UNIVERSIDAD
DE GRANADA

Facultad de Ciencias

Escuela Técnica Superior de Ingeniería
Informática y Telecomunicación

Doble Grado en Ingeniería
Informática y Matemáticas

TRABAJO DE FIN DE GRADO

Generación de música mediante redes neuronales profundas

Presentado por:

Antonio Martín Ruiz

Curso académico 2019-2020

Generación de música mediante redes neuronales profundas

Antonio Martín Ruiz

Antonio Martín Ruiz *Generación de música mediante redes neuronales profundas*.

Trabajo de fin de Grado. Curso académico 2019-2020.

Responsable de tutorización

Francisco Herrera Triguero
Ciencias de la computación e inteligencia artificial

Francisco David Charte Luque
Ciencias de la computación e inteligencia artificial

Doble Grado en Ingeniería Informática y Matemáticas

Facultad de Ciencias

Escuela Técnica Superior de Ingeniería Informática y Telecomunicación

Universidad de Granada

Declaración de originalidad

D./Dña. Antonio Martín Ruiz

Declaro explícitamente que el trabajo presentado como Trabajo de Fin de Grado (TFG), correspondiente al curso académico 2019-2020, es original, entendida esta, en el sentido de que no ha utilizado para la elaboración del trabajo fuentes sin citarlas debidamente.

En Granada a 7 de septiembre de 2020

Fdo: Antonio Martín Ruiz

Índice general

Índice de figuras

Agradecimientos

A mis tutores, por permitirme llevar a cabo este proyecto. A David Charte, por su dedicación. A mis compañeros, cuya ayuda ha sido imprescindible durante estos años y para este trabajo. A José Luis Ramos, por su apoyo en el desarrollo del programa. A quienes me han acompañado durante todo el proceso. A quienes han mostrado su interés en el proyecto y a quienes lo están utilizando para hacer música. A mi familia y amigos.

Resumen

En este trabajo se estudia la aplicación de técnicas del ámbito del aprendizaje profundo en la generación de música. Se enuncia y demuestra el teorema de aproximación universal que caracteriza la capacidad representativa de las redes neuronales, se realiza un estudio de los fundamentos del aprendizaje profundo, las técnicas de aprendizaje profundo para la modelización de datos secuenciales y el aprendizaje de características. Se desarrollan los fundamentos matemáticos del *autoencoder* variacional, basado en inferencia variacional, y se aplica en un modelo para la manipulación y generación de melodías musicales. Por último se desarrolla una herramienta como aplicación de dicho modelo.

Palabras clave: aprendizaje automático, aprendizaje profundo, inferencia variacional, autoencoder variacional, generación de música

Summary

This work studies the application of deep learning techniques to music generation and the development of a software tool as an example of the application of those techniques, covering from the basic theoretical foundations to state of the art models and their application in the real musical composition process.

In the first chapter we study the universal approximation theorem and its proof, which characterizes the representation capacities of neural networks. We review some preliminary results, including the Hahn-Banach theorem and the Riezs representation theorem. After that we prove the main theorem, which states that the set of functions represented by deep neural networks with one hidden layer of arbitrary size and sigmoid activation function are dense in the space of continuos functions over the N-dimensional unit hypercube, hence deep neural neworks with one hidden layer of arbitrary and sigmoid activation function are universal approximators. Then some of the existing generalizations and related results are mentioned, including different activation functions and fixing the hidden layer size but making the network depth arbitrary.

The second chapter covers the basics of deep learning. We contextualize it in the machine learning domain and define its basic model: feedforward deep networks. A description of all the components of the model is providen, including cost function, output units and hidden units. Then we describe its functioning using the forward propagation algorithm and its optimization process which includes various grandient-based methods and the principal algorithm for the computation of the gradient of the network, the backpropagation algorithm.

In the third chapter we discuss the deep learning techniques avaliable for sequencial data handling. We provide some examples of secuential data and its applications. Recurrent neural networks and its most important variants are introduced, including different network architectures, bidirectional recurrent networks and deep recurrent netwokds. We describe the problems they

present with long-term information processing. Subsequently some more recent models to overcome these difficulties are introduced, including LSTM and GRU.

The fourth chapter focuses on representation learning. Background information is discussed, including feature selection and classical representation learning techniques as PCA and others. After that we introduce a deep learning model for representation learning, the autoencoder. A comprehensive description of this model is providen, including its variants and particularities compared to deep feedforward networkds. Subsequently we study the generative version of that model, the variational autoencoder, based on variational inference. An explicit expression of its cost function is calculated, as well as an optimization method.

The fifth chapter introduces MusicVAE, a model based on LSTM and variational autoencoder for the manipulation and generation of musical melodies. We describe its architecture, which consist in a variational autoencoder whose encoder and decoder have the structure of recurrent networkds. Then its training process is described, along with some useful properties derived from its use.

Finally, in the sixth chapter the software tool AutoLoops is described. It consists on a web application for the generation of music melodies in MIDI format. Its development is based on the Magenta library for JavaScript, which includes an API for the MusicVAE pre-trained model. The interface for the program is based on HTML and CSS. Its use and inner working are explained in detail.

Keywords: machine learning, deep learning, variational inference, variational autoencoder, music generation

Introducción

En este trabajo se estudia la aplicación de técnicas del ámbito del aprendizaje profundo en la generación de música y se desarrolla una herramienta software como ejemplo de aplicación de dichas técnicas.

Uno de los grandes desafíos actuales de la Inteligencia Artificial es modelar datos de naturaleza compleja, como el lenguaje, la música o las imágenes. Mediante esta modelización pueden obtenerse nuevas formas de manipular y generar estos datos. Esto puede suponer una herramienta importante para el ámbito artístico.

En la primera mitad del siglo XX se inventa el amplificador para aumentar el volumen al que podían emitir sonido instrumentos musicales como las guitarras. La degradación de estos equipos producía distorsión en la señal, lo cual en principio era un efecto indeseable. Sin embargo muchos artistas tomaron este defecto como herramienta de expresión. La introducción de nuevas herramientas tecnológicas en el ámbito artístico puede suponer efectos insospechados. Es por ello que la implantación de las técnicas de aprendizaje profundo puede llevar a nuevas formas de manipulación y expresión musical.

La naturaleza de los datos musicales es eminentemente secuencial. Ya sea en forma de onda de sonido o de manera más abstracta como la representación en partituras, siempre se trata con secuencias de datos en el tiempo. Por ello el uso de técnicas específicas para el tratamiento de este tipo de datos será imprescindible.

El aprendizaje de representaciones es un campo ampliamente explorado en el aprendizaje automático y supone una de las vías de investigación más importantes en la actualidad. Una representación sencilla de datos complejos, que pueda permitir su manipulación de manera intuitiva, puede resultar una herramienta muy importante para el trabajo de los artistas. Los modelos generativos, además de obtener una representación compacta, permiten la generación de nuevos datos. Esta puede resultar también una herramienta muy útil para la creación artística.

Se marcan por tanto los siguientes objetivos para el trabajo.

1. Estudiar y demostrar que las redes neuronales son aproximadores universales.
2. Estudiar las bases matemáticas del autoencoder variacional.
3. Conocer los fundamentos del aprendizaje profundo y del tratameinto de secuencias.
4. Estudiar el modelo MusicVAE para generación de música.
5. Aplicar dicho modelo en una herramienta software.

En el trabajo se recogen los fundamentos teóricos básicos del aprendizaje profundo, para después revisar las técnicas más relevantes para el modelado de secuencias y el aprendizaje de características. Por último se estudia un modelo de aprendizaje profundo especialmente diseñado para la codificación y generación de melodías musicales, y se realiza la implementación de una herramienta software para su aplicación en el contexto real de la composición musical.

La herramienta permite cargar una melodía, extraer sus características y modificar las mismas para generar melodías nuevas. También permite exportar las nuevas melodías generadas en formato MIDI, facilitando su uso para la producción musical. Su principal pretensión es la de ser aplicable en el trabajo compositivo, y de hecho está ya siendo utilizada por varios artistas para la confección de obras. Algunas de ellas pueden encontrarse en este enlace.

En el primer capítulo de este trabajo se enuncia y demuestra el teorema de aproximación universal, un resultado que caracteriza la capacidad representativa de las redes neuronales. En el segundo capítulo se realiza un estudio de los fundamentos del aprendizaje profundo, describiendo el modelo fundamental, la red prealimentada profunda, y las técnicas que se utilizan para su optimización. En el tercer capítulo se realiza un estudio de las técnicas de aprendizaje profundo para la modelización de datos secuenciales, incluyendo las redes neuronales recurrentes y modelos más actuales. En el cuarto capítulo se repasan técnicas para el aprendizaje de características. Se hace especial hincapié en las técnicas del ámbito del aprendizaje profundo y de entre estas se desarrolla el *autoencoder* variacional, que hace uso de la inferencia variacional para generar nuevos datos. En el sexto capítulo se expone el modelo

MusicVAE, basado en el *autoencoder* variacional y cuyo objetivo es la manipulación y generación de melodías musicales. Por último en el séptimo capítulo se desarrolla la herramienta AutoLoops como aplicación de MusicVAE.

De entre las diversas fuentes de información consultadas, aquellas más fundamentales para el trabajo son:

- *Deep Learning*, por Goodfellow, Bengio y Courville. Es la principal referencia en el contenido sobre aprendizaje automático y aprendizaje profundo.
- *Approximation by superpositions of a sigmoidal function*, por Cybenko. Empleado para la demostración del teorema de aproximación universal.
- *Auto-encoding variational bayes*, por Kingma, Diederik y Welling. Contiene la descripción del modelo del *autoencoder* variacional.
- *A hierarchical latent vector model for learning long-term structure in music*, por Roberts, Adam, Engel, Jesse, Raffel, Colin, Hawthorne, Curtis, Eck y Douglas. Desarrolla el modelo MusicVAE.

1 Aproximación por superposición de funciones sigmoidales

Las redes neuronales, cuya estructura y funcionamiento se explica en Capítulo 2, son un método para la representación de funciones de variable real n-dimensional, $\mathbf{x} \in \mathbb{R}^n$, mediante la aplicación sucesiva de transformaciones de la forma

$$\sum_{j=1}^{n} \alpha_j \sigma(\mathbf{y}_j^T \mathbf{x} + \theta_j),$$

donde $\mathbf{y}_j \in \mathbb{R}^n$ y $\alpha_j, \theta_j \in \mathbb{R}$ son fijas. La función de una variable σ elegida depende del contexto de la aplicación.

Una de los tipos de funciones σ más utilizados es el de las funciones sigmoidales, es decir, aquellas que cumplen

$$\lim_{t \to -\infty} \sigma(t) = 0 \quad \text{y} \quad \lim_{t \to +\infty} \sigma(t) = 1,$$

continuas. El objetivo de la sección es demostrar que las funciones de la forma $\sum_{j=1}^{n} \alpha_j \sigma(\mathbf{y}_j^T \mathbf{x} + \theta_j)$, aquellas que representan una red neuronal con una única capa oculta de anchura arbitraria, con σ sigmoidal y continua son densas en el espacio $C(I^n)$ de las funciones continuas en el $I^n = [0,1]^n$. Este resultado es el llamado teorema de aproximación universal [Cyb89] e implica que las redes neuronales con función de activación sigmoidal y al menos una capa profunda pueden aproximar cualquier función de $C(I^n)$. Este resultado no es constructivo, por lo que no proporciona un método para obtener dicha aproximación. Los métodos más comunes para esta tarea serán revisados en capítulos posteriores.

Previo a la demostración del teorema se repasan algunos resultados necesarios durante dicha demostración. Por último se revisan otros resultados relacionados con la aproximación de funciones mediante redes neuronales.

1.1. Definiciones y resultados previos

En este apartado se realizan algunas definiciones previas y se repasan los dos resultados principales que se aplicarán en la demostración del teorema de aproximación universal: el teorema de Hahn-Banach y el teorema de representación de Riesz.

Definición 1.1. Dada una σ-álgebra $\mathcal{M}$ una *medida con signo* μ sobre $\mathcal{M}$ es una función de conjunto $\mu : \mathcal{M} \to [-\infty, +\infty]$ σ-aditiva.

Definición 1.2. Una medida se dice *de Borel* si está definida sobre una σ-álgebra de Borel, es decir, la engendrada por los abiertos del espacio.

Definición 1.3. Una *medida con signo regular de Borel* sobre una σ-álgebra $\mathcal{M}$ es una medida con signo que cumple

$$\mu(E) = \text{ínf}\{\mu(V) : V \supset E, V \text{ abierto}\} = \sup\{\mu(C) : C \subset E, V \text{ cerrado}\}$$

para todo conjunto de Borel $E \in \mathcal{M}$.

Se denota por $M(I^n)$ al conjunto de medidas finitas con signo regulares de Borel sobre I^n.

1.1.1. Teorema de Hahn-Banach

Pasamos ahora a recordar uno de los resultados principales que se utilizan en la demostración, el teorema de Hahn-Banach. Se realiza primeramente una definición previa.

Definición 1.4. [Ash14] Un *funcional sublineal* en un espacio vectorial X es una aplicación $\varphi : X \to \mathbb{R}$ que verifica la desigualdad triangular y es homogénea por homotecias, es decir

$$\varphi(x + y) \leq \varphi(x) + \varphi(y)$$

$$\varphi(rx) = r\varphi(x)$$

para todo $x, y \in X$ y para todo $r \in \mathbb{R}^+$.

Con esto podemos ya enunciar el teorema.

Teorema 1.1. *Sea X un espacio vectorial y φ un funcional sublineal en X. Sea M un subespacio de X y g un funcional lineal en M, dominado por φ, es decir,*

$$\operatorname{Re} g(y) \leq \varphi(y) \quad \forall y \in M$$

Entonces existe un funcional lineal f en X que extiende a g y está dominado por φ, es decir,

$$f(y) = g(y) \quad \forall y \in M, \quad \operatorname{Re} f(x) \leq \varphi(x) \quad \forall x \in X.$$

La aplicación del teorema no será directa. Utilizaremos un corolario que se demuestra a partir de un teorema que es resultado directo del teorema de Hahn-Banach, el teorema de extensión de Hahn-Banach.

Teorema 1.2. *Sea X un espacio normado, M un subespacio de X y $g \in M^*$. Entonces existe $f \in X^*$ tal que f extiende a g y verifica que $\|f\| = \|g\|$.*

Demostración. Definiendo $\varphi(x) = \|g\|\|x\|$ para todo $x \in X$, φ es una seminorma en X, y de hecho es una normal salvo en el caso $g = 0$. La continuidad de g implica que $\operatorname{Re} g(y) \leq |g(y)| \leq \varphi(y)$ para todo $y \in M$. El teorema de Hahn-Banach (Teorema 1.1) asegura la existencia de un funcional lineal f en X que verifica

$$f(y) = g(y) \quad \forall y \in M,$$

$$|f(x)| \leq \varphi(x) = \|g\|\|x\| \quad \forall x \in X.$$

$f \in X^*$ con $\|f\| \leq \|g\|$ pero además f extiende a g, luego también $\|f\| \geq \|g\|$. Por lo tanto $\|f\| = \|g\|$.

□

A la extensión f se le llama extensión de Hahn-Banach de g. Puede ya demostrarse el resultado concreto que se utilizará en la demostración del teorema de aproximación universal.

Corolario 1.1. *Sea X un espacio normado, M un subespacio de X. Si $x_0 \notin \overline{M}$, existe un $f \in X^*$ tal que $f(x) = 0\ \forall x \in M$, $f(x_0) = 1$, y $\|f\| = \frac{1}{d}$, donde d es la distancia de x_0 a M.*

Demostración. Sea $N = L(M \cup \{x_0\})$ el conjunto de todos los ejemplos $y = x + ax_0$ con $x \in M$, $a \in \mathbb{C}$. Como $x \notin \overline{M}$, el valor de a está determinado por el valor de y. Se define f sobre N como $f(x + ax_0) = a$. f es lineal, y veamos que $\|f\| = \frac{1}{d}$.

$$\|f\| = \sup\left\{\frac{|f(y)|}{\|y\|} : y \in N, y \neq 0\right\} =$$
$$\sup\left\{\frac{|a|}{\|x + ax_0\|} : a \in \mathbb{C}, x \neq 0 \ \ o \ \ a \neq 0\right\} =$$
$$\sup\left\{\frac{|a|}{\|x + ax_0\|} : a \in \mathbb{C}, x \neq 0 \ \ o \ \ a \neq 0\right\}$$

ya que $f(y) = 0$ cuando $a = 0$. Podemos reescribir

$$\frac{|a|}{\|x + ax_0\|} = \frac{1}{\|x_0 + \frac{x}{a}\|} = \frac{1}{\|x_0 - z\|}$$

para algún $z \in M$. Por lo tanto $\|f\| = (\text{ínf}\{\|x_0 - z\|\} : z \in M)^{-1} = \frac{1}{d}$. Basta aplicar el teorema de extensión de Hahn-Banach (Teorema 1.2) sobre f para obtener el funcional que se quería en M. □

1.1.2. Teorema de representación de Riesz

El otro resultado fundamental para la demostración del teorema de aproximación universal es el teorema de representación de Riesz. Se realizan primero unas definiciones previas.

Definición 1.5. [Ash14] Dada μ una norma con signo sobre la σ-álgebra $\mathcal{M}$, se define su *variación superior* como

$$\mu^+(A) = \sup\{\mu(B) : B \in \mathcal{M}, B \subset A\}.$$

Definición 1.6. Dada μ una norma con signo sobre la σ-álgebra $\mathcal{M}$, se define su *variación inferior* como

$$\mu^-(A) = \text{ínf}\{\mu(B) : B \in \mathcal{M}, B \subset A\}.$$

Definición 1.7. Dada μ una norma con signo sobre la σ-álgebra $\mathcal{M}$, se define su *variación total* como

$$|\mu|(A) = \mu^+(A) + \mu^-(A).$$

Podemos ya enunciar el teorema de representación de Riesz, que será aplicado directamente durante la demostración del teorema de aproximación universal.

Teorema 1.3. *[Rud87] Sea X un espacio de Hausdorff localmente compacto, y sea T un funcional lineal acotado sobre $C_0(X)$. Entonces existe una σ-álgebra $\mathcal{M}$ en X que contiene todos los conjuntos de Borel en X y existe una única medida con signo regular μ sobre $\mathcal{M}$ tal que*

$$T(f) = \int_X f d\mu$$

para cada $f \in C(X)$. La norma del funcional T es la variación total de μ:

$$\|T\| = |\mu|(X).$$

1.2. Teorema de aproximación universal

Conociendo los resultados previos puede ya enunciarse y demostrarse el teorema de aproximación universal [Cyb89]. Para ello se realizará la definición de función discriminatoria y se probará una primera versión del teorema para estas funciones.

Definición 1.8. Una función $\sigma : \mathbb{R} \to \mathbb{R}$ es *discriminatoria* si, cuando para una medida $\mu \in M(I^n)$ se tiene que

$$\int_{I^n} \sigma(\mathbf{y}^T \mathbf{x} + \theta) d\mu(\mathbf{x}) = 0$$

para todo $\mathbf{y} \in \mathbb{R}^n$, $\theta \in \mathbb{R}$ implica que $\mu = 0$.

Teorema 1.4. *Sea σ una función continua discriminatoria. Entonces el conjunto de funciones*

$$S = \left\{ g : g(\boldsymbol{x}) = \sum_{j=1}^{k} \alpha_j \sigma(\boldsymbol{y}_j^T \boldsymbol{x} + \theta_j) \right\}$$

es denso en $C(I^n)$, es decir, dada una $f \in C(I^n)$ y $\varepsilon > 0$, existe una $g \in S$ tal que

$$|g(\boldsymbol{x}) - f(\boldsymbol{x})| < \varepsilon$$

para todo $\boldsymbol{x} \in I^n$.

Demostración. S es un subespacio lineal de $C(I^n)$. Afirmemos que $\overline{S} = C(I^n)$.

Si, por el contrario, $\overline{S} \subsetneq C(I^n)$, aplicando el Corolario 1.1 existe un funcional lineal acotado $F \neq 0$ sobre $C(I^n)$ tal que

$$F(\overline{S}) = F(S) = 0.$$

Por el teorema de representación de Riesz (Teorema 1.3) este funcional lineal es de la forma

$$F(h) = \int_{I^n} h(\mathbf{x}) d\mu(\mathbf{x})$$

para alguna medida $\mu \in M(I^n)$ y para todo $h \in C(I^n)$. En particular, como $\sigma(\mathbf{y}^T\mathbf{x} + \theta) \in \overline{S}$ para todo $y \in \mathbb{R}^n$ y para todo $\theta \in \mathbb{R}$, debe cumplirse que

$$\int_{I_n} \sigma(\mathbf{y}^T\mathbf{x} + \theta) d\mu(\mathbf{x}) = 0$$

para todo $\mathbf{y} \in I^n$ y para todo $\theta \in \mathbb{R}$. Puesto que σ es discriminatoria, se tiene que $\mu = 0$, lo cual es una contradicción con que $F \neq 0$. Por tanto el subespacio S es denso en $C(I_n)$. □

Una vez demostrado el teorema para funciones discriminatorias queda ver que las funciones sigmoidales continuas en I^n son discriminatorias. La demostración se realizará para un conjunto de funciones más grande en el que las funciones sigmoidales continuas están incluidas.

Lema 1.1. *Una función sigmoidal medible y acotada $\sigma : \mathbb{R} \to \mathbb{R}$ es discriminatoria. En particular, las funciones sigmoidales continuas son discriminatorias.*

Demostración. Sea $\sigma_\lambda(\mathbf{x}) = \sigma(\lambda(\mathbf{y}^T\mathbf{x} + \theta) + \phi)$. Entonces, para cualquier $\mathbf{x}$, $\mathbf{y}$, θ, ϕ se tiene que

$$\sigma(\lambda(\mathbf{y}^T\mathbf{x} + \theta) + \phi) \begin{cases} \to 1 & si \quad \mathbf{y}^T\mathbf{x} + \theta > 0 \text{ cuando } \lambda \to +\infty, \\ \to 0 & si \quad \mathbf{y}^T\mathbf{x} + \theta < 0 \text{ cuando } \lambda \to +\infty, \\ = \sigma(\phi) & si \quad \mathbf{y}^T\mathbf{x} + \theta = 0 \text{ para todo } \lambda. \end{cases}$$

Por lo tanto la función $\sigma_\lambda(\mathbf{x}) = \sigma(\lambda(\mathbf{y}^T\mathbf{x} + \theta) + \phi)$ converge puntualmente a la función

$$\gamma(\mathbf{x}) = \begin{cases} 1 & si \quad \mathbf{y}^T\mathbf{x} + \theta > 0 \\ 0 & si \quad \mathbf{y}^T\mathbf{x} + \theta < 0 \\ \sigma(\phi) & si \quad \mathbf{y}^T\mathbf{x} + \theta = 0 \end{cases}$$

cuando $\lambda \to +\infty$.

Supongamos que $\int_{I^n} \sigma(\lambda(\mathbf{y}^T\mathbf{x}+\theta)+\phi)d\mu(\mathbf{x}) = 0$ para una medida $\mu \in M(I^n)$ y comprobemos que $\mu = 0$. Sean $\Pi_{\mathbf{y},\theta} = \{\mathbf{x}|\mathbf{y}^T\mathbf{x}+\theta > 0\}$ y $H_{\mathbf{y},\theta} = \{\mathbf{x}|\mathbf{y}^T\mathbf{x}+\theta > 0\}$, y $\mu \in M(I^n)$. Se tiene que

$$\lim_{\lambda\to\infty} \int_{I^n} \sigma(\lambda(\mathbf{y}^T\mathbf{x}+\theta)+\phi)d\mu(\mathbf{x}) = \lim_{\lambda\to\infty} 0 = 0$$

y por el teorema de convergencia dominada de Lebesgue

$$0 = \lim_{\lambda\to\infty} \int_{I^n} \sigma(\lambda(\mathbf{y}^T\mathbf{x}+\theta)+\phi)d\mu(\mathbf{x}) = \int_{I^n} \lim_{\lambda\to\infty} \sigma(\lambda(\mathbf{y}^T\mathbf{x}+\theta)+\phi)d\mu(\mathbf{x}) =$$

$$\int_{I^n} \gamma(\mathbf{x})d\mu(\mathbf{x}) = \sigma(\phi)\mu(\Pi_{\mathbf{y},\theta}) + \mu(H_{\mathbf{y},\theta})$$

para todo θ, ϕ, $\mathbf{y}$. Veamos que si $\mu(\Pi_{\mathbf{y},\theta}) + \mu(H_{\mathbf{y},\theta}) = 0$ para todo $\mathbf{y}$ y para todo θ entonces $\mu = 0$. Fijado $\mathbf{y}$, para una función medible acotada h se define el funcional lineal F como

$$F(h) = \int_{I^n} h(\mathbf{y}^T\mathbf{x})d\mu(\mathbf{x}).$$

F es un funcional sobre $L^\infty(\mathbb{R})$ y acotado por ser μ una medida con signo finita [Hal76]. Tomando como h la función indicadora del intervalo $[\theta, +\infty)$ ($h(u) = 1$ si $u \geq \theta$, $h(u) = 0$ si $u < \theta$) se obtiene

$$F(h) = \int_{I^n} h(\mathbf{y}^T\mathbf{x})d\mu(\mathbf{x}) = \mu(\Pi_{\mathbf{y},-\theta}) + \mu(H_{\mathbf{y},-\theta}) = 0.$$

El mismo resultado se obtiene tomando la función indicadora en el intervalo $(\theta, +\infty)$. Por la linearidad del operador, $F(h) = 0$ para la función indicadora de cualquier intervalo, y por tanto para cualquier función simple (sumas de funciones indicadoras de intervalos). Puesto que las funciones simples son densas en $L^\infty(\mathbb{R})$, $F = 0$.

En particular para las funciones acotadas y medibles $s(u) = sin(m \cdot u)$ y $c(u) = cos(m \cdot u)$ se tiene

$$F(s+ic) = \int_{I^n} (\cos(m^T x) + i\sin(m^T x))d\mu(x) = \int_{I^n} exp(im^T x)d\mu(x) = 0$$

para todo m. Entonces la transformada de Fourier de μ es nula e igualmente debe serlo μ [Duo03]. Por lo tanto σ es discriminatoria. □

Habiendo probado los dos resultados previos, la demostración del teorema de aproximación universal es directa.

Teorema 1.5. *Sea σ una función sigmoidal continua. Entonces el conjunto de funciones*

$$S = \left\{ g : g(\boldsymbol{x}) = \sum_{j=1}^{k} \alpha_j \sigma(\boldsymbol{y}_j^T \boldsymbol{x} + \theta_j) \right\}$$

es denso en $C(I^n)$, es decir, dada una $f \in C(I^n)$ y $\varepsilon > 0$, existe una $g \in S$ tal que

$$|g(\boldsymbol{x}) - f(\boldsymbol{x})| < \varepsilon$$

para todo $\boldsymbol{x} \in I^n$.

Demostración. Basta combinar el Teorema 1.4 con Lema 1.1 teniendo en cuenta que las funciones sigmoidales continuas satisfacen las condiciones del lema. □

1.3. Otros resultados de aproximación

Existen otros resultados sobre la capacidad de representación de las redes neuronales que generalizan y extienden el teorema de representación universal. Kurt Hornik generaliza el resultado para redes con funciones σ acotadas y no constantes [HSW+89], posteriormente para funciones Riemann-integrables no polinomiales [Hor91].

Todos estos resultados contemplan redes neuronales de una sola capa oculta con anchura arbitraria. Existen otros más recientes en los que el factor indeterminado es la profundidad de la red. Una red neuronal con una anchura fijada de $n + 4$ donde n es el tamaño de la entrada y con profundidad arbitraria y función de activación ReLU (definidas en Subsubsección 2.1.3.1) es también un aproximador universal [LPW+17].

Ninguno de los resultados mencionados en esta sección es constructivo. En el siguiente capítulo se estudiarán los métodos utilizados para la optimización de los parámetros de las funciones vistas en este.

2 Aprendizaje profundo

Un algoritmo de aprendizaje automático o *machine learning* es un algoritmo capaz de aprender de datos. Según [M+97] un programa aprende de la experiencia E con respecto a una tarea T y una medida de rendimiento P si su rendimiento en la tarea T, medido por P, mejora con la experiencia E. Podemos aplicar este tipo de algoritmos en problemas para los cuales no tenemos una solución analítica, pero sí contamos con datos con los que construir una solución empírica.

Algunos ejemplos de problemas para los que se ha aplicado el aprendizaje automático son problemas financieros como detección de fraude o análisis de riesgo, aplicaciones médicas como asistencia al diagnóstico de enfermedades, detección y clasificación de *spam* o reconocimiento de voz.

En general para este tipo de problemas tendremos los siguientes componentes [AMMIL12]:

- Un espacio de entradas $\mathcal{X}$ con todas las posibles entadas del problema. Cada ejemplo que se pretenda resolver será un $\mathbf{x} \in \mathcal{X}$.
- Un espacio de salidas $\mathcal{Y}$ con todas las posibles soluciones.
- Una función objetivo $f^* : \mathcal{X} \to \mathcal{Y}$ desconocida, que lleva cada $\mathbf{x} \in \mathcal{X}$ en su solución $y \in \mathcal{Y}$ correcta.
- Un conjunto de entrenamiento $\mathcal{D}$, la experiencia anteriormente citada, formado por ejemplos de datos del problema, que pueden estar resueltos o no.
- Un conjunto de hipótesis $\mathcal{H}$ formado por aplicaciones de $\mathcal{X}$ en $\mathcal{Y}$.
- Un algoritmo de aprendizaje que utiliza el conjunto $\mathcal{D}$ para elegir de $\mathcal{H}$ una aplicación $f : \mathcal{X} \to \mathcal{Y}$ que aproxime f^*.

Las tareas y experiencias pueden ser muy diversas, lo que nos permite diferenciar varios tipos de algoritmos. Según el tipo de experiencia, en general podemos diferenciar entre aprendizaje supervisado cuando al algoritmo se le

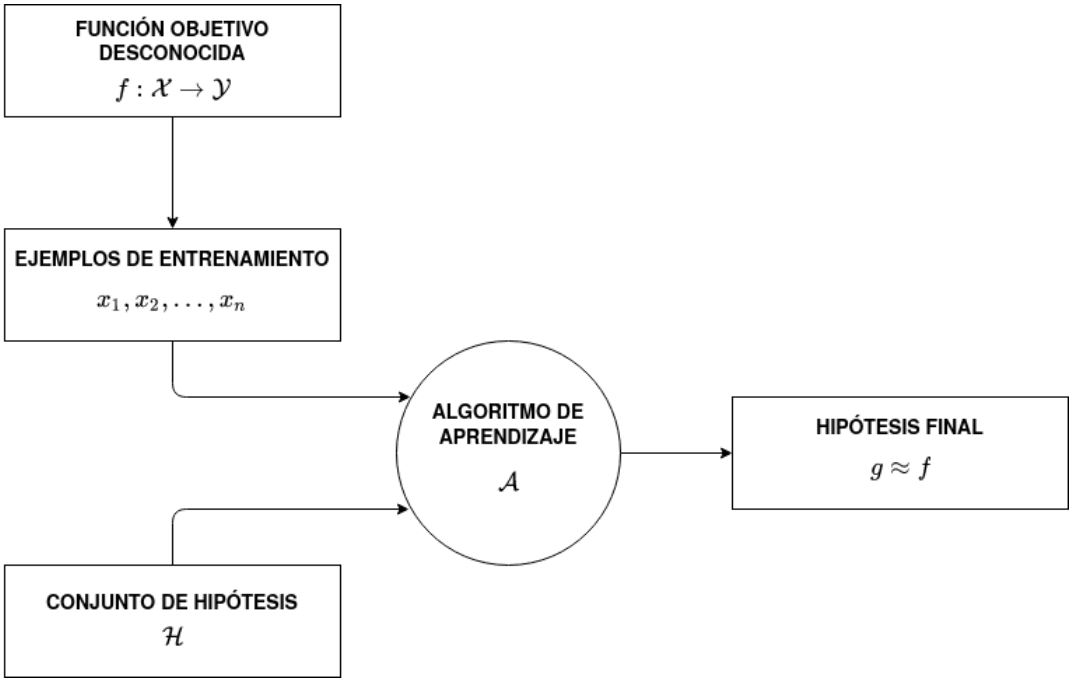

Figura 2.1: Componentes de un problema de aprendizaje automático

proporciona un conjunto de ejemplos para los cuales la tarea está ya resuelta o no supervisado cuando el algoritmo cuenta solo con ejemplos que no están resueltos. Cuando se trabaja con ejemplos resueltos y sin resolver se trata de aprendizaje semi-supervisado, y cuando no se proporciona una respuesta a cada ejemplo pero sí se asigna una puntuación a cada posible respuesta se trata de aprendizaje por refuerzo. Algunas de las tareas a las que pueden aplicarse son:

- Clasificación: el programa debe deducir a qué clase pertenece cada instancia de los datos.
- Regresión: el programa debe asignar un valor real a cada instancia.
- Síntesis y muestreo: el programa debe generar nuevos ejemplos parecidos a los usados para su entrenamiento.

Una primera aproximación a la solución de problemas de aprendizaje son los modelos lineales, como el de regresión lineal o regresión logística, modelos cuyo conjunto de hipótesis es de funciones lineales sobre el conjunto de entrada. Su principal desventaja es su limitación a la hora de aproximar funciones

no lineales, ya que no son capaces de expresar las posibles interacciones entre variables.

Existen también modelos no lineales, entre los que se encuentran los modelos basados en árboles de decisióno modelos basando en distancias entre ejemplos como los k vecinos más cercanos. Estos modelos sí permiten modelar relaciones no lineales entre los datos.

El aprendizaje profundo o *deep learning* es una rama del aprendizaje automático basada en el uso de redes neuronales. Surge en parte como respuesta al problema de los modelos lineales, ya que permite la aproximación de funciones no lineales mediante el aprendizaje de sucesivas transformaciones no lineales de los datos de entrada. En este apartado se introducen estos modelos y su entrenamiento.

2.1. Redes neuronales prealimentadas

Las redes prealimentadas profundas, redes neuronales profundas o perceptrones multicapa, *deep feedforward networks*, *feedforward neural networks* o *multilayer perceptron* (MLP) en inglés, son el modelo canónico del aprendizaje profundo [AMMIL12]. Su objetivo es aproximar la función f^* mediante una aplicación $f(\mathbf{x};\theta)$ y aprendiendo el valor de los parámetros θ que permita un mejor resultado.

Se llaman redes ya que suelen representarse componiendo varias funciones diferentes. El modelo puede asociarse con un grafo dirigido acíclico que describe cómo se componen las funciones. Un ejemplo de esta estructura es, teniendo tres funciones vectoriales f_1, f_2 y f_3, componerlas de la forma $f(\mathbf{x}) = f_3(f_2(f_1(\mathbf{x})))$. En este caso la función f_i se llama capa i-ésima de la red. La longitud de la cadena es la profundidad de la red, de donde surge la nomeclatura redes profundas. La última capa de la red se llama capa de salida.

Durante el entrenamiento se intenta conseguir que $f(\mathbf{x})$ se aproxime a $f^*(\mathbf{x})$. Los datos del conjunto de entrenamiento ofrecen ejemplos, aproximados y ruidosos, de la función f^* evaluada en diferentes $\mathbf{x} \in \mathcal{X}$. Cada ejemplo $\mathbf{x}$ está normalmente acompañado de una salida deseada $\mathbf{y} \approx f^*(\mathbf{x})$. Así, los ejemplos especifican qué resultado debe ofrecer la capa de salida para cada uno de ellos. El comportamiento del resto de capas no está especificado, y es

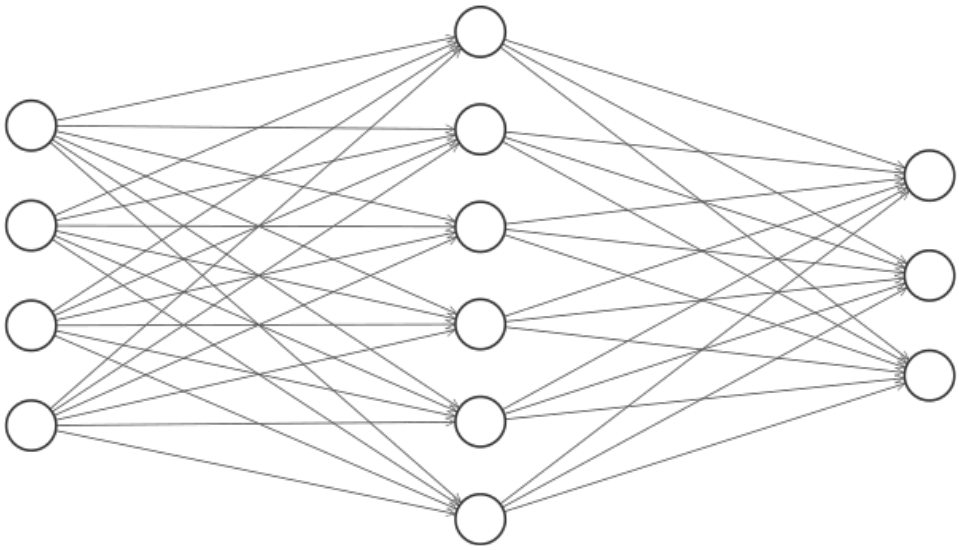

Figura 2.2: Ejemplo de red neuronal prealimentada de profundidad 3

el algoritmo de aprendizaje el que debe configurarlo para obtener la mejor aproximación de f^*.

Que el grafo sea acíclico implica que no existe retroalimentación con la salida de ninguna de las unidades de la red. Cuando el modelo se extiende para permitir la retroalimentación se trata de una red neuronal recurrente, que veremos posteriormente.

El componente básico de cada una de las capas, al que llamamos unidad, consiste en una función de $\mathbb{R}^n$ en $\mathbb{R}$, donde n es el tamaño de la capa anterior. Cada unidad recibe la salida de las unidades de la capa anterior y calcula un valor que devuelve, su valor de activación. Este comportamiento se inspira en las neuronas biológicas [Izeo8], por eso este tipo de modelos recibe el nombre de red neuronal.

Puesto que el objetivo es aproximar funciones no lineales, no basta con que las transformaciones que aplican estas unidades sean transformaciones lineales de los datos de entrada. De ser así, podríamos resumir el comportamiento de la red al completo en una única transformación lineal. Se busca por tanto el aprendizaje de una transformación no lineal ϕ de entre una clase de funciones

parametrizadas. Se define así un modelo del tipo

$$y = f(\mathbf{x}; \theta, \mathbf{w}) = \phi(\mathbf{x}; \theta)^T \mathbf{w}$$

donde θ es un vector de parámetros que permite seleccionar ϕ de entre la familia que elegimos y w se utiliza para aplicar la transformación de los datos obtenida en la salida deseada. La clase de funciones de entre los que el modelo busca la aproximación, el conjunto de hipótesis, se determina al seleccionar la estructura de la red y los tipos de unidades ocultas y de salida.

2.1.1. Función de coste

La mayoría de diseños de redes neuronales se realizan definiendo la distribución $P(y|x;\theta)$ y aplicando el principio de máxima verosimilitud. Si $\mathbf{X} = \{\mathbf{x}_1, ..., \mathbf{x}_n\}$ representa los ejemplos, $\mathbf{Y} = \{\mathbf{y}_1, ..., \mathbf{y}_n\}$ sus soluciones, el estimador de máxima verosimilitud vendrá dado por

$$\theta_{ML} = \arg\max_{\theta} P(\mathbf{Y}|\mathbf{X};\theta).$$

Asumiendo que los ejemplos son independientes e idénticamente distribuidos

$$\theta_{ML} = \arg\max_{\theta} \prod_{i=1}^{n} P(\mathbf{y}_i|\mathbf{x}_i;\theta).$$

Puesto que el logaritmo es una función estrictamente creciente, tomar el logaritmo nos da un problema de optimización equivalente, por lo que

$$\theta_{ML} = \arg\max_{\theta} \sum_{i=1}^{n} \log P(\mathbf{y}_i|\mathbf{x}_i;\theta).$$

Escalar la función tampoco cambia cuál es el máximo, por lo que podemos dividir entre n para obtener una expresión respecto de la esperanza de la distribución empírica $\hat{P}_{\text{data}}$ de los datos de entrenamiento

$$\theta_{ML} = \frac{1}{n} \arg\max_{\theta} \sum_{i=1}^{n} \log P(\mathbf{y}_i|\mathbf{x}_i;\theta) = \mathbb{E}_{\mathbf{x},\mathbf{y}\sim\hat{P}_{\text{data}}} \log P(\mathbf{y}|\mathbf{x}).$$

Teniendo esto, podemos definir la función de coste como

$$J(\theta) = -\mathbb{E}_{\mathbf{x},\mathbf{y}\sim\hat{P}_{\text{data}}} \log P(\mathbf{y}|\mathbf{x}).$$

La expresión concreta de la función de coste cambia con el modelo, dependiendo de la distribución P. Normalmente también se añade un término de regularización. Con este, la función de coste es de la forma

$$J(\theta) = -\mathbb{E}_{\mathbf{x},\mathbf{y}\sim\hat{p}_{\text{data}}} \log P(\mathbf{y}|\mathbf{x}) + \lambda\Omega(\theta).$$

2.1.2. Unidades de salida

Siguiendo el método previo, la elección de la representación de la salida de una red prealimentada determina la expresión de su función de coste. En esta sección estudiamos qué tipo de unidades suelen utilizarse como salida de la red. Para ello supondremos que las unidades ocultas nos proporcionan un vector de características $\mathbf{h} = f(\mathbf{x};\theta)$. La capa de salida transforma estas características para completar la tarea que la red debe cumplir.

2.1.2.1. Unidades lineales

Dadas las características $\mathbf{h}$, una capa con unidades lineales produce un vector $\mathbf{y} = \mathbf{W}^T\mathbf{h} + \mathbf{b}$, sin transformaciones no lineales. Suelen usarse para producir la media de una distribución condicional normal

$$p(\mathbf{y}|\mathbf{x}) = \mathcal{N}(\mathbf{y}^*;\mathbf{y},I).$$

En este caso obtener la máxima verosimilitud equivale a minimizar el error cuadrático medio.

2.1.2.2. Unidades con activación sigmoidal

Muchas tareas requieren predecir una variable binaria y. Un ejemplo de ello son los problemas de clasificación con dos clases. En este caso la aproximación del método de máxima verosimilitud es definir una distribución de Bernouilli sobre y condicionada a $\mathbf{x}$. La red neuronal tendrá entonces que predecir $P(y = 1|\mathbf{x})$. El número generado por la red, por tanto, estará en el intervalo $[0, 1]$.

Para un correcto entrenamiento mediante técnicas basadas en gradiente es necesario que no genere un gradiente nulo o muy cercano a 0 cuando el modelo no se acerque a la solución. Para ello se aplica una función de activación

sigmoidal a la salida, como la función logística

$$y = \sigma(\mathbf{w}^T\mathbf{h} + b) = \frac{1}{1 + e^{-(\mathbf{w}^T\mathbf{h}+b)}}.$$

Definamos entonces una distribución de probabilidad sobre y usando el valor de $z = \mathbf{w}^T\mathbf{h} + b$. La elección de la función de activación sigmoidal puede justificarse construyendo una distribución de probabilidad no normalizada $\hat{P}$(y), cuya suma no es 1, que supondremos log-lineal en z e y. Entonces podemos exponenciar para obtener una distribución de probabilidad válida y normalizando obtendremos una distibución de Bernouilli controlada por una transformación sigmoidal de z:

$$\log \hat{P}(y) = yz$$
$$\hat{P}(y) = e^{yz}$$

$$P(y) = \frac{e^{yz}}{e^{0z} + e^{1z}} = \frac{e^{yz}}{1 + e^z} = \sigma((2y - 1)z).$$

De esta manera la función de coste es

$$J(\theta) = -\log P(y|\mathbf{x}) = -\log \sigma((2y - 1)z).$$

Esta función solo se satura cuando $y = 1$ y z es muy positivo o cuando $y = 0$ y z es muy negativo, es decir, cuando el modelo alcanza la respuesta. Esta función está bien definida ya que cualquier función sigmoidal toma valores en $(0, 1)$, y por tanto el logaritmo es finito.

2.1.2.3. Unidades softmax

La función de activación softmax puede utilizarse para representar una distribución de probabilidad sobre una variable discreta con n valores posibles. Este es el caso de los problemas de clasificación con más de dos clases. Puede interpretarse por tanto como una generalización para las funciones de activación sigmoidal.

En este caso se necesita generar un vector $\mathbf{y}$, con $y_i = P(y = i|\mathbf{x})$. Cada elemento de $\mathbf{y}$ debe estar en $[0, 1]$, y la suma de todos ellos debe ser 1 para representar una distribución de probabilidad válida.

Siguiendo el razonamiento anterior, una capa lineal predice probabilidades logarítmicas no normalizadas,

$$\mathbf{z} = \mathbf{W}^T\mathbf{h} + b$$

donde $z_i = \log \hat{P}(y = i|\mathbf{x})$. La función softmax, definida componente a componente como

$$\text{softmax}(\mathbf{z})_i = \frac{e^{z_i}}{\sum_j e^{z_j}},$$

exponencia y normaliza $\mathbf{z}$ para obtener la $\mathbf{y}$ deseada. Igual que en el caso anterior, la función de coste puede obtenerse aplicando el principio de máxima verosimilitud.

2.1.3. Unidades ocultas

Cualquiera de los tipos de unidad anterior puede utilizarse en una capa oculta, pero existen más diseños. En este apartado se describen algunos de los tipos más comunes. Por lo general, las capas ocultas aceptan un vector de entrada $\mathbf{x}$ sobre el que realizan una transformación afín $\mathbf{z} = \mathbf{W}^T\mathbf{x} + \mathbf{b}$, donde la matriz $\mathbf{W}$ se llama matriz de pesos y el vector $\mathbf{b}$ vector de sesgo, y después aplican una transformación no lineal $g(\mathbf{z})$, llamada función de activación. La elección de esta transformación es la que marca el tipo de unidad que se selecciona.

Algunas de las unidades ocultas aquí expuestas no son diferenciables en todos los puntos. Podría parecer que esto las invalida para el aprendizaje mediante métodos de gradiente. En la práctica el descenso del gradiente funciona bien con estas funciones de activación ya que no se suele llegar a un mínimo de la función de coste, sino reducir su valor. Por ello no se obtendrá un valor en el que el gradiente sea nulo, y es aceptable que el mínimo de la función de coste corresponda a valores en los que el gradiente no está definido. Por lo tanto, en la práctica, pueden no tenerse en cuenta los puntos en los que las funciones aquí descritas no son diferenciables.

2.1.3.1. Unidades lineales rectificadas

Las unidades lineales rectificadas (*ReLU*) utilizan la función de activación $g(\mathbf{z})_i = \text{máx}\{0, z_i\}$.

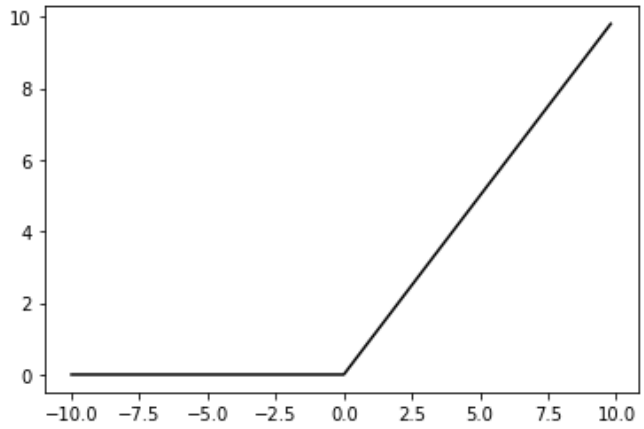

Figura 2.3: Gráfica de la función *ReLU*

Estas unidades son fáciles de optimizar por su parecido a las unidades lineales. La única diferencia con estas es que las rectificadas se anulan en la mitad de su dominio. Existen múltiples generalizaciones de este tipo de unidades:

- Rectificación mediante valor absoluto: $g(\mathbf{z})_i = |z_i|$.
- Leaky *ReLU*: $g(\mathbf{z})_i = \text{máx}\{0, z_i\} + \alpha\, \text{mín}\{0, z_i\}$ con un α fijo, normalmente $0,01$.
- *ReLU* paramétríca: $g(\mathbf{z})_i = \text{máx}\{0, z_i\} + \alpha_i\, \text{mín}\{0, z_i\}$ con α_i un parámetro optimizable.

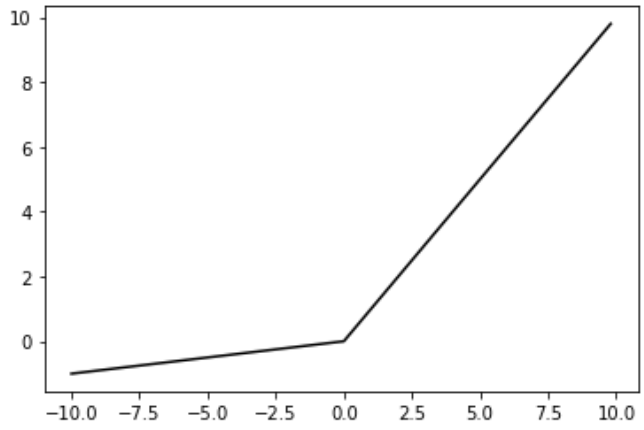

Figura 2.4: Gráfica de la función *leaky ReLU* con $\alpha = 0,1$

Otra generalización importante son las unidades maxout, que agrupa los elementos de z en subconjuntos de k elementos. Cada una de las unidades

de una capa de este tipo devuelve el máximo de uno de estos subconjuntos. Así,

$$g(\mathbf{z})_i = \max_{j \in \mathbb{G}(i)} z_j$$

donde $\mathbb{G}(i) = \{(i-1)k+1, ..., ik\}$ es el conjunto de índices para el subconjunto i-ésimo. Una capa de estas unidades puede aprender una función convexa lineal a trozos de hasta k trozos. Podemos considerar que más que la relación entre las unidades, estas capas aprenden la función de activación misma. Con un k suficientemente grande, estas unidades pueden aprende a aproximar cualquier función convexa con precisión arbitraria [GBC16].

2.1.3.2. Unidades con activación sigmoidal

Otras dos unidades comunes utilizan funciones sigmoidales, la función logística

$$g(\mathbf{z})_i = \sigma(z_i) = \frac{1}{1 + e^{-z_i}}$$

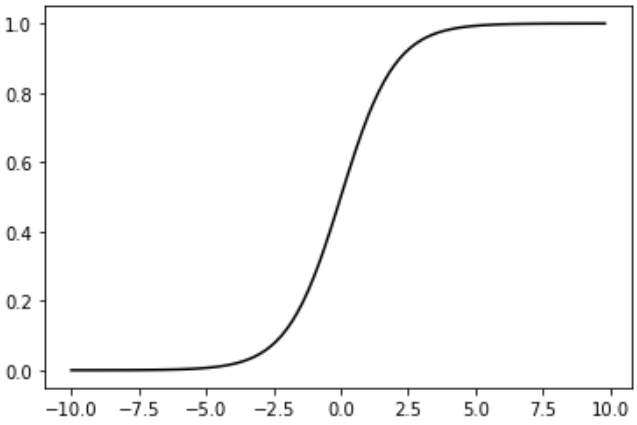

Figura 2.5: Gráfica de la función logística

y la función tangente hiperbólica

$$g(\mathbf{z})_i = \tanh(z_i) = \frac{e^{z_i} - e^{-z_i}}{e^{z_i} + e^{-z_i}}.$$

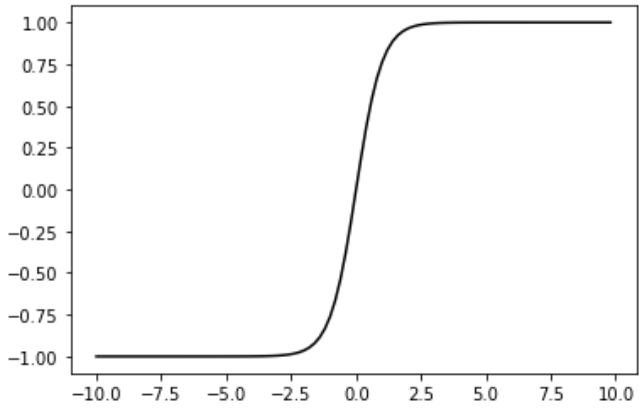

Figura 2.6: Gráfica de la función tangente hiperbólica

Estas dos funciones están muy relacionadas, ya que $\tanh(z) = 2\sigma(2z) - 1$. Las funciones sigmoidales se saturan en gran parte de su dominio, lo que puede dificultar el aprendizaje. Es por ello que su uso ha disminuido desde la introducción de las *ReLU*.

2.1.4. Propagación hacia adelante

Una vez configurada la red, esta recibe una entrada que se propaga a través de cada una de las capas, que le aplican las transformaciones correspondientes hasta la salida. Este proceso se llama propagación hacia adelante, y se describe en el algoritmo 1.

Algoritmo 1: Propagación hacia adelante en una red neuronal profunda con función de activación g para una entrada $\mathbf{x}$.

Entrada: ℓ profundidad de la red
Entrada: $\mathbf{W}_i$ matriz de pesos de la capa i-ésima, $i = 1, ..., \ell$
Entrada: $\mathbf{b}_i$ vector de sesgos de la capa i-ésima, $i = 1, ..., \ell$
Entrada: $\mathbf{x}$ entrada de la red
$\mathbf{h}_0 \leftarrow \mathbf{x}$;
para $i = 1, ..., \ell$ **hacer**
  $\mathbf{z}_i \leftarrow (\mathbf{W}_i)^T \mathbf{x}_{i-1} + \mathbf{b}_i$;
  $\mathbf{h}_i \leftarrow g_i(\mathbf{z}_i)$;
fin
$\mathbf{y} \leftarrow \mathbf{h}_\ell$;

2.2. Entrenamiento de redes neuronales prealimentadas

2.2.1. Optimización

Una vez conocida la función de coste $J(\theta)$ el siguiente paso es buscar unos parámetros θ que la minimicen. Para resolver este problema de optimización, al menos de forma aproximada, se aplica la técnica del descenso del gradiente. Este método fue inicialmente propuesto por Cauchy [Cau47] en 1847 y su convergencia fue estudiada por primera vez por Curry [Cur44] en 1944.

2.2.1.1. Descenso del gradiente

Sea f un campo escalar de $S \subset \mathbb{R}^n$ en $\mathbb{R}$, continuo en S y derivable con derivada continua en el interior de S. La dirección en la que la función desciende más rápidamente vendrá dada por la del vector

$$-\varepsilon \nabla f(x_1, ...x_n) = \left(-\varepsilon \frac{\partial f}{\partial x_1}(x_1, ..., x_n), ..., -\varepsilon \frac{\partial f}{\partial x_n}(x_1, ..., x_n) \right),$$

donde $\varepsilon > 0$ es un factor de proporcionalidad al que suele llamarse tasa de aprendizaje. El método consiste en, desde un punto inicial $\mathbf{x}_0$ e iterativamente, calcular el opuesto del gradiente y avanzar en esa dirección una distancia determinada por ε. Así, en la iteración i-ésima, $\mathbf{x}^{(i)} = \mathbf{x}^{(i-1)} - \varepsilon \nabla f(\mathbf{x}^{(i-1)})$. El algoritmo termina cuando el valor de $\nabla f(x_1, ..., x_n)$ es cercano a cero, con una tolerancia dada.

La convergencia del proceso al mínimo está probada cuando el mínimo de la función sobre la que se aplica es único. Este no es el caso de las funciones a las que se aplica en el contexto del aprendizaje profundo. Las funciones de coste que se intentan optimizar pueden tener muchos mínimos locales o puntos de silla en los que el algoritmo puede terminar sin obtener una buena solución. Otro inconveniente importante es el coste computacional que supone la aplicación del algoritmo a muestras grandes de datos, como suele hacerse en el aprendizaje automático para obtener modelos suficientemente generales. Es por ello que en la práctica se utilizan versiones del procedimiento que responden a estas cuestiones.

2.2.1.2. Descenso del gradiente estocástico (SGD)

La aplicación del descenso del gradiente a un problema de aprendizaje automático se realiza calculando la media aritmética de los gradientes de la función de pérdida para cada uno de los datos del conjunto de entrenamiento. Esto implica evaluar la función de coste para cada ejemplo del conjunto de entrenamiento antes de actualizar el gradiente, lo que puede ser computacionalmente inasumible. Es por ello que en la actualidad la técnica más usada es una aproximación en la que se utiliza un solo ejemplo para cada actualización. Este es, en la actualidad, el algoritmo más utilizado. Se describe detalladamente en 2.

Algoritmo 2: Algoritmo de descenso del gradiente estocástico.

Entrada: θ vector inicial de parámetros
Entrada: ε tasa de aprendizaje
Entrada: $\mathbf{x}_1, ..., \mathbf{x}_n$ ejemplos de entrenamiento
Entrada: $\mathbf{y}_1, ..., \mathbf{y}_n$ objetivo de cada ejemplo
mientras *no se alcanza un mínimo aproximado* **hacer**
  Mezclar aleatoriamente los ejemplos del conjunto de entrenamiento;
  para $i = 1, ..., n$ **hacer**
    $\theta \leftarrow \theta - \varepsilon \nabla L(f(\mathbf{x}_i; \theta), \mathbf{y}_i)$;
  fin
fin

2.2.1.3. Variantes de SGD

Mediante la aplicación de SGD se mitiga el problema del coste computacional del descenso del gradiente, pero la posibilidad de terminar en un mínimo local no óptimo o en un punto de silla sigue existiendo. Además, la aproximación que realiza SGD del gradiente puede no ser suficientemente buena. Existen diferentes alternativas que afrontan estos problemas.

- Gradiente descendente estocástico con minilotes: para realizar una aproximación más fiel del gradiente de la función toma minilotes de elementos del conjunto de entrenamiento en lugar de un solo elemento. Se describe detalladamente en 3.
- Método del momento [Sut13]: en cada paso se recuerda la anterior actualización y la siguiente se calcula como una combinación lineal entre

esta y el gradiente. De esta manera la variación de la dirección de una iteración a otra disminuye, evitando el comportamiento de zigzagueo. Se describe detalladamente en 4.

- AdaGrad [DHS11]: se basa en adaptar la tasa de aprendizaje para cada parámetro de forma inversamente proporcional a la raíz cuadrada de la suma de los valores anteriores. Así se decrementa más rápidamente la tasa de aprendizaje para los parámetros con mayor derivada parcial y se progresa más rápidamente en las zonas de menor pendiente. Se describe detalladamente en 5.
- RMSProp [TH12]: similar a adagrad, modifica la tasa de aprendizaje para cada parámetro. En este caso se cambia la acumulación del gradiente por una media con pesos exponenciales para mejorar su comportamiento en funciones no convexas. Se describe detalladamente en 6.
- Adam [KB14]: es otro algoritmo con tasa de aprendizaje adaptativa. Es una combinación entre RMSProp y método del momento, ya que añade un momento adaptativo. Se describe detalladamente en 7.

Algoritmo 3: Algoritmo de descenso del gradiente estocástico con minilotes.

Entrada: θ vector inicial de parámetros
Entrada: m tamaño de minilote
Entrada: ε tasa de aprendizaje
Entrada: $\mathbf{x}_1, ..., \mathbf{x}_n$ ejemplos de entrenamiento
Entrada: $\mathbf{y}_1, ..., \mathbf{y}_n$ objetivo de cada ejemplo
mientras *no se alcanza un mínimo aproximado* **hacer**
  Seleccionar un minilote de m ejemplos del conjunto de entrenamiento $\{\mathbf{x}_1, ..., \mathbf{x}_m\}$ y sus correspondientes objetivos $\mathbf{y}_i$;
  $\theta \leftarrow \theta - \frac{\varepsilon}{m} \nabla \sum_i L(f(\mathbf{x}_i; \theta), \mathbf{y}_i)$;
fin

2.2.2. Propagación hacia atrás

El cálculo del gradiente de la función de coste es básico para el entrenamiento de redes neuronales. Obtener su expresión analítica es sencillo, pero evaluar numéricamente esa expresión puede ser muy costoso computacionalmente. El algoritmo de propagación hacia atrás (*back propagation*) permite realizar este

Algoritmo 4: SGD con método del momento.

Entrada: θ vector inicial de parámetros
Entrada: ε tasa de aprendizaje
Entrada: α valor del momento
Entrada: v velocidad inicial
Entrada: $\mathbf{x}_1, ..., \mathbf{x}_n$ ejemplos de entrenamiento
Entrada: $\mathbf{y}_1, ..., \mathbf{y}_n$ objetivo de cada ejemplo
mientras *no se alcanza un mínimo aproximado* **hacer**
  Seleccionar un minilote de m ejemplos del conjunto de entrenamiento $\{\mathbf{x}_1, ..., \mathbf{x}_m\}$ y sus correspondientes objetivos $\mathbf{y}_i$;
  $v \leftarrow \alpha v - \frac{\varepsilon}{m} \nabla \sum_i L(f(\mathbf{x}_i; \theta), \mathbf{y}_i)$;
  $\theta \leftarrow \theta + v$;
fin

Algoritmo 5: Adagrad. $\odot$ representa el producto componente a componente, $\sqrt{\cdot}$ la raíz cuadrada componente a componente y la división $\frac{\varepsilon}{\delta + \sqrt{r}}$ se realiza componente a componente.

Entrada: θ vector inicial de parámetros
Entrada: δ constante pequeña, normalmente 10^{-6}
Entrada: ε tasa de aprendizaje inicial
Entrada: $\mathbf{x}_1, ..., \mathbf{x}_n$ ejemplos de entrenamiento
Entrada: $\mathbf{y}_1, ..., \mathbf{y}_n$ objetivo de cada ejemplo
$r \leftarrow 0$;
mientras *no se alcanza un mínimo aproximado* **hacer**
  Seleccionar un minilote de m ejemplos del conjunto de entrenamiento $\{\mathbf{x}_1, ..., \mathbf{x}_m\}$ y sus correspondientes objetivos $\mathbf{y}_i$;
  $g \leftarrow \frac{1}{m} \nabla \sum_i L(f(\mathbf{x}_i; \theta), \mathbf{y}_i)$;
  $r \leftarrow r + g \odot g$;
  $\theta \leftarrow \theta - \frac{\varepsilon}{\delta + \sqrt{r}} \odot g$;
fin

Algoritmo 6: RMSProp. $\odot$ representa el producto componente a componente, $\sqrt{\cdot}$ la raíz cuadrada componente a componente y la división $\frac{\varepsilon}{\delta+\sqrt{r}}$ se realiza componente a componente.

Entrada: θ vector inicial de parámetros
Entrada: ρ ratio de decaimiento
Entrada: δ constante pequeña, normalmente 10^{-6}
Entrada: ε tasa de aprendizaje inicial
Entrada: $\mathbf{x}_1, ..., \mathbf{x}_n$ ejemplos de entrenamiento
Entrada: $\mathbf{y}_1, ..., \mathbf{y}_n$ objetivo de cada ejemplo
$r \leftarrow 0$;
mientras *no se alcanza un mínimo aproximado* **hacer**
  Seleccionar un minilote de m ejemplos del conjunto de entrenamiento $\{\mathbf{x}_1, ..., \mathbf{x}_m\}$ y sus correspondientes objetivos $\mathbf{y}_i$;
  $g \leftarrow \frac{1}{m} \nabla \sum_i L(f(\mathbf{x}_i; \theta), \mathbf{y}_i)$;
  $r \leftarrow \rho r + (1 - \rho) g \odot g$;
  $\theta \leftarrow \theta - \frac{\varepsilon}{\delta+\sqrt{r}} \odot g$;
fin

cálculo de manera eficiente, utilizando la regla de la cadena para evitar repetir el cálculo de derivadas parciales que aparecen varias veces en el proceso.

Tras aplicar propagación hacia adelante se aplica el algoritmo descrito en 8, que obtiene los gradientes de la función de activación de cada capa, empezando por la capa de salida y yendo hacia atrás hasta la primera capa oculta. De estos gradientes, que pueden ser interpretados como la indicación de cómo la salida de cada capa debería cambiar para reducir el error, se puede obtener el gradiente respecto de los parámetros de cada capa. Obtenido el gradiente respecto de los pesos y los sesgos, puede utilizarse para aplicar alguno de los métodos de optimización basados en gradiente.

Algoritmo 7: Adam. $\odot$ representa el producto componente a componente, $\sqrt{\cdot}$ la raíz cuadrada componente a componente y la división $\frac{\varepsilon}{\delta+\sqrt{r}}$ se realiza componente a componente.

Entrada: θ vector inicial de parámetros
Entrada: $\rho_1, \rho_2 \in [0, 1)$ ratios de decaimeiento exponencial, sugeridos por defecto $0,9$ y $0,99$
Entrada: δ constante pequeña, normalmente 10^{-6}
Entrada: ε tasa de aprendizaje inicial
Entrada: $\mathbf{x}_1, ..., \mathbf{x}_n$ ejemplos de entrenamiento
Entrada: $\mathbf{y}_1, ..., \mathbf{y}_n$ objetivo de cada ejemplo
$r \leftarrow 0$;
$s \leftarrow 0$;
$t \leftarrow 0$;
mientras *no se alcanza un mínimo aproximado* **hacer**
  Seleccionar un minilote de m ejemplos del conjunto de entrenamiento $\{\mathbf{x}_1, ..., \mathbf{x}_m\}$ y sus correspondientes objetivos $\mathbf{y}_i$;
  $g \leftarrow \frac{1}{m} \nabla \sum_i L(f(\mathbf{x}_i; \theta), \mathbf{y}_i)$;
  $t \leftarrow t + 1$;
  $r \leftarrow \rho_1 r + (1 - \rho_1) g$;
  $r \leftarrow \rho_2 r + (1 - \rho_2) g \odot g$;
  $\hat{s} \leftarrow \frac{s}{1-\rho_1^t}$;
  $\hat{r} \leftarrow \frac{s}{1-\rho_2^t}$;
  $\theta \leftarrow \theta - \frac{\hat{s}}{\sqrt{\hat{r}}+\delta} \odot g$;
fin

Algoritmo 8: Propagación hacia atrás con entrada $\mathbf{x}$, salida $\mathbf{y}$, objetivo $\mathbf{y}^*$ y profundidad ℓ. L representa la función de coste sin regularización, y J la función de coste completa. $\odot$ denota el producto componente a componente.

$\mathbf{d} \leftarrow \nabla_{\mathbf{y}} J(\mathbf{y}, \mathbf{y}^*; \theta) = \nabla_{\mathbf{y}} L(\mathbf{y}, \mathbf{y}^*)$;
para $k = \ell, ..., 1$ **hacer**
  $\mathbf{d} \leftarrow \nabla_{\mathbf{z}_k} J = \mathbf{d} \odot g'(\mathbf{z}_k)$;
  $\nabla_{\mathbf{b}_k} J = \mathbf{d} + \lambda \nabla_{\mathbf{b}_k} \Omega(\theta)$;
  $\nabla_{\mathbf{W}_k} J = \mathbf{d}(h_{k-1})^T + \lambda \nabla_{\mathbf{W}_k} \Omega(\theta)$;
  $\mathbf{d} \leftarrow \nabla_{h_{k-1}} J = (\mathbf{W}_k)^T \mathbf{d}$;
fin

3 Tratamiento de secuencias

Hasta ahora se han descrito redes neuronales que tratan los datos de manera aislada. Cada instancia es procesada independientemente y se asigna una salida solo en función de la entrada facilitada. No existe ningún tipo de realimentación en la red. Esta estructura es adecuada para, por ejemplo, problemas de clasificación en los cuales los datos son independientes entre sí y la clasificación de un ejemplo no debería afectar a la del resto. Sin embargo pueden plantearse muchos problemas de aprendizaje en los que no exista tal independencia. El contexto de un ejemplo puede ser relevante para su resolución.

Un caso de este tipo de problemas es el tratamiento de secuencias, conjuntos de datos ordenados en los cuales cada instancia tiene cierta relación con sus antecesores y predecesores. Algunos tipos de datos que podemos interpretar como secuencias son textos, vídeos, grabaciones de voz, cadenas de ADN o música. En las tareas que impliquen su manejo será relevante tener en cuenta la relación de cada elemento con los previos y posteriores. Algunos ejemplos de problemas en los que se usan son:

- Generación de texto [LZZ+18]: consiste en producir texto en lenguaje natural. Puede aplicarse a multitud de fines, entre los que se incluyen completar textos automáticamente, generación de poemas o descripción textual de vídeo o imágenes.
- Traducción automática [SKD+17]: conversión automática de un lenguaje natural a otro lenguaje natural diferente.
- Análisis de secuencias de ADN [LFY+19]: dada una secuencia de ADN, detección de determinadas estructuras o modificaciones en el mismo.
- Reconocimiento de voz [GMH13]: transcripción del lenguaje hablado captado como audio a texto.
- Análisis de sentimientos [VLH17]: identificación y clasificación de opiniones expresadas en forma de texto para determinar si la actitud de su autor hacia un tema concreto es positiva, negativa o neutral.

- Generación de música: creación automática de música. Algunas de las aproximaciones existentes son completar una obra parcialmente compuesta con acompañamientos y contrapunto [HCR+17] o generar música con un estilo determinado [CZ18].

Además de la dependencia entre elementos es también destacable que por su naturaleza secuencial, los datos con los que trabaja este tipo de problemas pueden ser más extensos de lo que sería práctico computar mediante una red prealimentada profunda. Además pueden tener extensiones diferentes, lo que también resulta problemático. Es por ello que su tratamiento mediante las técnicas vistas hasta ahora resulta muy difícil y se hace necesaria la utilización de modelos especializados.

La principal idea a la hora de enfrentar tareas de esta índole es añadir retroalimentación a la red. De esta manera puede conservarse información a través de las diferentes resoluciones de ejemplos. Cuando se incluye esta característica estamos ante modelos llamados redes neuronales recurrentes (RNN), *recurrent neural networks* en inglés. En este capítulo se repasan las arquitecturas más importantes de este tipo y cómo se afronta el problema de la dependencia entre instancias y la conservación de información entre las sucesivas resoluciones.

3.1. Redes neuronales recurrentes

En este apartado consideramos una secuencia como un conjunto finito de ejemplos $\{\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, ..., \mathbf{x}^{(T)}\}$. Diremos que la secuencia tiene longitud T. La idea de introducir retroalimentación en la red es equivalente a introducir un nuevo parámetro a la función que representa la red, al cual llamaremos estado o estado oculto. En cada posición o instante t el estado $\mathbf{h}^{(t)}$ dependerá del estado anterior $\mathbf{h}^{(t-1)}$ y de la entrada actual $\mathbf{x}^{(t)}$. De igual manera, la salida del instante t, $\mathbf{y}^{(t)}$ ya no solo dependerá de la entrada $\mathbf{x}^{(t)}$ y los parámetros θ, sino también del estado anterior $\mathbf{h}^{(t-1)}$.

Para realizar la propagación hacia adelante en una red recurrente como la de la figura (insertar cita a figura anterior) basta con aplicar las siguientes ecuaciones en cada paso t, desde $t = 1$ hasta $t = T$:

$$\mathbf{h}^{(t)} = g_1(\mathbf{b} + \mathbf{W}\mathbf{h}^{(t-1)} + \mathbf{U}\mathbf{x}^{(t)}),$$

$$\mathbf{y}^{(t)} = g_2(\mathbf{c} + \mathbf{V}\mathbf{h}^{(t)})$$

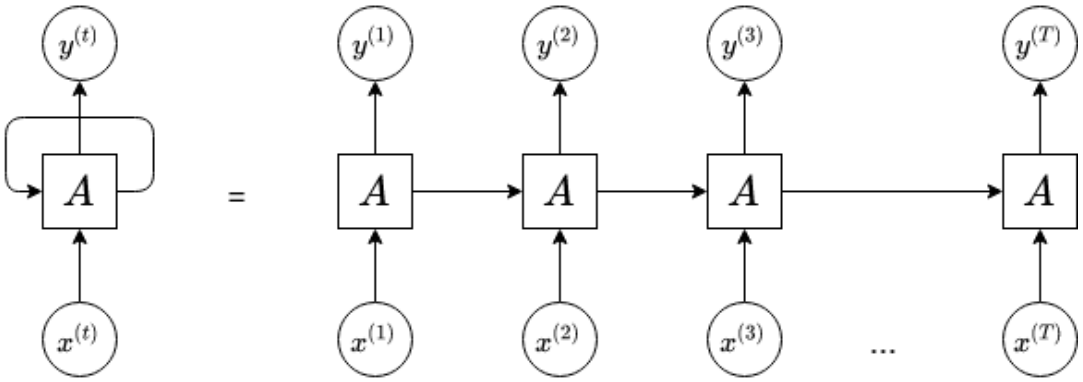

Figura 3.1: Desenrollado de una red neuronal recurrente

donde los parámetros son los vectores de de sesgos **b** y **c** junto con las matrices de pesos **U**, **V** y **W**, que conectan la entrada con el estado oculto, el estado oculto con la salida y el estado oculto actual con el siguiente, respectivamente. Las funciones g_1 y g_2 son funciones de activación que pueden ser diferentes, siendo la primera normalmente una función ReLU o tangente hiperbólica y encargada de generar el siguiente estado y la segunda la encargada de la salida de la red, lo cuál condicionará su forma. Es importante tener en cuenta que todas estas componentes son compartidas por todos los pasos. Por ello esta estructura es equivalente a introducir retroalimentación en la red. Se puede añadir también un estado oculto inicial $\mathbf{h}^{(0)}$, cuyo valor será nulo si no tenemos información previa.

La función coste total para una secuencia se calcula como la suma de los costes para cada uno de sus pasos. Calcular el gradiente de esta con respecto de cada parámetro es una operación computacionalmente costosa, ya que implica realizar una propagación hacia adelante de izquierda a derecha del modelo a través de cada uno de los pasos y después una propagación hacia atrás en sentido contrario a través de todo el grafo. El tiempo de ejecución es $O(T)$ y no puede ser reducido mediante paralelización ya que los grafos con los que trabajamos son secuenciales por definición. Los estados calculados en la propagación hacia adelante deben ser almacenados hasta su utilización en la propagación hacia atrás, por lo que el coste de memoria también es $O(T)$. Las redes recurrentes son por tanto muy útiles en muchos casos, pero también muy costosas de entrenar.

3.1.1. Arquitecturas más comunes

Hasta ahora se ha supuesto que la entrada y la salida tienen la misma longitud, pero no tiene por qué ser así. De hecho, uno de los problemas que solucionaba el uso de redes neuronales recursivas era la desigualdad en dichas longitudes. Veamos las diferentes situaciones que pueden darse en este sentido y los modelos resultantes.

- Arquitectura *one-to-one*: las redes tradicionales son un caso particular de las redes recurrentes en el que tanto la entrada como la salida tienen una longitud $T_x = T_y = 1$.

- Arquitectura *one-to-many*: en este caso la longitud de la entrada es uno, $T_x = 1$, pero la de la salida es mayor que uno, $T_y > 1$. En este caso, para $t \in \{2, ..., T\}$ se añade como entrada la salida del instante anterior, $\mathbf{y}^{(t-1)}$. Esta arquitectura suele aplicarse en generación de texto o música.

- Arquitectura *many-to-one*: al contrario que en el caso anterior, ahora $T_x > 1$ y $T_y = 1$. Con estas condiciones solo el último instante de la red tendrá salida, mientras que el resto solo leerán la secuencia de entrada y generan los respectivos estados ocultos. Es útil, por ejemplo, para clasificar o etiquetar secuencias. Una aplicación es el ya visto análisis de sentimientos.

- Arquitectura *many-to-many*, igual longitud: en este caso la secuencia de entrada y de salida tienen la misma longitud. Es la representada al principio del apartado. Se utiliza en casos en los que la secuencia de entrada y de salida están directamente relacionadas, como por ejemplo para el reconocimiento de entidades en un texto.

- Arquitectura codificador-decodificador: se trata de otra variante de la arquitectura *many-to-many*, pero en este caso $T_x \neq T_y$. Podemos verla como la unión de dos estructuras. La primera, el codificador, lee la secuencia de entrada sin producir salida una salida y genera los estados ocultos correspondientes. La segunda, el decodificador, produce las salidas sin una entrada, solo a través de los estados ocultos. Tiene múltiples utilidades como la traducción automática o el reconocimiento de voz.

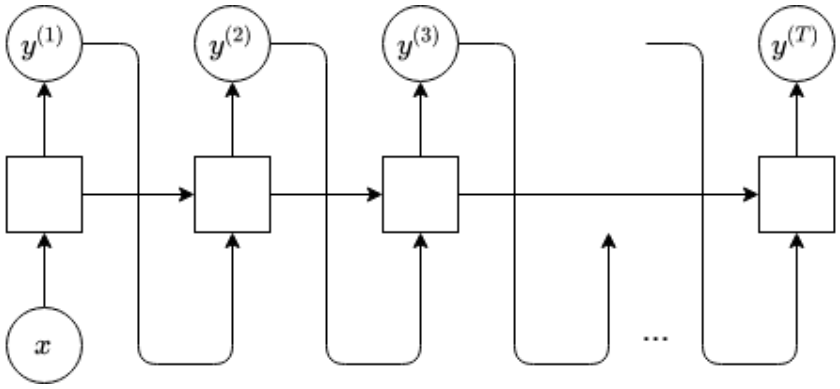

Figura 3.2: Red recurrente con arquitectura *one-to-many*

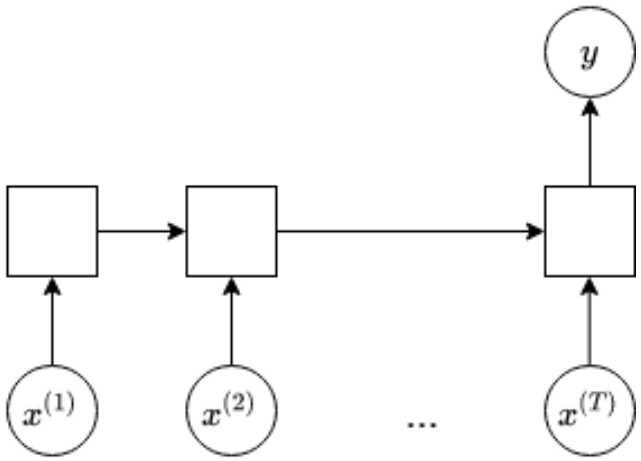

Figura 3.3: Red recurrente con arquitectura *many-to-one*

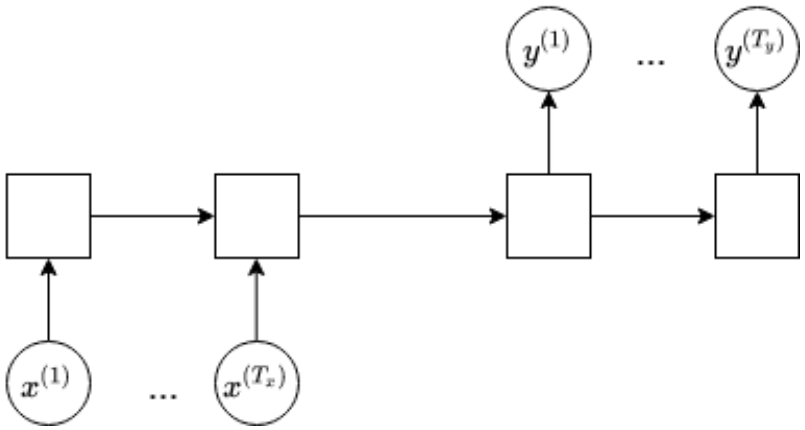

Figura 3.4: Red recurrente con arquitectura codificador-decodificador

3.1.2. Redes recurrentes bidireccionales

Las redes recursivas mencionadas hasta el momento, en el instante t, solo capturan información de las entradas pasadas y presentes, $\mathbf{x}^{(1)}, ..., \mathbf{x}^{(t-1)}, \mathbf{x}^{(t)}$. En muchas aplicaciones, sin embargo, la salida deseada no depende solo de las posiciones previas en la secuencia, sino de la secuencia de entrada completa. Podemos encontrar un ejemplo de esto en el reconocimiento de voz. La correcta interpretación de un fonema puede depender de los siguientes, o incluso de las siguientes palabras. Si hay varias interpretaciones de una palabra, ambas acústicamente plausibles, puede ser necesario recurrir a la información que se tiene del futuro y del pasado para elegir la más correcta. Las redes neuronales recurrentes bidireccionales (RNNs bidireccionales) se crearon para afrontar este tipo de problemas.

Las redes recurrentes bidireccionales combinan dos redes recurrentes, una que avanza hacia adelante en el tiempo, desde el principio hasta al final de la secuencia de entrada, y otra que se mueve en dirección contraria, desde el final de la secuencia de entrada hasta su principio. La salida de cada instante dependerá de los estados ocultos de ambas subredes. De esta manera las unidades de salida producen una representación que depende del pasado y del futuro, y que es más sensible a entradas cercanas en el tiempo.

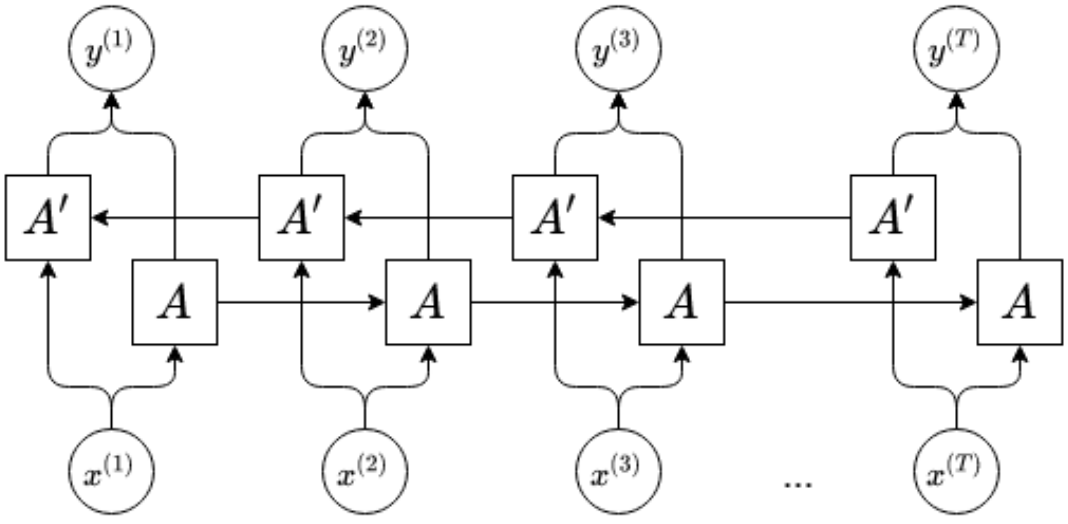

Figura 3.5: Red recurrente bidireccional

3.1.3. Redes recurrentes profundas

Podemos clasificar las operaciones que realizan la mayoría de redes neuronales profundas en tres categorías de parámetros y sus transformaciones asociadas:

1. desde la entrada hasta el estado oculto,
2. desde el estado oculto previo hasta el siguiente estado oculto y
3. desde el estado oculto a la salida.

Hasta ahora cada uno de estos bloques ha estado asociado a una única matriz de pesos, es decir, realiza una transformación que puede ser representada por una única capa de una red prealimentada profunda. La evidencia experimental ([GMH13], [PGCB13]) sugiere que puede ser beneficioso introducir una mayor profundidad en cada una de las categorías. Si se piensa en la capacidad de representación, aumentar la profundidad resulta beneficioso. Sin embargo puede ser contraproducente para el aprendizaje, ya que hace la optimización más difícil.

La profundidad de una red recurrente puede aumentarse añadiendo capas ocultas, que pueden introducirse como capas ocultas recurrentes o como capas previas o posteriores a la capa o capas recurrentes. La estructura de una red recurrente profunda queda reflejada en la Figura 3.6.

3.2. Dependencia a largo plazo

Para muchas de las aplicaciones que hemos visto hasta ahora de las redes neuronales recurrentes es importante que el modelo conserve cierta información a través del tiempo hasta instantes lejanos. Cuando los grafos de los modelos usados se vuelven muy profundos, y especialmente en el caso de las redes recurrentes en las que la profundidad consiste en la aplicación múltiples veces de la misma operación, surgen problemas en el aprendizaje. En esta sección se describe el motivo de estos problemas y algunas propuestas para su afrontamiento.

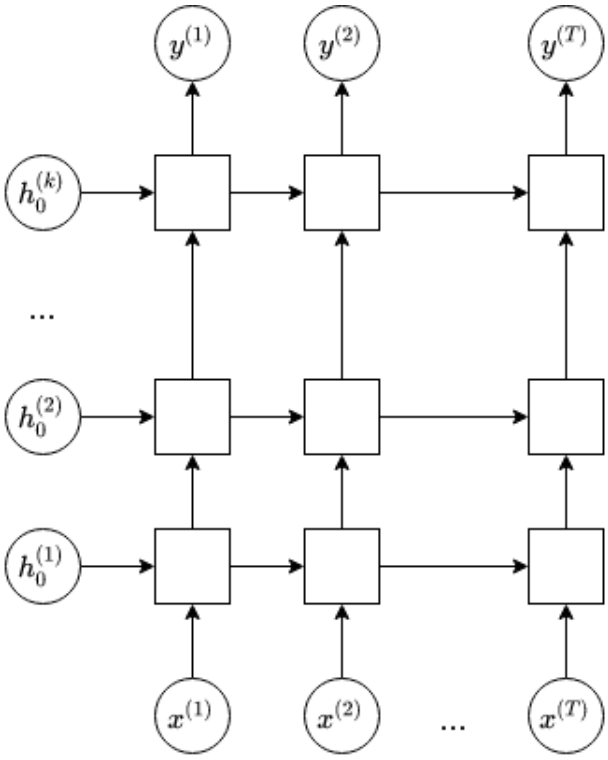

Figura 3.6: Red recurrente profunda

3.2.1. Problemas del gradiente en redes recurrentes

El problema clave en el aprendizaje de dependencias a largo plazo es que el gradiente, propagado a través de gran cantidad de etapas, tiende a desvanecerse hacia valores casi nulos en la mayoría de los casos o, más raramente pero siendo más dañino para la optimización, a explotar hacia valores muy grandes. Estos son los conocidos como *vanishing gradient problem* y *exploding gradient problem*.

Las redes recurrentes se basan en la composición de la misma función, una vez cada paso. La composición sucesiva de funciones no lineales puede resultar en un comportamiento extremadamente no lineal, con una gran mayoría de valores con derivada casi nula, algunos valores con derivadas muy grandes y gran cantidad de alteraciones entre crecimiento y decrecimiento.

En particular, la composición empleada por las redes recurrentes neuronales es parecida a la multiplicación de matrices. Podemos tomar un ejemplo simplificado con

$$\mathbf{h}^{(t)} = \mathbf{W}^T \mathbf{h}^{(t-1)},$$

una red recurrente muy simple con función de activación lineal y sin entrada. En este caso, la aplicación del paso t-ésimo consiste en multiplicar la matriz de pesos $\mathbf{W}$ traspuesta con el estado anterior. Por lo tanto llegar al estado t-ésimo es, simplemente, aplicar esta operación t veces sobre el estado inicial,

$$\mathbf{h}^{(t)} = \left(\mathbf{W}^t\right)^T \mathbf{h}^{(0)}.$$

Suponiendo $\mathbf{W}$ diagonalizable, con

$$\mathbf{W} = \mathbf{Q}\mathbf{A}\mathbf{Q}^T,$$

con $\mathbf{Q}$ ortogonal, podemos expresar

$$\mathbf{h}^{(t)} = \mathbf{Q}^T \mathbf{A}^t \mathbf{Q} \mathbf{h}^{(0)}.$$

Los valores propios están elevados a t, lo que produce que aquellos con valor propio menor que 1 tenderán a desvanecerse hacia o y los que tengan valor propio mayor que 1 a explotar con valores muy grandes. Por tanto, cualquier componente de $\mathbf{h}^{(0)}$ que no esté alineada con valores propios grandes será descartada.

A continuación se exponen algunas alternativas a las redes recurrentes cuyo objetivo es evitar problemas con el gradiente y memorizar así dependencias a largo plazo.

3.2.2. Redes recurrentes con puertas

Los modelos que han demostrado ser más efectivos en aplicaciones prácticas para el uso de secuencia son las redes neuronales recursivas con puertas (*gated RNNs*). Estos se basan en la idea de crear caminos a lo largo del tiempo cuyo gradiente no se desvanezca ni explote. A través de estos las redes recursivas con puertas podrán acumular información. Cuándo la información se almacena y se descarta será también aprendido por la red. Existen dos alternativas principales, modelos con memoria larga a corto plazo (*long-short term memory, LSTM*), y las redes basadas en unidades recurrentes con puertas (*gated recurrent unit, GRU*).

3.2.2.1. LSTM

Las unidades de las redes neuronales simples son sustituidas por estructuras más complejas, a las que se llama bloques LSTM. Así, una red con LSTM tiene la misma forma de cadena de una RNN, pero utilizando estas nuevas estructuras cuyo comportamiento está dirigido a conservar información en el largo plazo [HS97].

La estructura de una célula es la representada en la gráfica Figura 3.7. La adición clave es una celda de memoria en cada bloque, $\mathbf{c}^{(t)}$. La información que se se almacena desde la entrada y el estado oculto en la celda y que se entrae de la misma hacia la salida viene filtrada por diferentes puertas, unidades con su correspondiente función de activación.

El primer paso es decidir qué información se elimina de la celda. Esta decisión se realiza mediante la puerta de olvido, que contiene una unidad sigmoidal cuyas entradas son el estado oculto anterior $\mathbf{h}^{(t-1)}$ y la entrada del bloque $\mathbf{x}^{(t)}$. Su salida $\mathbf{f}^{(t)}$, entre 0 y 1, señala con qué intensidad se mantiene la información de la celda.

El siguiente paso es decidir qué nueva información se almacena en la celda. Se realiza en dos etapas. En la primera una unidad sigmoidal llamada puerta de entrada decide qué valores se actualizarán. Después una unidad tanh crea un vector con los nuevos datos candidatos $\mathbf{z}^{(t)}$, que serán sumados a la celda. Por último se combinan estos dos pasos, multiplicando coordenada a coordenada, para definir finalmente la actualización de la celda.

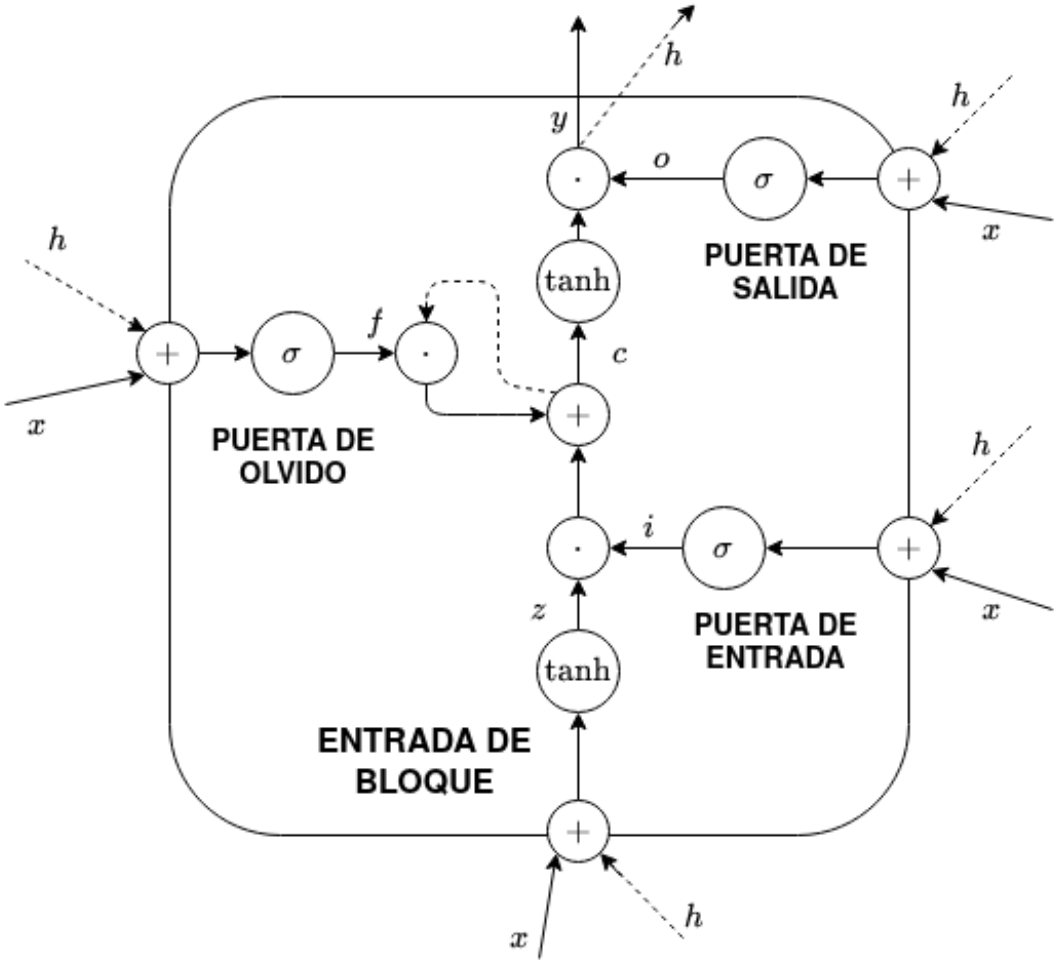

Figura 3.7: Bloque LSTM. Las líneas discontinuas representan conexiones con retardo en el tiempo

A continuación se actualiza el estado anterior de la celda. Basta multiplicar coordenada a coordenada el estado anterior por el vector obtenido en la puerta de olvido y sumarle el vector obtenido en la puerta de entrada.

Por último se procesa la salida, que será una versión filtrada de la celda de memoria utilizando la entrada y el estado oculto anterior. El estado oculto y la entrada son transformadas por una unidad sigmoidal. El estado actual de la célula pasa por una unidad de tanh y se multiplica componente a componente su salida con la de la puerta de salida.

Podemos resumir el modelo en las siguientes ecuaciones, donde $\odot$ representa el producto componente a componente [GSK+16]:

- Entrada de bloque: $\mathbf{z}^{(t)} = \tanh(\mathbf{W}_z\mathbf{x}^{(t)} + \mathbf{R}_z\mathbf{h}^{(t-1)} + \mathbf{b}_z)$
- Puerta de entrada: $\mathbf{i}^{(t)} = \sigma(\mathbf{W}_i\mathbf{x}^{(t)} + \mathbf{R}_i\mathbf{h}^{(t-1)} + \mathbf{b}_i)$
- Puerta de olvido: $\mathbf{f}^{(t)} = \sigma(\mathbf{W}_f\mathbf{x}^{(t)} + \mathbf{R}_f\mathbf{h}^{(t-1)} + \mathbf{b}_f)$
- Celda: $\mathbf{c}^{(t)} = \mathbf{z}^{(t)} \odot \mathbf{i}^{(t)} + \mathbf{c}^{(t-1)} \odot \mathbf{f}^{(t)}$
- Puerta de salida: $\mathbf{o}^{(t)} = \sigma(\mathbf{W}_o\mathbf{x}^{(t)} + \mathbf{R}_o\mathbf{h}^{(t-1)} + \mathbf{b}_o)$
- Salida de bloque: $\mathbf{h}^{(t)} = \tanh(\mathbf{c}^t) \odot \mathbf{o}^{(t)}$

donde $\mathbf{W}_z$, $\mathbf{W}_i$, $\mathbf{W}_f$, $\mathbf{W}_o$ son las matrices de pesos para la entrada en cada una de las unidades, $\mathbf{R}_z$, $\mathbf{R}_i$, $\mathbf{R}_f$, $\mathbf{R}_o$ las matrices de pesos para los estados ocultos previos y $\mathbf{b}_z$, $\mathbf{b}_i$, $\mathbf{b}_f$, $\mathbf{b}_o$ los respectivos vectores de sesgos.

Sobre el bloque LSTM visto hasta ahora pueden introducirse varias modificaciones. La más popular es la que introduce conexiones de mirilla (*peephole connections*) [GS00], mediante las cuales se conecta el estado de la celda de memoria con cada una de las unidades de las puertas. Estas conexiones pueden añadirse a todas las puertas o solo a algunas.

3.2.2.2. GRU

Una variación que surge de la simplificación de las celdas LSTM vistas anteriormente son las unidades recurrentes con puertas (*gated recurrent unit*, GRU) [CVMG+14]. La principal diferencia es que combina la puerta de entrada y la puerta de olvido en una sola unidad, la puerta de actualización (*update gate*). Tampoco diferencia entre estado oculto y celda de memoria, por

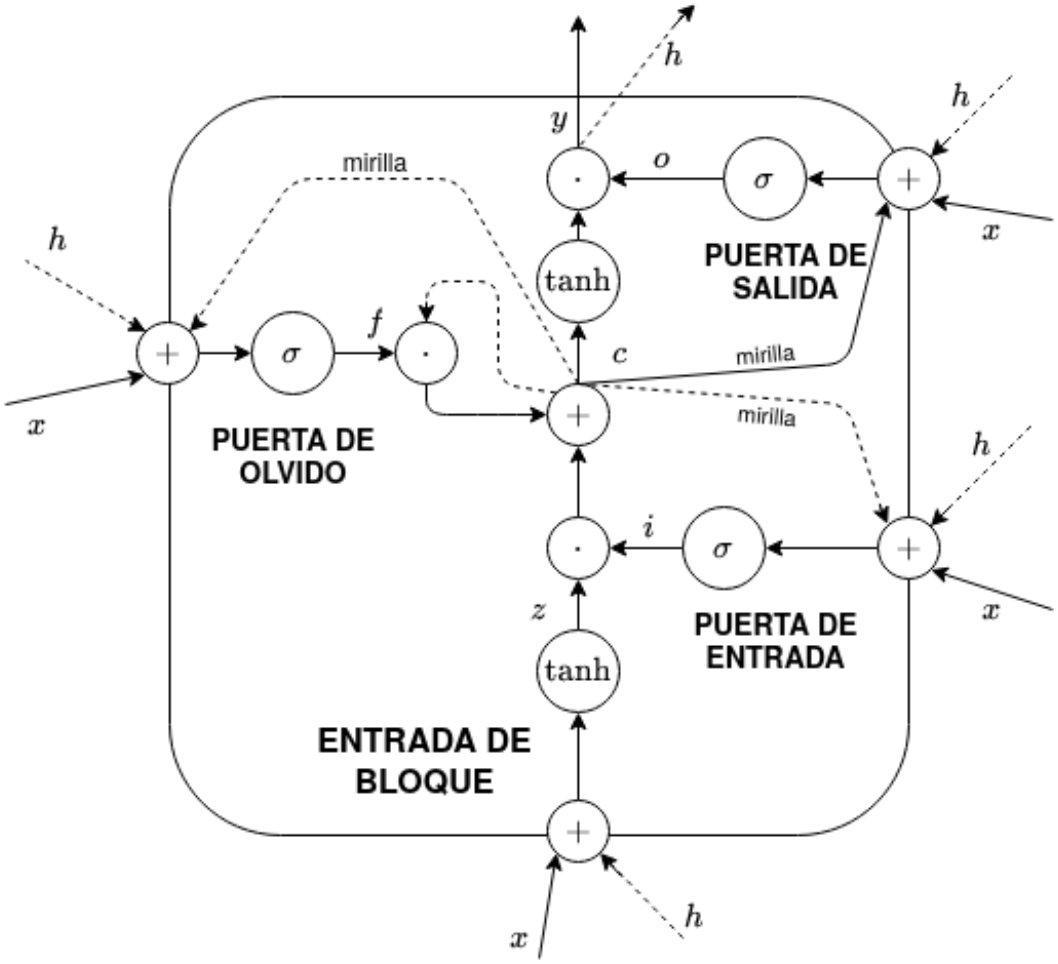

Figura 3.8: Bloque LSTM con conexiones de mirilla. Las líneas discontinuas representan conexiones con retardo en el tiempo

lo que no es necesario el uso de una puerta de salida. A continuación se describe su funcionamiento detalladamente.

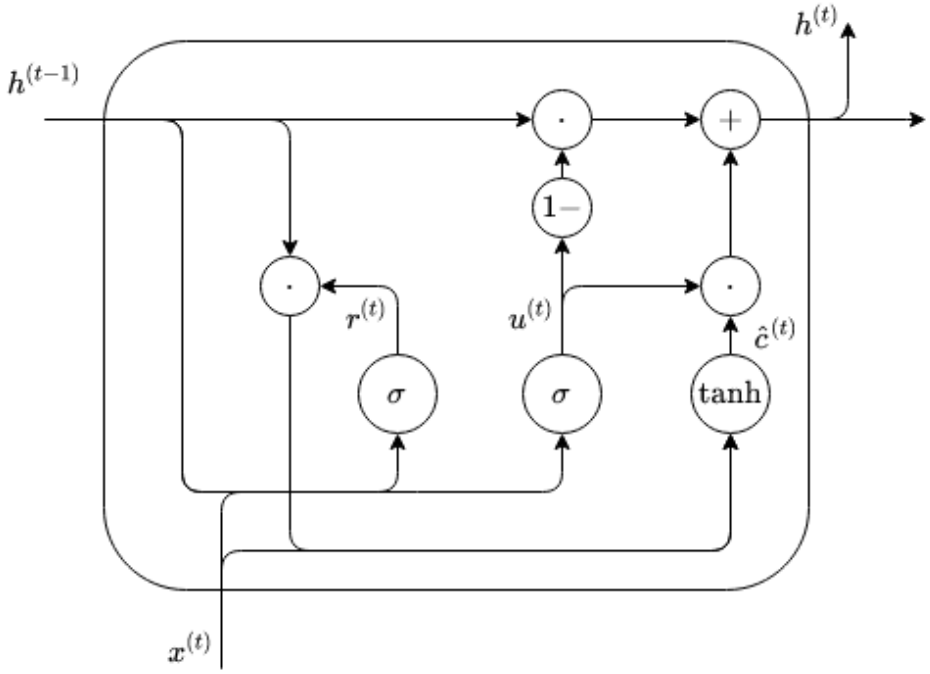

Figura 3.9: Bloque GRU

Primeramente el estado previo anterior $\mathbf{h}^{(t-1)}$ y la entrada actual $\mathbf{x}^{(t)}$ son filtrados a través de la puerta de relevancia, que normalmente es una unidad sigmoidal. Su salida $\mathbf{r}^{(t)}$ indica cómo de relevante es el estado oculto previo.

A continuación se calcula el valor $\mathbf{u}^{(t)}$ de la puerta de actualización mediante otra unidad sigmoidal. El valor $1 - \mathbf{u}^{(t)}$ sustituye al que en LSTM correspondería a la salida de la puerta de olvido.

Los datos candidatos se calculan, por tanto, aplicando una unidad tanh a la multiplicación componente a componente de la salida de la puerta de relevancia, $\mathbf{r}^{(t)}$, y la entrada actual, $\mathbf{x}^{(t)}$. La actualización del estado oculto se realiza entonces sumando la multiplicación componente a componente de los datos candidatos por la salida de la puerta de actualización con el producto componente a componente de $1 - \mathbf{u}^{(t)}$ con el valor oculto anterior.

Podemos resumir el modelo en las siguientes ecuaciones, donde $\odot$ representa el producto componente a componente:

- Puerta de relevancia: $\mathbf{r}^{(t)} = \sigma(\mathbf{W}_r \mathbf{x}^{(t)} + \mathbf{R}_r \mathbf{h}^{(t-1)} + \mathbf{b}_r)$

- Puerta de actualización: $\mathbf{u}^{(t)} = \sigma(\mathbf{W}_u \mathbf{x}^{(t)} + \mathbf{R}_u \mathbf{h}^{(t-1)} + \mathbf{b}_u)$
- Candidato a estado oculto: $\hat{\mathbf{c}}^{(t)} = \tanh(\mathbf{W}_c \mathbf{x}^{(t)} + \mathbf{R}_c(\mathbf{r}^{(t)} \odot \mathbf{h}^{(t-1)}) + \mathbf{b}_c)$
- Salida: $\mathbf{h}^{(t)} = (1 - \mathbf{u}^{(t)}) \odot \mathbf{h}^{(t-1)} + \mathbf{z}^{(t)} \odot \hat{\mathbf{c}}^{(t)}$

donde $\mathbf{W}_r$, $\mathbf{W}_u$, $\mathbf{W}_c$ son las matrices de pesos para la entrada en cada una de las unidades, $\mathbf{R}_r$, $\mathbf{R}_u$, $\mathbf{R}_c$, las matrices de pesos para los estados ocultos previos y $\mathbf{b}_r$, $\mathbf{b}_u$, $\mathbf{b}_c$ los respectivos vectores de sesgos.

4 Aprendizaje de características

La efectividad de los métodos de aprendizaje automático depende en gran medida de la representación de los datos utilizada. La obtención de una buena representación es crucial en el proceso de aprendizaje. Dicha tarea es una de las que más tiempo consumen en cualquier proyecto de esta área [Dom12]. El desarrollo de técnicas que permitan la obtención de buenas representaciones automáticamente se ha convertido en un importante campo dentro del aprendizaje automático. A este conjunto de técnicas se le denomina aprendizaje de características o de representación. [BCV13].

Obtener representaciones mejores para la aplicación de técnicas de aprendizaje no es el único motivo para la aplicación del aprendizaje de características. En muchas aplicaciones la obtención de características es la meta en sí. Algunos ejemplos de esto son la representación de documentos en códigos binarios compactos que faciliten su búsqueda [SH09], compresión de imágenes minimizando la pérdida de información [TSCH17], la transformación del dominio de un problema a otro diferente [DZMS13] o la generación de imágenes restauradas a partir de versiones distorsionadas [XXC12].

En el caso de este trabajo se aplica esta idea a la manipulación y generación de melodías musicales. Estas son datos de naturaleza compleja, difíciles de manejar de forma rápida y efectiva, y más aún con la inmediatez que requiere una interpretación en directo. Utilizaremos melodías monofónicas de dos compases de 4/4, es decir, como máximo una nota suena a la vez y hay cuatro pulsos por compás. Supongamos además, con afán de simplificación, que la máxima subdivisión posible para dichas melodías es la semicorchea (un cuarto de pulso) e ignoremos el tempo al que se interpreta la secuencia y el volumen. Si suponemos que las notas diferentes posibles son las 88 que tienen los pianos actuales y que en cada pulso podemos tocar una de las 88 teclas, levantar la tecla que se venía tocando o realizar un silencio, esto nos deja 90 posibles acciones para cada semicorchea. Necesitaríamos entonces 7 bits por semicorchea para su cifrado, y tenemos 32 semicorcheas por melodía. Explorar todas las posibilidades bit a bit es una tarea intratable y la mayoría de melodías obtenidas serían incoherentes [REOE18].

El objetivo será por tanto obtener una representación de estas melodías fácil de manipular por parte del artista, que permita un acercamiento diferente a la composición y la interpretación. Para ello será clave la extracción de representaciones de baja dimensionalidad.

4.1. Selección de características

El precedente para la extracción de características se encuentra en la selección de características para problemas de aprendizaje automático. En los últimos años del número de características utilizadas ha crecido desde las pocas decenas hasta las miles o decenas de miles [GE03]. Con este crecimiento surge también la necesidad de seleccionar aquellas características más útiles para la tarea que se pretende afrontar.

La selección de características consiste en seleccionar un subconjunto de entre las características disponibles para mejorar la tarea de aprendizaje. Existen muchos beneficios potenciales que pueden derivarse de esta acción: facilitar la visualización y entendimiento de los datos, disminución de los recursos computacionales necesarios, tanto en espacio de memoria para almacenaje de datos como en tiempo de cómputo en el entrenamiento y predicción, o evitar la maldición de la dimensionalidad (*the curse of dimensionality*) que obstaculiza o imposibilita el aprendizaje cuando se cuenta con pocos ejemplos y un número de características elevado [BGRS99].

Un primer acercamiento a la selección de características consiste en la jerarquización de variables utilizando alguna métrica del valor de cada variable para el aprendizaje y descartando las menos útiles. Es una técnica ampliamente usada por su simplicidad, escalabilidad y buen desempeño empírico, pero presenta un problema importante. La mayoría de las veces la utilidad de una variable no puede juzgarse individualmente, y la inclusión de variables que resultan redundantes por sí solas pueden producir una mejora significativa de los resultados por su relación con otras.

Como respuesta a este hecho surgen métodos que no tratan de seleccionar características individualmente, sino conjuntos de características. De entre estos podemos distinguir tres categorías:

- Métodos de filtro: seleccionan el subconjunto de características como paso de preprocesamiento de datos, independientemente del predictor elegido.

- Métodos de envoltura: utilizan el desempeño de un algoritmo de aprendizaje automático para medir la utilidad del subconjunto seleccionado. Su alto coste computacional implica que una búsqueda exhaustiva sea imposible en la práctica, por lo que la estrategia de búsqueda utilizado es clave en este tipo de métodos.
- Métodos integrados: realizan la selección de características durante el aprendizaje y son específicos del algoritmo de aprendizaje usado.

4.2. Extracción de características

Frente a la selección de entre las características ya existentes aparece una perspectiva diferente: la creación de nuevas representaciones a partir de la existente, no solo mediante la selección sino mediante la generación de nuevas características a través de transformaciones de las iniciales. Surge no solo con la intención de generar representaciones más eficientes para el aprendizaje, sino también más comprensibles y fáciles de manejar.

Puede pasar que todas las características de los datos sean relevantes, y aún así la dimensionalidad de los mismos sea todavía demasiado elevada. Descubrir las dependencias entre las características puede permitir reducir dicha dimensionalidad. Cuando dos características están altamente correladas, conocer información sobre una implica conocer información sobre la otra. En lugar de eliminar arbitrariamente una de estas características, una manera de reducir la dimensionalidad es obtener una representación a partir de una transformación de dichas características. Esta perspectiva viene motivada por el hecho de que las dependencias entre variables pueden ser muy complejas, y eliminar una de ellas puede resultar en perder parte de la información que ambas comunican [LV07].

El nuevo conjunto de características por lo general debe tener un número menor de variables pero debe preservar la información del conjunto inicial. Se buscan por tanto transformaciones de las variables con unas propiedades tales que se asegure que la transformación no altera la información que contenían las variables iniciales, solo su representación.

Estas transformaciones pueden intentar cumplir dos objetivos diferentes. El primero, y más simple, es detectar y eliminar dependencias. Para ello la transformación debe reducir el número de variables. El segundo, y más complejo,

es recuperar las variables latentes, esto es, aquellas que son el origen de las observadas, pero que no pueden medirse directamente.

La construcción de una representación nueva podría considerarse una tarea muy específica del dominio de la información que se trata, y puede ser la manera de introducir conocimiento específico sobre dicho dominio. Sin embargo, se han desarrollado gran cantidad de técnicas genéricas para la construcción de nuevas representaciones. En esta sección se repasan algunas de ellas. Se explicarán algunas de las técnicas tradicionales para después hacer especial hincapié en aquellas para las que se aplica aprendizaje profundo.

4.2.1. Técnicas tradicionales de extracción de características

Podemos separar las técnicas entre aquellas que realizan transformaciones lineales de los datos de entrada y las que realizan transformaciones no lineales. De entre las técnicas lineales la más extendida es el análisis de componentes principales.

El análisis de componentes principales (*principal component analysis*, PCA) se base en la asunción de que las D variables observadas, expresadas en el vector aleatorio $\mathbf{Y} = (Y_1, ..., Y_D)^T$ son el resultado de una transformación lineal $\mathbf{W}$ de variables P latentes desconocidas, $\mathbf{X} = (X_1, ..., X_M)^T$:

$$\mathbf{Y} = \mathbf{W}\mathbf{X}.$$

Se asume también que todas las variables latentes tienen una distribución normal, y que la transformación $\mathbf{W}$ es un cambio de ejes, es decir, que sus columnas son ortogonales entre sí y tienen norma unitaria. En definitiva, $\mathbf{W} \in \mathcal{M}_{D\times P}$, con $\mathbf{W}^T\mathbf{W} = \mathbf{I}_P$. Por último, pero menos restrictivo, PCA asume que las variables observadas y y las variables latentes $\mathbf{x}$ están centradas, es decir, $E[\mathbf{X}] = 0_D$ y $E[\mathbf{Y}] = 0_P$.

A partir de estas condiciones se pretende obtener la dimensión P y la transformación $\mathbf{W}$ a partir de un conjunto de N observaciones del vector aleatorio $\mathbf{Y}$, que pueden expresarse como una matriz

$$Y = [\mathbf{Y}^{(1)}, ..., \mathbf{Y}^{(N)}].$$

Si asumimos que las variables de $\mathbf{X}$ son incorreladas, es decir, que la matriz de covarianzas $Cov(\mathbf{X}) = E[\mathbf{X}\mathbf{X}^T]$ es diagonal. Tras la transformación por $\mathbf{W}$ las variables de $\mathbf{y}$ sí estarán correladas, y su matriz de covarianzas no será

diagonal. El objetivo de PCA es obtener las P variables incorreladas de $\mathbf{X}$ a través de las de $\mathbf{Y}$. Podemos expresar la matriz de covarianzas de $\mathbf{Y}$ como

$$Cov(\mathbf{Y}) = E[\mathbf{Y}\mathbf{Y}^T] = E[\mathbf{W}\mathbf{X}\mathbf{X}^T\mathbf{W}^T] = \mathbf{W}E[\mathbf{X}\mathbf{X}^T]\mathbf{W}^T = \mathbf{W}Cov(\mathbf{X})\mathbf{W}^T.$$

Como $\mathbf{W}^T\mathbf{W} = \mathbf{I}$, multiplicando a la izquierda por $\mathbf{W}^T$ y a la derecha por $\mathbf{W}$ obtenemos

$$Cov(\mathbf{X}) = \mathbf{W}^T Cov(\mathbf{Y})\mathbf{W}.$$

Podemos entonces factorizar la matriz Cov($\mathbf{Y}$), que por ser simétrica es diagonalizable, como

$$Cov(\mathbf{Y}) = \mathbf{V}\mathbf{A}\mathbf{V}^T,$$

donde $\mathbf{V}$ es una matriz de vectores propios normalizados $\mathbf{v}_d$ y $\mathbf{A}$ es una matriz diagonal con los valores propios asociados λ_d, en orden descendiente. Como la matriz de covarianzas es simétrica y semidefinida positiva los vectores propios son ortogonales y los valores propios son reales no negativos. Sustituyendo en la expresión de $Cov(\mathbf{X})$ obtenemos

$$Cov(\mathbf{X}) = \mathbf{W}^T\mathbf{V}\mathbf{A}\mathbf{V}^T\mathbf{W},$$

donde, en el caso ideal en el que todas las hipótesis son respetadas y las observaciones no tienen ruido, solo los primeros P valores propios en $\mathbf{A}$ son positivos, siendo el resto nulos. Tomamos entonces los vectores propios asociados a estos P valores,

$$\mathbf{W} = \mathbf{V}\mathbf{I}_{D\times P},$$

obteniendo así

$$Cov(\mathbf{X}) = \mathbf{I}_{P\times D}\mathbf{A}\mathbf{I}_{D\times P}.$$

Esto significa que los valores propios en $\mathbf{A}$ se corresponden con las varianzas de las variables latentes.

En la práctica los datos observados serán ruidosos y por tanto todos los valores propios de $Cov(\mathbf{Y})$ serán mayores que cero, por lo que la elección de las P columnas en $\mathbf{V}$ es más complicada. Si las variables latentes tienen varianzas mayores que el ruido, basta tomar los vectores correspondientes a los mayores valores propios.

Desde un punto de vista geométrico, las columnas de $\mathbf{V}$ indican las direcciones en $\mathbb{R}^D$ sobre las que se extiende el subespacio de las variables latentes. Si llamamos a estas columnas componentes, la elección de las columnas asociadas a las mayores varianzas es la elección de las componentes principales.

En una situación real la covarianza de **Y** no es conocida, pero puede ser aproximada mediante la covarianza de la muestra:

$$Cov(Y) = \frac{1}{N} YY^T.$$

Otra técnica lineal muy extendida es el análisis discriminante lineal. En este caso se trata de un método supervisado para datos clasificados. Genera una representación de los datos con una dimensión menor que respeta la estructura de clases y maximiza la separación entre las mismas. Es habitual aplicarla tras PCA [YJL05].

En cuanto a las técnicas que utilizan transformaciones no lineales, el escalamiento multidimensional (*multidimensional scaling*, MDS) es una técnica que además sirve como base para otros algoritmos. Consiste en encontrar nuevas coordenadas en un espacio de menor dimensión manteniendo las distancias relativas entre los datos de manera lo más fiel posible. Isomap se basa en MDS, extendiéndolo para encontrar coordenadas que describan los verdaderos grados de libertad de los datos mientras que se preservan las distancias entre puntos. Por último la máquina de Boltzmann restringida (*restricted boltzmann machine*, RMB) es un modelo que cuenta con una capa oculta y una visible. Están definidas por una distribución de probabilidad conjunta determinada por una función de energía. Son una alternativa para la inicialización de *autoencoder* por capas, que veremos a continuación.

4.2.2. Autoencoder

Los *autoencoder* (AE) son redes neuronales con una estructura simétrica, entrenados para reconstruir la entrada en la salida. La capa central representa la codificación de los datos de entrada. Por tanto, en este caso el interés no se encuentra en la salida de la red sino en la codificación obtenida mediante la misma. El objetivo del *autoencoder* es obtener una representación de los datos de entrada con determinadas características, que estarán determinadas por la arquitectura de la red. Podemos por tanto considerar la estructura básica reflejada en Figura 4.1: la entrada **x** que se aplica en la codificación **h** mediante el codificador (*encoder*), representado por f, y la reconstrucción **r**, obtenida a partir de **y** mediante el decodificador (*decoder*), representado por g [GBC16].

El *autoencoder* más básico es una red prealimentada profunda con esta estructura. Puesto que se copia la entrada **x** en la salida **r**, ambas tendrán la

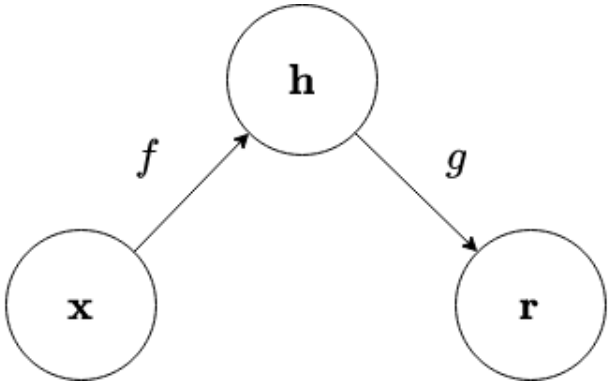

Figura 4.1: Estructura básica de un *autoencoder*

misma dimensión. La codificación **y** es variable, y puede tener una dimensión menor o mayor en función de las propiedades deseadas en la representación de los datos de entrada. Tanto el codificador como el decodificador pueden tener tantas capas como sean necesarias, normalmente dispuestas de manera simétrica. Un ejemplo de red prealimentada profunda de esta forma puede observarse en la Figura 4.2.

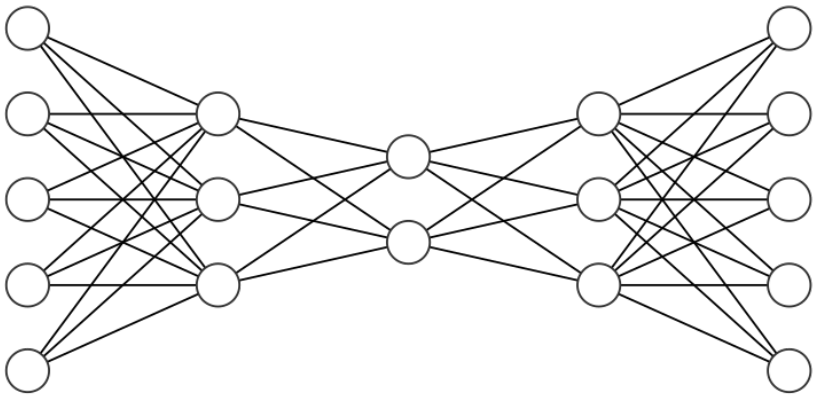

Figura 4.2: Estructura de un *autoencoder* como red prealimentada profunda

4.2.2.1. Arquitecturas

Los diferentes tipos de *autoencoders* pueden ser clasificados según múltiples criterios. En este apartado se repasan las diferentes estructuras que puede tomar la red. Todas ellas son simétricas, con el codificador y el decodificador

teniendo el mismo número de capas y de unidades en cada capa, en orden inverso. Puede considerarse que el codificador empieza en la entrada y tiene su salida en la codificación, que es la entrada del decodificador. La salida de este es la reconstrucción. Por tanto ambas estructuras comparten la capa de codificación. En función de las dimensiones de esta capa podemos considerar dos tipos de red:

- Incompleto (*undercomplete*): la capa de codificación tiene dimensión menor que la de la entrada (y por tanto también que la salida). El menor número de unidades supone una restricción en la red, por lo que durante el entrenamiento el *autoencoder* aprenderá una representación más compacta de la entrada. Cuando se tiene esta estructura la capa de codificación suele llamarse cuello de botella.
- Sobrecompleto (*overcomplete*): la capa de codificación tiene una dimensión igual o mayor a la de la entrada. Puede permitir que el *autoencoder* aprenda la función identidad para copiar la entrada en la salida, por lo que suelen aplicarse otras restricciones. Mediante estas puede obtenerse también una representación compacta de los datos.

Además del número de unidades por capa, también pueden clasificarse los *autoencoders* según la cantidad de capas que tienen. Así, podemos distinguir dos cateogrías:

- Superficial (*shallow*): cuenta solamente con las capas de entrada, codificación y salida. Es la estructura más simple posible, pues tiene una única capa oculta, la capa de codificación.
- Profundo (*deep*): tiene más de una capa oculta. Puede ser entrenado capa a capa como si se tratara de varios *autoencoders* superficiales superpuestos o como una única red profunda, como se explica en la Subsubsección 4.2.2.4.

4.2.2.2. Funciones de activación más comunes

Las unidades de activación generalmente utilizadas en los *autoencoder* son las ya vistas en Subsección 2.1.3. Cuando se utilizan unidades de activación lineal con una sola capa oculta de k unidades y se minimiza el error cuadrático medio, el modelo es equivalente a obtener las k componentes principales mediante PCA.

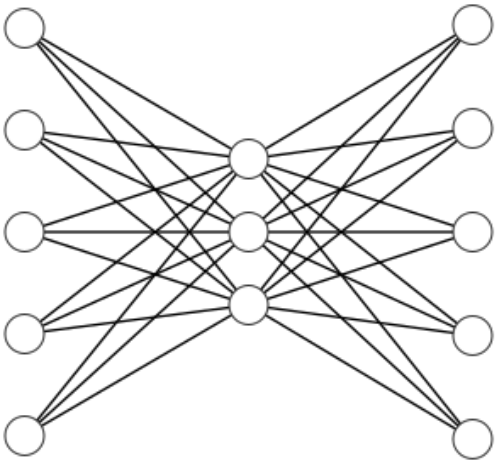

Figura 4.3: *Autoencoder* incompleto superficial

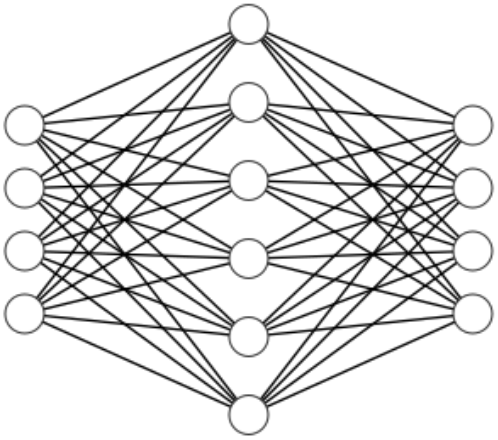

Figura 4.4: *Autoencoder* sobrecompleto superficial

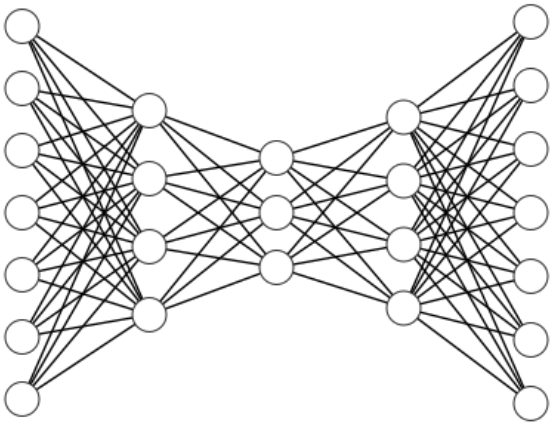

Figura 4.5: *Autoencoder* incompleto profundo

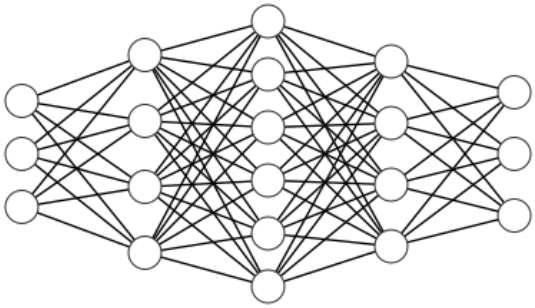

Figura 4.6: *Autoencoder* sobrecompleto profundo

Las unidades ReLU, muy utilizadas en muchos modelos de aprendizaje profundo, pueden resultar menos útiles para los *autoencoder*. Puesto que su salida es nula para todos los valores negativos dificultan el proceso de reconstruir la entrada en la salida. Existe una alternativa reciente, las unidades lineales exponenciales escaladas (*Scaled Exponential Linear Units*, SELU) [KUMH17], cuya función de activación es

$$g(\mathbf{z})_i = \lambda \begin{cases} \alpha(e^x - 1) & si \quad x \leq 0 \\ x & si \quad x > 0 \end{cases},$$

con $\lambda > 1$.

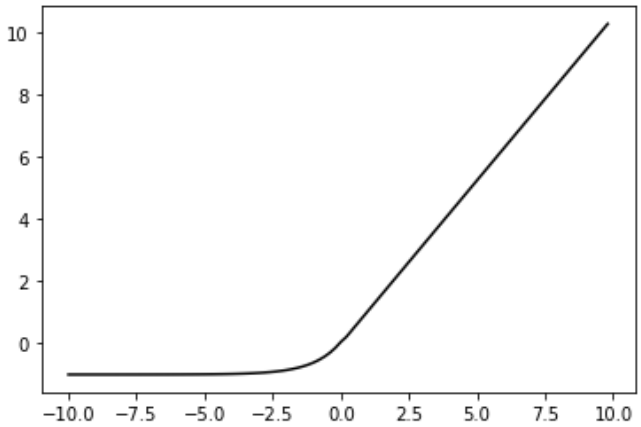

Figura 4.7: Gráfica de la función de activación de una SELU

Es por esto que, en general, las funciones de activación más usadas son las sigmoides, tanto la función logística como la tangente hiperbólica.

4.2.2.3. Función de coste

La función de coste $J(W, b; S)$ del *autoencoder* se calcula como la suma de las funciones de coste para cada elemento del conjunto de entrenamiento $L : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$. Así,

$$J(W, b; S) = \sum_{x \in S} L(x, (g \circ f)(x)),$$

donde S es el conjunto de entrenamiento y f y g las funciones del codificador y decodificador respectivamente, determinadas por los pesos W y los sesgos b. El objetivo será por tanto aprender W y b para minimizar J. La función L

suele tomarse como la log-verosimilitud negativa del ejemplo x dada la salida $(g \circ f)(x)$.

4.2.2.4. Entrenamiento

Para la optimización de *autoencoders* pueden aplicarse los algoritmos vistos en la Subsección 2.2.1 basados en el descenso del gradiente como el descenso del gradiente estocástico y variantes como AdaGrad, RMSProp y Adam. Para el cálculo del gradiente se usa el algoritmo de propagación hacia atrás, también visto en la Subsección 2.2.2.

Al igual que en los modelos de aprendizaje profundo del anterior capítulo, puede añadirse un término de regularización en la función de coste para evitar sobreajuste. Este término puede introducirse de distintas maneras, pero en general depende de los pesos para reducir su tamaño.

Conforme aumenta la profundidad de los *autoencoders* el proceso de entrenamiento depende cada vez más de la inicialización de los pesos, ya que su número aumenta. Esta inicialización puede realizarse superponiendo sucesivos *autoencoders* superficiales y entrenándolos capa a capa. Así, se comienza entrenando la primera capa oculta como si se tratara de un único *autoencoder* superficial. La segunda capa se entrena de igual manera, tomando la salida de la primera capa como entrada, y así hasta haber completado todas las capas hasta llegar a la de codificación. Con esto se consigue una inicialización de los pesos para todas las capas del codificador. Las capas del decodificador se inicializan con la trasposición de los pesos de su capa simétrica. Finalmente puede realizarse un entrenamiento normal.

4.2.3. Autoencoder variacional

Los modelos descritos hasta el momento tienen su principal utilidad en generar representaciones más compactas de los datos de entrada. En contraposición, el *autoencoder* variacional (*variational autoencoder*, VAE) es un modelo generativo ([KW13], [RMW14]). Los modelos generativos aprenden una distribución de probabilidad de la que generar nuevos datos diferentes a los observados. Los *autoencoder* son capaces de reconstruir datos codificados, pero no tienen por qué generar ejemplos válidos a partir de codificaciones arbitrarias. Los *autoencoder* variacionales aprenden un modelo mediante el que pueden generarse nuevas instancias.

Consideremos el conjunto de datos $\mathbf{X} = \{\mathbf{x}^{(i)}\}_{i=1}^{N}$ consistente en N muestras de una variable aleatoria X. Asumimos que los datos son generados por un proceso aleatorio que implica una variable aleatoria no observada Z. El proceso consiste de dos pasos: primero un valor $\mathbf{z}^{(i)}$ se genera de la distribución a priori $p_{\boldsymbol{\theta}^*}(\mathbf{z})$, y después el valor $\mathbf{x}^{(i)}$ se genera de una distribución condicional $p_{\boldsymbol{\theta}^*}(\mathbf{x}|\mathbf{z})$. Asumimos que la distribución a priori $p_{\boldsymbol{\theta}^*}(\mathbf{z})$ y la distribución condicionada $p_{\boldsymbol{\theta}^*}(\mathbf{x}|\mathbf{z})$ vienen de familias paramétricas $p_{\boldsymbol{\theta}}(\mathbf{z})$ y $p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z})$. En concreto, en el caso del *autoencoder* variacional, asumimos que la distribución a priori es una distribución normal multivariante de la forma $p_{\boldsymbol{\theta}}(\mathbf{z}) = \mathcal{N}(\mathbf{z}; \mathbf{0}, \mathbf{I})$ y la distribución $p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z})$ una distribución normal multivariante, que actuará como decodificador.

Tanto el codificador como el decodificador toman la forma de redes neuronales que modelan distribuciones multivariantes normales con covarianza diagonal. En el caso del decodificador

$$\log p(\mathbf{x}|\mathbf{z}) = \log \mathcal{N}(\mathbf{x}; \boldsymbol{\mu}, \sigma^2 \mathbf{I}),$$

con

$$\mu = \mathbf{W}_2\mathbf{h} + \mathbf{b}_2,$$

$$\log \sigma^2 = \mathbf{W}_3\mathbf{h} + \mathbf{b}_3,$$

$$\mathbf{h} = tanh(\mathbf{W}_1\mathbf{z} + \mathbf{b}_1).$$

Así, $\{\mathbf{W}_1, \mathbf{W}_2, \mathbf{W}_3, \mathbf{b}_1, \mathbf{b}_2, \mathbf{b}_3\}$ son los pesos y sesgos de la red.

Utilizando este modelo hay que resolver dos problemas:

- Aproximar los parámetros $\boldsymbol{\theta}$ que rigen la distribución. Pueden ser de interés por sí mismos, para analizar un proceso natural, pero también permiten imitar la variable aleatoria oculta para generar nuevos datos parecidos a los reales.
- Aproximar la distribución a posteriori de la variable latente $\mathbf{z}$ dado el valor de $\mathbf{x}$ y elegidos unos parámetros $\boldsymbol{\theta}$, $p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z})$. Con ello obtenemos la codificación de los datos en las variables latentes.

El principal inconveniente es que, a la hora de calcular la distribución a posteriori que usar en el codificador mediante la regla de Bayes

$$p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x}) = \frac{p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z})p_{\boldsymbol{\theta}}(\mathbf{z})}{p_{\boldsymbol{\theta}}(\mathbf{x})}$$

el cálculo de la integral de la distribución marginal $p_{\boldsymbol{\theta}}(\mathbf{x}) = \int p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z}) d\mathbf{z} = \int p_{\boldsymbol{\theta}}(\mathbf{z}) p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z}) d\mathbf{z}$ es intratable [KW19].

Para solucionarlo se introduce el modelo de reconocimiento $q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})$, una aproximación de la verdadera distribución a posteriori $p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x})$. En este caso tomamos

$$\log q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}^{(i)}) = \log \mathcal{N}(\mathbf{z}; \boldsymbol{\mu}^{(i)}, \boldsymbol{\sigma}^{2(i)}\mathbf{I}),$$

donde la media y la desviación típica, $\boldsymbol{\mu}^{(i)}$ y $\boldsymbol{\sigma}^{(i)}$ son las salidas de la red neuronal del codificador.

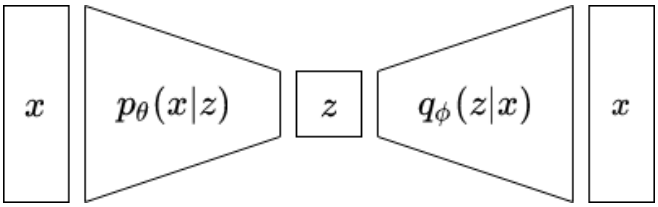

Figura 4.8: Estructura de un *autoencoder* variacional

4.2.3.1. Función de coste

Optimizando la función de coste se quiere conseguir que la aproximación $q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})$ sea lo más cercana posible a la verdadera distribución a posteriori $p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x})$. Para ello debe introducirse una medida de la similitud entre dos funciones de distribución, la divergencia de Kullback-Leibler [Kul97].

Definición 4.1. Dadas dos distribuciones de probabilidad P y Q sobre el mismo espacio de probabilidad, se define la *divergencia de Kullback-Leibler* como

$$D_{KL}(Q \parallel P) = E_Q \left[\log \frac{Q(x)}{P(x)} \right].$$

Proposición 4.1. *Dadas dos distribuciones de probabilidad P y Q sobre el mismo espacio de probabilidad, se cumple que*

$$D_{KL}(Q \parallel P) \geq 0.$$

Realizamos también el cálculo de la divergencia entre dos distribuciones normales, que será útil para calcular la expresión concreta de la función de coste.

Proposición 4.2. *Dadas las distribuciones de probabilidad* $p(\boldsymbol{x}) = N(\boldsymbol{x}; \boldsymbol{\mu}_1, \boldsymbol{\Sigma}_1)$ *y* $q(\boldsymbol{x}) = N(\boldsymbol{x}; \boldsymbol{\mu}_2, \boldsymbol{\Sigma}_2)$*, ambas de dimensión k, se cumple que* $D_{KL}(p(\boldsymbol{x}) \parallel q(\boldsymbol{x})) = \frac{1}{2}\left(\log \frac{|\boldsymbol{\Sigma}_1|}{|\boldsymbol{\Sigma}_2|} - k + tr(\boldsymbol{\Sigma}_2^{-1}\boldsymbol{\Sigma}_1) + (\boldsymbol{\mu}_2 - \boldsymbol{\mu}_1)^T \boldsymbol{\Sigma}_2^{-1}(\boldsymbol{\mu}_2 - \boldsymbol{\mu}_1)\right)$.

Demostración. Comenzamos tomando el logaritmo de las funciones de densidad:

$$\log p(\mathbf{x}) = -\frac{k}{2}\log(2\pi) - \frac{1}{2}|\boldsymbol{\Sigma}_1| - \frac{1}{2}(\mathbf{x} - \boldsymbol{\mu}_1)^T \boldsymbol{\Sigma}_1^{-1}(\mathbf{x} - \boldsymbol{\mu}_1),$$

$$\log p(\mathbf{x}) = -\frac{k}{2}\log(2\pi) - \frac{1}{2}|\boldsymbol{\Sigma}_2| - \frac{1}{2}(\mathbf{x} - \boldsymbol{\mu}_2)^T \boldsymbol{\Sigma}_2^{-1}(\mathbf{x} - \boldsymbol{\mu}_2).$$

Podemos entonces sustituir en la expresión de la divergencia de Kullback-Leibler:

$$D_{KL}(Q \parallel P) = E_p\left[\log \frac{p(x)}{q(x)}\right] = E_p\left[\log p(x) - \log q(x)\right] =$$

$$E_p\left[-\frac{1}{2}\log \frac{|\boldsymbol{\Sigma}_2|}{|\boldsymbol{\Sigma}_1|} - \frac{1}{2}(\mathbf{x} - \boldsymbol{\mu}_1)^T \boldsymbol{\Sigma}_1^{-1}(\mathbf{x} - \boldsymbol{\mu}_1) + \frac{1}{2}(\mathbf{x} + \boldsymbol{\mu}_2)^T \boldsymbol{\Sigma}_2^{-1}(\mathbf{x} - \boldsymbol{\mu}_2)\right] =$$

$$E_p\left[-\frac{1}{2}\log \frac{|\boldsymbol{\Sigma}_2|}{|\boldsymbol{\Sigma}_1|}\right] - E_p\left[\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu}_1)^T \boldsymbol{\Sigma}_1^{-1}(\mathbf{x} - \boldsymbol{\mu}_1)\right] + E_p\left[\frac{1}{2}(\mathbf{x} + \boldsymbol{\mu}_2)^T \boldsymbol{\Sigma}_2^{-1}(\mathbf{x} - \boldsymbol{\mu}_2)\right].$$

Calculando por separado los sumandos, en el primero podemos quitar la esperanza de la expresión puesto que se trata de una constante. En el segundo

$$E_p\left[\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu}_1)^T \boldsymbol{\Sigma}_1^{-1}(\mathbf{x} - \boldsymbol{\mu}_1)\right] = E_p\left[tr\left(\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu}_1)^T \boldsymbol{\Sigma}_1^{-1}(\mathbf{x} - \boldsymbol{\mu}_1)\right)\right] =$$

$$E_p\left[tr\left(\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu}_1)^T (\mathbf{x} - \boldsymbol{\mu}_1)\boldsymbol{\Sigma}_1^{-1}\right)\right] = tr\left(E_p\left[\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu}_1)^T (\mathbf{x} - \boldsymbol{\mu}_1)\boldsymbol{\Sigma}_1^{-1}\right]\right) =$$

$$tr\left(E_p\left[(\mathbf{x} - \boldsymbol{\mu}_1)^T (\mathbf{x} - \boldsymbol{\mu}_1)\right]\frac{1}{2}\boldsymbol{\Sigma}_1^{-1}\right) = tr\left(\boldsymbol{\Sigma}_1 \frac{1}{2}\boldsymbol{\Sigma}_1^{-1}\right) = tr\left(\frac{1}{2}I_k\right) = \frac{k}{2}.$$

En el tercer sumando

$$E_p\left[\frac{1}{2}(\mathbf{x} + \boldsymbol{\mu}_2)^T \boldsymbol{\Sigma}_2^{-1}(\mathbf{x} - \boldsymbol{\mu}_2)\right] =$$

$$E_p\left[\frac{1}{2}((\mathbf{x}-\boldsymbol{\mu}_1)+(\boldsymbol{\mu}_1-\boldsymbol{\mu}_2))^T\boldsymbol{\Sigma}_2^{-1}((\mathbf{x}-\boldsymbol{\mu}_1)+(\boldsymbol{\mu}_1-\boldsymbol{\mu}_2))\right]=$$

$$E_p\left[\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu}_1)^T\boldsymbol{\Sigma}_2^{-1}(\mathbf{x}-\boldsymbol{\mu}_1)+(\mathbf{x}-\boldsymbol{\mu}_1)^T\boldsymbol{\Sigma}_2^{-1}(\boldsymbol{\mu}_1-\boldsymbol{\mu}_2)+\frac{1}{2}(\boldsymbol{\mu}_1-\boldsymbol{\mu}_2)^T\boldsymbol{\Sigma}_2^{-1}(\boldsymbol{\mu}_1-\boldsymbol{\mu}_2)\right]=$$

$$\frac{1}{2}tr(\boldsymbol{\Sigma}_2^{-1}\boldsymbol{\Sigma}_1)+\frac{1}{2}(\boldsymbol{\mu}_1-\boldsymbol{\mu}_2)^T\boldsymbol{\Sigma}_2^{-1}(\boldsymbol{\mu}_1-\boldsymbol{\mu}_2)+0.$$

Sustituyendo cada uno de los sumandos se obtiene el resultado buscado.

□

Por lo tanto queremos minimizar la expresión

$$D_{KL}(q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}) \parallel p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x})) = E_{z\sim q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}\left[\log\frac{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}{p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x})}\right]=$$

$$E_{z\sim q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}\left[\log q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})-\log p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x})\right].$$

Sustituyendo $p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x})$ según la regla de Bayes obtenemos

$$D_{KL}(q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}) \parallel p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x})) = E_{z\sim q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}\left[\log q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})-\log\frac{p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z})p_{\boldsymbol{\theta}}(\mathbf{z})}{p_{\boldsymbol{\theta}}(\mathbf{x})}\right]=$$

$$E_{z\sim q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}\left[\log q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})-\log p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z})-\log p_{\boldsymbol{\theta}}(\mathbf{z})+\log p_{\boldsymbol{\theta}}(\mathbf{x})\right].$$

Como $\log p_{\boldsymbol{\theta}}(\mathbf{x})$ no depende de $\mathbf{z}$ puede extraerse de la esperanza:

$$D_{KL}(q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}) \parallel p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x})) - \log p_{\boldsymbol{\theta}}(\mathbf{x}) =$$

$$E_{z\sim q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}\left[\log q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})-\log p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z})-\log p_{\boldsymbol{\theta}}(\mathbf{z})\right].$$

Utilizando la linealidad de la esperanza:

$$\log p_{\boldsymbol{\theta}}(\mathbf{x}) - D_{KL}(q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}) \parallel p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x})) =$$

$$E_{z\sim q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}\left[\log p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z})\right] - E_{z\sim q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}\left[\log q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})-\log p_{\boldsymbol{\theta}}(\mathbf{z})\right] =$$

$$E_{z\sim q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}\left[\log p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z})\right] - D_{KL}(q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}) \parallel p_{\boldsymbol{\theta}}(\mathbf{z})).$$

La expresión anterior recibe el nombre de cota inferior de la evidencia (*evidence lower bound*, ELBO), ya que a $\log p_{\boldsymbol{\theta}}(\mathbf{x})$ se le llama evidencia y la divergencia de Kullback-Leibler siempre positiva. Por lo tanto la ELBO es siempre una cota inferior de la evidencia. Maximizando la expresión conseguimos, por un

lado, maximizar $p_{\boldsymbol{\theta}}(\mathbf{x})$, lo que se traduce en una mejor generación de ejemplos, y por otro minimizar la divergencia entre la aproximación $q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})$ y la verdadera distribución a priori $p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x})$) [KW19]. Así, la función de coste a minimizar es

$$L(\boldsymbol{\theta}, \boldsymbol{\phi}; \mathbf{x}) = -E_{z\sim q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}\left[\log p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z})\right] + D_{KL}(q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}) \parallel p_{\boldsymbol{\theta}}(\mathbf{z})).$$

Podemos obtener la expresión concreta de la función de coste sustituyendo los valores de las distribuciones conocidas en la divergencia de Kullback-Leibler. Previamente se fijó que $p_{\boldsymbol{\theta}}(\mathbf{z}) = \mathcal{N}(\mathbf{z}; \mathbf{0}, \mathbf{I})$ y $q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}) = \mathcal{N}(\mathbf{z}; \boldsymbol{\mu}, \boldsymbol{\sigma}^2\mathbf{I})$. Aplicando la Proposición 4.2.3.1

$$D_{KL}(q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}) \parallel p_{\boldsymbol{\theta}}(\mathbf{x})) = D_{KL}(\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\sigma}^2\mathbf{I}) \parallel \mathcal{N}(\mathbf{0}, \mathbf{I})) =$$

$$\frac{1}{2}\left(-\log|\boldsymbol{\sigma}^2\mathbf{I}| - k + tr(\boldsymbol{\sigma}^2\mathbf{I}) + \boldsymbol{\mu}^T\boldsymbol{\mu}\right)$$

donde k es la dimensión de las distribuciones. Podemos simplificar esta expresión como

$$D_{KL}(q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}) \parallel p_{\boldsymbol{\theta}}(\mathbf{x})) = \frac{1}{2}\left(-\log|\boldsymbol{\sigma}^2\mathbf{I}| - k + tr(\boldsymbol{\sigma}^2\mathbf{I}) + \boldsymbol{\mu}^T\boldsymbol{\mu}\right) =$$

$$\frac{1}{2}\left(-\log(\prod_{j=1}^{k}\sigma_j^2) - \sum_{j=1}^{k}1 + \sum_{j=1}^{k}\sigma_j^2 + \sum_{j=1}^{k}\mu_j^2\right) =$$

$$\frac{1}{2}\left(-\sum_{j=1}^{k}\log(\sigma_j^2) - \sum_{j=1}^{k}1 + \sum_{j=1}^{k}\sigma_j^2 + \sum_{j=1}^{k}\mu_j^2\right) =$$

$$\frac{1}{2}\sum_{j=1}^{k}\left(-log(\sigma_j^2) - 1 + \sigma_j^2 + \mu_j^2\right).$$

Sustituyendo en la expresión original, la función de coste definitiva es

$$L(\boldsymbol{\theta}, \boldsymbol{\phi}; \mathbf{x}) = -E_{z\sim q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}\left[\log p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z})\right] + \frac{1}{2}\sum_{j=1}^{k}\left(-log(\sigma_j^2) - 1 + \sigma_j^2 + \mu_j^2\right).$$

4.2.3.2. Optimización

Existe un problema para aplicar los algoritmos de optimización vistos en la Subsección 2.2.1. La función de coste incluye un proceso de muestreo de la distribución $q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})$, por lo que no puede calcularse su gradiente. Para poder calcularlo se utiliza, en lugar de $E_{z\sim q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}[\log p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z})]$, un estimador mediante el truco de reparametrización [KW13].

Este consiste en expresar $\mathbf{z}^{(l)} \sim q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})$ mediante una transformación diferenciable de una variable aleatoria $\epsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ que actúa como ruido,

$$\mathbf{z}^{(l)} = g_{\boldsymbol{\phi}}(\mathbf{x}, \boldsymbol{\epsilon}^{(l)}) = \boldsymbol{\mu} + \boldsymbol{\sigma} \odot \boldsymbol{\epsilon}^{(l)},$$

donde $\boldsymbol{\mu}$ y $\boldsymbol{\sigma}$ son tales que $q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}) = \mathcal{N}(\mathbf{z}; \boldsymbol{\mu}, \boldsymbol{\sigma}^2\mathbf{I})$. Así, el estimador de la función de coste para el punto $\mathbf{x}^{(i)}$ será

$$\tilde{L}(\boldsymbol{\theta}, \boldsymbol{\phi}; \mathbf{x}) = -\frac{1}{\mathcal{L}} \sum_{l=1}^{\mathcal{L}} \log p_{\boldsymbol{\theta}}(\mathbf{x}^{(i)}|\mathbf{z}^{(i,l)}) + \frac{1}{2} \sum_{j=1}^{k} \left(-log(\sigma_j^2) - 1 + \sigma_j^2 + \mu_j^2 \right).$$

Dado un conjunto $\mathbf{X}$ con N datos, puede construirse un estimador del conjunto entero, basado en minilotes:

$$\tilde{L}(\boldsymbol{\theta}, \boldsymbol{\phi}; \mathbf{X}) = \tilde{L}^M(\boldsymbol{\theta}, \boldsymbol{\phi}; \mathbf{X}^M) = \frac{N}{M} \sum_{i=1}^{M} \tilde{L}^M(\boldsymbol{\theta}, \boldsymbol{\phi}; \mathbf{x}^{(i)})$$

donde el minilote $\mathbf{X}^M = \{\mathbf{x}^{(i)}\}_{i=1}^M$ es un subconjunto de M elementos de $\mathbf{X}$ elegidos aleatoriamente. La aplicación en el entrenamiento se detalla en el algoritmo 9.

Algoritmo 9: Versión con minilotes del algoritmo .

$\boldsymbol{\theta}$, $\boldsymbol{\phi} \leftarrow$ Inicializa parámetros;
mientras *los parámetros* $\boldsymbol{\theta}$, $\boldsymbol{\phi}$ *no convergen* **hacer**
  $\mathbf{X}^M \leftarrow$ Minilote aleatorio de M datos;
  $\boldsymbol{\epsilon} \leftarrow$ Muestras aleatorias de la distribución de ruido;
  $g \leftarrow \nabla_{\boldsymbol{\theta},\boldsymbol{\phi}} \tilde{L}^M(\boldsymbol{\theta}, \boldsymbol{\phi}; \mathbf{x}^{(i)})$;
  $\boldsymbol{\theta}$, $\boldsymbol{\phi} \leftarrow$ Actualiza los parámetros mediante un algoritmo basado en gradiente
fin
$\mathbf{y} \leftarrow \mathbf{h}_\ell$;

5 MusicVAE

Magenta es un proyecto de código abierto cuyo objetivo es el desarrollo de herramientas de *machine learning* para la creación artística, especialmente la creación musical. Estas herramientas se construyen sobre la biblioteca TensorFlow. También han publicado numerosos conjuntos de datos utilizados en el entrenamiento de estas herramientas.

MusicVAE [RER+18] es un modelo desarrollado por Magenta y cuyo objetivo es la codificación y generación de melodías musicales. Se basa en un *autoencoder* variacional cuyo codificador y decodificador toman la forma de redes recurrentes para el tratamiento de secuencias.

5.1. Modelo

Una gran variedad de estructuras de redes neuronales han sido utilizadas como codificador y decodificador de un *autoencoder* variacional. En este caso se han elegido redes recurrentes para desempeñar dicho cometido. El codificador, $q_\lambda(\mathbf{z}|\mathbf{x})$ es una red neuronal recurrente que procesa una secuencia de entrada $x = \{\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, ..., \mathbf{x}^{(T)}\}$ y produce una secuencia de estados ocultos $\mathbf{h}_1, ..., \mathbf{h}_T$. Los parámetros de la distribución sobre las variables latentes $\mathbf{z}$ son una función de h_T. El decodificador $q_\theta(\mathbf{x}|\mathbf{z})$ utiliza el vector latente $\mathbf{z}$ para generar la configuración inicial de la red recurrente, que produce la secuencia de salida $y = \{\mathbf{y}^{(1)}, \mathbf{y}^{(2)}, ..., \mathbf{y}^{(T)}\}$.

Existen dos principales inconvenientes en esta estructura. El primero es que las redes neuronales recurrentes son por sí mismas un modelo capaz de reconstruir secuencias. En consecuencia el decodificador puede resultar suficiente para producir un modelo efectivo de los datos y podría ignorar las variables latentes para ello. Esto puede implicar que el término de la convergencia de Kullback-Leibler en el ELBO se establezca trivialmente en cero, con lo que el modelo no estaría actuando como un decodificador. El segundo inconveniente es que el modelo debe comprimir una secuencia completa en

un único vector latente, lo cuál puede resultar problemático conforme aumenta el tamaño de la secuencia [BCB14]. El modelo tiene en cuenta estas dos limitaciones e incluye modificaciones en su estructura para evitarlas.

5.1.1. Codificador bidireccional

En el codificador $q_\lambda(\mathbf{z}|\mathbf{x})$ se utiliza una red LSTM (Subsubsección 3.2.2.1) bidireccional (Subsección 3.1.2) de dos capas (Subsección 3.1.3). Se procesa la secuencia para obtener dos vectores de estado oculto finales de la segunda capa, $\overrightarrow{\mathbf{h}_T}$ y $\overleftarrow{\mathbf{h}_T}$. Estos dos vectores son concatenados en un único vector $\mathbf{h}_T$ y sirven de entrada para una capa que produce los parámetros de la distribución como

$$\mu = \mathbf{W}_{h\mu}\mathbf{h}_T + \mathbf{b}_\mu$$
$$\sigma = \log(\exp(\mathbf{W}_{h\sigma}\mathbf{h}_T + \mathbf{b}_\sigma) + 1)$$

donde $\mathbf{W}_{h\mu}$, $\mathbf{W}_{h\sigma}$ son las matrices de pesos y $\mathbf{b}_{h\mu}$, $\mathbf{b}_{h\sigma}$ los vectores de sesgos de la capa. El tamaño del vector de estado oculto de LSTM es de 2048 para cada capa y se tienen 512 dimensiones latentes.

5.1.2. Decodificador jerárquico

En el decodificador se establece una modificación en la estructura canónica de una red recurrente para obtener mejores resultados en el muestreo y reconstrucción de secuencias largas, mitigando los problemas vistos en la Sección 5.1.

La estructura propuesta es la de un decodificador recurrente jerárquico. Asumiendo que la secuencia de entrada x (y también la secuencia objetivo de salida) puede segmentarse en U subsecuencias no superpuestas y_u con final en el punto i_u, tales que

$$y_u = \{x_{i_u}, x_{i_u+1}, x_{i_u+2}, ..., x_{i_{u+1}-1}\},$$
$$x = \{y_1, y_2, ..., y_U\}$$

donde se define el caso especial $i_{U+1} = T$. El vector latente $\mathbf{z}$ pasa a través de una capa con función de activación *tanh* para obtener el estado inicial de una

red recurrente "directora". La red directora produce U vectores $c = \{\mathbf{c}_1, ..., \mathbf{c}_U\}$, uno por cada subsecuencia. Para la red directora se utiliza una red recurrente LSTM de dos capas unidireccional con tamaño de estado oculto de 1024 y dimensión de salida de 512.

Una vez que la capa directora produce la secuencia de vectores c, cada uno es tratado individualmente por una capa compartida con función de activación *tanh* que produce los estados iniciales para una última capa recurrente de decodificación. Esta capa produce una secuencia de distribuciones sobre cada unidad de información (*token*) de la salida, en este caso sobre cada corchea para cada subsecuencia y_u mediante una capa de salida softmax. En cada paso de esta última capa de decodificación el vector $\mathbf{c}_u$ de la capa directora correspondiente se concatena con el *token* anterior para ser utilizados como entrada. Para este proceso se usan dos capas de LSTM con 1024 unidades por capa.

Al limitar la capacidad del decodificador de transmitir información se le fuerza a que utilice la información de los vectores latentes. En este caso la única manera que tiene el decodificador de obtener información a largo plazo es mediante los vectores obtenidos de la capa directora, que depende directamente de los vectores latentes.

5.1.3. Modelo para múltiples secuencias

A diferencia de otras fuentes de datos secuenciales, como textos, en los que solo existe una secuencia simultánea, en la música normalmente se combinan múltiples secuencias (como múltiples intérpretes tocando diferentes partes o instrumentos de una canción a la vez). Es por ello que MusicVAE incluye también un modelo como el descrito, pero que produce tres secuencias simultáneas: percusión, bajo y melodía. Para ello en la salida se producen tres distribuciones separadas para cada uno de los instrumentos. Cada una de estas distribuciones se produce mediante un decodificador diferente.

5.2. Entrenamiento

Para el entrenamiento se utilizaron archivos MIDI, que contienen instrucciones para que pueda ser reproducida cada nota de cada instrumento de una

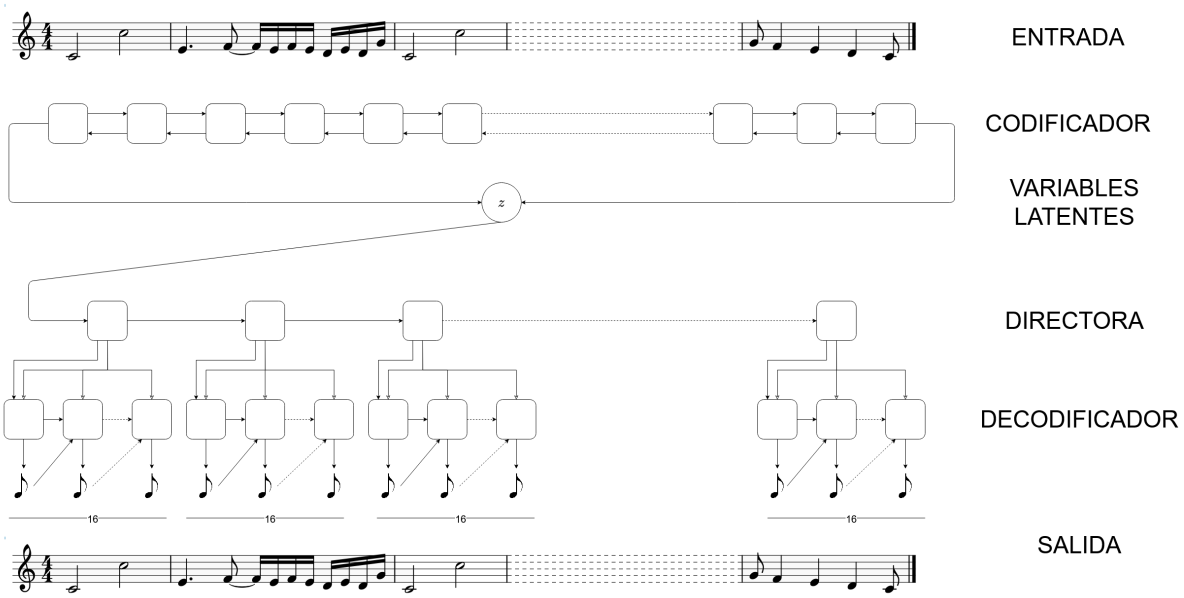

Figura 5.1: Estructura del modelo MusicVAE

canción, así como información de medida (del tiempo). Fueron usados alrededor de 1,5 millones de estos archivos, de los cuales fueron extraídas melodías de 2 compases, melodías de 16 compases, percusiones de 2 y 16 compases y tríos (melodía, percusión y bajo) de 16 compases. Todos los archivos utilizados se encuentran en el compás de $4/4$ y la unidad mínima de información considerada es la corchea (un dieciseisavo de compás). En total se obtuvieron 28 millones de ejemplos diferentes para melodías de dos compases, $19,5$ millones para melodías de dieciséis compases, $3,8$ millones para percusiones de dos compases, $11,4$ millones para percusiones de dieciséis compases y $9,4$ millones para tríos.

Las melodías monofónicas se modelaron con un espacio de salida de dimensión 130, con 128 dimensiones representando el evento de "pulsar una nota" para cada uno de los 128 tonos contemplados, un evento de "dejar de tocar una nota" y un evento de "silencio". Para la percusión se agruparon las sesenta y una clases de percusión del standard MIDI en 9 clases y se representaron todas las posibles combinaciones de golpes con 512 elementos categóricos. En ambos casos se consideran dieciséis eventos por compás, por lo que para los datos de dos compases se tiene $T = 32$ y para los de dieciséis compases se tiene $T = 256$. En todos los casos se utiliza $U = 16$, por lo que cada subsecuencia representa un compás completo. Todos los modelos se entrenan

mediante el algoritmo Adam (7) con una tasa de aprendizaje entre 10^{-3} y 10^{-5}, con ratio de decaimiento exponencial de $0,9999$ y tamaño de lotes de 512. Los modelos de dos compases se entrenaron con 50000 actualizaciones del gradiente y los de dieciséis compases con 100000.

5.3. Propiedades

De la utilización del espacio latente para la representación de melodías se derivan ciertas propiedades que pueden ser explotadas en la utilización creativa de la herramienta que supone el modelo para la generación de música.

5.3.1. Interpolación

Dado un punto del espacio latente que se identifica con un dato real, los puntos cercanos en el espacio latente deberían corresponderse con datos reales parecidos. Esto implica que todos los puntos a través de una curva continua que conecta dos puntos en el espacio latente deberían ser identificables a una serie de ejemplos reales que produzcan una interpolación suave entre los ejemplos de los extremos. Para ello se requiere que el espacio latente no tenga regiones que no se identifiquen con ejemplos realistas. Estos requerimientos deberían cumplirse si ambos términos del ELBO son suficientemente bajos.

Una comprobación práctica de esta propiedad es realizar una interpolación entre puntos del espacio latente y comprobar si el resultado con sus correspondientes datos reales es adecuado. Dados $\mathbf{z}_1$ y $\mathbf{z}_2$ vectores latentes correspondientes a los datos $\mathbf{x}_1$ y $\mathbf{x}_2$, se puede realizar la interpolación lineal

$$\mathbf{c}_\alpha = \alpha\mathbf{z}_1 + (1-\alpha)\mathbf{z}_2$$

para $\alpha \in [0,1]$. Se realizará la interpolación satisfactoriamente si para todo α el punto $p_\theta(\mathbf{x}|\mathbf{c}_\alpha)$ es un punto realista y la transición entre $p_\theta(\mathbf{x}|\mathbf{c}_0)$ y $p_\theta(\mathbf{x}|\mathbf{c}_1)$ se realiza de manera suave conforme se varía α.

5.3.2. Aritmética de vectores de atributos

Otra propiedad deseada es la obtención de vectores de atributos, que son calculados como la media de los vectores latentes con una determinada característica. Si el modelo es capaz de generar dichos vectores, entonces podría

cargarse un ejemplo con determinada característica y, sustrayendo el atributo correspondiente a dicha característica, eliminarla del dato real. Este mismo proceso debe poder realizarse para añadir una característica, sumando el vector de atributo.

Durante los experimentos realizados con el modelo se tomaron cinco características que pueden ser medidas en las melodías: pertenencia a las escala de Do diatónica, densidad de notas, intervalo medio y presencia de síncopas en la subdivisión de corcheas y semicorcheas. Las características fueron medidos en conjuntos de 370000 ejemplos y para cada una el conjunto fue ordenado en función de la cantidad de presencia del atributo. Se dividió entonces el conjunto en cuartiles y se calculó el vector del atributo restando a la media del vector lantente cuartil superior la media del vector latente del cuartil inferior. Después de generaron 256 muestras aleatorias y se comprobó que sumar el vector obtenido producía un aumento en la característica correspondiente para la melodía. De igual manera la sustracción del vector producía una reducción en la medida de dicha característica.

6 AutoLoops

Como ejemplo de aplicación del contenido teórico del trabajo se ha desarrollado la herramienta AutoLoops, basada en el modelo MusicVAE de Magenta. Permite la manipulación sencilla del espacio latente de melodías de dos compases, permitiendo la codificación de una melodía ya existente y la exploración del espacio cercano para la generación de nuevos ejemplos relacionados.

La herramienta se inspira en las múltiples demostraciones ya disponibles en la web que implementan los modelos desarrollados por Magenta, pero con pretensión de aplicación al proceso real de composición musical. Desde su concepción inicial se establece no solo como un ejemplo de las capacidades del modelo, sino también como una herramienta que pueda ser utilizada directamente por artistas en el contexto real de la composición y producción musical.

En cuanto a su utilización, la herramienta se basa en el método de trabajo que muchas veces se utiliza para el diseño sonoro en los sintetizadores. Estos permiten la generación y modificación de ondas sonoras para la creación de timbres. Muchas veces es difícil entender la repercusión exacta que tendrá en el sonido producido la modificación de alguna de estas características, o la interacción entre varias de estas modificaciones. Es por ello que muchas veces la construcción de un sonido se realiza explorando el espacio que las características ofrecen mediante el método de prueba y error.

Se trata por tanto de traducir este método al trabajo con melodías. Mediante el modelo MusicVAE (Sección 5.1) se obtienen las características de una melodía, aunque su significado es oculto. El artista puede cargar su propia melodía y generar nuevas a partir de la misma modificando las características que se le proporcionan. Podrá generar una secuencia de melodías de dos compases o una sola melodía que podrá exportar como archivo MIDI, que puede usarse en la producción de una canción.

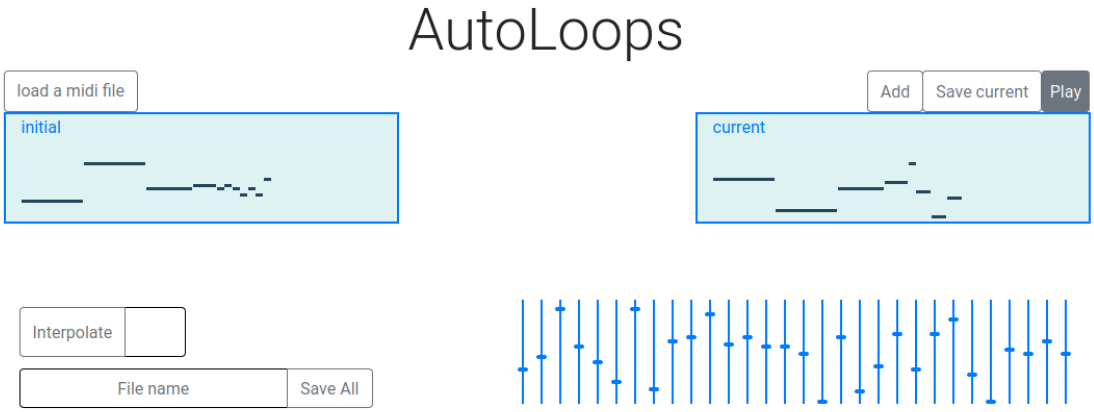

Figura 6.1: Interfaz de AutoLoops

6.1. Marco de trabajo

La herramienta se ha desarollado en forma de aplicación web para la que se ha utilizado HTML y CSS para el diseño de la interfaz y JavaScript para el funcionamiento interno.

6.1.1. HTML

Para la maquetación de la interfaz se ha utilizado HTML, lenguaje de marcado estándar a cargo del *World Wide Web Consortium* (W3C). Sobre este se ha utilizado la biblioteca Bootstrap para el diseño con sistema de rejilla. Puede incluirse dicha biblioteca mediante la siguiente línea en la cabecera.

```
<link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/
    bootstrap/4.5.2/css/bootstrap.min.css">
```

Además también se ha utilizado una hoja de estilos CSS.

6.1.2. CSS

Mientras que HTML tiene el contenido de la interfaz, la hoja CSS contiene el diseño de los elementos de la misma. Para su inclusión se debe añadir la siguiente línea en la cabecera.

```
<link rel="stylesheet" href="assets/css/index.css">
```

6.1.3. JavaScript

Para la programación del funcionamiento de la aplicación se utiliza JavaScript, un lenguaje de programación interpretado utilizado principalmente en páginas web y aplicaciones de servidor. Para exportar los archivos MIDI necesarios se ha utilizado la biblioteca FileSaver, que se incluye mediante la siguiente línea en la cabecera del archivo HTML.

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/FileSaver.js
    /1.3.8/FileSaver.min.js" defer></script>
```

Para la visualización de datos se utiliza la biblioteca D3, que se incluye mediante la siguiente línea en la cabecera del archivo HTML.

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/d3/5.15.0/d3.js
    "></script>
```

Por último, y aquella con uso más extensivo por ser la que aporta el núcleo de la funcionalidad del programa, se ha utilizado la biblioteca Magenta.js.

6.1.4. Biblioteca Magenta.js

La biblioteca Magenta.js provee al desarrollador de una API de alto nivel para la aplicación de modelos de aprendizaje automático, especialmente aprendizaje profundo, para el modelado y manipulación de datos relacionados con el arte como imágenes y música. Está desarrollada sobre TensorFlow, una biblioteca para la creación de modelos de aprendizaje automático [RHS18].

El paquete *magenta/music* contiene una API para JavaScript para la interacción con modelos para la generación de música. Puede incluirse añadiendo la siguiente línea en la cabecera del archivo HTML.

```
<script src="https://cdn.jsdelivr.net/npm/@magenta/music@1.12.1/dist
    /magentamusic.min.js"></script>
```

El paquete contiene dos funcionalidades principales: procesamiento de datos y API para modelos, incluídos modelos pre-entrenados.

6.1.4.1. Procesamiento de datos

Se establece la clase *NoteSequence* como representación de partituras musicales. Esta representación almacena aspectos fundamentales de una secuencia de notas como son tiempos, tonos e instrumentos, pero también metadatos como etiquetas de secciones o información sobre acordes. El paquete también contiene una re-implementación de la librería *NoteSequence* de Magenta Python, que incluye funcionalidades para convertir MIDI en *NoteSequence* y viceversa, así como para la síntesis y reproducción de sonido. También existe la posibilidad de convertir las *NoteSequence* en tensores de la biblioteca TensorFlow, que son utilizados como entrada y salida de los modelos.

6.1.4.2. Interfaces de modelos

La biblioteca también incluye interfaces para los múltiples modelos de Magenta. Entre los modelos dedicados a la música se encuentran interfaces para MusicRNN [SO17], MidiMe [DER19], Piano Genie [DSD18], GANSynth [EAC+19], y el modelo que se utiliza en este proyecto, MusicVAE.

Con respecto a este último modelo se implementan métodos que incluyen *encode* y *decode* para codificar *NoteSequence* en vectores latentes y al revés. Además se añaden los métodos *interpolate* para facilitar la operación de interpolación entre melodías (Subsección 5.3.1) y *sample* para generar muestras de melodías.

Para evitar que el desarrollador tenga que llevar a cabo la costosa operación del entrenamiento del modelo se facilitan también archivos con los pesos del modelo. Estos se facilitan en forma de archivos de configuración JSON. En este caso utilizamos el modelo de MusicVAE para melodías de dos compases, por lo que cargamos el modelo mediante

```
const model = new mm.MusicVAE('https://storage.googleapis.com/
   magentadata/js/checkpoints/music_vae/mel_2bar_small');
```

6.2. Uso de la aplicación

Para abrir la aplicación el usuario solo debe descargar los archivos y abrir *index.html* con un navegador. Hecho esto se visualizará la interfaz de la Figura 6.1.

Para comenzar a utilizar la herramienta el usuario deberá cargar una melodía en formato MIDI monofónica de dos compases. Esta acción puede llevarse a cabo mediante el botón *load a midi file*, que permitirá elegir el archivo sobre el que se quiere trabajar. Una vez la melodía ha sido introducida se mostrará en los cuadros *initial* y *current*.

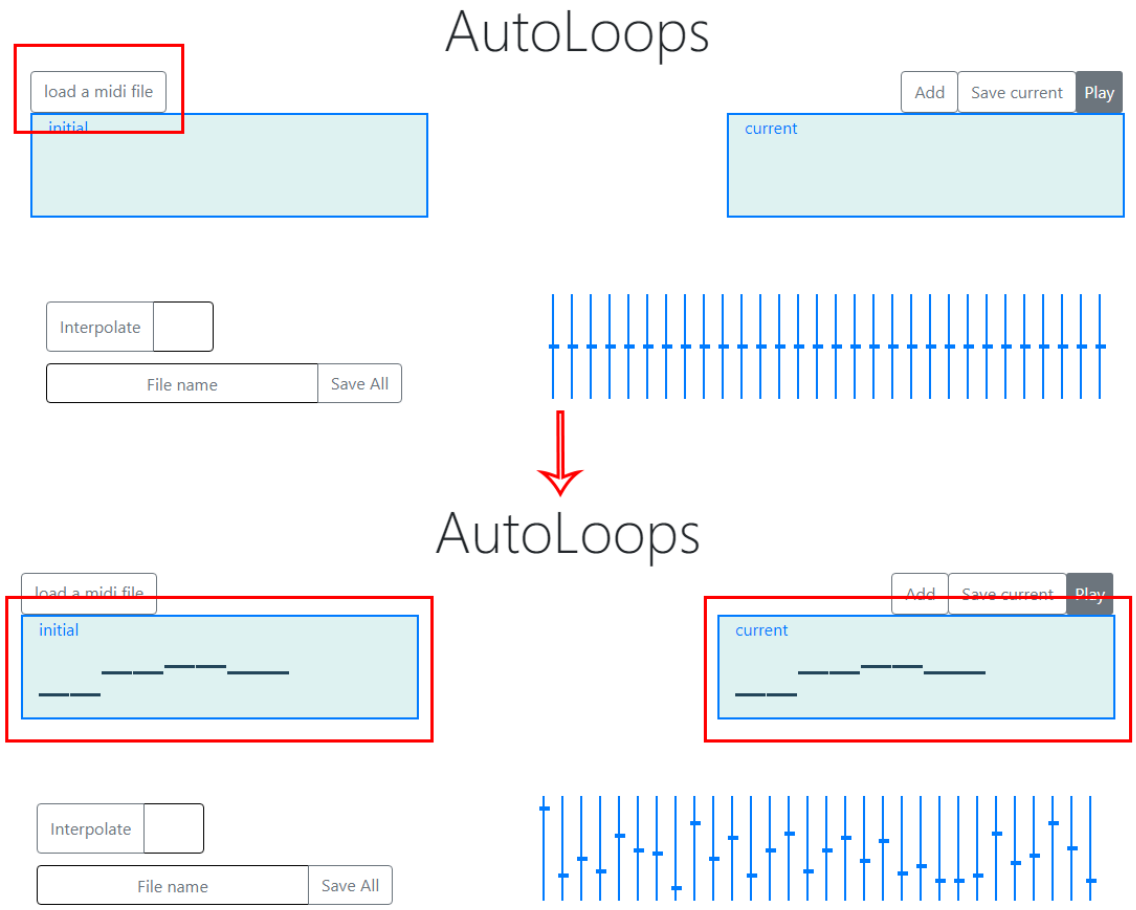

Figura 6.2: Cargar una melodía a AutoLooops

Una vez la melodía está cargada puede comenzarse a generar nuevas melodías a partir de la misma. Los controles deslizantes reflejarán las características en el espacio latente de la melodía que se ha cargado. El usuario puede modificarlas y con ello se generarán nuevas melodías. El cambio podrá apreciarse visualmente en el cuadro *current*, que mostrará la melodía correspondiente a la codificación actual marcada por los controles deslizantes. La melodía actual podrá también escucharse mediante el botón *play*.

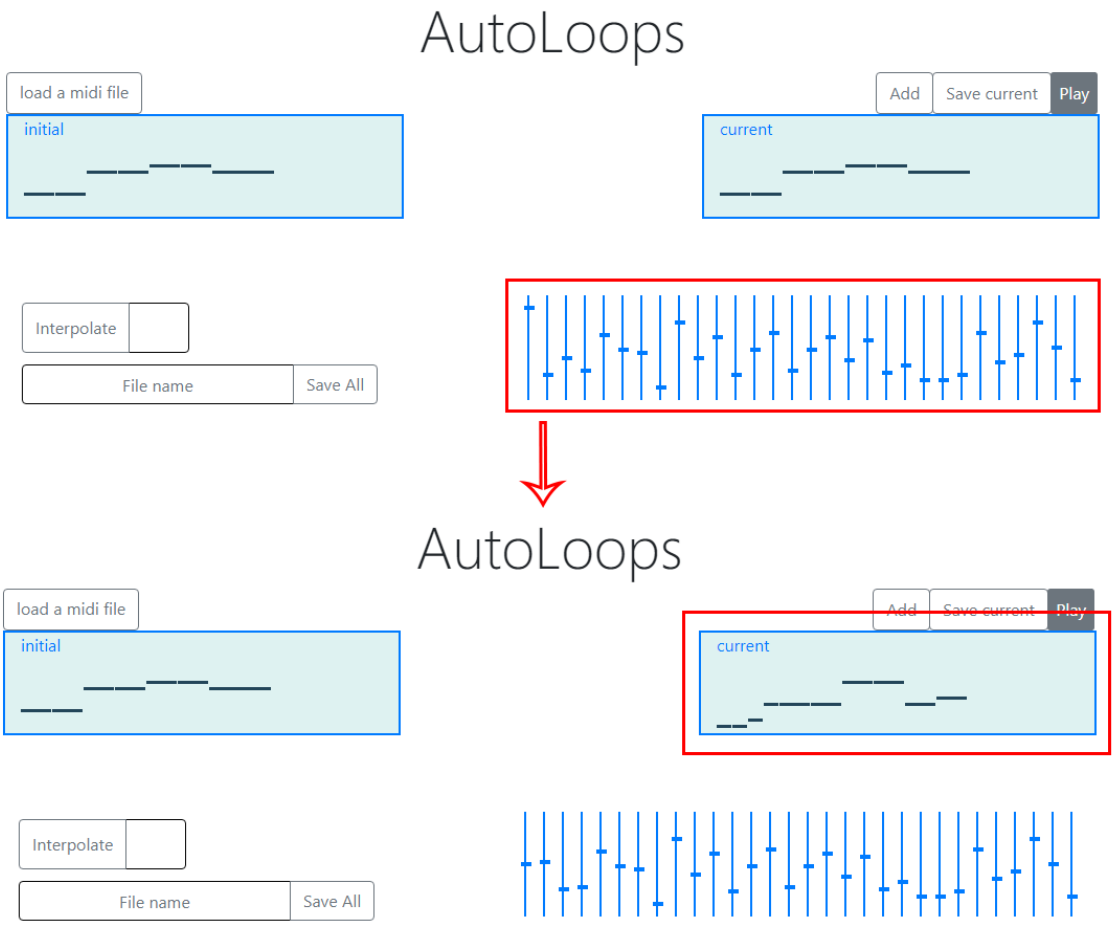

Figura 6.3: Crear una nueva melodía a AutoLooops

Cada vez que el usuario lo considere oportuno puede utilizar el botón *add* para guardar la melodía actual. Estas melodías guardadas podrán ser después exportadas como una secuencia completa mediante el botón *save all*, que también cuenta con un campo para recoger el nombre para el archivo MIDI en el que se exportarán las melodías.

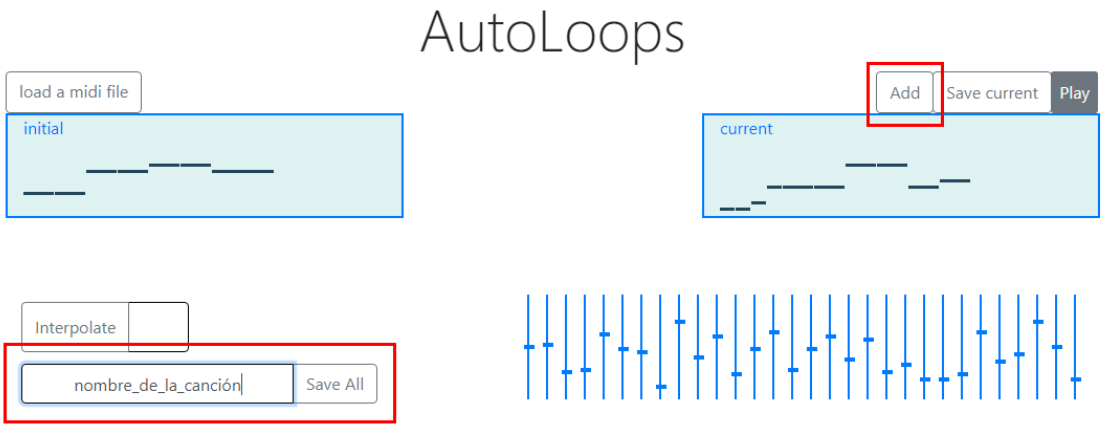

Figura 6.4: Guardar y exportar una secuencia de melodías en AutoLoops

También puede exportarse únicamente la melodía actual mediante el botón *save current*.

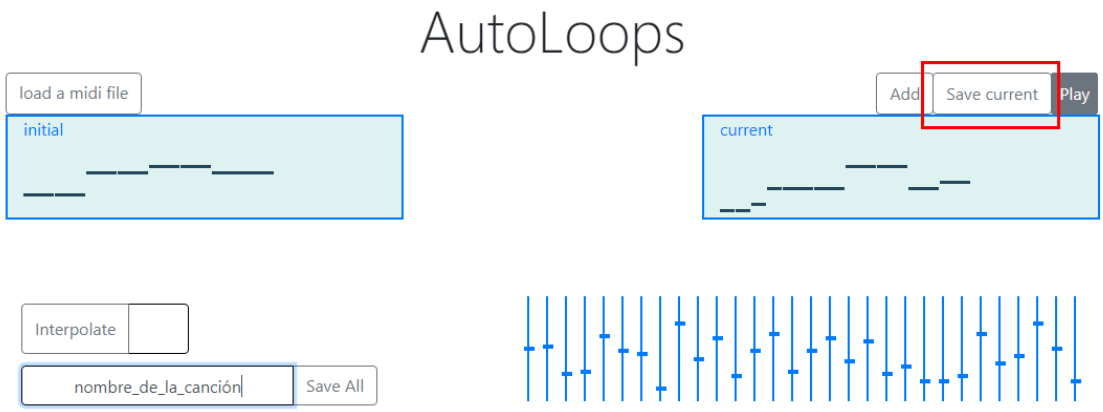

Figura 6.5: Exportar la melodía actual en AutoLoops

Por último se ofrece también la posibilidad de interpolar entre la melodía actual y la melodía inicial. Para ello se hará uso del botón *interpolate*, junto al

que se incluye un campo para recoger el número de pasos que se quieren realizar. El usuario puede introducir un número y se guardará una interpolación entre la melodía de *current* y la de *initial* con ese número de melodías, siendo la primera de la secuencia la melodía actual y la última la melodía inicial. Hecho esto la melodía inicial pasará a ser la actual y puede proseguirse con la utilización del programa.

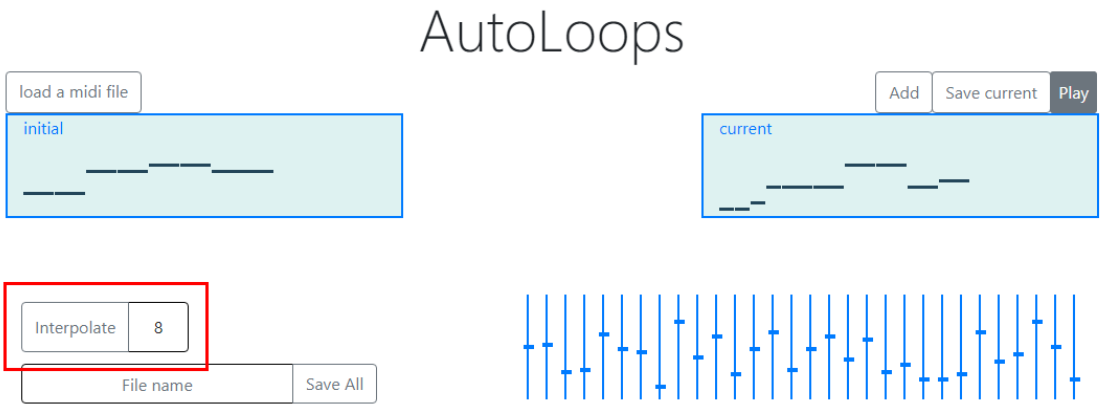

Figura 6.6: Intepolación entre la melodía actual y la inicial en 8 pasos

6.3. Funcionamiento interno

Comprendida la estructura general del programa puede detallarse su funcionamiento. Cuando el usuario carga una melodía MIDI se utiliza el método *blobToNoteSequence* facilitado por la biblioteca *magenta music* para convertir la secuencia cargada en un objeto del tipo *NoteSequence*, que es la representación de melodías con las que se trata. Se utiliza entonces el modelo de MusicVAE para obtener la codificación de la melodía mediante el método *encode*. Con la codificación obtenida se actualizan los controles deslizantes, haciendo que su posición corresponda al valor de la variable latente correspondiente. Hecho esto se decodifica la melodía mediante el método *decode* y se toma dicha decodificación como melodía inicial y melodía actual. Por último se visualizan las melodías mediante el método *PianoRollSVGVisualizer*, también de la biblioteca *magenta music*.

Una vez establecidas las melodías inicial y actuales pueden moverse los controles deslizantes. Cualquier movimiento se captura y se maneja, creando un

tensor con los valores modificados y decodificando dicho tensor para obtener la melodía actual.

Cada vez que se pulsa *add* la melodía actual es añadida a las melodías ya guardadas. Se cuenta con una instancia de *NoteSequence* para almacenar estas melodías. Cada vez que se concatena la que contiene las guardadas con la actual. Esta acción se realiza mediante el método *concatenate* de *magenta music*.

Cuando el usuario pulsa el botón *save all* las melodías guardadas son exportadas a un archivo MIDI del nombre que el usuario elija. Para ello primero se transforma la melodía de *NoteSequence* a MIDI mediante el método *sequenceProtoToMidi* de *magenta music*. Hecho esto el archivo se guarda mediante la función *saveAs* de la biblioteca *FileSaver*. De la misma manera cuando se pulsa *save current* se transforma solo la melodía actual y se exporta.

Para realizar la interpolación se utiliza el método *interpolate* correspondiente al modelo MusicVAE. Se toma como primera melodía de la interpolación la melodía actual y como última la melodía inicial, y se generan una secuencia de tantos pasos como los indicados por el usuario. Hecho esto cada elemento de la secuencia se añade a las secuencias guardadas.

7 Conclusiones y trabajo futuro

En este trabajo se ha estudiado la aplicación de las técnicas de aprendizaje profundo a la generación de música, desde los fundamentos teóricos hasta la aplicación práctica mediante un programa para su uso en el proceso de composición real.

Como punto de partida se ha estudiado el teorema de aproximación universal, que establece el poder de representación que tienen las redes neuronales. Se han repasado los resultados previos necesarios para su demostración y realizado la misma, para terminar con otros resultados que generalizan o complementan este.

A continuación se han tratado cuestiones básicas sobre aprendizaje profundo, incluyendo su contextualización dentro del aprendizaje automático, la definición del modelo más simple, la red profunda prealimentada, y los algoritmos necesarios para la optimización de los mismos.

Más adelante se han tratado las técnicas que ofrece el aprendizaje profundo para el tratamiento de secuencias, esencial para el trabajo con música por la naturaleza secuencial de la misma. Se ha introducido el modelo básico para este fin, la red neuronal recurrente, y los modelos más actuales en este ámbito.

Tras esto se han repasado técnicas para el aprendizaje de características, que permiten el tratamiento simplificado de datos de naturaleza compleja como lo son aquellos derivados de la música. Se ha hecho especial hincapié en los modelos de aprendizaje automático y de entre ellos en el *autoencoder* variacional, que cumple también la función de modelo generativo. El estudio de este modelo se ha realizado desde la perspectiva teórica de la inferencia estadística, habiéndose completado con la obtención de la expresión explícita de su ELBO.

Estudiados los fundamentos teóricos se ha introducido un modelo dirigido a la generación de música, el MusicVAE. Se han discutido su estructura y características básicas, así como el proceso de entrenamiento y las funcionalidades que ofrece.

Por último se ha desarrollado una implementación práctica del modelo en una herramienta llamada AutoLoops. La herramienta, basada en la biblioteca de Magenta que funciona como API para el modelo, se ha planteado como una herramienta para la exploración del espacio latente cuyo objetivo es la generación de melodías. Esta herramienta tiene pretensión de ser utilizada en el contexto real de la composición musical, y de hecho ya está siendo aplicada en la creación de obras musicales. Algunas de estas obras pueden encontrarse en este enlace.

Con esto se considera que los objetivos iniciales marcados para el trabajo han sido cumplidos. Sin embargo, la realización del mismo deja abiertas varias vías de trabajo futuras:

- Inclusión en la herramienta AutoLoops de capacidad para ver y manipular las melodías guardadas.
- Mejora del reproductor de sonido de la herramienta, permitiendo cambiar el tempo de reproducción de las melodías y el timbre con el que se reproducen.
- Adaptación de la herramienta para su utilización en interpretaciones musicales en directo, permitiendo fijar un tempo y reproducir en bucle la melodía actual a ese tempo, pudiendo además modificarla a la vez.
- Adaptación de la herramienta para que pueda ser utilizada como controlador MIDI, para poder así manejar dispositivos externos directamente desde la herramienta.
- Implementación en la herramienta del resto de variantes del modelo MusicVAE, permitiendo así la generación de secuencias de percusión, tríos y melodías de 12 compases.
- Implementación de herramientas similares a AutoLoops basadas en otros modelos actuales para facilitar el acceso de artistas a modelos de aprendizaje profundo que permiten nuevas formas de generar y manipular música.

Bibliografía

Las referencias se listan por orden alfabético. Aquellas referencias con más de un autor están ordenadas de acuerdo con el primer autor.

[AMMIL12] Yaser S Abu-Mostafa, Malik Magdon-Ismail, and Hsuan-Tien Lin. *Learning from data*, volume 4. AMLBook New York, NY, USA:, 2012. [Citado en págs. 9 and 11]

[Ash14] Robert B Ash. *Real Analysis and Probability: Probability and Mathematical Statistics: a Series of Monographs and Textbooks*. Academic press, 2014. [Citado en págs. 2 and 4]

[BCB14] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014. [Citado en pág. 62]

[BCV13] Yoshua Bengio, Aaron Courville, and Pascal Vincent. Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence*, 35(8):1798–1828, 2013. [Citado en pág. 43]

[BGRS99] Kevin Beyer, Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. When is "nearest neighbor" meaningful? In *International conference on database theory*, pages 217–235. Springer, 1999. [Citado en pág. 44]

[Cau47] Augustin Cauchy. Méthode générale pour la résolution des systemes d'équations simultanées. *Comp. Rend. Sci. Paris*, 25(1847):536–538, 1847. [Citado en pág. 20]

[Cur44] Haskell B Curry. The method of steepest descent for non-linear minimization problems. *Quarterly of Applied Mathematics*, 2(3):258–261, 1944. [Citado en pág. 20]

[CVMG+14] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014. [Citado en pág. 38]

[Cyb89] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989. [Citado en págs. 1 and 5]

[CZ18] CJ Carr and Zack Zukowski. Generating albums with samplernn to imitate metal, rock, and punk bands. *arXiv preprint arXiv:1811.06633*, 2018. [Citado en pág. 28]

[DER19] Monica Dinculescu, Jesse Engel, and Adam Roberts, editors. *MidiMe: Personalizing a MusicVAE model with user data*, 2019. [Citado en pág. 70]

[DHS11] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(7), 2011. [Citado en pág. 22]

[Dom12] Pedro Domingos. A few useful things to know about machine learning. *Communications of the ACM*, 55(10):78–87, 2012. [Citado en pág. 43]

[DSD18] Chris Donahue, Ian Simon, and Sander Dieleman. Piano genie. *CoRR*, abs/1810.05246, 2018. [Citado en pág. 70]

[Duo03] Javier Duoandikoetxea. Lecciones sobre las series y transformadas de fourier. *Recuperado de http://www. ugr. es/acanada/docencia/matematicas/analisisdefourier/Duoandikoetxeafourier. pdf*, 2003. [Citado en pág. 7]

[DZMS13] Jun Deng, Zixing Zhang, Erik Marchi, and Björn Schuller. Sparse autoencoder-based feature transfer learning for speech emotion recognition. In *2013 humaine association conference on affective computing and intelligent interaction*, pages 511–516. IEEE, 2013. [Citado en pág. 43]

[EAC+19] Jesse Engel, Kumar Krishna Agrawal, Shuo Chen, Ishaan Gulrajani, Chris Donahue, and Adam Roberts. Gansynth: Adversarial neural audio synthesis. *arXiv preprint arXiv:1902.08710*, 2019. [Citado en pág. 70]

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org. [Citado en págs. 18 and 48]

[GE03] Isabelle Guyon and André Elisseeff. An introduction to variable and feature selection. *Journal of machine learning research*, 3(Mar):1157–1182, 2003. [Citado en pág. 44]

[GMH13] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In *2013 IEEE international conference on acoustics, speech and signal processing*, pages 6645–6649. IEEE, 2013. [Citado en págs. 27 and 33]

[GS00] Felix A Gers and Jürgen Schmidhuber. Recurrent nets that time and count. In *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks. IJCNN 2000. Neural Computing: New Challenges*

and Perspectives for the New Millennium, volume 3, pages 189–194. IEEE, 2000. [Citado en pág. 38]

[GSK+16] Klaus Greff, Rupesh K Srivastava, Jan Koutník, Bas R Steunebrink, and Jürgen Schmidhuber. Lstm: A search space odyssey. *IEEE transactions on neural networks and learning systems*, 28(10):2222–2232, 2016. [Citado en pág. 38]

[Hal76] P.R. Halmos. *Measure Theory*. Graduate Texts in Mathematics. Springer New York, 1976. [Citado en pág. 7]

[HCR+17] Cheng-Zhi Anna Huang, Tim Cooijmans, Adam Roberts, Aaron Courville, and Douglas Eck. Counterpoint by convolution. In *International Society for Music Information Retrieval (ISMIR)*, 2017. [Citado en pág. 28]

[Hor91] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural networks*, 4(2):251–257, 1991. [Citado en pág. 8]

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997. [Citado en pág. 36]

[HSW+89] Kurt Hornik, Maxwell Stinchcombe, Halbert White, et al. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989. [Citado en pág. 8]

[Izeo8] Alan Julian Izenman. Modern multivariate statistical techniques. *Regression, classification and manifold learning*, 10:978–0, 2008. [Citado en pág. 12]

[KB14] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014. [Citado en pág. 22]

[Kul97] Solomon Kullback. *Information theory and statistics*. Courier Corporation, 1997. [Citado en pág. 55]

[KUMH17] Günter Klambauer, Thomas Unterthiner, Andreas Mayr, and Sepp Hochreiter. Self-normalizing neural networks. In *Advances in neural information processing systems*, pages 971–980, 2017. [Citado en pág. 52]

[KW13] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013. [Citado en págs. 53 and 59]

[KW19] Diederik P Kingma and Max Welling. An introduction to variational autoencoders. *arXiv preprint arXiv:1906.02691*, 2019. [Citado en págs. 55 and 58]

[LFY+19] Qian Liu, Li Fang, Guoliang Yu, Depeng Wang, Chuan-Le Xiao, and Kai Wang. Detection of dna base modifications by deep recurrent neural network on oxford nanopore sequencing data. *Nature communications*, 10(1):1–11, 2019. [Citado en pág. 27]

[LPW+17] Zhou Lu, Hongming Pu, Feicheng Wang, Zhiqiang Hu, and Liwei Wang. The expressive power of neural networks: A view from the width. In *Advances in neural information processing systems*, pages 6231–6239, 2017. [Citado en pág. 8]

[LV07] John A Lee and Michel Verleysen. *Nonlinear dimensionality reduction*. Springer Science & Business Media, 2007. [Citado en pág. 45]

[LZZ+18] Sidi Lu, Yaoming Zhu, Weinan Zhang, Jun Wang, and Yong Yu. Neural text generation: Past, present and beyond. *arXiv preprint arXiv:1803.07133*, 2018. [Citado en pág. 27]

[M+97] Tom M Mitchell et al. *Machine learning. WCB*. McGraw-Hill Boston, MA:, 1997. [Citado en pág. 9]

[PGCB13] Razvan Pascanu, Caglar Gulcehre, Kyunghyun Cho, and Yoshua Bengio. How to construct deep recurrent neural networks. *arXiv preprint arXiv:1312.6026*, 2013. [Citado en pág. 33]

[REOE18] Adam Roberts, Jesse H Engel, Sageev Oore, and Douglas Eck. Learning latent representations of music to generate interactive musical palettes. In *IUI Workshops*, 2018. [Citado en pág. 43]

[RER+18] Adam Roberts, Jesse Engel, Colin Raffel, Curtis Hawthorne, and Douglas Eck. A hierarchical latent vector model for learning long-term structure in music. *arXiv preprint arXiv:1803.05428*, 2018. [Citado en pág. 61]

[RHS18] Adam Roberts, Curtis Hawthorne, and Ian Simon. Magenta.js: A javascript api for augmenting creativity with deep learning. In *Joint Workshop on Machine Learning for Music (ICML)*, 2018. [Citado en pág. 69]

[RMW14] Danilo Jimenez Rezende, Shakir Mohamed, and Daan Wierstra. Stochastic backpropagation and approximate inference in deep generative models. *arXiv preprint arXiv:1401.4082*, 2014. [Citado en pág. 53]

[Rud87] W. Rudin. *Real and Complex Analysis*. Mathematics series. McGraw-Hill, 1987. [Citado en pág. 5]

[SH09] Ruslan Salakhutdinov and Geoffrey Hinton. Semantic hashing. *International Journal of Approximate Reasoning*, 50(7):969–978, 2009. [Citado en pág. 43]

[SKD+17] Shashi Pal Singh, Ajai Kumar, Hemant Darbari, Lenali Singh, Anshika Rastogi, and Shikha Jain. Machine translation using deep learning: An overview. In *2017 International Conference on Computer, Communications and Electronics (Comptelix)*, pages 162–167. IEEE, 2017. [Citado en pág. 27]

[SO17] Ian Simon and Sageev Oore. Performance rnn: Generating music with expressive timing and dynamics. https://magenta.tensorflow.org/performance-rnn, 2017. [Citado en pág. 70]

[Sut13] Ilya Sutskever. *Training recurrent neural networks*. University of Toronto Toronto, Canada, 2013. [Citado en pág. 21]

[TH12] Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26–31, 2012. [Citado en pág. 22]

[TSCH17] Lucas Theis, Wenzhe Shi, Andrew Cunningham, and Ferenc Huszár. Lossy image compression with compressive autoencoders. *arXiv preprint arXiv:1703.00395*, 2017. [Citado en pág. 43]

[VLH17] Ana Valdivia, M Victoria Luzón, and Francisco Herrera. Sentiment analysis in tripadvisor. *IEEE Intelligent Systems*, 32(4):72–77, 2017. [Citado en pág. 27]

[XXC12] Junyuan Xie, Linli Xu, and Enhong Chen. Image denoising and inpainting with deep neural networks. In *Advances in neural information processing systems*, pages 341–349, 2012. [Citado en pág. 43]

[YJL05] Jieping Ye, Ravi Janardan, and Qi Li. Two-dimensional linear discriminant analysis. In *Advances in neural information processing systems*, pages 1569–1576, 2005. [Citado en pág. 48]