

UNIVERSIDAD DE CANTABRIA

FACULTAD DE CIENCIAS

GRADO EN INGENIERÍA INFORMÁTICA

Trabajo de Fin de Grado

*Interpretación de música de piano usando
técnicas de Deep Learning*



Autor: Álvaro López García.
Director: Cristina Tîrnăucă.

Fecha: 8 de Septiembre de 2022.

AGRADECIMIENTOS

El desarrollo de este documento no hubiera sido posible sin el apoyo incondicional de mi familia. Desde mi núcleo familiar más cercano hasta los satélites allegados, todos han colaborado en esta hazaña. He de dar las gracias a mis amigos de “Casi Pitas” por ayudarme a guardar la compostura, cambiarme las mascarillas mojadas y entenderme. También he de dirigirme a mis amigos del conservatorio para agradecerles su apoyo. Es muy difícil concretar los beneficios que la música aporta a una persona, pero en mi caso se podrían resumir en unos nombres.

Agradecemos el apoyo del grupo de supercomputación de Santander de la Universidad de Cantabria que proporcionó acceso al supercomputador Altamira del Instituto de Física de Cantabria (IFCA-CSIC), miembro de la Red Española de Supercomputación, para realizar simulaciones/análisis. Esta sección no puede concluir sin dirigirme a mi tutora Cristina, por los valores que inspira en todos los que hemos acudido a sus clases, y para agradecerle el honor que supone ser su alumno asignado.

RESUMEN

La correcta interpretación de la música es una tarea extremadamente compleja. Los intérpretes profesionales dedican toda una vida a perfeccionarse en esta labor. La preparación requerida junto con los costes asociados a la infraestructura hace que la producción de una determinada obra sea altamente costosa. Y pese a esto, la ciencia aún no nos ha brindado una solución que no requiera de tantos recursos.

El objetivo de este Trabajo Fin de Grado es el desarrollo de un agente inteligente basado en técnicas de Aprendizaje Profundo (en inglés, *Deep Learning*) que sea capaz de interpretar música de piano de la forma más humana posible. Hasta ahora, las técnicas existentes para la generación de interpretaciones se basaban en grandes bancos de sonido y las interpretaciones que ofrecían como resultado eran evidentemente artificiales. Es por esto por lo que se desea crear un agente que no precise de tales cantidades de memoria y que realice interpretaciones indistinguibles de las realizadas por humanos.

Para el desarrollo de dicho agente se emplearán Redes Neuronales Autorregresivas, dados los buenos resultados que estas han mostrado en tareas de *Text-to-Speech*, que a su vez presentan una gran similitud con el tema abordado. El entrenamiento de dicho modelo se realizará sobre el “MAESTRO” dataset (*MIDI and Audio Edited for Synchronous TRacks and Organization*) y su implementación se realizará en Python, usando la librería *TensorFlow*.

Palabras clave

Música

TTS

Aprendizaje Automático

MAESTRO dataset

ARNNs

Aprendizaje Profundo

WaveNet

Redes Neuronales

Redes Neuronales Convolucionales

ABSTRACT

The correct interpretation of music is an extremely complex task. Professional interpreters dedicate a lifetime to perfecting themselves in this work. The preparation required, together with the costs associated with the infrastructure, makes the production of a given work highly expensive. And despite this, science has not yet provided us with a solution that does not require so many resources.

The objective of this Final Degree Project is the development of an intelligent agent based on Deep Learning techniques that is capable of interpreting piano music in the most human way possible. Until now, existing techniques for generating performances were based on large banks of sounds, and the resulting performances were patently artificial. This is why it would be very useful to create an agent that does not require such amounts of memory and that performs interpretations indistinguishable from those made by humans.

For the development of said agent, Autoregressive Neural Networks will be used, given the good results that these have shown in *Text-to-Speech* tasks, which in turn present a great similarity with the topic addressed. The training of said model will be carried out on the “MAESTRO” *dataset* (MIDI and Audio Edited for Synchronous TRAcks and Organization) and its implementation will be carried out in Python, using the *TensorFlow* library.

Keywords

Music TTS Machine Learning MAESTRO dataset ARNNs
Deep Learning WaveNet Neural Networks Convolutional Neural Networks

Índice de figuras

1.1	Extracto de la melodía del 1 ^{er} movimiento de <i>Eine Kleine Nachtmusik</i> (W.A. Mozart, K.525).	1
1.2	Espectrograma que se corresponde al comienzo de <i>Grandes Etudes de Paganini - I</i> (F. Liszt, S.141).	2
1.3	Objetivos de <i>Automatic Speech Recognition</i> y <i>Text-to-Speech</i>	2
1.4	Esquema del sintetizador de voz inventado por Kempelen.	3
1.5	Arquitectura del modelo Tacotron2.	3
2.1	Representación de la función Sigmoide $\sigma(x)$	8
2.2	Estructura de la neurona.	9
2.3	Estructura de una red neuronal artificial completamente conexa de cuatro capas.	10
2.4	Ejemplo de la convolución entre dos funciones f y g para distintos valores de ϵ	14
2.5	Multiplicación de los valores de entrada de una capa convolucional 6×6 con un kernel 3×3	14
2.6	Arquitectura completa de una red neuronal convolucional.	15
3.1	Presentación de <i>WaveNet</i> como el motor detrás de las voces del asistente de voz de Google en la Google I/O 2018.	16
3.2	Composición de un acorde de La Mayor como suma de ondas.	17
3.3	Esquema de la arquitectura del modelo.	17
3.4	Representación de un <i>stack</i> de <i>causal convolutional layers</i>	18
3.5	Representación de un <i>stack</i> de <i>dilated causal convolutional layers</i>	18
3.6	Función aplicada para la codificación según la μ -law para distintos valores de μ	20
3.7	Ejemplo de reconstrucción de una onda sinusoidal con distintos valores de μ y con distintos valores de k dados por 2^{n^o} de bits	21
3.8	Valores de onda original y comprimida empleando el algoritmo descrito para un fichero del <i>dataset</i>	21
3.9	Ejemplo de aplicación de <i>One-Hot encoding</i> sobre un supuesto <i>dataset</i> de clasificación por colores.	22
3.10	Valores dados por $tanh(x)$	23
3.11	Valores de ReLU dados por $f(x)$	24
3.13	Valores normalizados del número de parámetros y error derivado de la compresión, y suma de estos.	27
3.12	Esquema de la arquitectura del modelo desarrollado.	28
4.1	Estadísticas de los datos en el <i>dataset</i> en función del conjunto: entrenamiento, validación o test.	29
4.2	Distribución de los compositores en función del número de interpretaciones de piezas que se les atribuyen en el <i>dataset</i>	30
4.3	Representaciones de los dos tipos de datos que se manejan a alto nivel. “ <i>Pianoroll</i> ” para los archivos MIDI y valores de amplitud de onda para los archivos Wav.	31

4.4	Esquema del proceso aplicado a los datos para el entrenamiento.	32
4.5	Diagrama de bigotes con los valores de amplitud de onda durante los 5 primeros segundos por cada 0,25 segundos.	35
4.6	Evolución de la precisión y pérdida del modelo a lo largo del entrenamiento para los conjuntos de entrenamiento y <i>cross-validation</i>	36
A.1	Dispersión de los pares de valores tamaño del motor – precio.	40
A.2	Evolución de los coeficientes, el coste y la recta de la regresión durante el entrenamiento.	41
A.3	Dispersión de los pares de valores alcohol - dióxido de azufre total, así como su distribución a lo largo de ambos ejes.	42
A.4	Evolución de los coeficientes, el coste y la recta de la regresión durante el entrenamiento.	43

Índice general

1	Introducción	1
1.1	<i>Text-to-Speech</i> , un problema similar	2
2	Modelos anteriores	5
2.1	Regresión Lineal Multivariante	6
2.2	Regresión Logística	8
2.3	Redes Neuronales	9
2.3.1	Optimizaciones para el entrenamiento	12
2.4	Redes Neuronales Convolucionales	13
3	WaveNet	16
3.1	Capas convolucionales	17
3.1.1	Convoluciones Causales	17
3.1.2	Convoluciones Causales Dilatadas	18
3.2	Capa de salida	19
3.2.1	μ - <i>law companding</i>	19
3.2.2	<i>One-Hot encoding</i>	22
3.3	Funciones de activación	23
3.4	<i>Residual and skip connections</i>	24
3.5	WaveNets condicionadas	24
3.6	<i>Context stacks</i>	25
3.7	Arquitectura del modelo	25
3.7.1	Discretización y diferenciación	26
4	Entrenamiento del modelo y resultados	29
4.1	Maestro <i>dataset</i>	29
4.1.1	Ficheros MIDI	30
4.1.2	Ficheros Wav	30
4.2	Herramientas empleadas	31
4.3	Procesado de los datos	31
4.4	Implementación del entrenamiento	33
4.5	Métricas de evaluación	35
4.6	Resultados	36
5	Conclusiones	37
5.1	Aportaciones y trabajo futuro	37
A	Aplicaciones de modelos anteriores	40

CAPÍTULO 1

INTRODUCCIÓN

La correcta interpretación de la música es una tarea extremadamente compleja. La estética del género ha cambiado a lo largo de las distintas etapas de la historia en respuesta a todo tipo de factores. Y es por esto que los profesionales se entrena durante décadas hasta conseguir dominar la tarea interpretativa.

La complejidad que involucra una interpretación, así como la preparación previa requerida, hace que en el mundo de la producción musical haya unos grandes costes asociados a esto, sin haber mencionado siquiera los gastos que se derivan de la infraestructura y el personal que implica la grabación y el procesado de dicho material. Y pese a esto, la ciencia no ha producido otra forma que no implique semejante cantidad de recursos a la hora de obtener una previsualización (aunque pudiera ser de menor calidad) de cómo sonaría la interpretación real de una obra. Es por esto que nuestro propósito a lo largo de este documento es la creación de un agente inteligente basado en *Deep Learning* que nos permita obtener, mediante software, tal aproximación.

A priori se podría pensar que el uso de agentes inteligentes o modelos de predicción (como pretendemos) es innecesario, pues a partir de la partitura de una determinada obra (como la de la Figura 1.1) podemos extraer información acerca de la duración, altura (y por tanto frecuencia) e intensidad de cada una de las notas que la componen. También podríamos contar con una biblioteca de sonidos de duración arbitraria que estuvieran etiquetados en función de instrumento, altura e intensidad; pues aunque es grande, el rango de combinaciones entre los valores de estos tres parámetros no es infinito. Y con respecto a la duración, podríamos quedarnos solo con una sección de la duración que quisiéramos del comienzo de cada clip de audio.

De este modo podríamos concatenar los sonidos pertinentes hasta generar melodías que se ajustasen en la más estricta perfección a lo que en su momento especificó el compositor. Esta solución ha sido la más común cuando se ha tratado de abordar este problema. Pues algorítmicamente es sencilla, escalable, y es fácil ver que operaría en tiempo lineal, ya que el esfuerzo de decisión que hemos especificado anteriormente se puede modelar como una operación de tiempo constante que se realizaría una vez por cada nota a generar.

Esta es precisamente la filosofía detrás del formato MIDI (*Musical Instrument Digital Interface*). MIDI es un estándar tecnológico que entre otras cosas define un formato de archivo en el que se graban y especifican estos parámetros previamente enumerados de una determinada interpretación, de manera que son portables y pueden ser reconstruidos a pos-



Figura 1.1: Extracto de la melodía del 1^{er} movimiento de *Eine Kleine Nachtmusik* (W.A. Mozart, K.525).

teriori por cualquier tipo de software. Sin embargo, el resultado de estas estrategias suelen ser interpretaciones un tanto rígidas que, como hemos dicho anteriormente, se ajustan a la más estricta perfección a lo que el compositor especificó, pero por algún motivo a nuestro oído le resulta evidente que son artificiales. Hemos de preguntarnos entonces ¿qué hace que una interpretación sea percibida como una interpretación llevada a cabo por humanos? Y si procuramos crear un modelo artificial que aproxime una interpretación humana, ¿en qué elementos o atributos de una interpretación quedan patentes estos aspectos tan humanos?

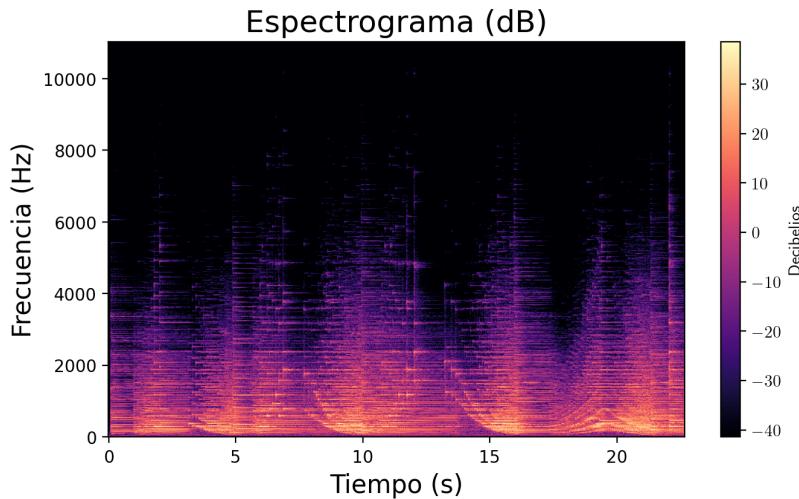


Figura 1.2: Espectrograma que se corresponde al comienzo de *Grandes Etudes de Paganini - I* (F. Liszt, S.141).

La respuesta corta a todo esto es que la música no es una ciencia exacta. Y existe un sinfín de interpretaciones que pueden satisfacer la especificación del compositor sin resultar del todo agradables al oído. En el producto de la interpretación intervienen muchísimos factores que se pueden ver plasmados en el espectrograma de la Figura 1.2. Sí, en ese espectrograma aparecen las pulsaciones del piano, pero también aparecen otras muchas frecuencias que pueden derivarse, desde el ruido ambiente captado por los micrófonos hasta la serie de frecuencias armónicas que genera el propio instrumento y que son únicas para cada uno.

1.1. *Text-to-Speech*, un problema similar

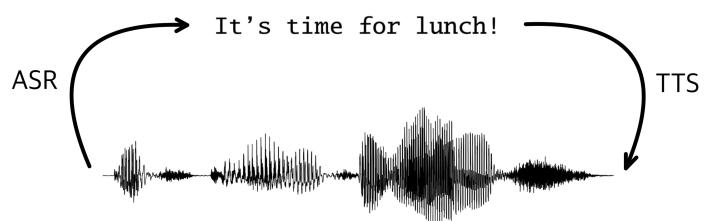


Figura 1.3: Objetivos de *Automatic Speech Recognition* y *Text-to-Speech*.

El lenguaje natural es una de las mayores abstracciones realizadas por el ser humano. Nos permite describir y construir realidades a partir de palabras y, mediante una serie de reglas de pronunciación, transmitirlas entre nosotros. Es por esto que la ciencia lleva coqueteando siglos con la creación de agentes inteligentes que sean capaces de reconocer palabras y/o

pronunciarlas. Como se ilustra en la Figura 1.3, el problema del reconocimiento de palabras recibe el nombre de *Automatic Speech Recognition* (ASR), mientras que el de la síntesis del audio correspondiente a una serie de palabras se llama *Text-to-Speech* (TTS).

Los primeros intentos de solucionar este último problema datan del Siglo XVIII, cuando el inventor Wolfgang von Kempelen (famoso también por otras hazañas) construyó el primer sintetizador de voz que era capaz de reproducir frases completas (Jurafsky & Martin, s.f.). Como se puede ver en la Figura 1.4, este sintetizador es una máquina compuesta por diversas partes que tratan de imitar los distintos órganos del cuerpo humano que intervienen en el habla. Entre estas partes se destacan: un fuelle, que exhala el aire como harían los pulmones; una boquilla de goma, con una abertura a modo de nariz; una lengüeta, que simula la vibración de las cuerdas vocales con el paso del aire; silbatos, para las consonantes fricativas; y fuelles auxiliares, para las oclusivas. Todo esto se podía manipular mediante una serie de mandos que debía accionar un operario, pero hoy en día se busca crear sistemas automáticos que no requieran intervención humana alguna.

Trasladándonos al pasado reciente, vemos cómo ambas vertientes del problema de la transcripción texto-audio han sido objeto de estudio durante décadas. Sin embargo, a nosotros nos interesa más el problema del TTS, pues la interpretación de música es un problema objetivamente similar. En ambos casos partimos de una especie de lenguaje que de algún modo determina los sonidos que queremos generar, pero pese a disponer de estos sonidos por separado, la concatenación directa de estos genera resultados evidentemente artificiales. Para esto necesitamos captar elementos que no se plasman en la especificación formal. En el caso del TTS, nos interesa captar elementos como los derivados del propio habla, como pueden ser el sonido percutivo de los labios del hablante, el acento si queremos emular un hablante de una determinada región geográfica o hasta elementos anatómicos que condicionan el timbre de la voz de esa hipotética persona. En la interpretación de música nos interesan las interacciones entre las distintas notas, la resonancia característica del propio instrumento o la sonoridad de la sala en la que artificialmente se está realizando la interpretación.

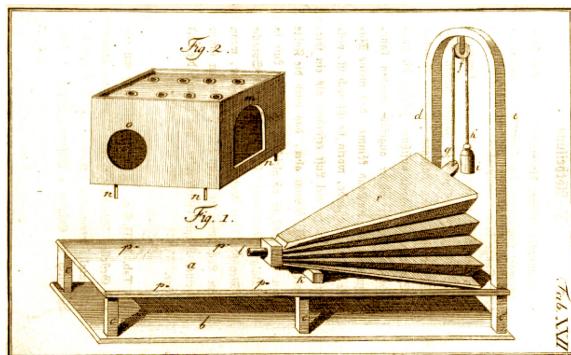


Figura 1.4: Esquema del sintetizador de voz inventado por Kempelen¹.

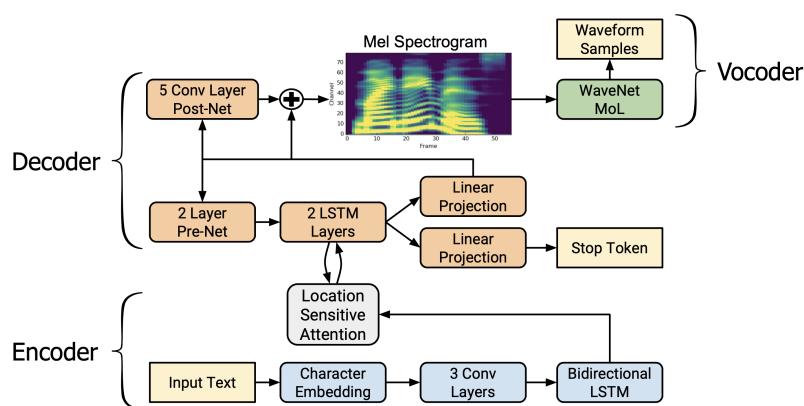


Figura 1.5: Arquitectura del modelo Tacotron2².

¹Fuente: Jurafsky y Martin, s.f.

Con respecto a aspectos más técnicos del problema podemos hablar de elementos como la arquitectura *encoder-decoder* o las Redes Neuronales Recurrentes (RNNs), cuyo uso viene siendo prácticamente obligatorio en las soluciones que se han propuesto recientemente a dicho problema. Como ejemplo de esto, mostramos en la Figura 1.5 la arquitectura del modelo Tacotron2.

Dicho modelo se apoya en la posibilidad que ofrece la arquitectura *encoder-decoder* de convertir cadenas de longitud variable en otras de longitud distinta. En ese caso las cadenas de entrada serían las secuencias de caracteres cuyos fonemas se quieren pronunciar y la salida serían los distintos valores de intensidad de las distintas frecuencias en cada instante de tiempo en forma de espectrograma Mel (como el de la Figura 1.2). Posteriormente, un segundo modelo (en este caso una *WaveNet*, como también se empleará en este caso) se encarga de traducir estos espectrogramas en ondas de audio. La técnica empleada en esta segunda parte del modelo en el que se traducen espectrogramas Mel en audio es lo que recibe el nombre de *Vocoding*.

Toda la primera parte de extracción de características concretas del audio a partir de texto debe hacerse debido a la notación de la que partimos. En el caso del texto, esta notación no incluye información acerca de la duración o la frecuencia de los sonidos a generar. Por lo que la mera extracción de dichos datos ha de realizarse por separado. En nuestro caso, la notación de la que partimos sí que contiene tal información: como más adelante se verá, sabemos qué tecla se pulsa, en qué momento, hasta qué momento y con qué intensidad. Por tanto, en este trabajo se entrenará un modelo que traduzca tal especificación en audio, a modo de *Vocoder*.

²Fuente: Jurafsky y Martin, s.f.

CAPÍTULO 2

MODELOS ANTERIORES

La motivación detrás de la aplicación de técnicas de Aprendizaje Automático muchas veces es la sencillez y eficiencia en lo que se refiere al *tradeoff* entre precisión de los resultados y el esfuerzo computacional que requieren. En muchas ocasiones las tareas que realizan estos modelos pueden ser desempeñadas mediante el diseño de soluciones algorítmicas. Pero la complejidad de estas soluciones aumenta considerablemente conforme la tarea se complica, además de la falta de escalabilidad y la especialización que requieren por parte del programador estos sucedáneos.

Es necesario, para obtener un correcto entendimiento de los modelos de predicción que se pretende emplear en este documento, comprender aquellos que los anteceden en el contexto del Aprendizaje Automático. Para ello se deben clarificar una serie de conceptos inherentes a dicha disciplina.

En el desarrollo de cualquier modelo de predicción se ha de tener claros tres conceptos clave: la tarea, que ese hipotético modelo ha de aprender a realizar; la experiencia, a partir de la cual aprende; y el rendimiento, que nos servirá para medir cuán bien nuestro modelo desempeña la mencionada tarea.

Entendemos por tarea a aquella labor que queremos que nuestro modelo aprenda a desempeñar. No ha de confundirse esto con el proceso de aprender a realizar dicha tarea. Esta tarea suele describirse dependiendo de cómo el modelo procese los datos de entrada. Podemos denotar una tupla de estos datos de entrada como un vector $x \in \mathbb{R}^n$, donde cada elemento x_i representa cada uno de los atributos de dicha tupla. La semántica de cada uno de estos elementos dependerá del *dataset* que se esté manipulando. Dependiendo de cómo se entrene al modelo, la tarea que desempeñará cambiará. Los avances en este campo han hecho que habitualmente los modelos se categoricen en función de la tarea que realizan. Algunas de las tareas más habituales son las de clasificación, regresión o transcripción.

El rendimiento de un modelo es una métrica que nos sirve para evaluar cuantitativamente cómo de bien realiza la tarea que pretendemos que desempeñe. Esta medida suele basarse en la precisión del modelo, para lo que a su vez pueden utilizarse métricas como el Error Cuadrático Medio (entre la predicción y el valor real) por ejemplo. Pero la labor de determinar qué métrica escoger para evaluar la precisión no es trivial, y dado que determinará la correcta evaluación del modelo, también es importante. Llegados a este punto hay que mencionar que nos interesa que esta evaluación del rendimiento del modelo se realice con datos que no han aparecido en la fase de entrenamiento, pues es de suponer que el error para esos datos ya se ha corregido en dicha etapa. Por ello se necesita determinar un conjunto de test que esté compuesto de datos que no aparezcan en el conjunto de entrenamiento, y que por tanto emulen el funcionamiento del modelo en el mundo real.

Con respecto a la experiencia de un modelo, se puede dividir a los modelos en dos grandes categorías: modelos supervisados y no supervisados. La diferencia entre estos dos grandes

grupos radica en la presencia (o ausencia) de etiquetas en el *dataset*. En los modelos supervisados a cada tupla del *dataset* se le asocia una etiqueta, mientras que en los no supervisados cada tupla se compone únicamente de una serie de características de la entidad que representa. Los modelos supervisados se emplean típicamente para labores de clasificación mientras que los no supervisados se emplean para otras tareas como el *clustering*. Aunque hay que tener en cuenta que la línea entre estas dos grandes clases es muy difusa ya que no existe una definición formal de esta categorización.

También hay que aclarar que, para la descripción del *dataset*, se empleará la notación conocida como “matriz de diseño”. En esta notación se emplea una gran matriz X para describir los ejemplos del *dataset*, donde cada fila de la matriz es una tupla de datos y donde en cada columna tenemos categorizados los distintos atributos de cada elemento del *dataset*. De forma que mediante X_{ij} denotaríamos la j -ésima característica del i -ésimo elemento del *dataset*. Para indicar el valor esperado de la predicción para cada tupla de valores de X nos serviremos del vector y . De este modo, y_i será el valor esperado de la predicción para la tupla de datos asociada al i -ésimo valor de *dataset*.

2.1. Regresión Lineal Multivariante

La Regresión Lineal Multivariante es uno de los algoritmos de Aprendizaje Automático más básicos y nos servirá para asentar los conceptos que se han expuesto anteriormente.

Como su nombre indica, este modelo sirve para resolver un problema de regresión, donde la entrada está constituida por vectores $x \in \mathbb{R}^n$ y la salida es un número $y \in \mathbb{R}$, de tal forma que el modelo aproxima una función lineal que mejor se ajuste a los datos de entrenamiento. Se utilizará \hat{y} para denotar la predicción del modelo e y para su valor real. Podemos definir entonces la predicción de nuestro modelo como

$$\hat{y} = w^t x + b$$

donde $w \in \mathbb{R}^n$ es un vector de parámetros y b (*bias parameter*) es el término independiente de la regresión. Dada esta especificación, w_j será el coeficiente que multiplique el j -ésimo término del vector de entrada. Estos parámetros serán los que condicione el comportamiento del modelo.

Hay que determinar ahora cómo evaluaremos el rendimiento de nuestro modelo. Y antes de abordar la especificación de la métrica en sí, se debe reservar una parte de nuestro *dataset* para la evaluación del modelo. Pues como se ha mencionado anteriormente, queremos evaluar el modelo en un contexto que se asemeje en la medida de lo posible al mundo real. Dicho esto, supondremos que tenemos una matriz de diseño de m filas (m ejemplos), que nos reservaremos únicamente para la evaluación del modelo; y un vector y , que contendrá a cada uno de los valores objetivo de la regresión para los mencionados ejemplos. Nos referiremos a este *dataset* de evaluación como conjunto de test, y denotaremos respectivamente con $X^{(\text{test})}$ e $y^{(\text{test})}$ a la matriz y vector descritos.

Una vez hemos definido esto, el rendimiento del modelo se puede evaluar mediante el error cuadrático medio (MSE, *Mean Squared Error*) entre los valores predichos y los valores esperados. Siendo coherentes con la notación anterior, $\hat{y}^{(\text{test})}$ indica los valores de la predicción para los ejemplos del conjunto de entrenamiento, para los que podemos definir el error cuadrático medio como

$$\text{MSE}_{\text{test}} = \frac{1}{m} \sum_{i=1}^m (\hat{y}_i^{(\text{test})} - y_i^{(\text{test})})^2.$$

De tal forma que tenemos una métrica con la que evaluar la precisión del modelo dados los coeficientes de la regresión w (y b) sobre este conjunto de test.

Podemos servirnos también de esta métrica del error cuadrático medio para entrenar al modelo. Pues intuitivamente vemos cómo el modelo habrá interpolado perfectamente los puntos que queremos aproximar mediante la regresión conforme esta métrica tienda a 0. Para hacer esto, podemos obtener aquel vector de coeficientes para los que el vector gradiente de la función MSE_{train} es 0, así como el valor del b (término independiente) que hace la derivada parcial con respecto de ese término nula. Sin embargo, el cálculo directo de estos valores puede ser costoso computacionalmente ya que conforme aumenta el número de ejemplos y de características del *dataset*, el número de operaciones requeridas para dicho cálculo aumenta sensiblemente (consecuencia directa del aumento de las dimensiones de X). Esta falta de escalabilidad hace que el uso de soluciones iterativas sea común y más conveniente.

Para ilustrar esto veremos el caso del algoritmo del gradiente descendente. El algoritmo del gradiente descendente es un algoritmo de entrenamiento que se sirve de la aplicación del vector gradiente sobre una función de coste (que definiremos a continuación) con el objetivo de calcular los coeficientes de la regresión que estadísticamente mejor aproximen los valores objetivo ($y^{(train)}$) de nuestros ejemplos de entrenamiento ($X^{(train)}$). Cada coeficiente de la regresión, que se irá actualizando en cada iteración, se modificará de acuerdo al vector gradiente y su módulo será proporcional a un metaparámetro que llamaremos “ratio de aprendizaje” (α). Para el cálculo de dichos coeficientes hemos de definir la siguiente función de coste:

$$J(w, b) = \frac{1}{2m} \sum_{i=1}^m ((w^t X_i^{(train)} + b) - y_i^{(train)})^2 \left(= \frac{1}{2} MSE_{(train)} \right).$$

Como hemos indicado y se puede apreciar, esta función de coste $J(w, b)$ viene del error medio cuadrático del que ya hemos hablado anteriormente, y por tanto hereda sus propiedades. Por lo que esta será la función a minimizar durante la fase de entrenamiento. Para ello emplearemos el siguiente algoritmo:

```

while no haya convergencia do
     $b = b - \alpha \frac{\partial}{\partial b} J(w, b) = b - \alpha * \frac{1}{m} \sum_{i=1}^m ((w^t X_i^{(train)} + b) - y_i^{(train)}) * 1$ 
     $w_1 = w_1 - \alpha \frac{\partial}{\partial w_1} J(w, b) = w_1 - \alpha * \frac{1}{m} \sum_{i=1}^m ((w^t X_i^{(train)} + b) - y_i^{(train)}) * x_1$ 
    ...
     $w_j = w_j - \alpha \frac{\partial}{\partial w_j} J(w, b) = w_j - \alpha * \frac{1}{m} \sum_{i=1}^m ((w^t X_i^{(train)} + b) - y_i^{(train)}) * x_j$ 
    ...
     $w_n = w_n - \alpha \frac{\partial}{\partial w_n} J(w, b) = w_n - \alpha * \frac{1}{m} \sum_{i=1}^m ((w^t X_i^{(train)} + b) - y_i^{(train)}) * x_n$ 
end while

```

Conceptualmente podemos imaginarnos J como una aplicación $J : \mathbb{R}^{n+1} \rightarrow \mathbb{R}$, donde para cada configuración de los n coeficientes de la regresión y el término independiente (de ahí el $n+1$), se obtiene el coste de una determinada configuración de parámetros. Y de acuerdo con esto, α se correspondería con el valor absoluto de la distancia entre las coordenadas en el espacio de entrada de dos configuraciones de iteraciones consecutivas.

Para visualizarlo, podemos tomar la particularización de $n = 1$. En este caso los parámetros a entrenar son 1 coeficiente de la regresión ($|w| = 1$) más su término independiente. De este modo, podemos construir con los valores resultado de aplicar J un espacio tridimensional en el que las configuraciones que mejor aproximan la regresión se corresponden con los puntos más bajos de la superficie generada. Y sabiendo que el vector gradiente tomará los valores de las pendientes en cada punto, podemos imaginarnos la evolución de nuestra configuración de parámetros como un camino que va en descenso a lo largo de esta superficie (de ahí el nombre).

2.2. Regresión Logística

Existen casos en los que nos interesa predecir valores sobre un conjunto discreto, y no aproximar una relación lineal sobre un dominio de valores continuo como sucedía en el caso de la regresión lineal multivariante. Cada uno de los valores de este conjunto se corresponderá con la pertenencia del ejemplo a una clase. De este modo se podrían inferir clasificaciones como por ejemplo si un alumno va a aprobar una futura asignatura a partir de sus anteriores calificaciones, distinguiendo entre dos clases: los aprobados y no aprobados.

La regresión logística es un algoritmo de Aprendizaje Automático cuyo objetivo es la categorización binaria de un ejemplo dada una serie de cualidades del mismo. La idea detrás de esto es predecir valores en un rango $[0, 1]$ y establecer un umbral de clasificación a 0,5. De nuevo tendremos una regresión compuesta de un vector de valores de entrada $x \in \mathbb{R}^n$, un vector de parámetros de la regresión $w \in \mathbb{R}^n$ y su término independiente b , de modo que para los valores resultado de la regresión menores que 0,5, nuestro modelo les asignará la etiqueta 0; y para los mayores de 0,5, les asignará la etiqueta 1. Pero, ¿cómo podemos realizar esto, si nuestra regresión puede devolver valores mayores que 1 y menores que 0?

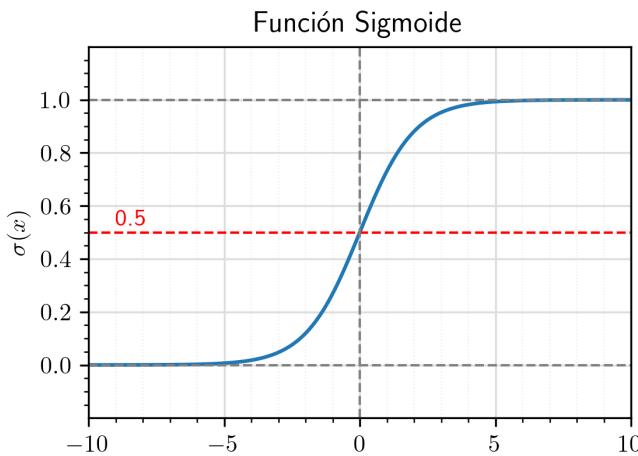


Figura 2.1: Representación de la función Sigmoide $\sigma(x)$.

Es aquí donde entra en juego la función Sigmoide ($\sigma(x)$, a veces referida como función Logística), característica de la Regresión Logística. Dicha función vendrá dada por

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

donde en nuestro caso η será el resultado de la regresión. Pues precisamente, como se puede ver en la Figura 2.1, queremos devolver 1 para los valores mayores que 0,5 y 0 para los valores menores. Dicho esto, podremos calcular el valor de la predicción mediante

$$\hat{y} = \sigma(w^t x + b) = \frac{1}{1 + e^{-(w^t x + b)}}$$

De esta forma, la composición de las funciones nos devolverá valores en $[0, 1]$ y, a la hora de hacer la predicción, se redondeará dicho valor como se ha descrito.

En el caso de la regresión lineal multivariante se pretendía intuir una relación de linealidad, por lo que se empleaba como función de coste el error cuadrático medio ya que este disminuía cuanto mejor se aproximaban los datos de entrenamiento. En este caso la predicción es binaria, por lo que en el peor de los casos, el error máximo por cada ejemplo será 1.

Como consecuencia de esto podemos definir la siguiente función de pérdida para un ejemplo x :

$$\text{Loss}(x, y) = \begin{cases} -\log(\sigma(x)) & \text{si } y = 1, \\ -\log(1 - \sigma(x)) & \text{si } y = 0 \end{cases}$$

Y a partir de esta función de pérdida podemos construir la siguiente función de coste, sobre la que podremos aplicar el mismo algoritmo de gradiente descendente que en la regresión lineal multivariante para entrenar el modelo.

$$\begin{aligned} J(w, b) = & -\frac{1}{m} \left[\sum_{i=1}^m y_i^{(\text{train})} * \log\left(\sigma(w^t X_i^{(\text{train})} + b)\right) \right. \\ & \left. + (1 - y_i^{(\text{train})}) * \log\left(1 - \sigma(w^t X_i^{(\text{train})} + b)\right) \right] \end{aligned}$$

En el comienzo de esta sección se ha mencionado que podríamos querer predecir valores sobre un conjunto discreto, y posteriormente se ha reducido este número de valores a dos: 0 o 1. Sin embargo, es muy común la necesidad de realizar una clasificación multiclas sobre más categorías. Pero, ¿cómo podemos realizar esta clasificación si nuestro modelo solo es capaz de diferenciar entre dos clases? La solución es aplicar regresión logística múltiples veces. Supongamos que queremos realizar una clasificación entre k clases. Para ello, tendremos que entrenar k modelos, donde el i -ésimo modelo ($h^{(i)}$, $1 \leq i \leq k$) será entrenado para distinguir entre dos clases: la conformada por los elementos pertenecientes a la i -ésima clase y los ejemplos pertenecientes a las $k - 1$ clases restantes. A la hora de predecir, catalogaremos un ejemplo como perteneciente a la j -ésima clase tal que $h^{(j)}$ es el máximo resultado de entre los predichos por los k modelos ($1 \leq j \leq k$). Para la ilustración con ejemplos prácticos de ambas, la Regresión Logística y la Regresión Lineal, recomiendo la lectura del Anexo A, donde se muestra el funcionamiento de cada modelo con un ejemplo distinto.

2.3. Redes Neuronales

Las Redes Neuronales son un modelo de Aprendizaje Automático utilizado para la clasificación no lineal de datos y que está inspirado en el comportamiento reactivo de las células del sistema nervioso humano: las neuronas. Las redes neuronales artificiales también han servido para asentar los conocimientos que se tienen sobre el aprendizaje a nivel celular.

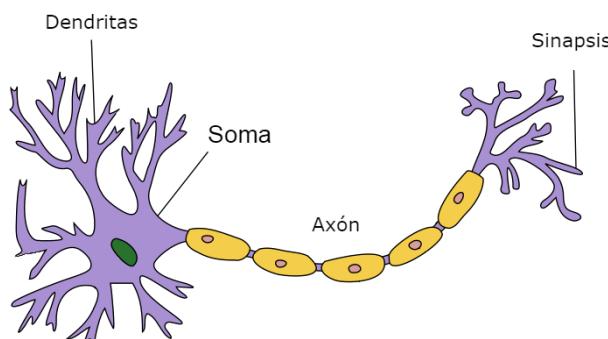


Figura 2.2: Estructura de la neurona¹.

¹Fuente: <https://interactivechaos.com/en/node/2268>

Conceptualmente, las neuronas están conectadas entre sí y se comunican a través de unas terminaciones como las que se pueden ver en la Figura 2.2. Cada neurona recibe una determinada información de las anteriores a través de las dendritas, la procesa, y la comunica a las siguientes por el axón. La computación realizada por estas células es sintetizable mediante interpolaciones matemáticas, que se han deducido mediante la realización de observaciones en el campo de la neurociencia. Es precisamente esta modularidad la que confiere al modelo su inteligencia y versatilidad, lo que las hace superiores a modelos lineales como los vistos previamente y que ha propiciado el resurgimiento en popularidad de la aplicación de esta técnica.

El objetivo de estos modelos es aproximar una función de clasificación no lineal cualquiera ($y = f^*(x)$) mediante generalización estadística. Como podemos ver en la Figura 2.3, es muy común la representación de estas redes como grafos dirigidos acíclicos, donde las neuronas están dispuestas en hilera (que llamaremos capas) y donde las aristas nos indican las conexiones entre neuronas.

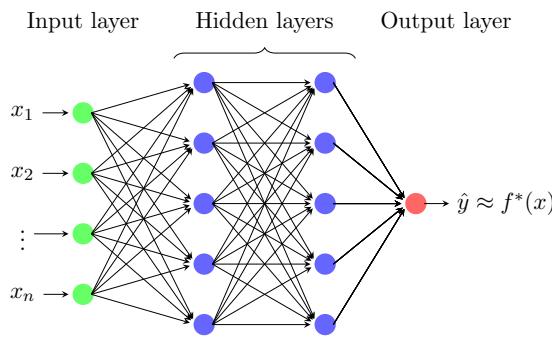


Figura 2.3: Estructura de una red neuronal artificial completamente conexa de cuatro capas.

El término de red, trasladado al campo de las matemáticas, implica que la función producida es fruto de la composición de las distintas funciones asociadas a sus capas. De esta forma, podríamos decir que, para el ejemplo de la Figura 2.3, nuestra predicción \hat{y} vendría dada por la composición de las funciones $f^{(1)}, f^{(2)}, f^{(3)}$ de la siguiente forma:

$$\hat{y} = f(x) = f^{(3)}(f^{(2)}(f^{(1)}(x))),$$

donde cada función se corresponde con cada una de las capas de la red en ese orden. Llegados a este punto, hay que mencionar que la capa de entrada es puramente simbólica y por tanto no realizará ninguna computación con dichos datos. Debido a esto, nuestra función $f(x)$ vendrá dada por la composición de $k - 1$ funciones si hay un total de k capas.

El número de capas de la red nos indicará su “profundidad” y el número de las neuronas que componen cada capa (o la aridad de sus correspondientes funciones) determinará lo que llamaremos su “anchura”. Cada una de las capas de la red puede ser interpretada como una función vector-a-vector ya que toma un vector de valores (los de la anterior capa) como entrada, y devuelve otro vector (que se compone de los resultados de cada unidad) como salida. Pero quizás esto no sea lo más correcto, pues estamos hablando de redes neuronales completamente conexas y la filosofía detrás de este modelo es la aproximación de una función a partir de muchas células independientes que son capaces de realizar computaciones de forma aislada. Es por esto que es más conveniente conceptualizar cada capa como una serie de unidades independientes que realizan una función vector-a-escalar de forma paralela.

Al igual que en los modelos anteriores, la computación realizada por la composición de las funciones $f^{(n)}$ ($0 < n < k$) vendrá determinada por una serie de parámetros que habremos

de perfeccionar en el entrenamiento. Recordemos que en los modelos de regresión teníamos para esto un vector w , que nos aportaba los pesos de los distintos términos de la regresión, y su término independiente, dado por la variable b . En dichos modelos la predicción venía dada por una sola función, pero este no es el caso ahora y por ello necesitamos extender esa notación. Emplearemos $W^{(j)}$ y $B^{(j)}$ para denotar la matriz de pesos y el vector de términos independientes entre el nivel j y $j + 1$ ($0 \leq j < k - 1$). Por lo que ahora la función de nuestra red de la Figura 2.3 vendrá dada por:

$$f(x; W, B) = f^{(3)}(f^{(2)}(f^{(1)}(x; W^{(0)}, B^{(0)}); W^{(1)}, B^{(1)}); W^{(2)}, B^{(2)}).$$

Hay que introducir ahora, el concepto de función de activación. La función de activación es un concepto inherente a cada una de las neuronas de forma individual. Para la i -ésima neurona contenida en la j -ésima capa de la red, su valor de activación será el resultado de su computación de acuerdo a los valores de pesos y término independiente correspondientes. Dicho valor vendrá dado por:

$$a_i^{(j)} = f^{(j)}((W^{(j-1)})^t f^{(j-1)}(x; W, B) + B_i^{(j-1)}),$$

donde $0 \leq j < k - 1$. Llegados a este punto hay que apuntar que entendemos por $f^{(0)}$ la función identidad, pues esta función se correspondería con la capa de input que como ya se ha dicho, no aplica ninguna transformación a estos datos. Veremos algún ejemplo de función de activación más adelante.

Una vez se ha introducido la notación para referirse a los parámetros de la red, hay que diseñar una función de coste que nos permita evaluar la calidad de una determinada inicialización de los mismos, y a partir de ahí procurar entrenar la red mejorando la calidad de estas soluciones. Recuperaremos para esto un par de nociones de la regresión logística. Si recordamos lo expuesto al final de dicha sección (2.2), recordaremos que una de sus grandes limitaciones era que solo podíamos clasificar entre dos clases por cada instancia del modelo. Las redes neuronales sobreponen esa limitación. Pues nos permiten la posibilidad de tener k neuronas en la capa de salida y, habiendo aplicado alguna función (como podría ser la sigmoide) para tener los valores de salida en un rango $[0, 1]$, predecir que clasificamos un determinado ejemplo como perteneciente a la clase i -ésima siendo el i -ésimo valor de salida el máximo de entre los k valores. La técnica que se aplica sobre las etiquetas del *dataset* para convertirlas en distribuciones de probabilidad sobre k clases (y así usarlas como valores objetivo en el entrenamiento) es conocida como *One-Hot Encoding*, y hablaremos de ella con más profundidad en posteriores secciones (3.2.2). Dicho esto, ahora el error de predicción será el acumulado entre múltiples salidas del modelo. Y haciendo un par de modificaciones a la función de coste que utilizamos en la regresión logística obtenemos la siguiente:

$$\begin{aligned} J(W, B) = -\frac{1}{m} \Big[& \sum_{i=1}^m \sum_{j=1}^k y_{ij} * \log(f(X_i^{(\text{train})}; W, B)_j) \\ & + (1 - y_{ij}) * \log(1 - f(X_i^{(\text{train})}; W, B)_j) \Big], \end{aligned}$$

donde $m = |X|$, $f(x; X, B)_j$ denota el valor de la j -ésima neurona en la capa de salida del modelo e y_{ij} indica el j -ésimo valor objetivo del i -ésimo ejemplo (para una etiqueta correspondiente a la k -ésima clase y_{ij} será 1 si $j = k$ o 0 en caso contrario). Siguiendo la metodología que hemos aplicado al resto de modelos, ahora procuraremos entrenar el modelo mediante el algoritmo del gradiente descendente. Cuando se habla de redes neuronales, el estándar actual para el cálculo del gradiente es emplear a su vez otro algoritmo llamado *back-*

propagation. *Back-propagation* es un algoritmo complejo y cuyos fundamentos trascienden el ámbito de este documento, por lo que no detallaremos su funcionamiento. Para conocer estos detalles recomiendo la lectura de la Sección 5 del Capítulo 6 de Goodfellow y col. (2016).

2.3.1. Optimizaciones para el entrenamiento

Llegados a este punto del documento, el lector se habrá familiarizado con la manera en la cual el algoritmo del gradiente descendente se emplea para la fase de entrenamiento. Este algoritmo consiste en modificar los parámetros de la red de acuerdo con el valor opuesto del vector gradiente de la función de coste y un metaparámetro llamado “ratio de aprendizaje” (que previamente hemos denotado con α) durante un número de iteraciones. Dicha función de coste se computaba sobre todos los ejemplos de entrenamiento antes de actualizar los parámetros en una iteración. Esto era viable cuando nuestros modelos eran más sencillos, como es el caso de la regresión que hemos visto previamente. Sin embargo, tal aproximación al cómputo del gradiente se hace impracticable cuando tratamos con modelos compuestos de muchas unidades independientes debido al *overhead* que esto implica. Este es el caso de las redes neuronales complejas (con muchas capas y/o neuronas por capa) y por ello se han desarrollado variantes del algoritmo original, así como estrategias de optimización que pretenden acelerar la convergencia de los parámetros que componen el modelo. En esta sección pretendemos comentar los rudimentos de lo más importante de ambas categorías.

Variantes del gradiente descendente

Las variantes más famosas del gradiente descendente (y cuyo uso es más extendido) son tres, cada una mejorando la anterior. Cada una presenta un determinado grado de adecuación entre la precisión de cada actualización de los parámetros del modelo en una iteración del algoritmo, y el tiempo que requiere la convergencia de los mismos (Ruder, 2016). A partir de ahora se empleará toda una nueva terminología relativa a la forma de manipular y extraer los datos del conjunto de entrenamiento. Los dos primeros nuevos términos serán: *epoch* (“época”), que se refiere a una iteración completa sobre todos los elementos del *dataset* (lo que en la versión tradicional del gradiente descendente era una iteración normal); y *batch* (“lote”), que será cada uno de los subconjuntos que extraeremos del conjunto de entrenamiento y cuya cantidad a su vez vendrá determinada por el tamaño de cada *batch*.

La primera versión que veremos es el *batch gradient descent*, que se corresponde con la versión estándar del gradiente que ya hemos descrito. Cada actualización de los parámetros requiere del cómputo del coste sobre todos los ejemplos. De acuerdo a la nueva terminología, el tamaño del *batch* será el tamaño del *dataset* completo y el número de *epochs* será el número de iteraciones del algoritmo. Cuando nuestro *dataset* es pequeño, esto puede considerarse, pero cuando tenemos un *dataset* muy grande (como más adelante veremos que es el caso), las limitaciones de memoria nos lo impiden. Sin embargo, esta versión del algoritmo garantiza la convergencia al mínimo global en superficies de error convexas, y al mínimo local en superficies no convexas.

El gradiente descendente estocástico (SGD, *Stochastic Gradient Descendent*) es la segunda variante, y quizás es la variante cuyo nombre ha sido más resonado. El SGD modifica los parámetros de acuerdo al valor inverso del gradiente de la función de coste, pero su diferencia radica en que esta función de coste solo evaluará uno de los ejemplos del *dataset* (por lo que el tamaño del *batch* será 1). El hecho de que cada ejemplo del *dataset* sea evaluado individualmente hace que la evolución de los coeficientes sea mucho más errática y, por ende, con mucha más desviación estándar. Esto le permite converger a mejores mínimos locales y más rápido. Sin embargo, tal fluctuación puede tener efectos indeseados y puede

mitigarse disminuyendo el ratio de aprendizaje. Pero en estos casos el SGD muestra el mismo comportamiento que el gradiente descendente estándar.

La última variante del gradiente descendente que veremos recibe el nombre de *mini-batch gradient descendant*. Esta variante toma lo mejor de las dos anteriores procurando sobreponer las limitaciones de ambas, y por ello es la más usada. Su funcionamiento consiste en la realización de una actualización del gradiente por cada uno de los *batches* que extraeremos del conjunto de entrenamiento (el tamaño de cada *batch* suele ir de 50 a 256). De este modo se reduce la varianza de las actualizaciones de los parámetros y se estabiliza la convergencia. Es muy común encontrarse en la bibliografía referencias a este algoritmo bajo el nombre de SGD, aunque en realidad se refieran al uso de *mini-batches*.

Adam

Adaptive Moment Estimation (“Adam”, Kingma y Ba (2014)) es la estrategia de optimización del gradiente descendente más extensamente utilizada en la actualidad y la empleada para el entrenamiento del modelo implementado en este proyecto. Adam es uno de toda una línea evolutiva de algoritmos de optimización. La diferencia más radical de este algoritmo con respecto al funcionamiento del gradiente descendente “básico” es que tiene un ratio de aprendizaje para cada uno de los parámetros de la red y modifica sus valores de forma adaptativa en tiempo de entrenamiento. La justificación para la presencia de todos estos ratios de aprendizaje es que cada parámetro va a afectar con mayor o menor importancia al cómputo final del modelo. Por esto, la evolución para cada uno de estos tendrá una varianza distinta y, por tanto, evaluarlos todos bajo el mismo factor no tiene sentido (Ruder, 2016). Para modificar estos ratios dinámicamente, Adam incorpora como novedad una media exponencialmente decreciente de los valores anteriores de los distintos gradientes y hereda elementos como la media exponencialmente decreciente de los anteriores gradientes al cuadrado de algoritmos anteriores como “Adadelta” (Zeiler, 2012) o “RMSprop” (Hinton y col., 2012). Mediante estos mecanismos, Adam acelera la velocidad de convergencia de los distintos parámetros de la red y además hace que cada actualización de los parámetros sea menos errática (como era el caso del SGD).

2.4. Redes Neuronales Convolucionales

Las Redes Neuronales Convolucionales (CNN, *Convolutional Neural Networks*) son un tipo de red neuronal especializadas en el procesado de datos que, debido a su naturaleza, pueden ser representados de forma matricial (Goodfellow y col., 2016). Su funcionamiento permite a estas redes detectar patrones en un determinado input abstrayendo la localidad, independientemente de la región de los datos en los que se encuentren. El ejemplo de aplicación más claro son las imágenes, pues inmediatamente se pueden contemplar como una matriz bidimensional de colores; pero también existen otros casos de uso no tan evidentes (como pueden ser las series temporales de datos, que se pueden imaginar como una matriz de una sola fila). Su característica fundamental, las convoluciones, se refieren a los fundamentos formales que hay detrás de este modelo, que precisamente se sirven del operador de “convolución”.

La convolución es una operación matemática que toma como entrada dos funciones que llamaremos f y g , y nos devuelve un valor que mide la superposición de f y una versión trasladada e invertida de g . Para denotarlo se suele emplear el operador $*$ y su resultado viene dado por:

$$c(\epsilon) = (f * g)(\epsilon) = \int f(x)g(\epsilon - x)dx.$$

En el mundo de las redes neuronales existe una determinada terminología cuando hablamos de convoluciones. Por convenio, nos referimos a la primera función de la convolución (en este caso f), como la “entrada”; y a la segunda (en este caso g), como el *kernel* (“núcleo”) de la convolución (como las mostradas en la Figura 2.4). También es habitual referirse al resultado de la convolución como *feature map* y a las dimensiones de la matriz del *kernel* como su tamaño (*kernel size*). Llegados a este punto hay que decir que, dado que no se trabaja con un dominio infinito, habitualmente se representarán los dominios de las funciones sobre las que aplicaremos la convolución con vectores o matrices (representando sus valores para una serie de puntos).

En este tipo específico de redes, la aplicación de las convoluciones se realiza en las llamadas “capas convolucionales” (*convolutional layers*). En el caso de las convoluciones bidimensionales, tanto los datos de entrada como el *kernel* serán matrices. La matriz resultado se construirá a partir de valores que se obtienen realizando la multiplicación elemento a elemento del *kernel* con todas las posibles “submatrices” de elementos adyacentes que podamos extraer de los datos de entrada. Hay que mencionar que, dado que los valores resultado de aplicar los *kernels* se sitúan en torno a los valores centrales de la matriz (como se puede ver en la Figura 2.5), es muy común la realización de una técnica llamada *padding*, que consiste en añadir ceros en los bordes de la matriz de entrada para poder aplicar los *kernels* también sobre estos valores límite.

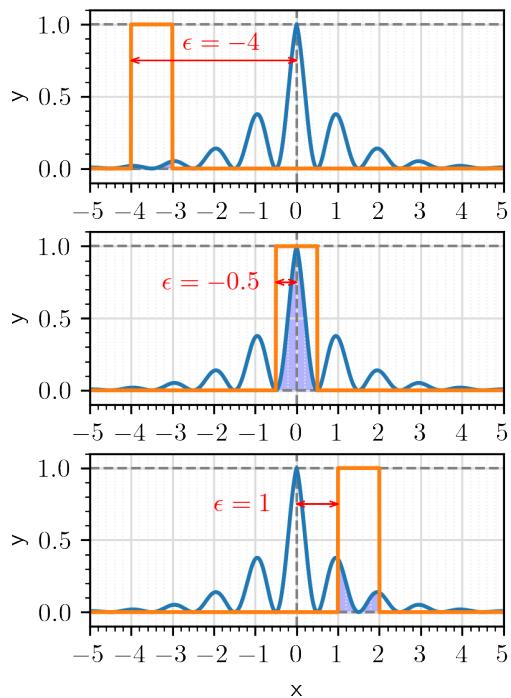


Figura 2.4: Ejemplo de la convolución entre dos funciones $(f(x) = \frac{1+\cos(2\pi x)}{2e^{|3x/4|}}$ y $g(x) = 1$ si $0 \leq x \leq 1$, o 0 en caso contrario) para distintos valores de ϵ .

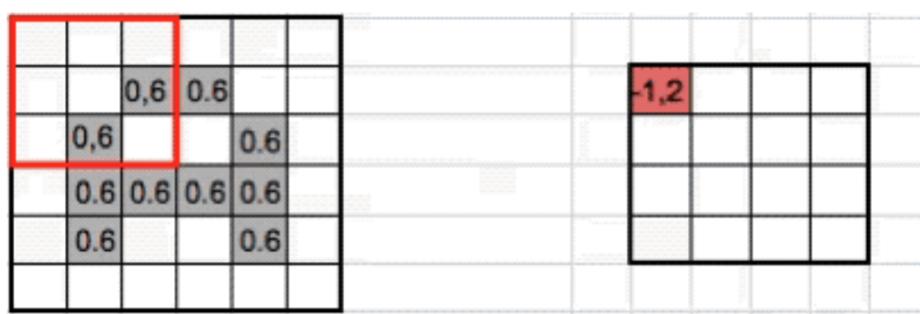


Figura 2.5: Multiplicación de los valores de entrada de una capa convolucional 6×6 con un kernel 3×3^2 .

La distribución de los distintos valores de los coeficientes del *kernel* harán que cada uno fomente unas características correspondientes a una serie de patrones distintos en la imagen original. Es por esto que es muy común la aplicación de varios *kernels* en una misma capa convolucional. Por lo que si para una hipotética capa convolucional los datos de input tienen

²Fuente <https://www.juanbarrios.com/redes-neurales-convolucionales/>.

dimensiones ($i_1 \times i_2$), los *kernels* tienen dimensiones ($k_1 \times k_2$) y n es el número de *kernels* a aplicar sobre dicho input, el output de dicha capa tendrá dimensiones $(i_1 - k_1 + 1) \times (i_2 - k_2 + 1) \times n$.

Dado que la tarea que queremos realizar mediante estos modelos es, en última instancia, la de clasificar un determinado ejemplo, las redes neuronales convolucionales suelen tener al final de su arquitectura una serie de capas completamente conexas para realizar precisamente esta clasificación. De modo que, mediante las convoluciones se explotan las características de la imagen, y mediante estas capas completamente conexas se realiza finalmente la clasificación.

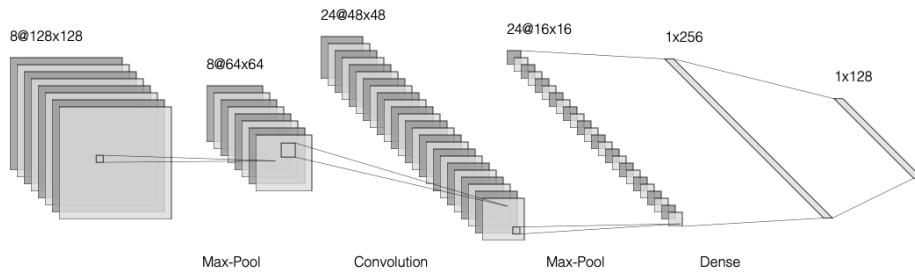


Figura 2.6: Arquitectura completa de una red neuronal convolucional³.

Para concluir esta sección se ha de mencionar que en el caso que nos ocupa las convoluciones que aplicaremos serán unidimensionales, pues esta es la condición de los datos con los que se trabaja. Su funcionamiento es análogo a las explicadas, solo que en vez de trabajar con matrices se emplearán vectores. La motivación detrás de utilizar como ejemplo estas convoluciones bidimensionales sobre las unidimensionales es que resultan mucho más visuales, además de que es precisamente sobre datos bidimensionales (imágenes, principalmente) donde la superioridad de estas redes se acentúa. Esto ha hecho que su uso sea prácticamente obligatorio cuando tratamos con imágenes y los resultados que consiguen son de nivel *state of the art*. También hay que decir que en las redes convolucionales es común el uso de otras capas características de estas arquitecturas (como las de *Max-Pool*) que se pueden ver en la Figura 2.6 pero en el caso que nos ocupa, por limitaciones de espacio y porque el modelo en el que nos inspiramos no las usa, no se explicarán.

³Figura obtenida mediante la herramienta <http://alexlenail.me/NN-SVG/LeNet.html>.

CAPÍTULO 3

WaveNet

Es momento ahora de presentar *WaveNet*, el modelo que ha servido como base para llevar a cabo este proyecto. *WaveNet* es una “Red Neuronal Profunda” (*Deep Neural Network*) descrita por primera vez en Van Den Oord y col. (2016), y que ha abierto toda una nueva línea de investigación de artículos que orbitan alrededor de este trabajo con variantes de su arquitectura.

WaveNet ha sido desarrollada por un equipo de investigadores de la empresa inglesa DeepMind, dedicada a la investigación en el campo de la inteligencia artificial y que es propiedad de Google. El modelo produce sus resultados más satisfactorios cuando se aplica a problemas de *Text-to-Speech*. Por esto, *WaveNet* es actualmente la encargada de generar las diferentes voces del asistente de voz de Google, que está incorporado en la práctica totalidad de los teléfonos con Android.

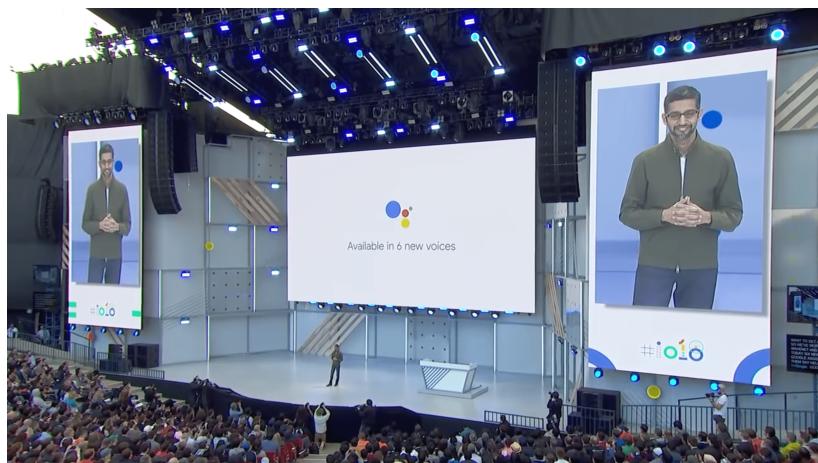


Figura 3.1: Presentación de *WaveNet* como el motor detrás de las voces del asistente de voz de Google en la Google I/O 2018¹.

Las dos características fundamentales de la arquitectura de *WaveNet* es que es un modelo autorregresivo y probabilista. A continuación se explica qué implicaciones tienen ambas características. La autorregresión del modelo es algo inherente a las características del problema. Entendemos por autorregresivos a aquellos modelos que trabajan sobre datos que tienen algún tipo de dependencia temporal entre ellos, es decir, se basan en una serie de ejemplos para predecir los siguientes y posteriormente emplea esas predicciones para inferir los valores consecutivos.

¹Fuente: <https://www.youtube.com/watch?v=JjK8apEishQ&t=71s>.

Este es el caso de los datos sobre los que se trabaja: el audio. Pues, como se puede ver en la Figura 3.2, la forma de onda de un determinado sonido (aunque puede ir variando) tiene una determinada periodicidad que se puede intuir a partir de los ejemplos anteriores. De este modo, la probabilidad conjunta de una onda $X = \{x_1, \dots, x_T\}$ se puede calcular como el producto de las probabilidades condicionales de este modo:

$$p(X) = \prod_{t=1}^T p(x_t|x_1, \dots, x_{t-1}).$$

Por lo que cada muestra x_t está condicionada por todas las anteriores. En el caso de nuestro modelo, esta probabilidad condicionada se deduce mediante una serie de capas convolucionales y se traduce en una distribución categórica mediante una capa *Softmax*.

Aprovechamos ahora para, mediante la Figura 3.3, presentar la arquitectura original de *WaveNet*. En dicha figura se pueden apreciar toda una variedad de elementos muy característicos de este modelo. Las siguientes secciones se dedicarán a explicar todos los entresijos de esta arquitectura.

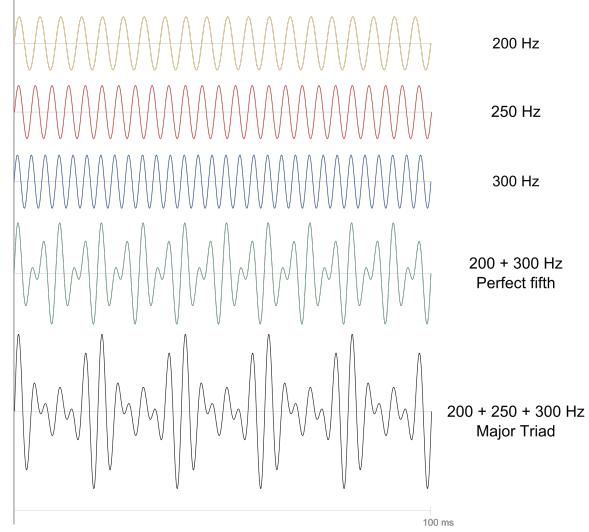


Figura 3.2: Composición de un acorde de La Mayor como suma de ondas².

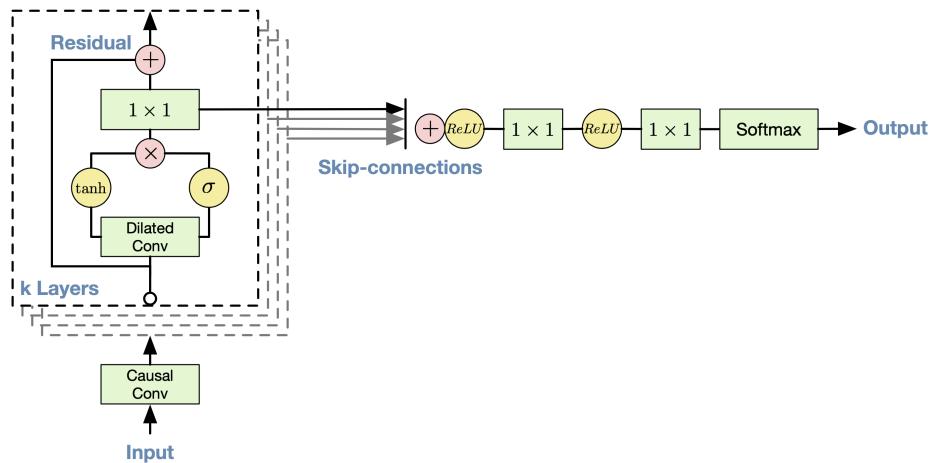


Figura 3.3: Esquema de la arquitectura del modelo³.

3.1. Capas convolucionales

3.1.1. Convoluciones Causales

Cuando trabajamos con modelos autorregresivos (como es el caso), el empleo de “convoluciones causales” (*causal convolutions*) es necesario. Este tipo de convolución está pensado para el procesado de datos secuenciales o de naturaleza temporal.

²Fuente: <https://es.wikipedia.org/wiki/Armon%C3%ADA>

³Fuente: Van Den Oord y col. (2016)

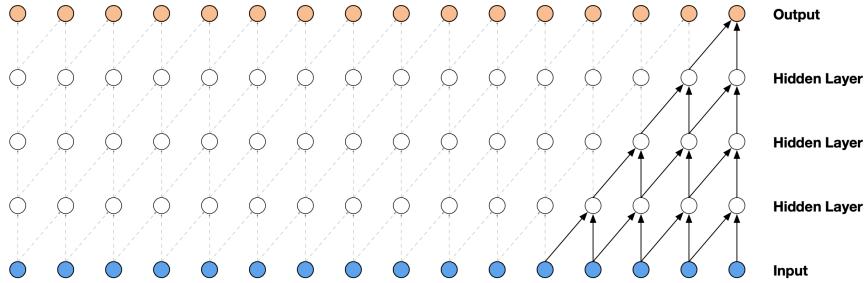


Figura 3.4: Representación de un *stack* de *causal convolutional layers*⁴.

A diferencia de las convoluciones tradicionales presentadas en la Sección 2.4, estas nos aseguran que el orden de los datos de entrada permanece inmutable. Así se garantiza de que la predicción $p(x_t|x_1, \dots, x_{t-1})$ para un instante $t - 1$ no dependa de valores futuros. Esto es apreciable en la Figura 3.4 en la forma en la que están conectadas las distintas neuronas y en el *padding* (“alineamiento”) que la red realiza con estas conexiones hacia la derecha. El mayor inconveniente de este tipo de capas es su “campo receptivo”, es decir, la porción de datos que toman como entrada. En el caso del ejemplo el campo receptivo sería 5 (número de capas + longitud del filtro – 1 = 4 + 2 – 1). Las soluciones inmediatas para incrementar este campo receptivo son dos: incrementar el número de capas o utilizar filtros más grandes. Pero cualquiera de estas dos soluciones tiene un inconveniente: el incremento de parámetros de entrenamiento de la red, que a su vez se traduce en un mayor coste computacional requerido para el entrenamiento del modelo. Es por eso que *WaveNet* utiliza una técnica ligeramente más sofisticada y eficiente: las “convoluciones causales dilatadas”.

3.1.2. Convoluciones Causales Dilatadas

Las convoluciones causales dilatadas (*dilated causal convolutions*) son un tipo de convolución causal cuya característica fundamental es el la aplicación de los filtros “saltándose” parámetros pero manteniéndose la longitud del filtro. Esto nos permite obtener un rango receptivo mayor manteniendo el mismo número de capas, algo que implica una mayor eficiencia computacional.

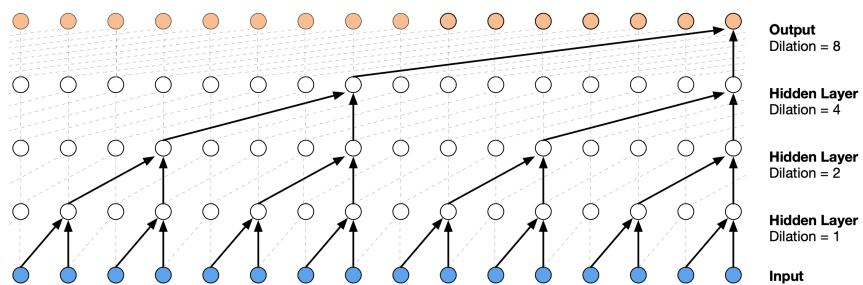


Figura 3.5: Representación de un *stack* de *dilated causal convolutional layers*⁵.

El número de parámetros que se saltan los filtros de una determinada capa responde a un parámetro llamado *dilation rate*. En el caso de la Figura 3.5, los valores de este parámetro para cada capa serían: 1, 2, 4 y 8. Esta técnica es parecida a otras como las *pooling convolutions* o *strided convolutions*, pero con la diferencia de que, en este caso, la cardinalidad de los datos se mantiene a lo largo de toda la red.

⁴Fuente: Van Den Oord y col. (2016)

⁵Fuente: Van Den Oord y col. (2016)

3.2. Capa de salida

Como se ha dicho anteriormente, las capas convolucionales son las encargadas de procesar cada secuencia de input y , mediante una capa *Softmax*, modelamos una distribución de probabilidad $p(x_t|x_1, \dots, x_{t-1})$ para cada muestra de audio que vaya a predecir el modelo. Los desarrolladores de *WaveNet* se decantaron por emplear distribuciones de probabilidad *Softmax* como output del modelo basándose en trabajos previos, pues ellos mismos dicen en Van Den Oord y col. (2016) que han demostrado ser más flexibles que otras soluciones incluso cuando los datos con los que trabajamos son de naturaleza continua.

Las capas *Softmax* tienen como objetivo modelar distribuciones categóricas entre k clases. Toman como input un vector v de k elementos y devuelven otro vector v' de igual longitud, donde el valor del i -ésimo elemento denota la probabilidad normalizada $v'_i \in [0, 1]$ ($\forall i, 1 \leq i \leq k$) de pertenencia del ejemplo a la i -ésima clase. La función que computa el valor de cada elemento viene dada por:

$$S(v) = \left(\frac{e^{v_1}}{\sum_j e^{v_j}}, \frac{e^{v_2}}{\sum_j e^{v_j}}, \dots, \frac{e^{v_k}}{\sum_j e^{v_j}} \right).$$

En este caso se trabaja con ficheros de audio que representan los valores de amplitud de la onda mediante secuencias de enteros de 16 bits que enumeran los distintos niveles de onda en un rango de -1 a 1 de forma equidistante. Por lo que la onda es susceptible de tomar 65536 (2^{16}) valores de amplitud distintos. De acuerdo a lo que hemos dicho antes, esto implicaría que la capa *Softmax* que tenemos como salida debería estar compuesta por 2^{16} elementos para diferenciar cada uno de estos niveles de amplitud. Como consecuencia de esto, el número de parámetros de la red incrementa y con él también lo hace la dificultad de entrenar un modelo que sea capaz de discernir entre 2^{16} clases. Por lo que tales niveles de precisión resultan siendo inconvenientes. Debido a esto, hay que aplicar algún tipo de compresión. En este proyecto se emplea el μ -law companding (descrito en la siguiente sección) pues así se realizó en el trabajo de referencia (Van Den Oord y col., 2016).

3.2.1. μ -law companding

El μ -law companding es un algoritmo de compresión descrito en el estándar G.711 (Recommendation, 1988) por la ITU-T (*International Telecommunication Union - Telecommunication Standardization Sector*) y utilizado en las telecomunicaciones mediante modulación por impulsos codificados (PCM, *Pulse-Code Modulation*) para la transmisión de voz. PCM está referido al uso de secuencias de bits para la representación de señales analógicas y, por tanto, su transformación en señales digitales. PCM es el estándar hoy en día.

Antes de indagar en el funcionamiento de la μ -law, hay que dejar claros cuáles son los objetivos que se pretenden alcanzar mediante su uso. Se quiere comprimir los valores de onda ($[-1, 1]$) que inicialmente están codificados como números enteros de 16 bits para, discretizando este rango, facilitar el entrenamiento del modelo. Pues como ya se ha explicado, el número de valores que puede tomar una onda (que se traduce en el número de clases a las que puede pertenecer un ejemplo) va a condicionar la dificultad del entrenamiento de un hipotético modelo hasta hacerlo impracticable. Por eso, la compresión de la μ -law se divide en dos etapas: la transformación y la cuantización. Antes de comenzar hay que decir que cuando se hable de los distintos valores de onda, nos referiremos a la amplitud en términos absolutos, llamando a los valores más cercanos al 0 valores de menor amplitud, mientras que los cercanos a los límites (-1 o 1) serán los valores de mayor amplitud.

La transformación es la primera parte del algoritmo. Consiste en pasar los valores de amplitud de la onda “en crudo” a una escala logarítmica como la mostrada en la Figura 3.6. Es en esta etapa donde se involucra el parámetro μ de donde el algoritmo obtiene su nombre. Como se puede ver en la citada figura, esta transformación sigue manteniendo los valores en un rango $[-1, 1]$, aunque los redistribuye. Los valores de menor amplitud se ven más favorecidos mientras que los de los valores límite se “acumulan” dentro de este rango. Podemos ver que para un valor $\mu = 1$ la transformación apenas modifica estos valores, pero conforme aumentamos μ la curva se acentúa y así lo hace la transformación. La transformación y su función inversa vienen dadas por:

$$F(x) = \text{sign}(x) \frac{\ln(1 + \mu|x|)}{\ln(1 + \mu)}, \quad F^{-1}(y) = \text{sign}(y) \frac{(1 + \mu)^{|y|} - 1}{\mu}.$$

La cuantización es la parte del algoritmo donde realmente se aplica la compresión. El objetivo en este segundo paso es dividir el rango de valores a los que se ha aplicado la transformación ($[-1, 1]$) en k niveles equidistantes, y aproximar cada valor de onda a su nivel más cercano. De este modo, podremos enumerar cada uno de los niveles en los que hemos dividido el rango con un número entero $j \in \{0, 1, \dots, k - 1\}$ que podremos representar con $\lceil \log_2(k) \rceil$ bits, y que finalmente representará la codificación de un determinado valor de onda. La obtención de este número j a partir de un valor de amplitud y y el proceso inverso vienen dados por:

$$j = \left\lfloor \frac{(k - 1) * (y + \Delta)}{2\Delta} \right\rfloor, \quad y = \frac{2\Delta * j}{k - 1} - \Delta$$

donde Δ indica el valor de la amplitud. Suponemos para la escritura de estas expresiones que, como en el caso que nos ocupa, los datos están centrados en torno al 0.

A priori podría parecer que la transformación que hemos aplicado es innecesaria, pues en la cuantización ya hemos dividido el rango de valores en distintos niveles equidistantes y por tanto hemos minimizado el error medio derivado de la compresión de forma uniforme en todos los puntos del dominio. Esto se cumple si asumimos una distribución equiprobable de los puntos del espacio. Pero esto no es así. La voz humana (el audio para cuya compresión está pensada la μ -law), por sus características tímbricas, tiende a hacer que sean más comunes los niveles de onda con menor amplitud. Es por esto que se aplica la transformación que hemos descrito (Figura 3.6) sobre los valores de la onda original, tras la que los valores de amplitud más pequeños son favorecidos. Y al aplicar la cuantización sobre estos datos transformados, los valores de la onda con menor amplitud (y como se decía anteriormente más probables) se pueden diferenciar mejor, minimizando así el error promedio derivado de la compresión.

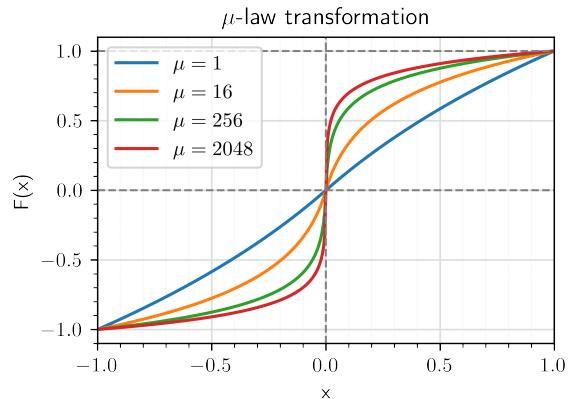


Figura 3.6: Función aplicada para la codificación según la μ -law para distintos valores de μ .

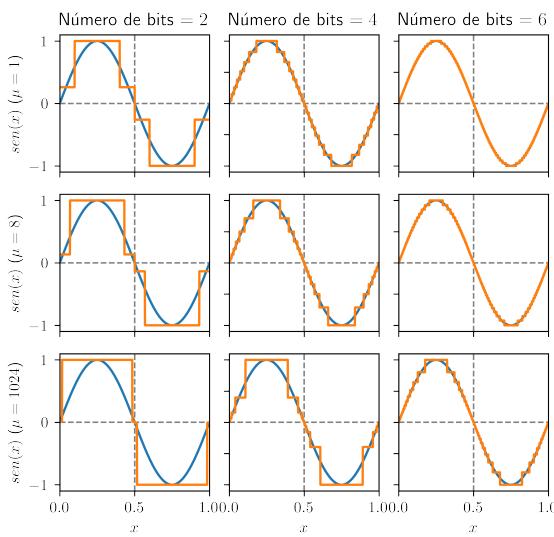


Figura 3.7: Ejemplo de reconstrucción de una onda sinusoidal con distintos valores de μ y con distintos valores de k dados por 2^{n^o} de bits.

Esto también es apreciable en la Figura 3.7, aunque dado que los datos que se muestran han sido reconstruidos después de la compresión quizás este fenómeno se aprecie mejor si se entiende de la forma inversa. En esta ocasión, lo que se puede apreciar como fruto de la transformación es cómo estos niveles de señal (en los que dividíamos el dominio en la cuantización) se acumulan alrededor de los valores de menor amplitud conforme aumentamos el valor de μ debido al paso a escala logarítmica de los datos. En el caso de la Figura 3.8, lo que se aproxima es un fragmento de audio de uno de los ficheros del *dataset*. Este caso sería más cercano a un caso de uso real, pues esta onda se corresponde con un audio de una grabación. En dicha figura también podemos apreciar cómo el error derivado de la compresión es mínimo con esta configuración de parámetros.

En el caso que nos ocupa, y al igual que dijeron los desarrolladores de *WaveNet* en su especificación original, se empleará un valor de $\mu = 256$. En dicho trabajo también tomaron la decisión de diseño de emplear 8 bits (256 posibles niveles distintos) para la codificación de la señal, pero no se debe dejar dejar al azar ese parámetro y por ello en posteriores secciones se explorará cómo determinarlo.

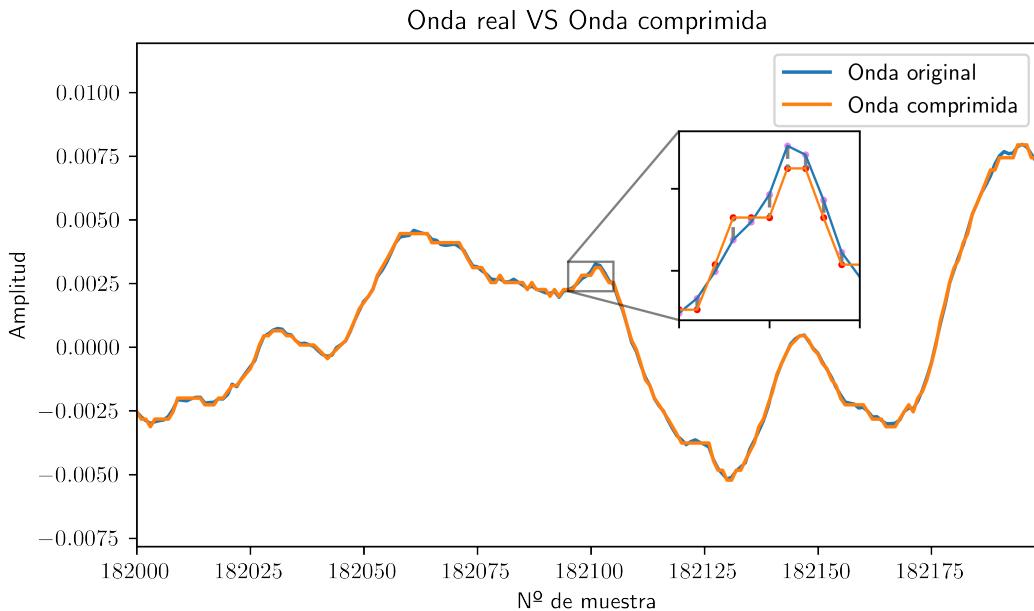


Figura 3.8: Valores de onda original y comprimida empleando el algoritmo descrito con un valor $\mu = 256$ y una división en 256 niveles (8 bits) distintos para un fichero del *dataset*⁶.

⁶Fichero: 2018/MIDI-Unprocessed_ Chamber3_ MID--AUDIO_ 10_ R3_ 2018_ wav--1.wav.

3.2.2. One-Hot encoding

Es momento ahora de introducir el concepto de *One-Hot encoding*, una técnica ampliamente utilizada para tareas de clasificación multiclas. Para ello recordaremos lo dicho en la Sección 3.2, donde teníamos un vector v' como output de la red y cada elemento v'_i representaba la probabilidad de pertenencia del ejemplo que se está clasificando a la i -ésima clase. Para poder entrenar las distribuciones de probabilidad que nos devolverá la red necesitamos algún tipo de encoding que maximice la probabilidad de pertenencia a la i -ésima clase (para un valor objetivo y que represente la i -ésima clase) y minimice el resto. Tomemos como referencia el siguiente ejemplo:

X						y
$X_{0,0}$...	$X_{0,j}$...	$X_{0,n}$	Amarillo	
:	..	:	..	:		
$X_{i,0}$...	$X_{i,j}$...	$X_{i,n}$	Rojo	
:	..	:	..	:		
$X_{m,0}$...	$X_{m,j}$...	$X_{m,n}$	Verde	

→

Clase	ID
Rojo	0
Amarillo	1
Verde	2

X					$One-Hot$			y'
					0	1	2	
$X_{0,0}$...	$X_{0,j}$...	$X_{0,n}$	0	1	0	[0, 1, 0]
:	..	:	..	:	:	:	:	:
$X_{i,0}$...	$X_{i,j}$...	$X_{i,n}$	1	0	0	[1, 0, 0]
:	..	:	..	:	:	:	:	:
$X_{m,0}$...	$X_{m,j}$...	$X_{m,n}$	0	0	1	[0, 0, 1]

Figura 3.9: Ejemplo de aplicación de *One-Hot encoding* sobre un supuesto *dataset* de clasificación por colores.

En este ejemplo, como en la mayoría de *datasets* para la clasificación mediante modelos supervisados, el atributo referido a la clase del ejemplo (columna y en este caso) viene dado por una serie de etiquetas en forma de *strings*. Es por esto que para realizar la clasificación necesitamos encontrar una biyección con $\{0, 1, \dots, k - 1\}$ para k clases, como la especificada en la tabla del ejemplo. Y después de esto construiremos para cada ejemplo y_i un vector en el que el elemento i -ésimo (fruto de esta biyección) tendrá valor 1 y el resto 0. De este modo construiremos y' , que se convertirá en la lista de valores objetivo de nuestra red para las k neuronas de salida. Estos valores objetivo cumplen la condición que buscábamos, pues maximizan la pertenencia a la i -ésima clase en cada caso correspondiente.

En el caso de *WaveNet* aplicamos este encoding a los distintos niveles de audio descritos en la anterior Sección 3.2.1, y así tratamos los distintos niveles de audio como k clases. Por lo que para un instante t y una ventana de tiempo τ podemos predecir el próximo nivel de onda (el nivel con más probabilidad) como

$$\arg \max_{0 \leq n \leq k-1} p(x_{t,n} | x_{t-\tau}, \dots, x_{t-2}, x_{t-1}),$$

donde $x_{i,j}$ denota el valor del j -ésimo elemento de la distribución de probabilidad para el i -ésimo instante. De modo que j será el índice asociado a la clase que maximice la probabilidad condicionada con respecto de los τ anteriores valores.

Una propiedad muy interesante del *One-Hot encoding* es que uniformiza el coste de la predicción en caso de fallo independientemente de cuál sea la clase en la que (erróneamente) se ha categorizado el ejemplo. Pues es evidente que el error (que podemos medir ahora con la distancia euclídea) entre los dos vectores correspondientes a dichas clase va a estar acotado

entre $[0,1]$. Esto lo hace especialmente adecuado para modelos no-lineales como los que nos ocupan. Pues uno podría pensar que estas tareas de clasificación se podrían realizar con modelos lineales, prediciendo las clases sobre los valores enteros del dominio de una función. Pero esto no es posible, ya que el modelo tendería a predecir siempre los valores centrales de ese dominio, pues esos son los que minimizan el error medio ya que este no es uniforme en todos los casos.

3.3. Funciones de activación

En la arquitectura original de *WaveNet* se detalla el uso de las llamadas *gated activation units*, pues han mostrado buenos resultados en trabajos previos de este mismo equipo y además la bibliografía (como Nair y Hinton (2010)) ratifica su superioridad frente a otras funciones no lineales como *ReLU* (que veremos más adelante). Dichas unidades vienen dadas por

$$z = \tanh(W_{f,k} * x) \odot \sigma(W_{g,k} * x),$$

donde $*$ es el operador de convolución, \odot indica la multiplicación elemento a elemento, σ es la función *Sigmoide*, k es el índice de la capa, W es el filtro de la convolución, y f y g denotan *filter* y *gate* respectivamente.

La intuición detrás del uso de estas unidades viene de las redes neuronales recurrentes, en las que se pueden producir recurrencias en las conexiones de la red. Para evitar que los datos se perpetúen indefinidamente, este tipo de redes suele tener algún parámetro para el filtrado de los datos en los que se produce esta recurrencia. En el caso de las redes neuronales autorregresivas esta recurrencia está implícita en los datos con los que se trabaja, por lo que su uso también tiene cabida. La motivación detrás de usar dos funciones es la mayor facilidad a la hora de aproximar funciones no lineales. Y la notación de *filter* y *gate* responden a la forma en la que operan estas funciones. La función *tanh* devuelve datos entre $[-1, 1]$, por lo que puede entenderse como un “filtro” sobre el input; y la función sigmoide los devuelve entre $[0, 1]$, por lo que puede entenderse como una “puerta” (*gate*) que se abre en mayor o menor medida para permitir o no la recurrencia de los datos.

La función *tanh* (Figura 3.10) es una función que devuelve valores en un rango $[-1, 1]$. Es una función trigonométrica, su nombre viene de “tangente hiperbólica” y responde a:

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

Hemos de darnos cuenta también de que, en la parte final de la red (donde realmente se modelan las distribuciones de probabilidad que predecimos), se aplican una serie de capas *ReLU*. La función *ReLU* (*Rectified Linear Unit*, Figura 3.11) es una función de activación muy comúnmente aplicada cuando hablamos de redes neuronales convolucionales, pues han mostrado una gran efectividad cuando se trata de realizar clasificaciones no lineales. Dicha activación viene dada por:

$$f(x) = \max\{0, x\}.$$

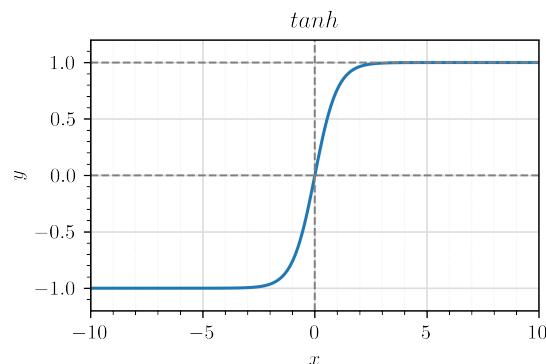


Figura 3.10: Valores dados por $\tanh(x)$.

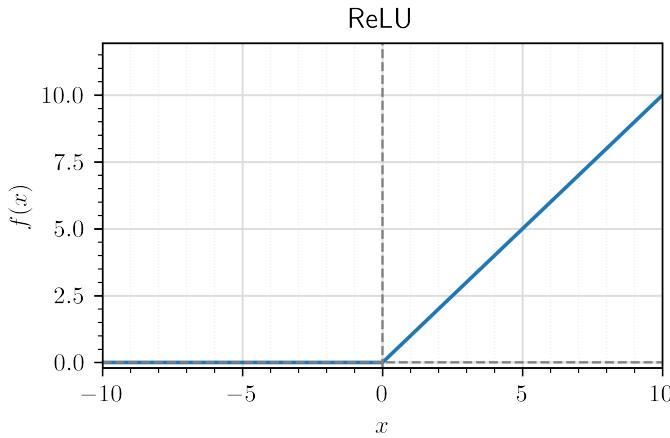


Figura 3.11: Valores de ReLU dados por $f(x)$.

3.4. *Residual and skip connections*

Una vez más, apoyándose en la literatura, los creadores de *WaveNet* emplean en el modelo el uso de conexiones “residuales” (*residual connections*) y “omitidas” (*skip connections*). Las conexiones residuales siguen el flujo unidireccional que los datos seguirían en una red tradicional, mientras que las conexiones omitidas conectan el final de la red con los datos después de cada una de las capas que se les aplican. Este tipo de conexiones aceleran la convergencia y, de este modo, permiten el entrenamiento de modelos profundos como es el que nos ocupa.

El flujo de datos que determinan este tipo de conexiones es apreciable de forma más clara en la Figura 3.3. Lo que vemos en este caso es la arquitectura del modelo para el caso de un solo bloque de k capas como las comentadas en la Sección 3.1.2, donde se muestran los distintos *dilation rates* para el caso de $k = 4$.

3.5. *WaveNets* condicionadas

Llegados a este punto hay que reparar en el hecho de que, dado que queremos generar audio de acuerdo a una especificación, necesitamos indicar al modelo algún tipo de abstracción de esta. Y de este modo, mientras se va generando el audio mediante la autorregresión, facilitaremos al modelo algún tipo de indicación (h) para que los sonidos generados se ajusten a lo que queremos en un instante dado.

$$p(x|h) = \prod_{t=1}^T p(x_t|x_1, \dots, x_{t-1}, h).$$

En el trabajo original (Van Den Oord y col., 2016) se detallan dos formas distintas de condicionamiento de la red: *global conditioning* (condicionado global) y *local conditioning* (condicionado local).

El condicionado global está pensado para indicar, como su propio nombre dice, características globales del sonido generado. En el contexto de uso original, el *Text-to-Speech*, este condicionado podría corresponderse con algún tipo de codificación de las características de la voz con la que se está generando el audio. Esto nos permitiría, cambiando esta codificación, generar distintas voces o incluso voces que la red no ha visto en tiempo de entrenamiento

y cuyas características representamos en dicha codificación. Mediante el condicionado local se pretende indicar al modelo una segunda serie de datos (probablemente con menor frecuencia de muestreo) con características del ámbito local de la onda en el instante que se está generando. Originalmente era utilizado para el carácter de la frase a generar que en un momento dado debería de estar pronunciándose. De este modo, el modelo ha de encargarse de producir las frecuencias correspondientes a los fonemas que le suministra el condicionado local así como las interacciones entre los mismos.

3.6. *Context stacks*

En la especificación original de *WaveNet* también se menciona una manera alternativa de aumentar el campo receptivo del modelo sin utilizar métodos clásicos como aumentar las capas, la longitud de los filtros, etc. Esta alternativa consiste en la creación de lo que llaman *context stacks*, que procesan una porción mayor del audio original y que a su vez condicionan la gran *WaveNet* que se encarga de la generación de audio en sí.

3.7. Arquitectura del modelo

En el caso de este trabajo, se provee una implementación en la librería *TensorFlow* de una variante del modelo *WaveNet* (Van Den Oord y col., 2016) en el repositorio de GitHub⁷ adjunto dentro de un *Jupyter Notebook*⁸. En este trabajo no prima la arquitectura software del modelo a desarrollar, por lo que consideramos que un cuaderno se ajusta a las necesidades y además permite mostrar el desarrollo del modelo de forma más interactiva. El modelo desarrollado hereda muchas características del original. Sin embargo, parte de los parámetros empleados para la construcción del mismo están inspirados en alguno de los mencionados trabajos que experimentan con la arquitectura original de *WaveNet* en aras de expandir sus fronteras.

En Hawthorne y col. (2018) se pretende crear un modelo capaz de transcribir, componer y sintetizar audio. Tales metas exceden las del trabajo que nos ocupa y por tanto la sección de contenido que nos es útil es reducida. En la sección dedicada a la síntesis de audio se habla de emplear una *WaveNet* para ello y detalla el empleo de 6 bloques de 10 capas como las que se han descrito anteriormente. Sin embargo, un modelo de tales características es demasiado complejo para el hardware disponible, por lo que ha habido que disminuir el número de capas por bloque de 10 a 7. En Hawthorne y col. (2018) también se detalla el uso de un *context stack* compuesto de 2 bloques cada uno de 6 capas, y así se ha realizado en el caso que nos ocupa. En la Figura 3.12 se muestra la arquitectura del modelo producido de acuerdo a la notación de la Figura 3.3. Este está compuesto por un total de 1444408 parámetros entrenables.

Como podemos ver en la mencionada figura, el modelo se compone de dos entradas y una salida. De este modo la red tomará, mediante la entrada “normal”, los τ niveles previos de audio. A partir de estos valores, de la información de condicionado local que se suministra a través de la otra entrada, y de los coeficientes propios del modelo, procurará inferir cuál es el siguiente nivel de onda. En este caso, el condicionado global podría corresponder con las características de distintos instrumentos para los que quisiéramos generar interpretaciones. Pero dado que el *dataset* empleado solo contiene audio correspondiente a interpretaciones de piano, carece de sentido emplear este condicionado y por ello no se ha utilizado. Con respecto al condicionado local, se suministrará al modelo vectores de 88 elementos (el número de teclas

⁷<https://github.com/alvaro-lg/TFG.git>

⁸Fichero TFG.ipynb

de un piano), donde cada valor del vector en un instante dado se corresponde con la velocidad normalizada ($\in [0, 1]$) con la que se está pulsando la tecla que representa en ese preciso instante. Los datos relativos a la autorregresión del audio están representados como una distribución categórica por cada muestreo de audio, como ya se ha descrito anteriormente. Para cada paso se empleará una ventana de tiempo que alberga tantos elementos como campo receptivo tenga la red, por lo que los datos con los que trabajamos tendrán la siguiente cardinalidad:

- **Entrada:** Rango receptivo \times número de niveles.
- **Entrada de condicionado local:** 1×88 .
- **Salida:** Rango receptivo \times número de niveles (como ya se ha dicho antes, las convoluciones causales dilatadas mantienen la cardinalidad de los datos).

Para el cálculo del campo receptivo se ha desarrollado una función⁹ que lo calcula en base al número de bloques y el número de capas por bloque. Para los valores concretos de 6 capas y 7 bloques, el campo receptivo tiene un valor de 757 muestras, y dado que se trabaja con audio muestreado a una frecuencia de 16kHz, esto se traduce en 47,31 ms de duración de la ventana de tiempo empleada en cada predicción. Por lo que lo único que queda por decidir para completar los distintos parámetros de la implementación es el número de bits a emplear para la codificación. A continuación se razona el valor a emplear para este campo.

3.7.1. Discretización y diferenciación

En la Sección 3.2.1 se ha hablado del algoritmo para la codificación de los datos relativos a los ficheros de audio y de cómo el uso de un determinado número de bits para dicha codificación condiciona la arquitectura de la red. También se menciona que los desarrolladores de *WaveNet* indican el empleo (en su trabajo original Van Den Oord y col., 2016) de 8 bits para esta codificación, aunque no detallan ningún argumento más para reforzar esta decisión de diseño. Por lo expuesto anteriormente, este parámetro va a repercutir tanto en la complejidad del modelo a entrenar como en el error derivado de la compresión de los datos. Y a su vez, estos factores influirán en el coste computacional requerido para el entrenamiento y la calidad del audio producido, respectivamente. Debido a esto, sería osado no detenerse a estudiar el valor óptimo de este parámetro en términos de complejidad del modelo y calidad del audio generado.

Para esto se ha tomado como métricas el número de parámetros del modelo y el error derivado de la compresión en un fichero del *dataset*¹⁰ en cada caso. Inicialmente las métricas fueron evaluadas en un dominio de números enteros desde 1, el mínimo número de bits al que se va a optar en la compresión, hasta 16, el máximo que podríamos emplear (pues es la codificación de la que partimos y que queremos comprimir). Sin embargo, en los valores finales de este dominio (15 y 16 bits) la red era extremadamente grande y el equipo empleado para la evaluación de estas métricas no era capaz siquiera de crear semejantes modelos. Como consecuencia de esto, la evaluación solo se ha realizado desde 1 hasta 14 bits, y podemos asumir que dado que la mera creación de los modelos para un mayor número de bits resulta impracticable, el entrenamiento de tales modelos tampoco será posible. También hay que mencionar que ambas métricas pertenecen a magnitudes muy distintas: el número de parámetros es del orden de millones mientras que el error de compresión es de decenas de miles. Por lo que hay que encontrar una manera de evaluar la interacción entre ambas

⁹Función `calculate_receptive_field`.

¹⁰Fichero: 2018/MIDI-Unprocessed_Chamber3_MID--AUDIO_10_R3_2018.wav--1.wav

funciones abstrayendo la magnitud de las mismas. La solución que proponemos para esto es normalizar los resultados, pues podemos aprovecharnos de que el máximo y el mínimo son conocidos en primera instancia para realizar la evaluación en un rango $[0, 1]$. De esta forma podemos evaluar la tendencia de ambas métricas al margen de sus diferentes magnitudes. En la Figura 3.13a se muestran los resultados de la evaluación normalizados.

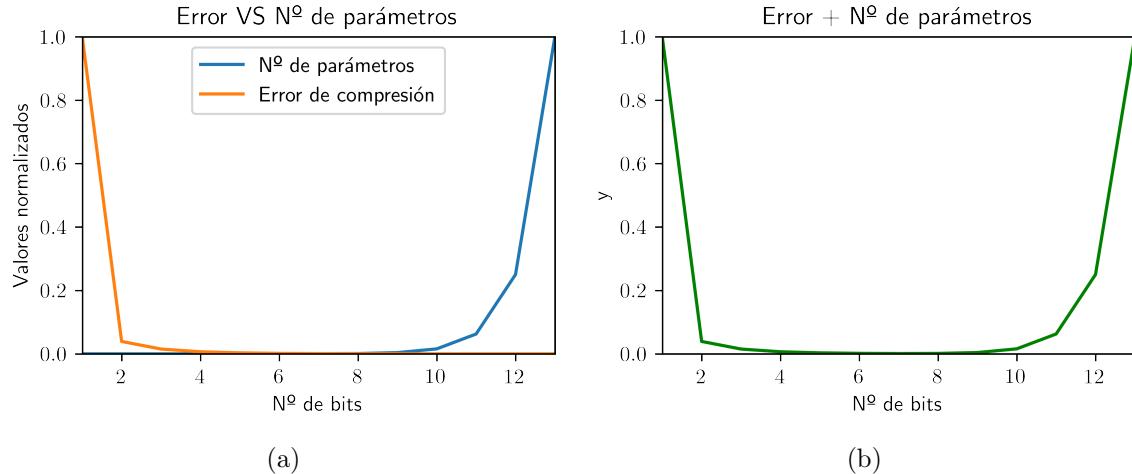


Figura 3.13: Valores normalizados del número de parámetros y error derivado de la compresión (3.13a), y suma de estos (3.13b).

Dado que el objetivo mediante la obtención de estos valores es conocer qué número de bits permite entrenar el modelo más rápido manteniendo un error de compresión bajo, la función a minimizar será la suma de estas dos magnitudes (Error + Nº de parámetros, Figura 3.13b). En esta función podemos apreciar un valle que va desde los valores correspondientes a 4 bits hasta los 8. Por lo que se puede considerar cualquier número de bits en este rango como válido, pues el esfuerzo computacional requerido para entrenar estos modelos se ve respaldado por un error en la compresión aceptable. Sin embargo hay un factor que ha de tenerse en cuenta y que no se puede evaluar de una forma tan inmediata: la inteligencia del modelo. Un mayor número de bits hace que el modelo sea más complejo, pero esto también hace que el modelo tenga mayor poder de decisión. Como consecuencia de esto, nosotros inicialmente optamos por el mayor número de bits en el mencionado rango: 8 bits. Sin embargo, este modelo resultó ser demasiado complejo para el hardware del que se dispone y por ello el valor que se ha empleado finalmente ha sido de 7 bits (128 niveles distintos de señal).

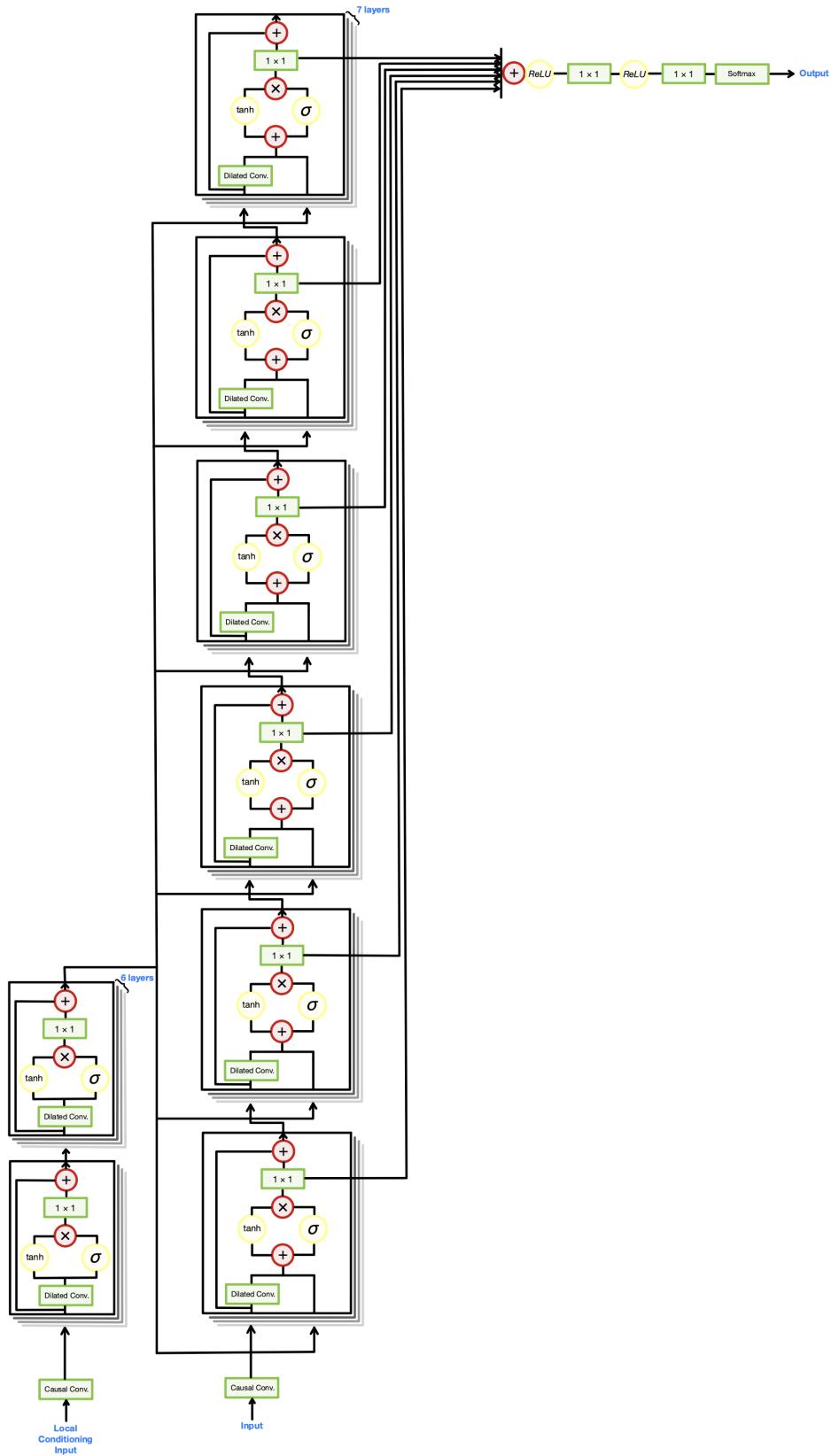


Figura 3.12: Esquema de la arquitectura del modelo desarrollado.

CAPÍTULO 4

ENTRENAMIENTO DEL MODELO Y RESULTADOS

4.1. Maestro *dataset*

El Maestro *dataset* (actualmente en su versión 3.0) es un *dataset* que inicialmente fue introducido en Dieleman y col. (2018) por la misma empresa DeepMind, en esta ocasión dentro del proyecto “Magenta”¹. El propósito de ese proyecto es la exploración del *Machine Learning* como una herramienta que nos asista en el proceso creativo. Como se explica en Dieleman y col. (2018), el *dataset* surge como fruto de una colaboración de esta entidad con la *International Piano-e-Competition*². El proceso de recopilación de datos se realizó mediante pianos equipados con capturadoras MIDI de alta precisión (hablaremos de este estándar más adelante).

Los datos recogidos en los ficheros MIDI albergan información relativa a la velocidad de pulsación de las teclas, así como información relativa a las pulsaciones de los pedales del instrumento. Estos ficheros están alineados con los de audio con una precisión de 3 milisegundos y también están divididos en interpretaciones individuales, con información acerca de su título, compositor y año de interpretación. Los ficheros de audio Wav tienen una frecuencia de muestreo de 44,1–48 kHz (16-bit PCM estéreo). El *dataset* también cuenta con una partición *train-validation-test* (cuya distribución a lo largo de distintas magnitudes se muestra en la Figura 4.1) realizada de antemano ya que en este caso no es trivial, y los creadores del *dataset* han tenido que encargarse de que la misma pieza no aparezca repetida en distintos subconjuntos aunque haya sido interpretada por distintos concursantes. Como se puede ver en la Figura 4.2, estas piezas pertenecen a una gran variedad de compositores de distintas épocas y estilos. Por lo que se puede decir que las interpretaciones abarcan una gran porción del repertorio pianístico.

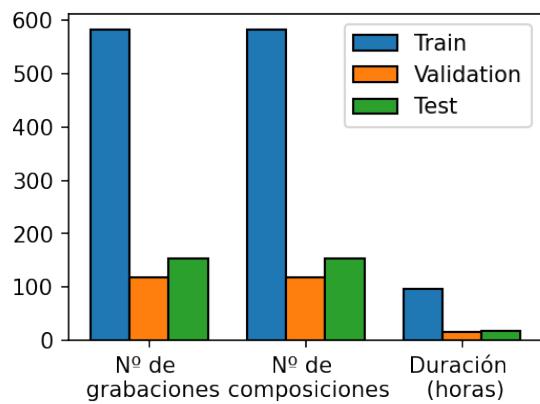


Figura 4.1: Estadísticas de los datos en el *dataset* en función del conjunto: entrenamiento, validación o test.

¹<https://magenta.tensorflow.org>

²<https://piano-e-competition.com>

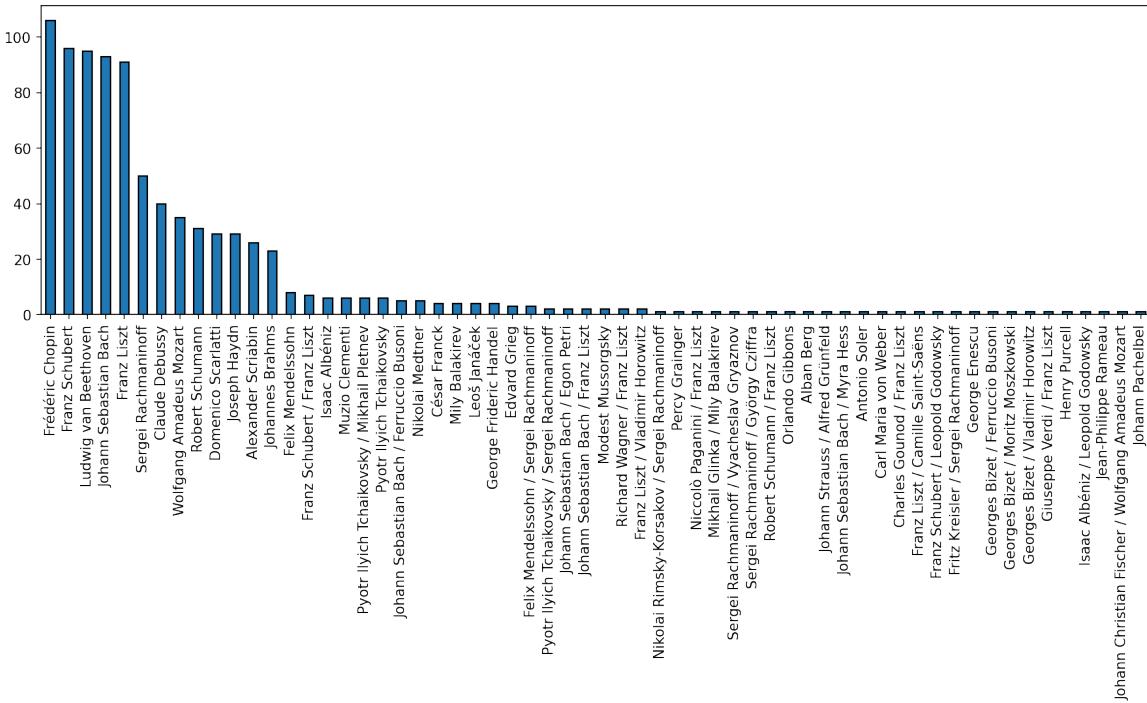


Figura 4.2: Distribución de los compositores en función del número de interpretaciones de piezas que se les atribuyen en el *dataset*.

4.1.1. Ficheros MIDI

Los archivos MIDI son un tipo de ficheros pertenecientes al estándar MIDI (*Musical Instrument Digital Interface*). La primera presentación del estándar tuvo lugar en la exhibición NAMM en Enero de 1983, pero no fue hasta agosto del mismo año que la versión 1.0 de dicho estándar vio la luz. El estándar MIDI alberga una especificación tanto software como hardware. Su objetivo principal es permitir el intercambio de información relativa a una performance entre diferentes instrumentos musicales o dispositivos como ordenadores, controladores de iluminación, etc (Association y col., 1996). Esta necesidad surge tras la irrupción de la electrónica en la industria de la producción musical, que se traduce tanto en la creación de nuevos instrumentos digitales capaces de crear todo tipo de nuevos timbres (“sintetizadores”), como en la digitalización del proceso de post-producción. Era muy común que estos instrumentos electrónicos fueran controlados mediante un teclado como el de un piano y dado que la forma de controlarlos siempre era la misma, sería conveniente tener algún tipo de abstracción digital (como es el estándar MIDI) que recabara la información relativa a las pulsaciones y demás elementos de una interpretación. Esta nos permitiría (entre otras cosas) trasladar esa información a otro instrumento que fuera compatible con este estándar y ver cómo sonaría esa misma interpretación con otro timbre.

4.1.2. Ficheros Wav

Wav es un formato de audio digital propietario desarrollado por Intel e IBM y utilizado para la digitalización de audio sin compresión. En estos archivos se almacenan los valores de la onda de audio como enteros de 16 bits que enumeran los distintos valores de onda de forma equidistante entre -1 y 1. Estos archivos suelen ocupar bastante memoria ya que, si cada muestra de audio ocupa 16 bits y queremos almacenar audio con una calidad similar a la de un CD (44,1 kHz), cada segundo de audio implicaría unos 86 kB.

4.2. Herramientas empleadas

El proyecto se ha implementado en un *Jupyter Notebook* que a su vez se ejecuta sobre Python 3. Como más adelante se detallará, se ha empleado toda una variedad de librerías de este lenguaje de programación para el procesado de los datos en las distintas etapas del mismo. Las más relevantes y conocidas son *TensorFlow* (para la construcción del modelo y las estructuras de datos relativas al entrenamiento), *numpy* (para diversos cálculos), y *matplotlib* (para todo lo relacionado con la representación de los datos, así como para la creación de gran parte de las ilustraciones que aparecen en este documento³). También hay que mencionar el uso de la librería *threading*, que más adelante veremos, y que ha servido para sacar el máximo partido del hardware disponible mediante el procesado paralelo de los datos.

4.3. Procesado de los datos

Tanto el código relativo al tratamiento de los ficheros MIDI como el de los ficheros Wav se han externalizado a las clases *midi_handler.py* y *wav_handler.py*, respectivamente. Esto se ha hecho con el propósito de simplificar el *Notebook*, así como potenciar la modularidad y, por ende, la mantenibilidad del código.

La clase *midi_handler.py* contiene los métodos necesarios para transformar los ficheros MIDI en listas de vectores como los mencionados anteriormente cuando se describía la entrada del condicionado global del modelo (Sección 3.5). También existe la posibilidad de muestrear estos ficheros a distintas frecuencias, así como de mostrar una representación de estas pulsaciones de un determinando fichero MIDI en lo que se conoce como un *pianoroll* (Figura 4.3a). Para la lectura de los datos de los ficheros MIDI, esta clase se apoya en la librería *mido*, para la representación de los *pianorolls* utiliza *matplotlib* (librería ampliamente conocida) y, con intención de agilizar el cómputo, los datos recabados se devuelven como objetos de la librería *numpy*. Con respecto a la clase *wav_handler.py*, proporciona la misma funcionalidad, solo que para los datos relativos a los ficheros Wav, y con la diferencia de que en este caso se emplea la librería *librosa* para la lectura de los ficheros.

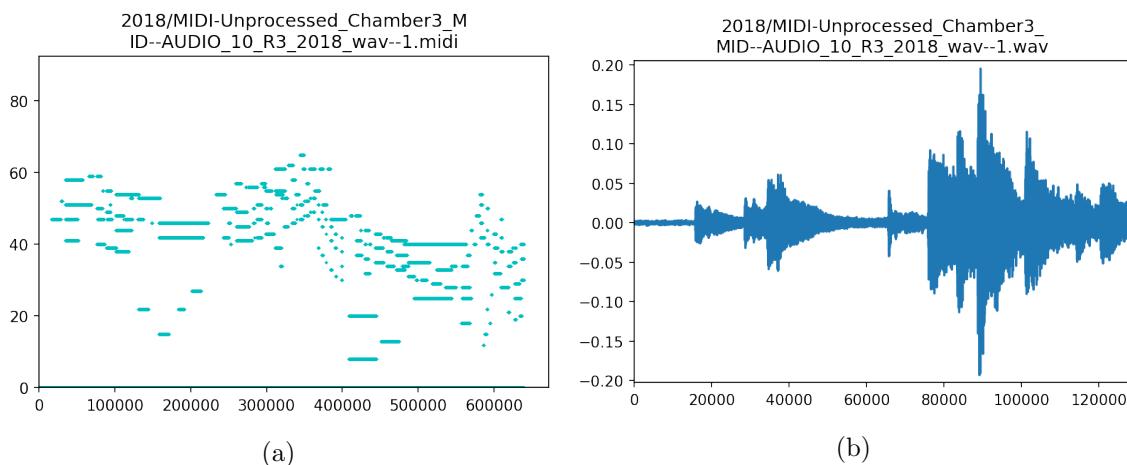


Figura 4.3: Representaciones de los dos tipos de datos que se manejan a alto nivel. “Pianoroll” para los archivos MIDI (4.3a) y valores de amplitud de onda para los archivos Wav (4.3b).

³El código relativo a la generación de estas figuras se encuentra bajo el directorio */figures/* del repositorio asociado.

Una vez leídos, hay que aplicarles a los datos alguna transformación más antes de suministrárselos al modelo para el entrenamiento. Como se puede ver en la Figura 4.4, la mayor parte de este procesado tiene que ver con los datos de los ficheros Wav. La clase `Wav_handler.py` los proporciona en forma de un array de `numpy` como valores de amplitud de onda ($[-1, 1]$) a una frecuencia determinada. Esta secuencia de valores hay que codificarla de acuerdo al μ -law encoding que ya se ha descrito (Sección 3.2.1), seccionarla en ventanas de tiempo de longitud igual al campo receptivo de la red (como se menciona antes), asociar como valor objetivo de una determinada ventana de valores aquella que la sigue y una vez están los datos estructurados de esta forma, aplicar el *One-Hot encoding* a cada uno de los valores codificados para transformarlos en una distribución categórica. Una vez se realiza esto, todo lo que queda es juntar los datos de los ficheros MIDI con los de los ficheros Wav procesados, mapear los datos con los nombres de las entradas del modelo y llamar al método `fit` de la red para comenzar el entrenamiento. Toda esta parte de transformación de los datos antes de entrenar el modelo se realiza mediante las clases y métodos de la API de `tf.data`. La búsqueda por la eficiencia y el máximo rendimiento ha hecho que sigamos las recomendaciones de usar esta API que se dan en la propia documentación de *TensorFlow*⁴, pues este conjunto de clases y métodos consiguen una integración máxima con el modelo (que también está implementado en *TensorFlow*), así como un gran rendimiento a la hora de construir canalizaciones de datos.

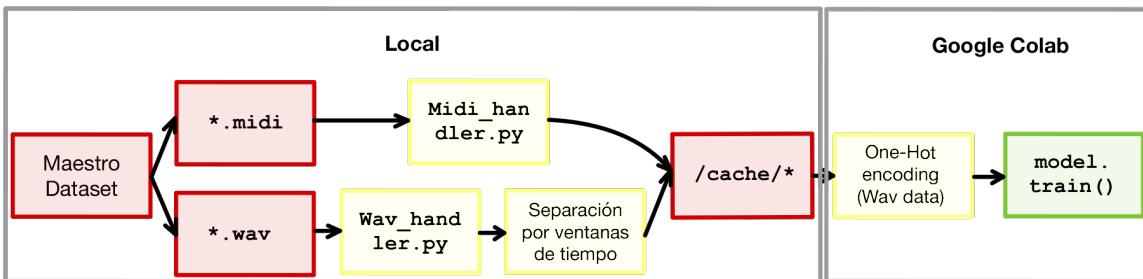


Figura 4.4: Esquema del proceso aplicado a los datos para el entrenamiento.

Para evitar las limitaciones derivadas del llenado de la memoria RAM (*Random Access Memory*, memoria de acceso aleatorio), se ha implementado una sección del código que procesa los datos empleando varios procesos paralelos (*multithreading*) y los escribe en ficheros a modo de *cache*. Estos procesos siguen el esquema típico de una cola de trabajos, y cada uno obtiene el primer elemento de la cola cuando ha terminado su trabajo, de tal forma que estos se asignan dinámicamente. Como ahora se comentará en más profundidad, se dispone de una gran cantidad de datos, por lo que se han almacenado los datos de la forma más compacta posible. Por ello se ha aplicado el algoritmo de compresión *GZIP* a los datos almacenados y estos se han guardado antes de realizar el paso al *One-Hot encoding* de los datos de los ficheros Wav, pues el *overhead* en tiempo de ejecución que implica esto es mínimo comparado con el incremento en memoria que supone tal nivel de redundancia en los datos. Toda esta parte adicional del procesado de los datos implica, además de las optimizaciones ya mencionadas en términos de uso de RAM, la desaparición del *overhead* que sucedía anteriormente cada vez que comenzaba el entrenamiento de un nuevo fichero. Como consecuencia de esto, donde antes había que leer y procesar desde cero cada uno de los ficheros siempre que se fueran a entrenar, ahora lo único que hay que hacer prácticamente es leer de memoria los datos directamente.

⁴<https://www.tensorflow.org/guide?hl=es-419>

4.4. Implementación del entrenamiento

Para el entrenamiento del modelo se han empleado un total de 100 *epochs*, un tamaño de batch de 25 y el optimizador Adam que ya se ha descrito anteriormente (Sección 2.3.1). Aparte de estos parámetros (que serían los tradicionales en cualquier proyecto de *Deep Learning* estándar), también hay una serie de peculiaridades del entrenamiento de este modelo que son dignas de mención. La mayoría de estas emanan del hecho que se ha comentado anteriormente de que se dispone de muchos datos. Concretamente, el *dataset* cuenta con un total de 1276 ejemplos (compuestos cada uno por un fichero MIDI y un Wav), cada uno de ellos con una duración media de 560 segundos y una frecuencia de muestreo de 16 kHz. Esto supone un total de 11.432.960.000 ejemplos, algo inabarcable en cualquiera de los equipos de los que se dispone.

A la vista de tal cantidad de datos, se ha procurado disponer de un equipo con mayor capacidad de cómputo. El primer intento de esto fue en el Instituto de Física de Cantabria⁵ (IFCA-CSIC), que facilitó uno de sus nodos para el entrenamiento del modelo. Por desgracia, ese equipo tampoco cumplía los requisitos suficientes para hacer posible el entrenamiento y por tanto hubo que buscar otra alternativa. Esta fue el servicio de pago Google Colab Pro⁶ y, aunque sigue sin ser suficiente para entrenar el modelo en todos los ejemplos (como se explica a continuación), nos acerca en una medida considerable a este objetivo. Pues a diferencia de el del IFCA, este dispone de tarjeta gráfica dedicada para la mejor parallelización del entrenamiento, hardware entrenado para la ejecución de *TensorFlow* empleando instrucciones vectoriales (en nuestro caso estas eran las instrucciones AVX de Intel), mayor capacidad de memoria RAM y además el servicio Google Colab está pensado para la ejecución de *Jupyter Notebooks*, como sucede con la implementación de este proyecto. Esta serie de características han hecho que el entrenamiento del modelo se realice empleando Google Colab Pro.

Recordemos ahora que en términos de espacio de disco, los datos disponibles conforman unos 130 GB. Pero en realidad esto puede traducirse en incluso más memoria en tiempo de entrenamiento, pues el hecho de trabajar con datos que hay que dividir de acuerdo a una ventana de tiempo implica redundancia en estos, ya que hay una serie de datos que son comunes a un ejemplo y a aquel que le sigue. Es por esto que si pretendemos construir un gran objeto en el que estén todos los ejemplos del *dataset*, muy probablemente llenaremos los 32 GB de memoria RAM que Google Colab nos ofrece. Debido a esto el entrenamiento se ha modificado de forma que los ficheros se procesan y entranan el modelo de uno en uno, para evitar llenar la RAM. Esto ha hecho que haya que encargarse de muchos aspectos que en otro contexto se delegarían a *TensorFlow*: registro de las estadísticas del entrenamiento, registro de las veces (las *epochs*) que se ha entrenado el modelo en cada fichero, gestión de las distintas *epochs* y evaluación del conjunto de *cross-validation* a la finalización de cada una de estas. A continuación se aporta una breve explicación de lo que se ha hecho en cada caso:

1. Para el loggeo de las estadísticas se ha empleado un *callback* de *TensorFlow* llamado “**CSVLogger**”. Este nos permite escribir en un fichero externo⁷ los valores de las estadísticas en cada paso del entrenamiento.
2. En el caso del registro de qué ficheros se han entrenado durante qué *epochs*, lo que se ha hecho ha sido añadir una columna al .csv original llamada “**epoch_trained**”. Esta columna inicialmente tiene un valor de 0 para cada fichero y se incrementa en 1 cada vez que el modelo se entrena para ese determinado fichero. Para garantizar el orden

⁵<https://ifca.unican.es/es-es>

⁶<https://colab.research.google.com/?hl=es>

⁷Fichero: **training_accuracy.csv**

entre las distintas *epochs*, en un primer instante el código consulta todos los valores de este campo para iniciarse en el mínimo y comenzar a entrenar por aquellos ficheros que tienen ese valor.

3. La gestión de las distintas *epochs* se ha hecho mediante un simple bucle `while` y realizando la consulta que se acaba de describir durante el número de *epochs* que hayamos fijado.
4. Para la evaluación del modelo en el conjunto de *cross-validation* se ha replicado el mismo código que en el entrenamiento pero llamando el método `model.evaluate` en vez de `model.fit` como se hacía. Y para el registro de los resultados se emplea otra instancia del `CSVLogger` del que ya se ha hablado, pero esta vez sobre otro fichero⁸.

También ha habido una serie de inconvenientes debido a los términos de servicio de Google Colab, ya que en estos se especifica una duración máxima de la ejecución del proceso de 24 horas. Por lo que también hay que hacer que nuestro código para el entrenamiento del modelo sea resistente a interrupciones externas. Para esto se ha hecho que el `.csv` original se actualice cada vez que se actualiza un valor de la columna “`epochs_trained`”, y que el estado del modelo en cada paso del entrenamiento se guarde. Para ello se ha empleado otro `callback` como los anteriores, aunque esta vez se trata del `callback` llamado `ModelCheckpoint`. Mediante sus parámetros se ha programado para guardar aquel modelo con mayor precisión en el entrenamiento en un fichero externo⁹. Y dado que durante las distintas iteraciones del entrenamiento la precisión del modelo puede empeorar ligeramente, también se guarda el último modelo en haber sido entrenado (que no tiene por qué ser necesariamente el mejor) en un fichero “`model.hdf5`”. En un primer instante el código del entrenamiento consulta si existe este último fichero: si existe, toma sus valores y distintos parámetros y los utiliza para continuar con el entrenamiento; y si no existe, crea un nuevo modelo y comienza a entrenarlo.

Y para hacer que el entrenamiento del modelo de acuerdo al hardware del que disponemos sea en un tiempo razonable y que nos permita realizar el número de *epochs* que pretendemos, ha habido que prescindir de una porción de los datos. La primera reducción que se ha aplicado a los ejemplos ha sido eliminar aquellas interpretaciones de piezas duplicadas. Es evidente, debido a las sutilezas de cada grabación, que los valores de los propios niveles de onda con los que se va a entrenar el modelo son distintos aunque se trate de dos grabaciones de una misma pieza. Pero pese a esto podemos asumir que sí que van a presentar un mayor grado de similaridad entre aquellas grabaciones de las mismas piezas que entre las de piezas distintas. Con este filtrado se reduce el número de ficheros del *dataset* de 1276 a 854. También se puede reducir el número de ejemplos fijándonos en el principio de las grabaciones. El comienzo de las grabaciones no está perfectamente alineado con el comienzo de las interpretaciones en sí, por lo que hay siempre unos segundos de silencio que no interesan. Para ello se han estudiado los valores de amplitud de onda de los primeros segundos de todos los ficheros.

Como podemos ver en la Figura 4.5, los valores de amplitud de la onda comienzan a pronunciarse a partir de 1,25 segundos en promedio. Por lo que, conociendo que tenemos una frecuencia de muestreo de 16 kHz, podemos ignorar los 20000 primeros ejemplos de cada fichero.

En términos de número de ejemplos por fichero, también se han aplicado una serie de reducciones ya que sigue habiendo demasiados ejemplos. Para ello se ha tomado como referencia el fichero de menor extensión del *dataset*, que en concreto es de 45.15 segundos. Dicha duración se ha multiplicado por un factor¹⁰ para reducir el número de ejemplos y poder

⁸Fichero: `crossval_accuracy.csv`

⁹Fichero: `best.hdf5`

¹⁰En el código se corresponde con la constante `DIM_FACTOR`

ajustar los tiempos mencionados. Por lo que para cada fichero se seleccionan el número de ejemplos a partir de 1,25 segundos, que se corresponde con la duración mínima de un *dataset* reducida de acuerdo a una constante.

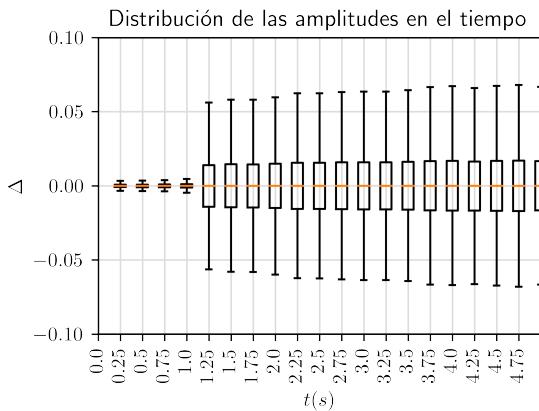


Figura 4.5: Diagrama de bigotes con los valores de amplitud de onda durante los 5 primeros segundos por cada 0,25 segundos.

de cada bucle. Tras aplicar el proceso descrito, sin una penalización aparente del tiempo de ejecución.

4.5. Métricas de evaluación

La función de pérdida para el entrenamiento del modelo ha sido la *Categorical Crossentropy*. Aunque en el trabajo original (Van Den Oord y col., 2016) se detalla el uso de la conocida *Negative Log-Likelihood*,¹¹ *Tensorflow* no cuenta con una implementación nativa de esta y también se ha optado por mantener un código más simple, por lo que se ha decantado por el uso de la *Categorical Crossentropy*. Además, el uso de esta métrica es muy recomendado cuando se trabaja con capas *Softmax* que a su vez implican la presencia de distribuciones de probabilidad. La función viene dada por

$$\text{Loss} = - \sum_i y_i * \log(\hat{y}_i).$$

Esta métrica mide la diferencia entre dos distribuciones de probabilidad. Dicha propiedad la hace muy conveniente ya que se pretende medir la disparidad entre las probabilidades predichas y aquellas distribuciones objetivo fruto del *One-Hot encoding*. El símbolo negativo del principio se debe a que el término del sumatorio devolverá valores negativos ($\log(x) < 0$, $\forall x \in (0, 1)$) y mediante este cambio de signo, la pérdida disminuirá conforme las distribuciones de probabilidad a evaluar sean más similares.

Para la evaluación también se empleará como métrica la precisión del modelo (*accuracy* en inglés). Dicha métrica nos indicará, de forma porcentual, la porción de predicciones correctas que el modelo ha realiza. Es inmediato intuir cómo esta métrica tendrá una tendencia inversa a la de la pérdida. El cálculo de la pérdida y la precisión es realizado automáticamente por *TensorFlow*.

¹¹Para más información recomendamos la lectura de Miranda (2017).

Además de estas reducciones, también ha habido que ignorar una serie de ficheros más debido a que, como consecuencia de su tamaño, la mera lectura de los mismos ha hecho que se alcancen los límites de memoria RAM. Los ficheros que se han eliminado son aquellos que tienen una duración mayor a 900 segundos, por lo que ahora quedan 709 ficheros en el *dataset*. Por último, también hay que decir que ha habido que encargarse del proceso que normalmente se conoce como “recolección de basura” (normalmente automático) para eliminar aquellas variables que se recalculan en las distintas iteraciones del bucle de entrenamiento y liberar así el espacio que ocupaban sus antiguas instancias. Para esto se ha invocado la instrucción `del` de Python con dichas variables al final

4.6. Resultados

Tras un entrenamiento durante 100 epochs (que se han traducido en unos 15 días) se han obtenido las siguientes estadísticas del entrenamiento.

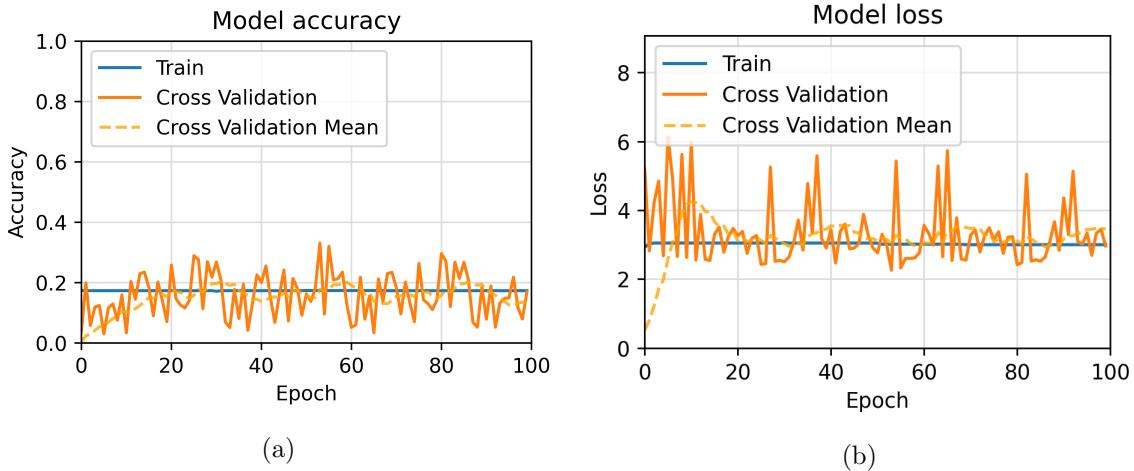


Figura 4.6: Evolución de la precisión (4.6a) y pérdida (4.6b) del modelo a lo largo del entrenamiento para los conjuntos de entrenamiento y *cross-validation*.

Como se puede ver en la Figura 4.6, los resultados cuantitativos muestran poco progreso en la precisión del modelo tras su entrenamiento. Sí que vemos un progreso en los valores de pérdida de las distintas configuraciones del modelo pues conforme estas avanzan en el entrenamiento, se puede apreciar una disminución de la desviación estándar de dichos valores. Esto nos hace pensar que el error de predicción se uniformiza para los distintos ejemplos del modelo y que la falta de precisión en el modelo podría mitigarse mediante más tiempo de entrenamiento y más poder computacional como de los que otros proyectos de este ámbito han empleado.

También hemos de tener en cuenta, a parte de los datos de precisión del modelo mostrados en dicha figura, la autorregresividad del modelo. Ya que en tiempo de generación de una determinada interpretación las predicciones futuras van a ser usadas para predecir las siguientes, un fallo en una de estas predicciones podría acarrear errores consecutivos en las siguientes predicciones. Por lo que hay que garantizar una gran precisión en el modelo tras su entrenamiento para así, hacer que este sea capaz de tolerar errores en el momento de predecir interpretaciones completas. De lo contrario el modelo llegaría a una deriva, ya que por muy pequeño que sea el umbral de error con el que contemos, este va a ser significativo al acumularse durante las 16000 predicciones que se realizan por cada segundo de audio generado. Y la única forma de obtener tales niveles de precisión tras el entrenamiento es entrenar la red durante más tiempo y con más datos, para lo que nuevamente se precisará de un hardware con mayor capacidad de cómputo.

CAPÍTULO 5

CONCLUSIONES

En resumen, en este proyecto se ha creado un modelo autorregresivo que teóricamente es capaz de generar audio a 16 kHz correspondiente a la interpretación de una determinada obra de piano dada su especificación MIDI. Este modelo está compuesto por dos entradas y una salida. Una de estas entradas toma las últimas muestras de audio correspondientes a una ventana de tiempo y, basándose en esto y en la información que se le suministra en la otra entrada (correspondiente al condicionado local, Sección 3.5), predice los valores de los siguientes valores de la ventana de tiempo desplazada. Las características del audio (comprimido de acuerdo al algoritmo μ -law *companding*, Sección 3.2.1) y la especificación MIDI se explotan mediante una serie de capas convolucionales para posteriormente traducirse en unas distribuciones categóricas de acuerdo a lo explicado en el Capítulo 3. El entrenamiento de este modelo se ha realizado sobre el Maestro *dataset*, compuesto por ficheros MIDI y sus correspondientes interpretaciones en ficheros Wav. El *dataset* supone un gran volumen de datos, y junto con el tamaño del modelo, ha hecho que el entrenamiento no haya sido inmediato y que haya habido que aplicar una serie de procesos en aras de hacerlo más eficiente.

Para realizar todo esto, se han visto involucrados los conocimientos asociados a una gran variedad de las asignaturas vistas en el grado. Las más reseñables son: *Aprendizaje Automático y Minería de Datos*, por la relación que tiene con el *Deep Learning* (disciplina que se explora en este trabajo); *Estadística y Optimización*, que trata la mayor parte de las herramientas empleadas para el análisis de datos; *Natural Language Processing*, necesaria para comprender cómo funciona la aplicación de *WaveNet* al campo del TTS y así poder trasladarlo a la interpretación de música; *Programación Paralela*, para la implementación paralela de los procesos que almacenan los datos de entrenamiento en ficheros (a modo de *cache*); aquellas de programación (*Introducción al Software*, *Métodos de programación y Estructuras de Datos*) y las que tratan sobre la creación de agentes inteligentes (como pueden ser *Sistemas Inteligentes* o *Representación del Conocimiento*).

También hay que decir que desde aquí criticamos la opacidad de la documentación disponible en este área. Diversas fuentes en internet aseguran que esto se debe a los intereses comerciales que hay en la creación del agente de TTS perfecto, ya que los asistentes por voz cada vez se integran en más dispositivos. Esto ha despertado una carrera investigadora entre las distintas grandes empresas del sector en esa dirección y, por ende, la documentación publicada es muy inconcreta, algo que ha dificultado el desarrollo de este proyecto.

5.1. Aportaciones y trabajo futuro

Mediante este proyecto (y los materiales asociados) se proveen los artefactos software necesarios para la creación y entrenamiento de un agente inteligente capaz de interpretar

música de piano dada su especificación MIDI. Dichos artefactos contemplan varios escenarios para el entrenamiento: tanto si este se puede realizar mediante una interfaz gráfica o como si por el contrario se debería realizar mediante una interfaz de línea de comandos (`ssh` en entornos HPC por ejemplo). Para estos dos escenarios se utilizarían el *Jupyter Notebook* (`TFG.ipynb`) y el *script de shell* (`train.sh`) respectivamente.

El desarrollo del modelo se ha realizado procurando una máxima fidelidad a la especificación original de WaveNet. Sin embargo, la lentitud en la convergencia de los valores de coste del modelo en tiempo de entrenamiento nos hace pensar que quizás haya margen para la mejora. Para ello habría que realizar más entrenamientos durante más tiempo, variando la arquitectura (por ejemplo, con más bloques, más capas por bloques, cambiando la función de pérdida, etc.). Pero dadas las limitaciones temporales del proyecto y de que el mero hecho de entrenar el modelo durante 100 *epochs* ha implicado 15 días, ha sido imposible poder probar estas variantes con suficiente temporalidad como para extraer alguna conclusión firme.

Otra vertiente del problema que quedaría abierta una vez se alcance una implementación completamente funcional de este modelo sería la de trasladar este mecanismo a la generación de interpretaciones para otros instrumentos. Pues como ya se ha dicho en la Introducción, el objetivo final de esta línea de investigación sería la automatización completa del proceso de producción musical. Sin embargo, los *datasets* que combinen archivos MIDI alineados con el audio Wav de una interpretación real (como sucede en el caso del Maestro *dataset*) escasean para el caso de otros instrumentos. Para la mayoría de instrumentos, debido a la forma que tienen de usarse, no es tan fácil recavar los elementos físicos que suceden en una interpretación (como en nuestro caso los *pianorolls* que contienen información acerca de las pulsaciones de un piano). Como consecuencia de esto, creeremos que esa escasez persistirá con el paso del tiempo, y consideramos que hay que tomar otro enfoque para abordar el problema.

BIBLIOGRAFÍA

- Association, M. M. y col. (1996). The complete MIDI 1.0 detailed specification. *Los Angeles, CA, The MIDI Manufacturers Association.*
- Dieleman, S., van den Oord, A. & Simonyan, K. (2018). The challenge of realistic music generation: modelling raw audio at scale. *Advances in Neural Information Processing Systems, 31.*
- Goodfellow, I., Bengio, Y. & Courville, A. (2016). *Deep Learning* [[http : / / www . deeplearningbook.org](http://www.deeplearningbook.org)]. MIT Press.
- Hawthorne, C., Stasyuk, A., Roberts, A., Simon, I., Huang, C.-Z. A., Dieleman, S., Elsen, E., Engel, J. & Eck, D. (2018). Enabling factorized piano music modeling and generation with the MAESTRO dataset. *arXiv preprint arXiv:1810.12247.*
- Hinton, G., Srivastava, N. & Swersky, K. (2012). Neural networks for machine learning. *Coursera, video lectures, 264(1), 2146-2153.*
- Jurafsky, D. & Martin, J. H. (s.f.). Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition.
- Kingma, D. P. & Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980.*
- Miranda, L. J. (2017). Understanding softmax and the negative log-likelihood". *ljvmiranda921.github.io. %5Curl%7Bhttps://ljvmiranda921.github.io/notebook/2017/08/13/softmax-and-the-negative-log-likelihood/%7D*
- Nair, V. & Hinton, G. E. (2010). Rectified linear units improve restricted boltzmann machines. *Icml.*
- Recommendation, C. (1988). Pulse code modulation (PCM) of voice frequencies. *ITU.*
- Ruder, S. (2016). An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747.*
- Van Den Oord, A., Dieleman, S., Zen, H., Simonyan, K., Vinyals, O., Graves, A., Kalchbrenner, N., Senior, A. W. & Kavukcuoglu, K. (2016). WaveNet: A generative model for raw audio. *SSW, 125, 2.*
- Zeiler, M. D. (2012). Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701.*

APÉNDICE A

APLICACIONES DE MODELOS ANTERIORES

Este anexo está pensado para aquellos lectores que deseen obtener una visión práctica de los rudimentos de aquellos modelos de regresión que se han explicado anteriormente.

Regresión Lineal Multivariante

Para ejemplificar el caso del modelo explicado en la Sección 2.1, nos basaremos en el ejemplo de la Figura A.1. Los datos están extraídos de un *dataset*¹ que recaba información acerca de distintos ejemplos de coche del mercado estadounidense. Para cada coche se almacenan distintas características (número de puertas, tracción, tipo de combustible...) así como su precio. Nuestro objetivo es ponernos en la situación de una nueva empresa que pretende entrar en el mercado y a partir de estos parámetros predecir cuál sería el precio de un futurable nuevo modelo de coche. Y puesto que nuestro objetivo es ilustrar un ejemplo, vamos a procurar realizar esto con tan solo uno de los parámetros del *dataset*.

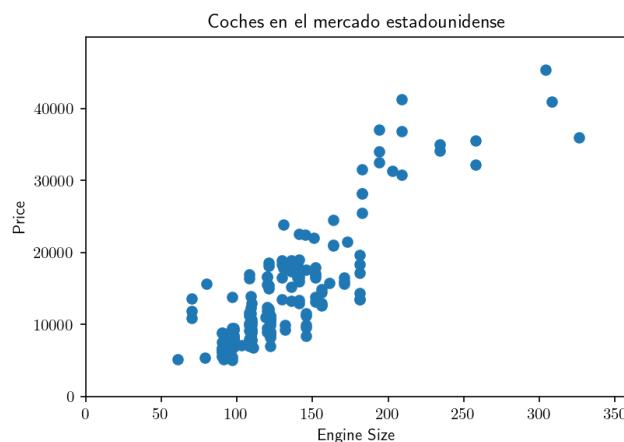


Figura A.1: Dispersión de los pares de valores tamaño del motor – precio.

Realizando un pequeño estudio de la correlación que tienen los distintos parámetros del *dataset* con el precio, hemos visto que el que mejor se ajusta es el tamaño del motor. Si lo representamos como en la Figura A.1 vemos cómo efectivamente es así, pues se puede intuir

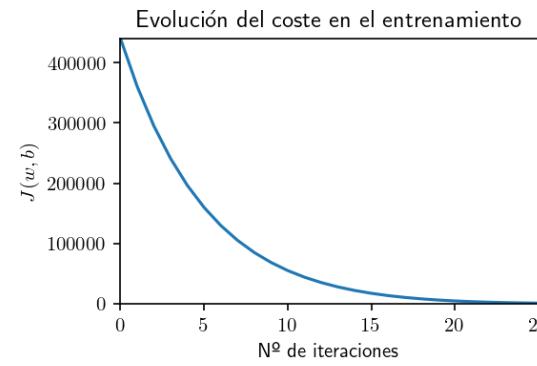
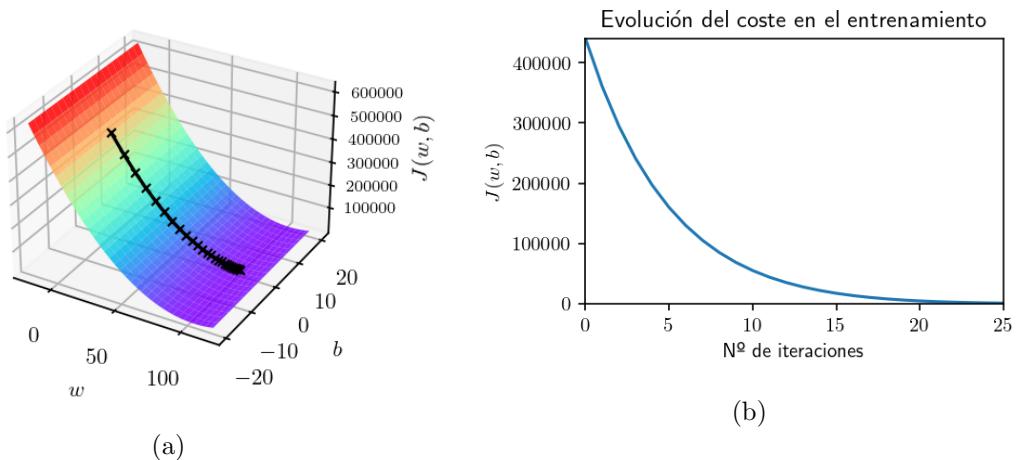
¹Fuente: <https://www.kaggle.com/datasets/hellbuoy/car-price-prediction>

una especie de linealidad entre ambos valores. Por lo que nuestra tarea ahora es encontrar aquella recta que aproxima los valores del precio con el menor error medio posible. Por lo que dicha recta será de la siguiente forma

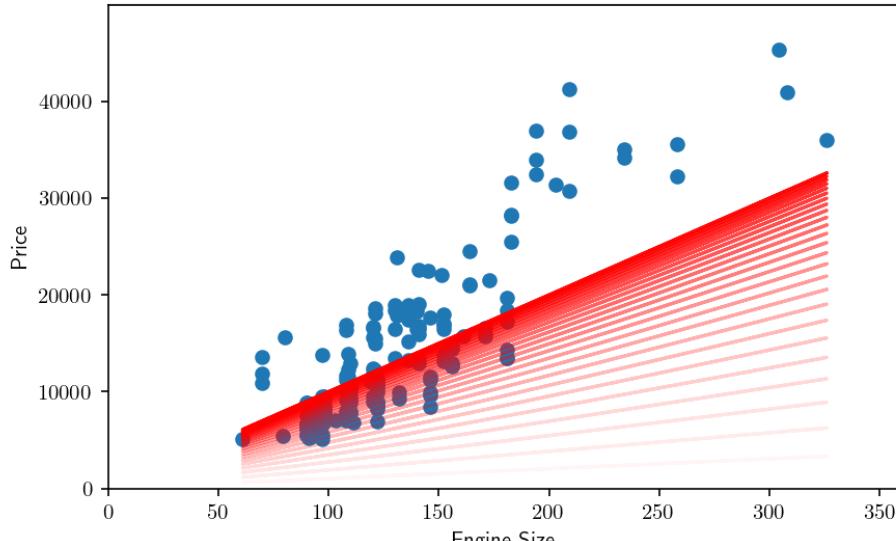
$$\hat{y} = wx + b.$$

Comenzaremos con una inicialización aleatoria de los parámetros de dicha recta, y mediante el algoritmo del gradiente descendente, procuraremos mejorarlo. Experimentalmente hemos determinado un número de 25 iteraciones, y un ratio de aprendizaje $\alpha = 5 * 10^{-6}$. En la Figura A.2 se muestra la evolución de los coeficientes mediante el algoritmo del gradiente descendente en el contexto de la función de coste, el valor de coste en cada iteración para dichos coeficientes, y la evolución de los valores de la regresión en cada iteración.

Evolución de los coeficientes en el entrenamiento



Evolución de la regresión en el entrenamiento



(c)

Figura A.2: Evolución de los coeficientes (A.2a), el coste (A.2b) y la recta de la regresión (A.2c) durante el entrenamiento.

Regresión Logística

Para el caso del modelo de la Sección 2.2, hemos utilizado un *dataset*² de vinos, donde se nos provee distintos parámetros acerca de dos tipos de vinos: tintos y blancos. Nuestro objetivo es, a partir de dos (para poder representarlo debidamente) de los atributos en el *dataset*, predecir la clase entre tinto o blanco de un hipotético vino. Hemos escogido los atributos *alcohol* y *total sulfur dioxide* ya que son los que mayor correlación presentan con la clase, el atributo que queremos predecir. Si representamos los datos de dichos atributos (Figura A.3), vemos que los ejemplos se distribuyen en dos nubes en el espacio, una por cada clase. Precisamente lo que buscamos mediante la regresión logística es un umbral de decisión que nos permita fraccionar el espacio de tal forma que se distingan las dos clases con la condición de coste mínimo.

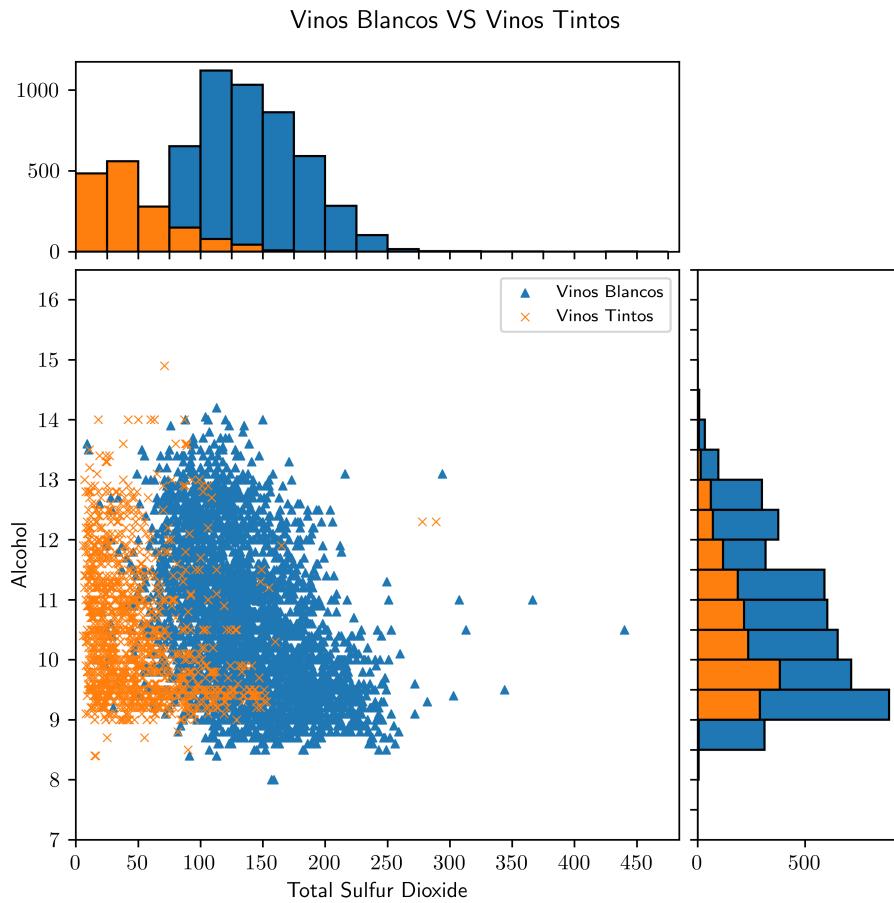


Figura A.3: Dispersión de los pares de valores alcohol - dióxido de azufre total, así como su distribución a lo largo de ambos ejes.

Dado que es visible que ambas nubes de datos se entrelazan, podemos suponer que no llegaremos a tener un coste 0 en nuestra solución, sino que trabajaremos sobre un umbral de coste. Esto se debe a que aunque sí que estaremos clasificando correctamente la mayoría de los datos, habrá una serie de ellos que se presentarán como una excepción a esta norma. Como podemos ver en la Figura A.4b, los números avalan esta tesis, pues es visible que el valor de convergencia del coste asociado a los coeficientes alcanzados después del entrenamiento está por encima de 0. Aprovechamos así para introducir, de forma complementaria, el concepto

²Fuente: <https://archive.ics.uci.edu/ml/datasets/wine+quality>

de *outliers*. Llamamos *outliers* a aquellos ejemplos del *dataset* que de forma evidentemente incumplen la norma que rige el resto. Normalmente esto se debe a imprecisiones en la toma de los datos, o a la poca correlación en los ejemplos. Este segundo caso sería el nuestro, pero esto a su vez se puede atribuir al hecho de que hemos ignorado una serie de atributos con el fin de poder ilustrar el funcionamiento del modelo, y por tanto hemos perdido poder de decisión a la hora de clasificar los ejemplos.

Evolución de los coeficientes en el entrenamiento

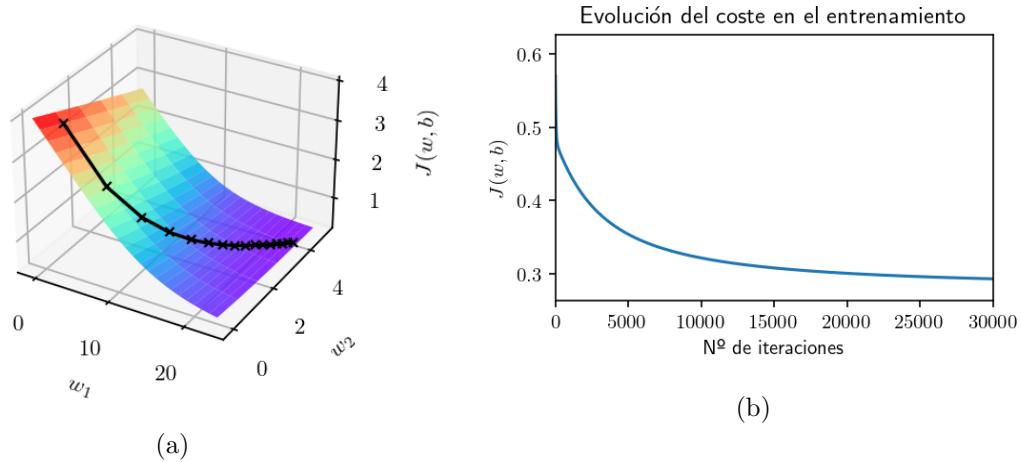


Figura A.4: Evolución de los coeficientes (A.4a), el coste (A.4b) y la recta de la regresión (A.4c) durante el entrenamiento.

Otros modelos más sofisticados, como las máquinas de soporte vectorial (SVM, *Support Vector Machines*), sí que tienen mecanismos para manejar mejor la presencia de estos *outliers*, pero a costa de un entrenamiento mucho más pesado computacionalmente hablando.