Universidad del País Vasco
Euskal Herriko Unibertsitatea

ZTF-FCT
Zientzia eta Teknologia Fakultatea
Facultad de Ciencia y Tecnología

Gradu Amaierako Lana / Trabajo Fin de Grado
Ingenieritza Elektronikoko Gradua / Grado en Ingeniería Electrónica

Introducción al procesamiento del habla mediante técnicas de deep learning

Egilea/Autor:
Asier López Zorrilla

Zuzendaria/Directora:
María Inés Torres Barañano
Zuzendariordea/Subdirectora:
Raquel Justo Blanco

Leioa, 2016ko Ekainaren 27a /Leioa, 27 de Junio de 2016

Acknowledgements

Firstly, I would like to express my sincere gratitude to my supervisors Manés and Raquel. Their guidance helped me in all the time of my project and in the writing of this work.

Of course, I am really grateful to Intelligent Voice, for giving me the opportunity of developing my project with them in London in such a nice working atmosphere. Especially, I would like to thank Gérard for making this possible and taking me in his house for a week.

Last but not the least, I would like to thank all my classmates, essentially for having made me feel happy during this last four years. More concretely, I hope that next years will be as good as these have been in the magnificent room 0.11.

Índice general

Introducción y objetivos

Los primeros artículos sobre la tarea del reconocimiento automático de voz fueron publicados a principios de la década de 1950. Los primeros dispositivos estaban basados en circuitos analógicos, y eran capaces de reconocer palabras aisladas dentro de un vocabulario de unas diez palabras. A partir de la década de 1960, comenzó el uso de computadoras para desarrollar sistemas de reconocimiento de voz. En esa misma década se desarrollaron los primeros reconocedores de habla continua. En la década de los 70 se comenzaron a aplicar metodologías de reconocimiento de patrones para resolver el problema del reconocimiento de voz. En las décadas de los 80 y 90, los mejores resultados en el ámbito de reconocimiento de voz fueron logrados mediante sistemas basados en modelos estadísticos, en especial en modelos ocultos de Markov. En estos últimos años, el uso de redes neuronales artificiales se ha extendido en todos los ámbitos del reconocimiento de formas, debido en gran medida al desarrollo de máquinas con gran capacidad capacidad de cómputo, y al acceso a un a gran cantidad de datos con los que entrenar estas redes.

A lo largo de este trabajo se estudiarán de forma teórica la arquitectura de los reconocedores de voz basados en modelos generativos. En concreto se analizarán dos sistemas distintos: los sistemas basados en modelos ocultos de Markov y mezclas de gaussianas, y los modelos híbridos entre modelos ocultos de Markov y redes neuronales. Para ello se comenzará realizando una introducción al problema del reconocimiento de voz. Después se analizarán de forma general modelos de mezclas de gaussianas, los modelos ocultos de Markov y las redes neuronales.

Finalmente se presentará la herramienta Kaldi, con la cuál se realizarán diversos experimentos para comparar y analizar las características de los distintos sistemas de reconocimiento de voz. En particular, nos centraremos en estudiar el comportamiento de las redes neuronales.

Capítulo 1

Base teórica de los reconocedores de voz

Si miramos a un reconocedor de voz ideal desde el punto de vista más abstracto posible, veremos una máquina capaz de transformar una señal acústica de voz en su correspondiente transcripción ortográfica. Aún así, es prácticamente imposible lograr que una máquina transcriba toda señal acústica a la secuencia de palabras que el hablante pretendía trasmitir haciendo uso de sus cuerdas vocales y su tracto vocal. De hecho, ni siquiera los humanos somos capaces de reconocer siempre las palabras que escuchamos, y eso que llevamos una cantidad considerable de tiempo dedicándonos a la tarea de la comunicación oral (distintos estudios hablan de entre 50.000 años y 6 millones de años [66]).

Consecuentemente, se suele ver la tarea de reconocimiento de voz como un problema de optimización: dada una señal acústica $\boldsymbol{x}$, encontrar la secuencia de palabras más probable $\widehat{\boldsymbol{g}}$. Es decir,

$$\widehat{\boldsymbol{g}} = \underset{\boldsymbol{g}\in\mathscr{G}}{\arg\max}\ P\left(\boldsymbol{g} \mid \boldsymbol{x}\right), \tag{1.1}$$

donde $\mathscr{G}$ es el conjunto de todas las posibles secuencias de palabras. En general, existen dos formas de realizar esta estimación.

La primera es intentar calcular directamente la probabilidad $P\left(\boldsymbol{g} \mid \boldsymbol{x}\right)$. En este caso hablamos de modelos discriminativos. La otra opción es dividir el problema utilizando el teorema de Bayes:

$$\widehat{\boldsymbol{g}} = \underset{\boldsymbol{g}\in\mathscr{G}}{\arg\max}\ P\left(\boldsymbol{g} \mid \boldsymbol{x}\right) = \underset{\boldsymbol{g}\in\mathscr{G}}{\arg\max}\ \frac{P\left(\boldsymbol{x} \mid \boldsymbol{g}\right)P(\boldsymbol{g})}{P(\boldsymbol{x})} = \underset{\boldsymbol{g}\in\mathscr{G}}{\arg\max}\ P\left(\boldsymbol{x} \mid \boldsymbol{g}\right)P(\boldsymbol{g}) \tag{1.2}$$

En la ecuación 1.2 se ha ignorado el denominador $P(\boldsymbol{x})$, ya que solo afecta al valor numérico de la probabilidad, no a la secuencia óptima de palabras. De acuerdo a la ecuación 1.2, debemos calcular las probabilidades $P\left(\boldsymbol{x} \mid \boldsymbol{g}\right)$ y $P(\boldsymbol{g})$ para realizar la estimación $\widehat{\boldsymbol{g}}$. Los sistemas que emplean este método son conocidos como sistemas generativos. Todos los reconocedores desarrollados en este trabajo son de este tipo, luego de aquí en adelante nos centraremos en su funcionamiento. El cálculo de las probabilidades $P\left(\boldsymbol{x} \mid \boldsymbol{g}\right)$ es tarea de la modelización acústica, mientras que para el cómputo de $P(\boldsymbol{g})$ se emplean modelos de lenguaje.

En la Figura 1.1 se muestra un esquema general del funcionamiento de los reconocedores de voz generativos. En la Secciones 1.1, 1.2 y 1.3 se describirán los bloques *Parametrización de la señal acústica, Modelos acústicos* y *Modelos de lenguaje* que se pueden observar en la Figura 1.1. El problema de la *Decodificación* se analizará en distintas Secciones de los capítulos 2, 4 y 5.

1.1. Parametrización de la señal acústica

Hasta ahora hemos hablado de $\boldsymbol{x}$ como la señal acústica que estamos procesando para conseguir su transcripción ortográfica. La señal acústica representa las variaciones en la presión del aire causadas (en este caso) por el hablante a lo largo del tiempo. Es necesario puntualizar que en la mayoría de reconocedores de voz (incluyendo los desarrollados en este trabajo) no se trabaja directamente con esta señal. Lo más común es parametrizarla. Para ello se separa en pequeñas ventanas de tiempo (las cuales se pueden

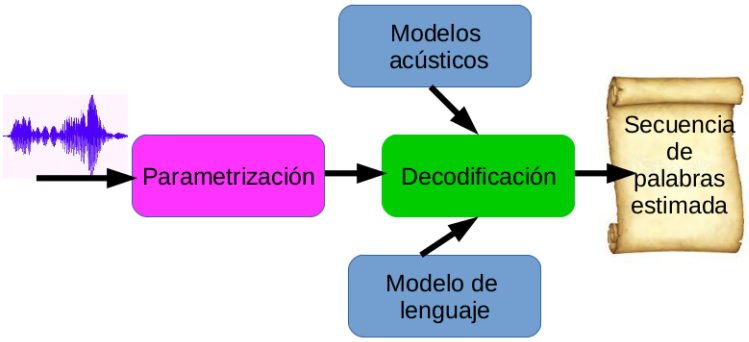


Figura 1.1: Esquema general de los reconocedores de voz generativos.

solapar), y cada una de las ventanas se procesa para ser representada con un vector. Es decir, $\boldsymbol{x}$ no es la señal acústica en sí, sino la secuencia de los vectores de parámetros o vectores acústicos que la representan.

Al parametrizar una señal acústica, hay que tener en cuenta el tamaño de las ventanas y la transformación que se va aplicar a cada una de ellas. Para elegir el tamaño de las ventanas, es importante entender primero qué transformación se va a utilizar y por qué.

1.1.1. Transformación para conseguir el vector de parámetros asociado a un fragmento de la señal acústica

Para distinguir entre todos los sonidos que somos capaces de percibir, los humanos detectamos la potencia de las ondas de presión a distintas frecuencias, aproximadamente dentro del rango de 20 Hz a 20 kHz. Pero no percibimos el sonido de forma lineal respecto a la frecuencia: si se analiza la diferencia de frecuencia entre tonos juzgados como intervalos equiespaciados, veremos que esta no es constante. Somos capaces de diferenciar variaciones pequeñas en la frecuencia de señales a frecuencias bajas, mientras que a frecuencias más altas tiene que cambiar más la frecuencia para que percibamos un cambio en el tono. Por ejemplo, ésta es la causa de que la distancia entre los trastes de una guitarra no sea todo el rato igual a lo del mástil. En 1937 Stevens, Volkmann y Newman propusieron la escala mel (el nombre *mel* viene de la palabra *melodía*), en la cual los intervalos de altura musical son constantes [60]. La relación entre hercios y mels (unidades de la escala mel) es logarítmica:

$$m(\text{ mels}) = 2595 \log_{10}\left(1 + \frac{f(\text{ Hz})}{700}\right) \tag{1.3}$$

Así, parece natural plantear que los vectores que van a representar cada ventana de audio contengan información sobre la energía de la señal a distintas frecuencias (frecuencias colocadas a distancia constante en la escala mel). Dichos vectores están compuestos por los coeficientes cepstrales en las frecuencias de mel (MFCC, por sus siglas en inglés). En reconocimiento de voz se suelen utilizar 13 coeficientes, que contienen información sobre las frecuencias en las que los humanos producimos sonidos. El algoritmo para calcular los MFCC consta de las siguientes etapas (opcionalmente se pueden añadir pasos intermedios, pero con estas etapas se entiende la filosofía del algoritmo) [35], [10], [64]:

1. Fragmentar el audio de acuerdo al tamaño y desplazamiento de ventana.

2. Aplicar la función de ventana, la más común es la ventana *hamming*. Esto facilita la transformada discreta de coseno (DFT) en la etapa 6.

3. Computar la transformada rápida de Fourier (FFT), para conseguir el espectro de nuestro fragmento de señal.

4. Calcular la energía en distintos bancos de frecuencias de igual tamaño, forma y distancia en la escala mel.

5. Calcular el logaritmo de las energías calculadas en la etapa 4. La razón de este paso es que no solo la percepción de las tonalidades musicales es logarítmica, sino también la percepción de la intensidad del sonido [1].

6. Finalmente aplicar la DFT a los logaritmos de las energías. El resultado son los MFCC.

Las parametrizaciones MFCC son muy importantes en el ámbito del reconocimiento de voz. Durante este trabajo utilizaremos parametrizaciones MFCC en distintas ocasiones, pero también otras parametrizaciones (las cuales partirán de los vectores de parámetros MFCC). Estas serán introducidas en la Sección 4.3.

1.1.2. Tamaño de las ventanas para fragmentar la señal acústica

Las ventanas han de ser suficientemente grandes como para contener información relevante sobre el sonido que se ha producido en ese intervalo de tiempo. Como se ha explicado en la sección 1.1.1, la información que los humanos extraemos de una señal acústica está en el espectro de dicha señal. Si el tamaño de ventana fuese muy pequeño, el resultado de la FFT de la señal que ha quedado dentro de la ventana no será demasiado valioso, porque la FFT se realizará con muy pocas muestras.

Por otra parte, nos interesa que el tamaño de ventana suficiente pequeño como para no mezclar demasiada información de distintos sonidos. Si conseguimos que cada ventana contenga información de solo un sonido, el espectrograma nos dará información relevante sobre el fonema que estamos analizando.

Por tanto en los reconocedores de voz aquí desarrollados se ha optado por ventanas 25 ms, y desplazamiento de 10 ms, así que las ventanas se solapan en este caso. Estos dos valores son muy comunes a la hora de realizar el cálculo de los MFCC.

1.2. Modelos acústicos

Los modelos acústicos son representaciones estadísticas de los vectores de parámetros explicados en la sección 1.1.

La modelización acústica se encarga de computar la probabilidad $P(\boldsymbol{x} \mid \boldsymbol{g})$ de la ecuación 1.2. Para ello, tendremos que contar con modelos acústicos capaces de modelar los sonidos correspondientes las unidades fonéticas mínimas del habla (quizás contextuales), y con un léxico que describa la pronunciación de cada palabra que aspiramos a reconocer.

1.2.1. Léxico

Podemos pensar que el léxico es un diccionario. Si buscamos una palabra en el léxico, éste nos dará su pronunciación, es decir, la secuencia de fonemas correspondiente a la palabra que hayamos buscado. El léxico ha de tener en cuenta que puede existir más de una forma de pronunciar una palabra. En ese caso, el léxico nos debe dar todas las pronunciaciones posibles y la probabilidad de cada una de ellas.

Los reconocedores de voz desarrollados en este trabajo tienen como objetivo reconocer secuencias de palabras en castellano. El castellano es un idioma regular en lo que a la transcripción de palabras a fonemas se refiere, sobre todo si lo comparamos con idiomas como el inglés o el francés. Con un conjunto de reglas de pronunciación podemos deducir la secuencia de fonemas que corresponden a cada palabra, en vez de tener que utilizar diccionarios de pronunciación. Con estas reglas conseguiremos pronunciaciones únicas. No obstante, es necesario mencionar que distintas palabras pueden tener una misma pronunciación, por ejemplo, *hola* y *ola*.

Nuestro objetivo es calcular $P(\boldsymbol{x} \mid \boldsymbol{g})$. Gracias al léxico podemos dividir una secuencia de palabras en la secuencia de fonemas correspondiente, simplemente uniendo las secuencias de fonemas cada palabra. Por tanto, podemos desarrollar la probabilidad anterior [15]:

$$P(\boldsymbol{x} \mid \boldsymbol{g}) = \sum_{\boldsymbol{l} \in \mathscr{L}} P(\boldsymbol{x} \mid \boldsymbol{l}) P(\boldsymbol{l} \mid \boldsymbol{g}), \tag{1.4}$$

donde $\mathscr{L}$ es el conjunto de todas las secuencias de fonemas o unidades léxicas. La ecuación 1.4 describe el caso general de las pronunciaciones; es decir, tiene en cuenta las distintas pronunciaciones posibles de una palabra, y la probabilidad de cada una de ellas. En nuestro caso, cada palabra tiene una y solo una pronunciación posible. Así, solo habrá una pronunciación en el conjunto $\mathscr{L}$ que presente una probabilidad $P(\boldsymbol{l} \mid \boldsymbol{g})$ no nula (y además igual a uno). Si denotamos esa pronunciación como $\boldsymbol{l_g}$:

$$P(\boldsymbol{x} \mid \boldsymbol{g}) = \sum_{\boldsymbol{l} \in \mathscr{L}} P(\boldsymbol{x} \mid \boldsymbol{l}) P(\boldsymbol{l} \mid \boldsymbol{g}) = P(\boldsymbol{x} \mid \boldsymbol{l_g}) \tag{1.5}$$

En este punto podríamos discutir cómo calcular $P(\boldsymbol{x} \mid \boldsymbol{l_g})$, es decir, cómo modelizar secuencias de unidades fonéticas. Pero como he mencionado al inicio de esta sección (1.2), lo más común es dar un paso más antes de realizar la modelización acústica: descomponer los fonemas en fonemas contextuales (trifonemas normalmente).

Durante este trabajo se analizarán tanto los sistemas basados en fonemas como en trifonemas.

1.2.2. Trifonemas: fonemas dependientes del contexto

Los humanos pronunciamos de forma distinta los fonemas en función de cual ha sido el anterior fonema, y de cual va a ser el siguiente. El uso de trifonemas permite tener en cuenta este efecto [25] y mejorar así el rendimiento del reconocedor de voz [4].

Un trifonema es una representación contextual de un fonema mediante tres fonemas. Habitualmente se utiliza el central para indicar el fonema que vamos a representar, el izquierdo para indicar el fonema anterior, y el derecho para indicar el posterior. Por ejemplo, la palabra *casa* se descompone de la siguiente forma en fonemas y trifonemas (con $\boldsymbol{c}$ nos referimos a la secuencia de fonemas contextuales o trifonemas, y con ε a la ausencia de fonema anterior o posterior):

$\boldsymbol{g}$	$\boldsymbol{l}$	$\boldsymbol{c}$
casa	/k/ /a/ /s/ /a/	/ε_k_a/ /k_a_s/ /a_s_a/ /s_a_ε/

Si utilizamos un reconocedor basado en trifonemas, podemos reescribir los resultados de las Ecuaciones 1.4 y 1.5. Para ello debemos tener en cuenta que dada una secuencia de fonemas, existe una única secuencia de trifonemas equivalente ($\boldsymbol{c_l}$) [15], como se puede observar en el ejemplo anterior. También es similar al caso de nuestro léxico, en el que dada una secuencia de palabras se consigue una única secuencia de fonemas.

$$P(\boldsymbol{a} \mid \boldsymbol{l}) = P(\boldsymbol{x} \mid \boldsymbol{c_l}) \tag{1.6}$$

En la ecuación 1.6 se han utilizado $\boldsymbol{l}$ y $\boldsymbol{c_l}$ para hacer referencia al caso más general (el correspondiente a la ecuación 1.4). Si se utiliza un léxico que solo contiene una pronunciación por palabra, se puede sustituir $\boldsymbol{l}$ por $\boldsymbol{l_g}$, y $\boldsymbol{c_l}$ por $\boldsymbol{c_{l_g}}$.

Tanto si vamos a utilizar un sistema basado en fonemas o trifonemas, ahora nuestra tarea es crear modelos acústicos. Estos modelos nos permitirán relacionar los vectores de parámetros con las unidades fonéticas mínimas que hayamos escogido. En el caso de los modelos generativos (este caso), relacionar indica calcular la probabilidad $P(\boldsymbol{x} \mid \boldsymbol{l})$ (sistema basado en fonemas) o $P(\boldsymbol{x} \mid \boldsymbol{c_l})$ (sistema basado en trifonemas).

A día de hoy, lo más común es utilizar modelos ocultos de Markov (HMM, por sus siglas en inglés) basados en mezclas de gaussianas (GMM, por sus siglas en inglés), o modelos híbridos entre HMM y redes

neuronales profundas (DNN, por sus siglas en inglés). Ambos modelos serán analizados durante este trabajo.

En los Capítulos 2.2 y 3 se estudiará el funcionamiento general de los modelos ocultos de Markov y de redes neuronales, respectivamente.

En la Sección 4 se analizará como aplicar los HMM y las DNN para crear modelos acústicos. Se describirán los modelos GMM-HMM (sección 4.1) y los híbridos DNN-HMM (sección 4.2).

1.3. Modelos de lenguaje

Los modelos de lenguaje nos permiten calcular la probabilidad a priori $P(\boldsymbol{g})$ de la ecuación 1.2, es decir, un modelo de lenguaje asigna una probabilidad a una secuencia de palabras.

Existen distintos tipos de modelos de lenguaje. Los más comunes son los n -gramas, que también son unos de los más sencillos. Se basan en la suposición de que la probabilidad de una palabra solo depende de ella misma y de las $n-1$ palabras anteriores. En el caso particular de $n=1$ unigrama y la probabilidad de cada palabra solo depende de la propia palabra. En general, la probabilidad de la secuencia $\boldsymbol{g} = g_0, g_1, ..., g_m$ será aproximadamente la siguiente:

$$P(\boldsymbol{g}) = P(g_0, g_1, ..., g_m) = \prod_{i=1}^{m} P(g_i \mid g_1, ..., g_{i-1}) \approx \prod_{i=1}^{m} P(g_i \mid g_{i-(n-1)}, ..., g_{i-1}) \tag{1.7}$$

Cada probabilidad condicional $P(g_i \mid g_{i-(n-1)}, ..., g_{i-1})$ puede ser estimada de forma sencilla a partir de un corpus de entrenamiento, simplemente contando (y normalizando) cuantas veces a aparecido la palabra g_i después de la secuencia $g_{i-(n-1)}, ..., g_{i-1}$. Si $C(\boldsymbol{g})$ representa la cantidad de ocurrencias de una secuencia $\boldsymbol{g}$ en el corpus:

$$P(g_i \mid g_{i-(n-1)}, ..., g_{i-1}) \approx \frac{C(g_{i-(n-1)}, ..., g_{i-1}, g_i)}{C(g_{i-(n-1)}, ..., g_{i-1})} \tag{1.8}$$

Esta forma de computar las probabilidades tiene un problema significativo, problema que aumenta cuanto mayor es el orden del modelo de lenguaje n y/o menor es el tamaño del corpus de entrenamiento. Si hiciesemos un modelo de lenguaje utilizando solo las ecuaciones 1.7 y 1.8, la probabilidad de secuencias de palabras no observadas durante el entrenamiento sería nula (aunque cada palabra sí haya sido observada individualmente). Normalmente esta aproximación es insuficiente. Por ejemplo, puede que hayamos observado las secuencias *mi casa azul* y *mi casa verde*, pero no la secuencia *mi casa roja*. Aún así, parece demasiado restrictivo decir que la probabilidad de *mi casa roja* es nula. La solución más común es realizar un *smoothing* [16]. Así, secuencias que no hayan sido observadas en el corpus podrán tener probabilidades no nulas en el modelo de lenguaje.

En la práctica (y en este trabajo) los n -gramas más utilizados son los que tienen $n=3$, los trigramas.

Capítulo 2

Modelos ocultos de Markov y modelos de mezclas de gaussianas

En este capítulo se analizarán de forma teórica los modelos ocultos de Markov (HMM). Como se detallará en la Sección 2.2, en la tarea de reconocimiento de voz los HMM dependen de los modelos de mezclas de gaussianas (GMM), por lo que se describirán éstas brevemente de forma previa al análisis de los HMM.

2.1. Modelos de mezclas de gaussianas

La distribución gaussiana es una distribución de probabilidad ampliamente utilizada en aplicaciones estadísticas. En el caso del reconocimiento de voz se suelen emplear para modelizar los vectores de parámetros conseguidos después de la parametrización de una señal acústica.

Una distribución gaussiana para una sola variable aleatoria x está definida mediante la siguiente función:

$$P(x) = \frac{1}{\sqrt{2\pi}\,\sigma}\exp\left[-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right] = \mathcal{N}\left(x\,;\mu,\sigma^2\right), \tag{2.1}$$

donde μ es la media y σ^2 la varianza.

Las distribuciones gaussianas se pueden extender a un espacio $\mathbb{R}^D$ mediante la siguiente función multivariable [69]:

$$P(\boldsymbol{x} = x_1, x_2, ..., x_D) = \frac{1}{(2\pi)^{D/2}|\boldsymbol{\Sigma}|^{1/2}}\exp\left[-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^T\boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu})\right] = \mathcal{N}\left(\boldsymbol{x}\,;\boldsymbol{\mu},\boldsymbol{\Sigma}\right) \tag{2.2}$$

donde $\boldsymbol{\mu}$ es el vector de medias de dimensión D y $\boldsymbol{\Sigma}$ la matriz de covarianzas de dimensión $D \times D$.

En ocasiones, por ejemplo en reconocimiento de voz, puede que la variable o vector aleatorio sea demasiado complejo como para que una sola distribución gaussiana lo modelice correctamente. En esos casos se puede sustituir la distribución gaussiana por una mezcla de varias distribuciones gaussianas. Si M es el número de gaussianas en la mezcla y c_m el peso de cada gaussiana, la GMM será la siguiente:

$$P(\boldsymbol{x}) = \sum_{m=1}^{M} c_m \mathcal{N}\left(\boldsymbol{x}\,;\boldsymbol{\mu}_m,\boldsymbol{\Sigma}_m\right), \tag{2.3}$$

donde $c_m > 0$ y $\sum_{m=1}^{M} c_m = 1$. En la Figura 2.1 se muestra un ejemplo de una GMM.

En el contexto de distribuciones formadas por mezclas de gaussianas, se define la función indicador $\mathbb{I}^l(\boldsymbol{x})$. Ésta indica si el vector $\boldsymbol{x}$ esta asociado a la gaussiana con índice l de la GMM, es decir,

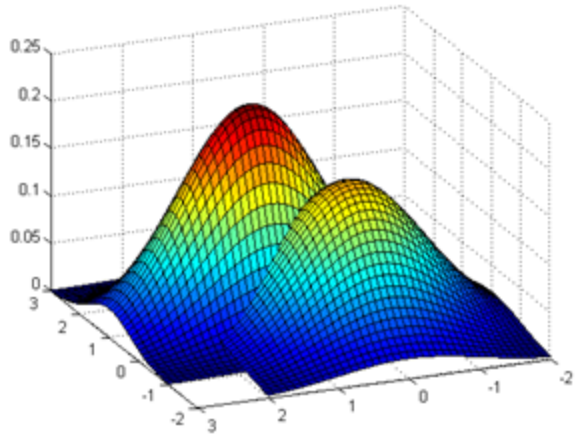


Figura 2.1: Representación gráfica de un ejemplo de una mezcla de dos gaussianas en un espacio bidimensional. Fuente: [26].

$$\mathbb{I}^l(\boldsymbol{x}) = \begin{cases} 1, & \text{si } l = \underset{1\leq m\leq M}{\arg\max} \left[\mathcal{N}\left(\boldsymbol{x}\,;\boldsymbol{\mu}_m,\boldsymbol{\Sigma}_m\right)\right] \\ 0, & \text{en cualquier otro caso.} \end{cases} \tag{2.4}$$

2.2. Modelos Ocultos de Markov

Los modelos ocultos de Markov son modelos estadísticos desarrollados por Braum, Eagon, Petrie, Soules y Weiss a finales de la década de 1960 [42]. Éstos son una extensión de las cadenas o modelos de Markov introducidos por Andrey Markov en 1906 [29].

Ambos sirven para modelizar sistemas estocásticos que carecen de memoria, es decir, sistemas en los que la probabilidad del siguiente estado solo depende del estado actual (propiedad de Markov) [52]. La diferencia está en que en las cadenas de Markov los estados son visibles, mientras que en los HMM estos estados están ocultos. Esto quiere decir que, a diferencia de las cadenas de Markov, en los HMM cada salida del sistema no está directamente ligada a un estado. Cada estado del HMM tiene asociado una o un conjunto de probabilidades o densidades de probabilidades para cada posible salida del sistema.

Como ejemplo, supongamos que queremos modelizar el tiempo en Bilbao. Asumiendo que el tiempo no cambia durante cada día y que los días solo pueden ser soleados o lluviosos, podríamos elaborar una estadística y estimar las probabilidades de cada situación meteorológica teniendo en cuenta el tiempo que hizo el día anterior (probabilidades de transición). El resultado podría ser la cadena de Markov que se muestra en la Figura 2.2.

Para ejemplificar un HMM, supongamos ahora que viajamos a Londres, y por tanto no podemos conocer directamente el tiempo en Bilbao. En cambio, podemos hablar con un amigo de Bilbao, que nos cuenta lo que éste ha hecho durante el día: salir de fiesta o estudiar modelos ocultos de Markov. Nosotros conocemos a nuestro amigo, lo que en este contexto significa que conocemos las probabilidades de que salga de fiesta o se quede estudiando en función del tiempo (probabilidades de emisión). Con esta información es posible modelizar el tiempo en Bilbao en base a las actividades diarias de nuestro amigo, mediante el HMM representado en la Figura 2.3.

Hay que decir que es posible tener un número infinito de salidas observables. Por ejemplo, nuestro amigo nos podría decir la temperatura media durante el día en vez de contarnos qué ha hecho. En ese caso, las probabilidades de emisión se sustituirían por densidades de probabilidad. Aún más, es posible que nuestro

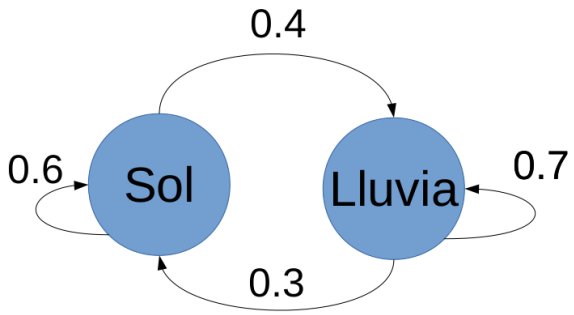


Figura 2.2: Representación gráfica del ejemplo de cadena de Markov. No se han indicado las probabilidades iniciales para no complicar el esquema.

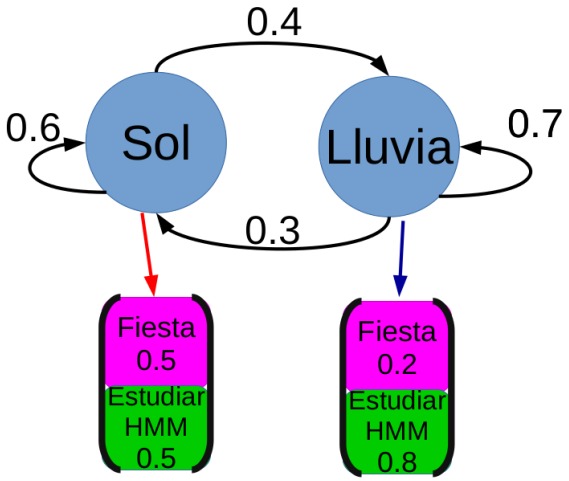


Figura 2.3: Representación gráfica del ejemplo de HMM. No se han indicado las probabilidades iniciales.

amigo no solo nos dé un dato al día, sino un vector de datos (por ejemplo: la temperatura media, la humedad y el precio de la gasolina). Así, en cada estado del HMM tendríamos asociada una probabilidad de emisión de un vector componentes continuos.

En este punto ya se puede intuir que en el caso del reconocimiento de voz las observaciones serán los vectores de parámetros descritos en la Sección 1.1 y que los estados del HMM representarán de alguna forma los fonemas o trifonemas que estamos intentando reconocer. Además, es latente la importancia de seleccionar vectores que representen información útil sobre la señal acústica. Siguiendo con la analogía de nuestro amigo de Bilbao, claramente la temperatura y la humedad nos van a dar mucha más información sobre el estado meteorológico que el precio de la gasolina. En la Sección 4 especificaremos la topología de los HMM utilizados en los reconocedores de voz y cómo se modelizan las probabilidades de emisión de cada componente de los vectores en cada estado.

2.2.1. Elementos de un HMM

En las Sección 2.2.2 analizaremos los algoritmos para utilizar los HMM. Para entenderlos es necesario introducir la notación formal para definir un HMM y las observaciones que se van a intentar modelizar.

Las posibles salidas u observaciones pueden ser tanto discretas como continuas, y tanto unidimensionales como n-dimensionales. Lo más común es que las observaciones sean símbolos discretos o vectores de variables continuas. Denotaremos una secuencia de observaciones como $\boldsymbol{o} = \{\boldsymbol{o}_t\} = \{\boldsymbol{o_1}, \boldsymbol{o_2}, ..., \boldsymbol{o_T}\}$, donde T es el tamaño de la secuencia. Si las observaciones son símbolos discretos, cada observación $\boldsymbol{o}_t = o_t$ será una de las posibles K salidas del HMM $\boldsymbol{v} = \{v^k\}$. Si por el contrario las observaciones son vectores de variables continuas, cada componente de ese vector $\boldsymbol{o}_t$ podrá, en general, tomar cualquier valor real.

Se puede definir la estructura de un HMM mediante las distribuciones de probabilidades $\boldsymbol{a}, \boldsymbol{b}$ y $\boldsymbol{\pi}$. $\boldsymbol{a}$ es la matriz de probabilidades de transición, $\boldsymbol{b}$ representa las probabilidades de emisión en un estado, y $\boldsymbol{\pi}$ contiene las probabilidades iniciales de cada estado.

- $\boldsymbol{a} = \{a_{ij}\}$. a_{ij} representa la probabilidad de transición del estado q_i al estado q_j, siendo $1 \leq i, j \leq N$, si N es la cantidad de estados del HMM.
- $\boldsymbol{b} = \{b_j(\boldsymbol{o}_t)\}$. $b_j(\boldsymbol{o}_t)$ representa la probabilidad de emisión de la posible salida $\boldsymbol{o}_t$ en el estado q_j. Si las salidas son vectores de variables continuas, $b_j(\boldsymbol{o}_t)$ será un densidad de probabilidad. Si las observaciones son símbolos discretos, $b_j(\boldsymbol{o}_t)$ contendrá las probabilidades de emisión de cada uno de símbolos de salida posibles.
- $\boldsymbol{\pi} = \{\pi_j\}$. π_j representa la probabilidad de comenzar en el estado q_j.

Una cadena de Markov se puede considerar un caso particular de un HMM en el que en cada estado la probabilidad de emisión es 1 para un símbolo discreto dado y 0 para todos los demás.

2.2.2. Métodos básicos en los HMM

A continuación se presentan las herramientas o métodos necesarios para poder utilizar los HMM.

En primer lugar, necesitamos saber cómo computar la secuencia óptima de estados que represente una secuencia de observaciones dada. Este problema se conoce como *decodificación*. En el caso de la meteorología en Bilbao, la decodificación nos permitiría conocer cuál ha sido el tiempo en Bilbao durante los días en los que nuestro amigo nos ha estado dando datos.

Por otro lado, tenemos que crear el modelo. Este proceso se conoce como *entrenamiento*. El problema se plantea de la siguiente forma: dada una topología de HMM y una secuencia de observaciones, determinar los parámetros del HMM de forma que se maximice la probabilidad de que la secuencia de observaciones haya sido producida por el HMM. En nuestro ejemplo simple, ajustaríamos las probabilidades de transición entre días soleados y lluviosos y, en el caso más simple, las probabilidades de que nuestro amigo salga de fiesta o estudie en función del estado del tiempo.

Existen distintos algoritmos para resolver (o resolver aproximadamente) los problemas de la decodificación y del entrenamiento. Nosotros nos centraremos en explicar los algoritmos usados durante este trabajo.

Decodificación mediante el algoritmo de Viterbi

Una forma de resolver el problema de la decodificación es, dada una secuencia de observaciones, calcular todas las secuencias de estados posibles y sus probabilidades, y elegir aquella secuencia con mayor probabilidad. Aunque parece que este método de fuerza bruta extremadamente costoso computacionalmente, puede resolverse de forma eficiente si se resuelve de forma iterativa mediante el algoritmo *forward-backward* [52].

El algoritmo de *Viterbi* es más eficiente (rápido) para resolver el problema de la decodificación, ya que únicamente calcula la secuencia óptima. Si $\boldsymbol{o} = \{\boldsymbol{o_1}, \boldsymbol{o_2}, ..., \boldsymbol{o_T}\}$ es la secuencia de observaciones, el algoritmo de *Viterbi* conseguirá la secuencia de estados más probable con los siguientes pasos [52]:

1. **Inicialización**. Computar la probabilidad de comenzar cada estado q_j ($\delta_1(j)$) teniendo en cuenta las probabilidades iniciales y las probabilidad de emisión de la primera observación de cada estado.

$$\delta_1(j) = \boldsymbol{b_j(o_1)}\, \pi_j \tag{2.5}$$

2. **Recursión**. Por cada observación $\boldsymbol{o_t}$ con $2 \leq t \leq T$, determinar la probabilidad de estar en cada estado q_j en el instante t (llamaremos $\delta_t(j)$ a esta probabilidad) y el estado en el instante anterior a dicho

estado $t-1$ (llamaremos $\psi_t(j)$ al estado anterior). En principio, podemos haber llegado al estado q_j en el instante t desde cualquier estado q_i en el instante $t-1$. Se elige como estado anterior aquél que presente la mayor probabilidad teniendo en cuenta la probabilidad de haber estado dicho estado $\delta_{t-1}(i)$ y haber transitado al estado actual (Ecuación 2.7). Así, la probabilidad de estar en el estado actual será: la probabilidad de haber estado en el estado anterior, haber transitado al estado actual y haber emitido la observación $\boldsymbol{o_t}$ (Ecuación 2.6).

$$\delta_t(j) = \max_{1 \le i \le N} \left[\delta_{t-1}(i)\ a_{ij}\right] \boldsymbol{b_j(o_t)} \tag{2.6}$$

$$\psi_t(j) = \arg\max_{1 \le i \le N} \left[\delta_{t-1}(i)\ a_{ij}\right] \tag{2.7}$$

3. **Finalización**. Realizando el paso 2 desde $t = 2$ hasta $t = T$, conseguiremos las probabilidades de cada estado en el instante T y el estado anterior a cada uno. Obviamente, elegiremos como último estado h_T aquél que presente una mayor probabilidad (Ecuación 2.8). Esa probabilidad será la probabilidad total de la secuencia más probable $\boldsymbol{h}$ (Ecuación 2.9).

$$h_T = \arg\max_{1 \le i \le N} [\delta_T(i)] \tag{2.8}$$

$$P_{\boldsymbol{h}} = \max_{1 \le i \le N} [\delta_T(i)] \tag{2.9}$$

4. **Secuencia óptima**. Como hemos ido guardando el estado anterior a cada estado, podemos completar de forma sencilla la secuencia $\boldsymbol{h}$, aplicando la regla mostrada en la Ecuación 2.10 desde $t = T-1$ hasta $t = 1$.

$$h_t = \psi_{t+1}(h_t + 1) \tag{2.10}$$

Entrenamiento mediante el procedimiento de Viterbi

El *entrenamiento mediante el procedimiento de Viterbi* o *entrenamiento de Viterbi* es un proceso iterativo que permite ajustar los parámetros de un HMM para maximizar la probabilidad de que el HMM haya producido una o un conjunto de secuencias de observación $\{\boldsymbol{o^r}\}$, con $1 \le r \le R$. Esta secuencia o (normalmente) conjunto de secuencias se denominan *datos de entrenamiento* o *datos de aprendizaje*.

Como su propio nombre indica, este proceso esta basado en el algoritmo de Viterbi anteriormente explicado (Sección 2.2.2). El entrenamiento consta de los siguientes pasos [5]:

1. **Inicialización**. Hay que definir cuales serán los parámetros del HMM $\boldsymbol{a}$, $\boldsymbol{b}$ y $\boldsymbol{\pi}$ antes de comenzar el entrenamiento. En función de la aplicación, se pueden inicializar de forma aleatoria, todos los parámetros iguales, etc. .

2. **Alineación**. Alinear cada secuencia de *datos de aprendizaje*. Alinear significa decodificar, es decir, encontrar la secuencia óptima de estados que correspondiente para cada secuencia de *datos de aprendizaje*. La alineación se consigue mediante el algoritmo de Viterbi (Sección 2.2.2). En el caso del reconocimiento de voz, en la primera iteración se realiza una alineación uniforme (asignar a cada estado el mismo número de observaciones) [13].

3. **Reestimación**. Una vez tenemos alineadas las secuencias de *datos de aprendizaje*, estimamos las probabilidades de transición $\boldsymbol{a}$ de acuerdo a la frecuencia relativa en las que éstas se han producido:

$$\hat{a}_{ij} = \frac{C(a_{ij})}{C(a_i)}, \tag{2.11}$$

donde $\hat{a}_{ij}$ es la nueva probabilidad de transición del estado q_i al estado q_j, $C(a_{ij})$ el número de transiciones del estado q_i al estado q_j en las secuencias alineadas conseguidas en el paso 2, y $C(a_i)$ el número total de transiciones desde el estado q_i a cualquier estado en las secuencias alineadas.

Las probabilidades iniciales $\boldsymbol{\pi}$ se pueden computar de forma similar, contando y normalizando cuantas veces se ha comenzado en cada estado. Aún así, en el caso del reconocimiento de voz no suele

tener demasiado sentido el cálculo probabilidades, ya que vienen dadas por el léxico y el modelo de lenguaje.

Las probabilidades de emisión $\boldsymbol{b}$ se calcularán de forma distinta en función de si las observaciones son discretas o continuas. Si son discretas, las probabilidades se estiman contando y normalizando cuantas veces se ha observado cada salida posible en cada estado:

$$\hat{b}_j(v^k) = \frac{C(v_j^k)}{C(v_j)}, \tag{2.12}$$

donde $\hat{b}_j(v^k)$ es la nueva probabilidad de emisión del símbolo v^k en el estado q_j, $C(v_j^k)$ la cantidad de observaciones de la salida v^k alineadas con el estado q_j durante el entrenamiento, y $C(v_j)$ la cantidad total de observaciones alineadas con el estado q_j durante el entrenamiento.

Si estamos estimando vectores de variables continuas mediante mezclas de gaussianas (sección 2.1), las reestimaciones se calculan mediante las relaciones 2.13, 2.14 y 2.15.

Cada vector de medias de cada gaussiana con índice m de cada estado q_j se estima haciendo la media entre todos las observaciones del entrenamiento asociadas a cada una de dichas gaussianas.

$$\hat{\boldsymbol{\mu}}_{jm} = \frac{\sum_{r=1}^{R}\sum_{t=1}^{T} \mathbb{I}_j^m(\boldsymbol{o_t^r})\ \boldsymbol{o}_t^r}{\sum_{r=1}^{R}\sum_{t=1}^{T} \mathbb{I}_j^m(\boldsymbol{o_t^r})}, \tag{2.13}$$

donde la función indicador $\mathbb{I}_j^m(\boldsymbol{o_t^r})$ será 1 si y solo si el vector $\boldsymbol{o}_t^r$ está asociado a la gaussiana con índice m del estado q_j.

La matriz de covarianzas se puede estimar una vez que tenemos el nuevo vector de medias.

$$\hat{\boldsymbol{\Sigma}}_{jm} = \frac{\sum_{r=1}^{R}\sum_{t=1}^{T} \mathbb{I}_j^m(\boldsymbol{o_t^r})\ (\boldsymbol{o}_t^r - \hat{\boldsymbol{\mu}}_{jm})(\boldsymbol{o}_t^r - \hat{\boldsymbol{\mu}}_{jm})^T}{\sum_{r=1}^{R}\sum_{t=1}^{T} \mathbb{I}_j^m(\boldsymbol{o_t^r})} \tag{2.14}$$

Finalmente, podemos calcular los pesos de cada gaussiana en función de la cantidad de observaciones asociadas a cada una de ellas y la cantidad de vectores alineados con el estado q_j durante el entrenamiento.

$$\hat{c}_{jm} = \frac{\sum_{r=1}^{R}\sum_{t=1}^{T} \mathbb{I}_j^m(\boldsymbol{o_t^r})}{\sum_{r=1}^{R}\sum_{t=1}^{T}\sum_{l=1}^{M} \mathbb{I}_j^l(\boldsymbol{o_t^r})} \tag{2.15}$$

Capítulo 3

Redes Neuronales

Las redes neuronales o redes neuronales artificiales (NN o ANN, por sus siglas en inglés) son un paradigma de programación que sirven para procesar información. Están inspiradas en el funcionamiento de las neuronas y las conexiones del cerebro. Con una cierta estructura, las redes neuronales son funciones globales, es decir, son capaces de computar cualquier función $f : \mathbb{R}^n \longrightarrow \mathbb{R}^m$ [40].

Warren S. McCulloch y Walter Pitts introdujeron los primeros modelos matemáticos de las redes neuronales biológicas en 1943 [30]. Desde entonces, se han ido desarrollando tanto los modelos de redes neuronales como los algoritmos relacionados con ellos. Aún así, el *boom* de las redes neuronales no se ha dado hasta esta última decada. Debido al gran número de parámetros que hay que ajustar para utilizar redes neuronales para resolver problemas complejos como el reconocimiento de patrones, los tiempos de computación para trabajar con redes neuronales no eran prácticos en la mayoría de casos. La mejora en las computadoras en estos últimos años y el uso de GPUs (tarjetas gráficas) para realizar cálculos en paralelo han permitido el uso de las redes neuronales para tareas como el reconocimiento de voz, reconocimiento de objetos en imágenes, etc.

Las redes neuronales están formadas por un conjunto de unidades denominadas neuronas y por las interconexiones entre estas neuronas.

3.1. Neuronas

Las neuronas son las encargadas de computar funciones matemáticas dentro de una red neuronal. Una neurona consta de los siguientes elementos [27]:

- Una o más variables de entrada. Nos referiremos a estas entrada con el vector $\boldsymbol{x} = x_1, x_2, ..., x_N$, siendo N el número de entradas a la neurona.
- Un peso por variable de entrada. Podemos representar estos pesos con el siguiente vector $\boldsymbol{w} = w_1, w_2, ..., w_N$. Cada peso w_i corresponde a la variable de entrada x_i.
- Un parámetro adicional llamado bias. Se denota como b.
- Una salida, a.
- Una función de propagación o transferencia f_{prop} que preprocesa las entradas, los pesos y el bias para conseguir la entrada a la función de activación z; $z = f_{prop}(x_1, x_2, ..., x_N, w_1, w_2, ..., w_N, b)$.
- Una función de activación f_{act} que permite calcular la activación de la neurona en función de la salida de la función de propagación, $a = f_{act}(z)$.

En la Figura 3.1 podemos ver una representación gráfica de una neurona artificial.

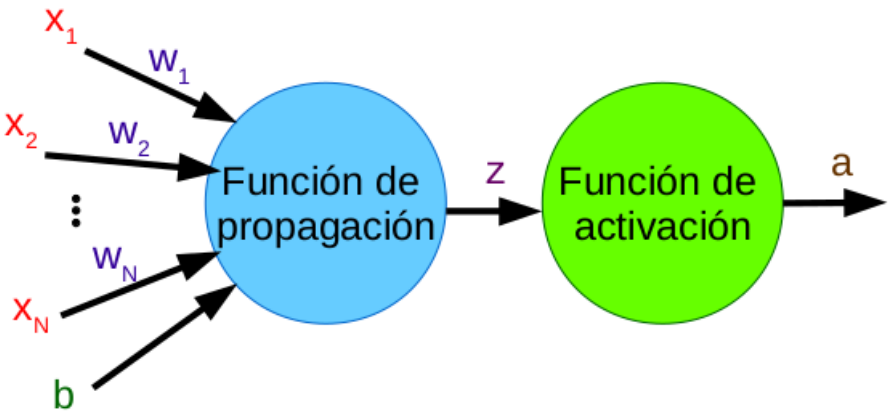


Figura 3.1: Representación gráfica de una neurona artificial.

3.1.1. Los pesos y el bias

Los pesos determinan la importancia de cada entrada a la neurona, el bias indica cómo de fácil es que una neurona se active. Esto está basado en el funcionamiento de las neuronas biológicas. Las neuronas biológicas se activan emitiendo un impulso eléctrico si a su entrada (las neuronas biológicas tienen una única entrada, z sería su equivalente en el modelo matemático) hay un potencial eléctrico mayor a un potencial umbral [27]. El bias es por tanto la contraposición del umbral. En el ámbito de reconocimiento de patrones, se suelen interpretar las salidas de las neuronas como indicadores de ciertas características de la muestra que se está examinando. Así, si entendemos que una neurona indica una característica dada, los pesos determinarán cómo de relacionada está cada entrada con la característica, y el bias cómo de fácil es detectar dicha característica.

3.1.2. Funciones de propagación y de activación

En cuanto a las funciones de propagación, se suele optar por la suma de las entradas (con pesos) y del bias. Durante este trabajo todas las funciones de propagación serán de este tipo. Es decir,

$$z = \sum_{i=1}^{N} w_i x_i + b \tag{3.1}$$

También es habitual que a una neurona le corresponda una única función de propagación, pero es posible que una neurona procese la salida de más de una función de propagación. En ese caso, la función de activación no será una función de una variable z, sino de varias variables (tantas como funciones de propagación), las cuales representaremos con el vector $\boldsymbol{z}$. Consecuentemente, no tendremos un solo bias por neurona y un solo peso por entrada, sino tantos como funciones de propagación estemos procesando. En la Figura 3.2 se muestra una representación gráfica de este tipo de neurona. Analizaremos dos ejemplos más adelante durante esta sección.

Las funciones de propagación y de activación comúnmente utilizadas en las redes neuronales hacen que las neuronas artificiales se comporten más o menos como las neuronas biológicas. Una diferencia notable es que mientras la salida de las neuronas biológicas es más bien binaria (activarse o no), las salidas de las neuronas artificiales son generalmente continuas. En nuestro caso, son funciones crecientes con todas sus variables, porque tanto las funciones de propagación como de activación son crecientes. Es decir, cuanto mayores sean las entradas (y sus correspondientes pesos) y menor sea el umbral (o mayor sea el bias) mayor será la salida. Otra propiedad usual de las neuronas es que la función de activación suele tener una salida acotada. Con estas tres propiedades (funciones continuas, crecientes y acotadas), se evitan inestabilidades numéricas durante el entrenamiento [70] y se facilita una interpretación probabilística de la salida de las neuronas.

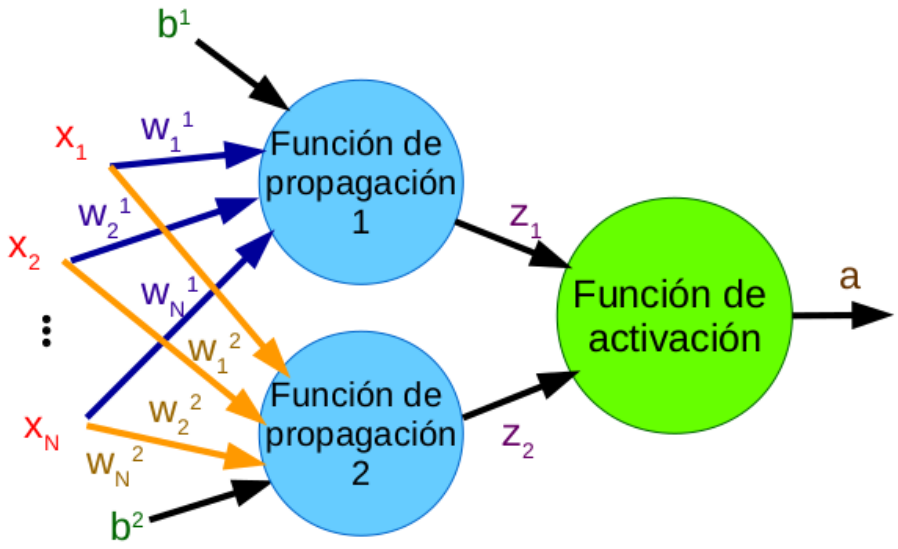


Figura 3.2: Representación gráfica de una neurona artificial con dos funciones de propagación.

A continuación introduciremos distintas funciones de activación utilizadas a lo largo de este trabajo.

Tangente hiperbólica

Un ejemplo muy común de función de activación es la tangente hiperbólica (Ecuación 3.2).

$$f_{act}(z) = \tanh(z) \tag{3.2}$$

Como se puede apreciar en la Figura 3.3, la tangente hiperbólica es una función creciente, suave y ofrece una salida acotada entre -1 y 1.

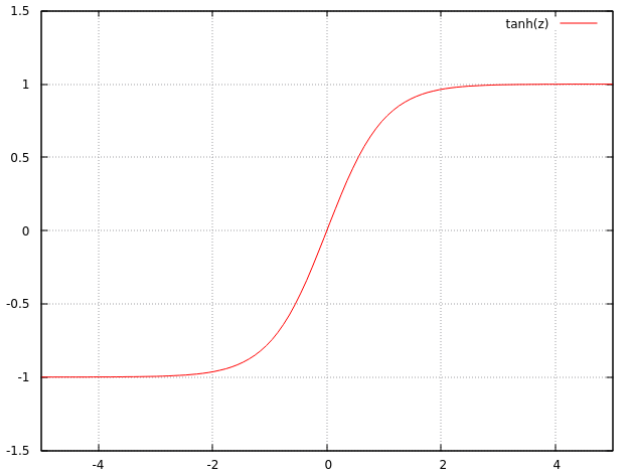


Figura 3.3: Gráfica de la tangente hiperbólica.

Función sigmoidal

La función sigmoidal o sigmoide (Ecuación 3.3) tiene la misma forma que la tangente hiperbólica, pero está acotada entre 0 y 1 (Figura 3.4).

$$f_{act}(z) = \sigma(z) \equiv \frac{1}{1+e^{-z}} \tag{3.3}$$

De hecho, la sigmoide es una versión escalada de la tangente hiperbólica (Ecuación 3.4).

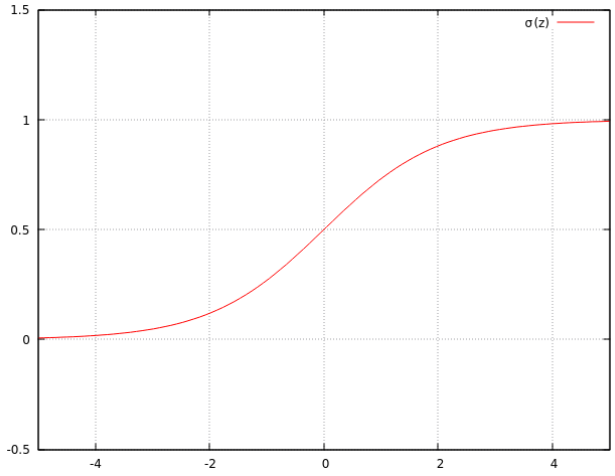


Figura 3.4: Gráfica de la función sigmoidana.

$$\left.\begin{aligned} \tanh(z) &\equiv \frac{e^z - e^{-z}}{e^z + e^{-z}} \\ \sigma(z) &\equiv \frac{1}{1+e^{-z}} \end{aligned}\right\} \Rightarrow \sigma(z) = \frac{1+\tanh(z/2)}{2} \tag{3.4}$$

Función ReLU

A continuación presentaremos dos funciones de activación muy diferentes a la tangente hiperbólica y a la sigmoide: las funciones ReLU y *pnorm*. La diferencia más notoria es que estas funciones no están acotadas, aunque ambas son crecientes y continuas.

La salida de la función ReLU (del inglés, *Rectifier Linear Unit*) será igual a la entrada si ésta es positiva, y 0 si la entrada es negativa (Ecuación 3.5, Figura 3.5).

$$f_{act}(z) = \text{ReLU}(z) \equiv \max(0, z) \tag{3.5}$$

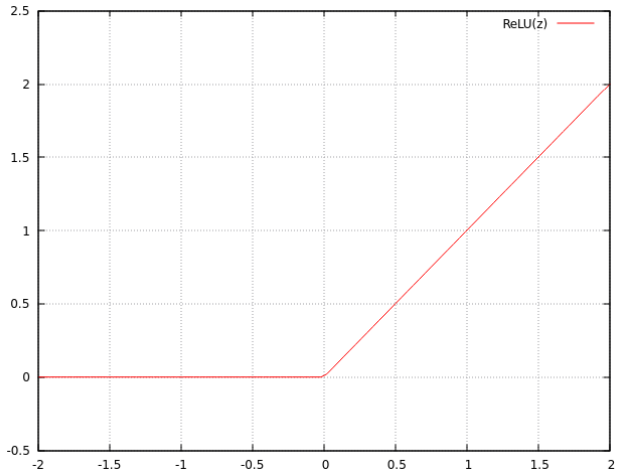


Figura 3.5: Gráfica de la función ReLU.

El hecho de que la función ReLU no sea acotada, por un lado, tiene la ventaja de que una red neuronal basada en esta función será más rápida de entrenar [39]. Por otro lado, las salidas muy grandes pueden acarrear problemas de estabilidad durante el entrenamiento [70]. Este problema se soluciona normalizando la salida de estas neuronas, con métodos que se describirán en la Sección 3.2.1.

Función *pnorm*

Como ejemplo de las neuronas que procesan la salida de más de una función de propagación, tenemos las denominadas neuronas *p-norm* [70], [18]. Su nombre proviene de la función de activación que utilizan, la función *p-norm*. Ésta permite calcular la p-norma entre salidas de distintas funciones de propagación z (Ecuación 3.6).

$$f_{act}(\boldsymbol{z}) = \left(\sum_{k=1}^{K} z_k^p \right)^{1/p} \tag{3.6}$$

donde $\boldsymbol{z} = z_1, z_2, ..., z_K$ es el vector que contiene la salida de las K funciones de propagación, es decir, las sumas de las entradas a la neuronas, computadas con distintos pesos y bias; y p es el orden de la normalización.

En la Figura 3.6 se muestra una gráfica de la función *pnorm* para dos variables y con $p = 2$.

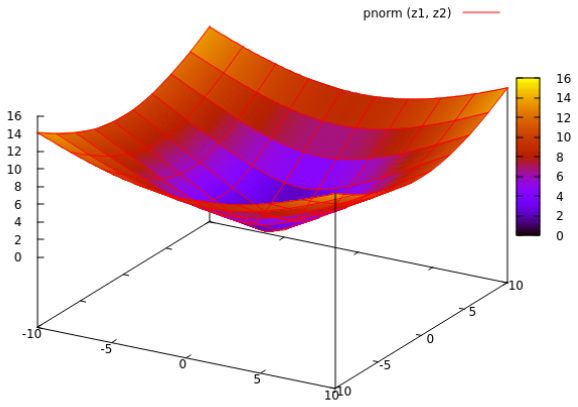


Figura 3.6: Gráfica de la función *pnorm* para dos variables. El valor de p se ha fijado en 2.

Al igual que ocurre con la función ReLU, aunque la función *p-norm* es continua y creciente, su salida puede ser cualquier número real positivo. Por tanto, en este caso es también necesario normalizar la salida de las neuronas.

Función *softmax*

Durante este trabajo siempre utilizaremos las redes neuronales como estimadores de probabilidad. La red tendrá tantas neuronas en la última capa o capa de salida como clases entre las que estamos intentando clasificar los vectores de entrada. La salida de cada neurona de la capa de salida se interpretará como la probabilidad de que un vector de entrada pertenezca a una de las posibles clases.

Hay que decir que para hacer una interpretación probabilística de la salida, es común (aunque no completamente necesario [8]) que las salidas de todas las neuronas de la capa de salida estén acotadas entre 0 y 1, y que la suma de todas ellas sea 1. Existen distintos métodos de lograr una salida de estas características. Uno de los más utilizados es que la función de activación de la última capa sea la función *softmax* [20]. La función *softmax* normaliza el exponencial la salida de la función de transferencia asociada a cada neurona, en función de la suma de los exponenciales de las salida de las funciones de transferencia correspondientes al resto de neuronas de la capa (Ecuación 3.7).

$$f_{act\,j}(\boldsymbol{z}) = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}, \tag{3.7}$$

donde $f_{act\,j}$ es la función de activación asociada a la neurona j, y $\boldsymbol{z} = z_1, z_2, ..., z_K$ es el vector que contiene las sumas de las entradas pesos y bias correspondientes a cada una de las K neuronas de la capa de salida.

Por tanto, la función de activación *softmax* computa la salida de las neuronas en función de las salidas de varias funciones de propagación, al igual que las neuronas del tipo *pnorm*.

3.2. Estructura de las redes neuronales

3.2.1. Redes *feedforward*

Para construir una red neuronal capaz de resolver problemas complejos es necesario interconectar un cierto número de neuronas. A la hora de realizar las interconexiones, son ampliamente utilizadas las estructuras por capas *feedforward*. En una red *feedforward*, las neuronas se organizan en distintas capas, de forma que las salidas de las neuronas de una capa se conecten a las entradas de todas (o varias) las neuronas de la capa posterior. La razón de agrupar las neuronas en capas es que así podremos entrenar eficientemente las redes con los algoritmos que describiremos en la Sección 3.3.

Para calcular el vector de salida correspondiente a un vector de entrada de una red neuronal *feedforward* basta conocer todos los pesos y bias de todas las neuronas de la red (y ser capaz de computar todos). El cálculo de la salida es sencillo: calculamos las salidas de la primera capa en función de las entradas de la red y de los parámetros de las neuronas de la primera capa mediante las funciones de propagación y activación, después calculamos las salidas de la segunda capa en función de las salidas de la primera capa de la misma forma, y repetimos el mismo procedimiento hasta conseguir el vector de salida, el vector que forman las salidas de las neuronas de la última capa.

Es decir, la salida de una red neuronal estructurada en capas se computa propagando hacia delante la entrada, mediante los bloques de funciones de propagación y las funciones de activación de cada capa. Para describir este proceso analíticamente, supongamos que tenemos una red neuronal formada por L capas, tal y como se muestra en la Figura 3.7.

Cada capa l (el nombre de este índice l proviene de la palabra inglesa *layer*) está formada por un bloque de funciones de propagación y por una función de activación por neurona. El bloque de funciones de propagación propaga el vector de salida de la capa anterior a la entrada de las funciones de activación de la propia capa. Hablaremos de esta cantidad de funciones de propagación como la dimensión de entrada de la capa. En principio, este número no ha de ser igual al número de neuronas de la capa, ya que, como se ha descrito en la Sección 3.1.2, puede que una neurona tome como entradas las salidas de varias funciones de activación (las neuronas tipo *pnorm*, por ejemplo). Cada una de las funciones de las mencionadas funciones de propagación propagación f_{prop_j} se computa tal y como aparece en la Ecuación 3.8.

$$f_{prop_j} = z_j^l = \sum_{i=1}^{N} w_{ij} a_i^{l-1} + b_j, \tag{3.8}$$

donde N es la dimensión del vector de salida de la capa $l-1$, a_i^{l-1} es la salida de la neurona i de la capa $l-1$, w_{ij} es el peso que relaciona la salida de la neurona i de la capa $l-1$ con la función de propagación f_{prop_j}, y b_j el bias asociado a la función de propagación f_{prop_j}.

Una vez listas las salidas del bloque de funciones de propagación, cada una de las salidas de la capa l se puede computar haciendo uso de, por ejemplo, las funciones de activación introducidas en la 3.1.2 (Ecuación 3.9)

$$a_k^l = f_{act}(\boldsymbol{z}_k^l), \tag{3.9}$$

donde a_k^l es la salida de la neurona k de la capa l, y $\boldsymbol{z}_k^l$ el vector de entrada a la función de activación. En el caso de que la función de activación tenga una sola entrada, podemos simplificar la Ecuación 3.9 como se muestra en la Ecuación Ecuación 3.10.

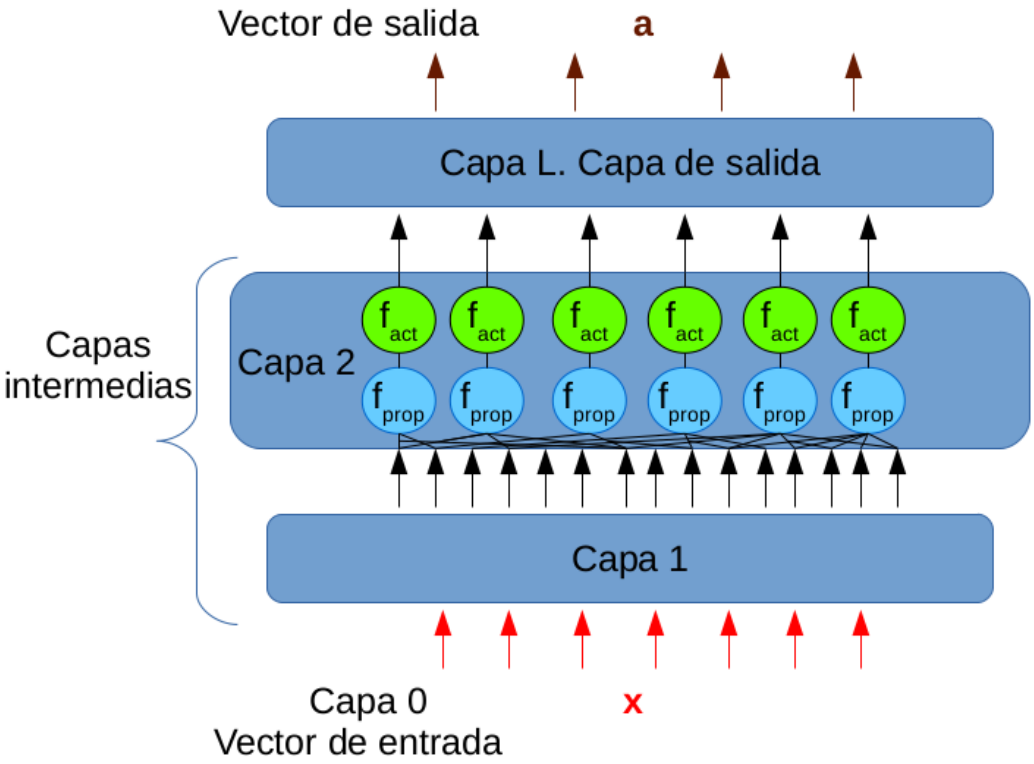


Figura 3.7: Representación gráfica de una red neuronal *feedforward*. Nótese que la primera capa ($l = 0$) mostrada en los ejemplo no es realmente una capa de neuronas como tal, simplemente se utiliza para representar las entradas a la red. La primera capa de neuronas que *realiza* cálculos es por tanto la segunda ($l = 1$).

$$a_k^l = f_{act}(z_j^l), \tag{3.10}$$

donde $k = j$. Las Ecuaciones 3.8 y 3.9 muestran como calcular la salida de una capa l de una red *feedforward* en función de la capa anterior $l-1$. Por tanto, para computar la salida de la red basta con aplicar secuencialmente las Ecuaciones 3.8 y 3.9 empezando por $l = 1$ hasta $l = L$.

Aún así, como hemos mencionado al analizar las funciones de activación ReLU y *pnorm*, en ocasiones es necesario añadir componentes de de normalización a algunas capas. En ese caso hay que añadir una ecuación más para computar la salida de éstas. La normalización que aplicaremos en este trabajo a las capas ReLU y *pnorm* se expresa en la Ecuación 3.11 [70].

$$\overline{a_k^l} = \begin{cases} a_k^l, & \text{si } \sigma \leq 1, \\ \dfrac{a_k^l}{\sigma}, & \text{si } \sigma > 1, \end{cases} \qquad \text{donde } \sigma = \sqrt{\frac{1}{M}\sum_{k=1}^{M} {a_k^l}^2}, \tag{3.11}$$

donde $\overline{a_k^l}$ es la salida normalizada. En la Ecuación 3.11 σ es la desviación estándar de a_k^l. Con esta normalización se consigue evitar que la desviación estándar exceda 1, lo cual tiende a estabilizar el entrenamiento [70].

Redes TDNN

Las redes TDNN (del inglés, *Time Delay Neural Network*) son un tipo de redes neuronales cuya estructura está diseñada para trabajar sobre secuencias de vectores de entrada. Las TDNN toman en su entrada una cantidad considerable de vectores de la secuencia (unos veinte o treinta vectores consecutivos en nuestro caso, por ejemplo). A diferencia de las redes *feedforward* descritas hasta ahora, en las TDNN las capas no

están completamente conectadas entre ellas. Grupos de vectores de entrada correspondientes a instantes distintos se procesan en paralelo de forma independiente a lo largo de la red. Este concepto es equivalente a que la salida de una capa depende de la salida de la capa anterior en distintos instantes de tiempo [65].

Para explicar esto supongamos que tenemos una red neuronal A *feedforward* convencional muy simple formada únicamente por una capa de entrada (la cual no realiza cálculos) y una de salida. Independientemente de la red A, tenemos una red B, formada también por una capa de entrada y una capa de salida. Supongamos ahora que introducimos a la red B un vector o conjunto de vectores centrados en el instante $t-1$. Obtenemos así $B(t-1)$, el vector de salida de la red B correspondiente a ese instante. Igualmente, podemos introducir a la red B un vector o conjunto de vectores centrados en el instante t para obtener $B(t)$. Si ahora conectamos los vectores $B(t-1)$ y $B(t)$ a la entrada de la red A, la red formada por la red A y por la red B en los dos instantes (llamaremos a esta red resultante $A + B_t + B_{t-1}$) es una TDNN.

La otra interpretación equivalente de la TDNN $A + B_t + B_{t-1}$ se analiza a continuación. La TDNN resultante es una red *feedforward* cuya capa de entrada es la unión de las capas de entrada de la red B centrada en los instantes t y $t-1$. Es decir, si por ejemplo la red B_{t-1} tomaba como entrada los vectores $\boldsymbol{x}_{t-2}$, $\boldsymbol{x}_{t-1}$, y $\boldsymbol{x}_t$; y la red B_t, $\boldsymbol{x}_{t-1}$, $\boldsymbol{x}_t$, y $\boldsymbol{x}_{t+1}$; la red $A + B_t + B_{t-1}$ tomará como entrada los vectores $\boldsymbol{x}_{t-2}$, $\boldsymbol{x}_{t-1}$, $\boldsymbol{x}_t$, y $\boldsymbol{x}_{t+1}$. La red $A + B_t + B_{t-1}$ tendrá una única capa intermedia dividida en dos subcapas idénticas, cada una de ellas igual a la capa de salida de la red B. La primera subcapa tomará como entrada $\boldsymbol{x}_{t-2}$, $\boldsymbol{x}_{t-1}$, y $\boldsymbol{x}_t$; y la segunda subcapa $\boldsymbol{x}_{t-1}$, $\boldsymbol{x}_t$, y $\boldsymbol{x}_{t+1}$. Por tanto, las capas intermedias de las redes TDNN no son *fully-connected*. Finalmente, la capa de salida de la TDNN $A + B_t + B_{t-1}$ es igual a la capa de salida de la red A. La capa de salida sí está *fully-connected* con la penúltima capa. En la Figura 3.8 se muestra este ejemplo de forma gráfica.

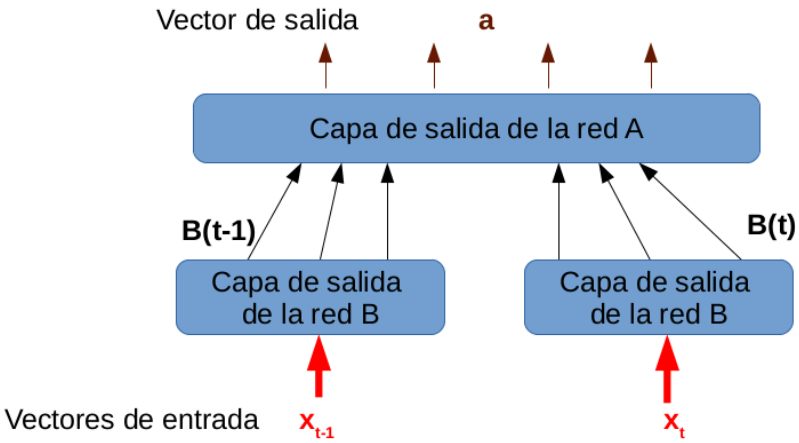


Figura 3.8: Ejemplo de la red TDNN simple.

Las TDNN que utilizaremos en este trabajo son más complejas, pero su funcionamiento es similar al del ejemplo anterior. Al igual que a la red A introducíamos la salida de la red B en varios instantes de tiempo, la red B puede tomar como entrada la salida de una red C en distintos instantes, etc. En la Figura 3.9 se muestran dos ejemplos de TDNN más grandes.

Con esta estructura se pretende la red aprenda la estructura temporal de los eventos acústicos [65] y que las características que aprenda cada capa sean independientes del instante en el que éstas ocurren, ya que las entradas correspondientes a distintos instantes se procesan de forma idéntica [65], [43].

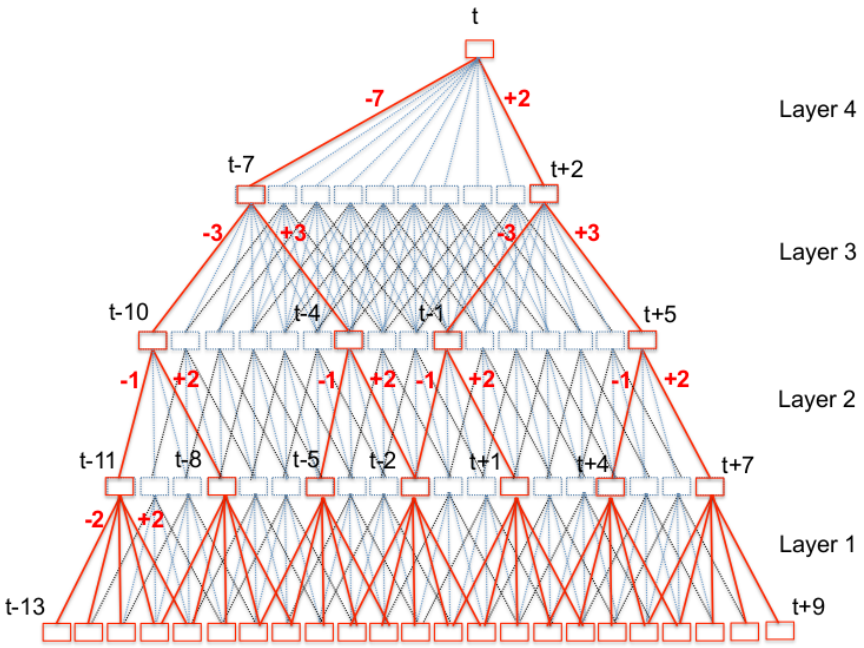


Figura 3.9: Dos ejemplos más de TDNN. En cada capa, cada rectángulo rojo simboliza las subcapas idénticas que forman la capa. Fuente: [43].

3.2.2. Redes recurrentes

En ciertas configuraciones, las salidas de algunas capas se realimentan, de forma que la salidas de una capa retroalimentada depende de sus salidas anteriores [20]. Las redes que incluyen este tipo de capas se denominan redes neuronales recurrentes (RNN, por sus siglas en inglés). La relación entre las salidas actuales de una capa se relacionan con las salidas en el instante anterior mediante una extensión del bloque de funciones de propagación, tal y como se muestra en la Figura 3.10.

Debido a su estructura, las RNN son útiles cuando se están analizando secuencias de vectores de entrada. Para detallar como computar la salida en este tipo de redes, supongamos que que queremos computar las salidas de la red para una secuencia de longitud T de vectores de entrada. Supongamos también que la red está compuesta por L capas, entre las cuales puede haber capas recurrentes y no recurrentes. Igual que en el caso de las redes *feedforward*, para calcular la salida de la red hemos de ir computando las salidas de cada una de las capas, empezando por las iniciales.

Si la capa l es una capa recurrente de la red, su salida para el instante t ($1 \leq t \leq T$) se calcula de la siguiente forma.

1. Calcular la salida de la función de propagación correspondiente a cada neurona (Ecuación 3.12). Por simpleza supondremos que a cada neurona le equivale una única función de propagación (el caso más general es muy similar).

$$f_{prop_j} = z_j^l(t) = \sum_{i=1}^{N} w_{ij} a_i^{l-1}(t) + \sum_{j=1}^{M} v_j a_j^l(t-1) + b_j, \tag{3.12}$$

   donde v_j es el peso correspondiente a la salida de la neurona j de la capa l en el instante $t-1$, y M la cantidad de neuronas de la capa l

2. Calcular la salida de cada neurona en función de las salidas de las funciones de propagación calculadas (Ecuación 3.13).

$$a_j^l(t) = f_{act}\left(z_j^l(t)\right) \tag{3.13}$$

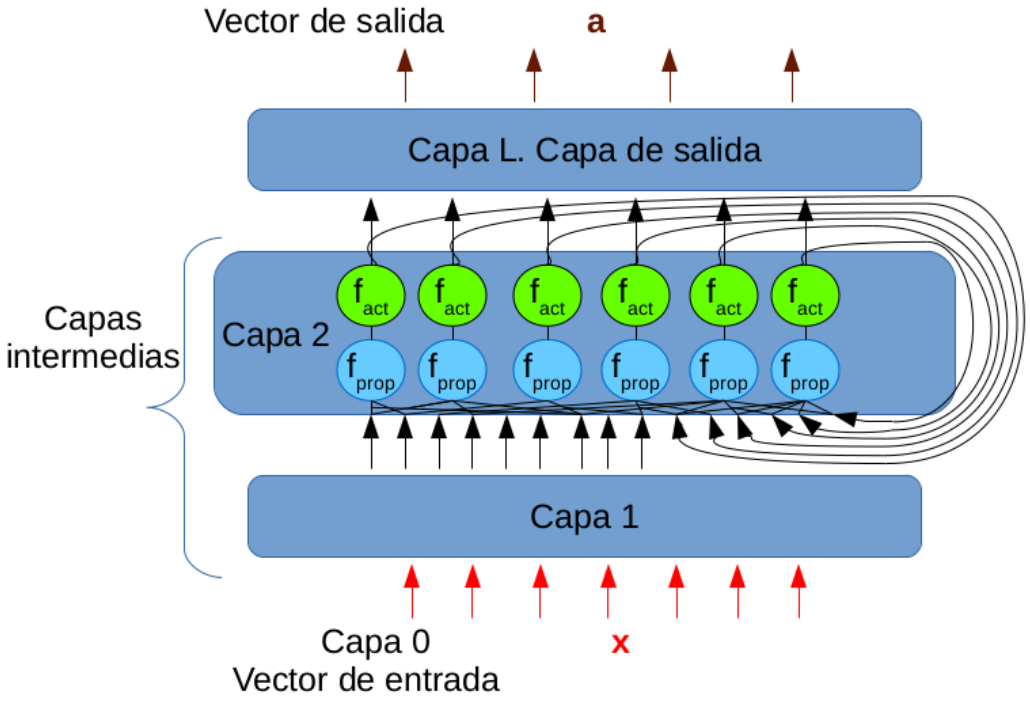


Figura 3.10: Representación gráfica de una red neuronal recurrente.

Para realizar los dos pasos anteriores se ha supuesto que conocemos las salidas de la capa l en el instante $t-1$. Para calcular este vector se aplican las Ecuaciones 3.12 y 3.13 desde $t=1$ (con unas condiciones iniciales dadas) hasta $t=t$. Por tanto, para calcular la salida en el instante t, necesitamos también conocer las entradas a la red correspondientes a los instantes $t=1,2,...,t-1$.

Redes recurrentes bidireccionales

Una extensión de las redes recurrentes son las redes recurrentes bidireccionales. En el caso anterior hemos descrito como funcionan las redes neuronales que contienen capas en las que su salida depende de su salida en el instante $t-1$ (lo que implica que depende de todos los instantes anteriores). Una versión completamente equivalente (matemáticamente) serían las redes recurrentes que contienen capas cuya salida depende de los instantes siguientes, es decir, de $t+1$ (y recursivamente de $t=T$).

Las redes recurrentes bidireccionales (BiRNN) contienen capas recursivas en ambos sentidos [31]. Por tanto, para computar la salida de este tipo de redes en el instante t necesitamos conocer las entradas a la red correspondientes a todos los instantes de la secuencia.

Redes LSTM

Las redes LSTM (del inglés *Long Short-Term Memory networks*) son un tipo de RNN cuyo rendimiento se ha demostrado ser alto en una gran variedad de aplicaciones, entre las cuales se encuentra el reconocimiento de voz [17]. Una capa de una red LSTM está formada por celdas de memoria. Estas celdas funcionan de forma similar a las RNN analizadas hasta ahora, con el añadido de que no solo se realimenta su salida, sino también el *estado* de la celda [41]. En la Figura 3.11 se muestra un esquema de una celda LSTM. Este esquema es el mismo utilizado por Alex Graves en [17].

En la Figura 3.11 se muestra el esquema de funcionamiento de una celda LSTM. Como se aprecia, la salida a_t depende de la entrada a la celda z_t, la memoria de la celda en el instante anterior c_{t-1}, de la salida de la celda en el instante anterior a_{t-1}. En la celda, además, podemos encontrar diferentes *gates* con diferentes cometidos. Cualitativamente, la memoria de la celda indica cómo de fácil es que se active la celda teniendo en cuenta el estado de la celda LSTM en los anteriores estados. El valor de la *forget gate* (f_t) se multiplica a

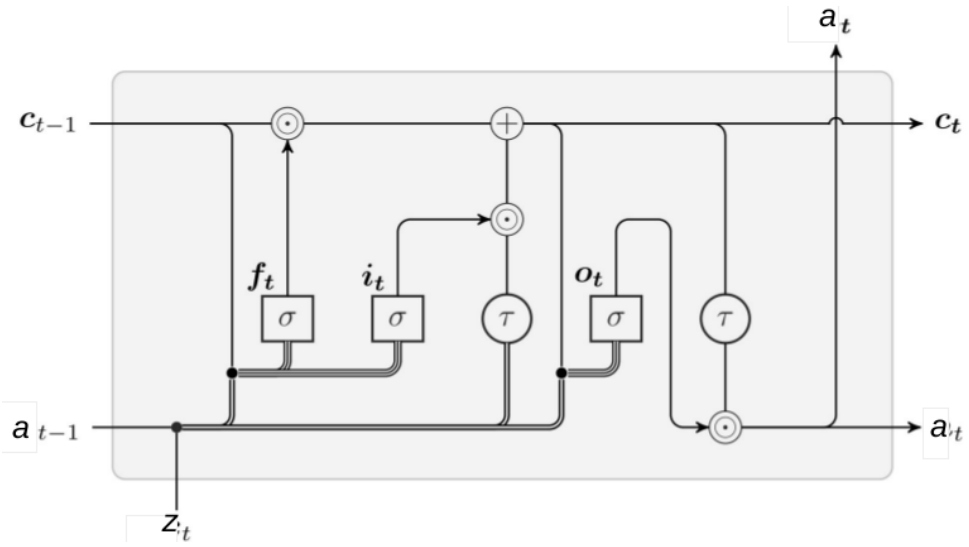


Figura 3.11: Esquema de una sola celda LSTM. La letra griega τ simboliza la función tanh. Autor: Majid al-Dosari.

la memoria de la celda. Así, la *forget gate* (acotada entre 0 y 1) decide cuanto de la memoria tenemos que olvidar o retener. La *input gate* (i_t) se encarga de controlar la cantidad que tenemos que añadir o restar a la memoria. Finalmente, la *output gate* (o_t), al igual que la memoria, interviene en el valor final de la salida de la celda [23], [41].

La salida de la celda a_t (Ecuación 3.18), se escribe en función de los valores f_t (Ecuación 3.15), i_t (Ecuación 3.14), o_t (Ecuación 3.17) y c_t (Ecuación 3.16) [17].

$$i_t = \sigma\left(w_z^i z_t + w_a^i a_{t-1} + w_c^i c_{t-1} + b^i\right) \tag{3.14}$$

$$f_t = \sigma\left(w_z^f z_t + w_a^f a_{t-1} + w_c^f c_{t-1} + b^f\right) \tag{3.15}$$

$$c_t = f_t c_{t-1} \tanh\left(w_z^c z_t + w_a^c a_{t-1} + b^c\right) \tag{3.16}$$

$$o_t = \sigma\left(w_z^o z_t + w_a^o a_{t-1} + w_c^o c_t + b^o\right) \tag{3.17}$$

$$a_t = o_t \tanh\left(c_t\right), \tag{3.18}$$

donde cada w_b^d es el peso correspondiente a la variable b y a la celda d. Hay que resaltar que todas las variables, valores de celda que aparecen en las Ecuaciones 3.14, 3.15, 3.16, 3.17 y 3.18 son escalares.

Hasta ahora se ha descrito el funcionamiento de una celda LSTM. Una capa LSTM está formada por varias (unas 1000, en los experimentos de este trabajo) de estas celdas. Para crear una capa LSTM tan solo nos falta añadir tantas funciones de propagación como celdas vaya a haber en la capa. Mediante cada función de propagación calcularemos las entradas a las celdas LSTM (la z_t que aparece en las cinco ecuaciones anteriores) de la forma habitual (Ecuación 3.8).

Al igual que con todas las RNN, se puede implementar una red LSTM bidireccional con distintas capas LSTM, si en algunas se tiene en cuenta el *pasado* y en las otras el *futuro* [17].

3.3. Entrenamiento de las redes neuronales

Al igual que en los HMM, para que las redes neuronales sean útiles en la práctica es necesario un algoritmo o conjunto de algoritmos que permita ajustar los parámetros de estas redes (en general, los pesos y bias) a partir de unos datos o ejemplos de entrenamiento. En este caso los ejemplos de entrenamiento

consisten en un conjunto de parejas $(\boldsymbol{x}_i, \boldsymbol{y}_i)$; donde $\boldsymbol{x}_i$ denota cada una de las entradas a la red, $\boldsymbol{y}_i$ la salida deseada para la entrada $\boldsymbol{x}_i$, y i es un índice que varía entre 1 y N, siendo N la cantidad de ejemplos de entrenamiento.

3.3.1. Función objetivo o de coste

El primer paso para entrenar redes neuronales es definir una función que indique como de bien se ajustan sus parámetros al conjunto de los ejemplos de entrenamiento [37]. Esta función se conoce como *función de coste* o *función objetivo*. Una propiedad usual de las funciones de coste es que se computan haciendo la media entre los costes correspondientes a cada ejemplo de entrenamiento (Ecuación 3.19).

$$C(\boldsymbol{W}, \boldsymbol{b}) = \frac{1}{N} \sum_{i=1}^{N} C_i(\boldsymbol{W}, \boldsymbol{b}), \tag{3.19}$$

donde $\boldsymbol{W}$ representa el conjunto de todos los pesos de la red neuronal, $\boldsymbol{b}$ todos los bias, y $C_i(\boldsymbol{W}, \boldsymbol{b})$ es el coste correspondiente al ejemplo de entrenamiento i. Para entender como funcionan estos costes, empecemos introduciendo la función de coste cuadrático C_C (Ecuación 3.20).

$$C_C(\boldsymbol{W}, \boldsymbol{b}) = \frac{1}{N} \sum_{i=1}^{N} \frac{\|\boldsymbol{y}_i - \boldsymbol{a}(\boldsymbol{x}_i)\|^2}{2}, \tag{3.20}$$

donde $\boldsymbol{a}(\boldsymbol{x}_i)$ es el vector de salida de la red para el vector de entrada $\boldsymbol{x}_i$, que aunque no se muestra explícitamente en la ecuación (por simplicidad), por supuesto depende de $\boldsymbol{W}$ y $\boldsymbol{b}$.

Si analizamos la función de coste cuadrático observamos que $C_C(\boldsymbol{W}, \boldsymbol{b}) \geq 0$. Además cuanto mayor sea la diferencia entre los vectores de salida deseados y los vectores de salida de la red, mayor será el coste cuadrático. Si por el contrario las salidas de la redes se parecen a los vectores deseados, el coste cuadrático disminuirá. Así, tenemos que ajustar los pesos y los bias para que $C_c(\boldsymbol{W}, \boldsymbol{b}) \approx 0$.

Aunque la función de coste es válida para ajustar los pesos y bias, existen otras funciones más adecuadas para el entrenamiento de redes neuronales. En este trabajo se utilizará la función de coste *cross-entropy* C_{CE} (Ecuación 3.21) [63] [39], la cual permite un entrenamiento más rápido que la función de coste cuadrático [39].

$$C_{CE}(\boldsymbol{W}, \boldsymbol{b}) = -\frac{1}{N} \sum_{i=1}^{N} \sum_{m=1}^{M} y_{i\,m} \ln a_m(\boldsymbol{x}_i), \tag{3.21}$$

donde $y_{i\,m}$ y $a_m(\boldsymbol{x}_i)$ son cada uno de los M componentes de cada uno de los vectores de salida deseados y obtenidos (a partir de la entrada $\boldsymbol{x}_i$), respectivamente.

Al igual que la función de coste cuadrático, la función de coste *cross-entropy* es siempre mayor o igual a cero, y más pequeña cuanto menos difieran las salidas esperadas y las obtenidas.

En cualquier caso, nuestro objetivo es minimizar la función de coste. Podría parecer que una opción sería calcular el mínimo de esa función analíticamente, pero las redes neuronales convencionales suelen depender (y de forma compleja) de al menos varios miles de variables, así que calcular todas las derivadas parciales, igualarlas a cero, y resolver el sistema analíticamente no es permisible. Por tanto, se suele resolver el problema de la optimización de forma numérica, mediante el algoritmo de descenso de gradiente.

3.3.2. Descenso de gradiente

El descenso de gradiente permite calcular un mínimo local de una función diferenciable de varias variables. Es un algoritmo iterativo cuyo funcionamiento se describe a continuación [37]:

1. Comenzar calculando el valor del gradiente de la función para un punto inicial (elegido aleatoriamente, por ejemplo). Si la función a minimizar es $f(\boldsymbol{x}) = f(x_1, x_2, ..., x_D)$, el gradiente en el punto inicial $\boldsymbol{x}_0$ se calcula de acuerdo a la Ecuación 3.22.

$$\nabla C(\boldsymbol{x}_0) = \left(\frac{\partial C}{\partial x_1}, \frac{\partial C}{\partial x_2}, ..., \frac{\partial C}{\partial x_D}\right)\Bigg|_{\boldsymbol{x}=\boldsymbol{x}_0} \tag{3.22}$$

2. Calcular el siguiente punto $\boldsymbol{x}_{t+1}$, basándonos en el punto anterior $\boldsymbol{x}_t$ ($\boldsymbol{x}_0$ en la primera iteración) y el gradiente correspondiente $\nabla C(\boldsymbol{x}_t)$ (Ecuación 3.23).

$$\boldsymbol{x}_{t+1} = \boldsymbol{x}_t - \eta \nabla C(\boldsymbol{x}_t), \tag{3.23}$$

donde η es una constante positiva conocida como velocidad de aprendizaje o *learning rate*. Es fácil demostrar que el valor de la función en el nuevo punto que encontramos con esta actualización de variables es menor al anterior. Siempre que la constante η sea suficientemente pequeña se puede considerar que el cambio en el valor de la función viene dado por la expresión mostrada en la Ecuación 3.24.

$$\Delta C \approx \nabla C \cdot \Delta \boldsymbol{x} = \nabla C \cdot (-\eta \nabla C(\boldsymbol{x}_t)) = -\eta |\nabla C|^2 \leq 0 \tag{3.24}$$

Una vez actualizadas las variables, podemos volver a calcular el gradiente de la función (Ecuación 3.22), o si consideramos que ya hemos reducido suficiente el valor de la función de coste, podemos acabar el algoritmo, quedándonos con los últimos valores de las variables. Cada una de las iteraciones es conocida como época de entrenamiento.

En el caso del entrenamiento de las redes neuronales, las ecuaciones de actualización de variables indican cómo hemos de cambiar los pesos y bias para conseguir que las salidas de la red sean las deseadas (Ecuaciones 3.25 y 3.26).

$$w_{j\,t+1} = w_{j\,t} - \eta \frac{\partial C}{\partial w_{j\,t}} \tag{3.25}$$

$$b_{j\,t+1} = b_{j\,t} - \eta \frac{\partial C}{\partial b_{j\,t}} \tag{3.26}$$

Aunque el algoritmo de descenso de gradiente funciona y permite ajustar los parámetros de las redes neuronales, presenta dos problemas. El primero es que tan solo nos permite encontrar un mínimo local de la función de coste, siempre que utilicemos un número de épocas y un valor de velocidad de aprendizaje adecuados (Figura 3.12).

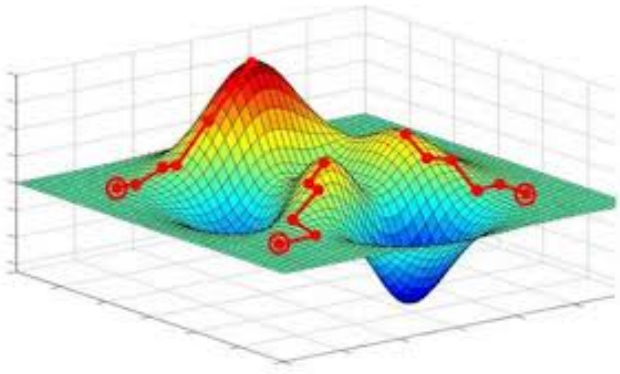


Figura 3.12: Visualización gráfica del algoritmo de descenso de gradiente sobre una función de dos variables. Se muestran tres ejemplos, empezando desde tres puntos distintos,en los que no se encuentra el mínimo absoluto de la función. Fuente [59].

El otro inconveniente es que es necesario computar el gradiente de la función de coste. Como se puede observar en la Ecuación 3.19, la funciones de coste constan de N componentes, una por ejemplo de entrenamiento. Por tanto, gracias a la linealidad del gradiente, podemos computar el gradiente de la función

de coste haciendo la media entre los gradiente correspondientes a cada ejemplo de entrenamiento. Si el volumen de datos de entrenamiento es muy grande (y en el caso de reconocimiento de voz suele serlo), el tiempo entre las actualizaciones de variables es demasiado grande.

Debido a estos dos inconvenientes se suele optar por una versión del algoritmo de descenso de gradiente conocido descenso de gradiente estocástico para entrenar redes neuronales.

3.3.3. Descenso de gradiente estocástico

El algoritmo de descenso de gradiente estocástico se basa en dividir cada época en lotes aleatorios conocidos como *mini-batches*, y actualizar los parámetros de la red neuronal con cada uno de ellos [37]. El funcionamiento de esta variante del descenso de gradiente se detalla a continuación.

1. Dividir aleatoriamente los ejemplos de entrenamiento en *mini-batches* de tamaño fijo K (el valor de K suele fijarse entre 50 y 256).

2. Por cada *mini-batch*, estimar el gradiente de la función de coste. Para ello se aplica la aproximación de la función de coste que se muestra en la Ecuación 3.27.

$$C(\boldsymbol{W},\boldsymbol{b}) = \frac{1}{N}\sum_{i=1}^{N} C_i(\boldsymbol{W},\boldsymbol{b}) \approx \frac{1}{K}\sum_{k=1}^{K} C_k(\boldsymbol{W},\boldsymbol{b}), \tag{3.27}$$

   donde C_k es cada uno de los componentes de la función de coste correspondientes al *mini-batch*. Con esta aproximación, y teniendo en cuenta la linealidad del gradiente, las ecuaciones de actualización de los pesos y bias son las expresadas en las Ecuaciones 3.28 y 3.29.

$$w_{j\,t+1} = w_{j\,t} - \frac{\eta}{K}\sum_{k=1}^{K}\frac{\partial C_k}{\partial w_{j\,t}} \approx w_{j\,t} - \eta\frac{\partial C}{\partial w_{j\,t}} \tag{3.28}$$

$$b_{j\,t+1} = b_{j\,t} - \frac{\eta}{K}\sum_{k=1}^{K}\frac{\partial C_k}{\partial b_{j\,t}} \approx b_{j\,t} - \eta\frac{\partial C}{\partial b_{j\,t}} \tag{3.29}$$

   Al igual que en el algoritmo de descenso de gradiente, en este punto podemos volver a entrenar la red una época más, o quedarnos con los últimos pesos y bias.

Una vez entendido el funcionamiento del algoritmo, podemos hacernos una idea de por qué el descenso de gradiente estocástico soluciona los mencionados problemas del descenso de gradiente. Por una parte, al actualizar las variables más a menudo, se acelera el crecimiento. Además, el hecho de computar una aproximación del gradiente hace que el algoritmo no tienda a converger hacia el mismo mínimo local todo el rato (Figura 3.13), de forma que es más probable acabar en un mínimo local de menor valor o en el mínimo global de la función [56].

Para completar la sección del entrenamiento de las redes neuronales, es necesario describir el algoritmo de *backpropagation* o propagación hacia atrás. Éste ofrece una forma rápida para computar las derivadas parciales que aparecen en las Ecuaciones 3.28 y 3.29.

3.3.4. Propagación hacia atrás

El algoritmo de propagación hacia atrás fue introducido en la década de los 70, pero hasta 1986 no se extendió su uso. En ese año David Rumelhart, Geoffrey Hinton y Ronald Williams demostraron que mediante la propagación hacia atrás se pueden entrenar de forma eficiente redes neuronales de varias capas.

Para calcular las derivadas parciales correspondientes a cada uno de los ejemplos de entrenamiento, se define la función de error δ correspondiente a una neurona de la red (Ecuación 3.30), que representa cuánto varía la función de coste (correspondiente a un ejemplo de entrenamiento) al cambiar la entrada a

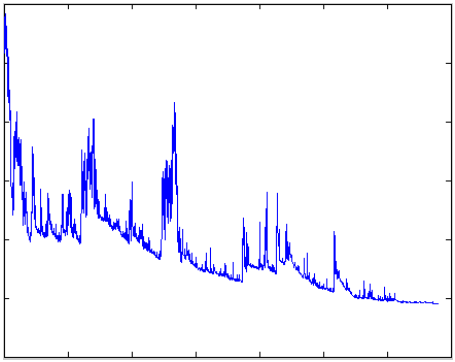


Figura 3.13: Fluctuaciones en la función de coste en función de las distintas actualizaciones de variables. Fuente: [67].

la función de activación de una neurona de la red. Por simpleza se analizará el caso en el que a cada neurona le corresponde una sola función de activación.

$$\delta_j^l \equiv \frac{\partial C_i}{\partial z_j^l}, \tag{3.30}$$

donde δ_j^l es el error correspondiente a la neurona j de la capa de la red neuronal l, z_j^l es la entrada a la función de activación correspondiente a la neurona j de la capa de la red neuronal l, y C_i es el coste correspondiente al ejemplo de entrenamiento i (se ha omitido que es una función de $\boldsymbol{W}$ y $\boldsymbol{b}$ por claridad).

Por tanto, con la expresión mostrada en la Ecuación 3.30 podemos conocer los errores correspondientes a todas las neuronas de la red: el índice l varía entre 1 y la cantidad de capas de la red L, y j entre 1 y la dimensión (cantidad de neuronas) de la capa l. Una vez definida la función de error, es conveniente reescribirla en términos de la salida de la neurona y de la función de activación (Ecuación 3.31).

$$\delta_j^l \equiv \frac{\partial C_i}{\partial z_j^l} = \frac{\partial C_i}{\partial a_j^l}\frac{\partial a_j^l}{\partial z_j^l} = \frac{\partial C_i}{\partial a_j^l} f_{act}^{'}(z_j^l), \tag{3.31}$$

El procedimiento para calcular la derivada parcial del coste de un ejemplo de entrenamiento respecto a un peso o bias dado se detalla a continuación [38].

1. El algoritmo de propagación hacia atrás se basa en propagar hacia las neuronas de las capas iniciales el error de las capas más cercanas a la salida de la red. El primer paso es, por tanto, calcular los errores en la capa de salida (capa con índice L, Ecuación 3.32).

$$\delta_j^L = \frac{\partial C_i}{\partial a_j^L} f_{act}^{'}(z_j^L), \tag{3.32}$$

   La Ecuación 3.32 muestra el error de una de neurona de la capa de salida. Para mostrar como funciona el algoritmo, es más conveniente utilizar la expresión matricial mostrada en la Ecuación 3.33, que muestra el error de todas las neuronas de la última capa.

$$\boldsymbol{\delta}^L = \nabla_{a^L} C_i \odot f_{act}^{'}(\boldsymbol{z}^L), \tag{3.33}$$

   donde $\nabla_{a^L} C_i$ contiene las derivadas parciales del la función de coste respecto a la salida de todas las neuronas de la última capa, $f_{act}^{'}(\boldsymbol{z}^L)$ es un vector cuyos componentes son $f_{act}^{'}(z_j^L)$, y $\odot$ representa el producto de Hadamardn (producto elemento a elemento). Estos errores se pueden calcular rápidamente.

Por un lado, conocemos cómo depende la función de coste correspondiente al ejemplo de entrenamiento C_i de la salida de la red neuronal. Por ejemplo, en el caso de la función de coste cuadrático,

$$C_i = \frac{\|\boldsymbol{y}_i - \boldsymbol{a}^L(\boldsymbol{x}_i)\|^2}{2} \Rightarrow \frac{\partial C_i}{\partial a_j^L} = a_j^L - y_{ij},$$

donde y_i es conocido (la salida deseada), y a_j^L se puede calcular a partir de los parámetros de la red y de la entrada $\boldsymbol{x}_i$.

Por el otro lado, como conocemos la función de activación, conocemos también su derivada. Por tanto podemos computar los componentes del vector $f_{act}^{'}(\boldsymbol{z}^L)$ calculando cada componente de $\boldsymbol{z}^L$ a partir de los parámetros de la red y de la entrada $\boldsymbol{x}_i$.

2. Propagación hacia atrás del error. Una vez se ha calculado el error en la capa de salida, se puede calcular el error de cualquier neurona de cualquier otra capa, computando hacia atrás los errores capa a capa. La ecuación 3.34 muestra cómo calcular el vector de errores de la capa l en función de los errores de la capa $l+1$.

$$\boldsymbol{\delta^l} = \nabla_{a^l} C_i \odot f_{act}^{'}(\boldsymbol{z}^l) = \left(\boldsymbol{W}^{l+1}\boldsymbol{\delta}^{l+1}\right) \odot f_{act}^{'}(\boldsymbol{z}^l), \tag{3.34}$$

donde $\boldsymbol{W}^{l+1}$ es la matriz de pesos correspondiente a la capa $l+1$.

La demostración de que $\nabla_{a^l} C_i = \boldsymbol{W}^{l+1}\boldsymbol{\delta}^{l+1}$ se muestra a en las Ecuaciones 3.35, 3.36, 3.37 y 3.38 en las que se han utilizado expresiones que se refieren a cada neurona de cada capa (expresiones no matriciales).

$$\delta_j^l \equiv \frac{\partial C_i}{\partial z_j^l} = \sum_k \frac{\partial C_i}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial z_j^l} = \sum_k \delta_k^{l+1} \frac{\partial z_k^{l+1}}{\partial z_j^l}, \tag{3.35}$$

donde z_k^{l+1} es la entrada a cada una de las funciones de activación de la capa $l+1$. En la ecuación 3.35 se ha aplicado la regla de la cadena a la definición de δ_j^l teniendo en cuenta que cada z_k^{l+1} depende de z_j^l. Para calcular la derivada $\frac{\partial z_k^{l+1}}{\partial z_j^l}$, hemos de escribir z_k^{l+1} en términos de ∂z_j^l (Ecuación 3.36).

$$z_k^{l+1} = \sum_j w_{kj}^{l+1} a_j^l + b_k^{l+1} = \sum_j w_{kj}^{l+1} f_{act}(z_j^l) + b_k^{l+1}, \tag{3.36}$$

donde w_{kj}^{l+1} es el peso que relaciona la salida a_j^l de la neurona j de la capa l con la neurona k de la capa $l+1$, y b_k^{l+1} el bias correspondiente a la neurona k de la capa $l+1$. En la Ecuación 3.36 se ha asumido que la función de propagación de las neuronas de la capa $l+1$ es la mostrada en la Ecuación 3.1.

Diferenciando la Ecuación 3.36 obtenemos la expresión de $\frac{\partial z_k^{l+1}}{\partial z_j^l}$:

$$\frac{\partial z_k^{l+1}}{\partial z_j^l} = w_{kj}^{l+1} f_{act}^{'}(z_j^l) \tag{3.37}$$

Sustituyendo el resultado de de la Ecuación 3.37 en la Ecuación 3.35 obtenemos una expresión elemento a elemento equivalente a la Ecuación 3.34 (la cual está escrita en forma matricial):

$$\delta_j^l = \sum_k w_{kj}^{l+1} \delta_k^{l+1} f_{act}^{'}(z_j^l) \tag{3.38}$$

3. Derivada parcial de la función de coste respecto a cualquier bias de la red. Como $\boldsymbol{z}^l = \boldsymbol{W}^l \boldsymbol{a}^{l-1} + \boldsymbol{b}^l$, y $\boldsymbol{\delta}^l \equiv \nabla_{z^l} C_i$, calcular las derivadas de la función de coste respecto a cualquier bias es inmediato una vez que hemos calculado los errores (Ecuación 3.39), simplemente hemos de aplicar la definición de $\boldsymbol{\delta}^l$.

$$\nabla_{b^l} C_i = \boldsymbol{\delta}^l \Leftrightarrow \frac{\partial C_i}{\partial b^l_j} = \delta^l_j \tag{3.39}$$

4. Derivada parcial de la función de coste respecto a cualquier peso de la red. Igual que con los bias, basta con aplicar la definición de $\boldsymbol{\delta}^l$, para conseguir la derivada de la función de coste respecto a cualquier peso (Ecuación 3.40).

$$\nabla_{w^l_k} C_i = a^{l-1}_k \boldsymbol{\delta}^l \Leftrightarrow \frac{\partial C_i}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j, \tag{3.40}$$

donde w^l_{jk} es el peso que relaciona la salida a^{l-1}_k de la neurona k de la capa $l-1$ con la neurona j de la capa l.

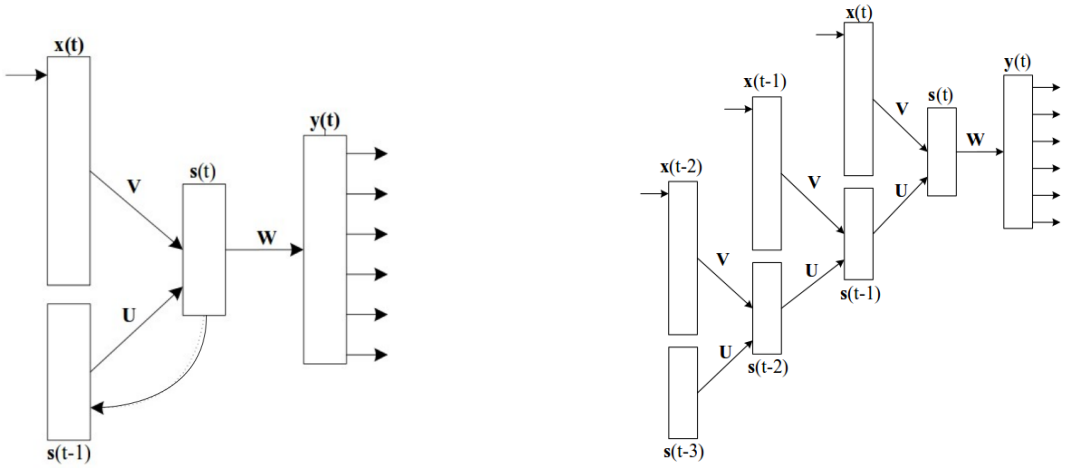
3.3.5. Propagación hacia atrás a lo largo del tiempo

El algoritmo de propagación hacia atrás permite calcular cómo afectan los pesos y bias de una red neuronal *feedforward* en la función de coste. En el caso de las redes neuronales recurrentes (RNN), es necesaria una extensión de este algoritmo, la cual se conoce como propagación hacia atrás a lo largo del tiempo o BPTT, por sus siglas en inglés. Esta extensión consiste en realizar un paso previo antes de aplicar el algoritmo de propagación hacia atrás estándar [19], [55].

Este paso previo se denomina *desplegar* la RNN. Al desplegar una RNN para un instante de tiempo t se genera una red neuronal *feedforward* equivalente para ese instante de tiempo. En la Figura 3.14 se muestra un ejemplo la estructura de una red neuronal *feedforward* equivalente a una RNN. Tal y como se analizó en la Sección 3.2.2, para calcular la salida de la red en el instante t es necesario saber (en este ejemplo) las salidas de la capa intermedia (recurrente) en el instante $t-1$. Igualmente, para conocer las salidas de la capa intermedia (recurrente) en el instante $t-1$, necesitamos conocer las entradas a la red en el instante $t-1$ y las salidas de la capa intermedia en el instante $t-2$. Y así hasta llegar al instante en el que comenzó la secuencia de entradas a la red.

Por tanto, si estamos entrenando la RNN mediante una secuencia de ejemplos de longitud T, necesitaremos desplegar la RNN T veces, para conseguir equivalentes *feedforward* para cada uno de esos instantes. Para los primeros ejemplos de entrenamiento la red equivalente no será demasiado grande, pero según nos vamos acercando al final de la secuencia las redes equivalentes contienen muchas capas. Para acelerar el entrenamiento se suelen limitar el número de instantes hacia atrás con los que preparar las redes equivalentes.

En cualquier caso, una vez que disponemos de las T redes neuronales *feedforward*, podemos aplicar el algoritmo de propagación hacia atrás estándar para calcular las derivadas necesarias en el algoritmo de descenso de gradiente estocástico, con la excepción de que hay que tener en cuenta que los pesos y bias en estas redes *feedforward* son compartidos, tenemos los mismos pesos y bias en distintos instantes de tiempo.



(a) RNN convencional.

(b) Equivalente *feedforward* para el instante t.

Figura 3.14: Representaciones de una RNN con una capa recurrente y una capa de salida no recurrente, y de su equivalente *feedforward*. $\boldsymbol{W}$, $\boldsymbol{V}$ y $\boldsymbol{U}$ son las matrices de pesos y bias de la RNN. $\boldsymbol{W}$ relaciona la capa intermedia con la de salida; $\boldsymbol{V}$ las entradas con la capa intermedia; y $\boldsymbol{U}$ las salidas de la capa intermedia en el instante anterior con la capa intermedia en el instante actual. Fuente: [19].

Capítulo 4

Sistemas completos de reconocimiento de voz

En el Capítulo 1 se introdujo el problema del reconocimiento de voz desde un punto de vista generativo. Con el objetivo de encontrar la secuencia de palabras más probable teniendo en cuenta una entrada acústica (Ecuación 1.1), se dividió el problema en dos: modelización acústica y del lenguaje (Ecuación 1.2). A lo largo del Capítulo 1 se describió también cómo obtener una secuencia de vectores que representen la señal acústica que queremos analizar (Sección 1.1), cómo descomponer una secuencia de palabras en secuencias de fonemas o trifonemas (Sección 1.2), y cómo estimar la probabilidad de una secuencia de palabras mediante un modelo de lenguaje (Sección 1.3). Así, la Ecuación 1.1 se puede desarrollar hasta llegar a la siguiente expresión (Ecuación 4.1).

$$\widehat{\boldsymbol{g}} = \underset{\boldsymbol{g}\in\mathcal{G}}{\arg\max}\ P(\boldsymbol{g} \mid \boldsymbol{x}) = \underset{\boldsymbol{g}\in\mathcal{G}}{\arg\max}\ P(\boldsymbol{x} \mid \boldsymbol{g})P(\boldsymbol{g}) = \begin{cases} \underset{\boldsymbol{g}\in\mathcal{G}}{\arg\max}\ P(\boldsymbol{x} \mid \boldsymbol{l}_{\boldsymbol{g}})P(\boldsymbol{g}), & \text{en el caso de fonemas,} \\ \\ \underset{\boldsymbol{g}\in\mathcal{G}}{\arg\max}\ P(\boldsymbol{x} \mid \boldsymbol{c}_{\boldsymbol{l}_{\boldsymbol{g}}})P(\boldsymbol{g}), & \text{en el caso de trifonemas,} \end{cases} \tag{4.1}$$

donde $\boldsymbol{x}$ representa una secuencia de vectores de parámetros correspondientes a una señal acústica, y $\boldsymbol{l}_{\boldsymbol{g}}$ y $\boldsymbol{c}_{\boldsymbol{l}_{\boldsymbol{g}}}$ son las secuencia de fonemas y trifonemas correspondientes a la secuencia de palabras $\boldsymbol{g}$.

Durante este capítulo analizaremos la estructura, entrenamiento y decodificación en los sistemas GMM-HMM y DNN-HMM. Finalmente, introduciremos parametrizaciones adicionales a los vectores MFFC para representar la señal acústica.

4.1. Modelos ocultos de Markov basados en mezclas de gaussianas

Los HMM son apropiados para resolver el problema de modelización acústica, ya que nuestras observaciones no son directamente las unidades fonológicas que queremos modelizar, sino, por ejemplo, los vectores de parámetros descritos en la Sección 1.1 o la variantes que introduciremos en la Sección 4.3. Inicialmente se aplicaba una cuantización de vectores [58] a los parámetros de vectores, para poder trabajar con HMM que modelasen secuencias de símbolos discretos. Después, estos símbolos fueron sustituidos por mezclas de gaussianas que modelan cada uno de los componentes de los vectores de parámetros.

En esta sección utilizaremos los HMM y las GMM para las probabilidades $P(\boldsymbol{x} \mid \boldsymbol{l})$ y $P(\boldsymbol{x} \mid \boldsymbol{c}_{\boldsymbol{l}})$. Como se ha explicado en la Sección 2.2, los HMM sirven para modelizar sistemas estocásticos sin memoria. Por tanto, asumiendo que las secuencias de vectores de parámetros correspondientes a un fonema o trifonema dado son generadas por un sistema estocástico sin memoria, podemos generar tantos HMM como unidades fonéticas se quieran representar. En el caso del castellano, serán unos 23 (24 si tenemos en cuenta el silencio) si estamos modelizando fonemas, y alrededor de 23^3 si modelamos trifonemas.

4.1.1. HMM de fonemas

Es típico utilizar HMM con topología de Bakis para representar cada fonema (incluyendo el silencio) [11]. Un HMM con esta topología está formado de tres estados: estado inicial, intermedio y final (Figura 4.1). En cada estado existe una probabilidad de transitar al siguiente estado y otra probabilidad de quedarse en el mismo estado. A cada estado se le asigna una GMM formada por 32 gaussianas para estimar la probabilidad de emisión de vectores de la misma dimensión que los vectores de parámetros utilizados (en el caso de la parametrización MFCC, la dimensión sería 13). Con esta estructura se pretenden representar las distintas partes del fonema: la transitoria, la estacionaria, y el final del fonema, aunque es verdad que el sonido correspondiente a ciertos fonemas no tiene esa estructura temporal.

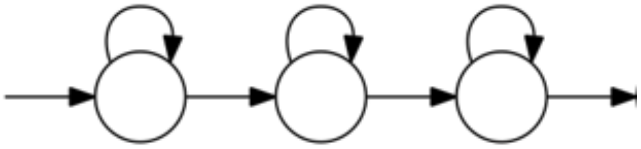


Figura 4.1: Estructura de un HMM con topología de Bakis.

Entrenamiento

Una vez fijada la estructura de todos los HMM, describiremos como entrenar éstos. Nuestro corpus de entrenamiento consta de audios y sus correspondientes transcripciones. Cada audio contiene en la mayoría de los casos frases de unas palabras. Por tanto no podemos aplicar directamente el entrenamiento mediante Viterbi analizado en la Sección 2.2.2, ya que la secuencia de vectores de parámetros extraída de cada audio no se va a corresponder a un solo fonema. Es decir, tenemos que tener en cuenta que la secuencia de vectores correspondiente a cada audio, está formada por subsecuencias correspondientes a cada fonema que ha sido pronunciado en ese audio. Así, por cada audio se pueden entrenar los HMM de los fonemas concatenándolos de acuerdo a la transcripción de ese audio y al léxico.

Por ejemplo, supongamos que tenemos un audio cuya transcripción es *hola*. De acuerdo a las reglas léxicas del castellano, el audio contendrá los siguientes fonemas: /o/ /l/ /a/, en este mismo orden. Con esta información, podemos concatenar los HMM correspondientes a esos fonemas [5] tal como se muestra en la Figura 4.2.

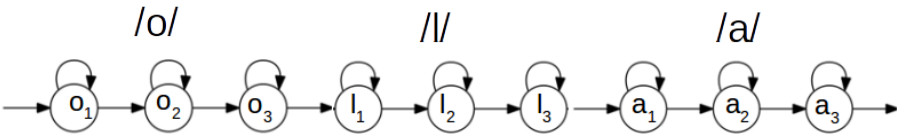


Figura 4.2: HMM correspondiente a la frase *hola*. Esta formado por los HMM de cada uno de los fonemas que se pronuncian al decir esta frase. f_i representa el estado i del HMM correspondiente al fonema f.

Una vez tenemos el HMM de la frase, podemos entrenarlo con el mismo procedimiento descrito en la Sección 2.2.2. Al llevar a cabo las reestimaciones en el entrenamiento, se ha de tener en cuenta que puede que distintos estados del HMM de la frase compartan sus parámetros, porque es posible que en una frase se pronuncie más de una vez un fonema. Repitiendo este procedimiento con todos los audios y transcripciones del corpus de entrenamiento, podemos entrenar las probabilidades de transición y los parámetros de las GMM de los estados de los HMM de todos los fonemas.

Decodificación

Con los HMM de los fonemas entrenados, podemos seguir desarrollando el problema del reconocimiento de voz. Como la secuencia de fonemas $\boldsymbol{l}$ se puede obtener a partir de distintas secuencias de estados $\boldsymbol{h}$ de los HMM entrenados, podemos reescribir la Ecuación 4.1 tal y como aparece a continuación (Ecuación 4.2) [15].

$$\widehat{\boldsymbol{g}} = \underset{\boldsymbol{g}\in\mathscr{G}}{\arg\max}\ P(\boldsymbol{g} \mid \boldsymbol{x}) = \underset{\boldsymbol{g}\in\mathscr{G}}{\arg\max}\ P(\boldsymbol{x} \mid \boldsymbol{l_g})P(\boldsymbol{g}) = \underset{\boldsymbol{g}\in\mathscr{G}}{\arg\max} \sum_{\boldsymbol{h}\in\mathscr{H}} P(\boldsymbol{x} \mid \boldsymbol{h})P(\boldsymbol{h} \mid \boldsymbol{l_g})P(\boldsymbol{g}), \tag{4.2}$$

donde $\mathscr{H}$ es el conjunto de todas las posibles secuencias de estados de los HMM de fonemas.

Nótese que en este punto todas las probabilidades de la Ecuación 4.2 son calculables. En primer lugar, la probabilidad de cualquier secuencia de palabras $\boldsymbol{g}$ se puede estimar a partir de un modelo de lenguaje. Después, esa misma secuencia puede ser descompuesta mediante reglas gramaticales en una única (en nuestro caso) secuencia de fonemas $\boldsymbol{l_g}$. $P(\boldsymbol{h} \mid \boldsymbol{l_g})$ indica la probabilidad de que la secuencia de fonemas $\boldsymbol{l_g}$ haya sido producida mediante la secuencia de estados $\boldsymbol{h}$. Esta probabilidad puede calcularse a partir de las probabilidades de transición de los HMM de los fonemas previamente entrenados (Ecuación 4.3).

$$P(\boldsymbol{h} \mid \boldsymbol{l_g}) = \begin{cases} \prod_{t=1}^{T-1} a_{ht,ht+1} & \text{en el caso de que la secuencia } \boldsymbol{l_g} \text{ haya podido ser generada a partir de la secuencia } \boldsymbol{h}, \\ 0 & \text{en el caso contrario,} \end{cases} \tag{4.3}$$

donde $a_{ht,ht+1}$ es la probabilidad de transición del estado h_t al estado h_{t+1}. Finalmente, el cálculo de $P(\boldsymbol{x} \mid \boldsymbol{h})$ puede llevarse a cabo con la información de las probabilidades de emisión de cada uno de los estados de los HMM de los fonemas (Ecuación 4.4).

$$P(\boldsymbol{x} \mid \boldsymbol{h}) = \prod_{t=1}^{T} P(x_t \mid h_t) = \prod_{t=1}^{T} b_{ht}(x_t), \tag{4.4}$$

donde $b_{h_t}(x_t)$ es la probabilidad de emisión del vector x_t en el estado h_t.

Aunque hemos mostrado como calcular la probabilidad de una secuencia de palabras y una secuencia de estados de HMM, existen infinitas secuencias de palabras y secuencias de estados de los HMM posibles. Además, aunque por cada secuencia de palabras no todas las secuencias de estados son posibles, existe un subconjunto infinito de secuencias de estados posibles para cada secuencia de palabras (debido a los bucles en los HMM de los fonemas). Por tanto, el método de fuerza bruta (calcular la probabilidad de todas las secuencias de palabras) no es válido para calcular la secuencia de palabras más probable.

En su lugar, se realiza una búsqueda de Viterbi sobre un HMM construido a partir del modelo de lenguaje, las reglas de pronunciación y los HMM de los fonemas. Como tenemos un HMM por fonema, podemos concatenarlos de acuerdo al léxico para formar HMMs de cada palabra en nuestro vocabulario. De la misma forma, una vez que tenemos los HMM de las palabras, podemos concatenarlos de acuerdo a un modelo de lenguaje y formar un HMM que englobe toda la información de nuestro reconocedor de voz. Realizando una decodificación mediante, por ejemplo, el algoritmo de Viterbi sobre dicho HMM, podríamos calcular la secuencia de palabras más probable dada una señal acústica. Es decir, podríamos encontrar la solución al problema planteado en la Ecuación 4.2. En la sección 5.1 especificaremos la implementación práctica de este gran HMM y del algoritmo de Viterbi sobre éste.

4.1.2. HMM de trifonemas

En este caso, el número de HMMs que se construirán será igual al número de trifonemas, y su topología será también de Bakis. Para entrenar estos HMM, se pueden deducir las secuencias de trifonemas correspondientes a las transcripciones del corpus a partir de la secuencia de fonemas extraída en el apartado

anterior, tal y como se mostró en la Sección 1.2.2.

Aún así, existen algunas diferencias entre los casos de fonemas y trifonemas. La cantidad de trifonemas en una lengua es mucho mayor al de fonemas (número de fonemas elevado a tres), luego el número de parámetros a entrenar aumenta en gran medida. Además existen trifonemas muy improbables, que van a ser vistos en reducidas o nulas ocasiones en durante el entrenamiento, /p_ k_ s/, por ejemplo. Para evitar los problemas que pudiesen darse por un entrenamiento insuficiente de los HMM de ciertos trifonemas, se agrupan algunos estados de HMMs de distintos trifonemas.

Habitualmente no se inicializan los HMM de los trifonemas desde cero. Se suelen tomar como base los modelos de fonemas descritos en el apartado anterior. Así, los parámetros de los HMM de cada trifonema se inicializan con los parámetros del HMM del fonema correspondiente. Por ejemplo, los parámetros iniciales del trifonema /k_ a_ s/ serán los parámetros del HMM del fonema /a/.

Una vez inicializados los HMM, se agrupan algunos estados de los HMM de los trifonemas correspondientes al mismo fonema. Se pueden utilizar tanto criterios fonéticos como puramente estadísticos para llevar acabo estos agrupamientos. Algunos algoritmos para realizar los agrupamientos están descritos en [68], [4], [45], [32].

Finalmente podemos entrenar los HMM de los trifonemas igual que en el caso de los fonemas: construyendo los HMM de las frases de transcripción y reestimando los parámetros de cada estado mediante un entrenamiento via Viterbi.

Para estimar la secuencia de palabras más probable correspondiente a una secuencia de vectores acústicos dada, podemos desarrollar la Ecuación 4.1 como en el caso de los fonemas. La única diferencia es que hemos descompuesto los fonemas en trifonemas. En la Ecuación 4.5 se muestra el desarrollo.

$$\widehat{\boldsymbol{g}} = \underset{\boldsymbol{g}\in\mathscr{G}}{\arg\max}\ P(\boldsymbol{g} \mid \boldsymbol{x}) = \underset{\boldsymbol{g}\in\mathscr{G}}{\arg\max}\ P(\boldsymbol{x} \mid \boldsymbol{c_{lg}})P(\boldsymbol{g}) = \underset{\boldsymbol{g}\in\mathscr{G}}{\arg\max} \sum_{\boldsymbol{h}\in\mathscr{H}} P(\boldsymbol{x} \mid \boldsymbol{h})P(\boldsymbol{h} \mid \boldsymbol{c_{lg}})P(\boldsymbol{g}) \tag{4.5}$$

El cálculo de $\widehat{\boldsymbol{g}}$ se puede llevar a cabo igual que en el caso de los fonemas.

4.2. Modelos híbridos entre modelos ocultos de Markov y redes neuronales

Aunque las mezclas de gaussianas sirven para realizar modelos acústicos de los distintos sonidos producidos al hablar, distintos experimentos y artículos destacan que mediante las redes neuronales se pueden conseguir reconocedores de voz de mayor precisión [22], [70], [62].

En los modelos híbridos, las redes neuronales sustituyen a las GMM. Las GMM permitían el cálculo de la probabilidad $P(\boldsymbol{x} \mid \boldsymbol{h})$, es decir, dada una secuencia de estado de HMM de fonemas o trifonemas, la probabilidad de que se haya producido la secuencia de observaciones de vectores acústicos $\boldsymbol{x}$. Las redes neuronales, sin embargo, se utilizan para estimar la probabilidad de cada estado de cada HMM de trifonemas (solo utilizaremos trifonemas en los modelos híbridos) en función de una entrada acústica, es decir, $P(\boldsymbol{h} \mid \boldsymbol{x})$. Para ello, la entrada a la red será un vector (o conjunto de vectores) de parámetros, y cada una de las neuronas de la capa salida representará la probabilidad de cada uno de los estados de los HMM de trifonemas [22]. Como la probabilidad $P(\boldsymbol{h} \mid \boldsymbol{x})$ no aparece explícitamente en la Ecuación 4.5 es necesario adaptar esta expresión para que tenga sentido en el contexto de modelos híbridos entre HMM y redes neuronales. Podemos aplicar el teorema de Bayes a la probabilidad $P(\boldsymbol{x} \mid \boldsymbol{h})$, tal y como se muestra a continuación (Ecuación 4.6).

$$\begin{aligned}\widehat{\boldsymbol{g}} &= \underset{\boldsymbol{g}\in\mathscr{G}}{\arg\max} \sum_{\boldsymbol{h}\in\mathscr{H}} P(\boldsymbol{x} \mid \boldsymbol{h})P(\boldsymbol{h} \mid \boldsymbol{c_{lg}})P(\boldsymbol{g}) = \underset{\boldsymbol{g}\in\mathscr{G}}{\arg\max} \sum_{\boldsymbol{h}\in\mathscr{H}} \frac{P(\boldsymbol{h} \mid \boldsymbol{x})P(\boldsymbol{x})}{P(\boldsymbol{h})} P(\boldsymbol{h} \mid \boldsymbol{c_{lg}})P(\boldsymbol{g}) = \\ &= \underset{\boldsymbol{g}\in\mathscr{G}}{\arg\max} \sum_{\boldsymbol{h}\in\mathscr{H}} \frac{P(\boldsymbol{h} \mid \boldsymbol{x})}{P(\boldsymbol{h})} P(\boldsymbol{h} \mid \boldsymbol{c_{lg}})P(\boldsymbol{g})\end{aligned} \tag{4.6}$$

En la Ecuación 4.6 podemos ignorar el factor $P(\boldsymbol{x})$ ya que afecta solo a los valores numéricos de las probabilidades, no a la secuencia óptima de palabras. Como hemos dicho, podemos calcular la probabilidad $P(\boldsymbol{h} \mid \boldsymbol{x})$ a partir de las salidas de la red neuronal (Ecuación 4.7).

$$P(\boldsymbol{h} \mid \boldsymbol{x}) = \prod_{t=1}^{T} P(h_t \mid x_t) = \prod_{t=1}^{T} a_{ht}(\boldsymbol{x}_t), \tag{4.7}$$

donde $a_{ht}(\boldsymbol{x}_t)$ es la salida de la neurona de la capa de salida correspondiente al estado h_t y al instante t. Es conveniente mencionar que en la Ecuación 4.7 se muestra que la salida de la red en el instante t no depende solo del vector acústico correspondiente a ese instante t. En su lugar, la salida a_{ht} depende de un conjunto de vectores acústicos $\boldsymbol{x}_t$, generalmente centrados en x_t. Es decir, la red neuronal estima las probabilidades de los estados del HMM basándose en el vector acústico correspondiente al instante t, y también en los vectores acústicos anteriores y posteriores a ese instante. Esto es conocido como *splice width*.

La probabilidad $P(\boldsymbol{h})$ de la Ecuación 4.6 se puede obtener a partir de las probabilidades a priori de cada estado h_t (Ecuación 4.8), las cuales se calculan a partir de las estadísticas acumuladas durante el entrenamiento de los HMM de los trifonemas [69].

$$P(\boldsymbol{h}) = \prod_{t=1}^{T} P(h_t) = \prod_{t=1}^{T} \frac{C(h_t)}{\sum_h C(h)}, \tag{4.8}$$

donde $C(h_t)$ es la cantidad de ocasiones que el estado h_t aparece en las alineaciones generadas mediante el entrenamiento de Viterbi de los HMM de trifonemas, y $\sum_h C(h)$ la cantidad total de vectores acústicos empleados en el entrenamiento.

4.2.1. Entrenamiento de las redes neuronales en el contexto de reconocimiento de voz

En la Sección 3.3 analizamos como entrenar redes neuronales mediante ejemplos de entrenamiento formados por parejas $(\boldsymbol{x}_i, \boldsymbol{y}_i)$, donde $\boldsymbol{x}_i$ es cada una de las entradas a la red del corpus de entrenamiento, y $\boldsymbol{y}_i$ la salida deseada para la entrada $\boldsymbol{x}_i$. Nuestro corpus de entrenamiento está formado por un conjunto de audios y sus correspondientes transcripciones. Por tanto, no podemos entrenar las redes neuronales directamente con este corpus, ya que la entrada a una red no serán todos los vectores acústicos extraídos del audio de una frase, sino el vector correspondiente a un instante dado más unos pocos vectores anteriores y posteriores (para que la red tenga información del contexto). Además las salidas de la red son las probabilidades de cada estado de los HMM de trifonemas, no palabras.

En consecuencia, es necesario un alineamiento de los vectores acústicos que representan los audios para poder entrenar las redes neuronales. Esta alineaciones se suelen conseguir mediante el entrenamiento de Viterbi de un sistema GMM-HMM basado en trifonemas. Así, el entrenamiento de las redes neuronales se realizará de forma posterior al entrenamiento del sistema GMM-HMM basado en trifonemas (el cual se ha entrenado a partir de un sistema GMM-HMM basado en fonemas). Una vez que tenemos los audios de entrenamiento alineados, podemos entrenar las redes neuronales tal y como se describió en la Sección 3.3. Cada ejemplo de entrenamiento constará de un conjunto de vectores acústicos centrados en el instante t y del vector de salida deseado $\boldsymbol{y}_t$, el cual se calcula fácilmente conociendo el alineamiento del audio: todos los componentes de $\boldsymbol{y}_t$ serán 0 excepto el correspondiente al estado *alineado*, que será 1 [63]. Con estado *alineado* nos referimos al estado de los HMMs que corresponde al instante t, según la alineación obtenida mediante el entrenamiento de Viterbi.

El hecho de tener dividido el corpus de entrenamiento en frases nos permite, por otro lado, definir funciones de coste más complejas que las funciones de coste cuadrático y *cross-entropy* (Ecuaciones 3.20 y 3.21, respectivamente). En este trabajo experimentaremos con la función MMI, la cual se basa en el criterio MMI (del inglés, *Maximum Mutual Information*) [69], [63], [48]. La función de coste MMI se muestra en la Ecuación 4.9.

$$C_{MMI} = -\sum_{m=1}^{M} \ln \frac{P(\boldsymbol{x}_m|\,\boldsymbol{h}_m)\,P(\boldsymbol{g}_m)}{\sum_{\boldsymbol{g}\in\mathcal{G}} P(\boldsymbol{x}_m|\,\boldsymbol{h}_{\boldsymbol{g}})\,P(\boldsymbol{g})}, \tag{4.9}$$

donde M es el número de frases del corpus, $\boldsymbol{x}_m$ la secuencia de vectores acústicos del ejemplo de entrenamiento m, $\boldsymbol{h}_m$ su alineación en estados de los HMM de trifonemas, $\boldsymbol{g}_m$ su transcripción, y $\mathcal{G}$ el conjunto de todas las posibles secuencias de palabras.

Las probabilidades $P(\boldsymbol{x}_m|\,\boldsymbol{h}_m)$ y $P(\boldsymbol{x}_m|\,\boldsymbol{h}_{\boldsymbol{g}})$ de la Ecuación 4.9 se pueden calcular aplicando el teorema de Bayes (Ecuaciones 4.7 y 4.8). Las probabilidades $P(\boldsymbol{g}_m)$ y $P(\boldsymbol{g})$ calculan mediante un modelo de lenguaje (Sección 1.3). En teoría, la suma que se muestra en el denominador ha de ser realizada sobre todas las posibles secuencias de palabras (y todas las posibles alineaciones en cada secuencia, aunque esta suma no se muestra). Obviamente esa suma no es computacionalmente viable. En su lugar se realiza sobre unas pocas secuencias de palabras que se obtienen aplicando el algoritmo *N-best* al ejemplo de entrenamiento m. El algoritmo *N-best* es una generalización del algoritmo de Viterbi [6]. Mientras que el algoritmo de Viterbi encuentra la secuencia de estados con mayor probabilidad teniendo en cuenta una secuencia de observaciones (y su probabilidad), el algoritmo *N-best* consigue las N mejores secuencias y sus probabilidades.

4.3. Parametrizaciones de los vectores acústicos

En la Sección 1.1 se describieron los vectores formados por los coeficientes MFCC. Se analizó la razón del uso de este tipo de vectores en la tarea de reconocimiento de voz, así como el procedimiento para la extracción de los coeficientes MFCC a partir de una señal acústica. A continuación introduciremos variaciones y extensiones de los coeficientes MFCC.

Normalización de la media y varianza cepstral

La normalización de la media y varianza cepstral (CMVN, por sus siglas en inglés) es una normalización que permite aumentar la robustez ante el ruido de los vectores MFCC [61]. La normalización CMVN normaliza cada componente de los vectores MFCC para que cada uno de estos presente media 0 y varianza 1 por frase o por hablante (Ecuación 4.10).

$$\widehat{x}_t(i) = \frac{x_t(i) - \mu_t(i)}{\sigma_t(i)}, \tag{4.10}$$

donde $x_t(i)$ es el componente i del vector acústico correspondiente al instante t, y $\widehat{x}_t(i)$ su versión normalizada. $\mu_t(i)$ y $\sigma_t(i)$ son la media y la varianza del componente i calculada sobre todos los componentes correspondientes a la frase o hablante (Ecuaciones 4.11 y 4.12).

$$\mu_t(i) = \frac{1}{N}\sum_n x_n(i) \tag{4.11}$$

$$\sigma_t(i) = \frac{1}{N}\sum_n \big(x_n(i) - \mu_t(i)\big)^2, \tag{4.12}$$

donde x_n es cada uno de los N vectores correspondientes a la misma frase o hablante.

Coeficientes Δ + $\Delta\Delta$

Ya que el acto de hablar es dinámico, puede ser beneficioso computar parámetros que describan la evolución de los coeficientes MFCC a lo largo del tiempo.

Se conocen como coeficientes Delta los asociados a la diferencia de valor de los MFCC entre ventana y ventana (coeficientes de velocidad); y como Delta Delta los asociados al cambio de los coeficientes Delta entre ventana y ventana (coeficientes de aceleración). Se obtendrán tantos coeficientes Delta y Delta-Delta como MFCC, y éstos serán añadidos al vector MFCC. Por tanto, si nuestro vectores MFCC son de dimensión

13, los vectores MFCC + Δ serán de dimensión 26, y los MFCC + Δ + Δ Δ de dimensión 39. En las Ecuaciones 4.13 y 4.14 se muestra cómo calcular los coeficientes Δ, y Δ Δ, respectivamente [28].

$$\delta_t(i) = \frac{\sum_{t=1}^{T} t\,(\,x_t(i) - x_{-t}(i)\,)}{T\sum_{t=1}^{T} t^2}, \tag{4.13}$$

donde $\delta_t(i)$ es el componente i del vector Δ correspondiente al instante t, ($x_t(i)$ es el componente i del vector MFCC correspondiente al instante t, y T el número de ventanas hacia delante y atrás que utilizamos para calcular estos coeficientes. En nuestro caso T será igual a 2. Los coeficientes Δ Δ se calculan igual que los Δ, pero a partir de los coeficientes Δ en vez de los MFCC.

$$\delta\delta_t(i) = \frac{\sum_{t=1}^{T} t\,(\,\delta_t(i) - \delta_{-t}(i)\,)}{T\sum_{t=1}^{T} t^2}, \tag{4.14}$$

donde $\delta\delta_t(i)$ es el componente i del vector Δ Δ correspondiente al instante t.

Transformación LDA

En el ámbito de reconocimiento de voz, la transformación LDA (del inglés, *Linear Discriminant Analysis*) es utilizada como una técnica de preprocesamiento de vectores acústicos. Una transformación LDA transforma un vector de entrada a un espacio de menor dimensión donde la separación entre las clases de los vectores sea mejor [53]. Una ejemplo gráfico y simple se muestra en la Figura 4.3.

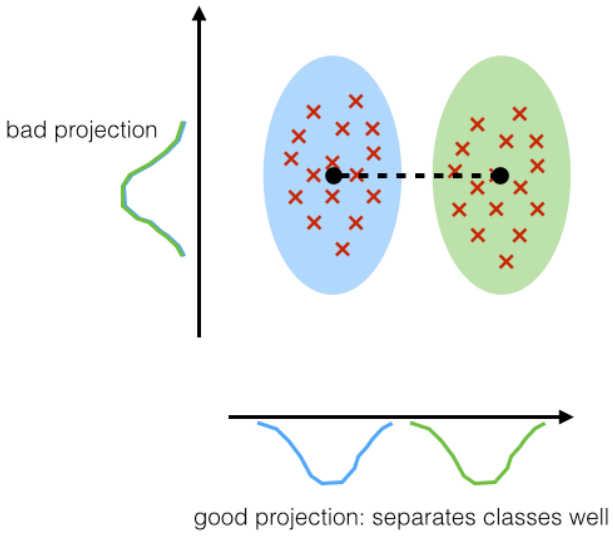


Figura 4.3: Ejemplo de dos proyecciones posibles de un conjunto de vectores dividido en dos clases. Ambas proyecciones reducen la dimensión de cada vector de dos a uno. Una de las proyecciones (la horizontal) permite clasificar fácilmente los vectores, mientras que la otra (la vertical) hace que la clasificación sea más difícil. Fuente: [53].

Por tanto, solo podemos aplicar esta transformación a vectores previamente clasificados (en nuestro caso, en estados de trifonemas). Esta clasificación se suele hacer mediante un sistema GMM-HMM. Además, como la transformación reduce la dimensión, se suele aplicar a una pequeña secuencia de vectores, para que el vector resultante contenga información del contexto. Por ejemplo, si queremos obtener el *vector LDA* correspondiente a un vector MFCC, podemos transformar nueve vectores MFCC (los cuatro previos, el central y los cuatro posteriores) [54]. Es habitual que el vector LDA resultante tenga dimensión 40.

Los detalles de la transformación LDA se especifican en [51].

Transformación MLLT

El objetivo de la transformación MLLT (del inglés, *Maximum Likelihood Linear Transform*) es el mismo que el de la transformación LDA: decorrelar un vector. La MLLT ortogonaliza un vector para que sus

parámetros puedan ser modelados más precisamente mediante gaussianas con una matriz de covarianza diagonal [54]. La transformación MLLT no reduce la dimensión de los vectores a los que se aplica. En la práctica, aplicaremos la MLLT a vectores LDA.

Los detalles de la transformación MLLT se especifican en [50].

Transformación fMLLR

La transformación fMLLR (del inglés, *feature-space Maximum Likelihood Linear Regression*) es una transformación que mantiene la dimensión de los vectores y permite adaptar vectores acústicos al hablante. La idea general de esta transformación es, como todos hablamos de forma distinta, generar unos vectores en los que las diferencias de los distintos hablantes sean lo menos influyentes posible. Para ello la fMLLR adapta la media y varianza de los vectores correspondientes a un mismo hablante y así normaliza la variabilidad general de los vectores de todos los hablantes.

Los detalles de la transformación fMLLR se especifican en [14].

Capítulo 5

Kaldi

Todos los experimentos llevados a cabo en este trabajo han sido realizados mediante la herramienta Kaldi. Kaldi es una herramienta de uso libre orientada a la investigación en el ámbito del reconocimiento de voz [46], [12]. Permite el desarrollo de reconocedores de voz basados tanto en los *clásicos* sistemas GMM-HMM como en los modelos híbridos DNN-HMM.

Kaldi está compuesto por un conjunto de librerías escritas en C++ creadas por los desarrolladores de esta herramienta, y por otras dos librerías externas (BLAS/LAPACK y OpenFST). En las librerías propias de Kaldi se encuentran todos los algoritmos y clases correspondientes a los HMM, GMM, DNN, parametrización de la señal acústica y también parte de los algoritmos correspondientes a los transductores de estados finitos con pesos (WFST, por sus siglas en inglés), los cuales introduciremos en la Sección 5.1. Kaldi utiliza la librería BLAS/LAPACK para realizar cálculos relacionados con la álgebra lineal y la librería OpenFST para implementar los mencionados WFST.

El acceso a las funcionalidades ofrecidas por estas librerías se provee mediante ejecutables escritos en C++, los cuales son generalmente llamados mediante scripts (de Bash o Python, en general).

Es importante destacar que Kaldi soporta el uso de tarjetas gráficas o GPUs (por sus siglas en inglés) en algunas de sus rutinas. Las GPUs están compuestas por miles de núcleos que son capaces de procesar instrucciones del tipo SIMD (*Single Instruction Multiple Data*). Es decir, pueden realizar una gran cantidad de cálculos en paralelo, siempre que la operación sea la misma. Tal y como se describe en [49], el entrenamiento de las redes neuronales puede llevarse a cabo mediante múltiples procesos en paralelo, los cuales se pueden realizar en las GPUs. Así se puede mitigar una de las mayores desventajas de las redes neuronales: la lentitud en el entrenamiento.

A continuación se introducirán los WFST y se describirán brevemente algunas características de las redes neuronales particulares de la herramienta Kaldi.

5.1. Transductores de estados finitos con pesos

En el Capítulo 4 desarrollamos la Ecuación 1.2 hasta llegar a varias expresiones en las que, en teoría, todos los términos son calculables: Ecuación 4.2 para el caso de los sistemas GMM-HMM basados en fonemas, Ecuación 4.5 para el caso de los sistemas GMM-HMM basados en trifonemas, y Ecuación 4.6 para el caso de los sistemas híbridos DNN-HMM. En ese capítulo también se introdujo la idea de realizar una búsqueda de Viterbi sobre un HMM que englobase todas las probabilidades de esas tres ecuaciones. A continuación describiremos cómo implementar ese HMM y cómo realizar la búsqueda sobre él.

Kaldi genera este HMM mediante un tipo de autómata conocido como WFST (*Weighted Finite State Transducer*). Un WFST es una máquina de estados finitos con una cinta de entrada y una de salida. Un WFST sirve para traducir una secuencia de símbolos del alfabeto de entrada en una (o varias) secuencia de símbolos del alfabeto de salida. Además, proporciona un peso a la secuencia de salida. En nuestro caso

utilizaremos WFSTs deterministas y estocásticos. Un WFST es determinista si convierte una secuencia de entrada en una sola secuencia de salida. Además, si es estocástico la suma de los pesos correspondientes a las transiciones desde un mismo estado sumará 1 [21].

Para entender el cometido de los WFST en el contexto de reconocimiento de voz, en la Figura 5.1 se muestran tres ejemplos muy simples de WFSTs representando un modelo de lenguaje, unas reglas de pronunciación, y el HMM de un fonema, respectivamente.

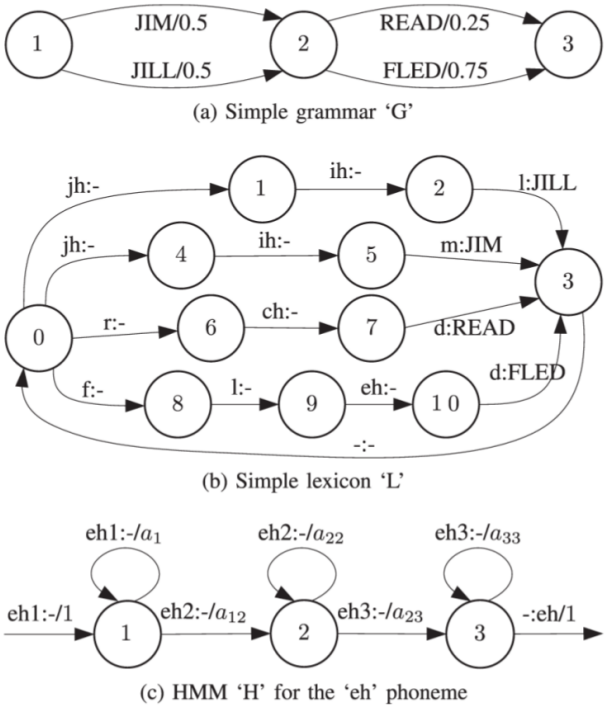


Figura 5.1: Tres ejemplos de WFST. La notación utilizada es la siguiente: (entrada) : (salida) / (peso). Fuente: [3].

Comencemos analizando el transductor G de la Figura 5.1. Este WFST representa un modelo de lenguaje. El alfabeto de entrada de G es el conjunto de palabras que forma un vocabulario. Aunque no se muestre explícitamente en el diagrama, la salida de G es exactamente la misma secuencia de entrada, más el peso correspondiente a ésta. Con una cierta estructura, se puede representar cualquier modelo de lenguaje basado en n-gramas con este WFST [33]. En modelo de lenguaje de nuestro ejemplo nos permite conocer la probabilidad, por ejemplo, del bigrama "JIM READ". Si introducimos esta cadena a G, la salida será: "JIM READ / 0.125". Si interpretamos los peso como probabilidades, la probabilidad de dicho bigrama es 0.125.

Analicemos ahora el WFST del léxico, L. L traduce secuencias de fonemas a secuencias de palabras. En el caso del castellano (y en el del ejemplo de la Figura 5.1 también), el WFST es simplemente un FST. Como solo contemplamos una pronunciación por palabra, cualquier cadena de salida tendrá probabilidad 1, y por tanto no es necesario especificar el peso. Por ejemplo, ante la entrada "/r/ /eh/ /d/", la salida de L será: "READ". Para finalizar con el léxico, merece la pena mencionar que en principio este WFST no tiene por qué cumplir la condición de ser determinista, debido a las palabras homófonas: una misma de secuencia de fonemas puede corresponder a más de una palabra. Este problema se soluciona mediante la inserción de símbolos de desambiguación [15].

El último WFST mostrado en la Figura 5.1 es el WFST H, el cual contiene la información de la estructura y de las probabilidades de transición del HMM de un fonema. H convierte secuencias de estados de HMMs en secuencias de fonemas, y además devuelve también la probabilidad (el peso) de que esa secuencia de estados haya producido la secuencia de fonemas. Por ejemplo, ante la entrada "eh1 eh2 eh2 eh3", H devolverá "eh / $(1 \cdot a_{12} \cdot a_{22} \cdot a_{23} \cdot 1)$", donde a_{ij} es la probabilidad de transición del estado del HMM i al estado j.

Con estos tres WFSTs y con las probabilidades de emisión de los estados de los HMM de fonemas tenemos suficiente para representar un sistema GMM-HMM que modela fonemas. Para introducir el concepto de trifonemas se necesita un WFST adicional, C, que traduce secuencias de trifonemas en secuencias de fonemas (algo que ya analizamos cómo hacer *a mano* en la Sección 1.2.2). Un ejemplo se muestra en la Figura 5.2.

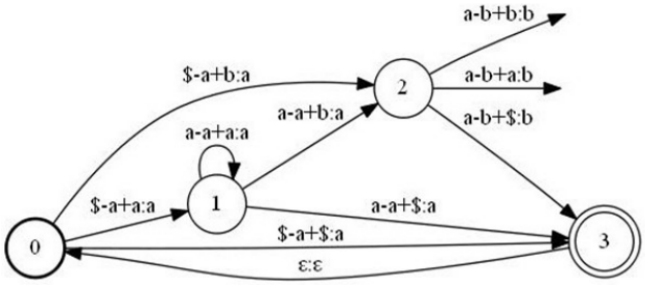


Figura 5.2: Ejemplo de un WFST C. La notación utilizada para la entrada (los trifonemas) es: (fonema anterior) - (fonema central) + (fonema posterior). Fuente: [36].

C, al igual que L (en el caso del castellano), no devuelve ningún peso, ya que a una secuencia de trifonemas le corresponde una única (o ninguna si la secuencia de trifonemas no es coherente) secuencia de fonemas. Por ejemplo ante la secuencia de entrada "/$-a+a/ /a-a+b/ /a-b+$/", C devolverá "/a/ /a/ /b/".

Con los cuatro WFSTs G, L, C, y H, podemos representar un sistema GMM-HMM o DNN-HMM basado en trifonemas. Aún así, hemos de *componer* estos WFST para lograr una traducción completa de secuencias de estados de HMM de trifonemas a palabras. La composición es una de las operaciones básicas de los WFST. Al concatenar ($\circ$) dos WFSTs ($A \circ B$) se obtiene un WFST (C) cuyo alfabeto de entrada es el mismo al del primer WFST (A) y cuyo alfabeto de salida es el mismo al del segundo WFST (B) [15], [33], [21]. Además, ante una secuencia de entrada $\boldsymbol{s}$ la salida de C es la igual a la salida B si a la entrada B se introduce la salida de A ante $\boldsymbol{s}$ (esta propiedad es similar a la composición de funciones). Por tanto, si componemos los cuatro WFSTs descritos, lograremos un WFST que traduzca de estados de los HMM a palabras, el cual denominaremos como $HCLG$ (Ecuación 5.1).

$$HCLG = H \circ C \circ L \circ G \tag{5.1}$$

En el caso de sistemas basados en fonemas solo habría que componer H, L y G.

Una vez que tenemos el WFST completo $HCLG$, podemos discutir como estimar la secuencia de palabras más probable para una secuencia de vectores acústicos dada. Para una secuencia de estados de HMM de trifonemas dada, $HCLG$ nos devuelve la secuencia de palabras correspondiente y su probabilidad. Mediante GMMs o DNNs (dependiendo del reconocedor que estemos utilizando) podemos calcular la probabilidad que cada componente de la secuencia de vectores acústicos haya sido emitido por cada estado de los HMM de trifonemas (Ecuación 4.4 o 4.7). Con estas probabilidades podemos realizar una búsqueda de Viterbi en el WFST $HCLG$. En el contexto de WFST, para realizar esta búsqueda se genera un WFST U que contiene todas las probabilidades estimadas por las GMM o DNN (Figura 5.3). Después U se compone a $HCLG$ para forma el grafo de búsqueda $S = U \circ HCLG$, sobre el cual se realiza finalmente la decodificación mediante el algoritmo de Viterbi [47], [24].

A la hora de realizar la decodificación hay que tener en cuenta que el grafo S es un grafo muy grande, suficientemente grande como para que el algoritmo de Viterbi tal y como se describió en la Sección 2.2.2 no sea práctico. En cada iteración de dicho algoritmo, por cada estado del HMM, se fijaba como estado anterior aquél que presentase la probabilidad máxima teniendo en cuenta la probabilidad de haber estado en ese estado en el instante anterior y haber transitado al estado actual. Por tanto en cada iteración se han de realizar del orden de N^2 cálculos, si N es el número de estados del HMM. El grafo S contiene todos los HMMs de todos los trifonemas, por lo que suele estar compuesto de al menos miles de nodos. Para evitar tener que hacer tantos cálculos, en cada iteración no se analizan todos los posibles anteriores, solo los que

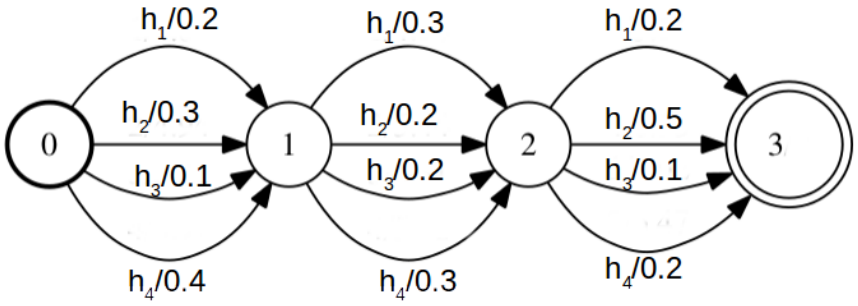


Figura 5.3: Ejemplo de un WFST U, para un HMM con 4 estados (h_1, h_2, h_3 y h_4).

mayor probabilidad acumulada presenten (por ejemplo, los 12 estados con mayor probabilidad) [2]. Esta versión del algoritmo de Viterbi es conocida como *beam search*.

5.2. Redes neuronales en Kaldi

A día de hoy, Kaldi ofrece tres *paquetes* de código distintos para el desarrollo de redes neuronales: *nnet1*, *nnet2* y *nnet3* [7]. En este trabajo se experimentarán con redes correspondientes a los paquetes *nnet2* y *nnet3*.

Las características comunes entre *nnet2* y *nnet3* más representativas son las siguientes:

- No se utiliza ningún tipo de pre-entrenamiento de las redes neuronales. Los parámetros iniciales se fijan de forma aleatoria.
- De forma previa al primer bloque de funciones de transferencia de la primera capa de las redes, se aplica una transformación lineal al conjunto de vectores de entrada. Esta transformación es fija (sus parámetros no se actualizan durante el entrenamiento), y aunque es similar a una transformación del tipo LDA, no reduce la dimensión de los vectores de entrada [9]. El objetivo de esta transformación es decorrelar los vectores de entrada, para facilitar la separación entre clases de éstos, y aumentar así el rendimiento de la red.
- La constante de velocidad de aprendizaje o *learning rate* decae a lo largo de las épocas, para favorecer la convergencia del descenso de gradiente estocástico [57]. Este decaimiento es exponencial.
- También con el objetivo de favorecer la convergencia del descenso estocástico de gradiente, se limita el cambio máximo de los pesos y bias de la red por *mini-batch*.
- Para obtener el modelo final de una red tras un entrenamiento, no se elige como modelo final el modelo correspondiente a la última época. En su lugar se hace una media pesada de los últimos modelos (de los últimos diez, por ejemplo). Los pesos se determinan optimizando la función de coste en un subconjunto de muestras aleatoriamente escogidas del conjunto de datos de entrenamiento [70].

Existen dos diferencias fundamentales entre *nnet2* y *nnet3*. La primera es los tipos de redes que pueden ser implementadas. El paquete *nnet2* está fundamentalmente diseñado para utilizar redes *feedforward* relativamente simples, como por ejemplo, redes basadas en funciones de activación como la tangente hiperbólica, la sigmoide o la función *pnorm*. En el paquete *nnet3* se pueden implementar, además de las redes correspondientes a *nnet2*, redes neuronales con estructuras más complejas como redes TDNN o LSTM.

En segundo lugar, el entrenamiento de las redes neuronales es diferente en los casos de *nnet2* y *nnet3*. En *nnet2* las redes se entrenan mediante la función de coste *cross-entropy* tal y como se describió en la Sección 3.3. En *nnet3* el entrenamiento se efectúa por secuencias, mediante la función de coste MMI, como analizamos en la Sección 4.2.1.

Capítulo 6

Experimentación

En este capítulo se realizarán distintos experimentos para analizar los diferentes sistemas de reconocimiento de voz descritos durante este trabajo. Comenzaremos explicando como evaluar un reconocedor de voz. Después describiremos los corpus utilizados para llevar a cabo los experimentos, con los cuales concluiremos el capítulo.

6.1. Evaluación del rendimiento de un reconocedor de voz

Supongamos que hemos desarrollado un sistema reconocedor de voz y queremos cuantificar su rendimiento.

Si nuestro sistema se tratase de un reconocedor de palabras aisladas (su salida sería una sola palabra en cualquier caso), podríamos cuantificar su rendimiento haciendo uso de una medida denominada WER (del inglés, *Word Error Rate*). Para calcular el WER se pasan al reconocedor una cantidad N de señales acústicas, en las que se ha pronunciado una palabra conocida por audio. Después se comparan cada una de las salidas del reconocedor para conseguir el porcentaje de palabras falladas (Ecuación 6.1).

$$WER\_palabras\ aisladas = 100\left(1 - \frac{C}{N}\right), \tag{6.1}$$

donde C es el número de palabras correctas.

Esta medida es válida para reconocedores de palabras aisladas, pero no para los reconocedores desarrollados en este trabajo. En nuestro caso, todos los sistemas desarrollados tienen como salida la secuencia de palabras que estiman que es más probable teniendo en cuenta una señal acústica. Una extensión directa del WER anteriormente definido sería la medida SER (del inglés, *Sentence Error Rate*). Como en este caso vamos a pasar al reconocedor señales acústicas correspondientes a frases, el valor del SER es el porcentaje de frases correctas.

Aún así, en este trabajo no utilizaremos el SER, ya que en muchos casos puede no ser representativo del rendimiento de un reconocedor de voz. Por ejemplo, supongamos que tenemos un reconocedor que de media falla una palabra por cada dos, y que las frases que le pasamos al reconocedor en el test tienen unas cuarenta palabras. El SER en ese caso sería muy alto (rondaría el 100 %), aunque estamos reconociendo correctamente la mitad de palabras. El problema es que la medida SER es altamente dependiente de la longitud de las frases, cuanto más largas, mayor será el SER, independientemente de la proporción de palabras acertadas.

En consecuencia, utilizaremos otra extensión del WER definido para el reconocimiento de palabras aisladas, que medirá la proporción de palabras erróneas, a la que también llamaremos WER. El WER para el reconocimiento de habla *continua* se define como se muestra a continuación (Ecuación 6.2) [34].

$$WER\_habla\ continua = 100\ \frac{palabras\ erróneas}{N} = 100\ \frac{S + D + I}{N}, \tag{6.2}$$

donde N es el número total de palabras en las transcripciones del test, S el número de sustituciones necesarias para convertir la salida del reconocedor en la transcripción correcta, D el número palabras que han de ser borradas para conseguir la transcripción correcta a partir de la salida del reconocedor, y I el número palabras que han de ser insertadas.

El WER es una medida del rendimiento de los reconocedores de voz mucho más fiel a la realidad que el SER. Buena muestra de ello es que el WER es utilizado en la inmensa mayoría de artículos de investigación y libros relacionados con el reconocimiento de voz.

6.2. Descripción de los corpus utilizados

Para desarrollar un reconocedor de voz son necesarios corpus para el desarrollo tanto para modelos acústicos como para modelos de lenguaje. En este trabajo se han utilizado los corpus mostrados en la Tabla 6.1 para el desarrollo modelos acústicos.

Corpus	Duración total (horas)	Número de frases	Número de palabras	Media de palabras por frase
Albayzin	14,8	15 600	152 646	9,785
Dihana-diálogos	5,4	6 279	47 515	7,567
Dihana-frases	4,8	3 600	42 106	11,696
Corpus UAM	5,7	2 080	64 737	31,124
TC-STAR	143,2	108 047	834 033	7,719

Tabla 6.1: Corpus utilizados para el entrenamiento y test de modelos acústicos.

Albayzin está formado por grabaciones a personas leyendo frases, Dihana es habla telefónica, el corpus UAM lo forman entrevistas de radio y los audios TC-STAR son grabaciones de sesiones parlamentarias. Todos los audios están filtrados a 8 kHz.

Para desarrollar modelos acústicos es necesario un corpus de entrenamiento, pero también de test, para poder examinar y analizar los modelos creados, y poder compararlos. En este trabajo se ha dividido cada uno de los cinco corpus en entrenamiento (80 % aproximadamente) y test (20 % restante). Es muy importante dividir el corpus en entrenamiento y test, en vez de utilizar como test un subcorpus del entrenamiento. En dicho caso no podríamos evaluar la capacidad para generalizar de los modelos creados, solo veríamos si son capaces de *aprender* los datos de entrenamiento.

Para construir los modelos de lenguaje empleados en el test, se utilizaron por una parte, las transcripciones de los audios de los corpus mostrados en la Tabla 6.1. Por otra parte, también se hicieron experimentos con un corpus extraído del periódico *El País*. Un resumen de los corpus utilizados para modelos de lenguaje se muestra en la Tabla 6.2.

Corpus	Vocabulario	Número de palabras en total
Albayzin (entrenamiento)	2 920	122 233
Dihana-diálogos (entrenamiento)	755	47 515
Dihana-frases (entrenamiento)	1 064	33 556
Corpus UAM (entrenamiento)	6 807	52 064
TC-STAR (entrenamiento)	24 484	591 871
Todas las transcripciones de entrenamiento	27 650	837 672
El País	91 148	3 284 620
El País + todas las transcripciones de entrenamiento	97 415	4 122 292

Tabla 6.2: Corpus utilizados para el entrenamiento de modelos de lenguaje.

Para construir los modelos de lenguaje a partir de los corpus de audios, no se han utilizado todas las

transcripciones, solo las de entrenamiento (el mencionado 80 %). Si se hubiesen incluido las transcripciones correspondientes al test los resultados no habrían sido realistas, ya que el modelo de lenguaje daría una probabilidad muy alta a las secuencias de palabras que se van a intentar reconocer, y se conseguirían las transcripciones correctas incluso si los modelos acústicos no han funcionado del todo bien.

6.3. Descripción de los modelos acústicos entrenados

Procedamos ahora a describir las especificaciones de los distintos modelos acústicos presentados a lo largo del trabajo para posteriormente evaluar su rendimiento y compararlos.

6.3.1. GMM-HMM basado en fonemas

Comenzaremos desarrollando un sistema GMM-HMM basado en fonemas. Se creará un HMM con topología de Bakis (Sección 4.1.1) por fonema. La probabilidad de emisión en cada estado del sistema será modelada mediante una mezcla de 32 gaussianas. Para desarrollar este sistema se realizará un entrenamiento de Viterbi tal y como se describió en las Secciones 2.2.2 y 4.1.1, en las que las observaciones serán vectores $\Delta + \Delta\,\Delta$ (Sección 4.3). Estos vectores se aplicarán a los vectores MFCC extraídos a partir de las señales acústicas, los cuales se normalizarán mediante una CMVN por hablante. Podemos realizar la normalización por hablante porque conocemos el hablante de todas las frases destinadas al entrenamiento de modelos acústicos.

6.3.2. GMM-HMM basado en trifonemas

Parametrización $\Delta + \Delta\,\Delta$

Una vez tenemos desarrollado el sistema GMM-HMM basado en fonemas, podemos inicializar un sistema GMM-HMM basado en trifonemas, basándonos en la alineación de los audios (a nivel de estado de HMM) de entrenamiento creada con el sistema de fonemas, tal y como se explicó en la Sección 4.5. La parametrización de la acústica es la misma que en el caso anterior, vectores $\Delta + \Delta\,\Delta$.

Parametrización LDA + MLLT

Podemos seguir desarrollando el sistema de trifonemas aplicando las transformaciones LDA (primero) y MLLT (después) a los vectores MFCC, los cuales han sido extraídos para computar los vectores $\Delta + \Delta\,\Delta$ de los casos anteriores. Como estas transformaciones requieren conocer las clases de los vectores a los que se van a aplicar, nos basaremos en una alineación generada con el anterior sistema de trifonemas.

Para conseguir cada vector LDA + MLLT, se aplicará la transformación LDA a grupos de 9 vectores MFCC, estando estos grupos formados por un vector MFCC central y cuatro vectores hacia cada lado. En total, la dimensión de entrada a la transformación será $13 \times 9 = 117$. El resultado será un vector LDA de dimensión 40, al que se le aplicará una transformación MLLT, para lograr el vector LDA + MLLT, también de dimensión 40.

Parametrización fMLLR

El último sistema GMM-HMM que se desarrollará estará basado también en trifonemas, y los vectores acústicos que se utilizarán serán los anteriormente calculados vectores LDA + MLLT, pero con adaptación al hablante. Para ello se utilizará la transformación fMLLR. Los vectores resultantes serán de la misma dimensión que los vectores de entrada, 40.

6.3.3. Redes neuronales

Los ejemplos de entrenamiento de las redes neuronales se consiguen alineando los audios de entrenamiento con el último sistema GMM-HMM. Construiremos redes neuronales correspondientes a los paquetes de Kaldi *nnet2* y *nnet3*.

Redes desarrolladas en el entorno *nnet2*

Se entrenarán redes neuronales basadas en las funciones de activación tangente hiperbólica y *pnorm*. Los detalles de la estructura y entrenamiento de estas dos redes se detallan en la Tabla 6.3.

	Tangente hiperbólica	*pnorm* ($p = 2$)
Cantidad de vectores acústicos que toma la red como entrada	5 vectores de contexto a la izquierda + vector central + 5 vectores de contexto a la derecha.	
Número de capas intermedias	2.	
Dimensión de las capas intermedias	375	1000 → 200. Se normaliza la salida de las neuronas mediante la normalización descrita en la Sección 3.2.1.
Capa de salida	*softmax*.	
Dimensión de la capa de salida	Número de estados de los HMM de trifonemas. Depende de la cantidad de datos de entrenamiento con los que se hayan entrenado los modelos GMM-HMM de trifonemas.	
Número de épocas de entrenamiento	30.	
Velocidad de aprendizaje o *learning rate*	En las primeras 20 épocas decae exponencialmente desde 0.02 hasta 0.004. En las últimas 10 se mantiene constante en 0.004.	
Modelo final	Media pesada de los parámetros de los parámetros de la red en las últimas 10 épocas.	

Tabla 6.3: Especificaciones de las redes *nnet2* con las que se van a experimentar.

Aunque no se menciona en la Tabla 6.3 antes de la primera capa se realiza una transformación LDA que no reduce dimensión y la función de coste a minimizar durante el entrenamiento es la función *cross-entropy*, tal y como se explicó en la Sección 3.3. También cabe mencionar que como la dimensión de entrada de cada capa *pnorm* es 1000 y la de salida 200, cada neurona dará una salida tomando como entrada la salida de 5 funciones de propagación únicas (la salida de cada función de propagación será utilizada por una sola neurona).

Probaremos ambas redes utilizando parametrizaciones tanto MFCC como fMLLR. La cantidad de vectores de entrada a la red se mantendrá constante, pero la dimensión de entrada a la red cambiará. Esta será $(5+1+5)\times 13 = 143$ en el caso de los MFCC, y $(5+1+5)\times 40 = 440$ en el caso de los vectores fMLLR.

Redes desarrolladas en el entorno *nnet3*

En el entorno de trabajo *nnet3* trabajaremos tanto con redes TDNN como LSTM. Experimentaremos con dos redes de cada tipo. Con las redes TDNN probaremos las funciones de activación ReLU y *pnorm*. Experimentaremos con las LSTM convencionales y también con las LSTM bidireccionales. Las especificaciones más notorias de las redes TDNN se muestran en la Tabla 6.4.

Para determinar la arquitectura de las TDNN no es suficiente con saber el número de capas y la dimensión de cada capa. Hay que especificar los instantes de tiempo en los que cada capa toma como entrada la salida de la capa anterior. Utilizando la misma notación usada en [43] y en la Figura 3.9, la estructura temporal de las dos TDNN utilizadas se resume mediante los siguientes índices temporales: {-1,0,1}, {-1,0,1,2}, {-3,0,3}, {-3,0,3}, {-3,0,3}, {-6,-3,0}, {0}. El primer conjunto pertenece a la primera capa, el segundo a la segunda, ..., y el séptimo a la de salida. La interpretación de cada uno de los conjuntos de índices se entiende de la siguiente forma. La última capa da una salida correspondiente al instante t, luego el último índice es {0}. Para ello, ésta toma como entrada las salidas de la capa anterior correspondientes a los instantes $t-6$, $t-3$ y t, de ahí los índices {-6,-3,0}. Cada subcapa de la penúltima capa tomará como entrada la salida de

<table>
<tr><th></th><th>TDNN ReLU</th><th>TDNN pnorm ($p = 2$)</th></tr>
<tr><td>Cantidad de vectores acústicos que toma la red como entrada</td><td colspan="2">17 vectores de contexto a la izquierda + vector central + 12 vectores de contexto a la derecha.</td></tr>
<tr><td>Número de capas intermedias</td><td colspan="2">7.</td></tr>
<tr><td rowspan="2">Dimensión de las subcapas intermedias</td><td>600</td><td>2500 → 250.</td></tr>
<tr><td colspan="2">Se normaliza la salida de las neuronas mediante la normalización descrita en la Sección 3.2.1.</td></tr>
<tr><td>Capa de salida</td><td colspan="2">No hay una capa softmax. Esta no es necesaria durante el entrenamiento, debido al uso de la función de coste MMI, en la cual ya se realiza una normalización [8].</td></tr>
<tr><td>Dimensión de la capa de salida</td><td colspan="2">Número de estados de los HMM de trifonemas. Depende de la cantidad de datos de entrenamiento con los que se hayan entrenado los modelos GMM-HMM de trifonemas.</td></tr>
<tr><td>Número de épocas de entrenamiento</td><td colspan="2">4.</td></tr>
<tr><td>Velocidad de aprendizaje o learning rate</td><td colspan="2">Decae exponencialmente desde 0.001 hasta 0.0001.</td></tr>
<tr><td>Modelo final</td><td colspan="2">Media pesada de los parámetros de los parámetros de la red en las últimas épocas.</td></tr>
</table>

Tabla 6.4: Especificaciones de las redes TDNN (*nnet3*) con las que se van a experimentar.

la capa anterior en los instantes relativos a su salida $t'-3$, t' y $t'+3$ (índices {-3,0,3}). Por ejemplo, si vamos a computar la salida de la penúltima capa correspondiente al instante $t-6$, tomaremos como entrada las salidas de la capa anterior correspondiente a los instantes $t-6-3=t-9$, $t-6$ y $t-6+3=t-3$. Siguiendo este razonamiento se puede deducir toda la estructura temporal de las TDNN utilizadas.

Por otra parte, en la Tabla 6.5 se muestran las características que tendrán las LSTM y BiLSTM utilizadas en el trabajo.

Aunque no se menciona en la Tabla 6.3 antes de la primera capa se realiza una transformación LDA que no reduce dimensión y la función de coste a minimizar durante el entrenamiento es la función MMI, tal y como se explicó en la Sección 5.2. En el caso de las redes *nnet3* utilizaremos siempre vectores de parametrización MFCC.

	LSTM	BiLSTM
Cantidad de vectores acústicos que toma la red como entrada	2 vectores de contexto a la izquierda + vector central + 2 vectores de contexto a la derecha.	
Número de capas intermedias	3.	
Dimensión de las subcapas intermedias	1024 celdas	2048 celdas. 1024 celdas en cada sentido.
Capa de salida	No hay una capa *softmax*. Esta no es necesaria durante el entrenamiento, debido al uso de la función de coste MMI, en la cual ya se realiza una normalización [8].	
Dimensión de la capa de salida	Número de estados de los HMM de trifonemas. Depende de la cantidad de datos de entrenamiento con los que se hayan entrenado los modelos GMM-HMM de trifonemas.	
Número de épocas de entrenamiento	8.	
Velocidad de aprendizaje o *learning rate*	Decae exponencialmente desde 0.001 hasta 0.0001.	
Modelo final	Media pesada de los parámetros de los parámetros de la red en las últimas épocas.	

Tabla 6.5: Especificaciones de las redes LSTM (*nnet3*) con las que se van a experimentar.

6.4. Comparación entre modelos acústicos

Una vez descritos los modelos acústicos específicos que se utilizarán, podemos compararlos. Las condiciones de este experimento serán las siguientes:

- **Corpus para los modelos acústicos.** 80 % los audios de todos los corpus y sus transcripciones.
- **Corpus para el modelo de lenguaje.** Transcripciones correspondientes a los audios de entrenamiento.
- **Características del modelo de lenguaje.** Modelo de lenguaje basado en trigramas, con *smoothing* o suavizado Witten Bell [44].
- **Corpus para el test.** 20 % restante de los audios y transcripciones.

Con estas condiciones se han entrenado y evaluado los modelos acústicos descritos en la Sección 6.3. Los resultados (WER) se muestran a continuación en la Tabla 6.6.

En la Tabla 6.6 se puede observar que todos los sistemas GMM-HMM basados en trifonemas han funcionado mejor que el sistema GMM-HMM basado en fonemas. Además, en el caso de los GMM-HMM basados en trifonemas, aplicar las transformaciones LDA y MLLT a los vectores MFCC ha resultado en una mejora del WER, debido a la mayor separación entre clases que ofrecen estas transformaciones y también al mayor contexto que codifican en cada vector. Mientras que en la parametrización $\Delta + \Delta\,\Delta$ se tienen en cuenta 5 vectores MFCC para computar cada vector, en el caso de los vectores LDA y MLLT se tienen en cuenta 11. La adaptación al hablante también ha logrado mejorar los resultados.

Por otro lado, el rendimiento de todos los modelos híbridos ha sido mayor al de los sistemas GMM-HMM. Dentro de las redes correspondientes al *nnet2*, la red *pnorm* ha funcionado mejor que la red basada en la tangente hiperbólica. Además, en ambos casos la adaptación al hablante ha contribuido a mejorar el WER, siendo mayor la mejora en el caso de la red basada en la tangente hiperbólica.

Finalmente, las redes *nnet3* han permitido mejorar el WER aún más. Posiblemente, el utilizar tanto entrenamiento mediante la función de coste MMI como redes neuronales con mayor capacidad de modelizar características temporales han contribuido a esta mejora en el rendimiento. Las TDNN han funcionado

Modelo acústico		Parametrización	WER
GMM-HMM	fonemas	Δ + Δ Δ	39.25
	trifonemas	Δ + Δ Δ	23.28
		LDA + MLLT	22.26
		fMLLR	19.59
DNN-HMM	*tanh*	MFCC	17.35
		fMLLR	16.31
	pnorm	MFCC	16.03
		fMLLR	15.48
	TDNN (*ReLU*)	MFCC	13.91
	TDNN (*pnorm*)	MFCC	13.91
	LSTM	MFCC	12.70
	BiLSTM	MFCC	11.70

Tabla 6.6: WER para distintos modelos acústicos entrenados con el 80 % de los corpus disponibles. El test se ha llevado acabo sobre el 20 % restante utilizando un modelo de lenguaje construido a partir de las transcripciones de los audios de entrenamiento.

igual independientemente de la función de activación. Las redes más apropiadas para el reconocimiento de voz en estas condiciones han resultado ser las dos redes LSTM, dada su habilidad para modelizar largas secuencias de vectores acústicos. Las LSTM bidireccionales han funcionado mejor que las unidireccionales, demostrando así que el *contexto hacia la derecha* también contiene información relevante en el procesamiento del habla.

6.4.1. Adaptación al hablante

Como se puede observar en la Tabla 6.6, utilizar vectores acústicos que tienen en cuenta información sobre el hablante puede ser beneficioso. Pero también existen casos en los que adaptación puede no mejorar los resultados. En la Tabla 6.7 se muestran los WER obtenidos para las redes *nnet2* en las siguientes condiciones.

- **Corpus para los modelos acústicos.** 80 % los audios y transcripciones del corpus Dihana-diálogos.
- **Corpus para el modelo de lenguaje.** Transcripciones correspondientes a los audios de entrenamiento.
- **Características del modelo de lenguaje.** Modelo de lenguaje basado en trigramas, con *smoothing* o suavizado Witten Bell [44].
- **Corpus para el test.** 20 % restante de los audios y transcripciones de Dihana-diálogos.

Modelo acústico		Parametrización	WER
DNN-HMM	*tanh*	MFCC	9.08
		fMLLR	8.66
	pnorm	MFCC	8.45
		fMLLR	8.80

Tabla 6.7: WER para distintos modelos acústicos entrenados con el corpus Dihana-diálogos.

En la Tabla 6.7 podemos observar que en el caso de la red *pnorm*, los resultados con adaptación al hablante son peores, y la mejora relativa en el caso de la red basada en la tangente hiperbólica es menor que en el experimento mostrado en la Tabla 6.6, la mejora desciende de un 6.00 % a un 4.63 %. La razón es que en el corpus de Dihana hay menos frases por hablante que si tenemos en cuenta todos los corpus: 22.32 contra 57.89.

6.5. Influencia del modelo de lenguaje en el reconocimiento de voz

En esta sección realizaremos dos experimentos para analizar ciertos aspectos de los modelos de lenguaje. En primer lugar, probaremos distintos modelos de lenguaje en un experimento en el que los corpus de entrenamiento y test no varían. Después, utilizaremos distintos pesos del modelo de lenguaje sobre un mismo modelo acústico.

6.5.1. Experimento con distintos modelos de lenguaje

Las condiciones de este experimento son las siguientes.

- **Corpus para los modelos acústicos.** 80 % los audios de todos los corpus.
- **Corpus para el modelo de lenguaje.** Se utilizarán tres corpus para construir distintos modelos de lenguaje: las transcripciones de los audios de entrenamiento de Dihana-diálogos, las transcripciones de los audios de entrenamiento de todos los corpus, y las transcripciones de los audios de entrenamiento de todos los corpus más *El País*.
- **Características del modelo de lenguaje.** Modelo de lenguaje basado en trigramas, con *smoothing* o suavizado Witten Bell [44].
- **Corpus para el test.** 20 % de los audios y transcripciones del corpus Dihana-diálogos, los correspondientes al test.

Para llevar a cabo la comparación entre modelos acústicos, se entrenaron sistemas GMM-HMM basados en fonemas y trifonemas, y sistemas híbridos con redes neuronales correspondientes al paquete de Kaldi *nnet2*. Los resultados obtenidos se muestran en la Tabla 6.8.

Modelo acústico		Parametrización	WER		
			LM 1	LM 2	LM 3
GMM-HMM	fonemas	$\Delta + \Delta\Delta$	19.69	25.12	29.28
	trifonemas	$\Delta + \Delta\Delta$	12.37	13.84	15.29
		LDA + MLLT	12.13	12.95	14.60
		fMLLR	11.36	12.52	13.77
DNN-HMM	*tanh*	MFCC	9.67	10.20	11.46
		fMLLR	9.21	9.88	11.24
	pnorm	MFCC	8.85	9.45	10.44
		fMLLR	8.98	9.47	10.48

Tabla 6.8: WER para tres modelos de lenguaje distintos. LM 1 indica el correspondiente a las transcripciones de entrenamiento de Dihana-diálogos, LM 2 el correspondiente a las transcripciones de entrenamiento de todos los corpus, y LM 3 el correspondiente a las transcripciones de entrenamiento de todos los corpus más *El País*.

Como se puede apreciar en los resultados de la Tabla 6.8, cuanto más general es el modelo de lenguaje, más difícil le resulta al sistema reconocer correctamente las secuencias de palabras. La razón es que un modelo de lenguaje entrenado mediante secuencias de palabras similares a las transcripciones del test atribuye una mayor probabilidad priori a esas secuencias, mientras un modelo de lenguaje general entrenado a partir de, entre otros corpus, del periódico *El País* no dará demasiada probabilidad a secuencias de palabras del test, que son propias de habla telefónica (este es el caso del corpus Dihana-diálogos).

6.5.2. Peso del modelo de lenguaje

A continuación se analizará como cambia el rendimiento de un reconocedor de voz en función del peso del modelo de lenguaje. Se llevarán a cabo dos experimentos, en las condiciones mostradas en la Tabla 6.9.

	Experimento con Albayzin	Experimento con el corpus UAM
Corpus para entrenar los modelos acústicos	80 % de los audios y transcripciones de todos los corpus disponibles.	
Modelo acústico	Sistema GMM-HMM basado en trifonemas, con vectores acústicos a los que se les ha aplicado las transformaciones LDA + MLLT.	
Corpus para realizar el test	20 % de los audios y transcripciones del corpus Albayzin	20 % de los audios y transcripciones del corpus UAM
Corpus para el modelo de lenguaje	Transcripciones de los audios de entrenamiento del corpus Albayzin	Las transcripciones de los audios de entrenamiento de todos los corpus

Tabla 6.9: Condiciones en los que se van a llevar a cabo los experimentos con el peso del modelo de lenguaje.

Los resultados de ambos experimentos se muestran en la Figura 6.1. En esa figura podemos ver que en los dos experimentos la relación del WER con el peso de modelo de lenguaje es muy distinta. Lo más representativo es la posición del mínimo de WER. En el caso de Albayzin el mínimo se da con un peso del modelo de lenguaje mayor que en el caso del corpus UAM. La razón es la siguiente: en el caso de Albayzin el modelo de lenguaje se adapta muy bien a las transcripciones del test, ya que en este corpus una gran cantidad de frases de entrenamiento y test son iguales. En el caso del UAM el modelo de lenguaje no es tan adecuado. Esta razón también explica que en el caso de Albayzin el aumentar mucho el peso de lenguaje no resulta demasiado perjudicial para el WER, mientras que si en el caso del corpus UAM nos alejamos del peso de modelo de lenguaje óptimo, el WER aumenta notablemente.

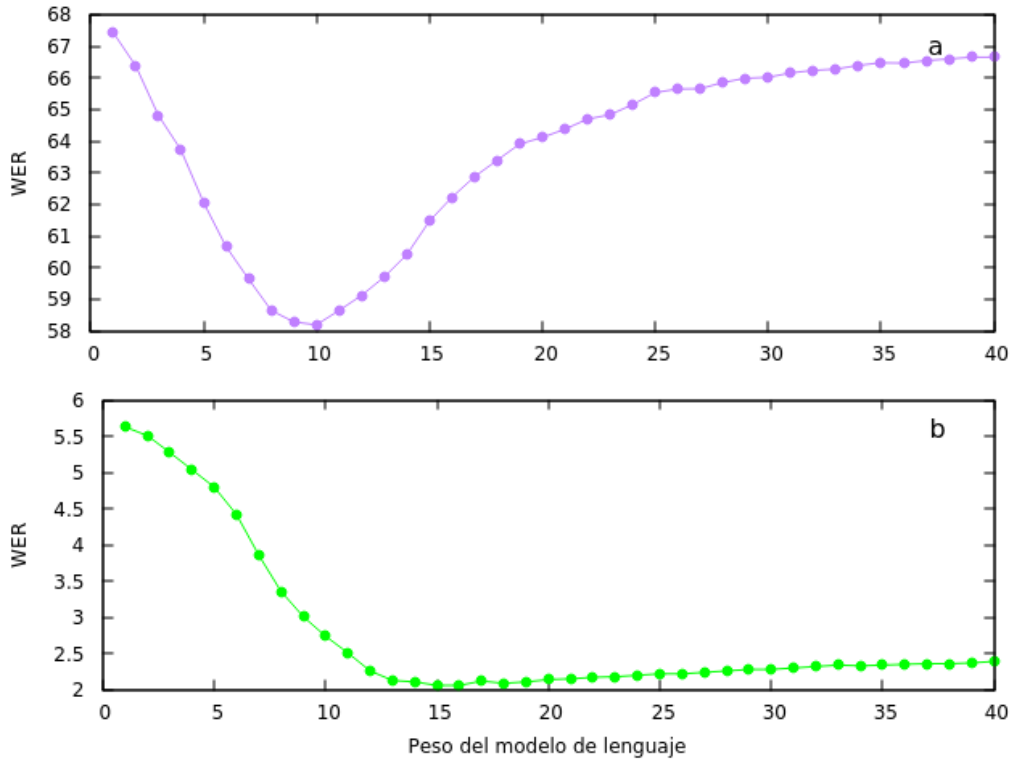


Figura 6.1: Evolución del WER en función del peso del modelo de lenguaje realizando un test sobre el corpus UAM (a) y Albayzin (b), con dos modelos de lenguaje distintos.

6.6. Análisis del entrenamiento y rendimiento de las redes neuronales

En la Sección 3.3 analizamos como entrenar las redes neuronales, es decir, como minimizar una función de coste dada para un conjunto de ejemplos de entrenamiento. Para ello, explicamos que son necesarias un cierto número de épocas de entrenamiento. Aún así, no establecimos ningún límite a este número. En esta sección realizaremos un conjunto de experimentos para demostrar que si nuestro objetivo es aumentar el rendimiento de una red neuronal, existe un límite en el número de épocas a entrenar la red. Analizaremos como localizar ese límite y como varía en función de la estructura de la red y de las condiciones del entrenamiento.

Comencemos analizando la curva de entrenamiento en un caso en el que el sobre-entrenamiento (entrenar durante un número excesivo de épocas) no es un problema excesivo. Las condiciones de este experimento son las siguientes.

- **Corpus para el entrenamiento.** 80 % los audios y transcripciones del corpus Dihana-diálogos.
- **Características de la red neuronal.** Se entrenará una red neuronal correspondiente al paquete *nnet2* basada en la función de activación tangente hiperbólica, con dos capas intermedias y 375 neuronas por capa.
- **Características del entrenamiento.** 45 épocas. En las primeras 30 el *learning rate* decrece de forma exponencial desde 0.02 hasta 0.004. En las últimas 15 se mantiene constante en 0.004.

En la Figura 6.2 se muestran las curvas de entrenamiento para esta red. Con curvas de red nos referimos a la evolución de la función de coste en los ejemplos de entrenamiento y en un pequeño conjunto de muestras independientes a lo largo de las épocas. El conjunto de estos ejemplos independientes no utilizados para entrenar la red se denominan datos de validación. Indican como de buena es la red neuronal que se está entrenando generalizando. Es decir, como de buena es dando salidas correctas a entradas que nunca ha *visto*.

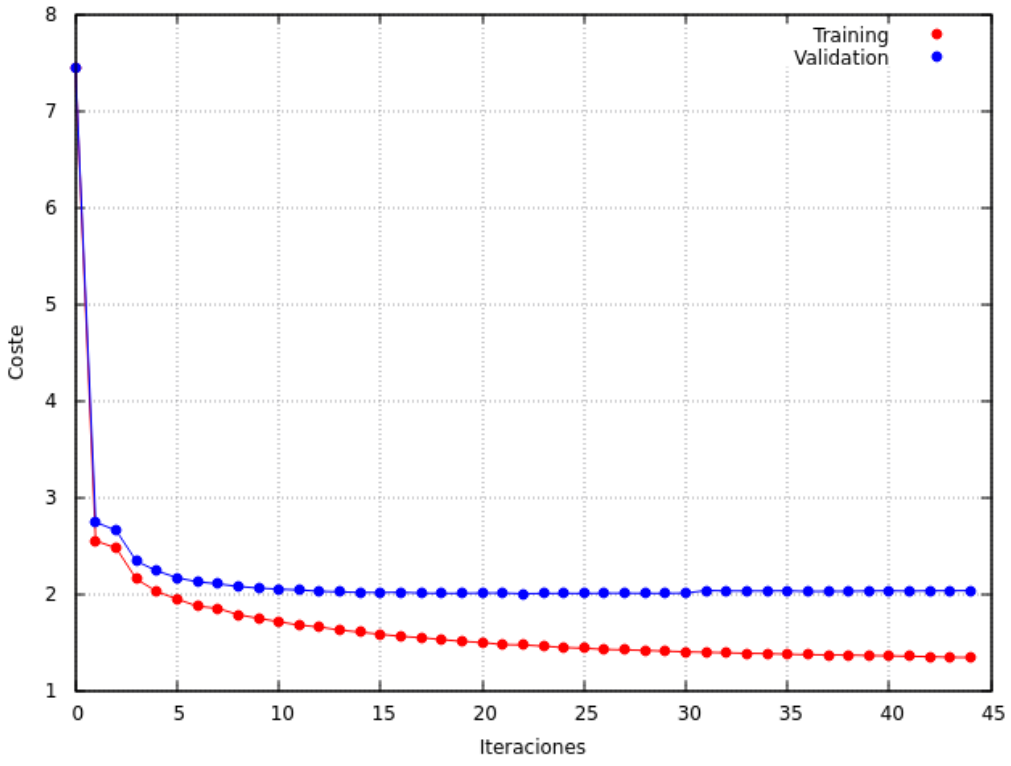


Figura 6.2: Curvas de entrenamiento en un caso en el que no hay sobre-entrenamiento claro.

En la Figura 6.2 se muestra que la función de coste en los ejemplos de entrenamiento no para de crecer si aumentamos el número de épocas. Aún así, en los ejemplos de validación la función de coste ha alcanzado un mínimo del que no baja en las últimas 30 épocas aproximadamente. Entrenar 45 épocas no parece un problema en lo que al rendimiento de la red se refiere, pero no tiene sentido entrenar tantas épocas si el rendimiento de la red no aumenta en las últimas 30.

A continuación se va a mostrar un ejemplo de sobre-entrenamiento claro. En este experimento vamos a mantener las condiciones del anterior, excepto que la red será más grande: contará con 3 capas (en vez de 2), y 1000 neuronas por capa (en vez de 375). En la Figura 6.3 se muestra una comparación entre los dos casos.

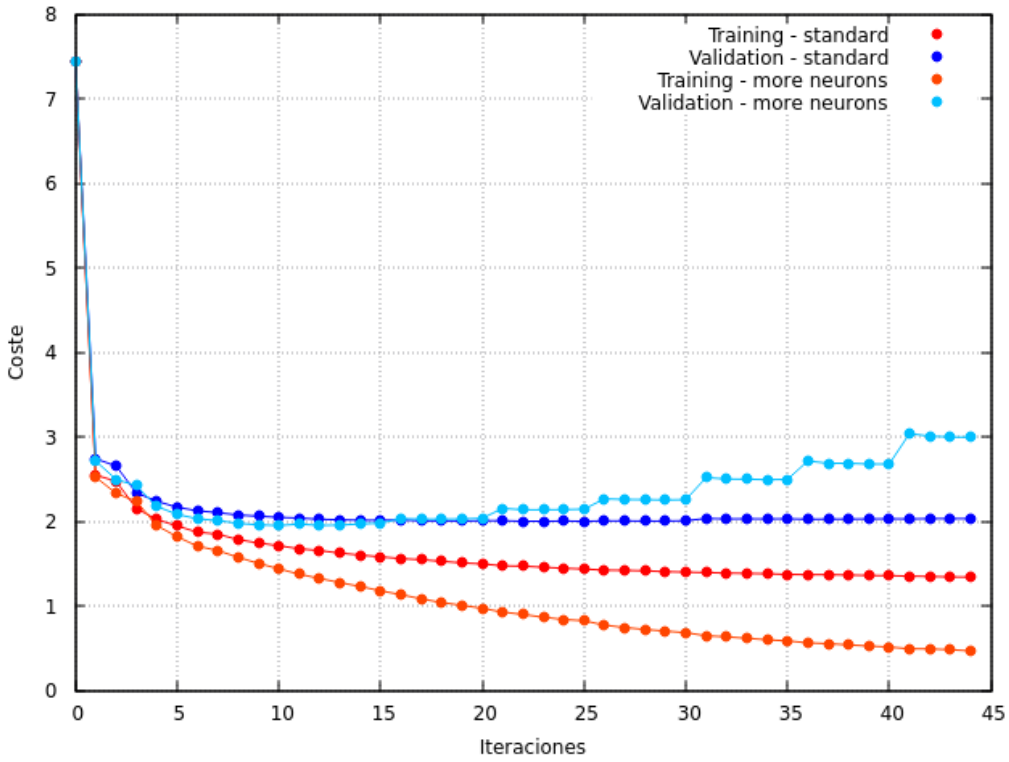


Figura 6.3: Curvas de entrenamiento en dos redes que se diferencian por el número total de neuronas.

Se pueden sacar varias conclusiones de la Figura 6.3. La primera es que disminuir indefinidamente la función de coste en el entrenamiento no implica que el rendimiento de la red vaya a ser mejor. Un claro contraejemplo son las curvas de entrenamiento de la red con más neuronas. La segunda es que las redes con un mayor número de neuronas tienden a sobre-entrenarse antes. La tercera es que la función de coste en el entrenamiento disminuye más rápidamente si la red cuenta con más neuronas. Finalmente, aunque la red con más neuronas tiende sobre-entrenarse antes, es interesante que el mínimo de la función de coste en los ejemplos de validación es ligeramente menor en la red con más neuronas. Es decir, si utilizamos el número de épocas óptimo en ambas redes, la red con más neuronas rinde mejor.

Estos cuatro conceptos están muy relacionados con el número de parámetros libres que contamos en cada caso para modelar algo tan complejo como el habla humana. Si utilizamos demasiados parámetros para modelar un proceso de forma estadística (mediante una red neuronal, por ejemplo), es más fácil que los parámetros aprendan tan sólo características específicas de los ejemplos con los que se ha realizado la estadística. Aún así, si el proceso que estamos modelando es complejo (y el habla lo es), una cantidad de parámetros demasiado pequeña no permite representar el proceso en su totalidad.

Por ejemplo, supongamos que estamos intentando modelizar la relación altura-tiempo de un piedra en caída libre para calcular cuanto va a tardar en llegar al suelo. Para ello medimos (con un cierto error)

la altura de dicho objeto en siete instantes de tiempo distintos. Ahora intentamos modelizar la relación altura-tiempo con dos parámetros. Como la relación altura-tiempo es parabólica (al menos en una buena aproximación), dos parámetros son insuficientes incluso para describir los siete puntos que hemos medido. Si realizamos una regresión lineal con esos siete puntos, la línea resultante ni se acercará a unir los siete puntos. Además, si intentamos deducir la posición del objeto en algún instante de tiempo en el que no hemos realizado ninguna medición a partir de esa línea, nuestra deducción no será correcta. Si por el contrario utilizamos 7 parámetros libres, lograremos una curva que se ajusta muy bien a las siete mediciones, demasiado bien. Se ajustará tan bien que esa curva modeliza más bien el ruido de las mediciones que el proceso que estamos intentando modelizar. Si intentamos deducir la posición de la piedra en algún instante de tiempo en el que no hemos realizado ninguna medición, fallaremos, al igual que en el caso con 2 parámetros. En el mejor caso, si utilizamos 3 parámetros conseguiremos una curva que no pasa exactamente por los puntos medidos, pero que sirve para generalizar la altura del objeto en distintos instantes de tiempo. Este ejemplo se muestra gráficamente en la Figura 6.4.

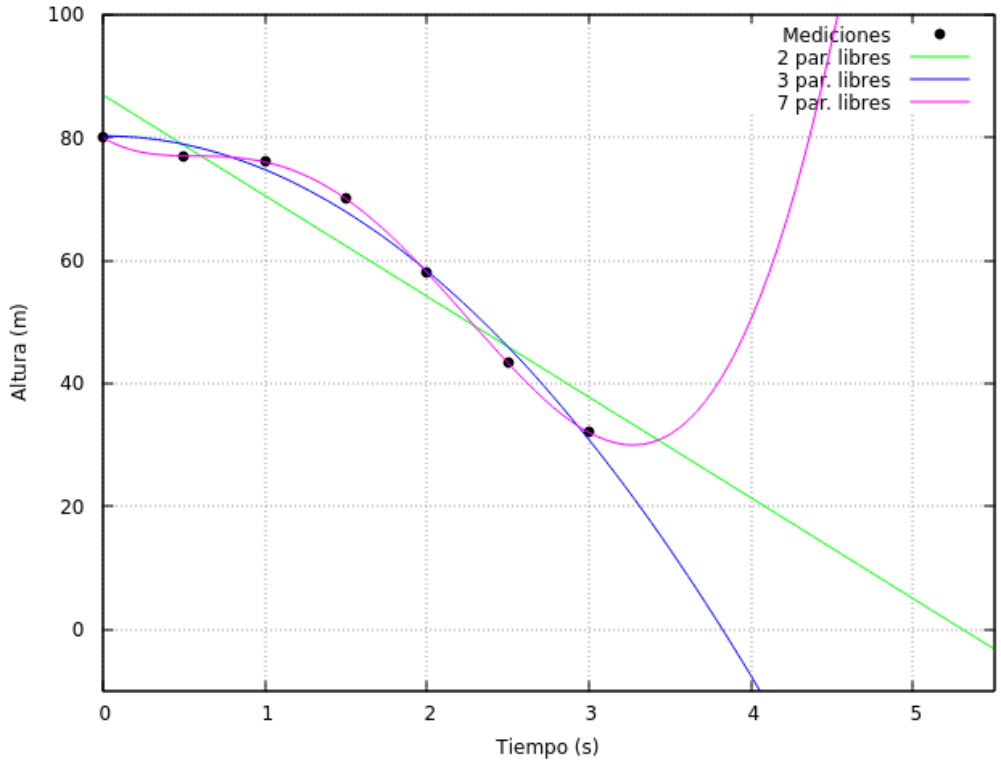


Figura 6.4: Distintas parametrizaciones de siete puntos correspondientes a la altura de una piedra en caída libre en distintos instantes de tiempo. De acuerdo a las ecuaciones de Newton, la piedra tardará 4 segundos en llegar al suelo. El modelo con dos parámetros libres predice que la piedra tardará unos 5.3 segundos en tocar el suelo, el modelo con tres parámetros libres 3.8 segundos y el modelo con 7 parámetros libres predice que la piedra será rescatada por Superman.

La analogía entre el ejemplo del objeto en caída libre y las redes neuronales es clara. Hemos de utilizar suficientes parámetros como para ser capaces de modelizar el proceso que estamos analizando, pero no demasiados como para solo aprender características específicas de los ejemplos de entrenamiento. A continuación realizaremos dos experimentos más. En el primero reduciremos la cantidad de ejemplos con los que entrenar la red neuronal de 3 capas y 1000 neuronas por capa del ejemplo anterior. Las curvas de entrenamiento para el 100 %, 50 % y 25 % de los ejemplos de entrenamiento se muestran en la Figura 6.5.

La Figura 6.5 es una muestra clara de que es más difícil que una red neuronal generalice cuantos menos ejemplos disponemos. El sobre-entrenamiento es más latente con menos ejemplos de entrenamiento, y el rendimiento de la red es menor (el mínimo de la función de coste en los ejemplos de validación es mayor).

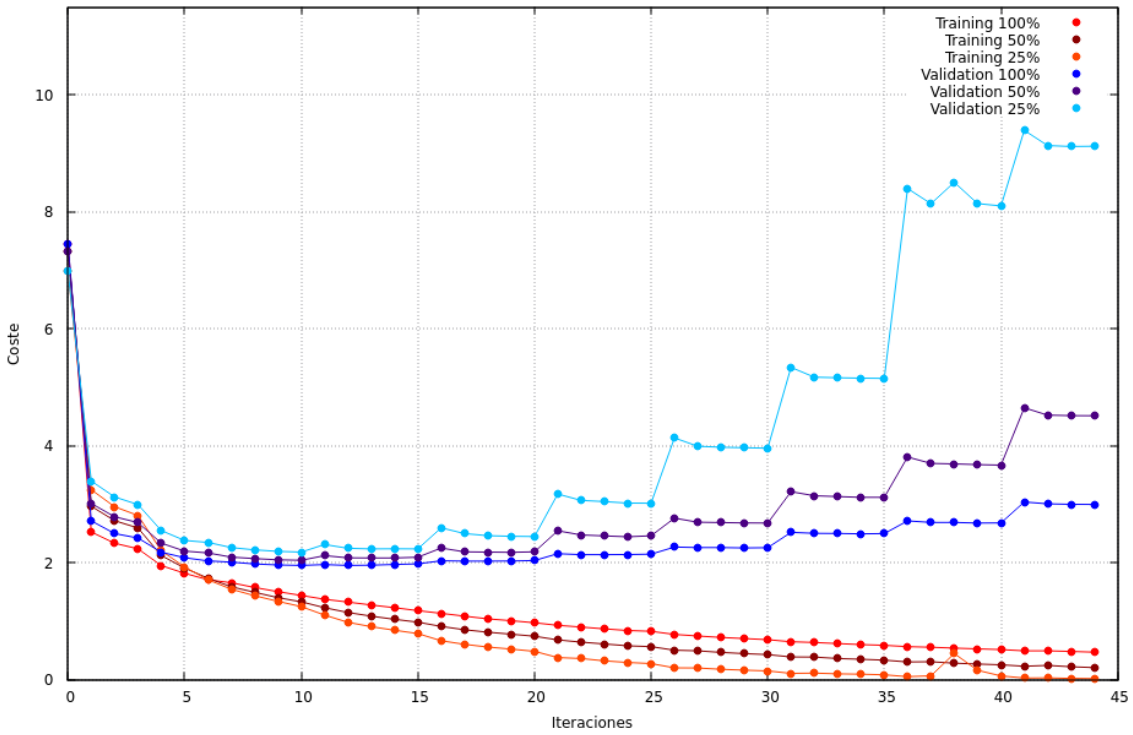


Figura 6.5: Curvas de entrenamiento en una misma red para entrenada con cantidades distintas de ejemplos de entrenamiento.

Como hemos analizado, utilizar un menor número de neuronas reduce el sobre-entrenamiento, pero tampoco es conveniente reducir demasiado el número de neuronas, porque corremos el riesgo de que la red no sea capaz de generalizar. En la Figura 6.6 se muestra el WER para distintos modelos acústicos obtenidos a partir de una red neuronal basada en la tangente hiperbólica con dos capas y distintos números de neuronas por capa. El corpus para el entrenar la red es el 80 % de los audios de todos los corpus, el corpus de test el 20 % de los audios del corpus Dihana-diálogos y el corpus para el modelo de lenguaje el 80 % de las transcripciones del corpus Dihana-diálogos (las de entrenamiento). Cuanto más reducimos la cantidad total de neuronas más se degrada el WER.

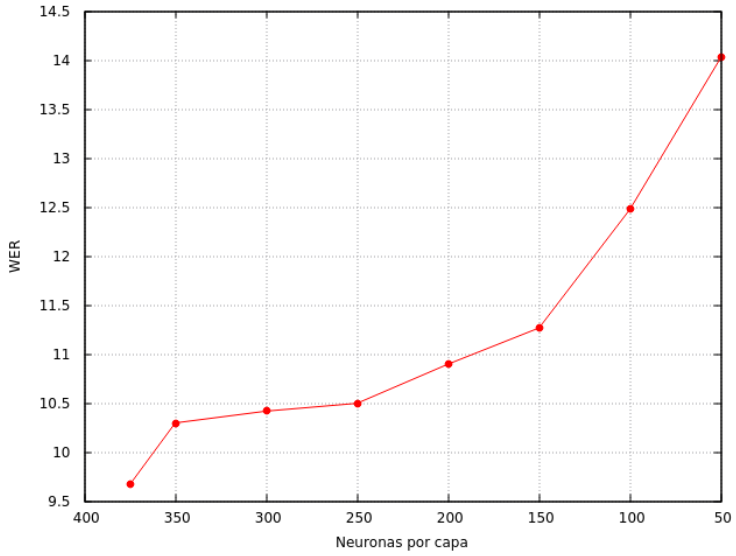


Figura 6.6: WER para modelos acústicos creados con redes con distinto número de neuronas por capa.

Conclusiones

En primer lugar, en este trabajo se han estudiado las estructuras matemáticas en las que se basan los reconocedores de voz generativos. Se han descrito los modelos de mezclas de gaussianas, los modelos ocultos de Markov, las redes neuronales, y también las líneas generales del funcionamiento de los transductores de estados finitos con pesos.

Después se ha analizado como integrar estos modelos para conseguir un reconocedor de voz de habla continua. Primero, se han descrito distintas técnicas para transformar una señal acústica en una secuencia de vectores que contenga información lo más representativa posible de la voz. En segundo lugar, en todo momento se ha asumido que cada vector acústico ha sido emitido por algún estado de un modelo oculto de Markov que modela cada unidad fonética (en la mayoría de casos contextual). Se han representado las probabilidades de emisión de cada estado de este HMM tanto con modelos de mezclas de gaussianas como con redes neuronales. El rendimiento de los reconocedores basados en sistemas híbridos DNN-HMM ha superado notoriamente al de los sistemas GMM-HMM.

En el caso de los sistemas GMM-HMM, la conclusión más importante es que los sistemas basados en trifonemas han mostrado ser más precisos que los sistemas basados en fonemas. La razón es clara, pronunciamos de forma diferente los fonemas en función de cual ha sido el fonema anterior, y de cual será es siguiente.

Respecto a las redes neuronales, los mejores reconocedores desarrollados en este trabajo han sido los que hacían uso de redes neuronales capaces de modelizar características temporales de la voz. En primer lugar, las LSTM bidireccionales entrenadas por secuencias mediante la función de coste MMI han demostrado ser las más precisas. Su compleja estructura recurrente es capaz de guardar información sobre un amplio contexto. Las TDNN también han funcionado muy bien. La estructura de estas redes es muy similar a las redes convolucionales que hoy en día son referentes en el procesamiento de imágenes. Las TDNN, al igual que las redes convolucionales, procesan de forma igual grupos de vectores acústicos centrados en instantes diferentes. Así, las capas más cercanas a la salida son capaces de relacionar las características temporalmente independientes que las primeras capas han detectado. Finalmente, entre las redes *feedforward* convencionales, la función *pnorm* ha funcionado mejor que la tangente hiperbólica como función de activación. La función de activación *pnorm* toma a su entrada la salida de varias funciones de propagación. Esta técnica fue inicialmente empleada en tareas de visión por ordenador, y ha resultado ser también eficaz en el entorno de procesamiento del habla.

En este trabajo también se ha analizado la influencia del modelo de lenguaje en distintas condiciones. Como era de esperar, al emplear un modelo de lenguaje que da una alta probabilidad a las transcripciones de los audios sobre los que se va a realizar un test, el peso del modelo de lenguaje óptimo es mayor que en el caso de utilizar modelos de lenguaje más genéricos. Además, la precisión del reconocedor es menor si utilizamos modelos de lenguaje generales en vez de modelos de lenguaje más ajustados al tipo de habla que vamos a intentar reconocer.

Finalmente, se han realizado varios experimentos para analizar el entrenamiento y rendimiento de las redes neuronales. Respecto al entrenamiento, se ha mostrado que no podemos entrenar indefinidamente una red neuronal, al menos si queremos mejorar su rendimiento. A partir de un punto, la red neuronal comenzará a aprender las características específicas de los ejemplos de entrenamiento. También se ha visto

que el sobre-entrenamiento será más notorio cuanto mayor sea el número de neuronas de la red y menor sea la cantidad de ejemplos de entrenamiento. Por tanto, si queremos reducir el sobre-entrenamiento, una solución posible es reducir el número de neuronas de la red. Aún así, como hemos observado, reducir demasiado la cantidad de neuronas también reduce el rendimiento de la red neuronal. En consecuencia, la cantidad de neuronas de una red ha de ser suficientemente grande como para dotar de capacidad de generalizar a ésta, pero suficientemente pequeña como para que la red no se vea afectada por las características específicas de los ejemplos de entrenamiento.

Bibliografía

[1] Juan Maria Aguirregabiria Aguirre. *Mekanika Klasikoa*. 2013.

[2] James Allen. Speech Recognition and Statistical Language Models. Lecture 9: Extending the Applicability of HMM Models. *Department of Computer Science, University of Rochester*, 2003.

[3] Louis-Marie Aubert, Roger Woods, Scott Fischaber, and Richard Veitch. Optimization of Weighted Finite State Transducer for Speech Recognition. *IEEE Transactions on Computers, Volume:62*, 2013.

[4] L.R. Bahl, P.V. de Souza, P.S. Gopalakrishnan, D. Nahamoo, and M.A. Picheny. Context Dependent Modeling of Phones in Continuous Speech Using Decision Trees. *IBM Research Division*, 1991.

[5] Senaka Buthpitiya, Ian Lane, and Jike Chong. A Parallel Implementation of Viterbi Training for Acoustic Models using Graphics Processing Units. *Department of Electrical and Computer Engineering, Carnegie Mellon University*, 2012.

[6] Yen-Lu Chow and Richard Schwartz. The N-Best Algorithm: An Efficient Procedure for Finding Top N Sentence Hypotheses. *BBN Systems and Technologies Corporation, Cambridge*, 1989.

[7] Desarrolladores de Kaldi. Deep Neural Networks in Kaldi. http://kaldi-asr.org/doc/dnn.html. Última consulta: 23/06/2016.

[8] Desarrolladores de Kaldi. Deep Neural Networks in Kaldi. “Chain" models. http://kaldi-asr.org/doc/chain.html. Última consulta: 20/06/2016.

[9] Desarrolladores de Kaldi. Deep Neural Networks in Kaldi. Dan’s DNN implementation. http://kaldi-asr.org/doc/dnn2.html. Última consulta: 23/06/2016.

[10] Desarrolladores de Kaldi. Feature extraction. http://kaldi-asr.org/doc/feat.html. Última consulta: 18/06/2016.

[11] Desarrolladores de Kaldi. HMM topology and transition modeling. http://kaldi-asr.org/doc2/hmm.html. Última consulta: 19/06/2016.

[12] Desarrolladores de Kaldi. Kaldi. http://kaldi-asr.org/doc/. Última consulta: 19/06/2016.

[13] Nikos Drakos. Viterbi Training (Hinit). http://www.ee.columbia.edu/ln/LabROSA/doc/HTKBook21/node111.html, 1996. Última consulta: 19/06/2016.

[14] Juri Ganitkevitch. Comparison of various feature decorrelation techniques in automatic speech recognition. *Seminar Automatic Speech Recognition. Rheinisch-Westfälische Technische Hochschule Aachen Lehrstuhl für Informatik VI.*, 2005.

[15] Philip N. Garner. A Weighted Finite State Transducer Tutorial. *IDIAP Comunication*, 2007.

[16] Joshua T. Goodman. A Bit of Progress in Language Modeling. *Machine Learning and Applied Statistics Group, Microsoft Research*, 2001.

[17] Alex Graves, Navdeep Jaitly, and Abdel rahman Mohamed. Hybrid Speech Recognition with Deep Bidirectional LSTM. *University of Toronto, Department of Computer Science*, 2013.

[18] Caglar Gulcehre, Kyunghyun Cho, Razvan <pascanu, and Yoshua Bengio. Learned-Norm Pooling for Deep Feedforward and Recurrent Neural Networks. *Département d'Informatique et de Recherche Opérationelle, Université de Montréal and CIFAR Fellow*, 2013.

[19] Jiang Guo. BackPropagation Through Time. *Harbin Institute of Technology*, 2013.

[20] Awni Hannun, Carl Case, Jared Casper, Bryan Catanzaro, Greg Diamos, Erich Elsen, Ryan Prenger, Sanjeev Sathessh, Shubho Sengupta, Adam Coates, and Andrew Y. Ng. Deep Speech: Scaling up end-to-end speech recognition. *Baidu Research – Silicon Valley AI Lab*, 2014.

[21] Lim Zhi Hao. Semirings and WFST. https://www.youtube.com/playlist?list=PLxbPHSSMPBeicXAHVfyFvGfCywRCq39Mp. Última consulta: 19/06/2016.

[22] Geoffrey Hinton, Li Deng, Dong Yu, George Dahl, Abdel rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara Sainath, and Brian Kingsbury. Deep Neural of Acoustic Modeling in Speech Recognition. *Signal Processing Magazine*, 2012.

[23] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Computation 9(8):1735-1780*, 1997.

[24] Björn Hoffmeister, Georg Heigold, David Rybach, Ralf Schlüter, and Hermann Ney. WFST Enabled Solutions to ASR Problems: Beyond HMM Decoding. *IEEE Transactions on Audio, Speech, and Language Processing, Vol. 20 No. 2*, 2012.

[25] Dan Jurafsky, Wayne Ward, Zhang Jianping, Keith Herold, Yu Xiuyang, and Zhang Sen. What Kind of Pronuntiation Variation is Hard for Triphones to Model? *Center for Spoken Language Research, University of Colorado, Boulder*, 2001.

[26] Hongsun Kim. What is an intuitive explanation of Gaussian mixture model? https://www.quora.com/What-is-an-intuitive-explanation-of-Gaussian-mixture-models. Última consulta: 19/06/2016.

[27] David Kriesel. *A Brief Introduction to Neural Networks*. 2005.

[28] James Lyons. Mel Frecuency Cepstral Coefficient (MFCC) tutorial. http://practicalcryptography.com/miscellaneous/machine-learning/guide-mel-frequency-cepstral-coefficients-mfccs/. Última consulta: 22/06/2016.

[29] A.A. Markov. Asprostranenie zakona bol'shih chisel na velichiny, zavisyaschie drug ot druga. *Izvestiya Fiziko-matematicheskogo obschestva pri Kazanskom universitete, 2-ya seriya, tom 15, pp. 135–156*, 1906.

[30] Warren S. McCulloch and Walter Pitts. A Logical Calculus of the Ideas Immanent in Nervous Activity. *University of Illinois and University of Chicago*, 1943.

[31] Yajie Miao, Mohammad Gowayyed, and Florian Metze. EESEN: End-to-End Speech Recongnition using Deep RNN Models and WFST-based Decoding. *Language Technologies Institute, School of Computer Science, Carnegie Mellon University*, 2015.

[32] Mehryar Mohri. Speech Recognition. Lecture 10: Pronunciation Models. http://www.cs.nyu.edu/~mohri/asr12/lecture_10.pdf. Última consulta: 19/06/2016.

[33] Mehryar Mohri, Fernando Pereira, and Michael Riley. Speech Recognition with Weighted Finite State Transducers. *Springer Handbook on Speech Processing and Speech Communication*, 2008.

[34] Andrew C. Morris, Viktoria Maier, and Phill Green. From WER and RIL to MER and WIL: improved evaluation measures for connected speech recognition. *Institute of Phonetics, Saarland University*, 2004.

[35] Lindasalwa Muda, Mumtaj Begam, and I. Elamvazuthi. Voice Recognition Algorithms using Mel Frecuency Cepstral Coefficient (MFCC) and Dynamic Time Warping (DTW) Techniques. *Journal of computing, Volume 2, Issue 3*, 2010.

[36] Walter Nash. 9.0 Speech Recognition Updates. Minimum-Classification-Error (MCE) and Discriminative Training A Primary Problem with the Conventional Training Criterion. http://slideplayer.com/slide/8720678/. Última consulta: 19/06/2016.

[37] Michel A. Nielsen. *Neural Networks and Deep Learning, Chapter 1*. Determination Press, 2015.

[38] Michel A. Nielsen. *Neural Networks and Deep Learning, Chapter 2*. Determination Press, 2015.

[39] Michel A. Nielsen. *Neural Networks and Deep Learning, Chapter 3*. Determination Press, 2015.

[40] Michel A. Nielsen. *Neural Networks and Deep Learning, Chapter 4*. Determination Press, 2015.

[41] Christopher Olah. Understanding LSTM Networks. http://colah.github.io/posts/2015-08-Understanding-LSTMs/, 2015. Última consulta: 19/06/2016.

[42] D.B. Paul. Speech Recognition Using Hidden Markov Models. *The Lincoln Laboratory Journal, Volume 3, Number 1*, 1990.

[43] Vijayadita Peddinti, Daniel Povey, and Sanjeev Khudanpur. A time delay neural networks architecture for efficient modelling of long temporal contexts. *Center for Language and Speech Processing and Human Language Technology Center of Excellence The Johns Hopkins University, Baltimore*, 2015.

[44] Michael Picheny and Stanley F. Chen. Topics in Signal Processing: Speech Recognition. Language Modeling Fever. Implement Witten-Bell smoothing. http://www.ee.columbia.edu/~stanchen/e6884/labs/lab3/x207.html. Última consulta: 24/06/2016.

[45] Daniel Povey. Speech Recognition - a practical guide. Lecture 3: Phonetic Context Dependency. http://www.danielpovey.com/files/Lecture3.pdf. Última consulta: 19/06/2016.

[46] Daniel Povey, Arnab Ghoshal, Gilles Boulianne, Lukáš Burget, Ondřej Glembek, Nagendra Goel, Mirko Hannemann, Petr Motlíček, Yanmin Qian, Petr Schwarz, Jan Silovsky, Georg Stemmer, and Karel Veselý. The Kaldi Speech Recognition Toolkit. *IEEE 2011 Workshop on Automatic Speech Recognition and Understanding, Hilton Waikoloa Village, Big Island, Hawaii, US IEEE Signal Processing Society*, 2011.

[47] Daniel Povey, Mirko Hannemann, Gilles Boulianne, Lukáš Burget, Arnab Ghoshal, Miloš Janda, Martin Karafiát, Stefan Kombrink, Petr Motlíček, Yanmin Qian, Korbinian Riedhammerand Karel Veselý, and Ngoc Thang Vu. Generating Exact Laticces in the WFST Framework. *IEEE International Conference on Acoustics, Speech and Signal Processing*, 2012.

[48] Daniel Povey, Vijayaditya Peddinti, Daniel Galvez, Pegah Ghahrmani, Vimal Manohar, Xingyu Na, Yiming Wang, and Sanjeev Khudanpur. Purely sequence-trained neural networks for ASR based on lattice-free MMI. *Submitted to Interspeech*, 2016.

[49] Daniel Povey, Xiaohui Zhang, and Sanjeev Khudanpur. Parallel Training of DNNs with Natural Gradient and Parameter Averaging. *Center for Language and Speech Processing and Human Language Technology Center of Excellence, The Johns Hopkins University, Baltimore*, 2015.

[50] J.V. Psutka and Luděk Müller. Comparison of various feature decorrelation techniques in automatic speech recognition. *Department of Cybernetics, University of West Bohemia, Pilsen, Czech Republic*, 2006.

[51] Janne Pylkkönen. LDA Based Feature Estimation Methods for LVCSR. *Adaptive Informatics Research Centre. Helsinki University of Technology, Finland*, 2006.

[52] L.R. Rabiner and B.H. Juang. An Introduction to Hidden Markov Models. *IEEE ASSp Magazine*, 1986.

[53] Sebastian Raschka. Linear Discriminant Analysis - Bit by Bit. http://sebastianraschka.com/Articles/2014_python_lda.html. Última consulta: 20/06/2016.

[54] Shakti P. Rath, Daniel Povey, Karel Veselý, and Jan "Honza" Černocký. Improved feature processing for Deep Neural Networks. *Proceedings of the 14th Annual Conference of the International Speech Communication Association (Interspeech 2013)., No. 8, Lyon, FR*, 2013.

[55] Peter Roelants. How to implement a recurrent neural network. Part 1. http://peterroelants.github.io/posts/rnn_implementation_part01/. Última consulta: 19/06/2016.

[56] Sebastian Ruder. An overvieuw of gradient descent optimization algorithms. http://sebastianruder.com/optimizing-gradient-descent/. Última consulta: 19/06/2016.

[57] Andrew Senior, Georg Heigold, Marc'Aurelio Ranzato, and Ke Yang. An empirical study of learning rates in deep neural networks for speech recognition. *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, 2013.

[58] Satyanand Singh and Dr. E.G. Rajan. Vector Quantization Approach for Speaker Recognition using MFCC and Inverted MFCC. *International Journal of Computer Applications (0975 – 8887) Volume 17– No.1*, 2011.

[59] Sten Sootla. Artificial neural network for image classification. *Computational neuroscience project. University of Tartu.*, 2015.

[60] S.S. Stevens. A Scale for the Measurement of the Psychological Magnitude Pitch. *Acoustical Society of America Journal*, 1937.

[61] Ole Morten Strand and Andreas Egeberg. Cepstral mean and variance normalization in the model domain. *COST278 and ISCA Tutorial and Research Workshop on Robustness Issues in Conversational Interaction, University of East Anglia*, 1989.

[62] Michael Trost. Context-Dependent Deep Neural Networks: Breakthrough of neural networks for speech recognition. http://recognize-speech.com/acoustic-model/knn/comparing-different-architectures/context-dependent-deep-neural-networks. Última consulta: 19/06/2016.

[63] Karel Veselý, Arnab Ghoshal, Lukáš Burget, and Daniel Povey. Secuence-discriminative training of deep neural networks. *Brno University of Technology, Czech Republic; Centre for Speech Technology Research, University of Edinburgh; Center for Language and Speech Processing, Johns Hopkins University*, 2013.

[64] Garima Vyas and Barkha Kumari. Speaker Recognition System Based On MFCC and DCT. *International Journal of Engineering and Advanced Technology*, 2013.

[65] Alexander Waibel, Toshiyuki Hanazawa, Geoffrey Hinton, Kiyohiro Shikano, and Kevin J. Lang. Phoneme Recognition Using Time-Delay Neural Networks. *IEEE Transactions on acoustics, speech, and signal processing, Vol. 37, No. 3*, 1989.

[66] Wikipedia. Origin of language. https://en.wikipedia.org/wiki/Origin_of_language. Última consulta: 18/06/2016.

[67] Wikipedia. Stochastic gradient descent. https://en.wikipedia.org/wiki/Stochastic_gradient_descent. Última consulta: 19/06/2016.

[68] S.J. Young, J.J. Odell, and P.C. Woodland. Tree-Based State Tying for High Accuracy Acoustic Modelling. *Cambridge University Engineering Department*, 1994.

[69] Dong Yu and Li Deng. *Automatic Speech Recognition: A Deep Learning Approach*. 2015.

[70] Xiaohui Zhang, Jan Trmal, Daniel Povey, and Sanjeev Khudanpur. Improving Deep Neural Network Acoustic Models using Generalized Maxout Networks. *Center for Language and Speech Processing and Human Language Technology Center of Excellence The Johns Hopkins University, Baltimore*, 2014.