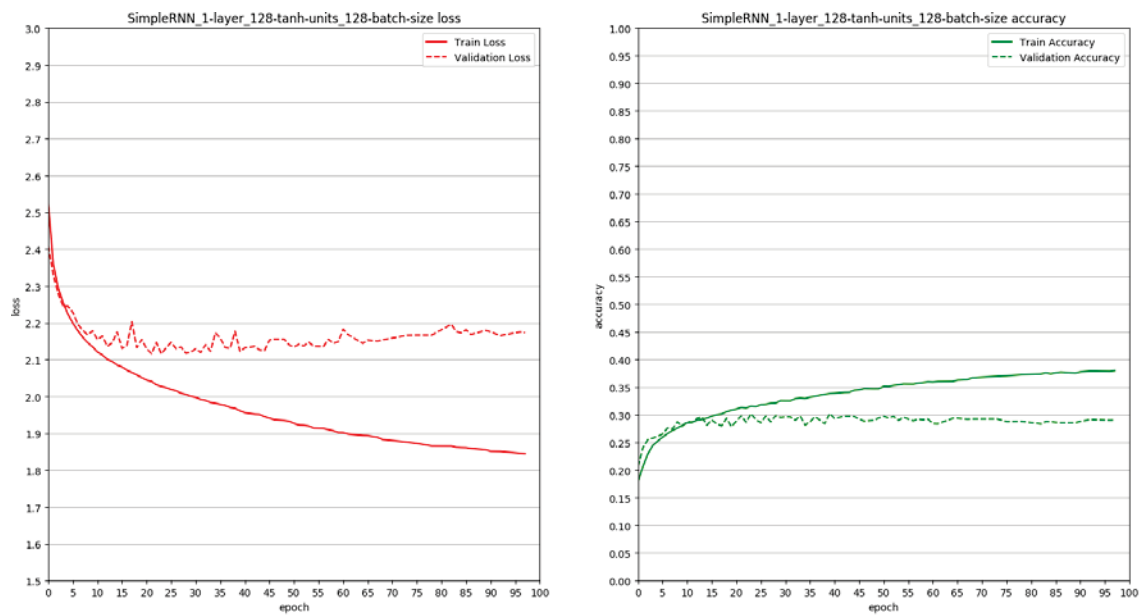


Figura 47 - Entrenamiento SimpleRNN 1-layer: 128 neuronas, 128 muestras/bloque

El *overfitting* se ve claramente y desde muy temprano en las pruebas realizadas con 128 neuronas, donde se hace evidente a partir de los *epochs* 10-15, el valor *loss* del conjunto de entrenamiento sigue descendiendo, mientras que el del conjunto de validación se mantiene o, en algunos casos, incluso aumenta. En la ejecución de bloque con 16 muestras los datos se detienen en el *epoch* 65, lo que indica que el valor *loss* no

mejora en las siguientes iteraciones (ya sea porque se ha alcanzado un mínimo, el algoritmo diverge empeorando los resultados, etc.). Por ello, se descarta definitivamente utilizar un tamaño de bloque de 16 muestras.

Por supuesto, los resultados obtenidos en estas últimas pruebas con 128 neuronas son, en su mayoría, inválidos para nuestro sistema, debido al *overfitting*. Por lo que deberemos emplear el estado de un modelo con 128 neuronas en un momento previo al *overfitting* o un modelo con 64 neuronas, que hemos visto que obtiene los mejores resultados, de las pruebas anteriores.

Con esto en mente, para cualquiera de los resultados obtenidos en la red de 128 neuronas (los resultados son similares), si tomamos como límite el *epoch* 15, en el que obtenemos un 30% de precisión aproximadamente, los valores *loss* de entrenamiento³⁹ y validación están aproximadamente en 2.06 y 2.12 respectivamente (en el mejor de los casos, bloques de 64 muestras) como se indica en la Figura 48 - Límite de overfitting.

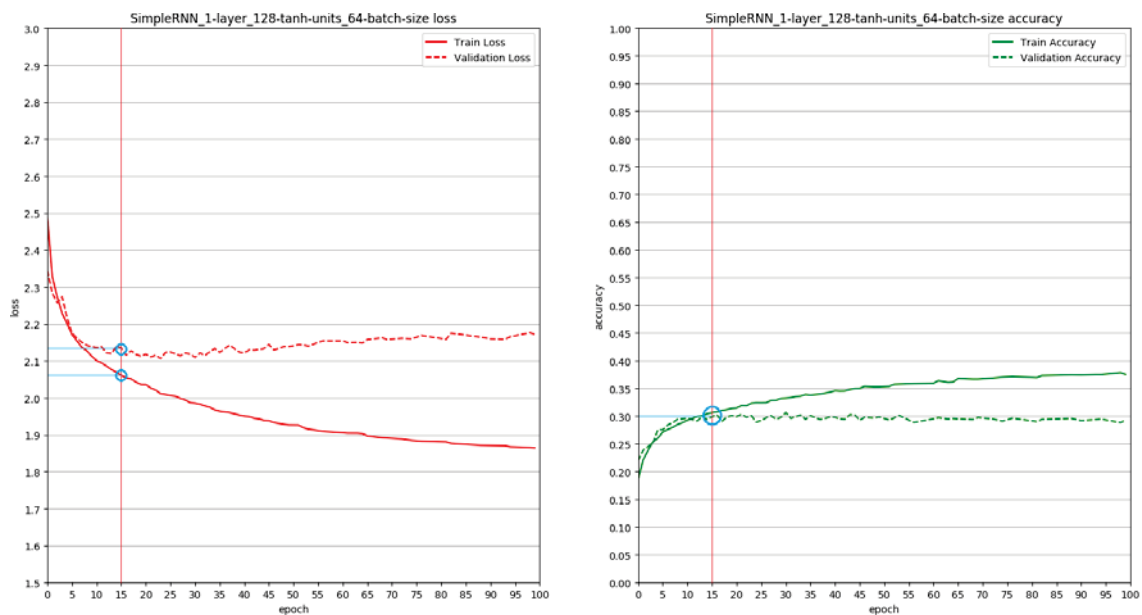


Figura 48 - Límite de overfitting
(128 unidades neuronales – 64 muestras por bloque)

Si trasladamos estos mismos valores (*loss* 2.12 y 2.06, precisión 30%) a las gráficas de 64 unidades neuronales, con 64 y 128 muestras por bloque, vemos que dichos valores son igualados o mejorados en varios puntos.

³⁹ El valor que realmente nos interesa es el de validación, ya que es el que verdaderamente muestra las capacidades de la red con datos no vistos en el entrenamiento.

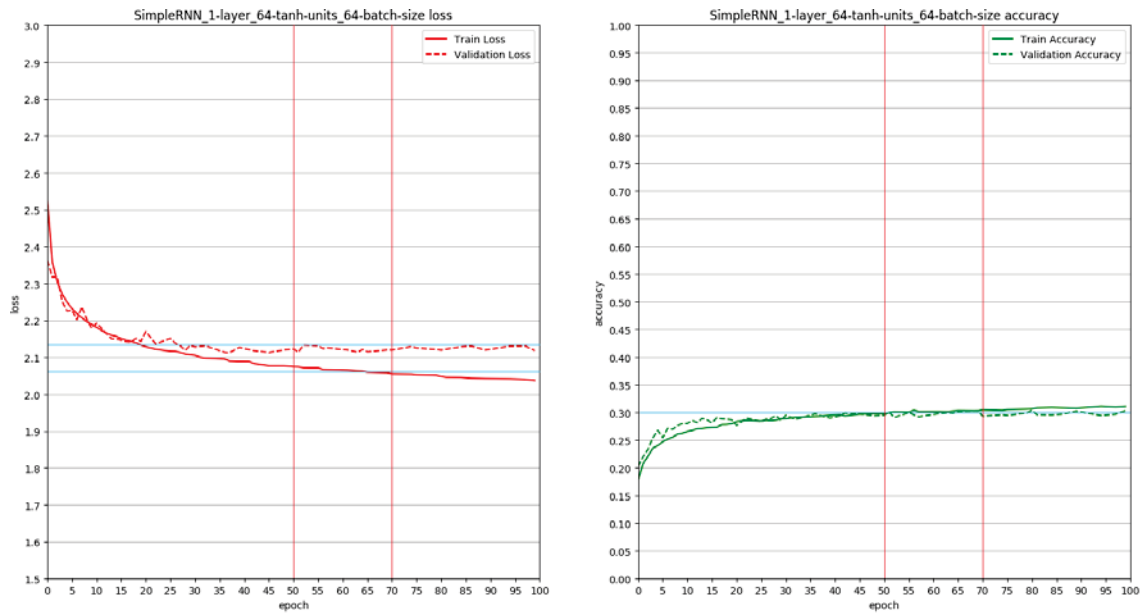


Figura 49 - Mejora con 64 neuronas y 64 muestras por bloque

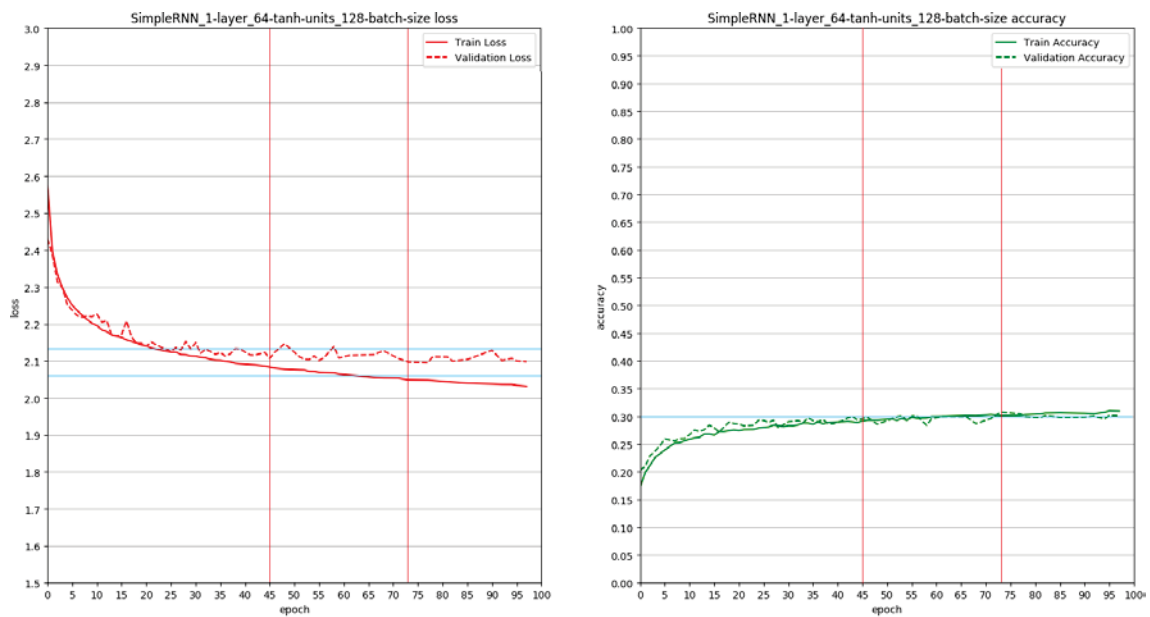


Figura 50 - Mejora con 64 neuronas y 128 muestras por bloque

En las figuras anteriores Figura 49 - Mejora con 64 neuronas y 64 muestras por bloque y Figura 50 - Mejora con 64 neuronas y 128 muestras por bloque, aparecen marcados en azul los resultados obtenidos con 128 neuronas. En rojo, algunos *epochs* cuyos valores igualan o mejoran los obtenidos con 128 neuronas. Ante estos resultados, se decide

utilizar el modelo generado en *epoch* 73 por la red con 64 neuronas, entrenada con bloques de 128 muestras.

Ante resultados iguales, es preferible el uso de redes con menos neuronas, puesto que el coste computacional y la memoria necesaria es menor en redes con menores dimensiones.

Si tenemos en cuenta la estructura de la red presentada en la Figura 30 - Estructura de red inicial (Iteración 1), tenemos 2 capas con pesos neuronales: SimpleRNN (recurrente) y Dense, ya que la capa de entrada no realiza operaciones y la capa de *Dropout* no contiene pesos neuronales que multiplicar.

Puesto que tenemos un único dato de entrada en cada instante temporal t , si la capa recurrente tiene N neuronas, cada neurona tendrá $N + 2$ entradas. Esto es N entradas que corresponden con la salida de las N neuronas en el instante $t - 1$, más la entrada correspondiente al *bias*, más la entrada correspondiente al dato del instante temporal t (actual). Puesto que tenemos N neuronas en la capa recurrente con $N + 2$ entradas cada una, esto hace un total de $N \cdot (N + 2)$ pesos en la capa recurrente.

$$\text{Tamaño}(W_{\text{SimpleRNN}}) = N \cdot (N + 2)$$

Ecuación 17 - Parámetros por capa SimpleRNN

En la capa Dense, tenemos 33 neuronas siempre (puesto que el diccionario tiene 33 palabras y se codifican como *One-Hot* vector). Cada neurona tendrá $N + 1$ entradas, correspondiente a los N valores de la capa recurrente, y un valor del *bias*. Siendo 33 neuronas, tenemos un total de $33 \cdot (N + 1)$ pesos en la capa Dense.

$$\text{Tamaño}(W_{\text{Dense}}) = 33 \cdot (N + 1)$$

Ecuación 18 - Parámetros por capa Dense

Haciendo un total de parámetros por red de:

$$\text{Parámetros} = N \cdot (N + 2) + 33 \cdot (N + 1)$$

Ecuación 19 - Parámetros por red (iteración 1)

Por lo que, para la red de 64 neuronas recurrentes, tendremos un total de 6369 parámetros totales. Mientras que, en la red de 128 neuronas recurrentes, el número de parámetros asciende a 20897.

Cabe mencionar que los resultados de esta red son muy similares, por lo que otros modelos generados tendrán probablemente el mismo comportamiento; y el modelo seleccionado no necesariamente tiene que ser el mejor, ya que los resultados (*loss*, precisión) se calculan para un conjunto de muestras concreto. Para otro conjunto nuevo los resultados pueden variar ligeramente.

En cualquier caso, los resultados obtenidos con el modelo de 1 capa recurrente simple son, para cualquiera de las pruebas realizadas, excesivamente pobres.

2.6.4. Elaboración de aplicación web

A continuación se detallan las fases seguidas para la elaboración de la aplicación web, a la que se ha llamado “AutoScore”.

2.6.4.1. Análisis de requisitos

Dado el enfoque ágil que se está siguiendo en el proyecto, en el que se prioriza la funcionalidad frente a la documentación extensiva (según el manifiesto ágil⁴⁰), y debido a la gran extensión que conlleva una especificación de requisitos completa (que podría perfectamente componer un documento independiente), se ha decidido detallar en este apartado las secciones que se han considerado más relevantes, para el caso que nos ocupa, del estándar IEEE 830-1998 [49].

2.6.4.1.1. Perspectiva del producto

El producto permitirá la introducción de notas y la composición y generación musical mediante el uso de modelos de aprendizaje automático, permitiendo visualizar las composiciones en un pentagrama, así como escucharlas y guardarlas. Para la visualización, almacenamiento y reproducción de composiciones el sistema a implementar hará uso de la plataforma Flat⁴¹.

El producto deberá poder almacenar, en el sistema operativo sobre el que funcione, archivos con código ABC, así como ejecutar el binario correspondiente a la traducción de código ABC a MIDI.

La interfaz de usuario debe permitir al usuario la inserción de notas mediante botones, así como la selección del modo de funcionamiento para la generación de notas (la nota más probable, o selección aleatoria probabilística).

2.6.4.1.2. Restricciones

Como se ha comentado, el sistema hará uso de la plataforma Flat para el almacenamiento, visualización y reproducción de las composiciones. El sistema deberá desarrollarse en lenguaje Python. Deberá ser funcional tanto en ordenadores personales

⁴⁰ <http://agilemanifesto.org/iso/es/manifesto.html>

⁴¹ <https://flat.io>

tradicionales como en dispositivos táctiles (*smartphones*, *tablets*...). El tiempo de respuesta del sistema no debe exceder los 3 segundos. El usuario no podrá introducir datos mediante teclado, con el fin de restringir la entrada de datos y evitar posibles erratas; sólo podrá introducir datos mediante botones. Los botones que correspondan a notas no manejadas por el modelo neuronal deberán estar deshabilitadas. Para la generación de notas, el sistema debe recibir un mínimo de 10 notas insertadas por el usuario.

Para el uso del producto es imprescindible la conexión a internet. Debido a que la generación de música y la transformación a MIDI que se realiza en el lado del servidor, como la reproducción online y representación en una plataforma externa, la conexión a internet es un elemento clave.

2.6.4.1.3. *Requisitos de interfaz*

- **RI1:** El sistema debe permitir la introducción de notas de forma gráfica pulsando sobre un pentagrama.
- **RI2:** El sistema debe mostrar el código ABC asociado a las notas introducidas.
- **RI3:** El sistema debe permitir seleccionar la alteración de las notas antes de introducirlas.
- **RI4:** El sistema debe permitir borrar la última nota introducida.
- **RI5:** El sistema debe permitir borrar todas las notas introducidas.

2.6.4.1.4. *Requisitos funcionales*

- **RF1:** El sistema debe permitir guardar el estado de la composición.
- **RF2:** El sistema debe permitir mostrar la composición gráficamente en un pentagrama.
- **RF3:** El sistema debe permitir reproducir una composición online.
- **RF4:** El sistema debe permitir descargar una composición en formato MIDI.
- **RF5:** El sistema debe permitir generar nuevas notas, a partir de una composición insertada.
- **RF6:** El sistema debe permitir indicar el número de notas que quieren ser generadas.

2.6.4.1.5. *Requisitos no funcionales*

- **RNF1:** El sistema debe generar salidas con un tiempo de respuesta inferior a 3 segundos.
- **RNF2:** El sistema debe ser completamente funcional tanto en ordenadores personales tradicionales como en dispositivos táctiles, *smartphones* y *tablets*.
- **RNF3:** El sistema debe tener un diseño *responsive*.
- **RNF4:** El sistema debe ser accesible en idioma español.

2.6.4.1.6. *Otros requisitos*

- **OR1:** El sistema debe restaurar el estado de la sesión del usuario al entrar, si existiese algún estado guardado.
- **OR2:** El sistema debe permitir borrar la sesión actual.

2.6.4.1.7. Seguridad

- **Para usuarios:** El sistema no requiere autenticación ni maneja ni almacena datos sensibles.
- **Para el sistema:** Los datos generados por los usuarios deben ser tratados y comprobados, con el fin de prevenir que pueda causar cualquier tipo de fallo en el sistema.

2.6.4.2. Diseño de la aplicación

2.6.4.2.1. Descripción del propósito del sistema

El sistema dará acceso al servicio de generación de y composición de música. En el sistema interactúa el usuario que envía composiciones para que se generen nuevas notas haciendo uso del modelo neuronal del sistema. También podrá descargar la composición actual en formato MIDI, guardar el estado de la composición actual y reproducirla. El sistema interactuará con una plataforma externa llamada Flat, que se muestra en la Figura 51 - Diagrama de contexto (iteración 1) como otra entidad.

2.6.4.2.2. Diagrama de contexto

La Figura 51 - Diagrama de contexto (iteración 1) nos da una vista de alto nivel del sistema, definiendo los límites entre el sistema y su entorno, y cómo interactúa con el resto de entidades:

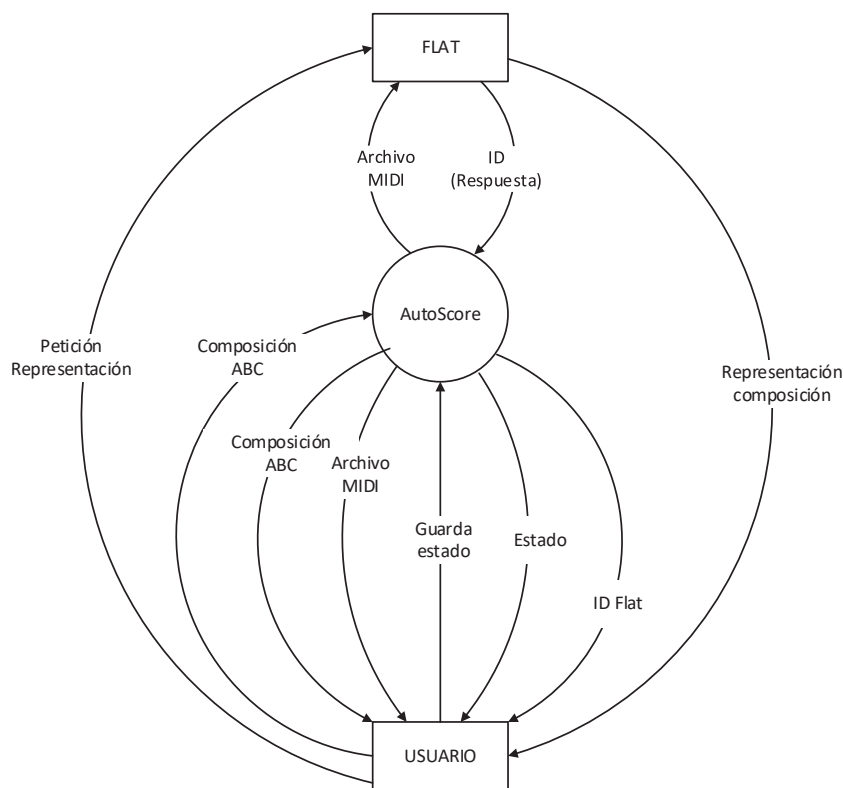


Figura 51 - Diagrama de contexto (iteración 1)
(Fuente: Elaboración propia)

En el diagrama se representa al usuario como entidad principal, y la plataforma Flat como otro interactivo externo.

El usuario puede enviar composiciones en formato ABC, así como recibir composiciones en formato ABC o MIDI. También puede solicitar guardar el estado y recibir el estado actual para ser guardado en el cliente. El sistema puede enviar archivos MIDI a la plataforma Flat, y recibe identificadores de las composiciones. Dichos identificadores pueden ser enviados también al usuario, mediante los cuales recibirán una representación de la composición directamente desde la plataforma Flat, sin pasar por el sistema.

2.6.4.2.3. Lista de acontecimientos

A continuación, se enumeran los acontecimientos posibles y se ilustra el comportamiento del *software* como consecuencia de esos acontecimientos externos que desencadenan una serie de acciones.

- AC.1: El usuario solicita guardar el estado.

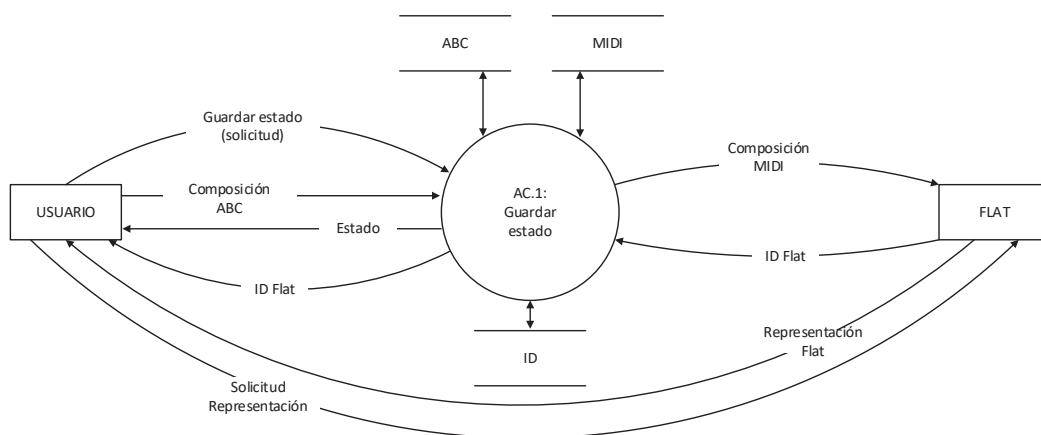


Figura 52 - Acontecimiento 1
(Fuente: Elaboración propia)

- AC.2: El usuario solicita guardar la composición en formato MIDI.

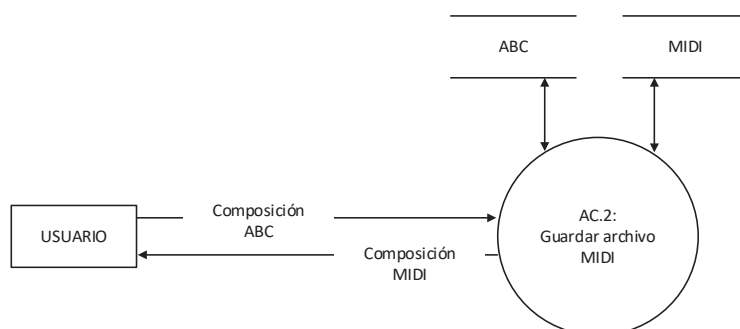


Figura 53 - Acontecimiento 2
(Fuente: Elaboración propia)

-
- ```
graph LR
 ABC[ABC] <--> AC3((AC.3: Generar composición))
 MIDI[MIDI] <--> AC3
 AC3 -- "Composición ABC" --> USUARIO[USUARIO]
 AC3 -- "ID Flat" --> USUARIO
 AC3 -- "Composición MIDI" --> FLAT[FLAT]
 FLAT -- "ID Flat" --> AC3
 USUARIO -- "Petición Representación" --> AC3
 AC3 -- "Representación Flat" --> USUARIO
```

A continuación, tenemos los procesos anteriores a un nivel de detalle más bajo:

- 
- ```

graph LR
    USUARIO[USUARIO]
    subgraph ABC [ABC]
        Recibir_ABC((Recibir ABC))
        Gestion_estado((Gestion estado))
        Enviar_ID_ABC((Enviar ID))
    end
    subgraph MIDI [MIDI]
        Generar_MIDI((Generar MIDI))
        Enviar_MIDI((Enviar MIDI))
        Recibir_ID_MIDI((Recibir ID))
        ID_MIDI[ID]
    end
    subgraph FLAT [FLAT]
        FLAT[FLAT]
    end

    USUARIO -- "Solicitud Representación" --> ABC
    ABC -- "Composición ABC" --> USUARIO
    ABC --> MIDI
    MIDI -- "Composición MIDI" --> ABC
    MIDI --> FLAT
    FLAT -- "ID" --> MIDI
    MIDI -- "ID" --> ABC
    ABC -- "Estado" --> MIDI
    MIDI -- "Enviar ID" --> ABC
    MIDI -- "Guardar estado" --> ABC
  
```

56

- AC.2: El usuario solicita guardar la composición en formato MIDI.

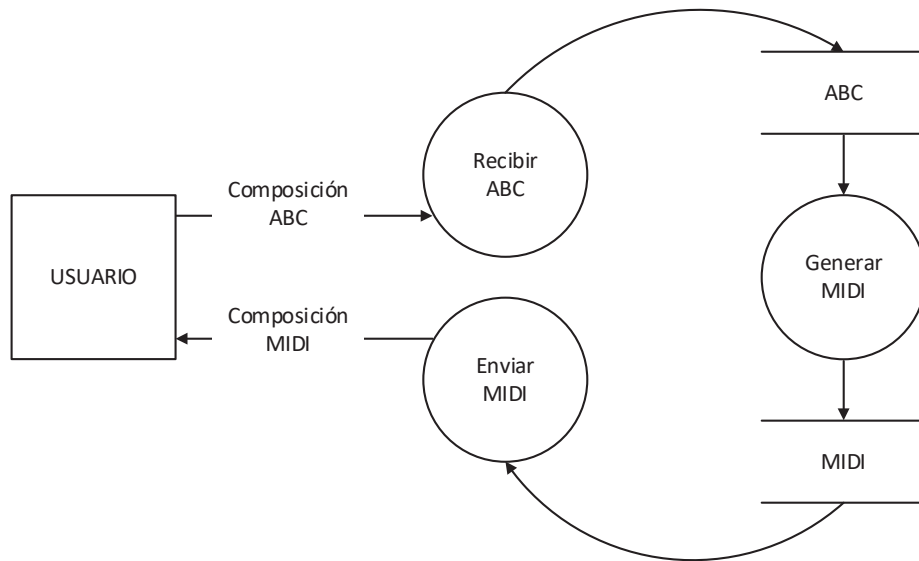


Figura 56 - Acontecimiento 2 detallado
(Fuente: Elaboración propia)

- AC.3: El usuario solicita generar notas con la red neuronal.

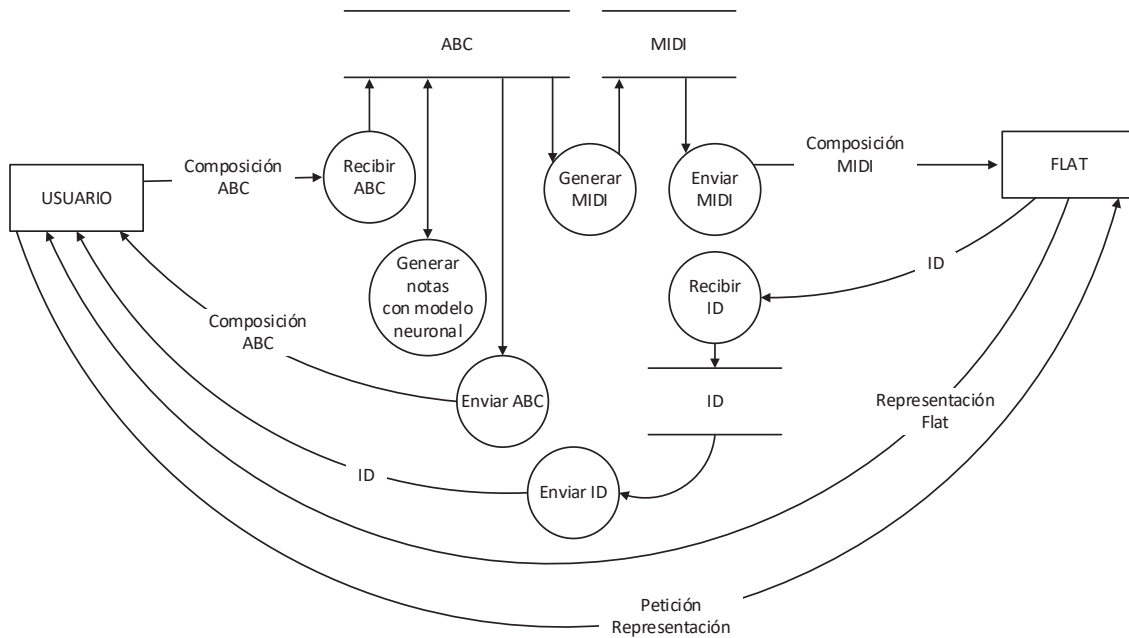


Figura 57 - Acontecimiento 3 detallado
(Fuente: Elaboración propia)

2.6.4.2.4. *Matriz de trazabilidad*

La Tabla 4 - Matriz de trazabilidad (iteración 1) muestra la traza de los requisitos funcionales, y en cuál de los acontecimientos listados anteriormente se ve implementado cada uno de los requisitos. Esto nos permite comprobar que la funcionalidad diseñada para la aplicación cumple todos los requisitos funcionales.

	AC.1	AC.2	AC.3
RF1	x		
RF2	x		x
RF3	x		x
RF4		x	
RF5			x
RF6			x

Tabla 4 - Matriz de trazabilidad (iteración 1)

2.6.4.2.5. *Diagrama de flujo de datos*

La Figura 58 - Diagrama de flujo de datos (iteración 1) muestra el conjunto de todos los flujos de datos posibles, resumiendo en él todos los acontecimientos en un solo gráfico, que se corresponde con una representación del diagrama de contexto a bajo nivel.

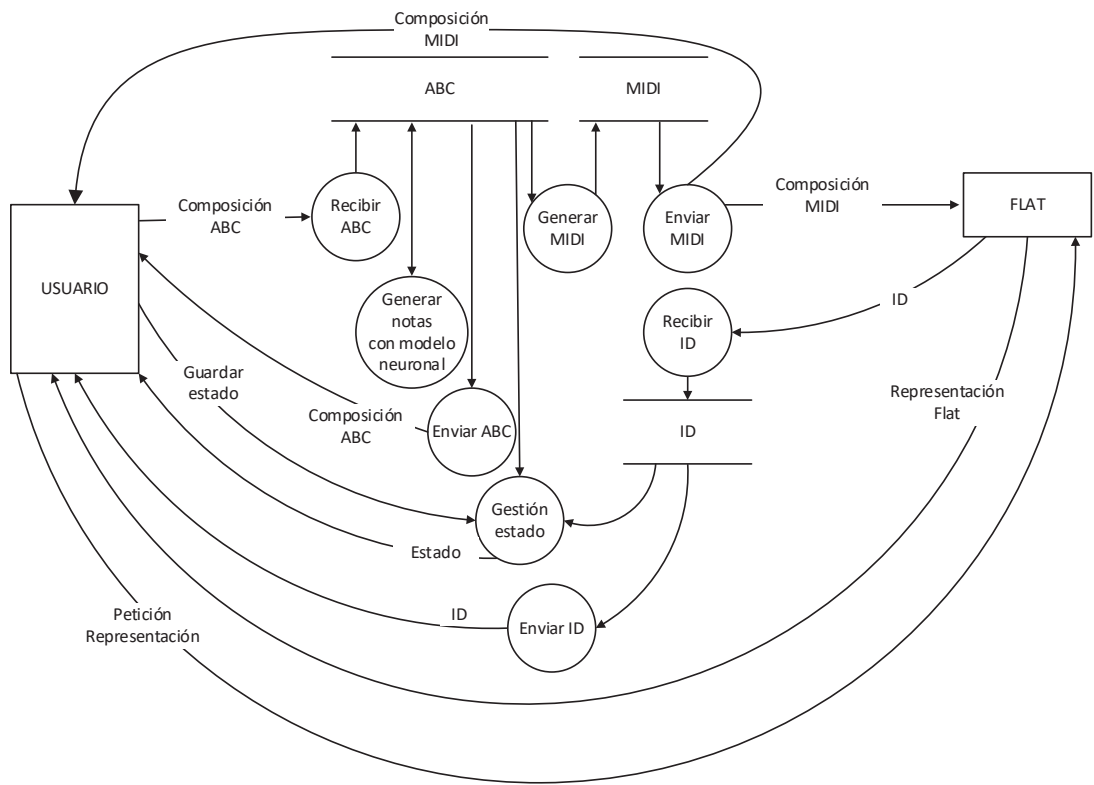


Figura 58 - Diagrama de flujo de datos (iteración 1)
(Fuente: Elaboración propia)

2.6.4.2.6. Diseño de interfaz

La interfaz propuesta para satisfacer los requisitos de interfaz, así como desencadenar los acontecimientos anteriormente mencionados, es la siguiente:



Figura 59 - Interfaz en PC (iteración 1)
(Fuente: Elaborado con Proto.io)

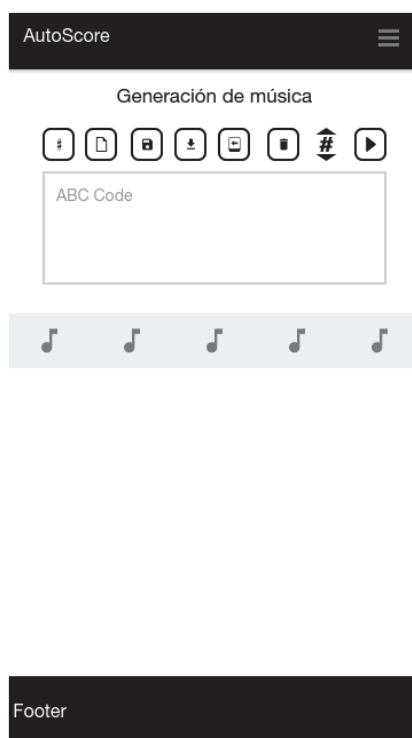


Figura 60 - Interfaz en Smartphone (iteración 1)
(Fuente: Elaborado con Proto.io)

2.6.4.3. Codificación

Como se mencionó en las Restricciones, el sistema se implementará en Python 3⁴². Para la codificación se hará uso de Flask⁴³ 0.12.2, un *microframework* para Python basado en Werkzeug⁴⁵ y Jinja2⁴⁴ con licencia BSD.

Werkzeug⁴⁵ es una librería WSGI (*Web Server Gateway Interface*) para Python, que es una especificación de interfaz simple y universal entre servidores y aplicaciones web.

Flask combina la interfaz WSGI de Werkzeug con Jinja2, un lenguaje de plantillas para Python que le permite crear interfaces web de forma sencilla y reutilizar código mediante la herencia de plantillas. Jinja2 también provee mecanismos de ejecución aislada (*sandboxed*) y sistemas de evasión HTML automáticos para prevenir fallos de seguridad XSS (*Cross-Site Scripting*), entre otras características.

Para el control de la interfaz y codificación necesaria en el lado del cliente se empleará JavaScript, con la librería jQuery. Y para el diseño de la interfaz Bootstrap⁴⁶ 4.0.0.

En Jinja2⁴⁴, y por extensión en Flask, los archivos se organizan en una estructura determinada. En el directorio raíz de trabajo se sitúa el controlador, que establece las rutas que tendrá nuestra web. En el directorio *templates* se encuentran las plantillas, y en *static* se ubican las hojas de estilo, archivos JavaScript, imágenes, etc.

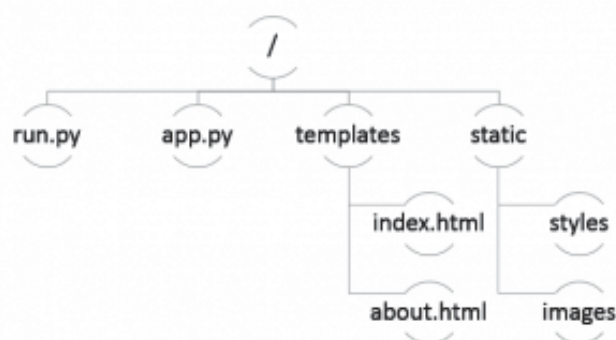


Figura 61 - Estructura de directorios de Jinja2
(Fuente: WarRoom (SecureState)⁴⁷)

Para el desarrollo de la aplicación se seguirá el patrón de arquitectura software Modelo-Vista-Controlador (MVC). Esta arquitectura separa los datos y la lógica de negocio (modelo), del módulo que se encarga de gestionar eventos y comunicaciones (controlador), y de la interfaz de usuario (vista).

⁴² <https://docs.python.org/3.5/>

⁴³ <http://flask.pocoo.org/docs/0.12/>

⁴⁴ <http://jinja.pocoo.org/docs/2.10/>

⁴⁵ <http://werkzeug.pocoo.org/docs/0.14/>

⁴⁶ <https://getbootstrap.com/docs/4.0/>

⁴⁷ Figura recuperada de: <https://goo.gl/YoztyY>

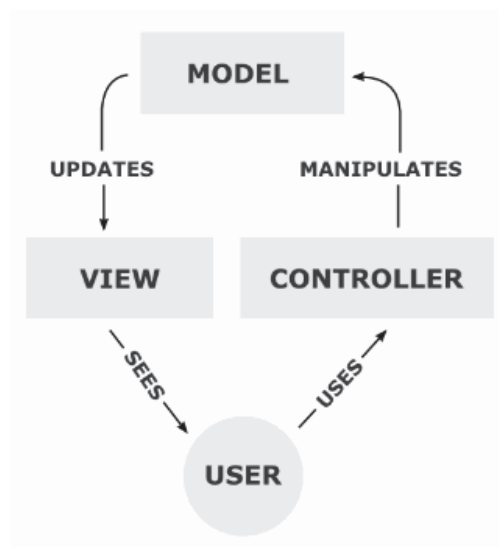


Figura 62 - Componentes MVC
(Fuente: Dominio público⁴⁸)

El usuario envía órdenes a la aplicación que son recibidas y tratadas por el controlador. Habitualmente, el controlador realizará las operaciones oportunas en el modelo. Los cambios y operaciones correspondientes se mostrarán en la vista, que corresponde a la interfaz que presenta el modelo al usuario. En ocasiones esto puede variar, por ejemplo, si se solicita un cambio en la presentación del modelo, donde el controlador enviará órdenes a la vista. Dependiendo de la funcionalidad, se puede ver el controlador como un intermediario entre el modelo y la vista. En Flask, cada ruta se asocia a un método, es decir, con una acción del controlador.

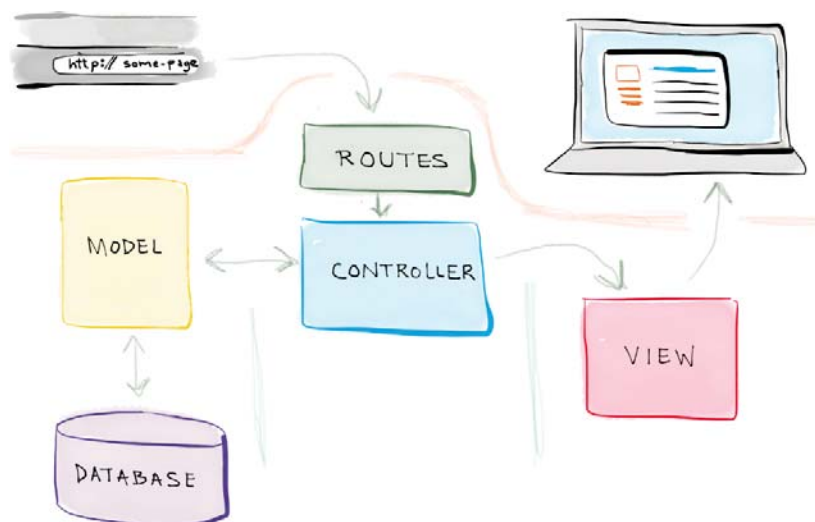


Figura 63 - Arquitectura MVC con rutas
(Fuente: self-taught coders by Alex Coleman⁴⁹)

⁴⁸ Figura recuperada de: <https://commons.wikimedia.org/wiki/File:MVC-Process.png>

⁴⁹ Figura recuperada de: <https://selftaughtcoders.com/model-view-controller-mvc-web-application>

En el caso que nos ocupa, el modelo estará compuesto principalmente por nuestra red neuronal. Para facilitar el manejo, la programación y hacer el sistema lo más modular posible, la red neuronal se establece en la clase *Generator* (en el archivo *generator.py*⁵⁰) que actúa de envoltura con métodos que permiten generar los pasos indicados.

Flask permite la creación de interfaces web mediante Jinja2, que implementa mecanismos de inclusión y extensión (herencia), por lo que podemos reutilizar código y crear una vista modular. Como base, se elabora una plantilla *layout.html*⁵¹ que contendrá la estructura principal de la web.

En esta estructura principal, se han incluido los archivos *section_nav.html*⁵² y *section_footer.html*⁵³ que corresponden a la barra de navegación y el pie de página respectivamente.

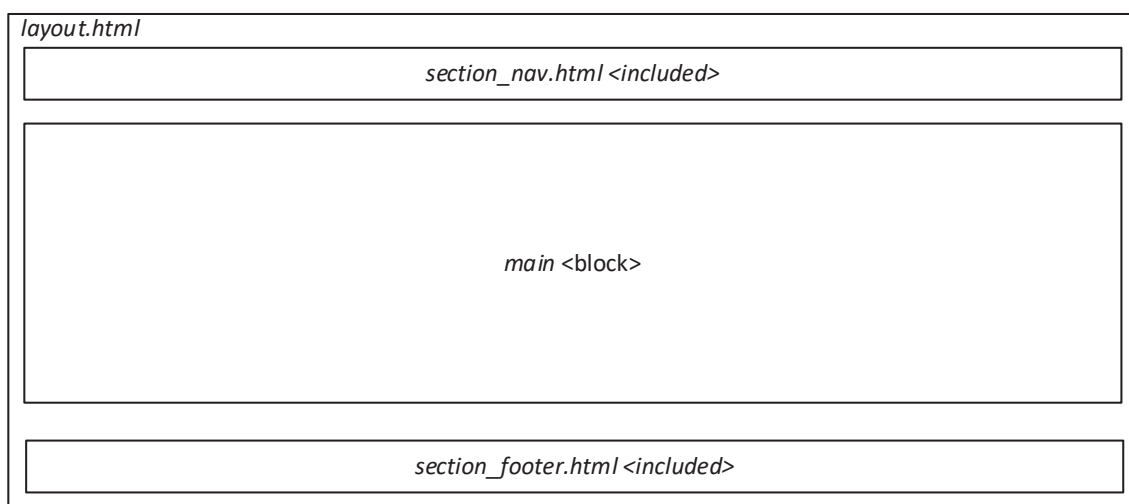


Figura 64 - Plantilla base
(Fuente: Elaboración propia)

En esta plantilla base, se definen los bloques vacíos con nombre “css”, “main” y “js”. De esta forma, las plantillas que extienden de *layout.html* solo tienen que implementar el bloque “main” que corresponda, incluyendo los bloques “css” para insertar hojas de estilos y “js” para código JavaScript si fuera necesario.

En cuanto a secciones de contenido, se ha creado una sección *index.html*⁵⁴, que hace de portada de web; y una sección *generar.html*⁵⁵ que presentará la aplicación. Ambas

⁵⁰ https://github.com/Tuxt/AutoScore/blob/master/03_web/autoscore/generator.py

⁵¹ https://github.com/Tuxt/AutoScore/blob/master/03_web/autoscore/templates/layout.html

⁵² https://github.com/Tuxt/AutoScore/blob/master/03_web/autoscore/templates/section_nav.html

⁵³ https://github.com/Tuxt/AutoScore/blob/master/03_web/autoscore/templates/section_footer.html

⁵⁴ https://github.com/Tuxt/AutoScore/blob/master/03_web/autoscore/templates/index.html

⁵⁵ https://github.com/Tuxt/AutoScore/blob/master/03_web/autoscore/templates/generar.html

extienden a la plantilla base *layout.html* como se indica en la Figura 65 - Diagrama UML de las plantillas de la vista.

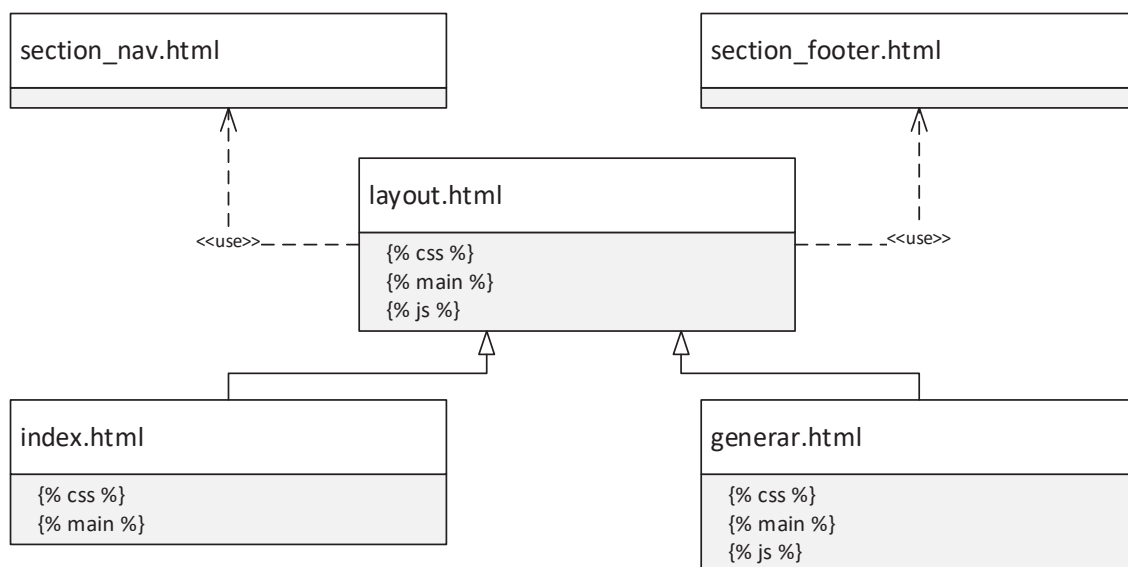


Figura 65 - Diagrama UML de las plantillas de la vista (iteración 1)
(Fuente: Elaboración propia)

Flask provee de diferentes mecanismos de almacenamiento de información como son los objetos “g” y “session”. Ambos actúan como diccionarios Python. La principal diferencia es que “g” permite compartir datos entre diferentes partes del código, para una petición concreta, por lo que al finalizar la petición en curso los datos de “g” son borrados. Por otro lado, “session” almacena información por navegador, no por petición. Por lo que la información se comparte entre peticiones para un mismo navegador. Esto se realiza mediante el uso de *cookies* firmadas criptográficamente, por lo que el usuario puede ver el contenido de las cookies pero no modificarlo (sin la contraseña secreta utilizada para firmar la *cookie*).

Además, la instancia de la clase Flask (la clase principal de la aplicación) permite el almacenamiento de información en el atributo “config”, que también actúa como diccionario. Esta información es almacenada por aplicación.

Por ello, para almacenar la información necesaria para guardar el estado de la sesión se utiliza el objeto “session”, puesto que almacena la información por usuario (navegador), siendo persistente al volver a la web tras haber cerrado la ventana. Puesto que no se trata de información sensible, como pueden ser credenciales, no es necesario implementar ningún mecanismo adicional de seguridad.

Para el almacenamiento de archivos ABC y MIDI, se creará un directorio dedicado. Este directorio se almacena en el diccionario “config”, puesto que es común a toda la aplicación, y se almacena en el lado del servidor.

El objeto “g”, en este caso, también está siendo utilizado. En él se guardan los *callbacks* para la petición en curso, como son las órdenes de borrado de archivos, que se encarga de borrar archivos temporales al finalizar la petición.

Otra información sensible, como son las credenciales de acceso a la plataforma Flat se almacenan en el lado del servidor. De su gestión se encarga el módulo “*flat_api*”, que proporciona la plataforma Flat como cliente Python para su API.

Finalmente, el sistema cuenta con un directorio “*bin*” en el que se almacenan los binarios externos a la aplicación, y un directorio “*model*” en el que se almacena el modelo neuronal que es cargado por la clase *Generator* ya mencionada. En este caso, el directorio “*bin*” cuenta con el archivo “*abc2midi*” que se encarga de transformar un código ABC a MIDI. El directorio “*model*” contiene la arquitectura neuronal y los pesos neuronales de la red, así como dos objetos *pickle* que contienen los diccionarios para traducir el código ABC a números para la red neuronal, y viceversa.

2.6.4.4. Pruebas

Para el caso dado, no se llevarán pruebas unitarias, dado que no existen módulos individuales que testear. Puesto que la lógica de negocio se concentra en la red neuronal, cuya codificación no se ha realizado durante este proyecto.

A continuación se listan las pruebas de integración realizadas:

Prueba	Escenario	Resultado
P1	Inserción de notas: se comprueba que todas las notas se insertan correctamente en la caja de texto, así como la activación y desactivación de alteración de notas, habilitando y deshabilitando las notas correspondientes.	OK
P2	Gestión de estado: se comprueba la funcionalidad de guardado de estado, así como la carga de la composición guardada al volver a la página.	OK
P3	Generación MIDI: se introduce una composición y se comprueba la generación en archivo MIDI.	OK
P4	Generación de notas y visualización: se introduce una composición y se comprueba que la funcionalidad de generación de notas genera nuevas notas, que la plataforma Flat recibe la composición y que esta se representa correctamente en el cliente web.	OK

Tabla 5 - Pruebas de integración (iteración 1)

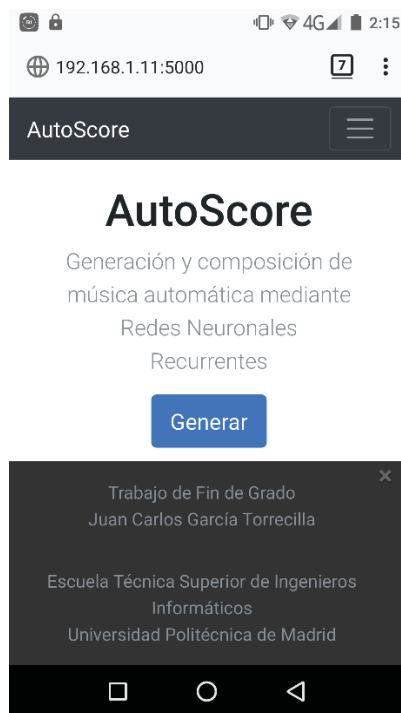


Figura 68 - Captura de index.html en SmartPhone (iteración 1)
(Fuente: Elaboración propia)

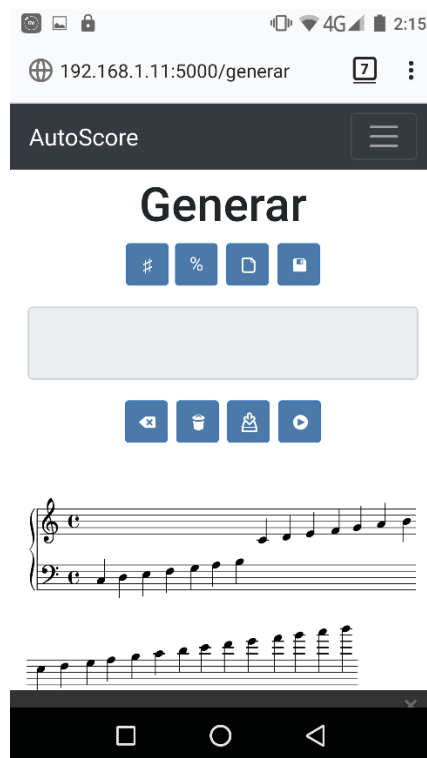


Figura 69 - Captura de generar.html en SmartPhone (iteración 1)
(Fuente: Elaboración propia)

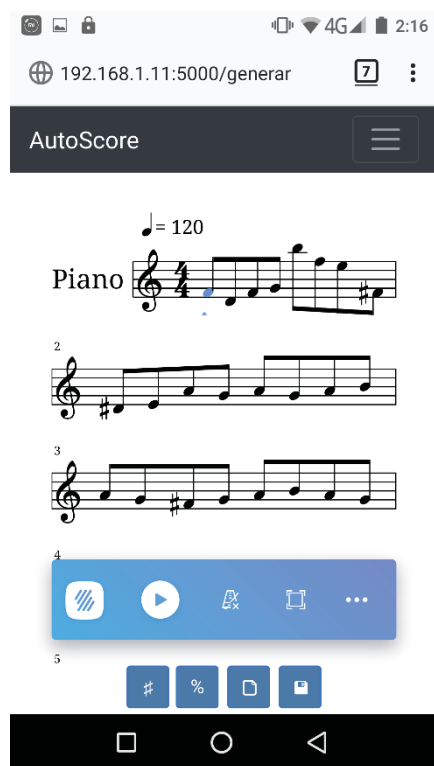


Figura 70 - Visualización de composición en SmartPhone (iteración 1)
(Fuente: Elaboración propia)

2.7. Segunda iteración

2.7.1. Preproceso

En esta segunda iteración, se ampliará en primer lugar las capacidades del sistema, aumentando el diccionario. Ahora no solo contemplará notas como en el caso preliminar que utilizamos como línea base, si no que incluirá multiplicadores de nota para variar el tiempo de cada nota y que las composiciones musicales no sean monótonas. Igualmente, se incluirán los símbolos de apertura y cierre de corchetes, que encierran notas pertenecientes a un acorde.

Además, se incluirán Adornos que, al tener dos representaciones, serán simplificados a su símbolo correspondiente.

Desafortunadamente, el estándar ABC no provee una implementación oficial para la representación de partituras y generación de archivos MIDI. El software utilizado en la actualidad es producido por terceros, por lo que en ocasiones no sigue la última versión del estándar, tiene características sin implementar o el proyecto está obsoleto o discontinuado. Una de las implementaciones más extendidas y utilizadas es abcMIDI, desarrollado por James Allwright (2002) y, posteriormente, mantenido por Seymour Shlien (2015).

Para este proyecto, se ha utilizado una versión actualizada⁵⁶ de dicha implementación, puesto que es la más completa que he encontrado.

Teniendo en cuenta lo anterior, respecto al tratamiento de los Adornos, algunas instrucciones son ignoradas por el programa conversor, o no tienen efecto en el sonido producido, ya que son indicaciones muy concretas sobre cómo se debe tocar. El programa conversor ignora concretamente las instrucciones para “*roll*”, “*accent*”, “*lowermordent*”, “*uppermordent*”, “*coda*”, “*segno*”, “*upbow*” y “*downbow*”. Igualmente, ignora las abreviaturas “P” (*uppermordent*), “O” (*coda*) y “S” (*segno*). Tanto *coda* como *segno* son indicaciones de repetición que debe realizar el intérprete cuando aparecen indicaciones textuales como “*Dal Coda al Segno*”, por lo que tampoco serían interpretadas por el programa. Lo mismo ocurre con *lowermordent* y *uppermordent*, que son indicaciones para cuerda frotada, por lo que sus abreviaturas “u” y “v” (no ignoradas), no producen ningún efecto diferente en el sonido producido. Tampoco afecta al sonido MIDI la abreviatura de *accent* “L”, que indica énfasis al inicio de la nota. La abreviatura “M” (*lowermordent*) no produce aviso en el programa conversor de que esté siendo ignorada, por lo que debería tener efecto en el audio producido, pero no produce cambios. Esto hace pensar que probablemente no esté implementada o sea ignorada también. El resto de signos e instrucciones mencionados en Adornos, sí producen alteraciones en el audio generado. A continuación, se presenta una tabla resumen:

⁵⁶ <https://github.com/leesavide/abcmidi>

<i>Adorno</i>	<i>Instrucción</i>		<i>Signo</i>	
<i>Roll</i>	!roll!	IGNORADO	~	FUNCIONAL
<i>Fermata</i>	!fermata!	FUNCIONAL	H	FUNCIONAL
<i>Accent</i>	!accent!	IGNORADO	L	SIN EFECTO
<i>Lower mordent</i>	!lowermordent!	IGNORADO	M	NO IMPL./IGNOR.
<i>Upper mordent</i>	!uppermordent!	IGNORADO	P	IGNORADO
<i>Coda</i>	!coda!	IGNORADO	O	IGNORADO
<i>Segno</i>	!segno!	IGNORADO	S	IGNORADO
<i>Trill</i>	!trill!	FUNCIONAL	T	FUNCIONAL
<i>Up-bow</i>	!upbow!	IGNORADO	u	SIN EFECTO
<i>Down-bow</i>	!downbow!	IGNORADO	v	SIN EFECTO
<i>Staccato</i>			.	FUNCIONAL

Tabla 6 - Influencia de adornos en archivos MIDI

Por ello, el siguiente paso en el preproceso será eliminar aquellos símbolos y directivas que no nos aportan contenido musical (ignorados o sin efecto). Para el caso de *roll*, cuya instrucción es ignorada pero su símbolo no, sustuiremos la instrucción por el símbolo. Lo mismo ocurrirá con las instrucciones funcionales (*trill* y *fermata*), para no tener varias representaciones de un mismo efecto. Por lo que, en total contaremos con los signos que corresponden a los adornos *roll*, *fermata*, *trill* y *staccato*. Estos símbolos se separarán para que no figuren contiguos a ninguna nota, igual que el símbolo guion “-” (ligadura de prolongación). Este proceso se realizará en el *script* *3-fix_decorators.py*⁵⁷, a continuación de los *scripts* *1-len_changer.py*⁵⁸ y *2-key_changer.py*⁵⁹ ya mencionados en el Preproceso de la Primera iteración, que permanecen igual.

Durante ese proceso, el archivo *3-fix_decorators.py*⁵⁷ también separa los elementos del código por espacios, separando los tonos de nota de su multiplicador, así como los paréntesis, cadenas de *string* que indican acompañamientos, etc. Opcionalmente, se le puede indicar que mantenga o elimine los *strings* de acompañamiento, así como los cambios de métrica en medio de la canción, ya que la métrica afecta al acompañamiento cuando se genera la música a partir del código. Se ha implementado esta opción pensando en una futura iteración en la que se trate esta característica, que de momento no se incluirá en el modelo, debido a la problemática que presenta su dependencia con la métrica y los compases.

El archivo *4-data_extractor.py*⁶⁰ se ha sustituido por *4-data_extractor2.py*⁶¹, cuyo funcionamiento es el mismo (extraer la información musical de todos los archivos y recogerla en un único archivo sin cabeceras y con una canción por línea). Se ha creado una nueva versión para adaptar el algoritmo al nuevo código musical fruto de los cambios

⁵⁷ https://github.com/Tuxt/AutoScore/blob/master/01_preproceso/3-fix_decorators.py

⁵⁸ https://github.com/Tuxt/AutoScore/blob/master/01_preproceso/1-len_changer.py

⁵⁹ https://github.com/Tuxt/AutoScore/blob/master/01_preproceso/2-key_changer.py

⁶⁰ https://github.com/Tuxt/AutoScore/blob/master/01_preproceso/4-data_extractor.py

⁶¹ https://github.com/Tuxt/AutoScore/blob/master/01_preproceso/4-data_extractor2.py

anteriores. También se ha simplificado el algoritmo, haciendo el código más simple y legible.

Lo mismo ha ocurrido con el archivo *5-note_simplifier.py*⁶², generando una nueva versión *5-note_simplifier2.py*⁶³. El archivo *6-recode.py*⁶⁴ ya no es necesario, puesto que el proceso que realizaba ya lo realiza el nuevo *script 3-fix_decorators.py*⁶⁵, lo que ha permitido un proceso más simple del texto simplificando los códigos posteriores (numerados 4 y 5).

Finalizado el preproceso, se ha decidido eliminar los símbolos “{” y “}” que indican *grace notes*, puesto que su funcionamiento es similar al de los acordes (símbolo de apertura y cierre) y aparecen únicamente 1 vez, lo que hace excesivamente complicado que la red pueda aprender la sintaxis. Es decir, la red no aprendería que por cada apertura debe haber un cierre. Eliminando estos dos símbolos, obtenemos un total de 50 palabras en el diccionario.

2.7.2. Modelado

En la Primera iteración, se estableció en 10 el número de *timesteps*. Una decisión principalmente guiada por las características de la implementación de la red, que espera que cada muestra de entrada tenga una dimensión específica y fija, por lo que si se indican 100 *timesteps*, el usuario debe introducir 100 notas para generar nuevas notas.

Para solucionar esta limitación, y permitir también la generación de música sin introducir una sola nota, se ha modificado el archivo *datamanager.py*⁶⁶ que proporcionaba funciones de carga de datos ya troceados en muestras. De forma que el nuevo archivo *datamanager2.py*⁶⁷ implementa un mecanismo de *padding*. Esto quiere decir que las muestras que no alcancen el número de *timesteps* necesarios se completarán con el valor indicado. De forma que si, anteriormente, la secuencia de valores $[x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9]$ se procesaba en muestras de 5 *timesteps* producía las siguientes muestras de entrenamiento:

$$x = \begin{bmatrix} [x_1, x_2, x_3, x_4, x_5] \\ [x_2, x_3, x_4, x_5, x_6] \\ [x_3, x_4, x_5, x_6, x_7] \\ [x_4, x_5, x_6, x_7, x_8] \end{bmatrix} \quad y = \begin{bmatrix} x_6 \\ x_7 \\ x_8 \\ x_9 \end{bmatrix}$$

Ahora, con el mecanismo de *padding* se generarán:

⁶² https://github.com/Tuxt/AutoScore/blob/master/01_preproceso/5-note_simplifier.py

⁶³ https://github.com/Tuxt/AutoScore/blob/master/01_preproceso/5-note_simplifier2.py

⁶⁴ https://github.com/Tuxt/AutoScore/blob/master/01_preproceso/6-recode.py

⁶⁵ https://github.com/Tuxt/AutoScore/blob/master/01_preproceso/3-fix_decorators.py

⁶⁶ https://github.com/Tuxt/AutoScore/blob/master/02_modelos/datamanager.py

⁶⁷ https://github.com/Tuxt/AutoScore/blob/master/02_modelos/datamanager2.py

$$x = \begin{bmatrix} [-1, -1, -1, -1, -1] \\ [-1, -1, -1, -1, x_1] \\ [-1, -1, -1, x_1, x_2] \\ [-1, -1, x_1, x_2, x_3] \\ [-1, x_1, x_2, x_3, x_4] \\ [x_1, x_2, x_3, x_4, x_5] \\ [x_2, x_3, x_4, x_5, x_6] \\ [x_3, x_4, x_5, x_6, x_7] \\ [x_4, x_5, x_6, x_7, x_8] \end{bmatrix} \quad y = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \end{bmatrix}$$

Esto, elimina la necesidad de aportar tantos datos como *timesteps* se utilicen, pudiendo generar música sin insertar una sola nota, puesto que los datos previos faltantes se rellenan con el valor seleccionado para hacer *padding*. Lo que además nos permite incrementar el número de *timesteps* sin problemas. Además, nos proporciona más muestras de entrenamiento.

Por otro lado, estamos ingresando igualmente otros datos a la red neuronal. Para tratar esto, introduciremos una capa de *Masking* después de la entrada para enmascarar los valores “-1”. Esta capa actúa como una máscara, que decide qué datos “pasan” (qué datos se tendrán en cuenta) y cuáles no.

En la iteración anterior también se introdujo una capa de *dropout* para prevenir el *overfitting* basándose en [50]. Una decisión que puede haber resultado contraproducente teniendo en cuenta el reducido número de muestras en nuestro caso.

Para comprobarlo, se han ejecutado 8 pruebas sobre la red empleando diferentes combinaciones de *dropout*, empleando el script *test002_SimpleRNN.py*⁶⁸. Además del *dropout* posterior a la capa recurrente, en ocasiones también se realiza un *dropout* después de la capa de entrada. De igual manera, es posible aplicar *dropout* a las capas recurrentes en el eje temporal, como se ve en la Figura 66 - Posiciones de dropout en la red, marcado en verde.

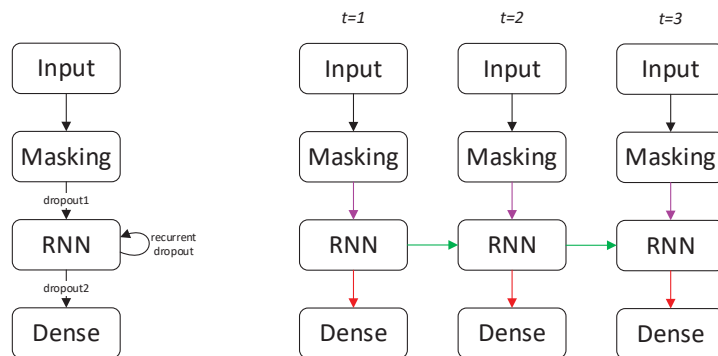


Figura 71 - Posiciones de dropout en la red
(Fuente: Elaboración propia)

⁶⁸ <https://goo.gl/qZEVbU>

Estos tres tipos de *dropout* se han evaluado y comparado en las 8 pruebas mencionadas:

- Sin *dropout*.
- *Dropout* antes de la capa recurrente (magenta).
- *Dropout* después de la capa recurrente (rojo).
- *Dropout* recurrente en el eje temporal (verde).
- *Dropout* antes y después de la capa recurrente.
- *Dropout* antes de la capa recurrente y en el eje temporal.
- *Dropout* después de la capa recurrente y en el eje temporal.
- Todas las posiciones de *dropout*.

Posteriormente, se evaluaron y compararon diferentes capas recurrentes (archivo *test003_1-layer.py*⁶⁹). Para ello, se entrenó la misma estructura de red (1 capa recurrente), con diferentes tipos de capa recurrente: SimpleRNN, LSTM y GRU. Estas pruebas se realizaron empleando 50 y 100 *timesteps* para cada red, con 64 neuronas y bloques de 64 muestras. Además, también se testeó la red SimpleRNN con el *dataset* de la iteración anterior y 50 *timesteps*, para evaluar la influencia de los cambios en el *dataset*.

2.7.3. Evaluación

Elaborado el test para comparar *dropouts* en un entrenamiento de 20 *epochs* obtenemos los siguientes resultados:

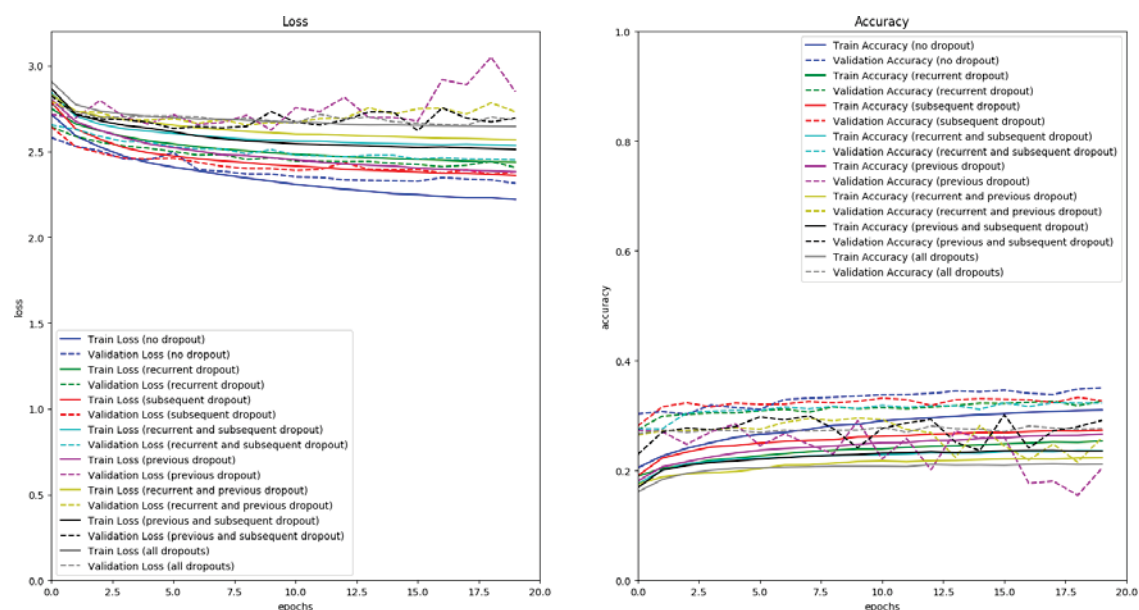


Figura 72 - Comparativa de dropout
(Fuente: Elaboración propia)

⁶⁹ https://github.com/Tuxt/AutoScore/blob/master/02_modelos/test003_1-layer/test003_1-layer.py

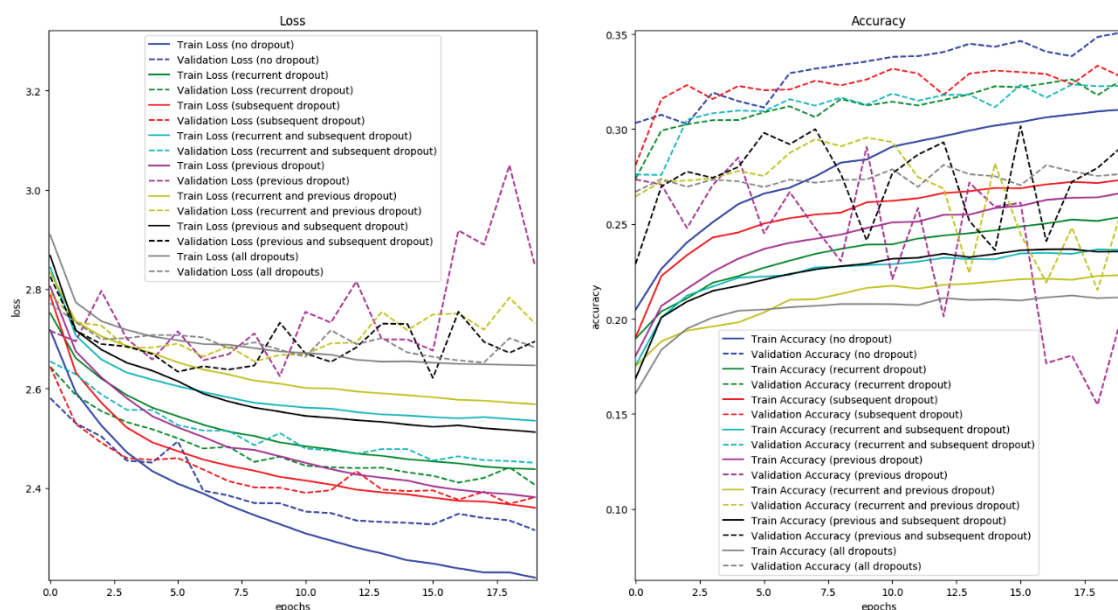


Figura 73 - Comparativa de dropout (ampliado)
(Fuente: Elaboración propia)

Tanto en *loss* como en *accuracy* obtenemos los mejores resultados sin *dropout* (trazo azul), tanto en datos de entrenamiento como de validación. Vemos también que los resultados son peores cuantos más *dropouts* se activan, siendo los peores resultados los correspondientes a la prueba con todos los *dropouts* en funcionamiento (trazo gris), por lo que en adelante no se empleará ningún tipo de *dropout*.

A partir de la versión 2 de Keras, las métricas de *precision*, *recall* y *f-measure* fueron retiradas, puesto que se ejecutaban por *batch* y no por *epoch*. Por lo que, tratándose de métricas globales, aplicarlas a cada *batch* resultaba más engañoso que útil.

En esta segunda iteración, para incluir estas métricas, se han implementado mediante la clase *Metrics* (en el archivo *metrics.py*⁷⁰), que extiende de la clase *Callback*. De esta manera, se añade como un *callback* al entrenamiento, pudiendo ejecutarse al finalizar cada *epoch*. Esta clase, calcula las 3 métricas haciendo uso de la librería *sklearn*. Puesto que se trata de métricas que se calculan para cada una de las clases, el valor final de cada métrica se ha calculado haciendo la media de todas las clases.

⁷⁰ https://github.com/Tuxt/AutoScore/blob/master/02_modelos/test003_1-layer/metrics.py

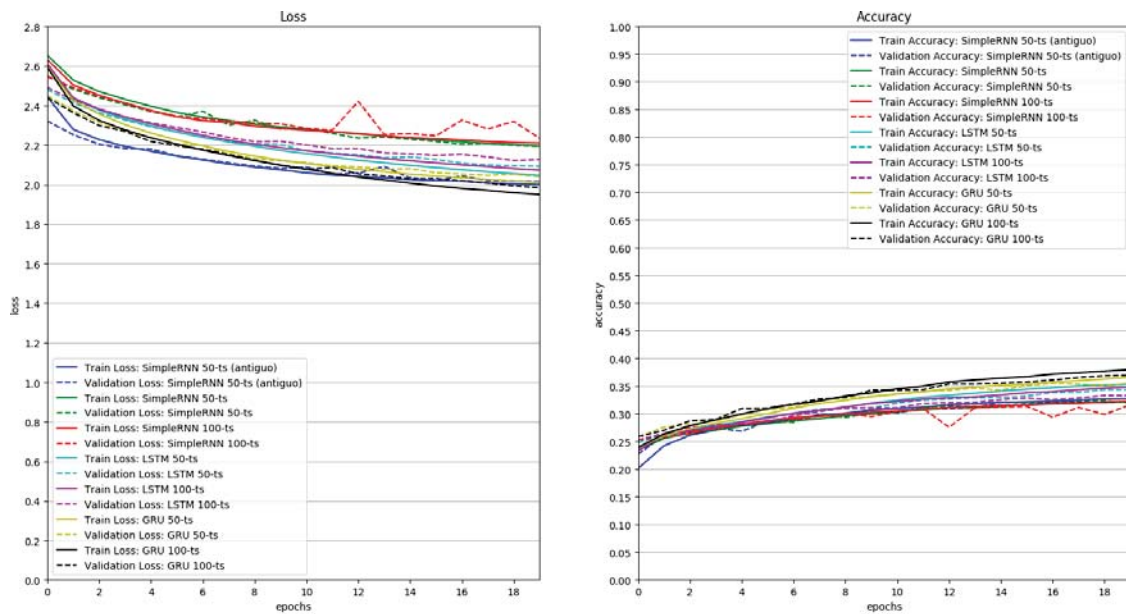


Figura 74 - Comparativa SimpleRNN-LSTM-GRU (loss y accuracy)
(Fuente: Elaboración propia)

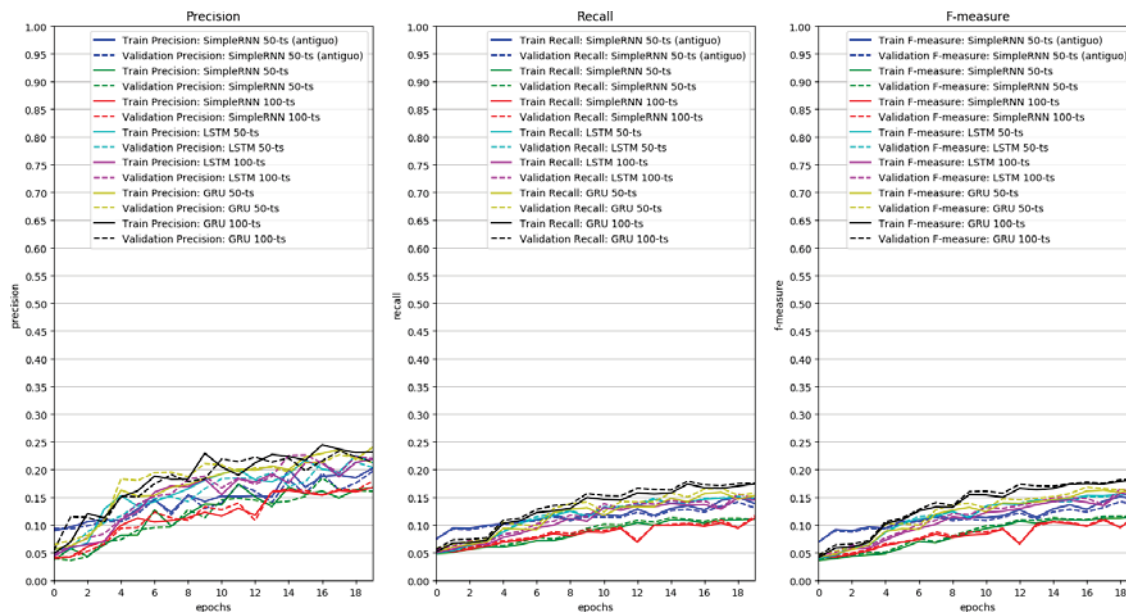


Figura 75 - Comparativa SimpleRNN-LSTM-GRU (precision, recall y f-measure)
(Fuente: Elaboración propia)

En las figuras Figura 69 - Comparativa SimpleRNN-LSTM-GRU (loss y accuracy) y Figura 70 - Comparativa SimpleRNN-LSTM-GRU (precision, recall y f-measure) se presentan los resultados obtenidos entre las estructuras SimpleRNN, LSTM y GRU. La vista general nos da una perspectiva de los valores alcanzados respecto del total.

Para compararlas más detalladamente, atiéndase a las figuras Figura 71 - Comparativa SimpleRNN-LSTM-GRU (loss y accuracy ampliado) y Figura 72 - Comparativa SimpleRNN-LSTM-GRU (precision, recall y f-measure ampliado):

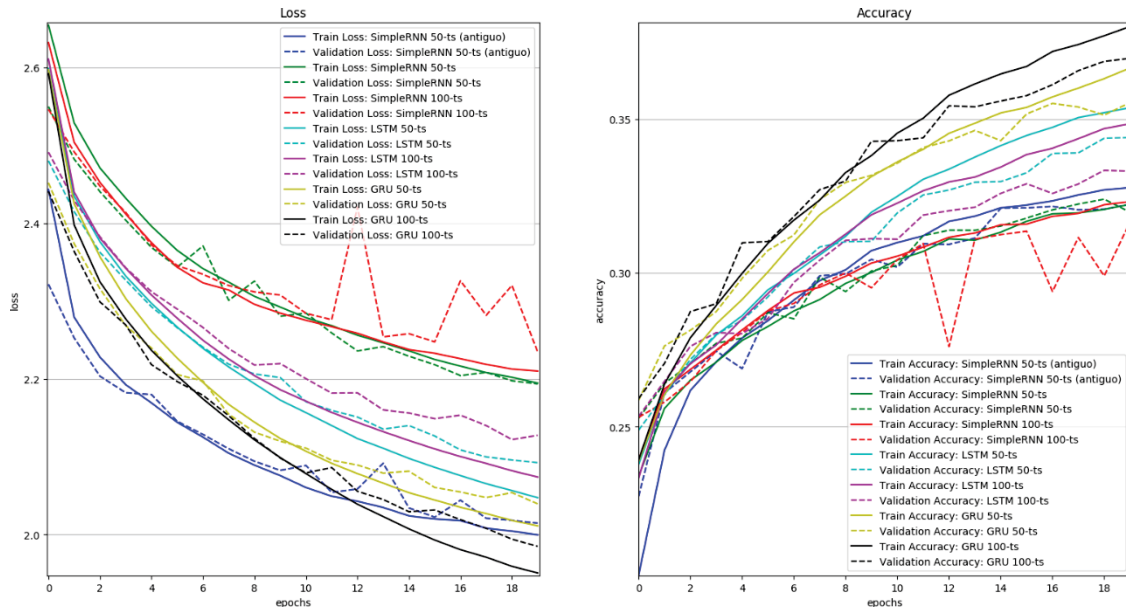


Figura 76 - Comparativa SimpleRNN-LSTM-GRU (loss y accuracy ampliado)
(Fuente: Elaboración propia)

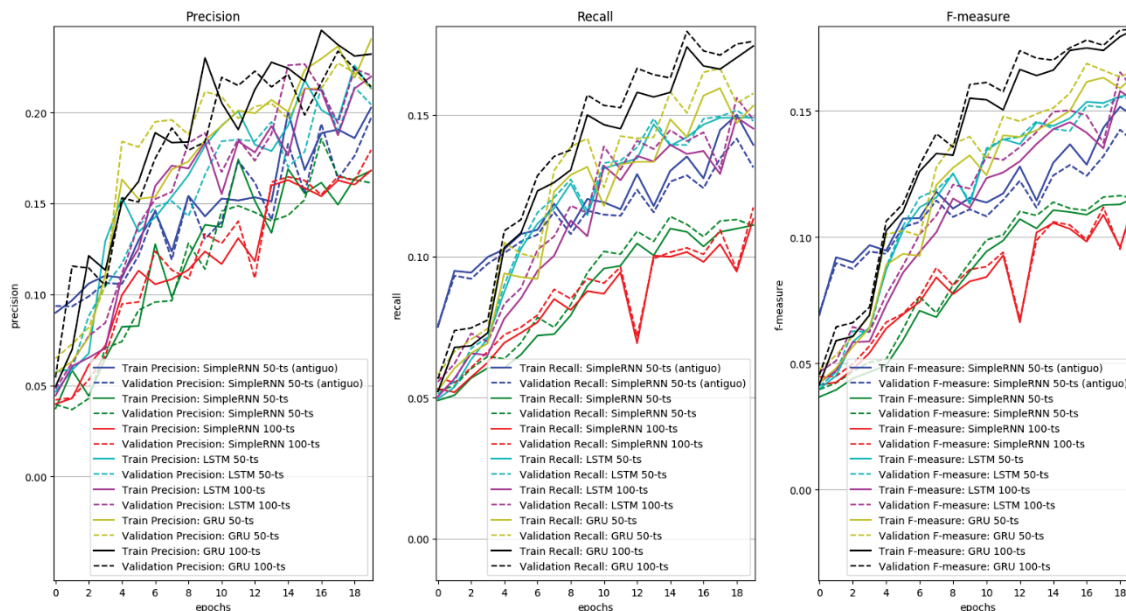


Figura 77 - Comparativa SimpleRNN-LSTM-GRU (precision, recall y f-measure ampliado)
(Fuente: Elaboración propia)

En primer lugar, vemos que el trazo azul tiene un *loss* mucho menor, y mejora en varias gráficas el trazo verde (misma red con el nuevo *dataset*). Debido a que emplea el *dataset* de la iteración anterior, el cual era mucho más simple.

En general, vemos que los resultados de las redes SimpleRNN son peores que el resto de redes. Las estructuras de red LSTM y GRU obtienen una precisión (*precision*) similar, siendo superior en el caso de las redes GRU. En el resto de métricas, las redes GRU obtienen resultados superiores a LSTM con una diferencia más notable.

En redes LSTM, obtenemos mejores resultados con 50 *timesteps* que con 100 *timesteps*, mientras que en GRU ocurre lo contrario. Por lo que, en este caso, LSTM parece no sacar beneficio del incremento de *timesteps*.

En la Figura 73 - Incremento de diferencia en los resultados LSTM vs GRU, se ve a lo largo de los *epochs* como la red GRU 100 *timesteps* aumentan cada vez más la diferencia (marcada en líneas naranjas) respecto a la red LSTM 100 *timesteps*. Si bien todas tienden a la línea horizontal con el paso del tiempo, la red GRU lo hace más lentamente que LSTM.

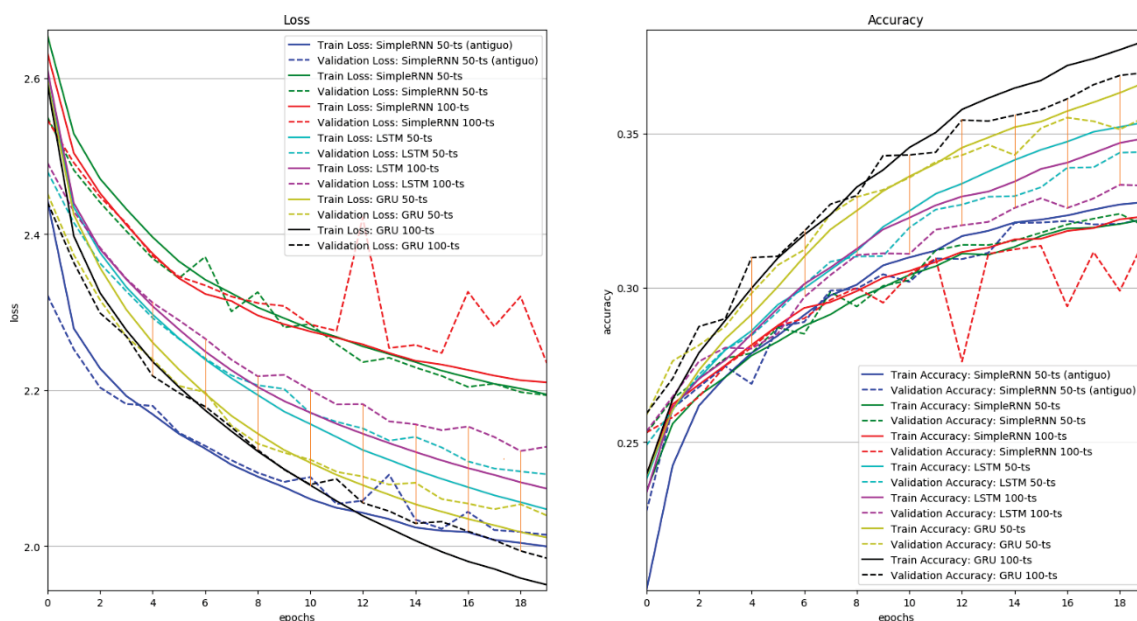


Figura 78 - Incremento de diferencia en los resultados LSTM vs GRU
(Fuente: Elaboración propia)

Como se vio en Redes neuronales artificiales, GRU es una estructura más simple, por lo que tiene menos parámetros (en este caso, 15922) que LSTM (en este caso, 20146). Es posible que esta diferencia pueda influir en los resultados entre LSTM y GRU para la misma red (64 neuronas).

En cualquier caso, los resultados con GRU son más prometedores que los obtenidos con LSTM, y dado también el menor número de parámetros, lo cual hace menos costoso el aprendizaje, se empleará el modelo GRU con 100 *timesteps*.

En general, los resultados siguen siendo muy pobres (37-38% de exactitud). Aun así, véase que se ha mejorado la exactitud respecto a la iteración anterior en torno a un 7-8% en solo 20 *epochs* (con un *dataset* más complejo), siendo en la iteración anterior de un 30% estable (sin mejora a lo largo del tiempo) en los *epochs* 50-70. Por ello, es de esperar que, si se entrena la red neuronal GRU con 100 *timesteps* durante más *epochs* los resultados se vean incrementados ligeramente.

2.7.4. Elaboración de aplicación web

En esta segunda iteración, se mantendrá la aplicación web como se tenía en la iteración anterior, sustituyendo el modelo antiguo por el nuevo modelo, y adaptando las características a las nuevas capacidades del modelo, como son: duración de notas, símbolos, la implementación de *padding* y *masking*, etc.

Además, se incluirá una sección en la que se podrán consultar y visualizar las composiciones realizadas por los usuarios.

2.7.4.1. Análisis de requisitos

2.7.4.1.1. Perspectiva del producto

Sin cambios.

2.7.4.1.2. Restricciones

Sin cambios.

2.7.4.1.3. Requisitos de interfaz

- **RI6:** El sistema debe permitir seleccionar la duración de la nota a introducir.
- **RI7:** El sistema debe permitir insertar los símbolos soportados por el modelo neuronal.
- **RI8:** El sistema debe permitir insertar acordes.

2.7.4.1.4. Requisitos funcionales

- **RF7:** El sistema debe mostrar una lista de composiciones existentes.

2.7.4.1.5. Requisitos no funcionales

Sin cambios.

2.7.4.1.6. Otros requisitos

Sin cambios.

2.7.4.1.7. Seguridad

Sin cambios.

2.7.4.2. Diseño de la aplicación

2.7.4.2.1. Descripción del propósito del sistema

Sin cambios.

2.7.4.2.2. Diagrama de contexto

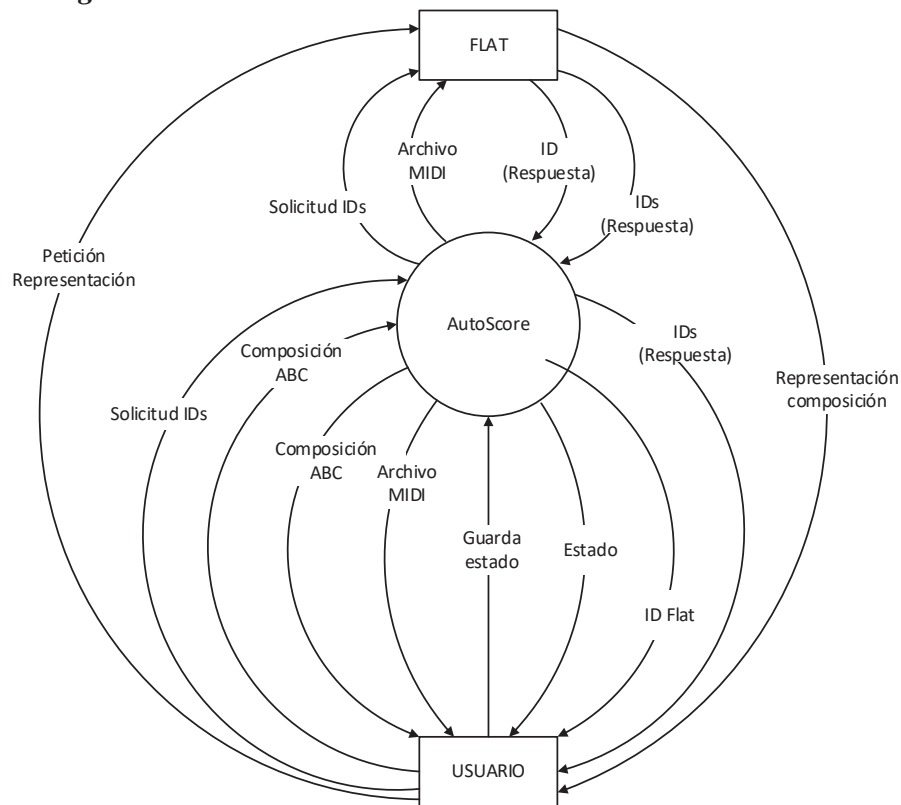


Figura 79 - Diagrama de contexto (iteración 2)
(Fuente: Elaboración propia)

Al diagrama de contexto de la anterior iteración se le han añadido las solicitudes de IDs a la plataforma Flat, correspondiente a la consulta de composiciones.

2.7.4.2.3. Lista de acontecimientos

- AC.4: El usuario solicita una composición de la lista de composiciones existentes.

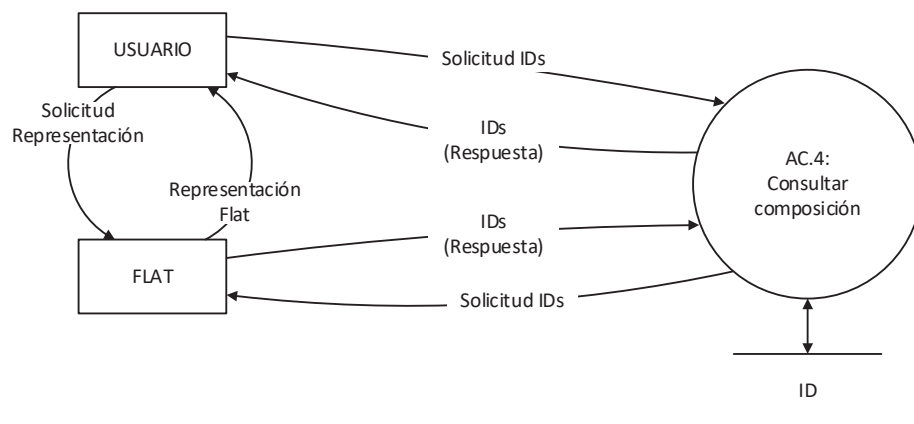


Figura 80 - Acontecimiento 4
(Fuente: Elaboración propia)

A continuación, tenemos el nuevo proceso a nivel de detalle más bajo:

- AC.4: El usuario solicita una composición de la lista de composiciones existentes.

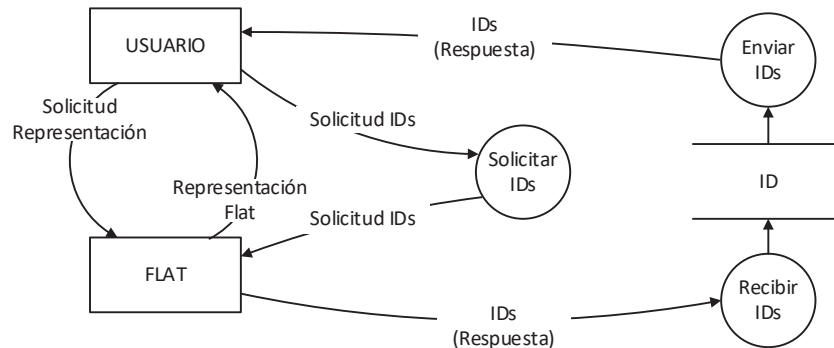


Figura 81 - Acontecimiento 4 detallado
(Fuente: Elaboración propia)

2.7.4.2.4. Matriz de trazabilidad

En la nueva matriz de trazabilidad se puede ver la correspondencia entre el nuevo requisito funcional añadido y el nuevo acontecimiento.

	AC.1	AC.2	AC.3	AC.4
RF1	x			
RF2	x		x	
RF3	x		x	
RF4		x		
RF5			x	
RF6			x	
RF7				x

Tabla 7 - Matriz de trazabilidad (iteración 2)

2.7.4.2.5. Diagrama de flujo de datos

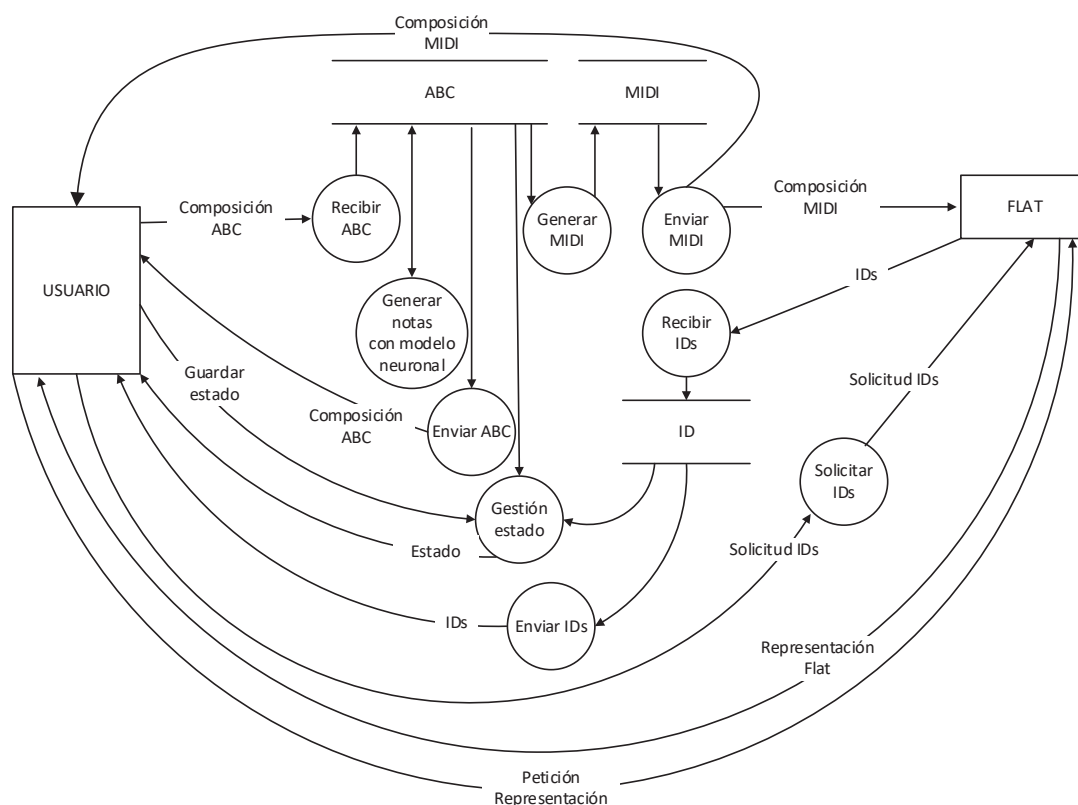


Figura 82 - Diagrama de flujo de datos (iteración 2)
(Fuente: Elaboración propia)

En el nuevo diagrama de flujo de datos, se ha añadido la solicitud de IDs del acontecimiento 4. La recepción y envío de múltiples IDs se ha unido con la recepción y envío de un único ID ya existente previamente, ya que es redundante tener ambas funcionalidades por separado.

2.7.4.2.6. *Diseño de interfaz*

La interfaz actualizada para satisfacer los nuevos requisitos de interfaz, y desencadenar los nuevos acontecimientos es la siguiente:



Figura 83 - Interfaz en PC: Generación (iteración 2)
(Fuente: Elaborado con Proto.io)

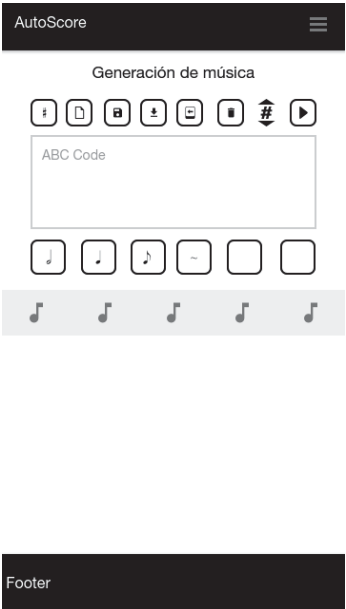


Figura 84 - Interfaz en Smartphone: Generación (iteración 2)
(Fuente: Elaborado con Proto.io)

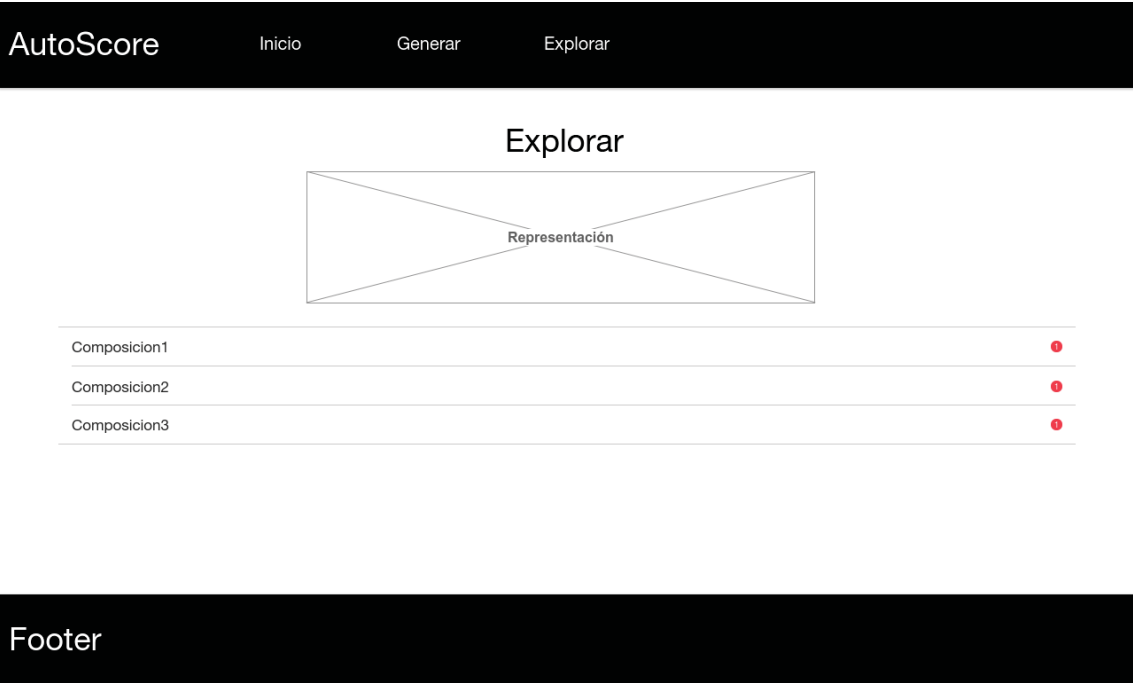


Figura 85 - Interfaz en PC: Exploración (iteración 2)
(Fuente: Elaborado con Proto.io)



Figura 86 - Interfaz en Smartphone: Exploración (iteración 2)
(Fuente: Elaborado con Proto.io)

2.7.4.3. Codificación

Al contenido que ya se tenía, se añaden los nuevos botones que amplían la funcionalidad de esta iteración, permitiendo seleccionar duración de notas, e introducir acordes y símbolos. Se elimina la restricción de un mínimo de 10 notas, puesto que ya no es necesaria en el nuevo modelo con *masking* y *padding* y se establece el nuevo modelo neuronal.

La interfaz de la nueva sección de explicación de composiciones se elabora extendiendo de *layout.html* igual que el resto de secciones ya existentes. Y se añade una nueva ruta en el controlador para manejar las peticiones. En lo que respecta a esta sección, no es necesario realizar cambios en el modelo, puesto que el sistema solo actúa como pasarela para mostrar las composiciones existentes.

Con la nueva interfaz, el diagrama UML que describe la jerarquía de plantillas queda como aparece en la Figura 82 - Diagrama UML de las plantillas de la vista (iteración 2).

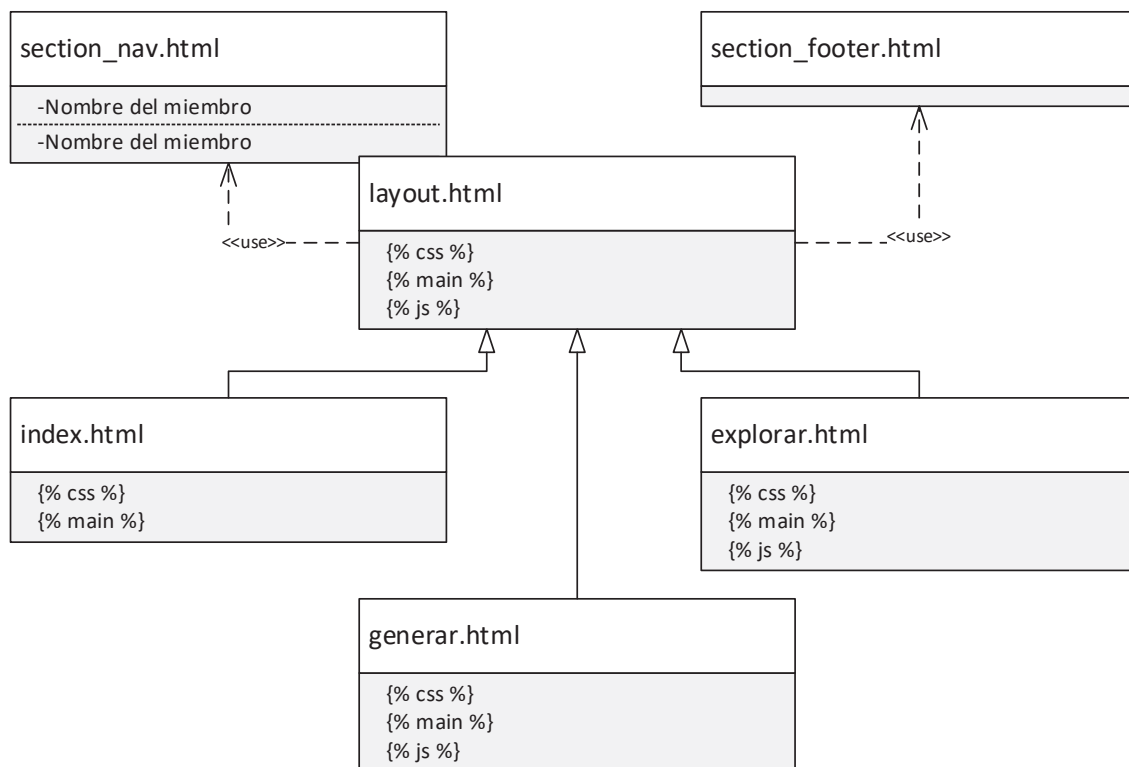


Figura 87 - Diagrama UML de las plantillas de la vista (iteración 2)
(Fuente: Elaboración propia)

2.7.4.4. Pruebas

A continuación se listan las pruebas de integración realizadas:

Prueba	Escenario	Resultado
P1	Inserción de notas: se comprueba que todas las notas se insertan correctamente en la caja de texto, así como la activación y desactivación de alteración de notas, habilitando y deshabilitando las notas correspondientes.	OK
P2	Gestión de estado: se comprueba la funcionalidad de guardado de estado, así como la carga de la composición guardada al volver a la página.	OK
P3	Generación MIDI: se introduce una composición y se comprueba la generación en archivo MIDI.	OK
P4	Generación de notas y visualización: se introduce una composición y se comprueba que la funcionalidad de generación de notas genera nuevas notas, que la plataforma Flat recibe la composición y que esta se representa correctamente en el cliente web.	OK
P5	Generación de notas y visualización: se realiza la generación de notas sin introducir valores y se comprueba que la funcionalidad de generación genera notas sin recibir datos.	OK
P6	Exploración de composiciones: se accede a la lista de composiciones y se selecciona una composición existente.	OK

Tabla 8 - Pruebas de integración (iteración 2)

2.7.4.5. *Producto final*

Tras esta segunda iteración, la interfaz queda de la siguiente manera:

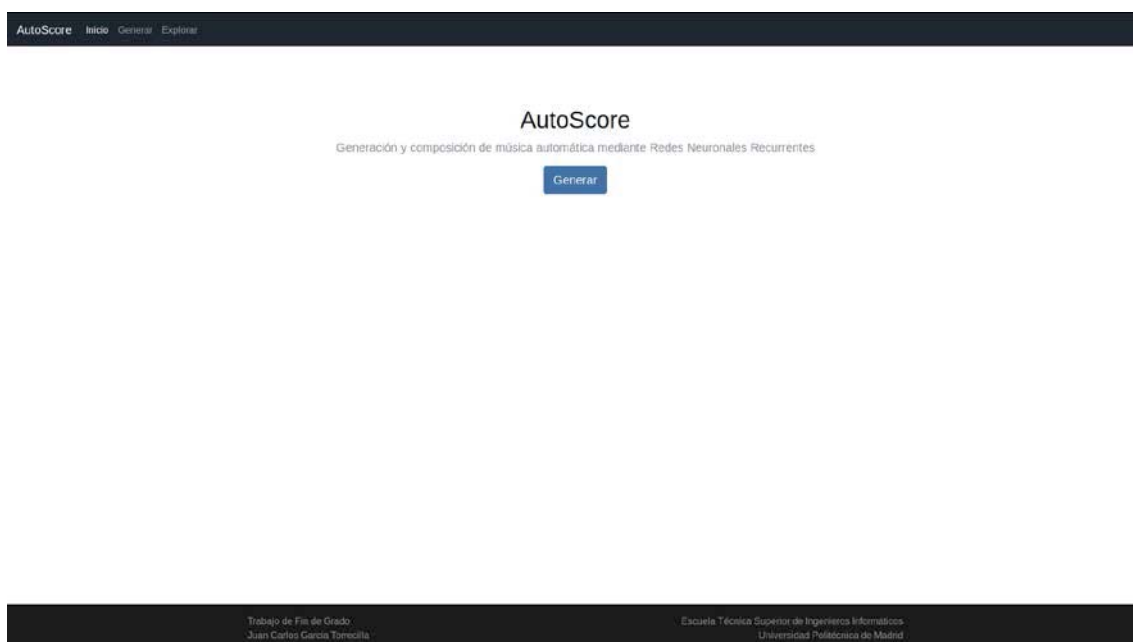


Figura 88 - Captura de index.html en PC (iteración 2)
(Fuente: Elaboración propia)

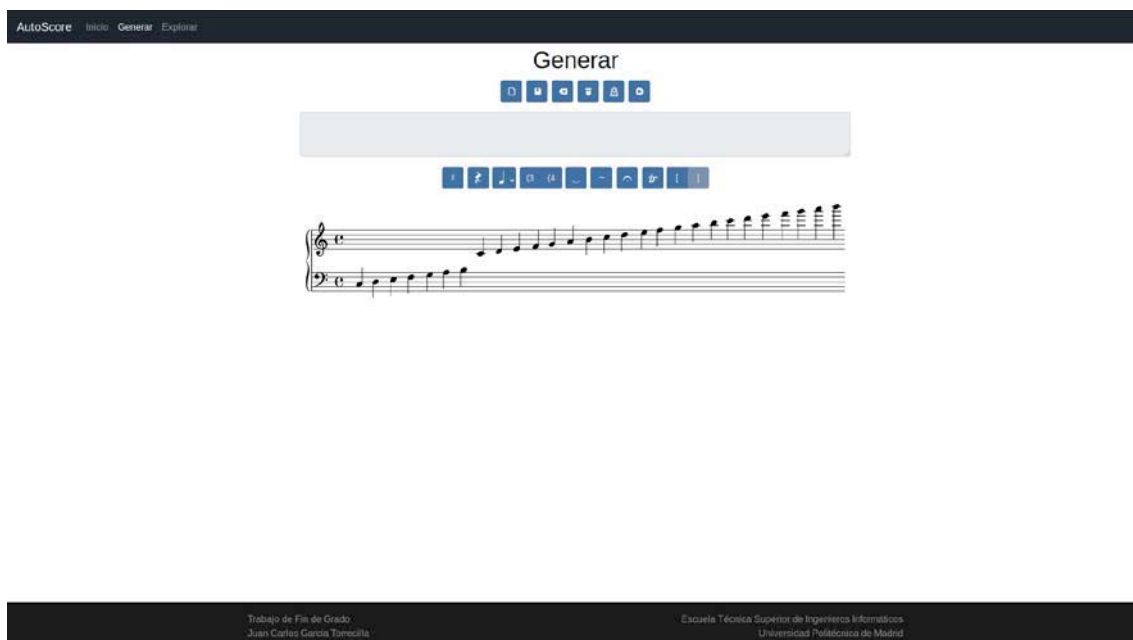


Figura 89 - Captura de generar.html en PC (iteración 2)
(Fuente: Elaboración propia)

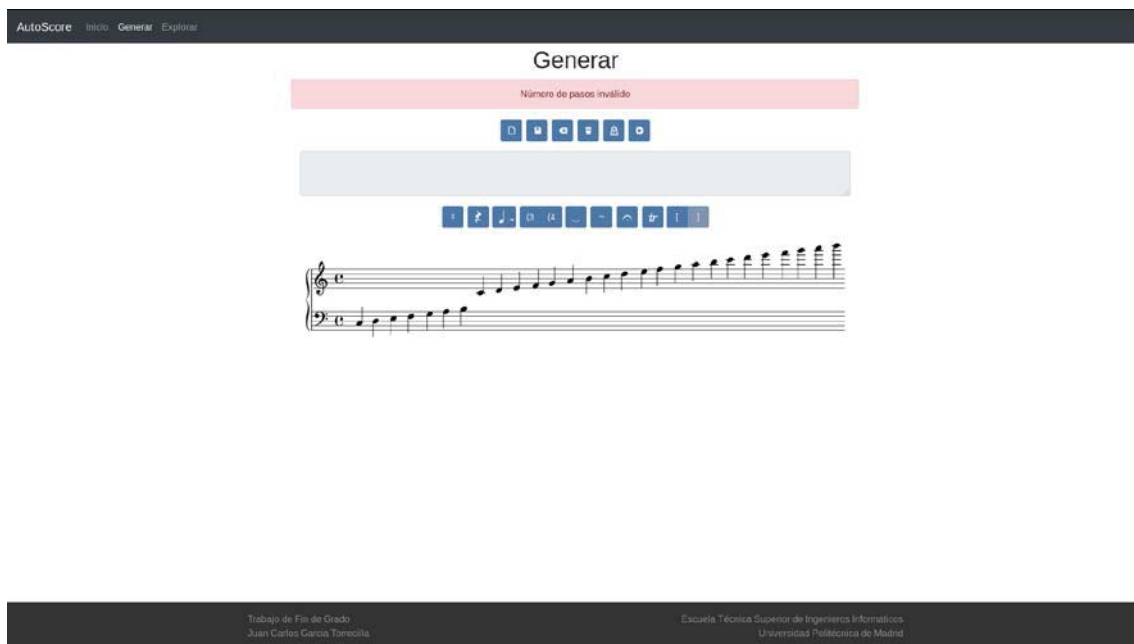


Figura 92 - Captura de control de errores en PC (iteración 2)
(Fuente: Elaboración propia)

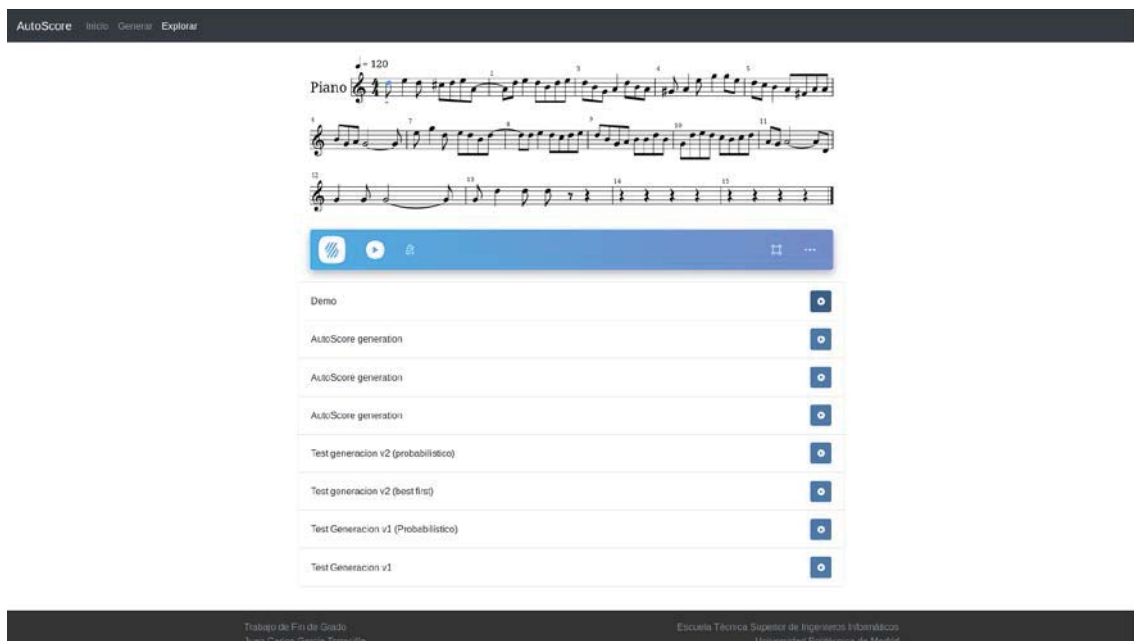


Figura 93 - Captura de explorar.html en PC (iteración 2)
(Fuente: Elaboración propia)

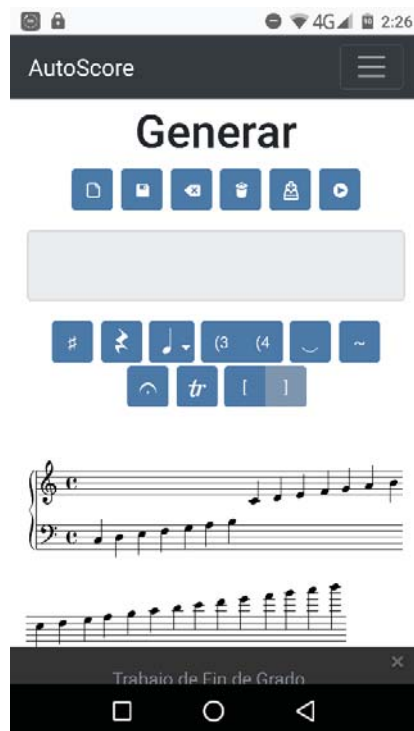
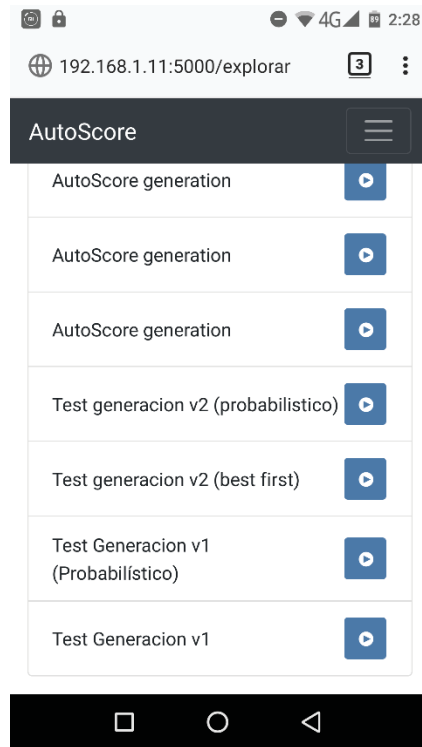


Figura 94 - Captura de generar.html en SmartPhone (iteración 2)
(Fuente: Elaboración propia)



Figura 95 - Detalle de spinner de carga en SmartPhone (iteración 2)
(Fuente: Elaboración propia)



*Figura 96 - Captura de explorar.html en SmartPhone (iteración 2)
(Fuente: Elaboración propia)*

3. RESULTADOS Y CONCLUSIONES

Llegados a este punto, y como resultado del proceso seguido a lo largo del proyecto:

1. Se ha desarrollado un modelo neuronal recurrente capaz de componer música de forma automática desde el enfoque del *language modeling*.
2. Se ha desarrollado una aplicación web *responsive* que permite hacer uso de la herramienta sin conocimientos técnicos. Esta herramienta pone al alcance de músicos y creativos una tecnología muy potente que habitualmente requiere conocimientos técnicos que no todo el mundo posee.

Teniendo esto en cuenta, se han cumplido los objetivos principales planteados en este proyecto. Se ha realizado un proceso de investigación y documentación de las tecnologías de aprendizaje automático, entendiendo sus características y cómo éstas influyen en los resultados. Se ha llevado a cabo una revisión de trabajos relacionados atendiendo al enfoque que dan al problema. Y se ha diseñado una metodología adecuada a las necesidades y objetivos del proyecto, para desarrollar y ejecutar un proceso de investigación de datos y de desarrollo de software de forma conjunta.

A pesar de que el rendimiento, en lo que a métricas estadísticas se refiere, es aun ampliamente mejorable, los últimos resultados son bastante prometedores, si tenemos en cuenta que se ha mejorado la exactitud en casi un 10% respecto a la iteración anterior.

Nótese que, dado el papel fundamental que juega el factor tiempo en este proyecto, y el excesivo tiempo que lleva entrenar una red neuronal, únicamente se han realizado 20 *epochs* de entrenamiento en todas las pruebas de la iteración 2. Por supuesto los resultados serían mejores tras un entrenamiento más prolongado, puesto que aún no se ha alcanzado un punto a partir del cual se produzca *overffiting* como sí ocurrió en la primera iteración. Aun así, 20 *epochs* nos han servido para realizar una comparativa con la que decidir un modelo que entrenar durante mayor tiempo posteriormente.

Los resultados son aun ampliamente mejorables (38% de exactitud). Aun así, véase que se ha mejorado la exactitud en torno a un 8% en la última iteración entrenando únicamente 20 *epochs*. Por ello, es de esperar que, si se continúa entrenando la red neuronal durante más *epochs* los resultados se verán incrementados ligeramente. Por lo que, una vez se han comparado en esta última iteración las diferentes estructuras neuronales recurrentes (SimpleRNN, LSTM, GRU) y decidido el uso de GRU, el siguiente paso es continuar el entrenamiento de dicha red para llevarla al límite e identificar su capacidad. Es decir, continuar el entrenamiento para ver hasta qué punto puede mejorar los resultados antes de alcanzar el *overffiting*.

Otro factor importante es el número de capas. Durante estas dos iteraciones se han realizado pruebas con diferentes estructuras, con una única capa recurrente. Como se explicó en Aprendizaje automático y en Modelado en la primera iteración, la amplitud de la red (cantidad de neuronas por capa) incrementa las capacidades de la red hasta cierto

punto, pero es la profundidad (cantidad de capas) la que le permite aprender conceptos más complejos a la red.

En la primera iteración, se veía como llegado un punto en el aprendizaje, los resultados se estancaban o incluso empeoraban a medida que se avanzaba, y el aumento de neuronas en la capa recurrente incluso empeoraba el *overffiting*. Esto se debe a que la red no es capaz de aprender patrones temporales tan complejos como los que se dan en el *dataset*. Para ello, es necesario también incrementar el número de capas en la red.

Por tanto, una vez se ha identificado qué estructura recurrente funciona mejor, y tras identificar el rendimiento de la red neuronal GRU con 1 capa, es el momento de experimentar con un mayor número de capas en la red en las siguientes iteraciones, para tratar de mejorar el rendimiento dotando a la red de la capacidad para aprender patrones más complejos.

En cualquier caso, como se explicó al inicio de este documento, una mayor exactitud, o cualquier otra métrica estadística, no necesariamente se traduce en una mayor calidad melódica. Puesto que la música tiene un importante componente matemático, pero también un componente creativo esencial, y somos los seres humanos en última instancia los que debemos juzgar si una composición musical es buena o mala, y si la máquina ha conseguido componer música que siga nuestro concepto de belleza musical.

Dicho esto, en la primera iteración, cuando se empleaba la selección de la nota con mayor probabilidad, las composiciones producidas eran habitualmente repetitivas, alternando dos notas que normalmente obtenían la mayor probabilidad. Esto se solucionaba al emplear la selección aleatoria de notas en base a la distribución probabilística, que proporcionaba un mecanismo para evitar esa repetitividad; pero, teniendo en cuenta que únicamente se manejaban notas de la misma duración, el resultado resultaba monótono.

En cambio, empleando el modo de selección de nota con mayor probabilidad, el modelo generado en la segunda iteración no produce composiciones repetitivas habitualmente. Éstas sólo se producen cuando aparecen símbolos poco habituales como “T” (*trill*), “H” (*fermata*), “~” (*roll*), o símbolos de acorde, en este último caso produciendo secuencias de acordes de dos notas que varían de un acorde a otro. Esto posiblemente se deba al desbalanceo de las notas en el *dataset*, ya que se trata de los símbolos que menos aparecen en las composiciones. Para el resto de casos, las composiciones producidas son variadas y se distinguen algunos fragmentos con melodía rítmica. Empleando el modo de selección aleatoria con distribución de probabilidad se solventa el problema de los símbolos producido por el desbalanceo del *dataset*, sin perder la capacidad melódica de las composiciones.

Posterior a la experimentación con múltiples capas recurrentes, considero que también sería interesante elaborar estructuras de red no secuenciales. Es decir, estructuras de red con bifurcaciones y capas neuronales paralelas. Así como combinar capas LSTM y GRU.

Además, otro enfoque alternativo en lo que a la manipulación de datos se refiere, es tratar las notas y su duración como un solo componente, en lugar de elementos temporales consecutivos. Puesto que una nota se compone de un tono y una duración, siendo ambos elementos de la misma nota, sería lógico tratar ambos datos como un solo dato temporal, cuyo vector de características tuviera dimensión 2 (tono, duración) en lugar de 1.

En general, el campo del aprendizaje automático es complejo y en él ninguna tarea es trivial. Entender los datos con los que se trabaja y realizar un preproceso adecuado de los mismos es tan importante o más que el ajuste de hiperparámetros de los algoritmos.

Otros desafíos que deberán ser abordados en futuras líneas de investigación incluyen el estudio y elaboración de un modelo neuronal que contemple la capacidad de producir música con acompañamiento. Es decir, ampliar las capacidades del sistema para que no solo aprenda patrones melódicos y produzca música monofónica; si no que aprenda patrones armónicos, entendiendo como encajan varias pistas musicales en una misma composición, y produciendo música polifónica.

4. BIBLIOGRAFÍA

- [1] L. A. Gatys, A. S. Ecker y M. Bethge, «A Neural Algorithm of Artistic Style,» 2015.
- [2] A. L. Samuel, «Some Studies in Machine Learning Using the Game of Checkers,» *IBM Journal of Research and Development*, vol. 3, nº 3, pp. 210-229, 1959.
- [3] R. Kohavi y F. Provost, «Glossary of terms,» *Machine Learning*, vol. 30, nº 2-3, pp. 271-274, 1998.
- [4] R. O. Duda, P. E. Hart y D. G. Stork, «Learning and Adaptation,» de *Pattern Classification (2nd Ed.)*, New York, John Wiley & Sons, 2001, pp. 16-17.
- [5] T. Jebara, *Machine Learning: Discriminative and Generative*, Boston, MA: Springer US, 2004.
- [6] A. Y. Ng y M. I. Jordan, «On Discriminative vs. Generative Classifiers: A comparison of logistic regression and naive Bayes,» de *Advances in Neural Information Processing Systems 14 (NIPS 2001)*, 2002.
- [7] Y. LeCun, Y. Bengio y G. Hinton, «Deep Learning,» *NATURE*, vol. 521, nº 7533, pp. 436-444, 2015.
- [8] J. Schmidhuber, «Deep Learning in Neural Networks: An Overview,» *NEURAL NETWORKS*, vol. 61, pp. 85-117, 2015.
- [9] W. S. McCulloch y W. H. Pitts, «A logical calculus of the ideas immanent in nervous activity,» *Bulletin of Mathematical Biophysics*, vol. 5, pp. 115-133, 1943.
- [10] M. A. Nielsen, *Neural Networks and Deep Learning*, Determination Press, 2015.
- [11] F. Rosenblatt, *Principles of neurodynamics; perceptrons and the theory of brain mechanisms*, Washington: Spartan Books, 1962.
- [12] M. L. Minsky y S. Papert, *Perceptrons: an introduction to computational geometry*, MIT Press, 1969.
- [13] D. E. Rumelhart, G. E. Hinton y R. J. Williams, «Learning representations by back-propagating errors,» *Nature*, vol. 323, nº 6088, pp. 533-536, 1986.
- [14] S. Haykin, *Neural Networks: A Comprehensive Foundation*, Prentice Hall, 1999 (2nd Ed.).

- [15] C. Olah, «Calculus on Computational Graphs: Backpropagation,» colah's blog, 31 Agosto 2015. [En línea]. Available: <http://colah.github.io/posts/2015-08-Backprop/>. [Último acceso: 19 Noviembre 2017].
- [16] Y. LeCun, I. Kanter y S. A. Solla, «Second Order Properties of Error Surfaces: Learning Time and Generalization,» de *Advances in Neural Information Processing Systems 3 (NIPS 1990)*, 1991.
- [17] Y. Bengio, P. Simard y P. Frasconi, «Learning long-term dependencies with gradient descent is difficult,» *IEEE Transactions on Neural Networks*, vol. 5, nº 2, pp. 157-166, 1994.
- [18] A. Krizhevsky, I. Sutskever y G. E. Hinton, «ImageNet classification with deep convolutional neural networks,» de *Advances in Neural Information Processing Systems 25 (NIPS 2012)*, 2012.
- [19] J. Guo, «Backpropagation Through Time,» 2013.
- [20] R. Pascanu, T. Mikolov y Y. Bengio, «On the difficulty of training Recurrent Neural Networks,» de *International Conference on Machine Learning*, 2013.
- [21] S. Hochreiter y J. Schmidhuber, «Long Short-Term Memory,» *Neural Computation (MIT Press)*, vol. 9, nº 8, pp. 1735-1780, 1997.
- [22] J. Chung, C. Gulcehre, K. Cho y Y. Bengio, «Empirical evaluation of gated recurrent neural networks on sequence modeling,» 2014.
- [23] K. Cho, B. Van Merriënboer, D. Bahdanau y Y. Bengio, «On the properties of neural machine translation: Encoder-decoder approaches,» 2014.
- [24] F. Liang, M. Gotham, M. Tomczak, M. Johnson y J. Shotton, «BachBot,» GitHub, [En línea]. Available: <https://github.com/feynmanliang/bachbot>. [Último acceso: 10 Diciembre 2017].
- [25] M. Tomczak, «Bachbot,» Agosto 2016. [En línea]. Available: <https://goo.gl/MMuEpw>. [Último acceso: 10 Diciembre 2017].
- [26] G. Hadjeres, F. Pachet y F. Nielsen, «DeepBach: a Steerable Model for Bach Chorales Generation,» 2016.
- [27] G. Hadjeres, «DeepBach,» GitHub, [En línea]. Available: <https://github.com/Ghadjeres/DeepBach>. [Último acceso: 10 Diciembre 2017].
- [28] Google Brain Team, «Magenta,» GitHub, [En línea]. Available: <https://github.com/tensorflow/magenta>. [Último acceso: 10 Diciembre 2017].

- [29] Google Brain Team, «Drums RNN,» GitHub, [En línea]. Available: https://github.com/tensorflow/magenta/tree/master/magenta/models/drums_rnn. [Último acceso: 05 Enero 2018].
- [30] Google Brain Team, «Melody RNN,» Github, [En línea]. Available: https://github.com/tensorflow/magenta/tree/master/magenta/models/melody_rnn. [Último acceso: 05 Enero 2018].
- [31] Google Brain Team, «Polyphony RNN,» GitHub, [En línea]. Available: https://github.com/tensorflow/magenta/tree/master/magenta/models/polyphony_rnn. [Último acceso: 05 Enero 2018].
- [32] Google Brain Team, «Performance RNN,» GitHub, [En línea]. Available: https://github.com/tensorflow/magenta/tree/master/magenta/models/performance_rnn. [Último acceso: 05 Enero 2018].
- [33] Google Brain Team, «Pianoroll RNN-NADE,» GitHub, [En línea]. Available: https://github.com/tensorflow/magenta/tree/master/magenta/models/pianoroll_rnn_nade. [Último acceso: 05 Enero 2015].
- [34] N. Boulanger-Lewandowski, Y. Bengio y P. Vincent, «Modeling Temporal Dependencies in High-Dimensional Sequences: Application to Polyphonic Music Generation and Transcription,» de *International Conference on Machine Learning (ICML 2012)*, Edinburgh, Scotland, 2012.
- [35] A. van den Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior y K. Kavukcoglu, «WaveNet: A Generative Model for Raw Audio,» 2016.
- [36] A. Nayebi y M. Vitelli, «GRUV: Algorithmic Music Generation using,» *Course CS224D: Deep Learning for Natural Language Processing (Stanford)*, 2015.
- [37] M. Vitelli y A. Nayebi, «GRUV,» GitHub, [En línea]. Available: <https://github.com/MattVitelli/GRUV>. [Último acceso: 10 Diciembre 2017].
- [38] D. D. Johnson, «Generating Polyphonic Music Using Tied Parallel,» de *EvoMusArt 2017*, Amsterdam, Netherlands, 2017.
- [39] D. D. Johnson, «Biaxial RNN Music Composition,» GitHub, [En línea]. Available: <https://github.com/hexahedria/biaxial-rnn-music-composition>. [Último acceso: 11 Diciembre 2017].
- [40] N. Agarwala, Y. Inoue y A. Sly, «Music Composition using Recurrent Neural Networks».

- [41] N. Agarwala, Y. Inoue y A. Sly, «CS224N Final Project,» GitHub, [En línea]. Available: https://github.com/yinoue93/CS224N_proj. [Último acceso: 11 Diciembre 2017].
- [42] A. Kelly, *Changing Software Development : Learning to be Agile*, John Wiley & Sons, 2008.
- [43] J. Sutherland, Ph.D. , K. Schwaber y Co-Creators of Scrum , *The Scrum Papers: Nuts, Bolts, and Origins of an Agile Process*, 2007.
- [44] A. I. R. L. Azevedo y M. F. Santos, «KDD, SEMMA and CRISP-DM: a parallel overview,» *IADS-DM*, 2008.
- [45] R. Wirth y J. Hipp, «CRISP-DM: Towards a standard process model for data mining,» *Proceedings of the 4th international conference on the practical applications of knowledge discovery and data mining*, pp. 29-39, 2000.
- [46] R. Eldan y O. Shamir, «The Power of Depth for Feedforward Neural Networks,» de *Conference on Learning Theory*, 2016.
- [47] H.-T. Cheng, L. Koc, J. Harmsen, T. Shaked, T. Chandra, H. Aradhye, G. Anderson, G. Corrado, W. Chai, M. Ispir, R. Anil, Z. Haque, L. Hong, V. Jain, X. Liu y H. Shah, «Wide & Deep Learning for Recommender Systems,» de *Proceedings of the 1st Workshop on Deep Learning for Recommender Systems*, 2016.
- [48] R. Ge, F. Huang, C. Jin y Y. Yuan, «Escaping From Saddle Points – Online Stochastic Gradient for Tensor Decomposition,» de *Conference on Learning Theory*, 2015.
- [49] IEEE, «IEEE Std 830: Recommended Practice for Software Requirements,» 1998.
- [50] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever y R. Salakhutdinov, «Dropout: A Simple Way to Prevent Neural Networks from Overfitting,» *Journal of Machine Learning Research*, vol. 15, nº 1, pp. 1929-1958, 2014.

Este documento esta firmado por



Firmante	CN=tfgm.fi.upm.es, OU=CCFI, O=Facultad de Informatica - UPM, C=ES
Fecha/Hora	Mon Jan 15 21:14:52 CET 2018
Emisor del Certificado	EMAILADDRESS=camanager@fi.upm.es, CN=CA Facultad de Informatica, O=Facultad de Informatica - UPM, C=ES
Numero de Serie	630
Metodo	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature)