Universidad de Buenos Aires

Facultad de Ingeniería

Año 2019

Tesis de Grado en Ingeniería Electrónica

Diseño e implementación de un sistema *end-to-end* para la conversión de habla a texto mediante redes neuronales

Nicolás Gabriel Zorzano

Director: Ing. Ricardo Veiga

Resumen

El presente trabajo de tesis aborda el problema de la conversión de señales de habla a texto mediante la utilización de redes neuronales. Para esto se analizaron dos arquitecturas que hoy en día son consideradas el estado del arte en el área:

- Redes CTC (*Connectionist Temporal Classification*)
- Mecanismos de atención (*Listen, Attend and Spell*)

Ambas arquitecturas fueron estudiadas de manera teórica, analizando los conceptos claves que explican sus principios de funcionamiento y las principales ventajas y desventajas de cada una de ellas. En base a esto, se realizó la correspondiente implementación en código de los distintos modelos, y se compararon los resultados obtenidos con los de otros autores de referencia.

Se analizó la influencia de las principales técnicas de regularización en el desempeño de los distintos modelos, y se comprobó la fuerte dependencia que existe entre el error de predicción y la cantidad de datos utilizados durante el entrenamiento.

Se comprobó que la red que utiliza mecanismos de atención alcanza un error menor que la red CTC, pero que la segunda tiene ventajas computacionales que facilitan su implementación, la elección de hiperparámetros y, principalmente, los tiempos de ejecución requeridos.

Todos los modelos fueron implementados en código y se encuentran disponibles en un repositorio público. También se analizaron herramientas *online* gratuitas que permiten el entrenamiento de redes neuronales en aquellos casos donde el *hardware* disponible no es suficiente.

Índice

1. Objetivo

El objetivo de este trabajo es el de analizar el estado del arte en el reconocimiento del habla mediante redes neuronales. Las dos estrategias principales para explorar son las siguientes:

- Redes CTC (*Connectionist Temporal Classification*) presentadas por Alex Graves en [12].
- Mecanismos de atención. En particular el trabajo *Listen, Attend and Spell* [5].

Donde ambas tienen como factor común la utilización de redes recurrentes.

De esta forma, el objetivo es el de poder analizar, implementar y validar ambas arquitecturas en *TensorFlow*, comparándolas entre sí y con los modelos de otros autores de referencia. Para realizar dichas comparaciones es necesario tener en cuenta factores como los tiempos de ejecución, la complejidad computacional, la capacidad de reconocimiento de dependencias de largo plazo y, principalmente, el valor de error alcanzado.

Otro objetivo importante es el de crear un repositorio público con los modelos implementados, de manera de facilitar el desarrollo de trabajos futuros relacionados, ya que en algunos casos se observó que la información disponible era escasa.

2. Introducción

Tanto el estudio del habla como el de las redes neuronales no son temas nuevos, sino que surgieron a mediados del siglo pasado. Sin embargo, no fue sino hasta los últimos años en que estos modelos lograron resultados mejores que los alcanzados mediante los métodos clásicos.

Un punto de inflexión para el desarrollo de redes neuronales ocurrió en 2012, cuando Alex Krizhevsky, Ilya Sutskever y Geoff Hinton presentaron una red convolucional profunda en el concurso ***Imagenet*** de detección y clasificación de imágenes [8], mejorando casi en un 10 % la precisión con respecto a los métodos aplicados hasta la fecha. Cercano a esa época comenzaron a proponerse modelos de reconocimiento de habla basados en redes neuronales que alcanzaban resultados cercanos a los de los métodos tradicionales.

El incremento en la capacidad de cómputo, en la capacidad de almacenamiento, en la disponibilidad de datos y, principalmente, la aparición de placas de video, permitieron que hoy en día la mayoría de los sistemas dentro de la categoría de *Deep Learning* puedan alcanzar resultados incluso mejores a los obtenidos por humanos. Es por esta razón que esta área resulta de gran interés, ya que permite resolver problemas altamente complejos y es posible que en un futuro (no tan lejano) resulte ser un recurso indispensable para la mayoría de las tareas.

El trabajo se encuentra dividido en las siguientes secciones:

- **Marco teórico**: Se presentan las herramientas necesarias para poder desarrollar los conceptos teóricos de los modelos a analizar.
- **Arquitecturas bajo estudio**: Se utilizan las herramientas desarrolladas en la sección previa para explicar el principio de funcionamiento teórico de las arquitecturas de interés.
- **Implementaciones**: Se detallan los factores a tener en cuenta al implementar el código de cada uno de los modelos. Se mencionan detalles importantes sobre la utilización de TensorFlow para el armado de redes neuronales, así como también se explican configuraciones particulares para este trabajo.
- **Resultados y discusión**: Se analizan los resultados obtenidos y se extraen conclusiones.
- **Apéndices**: Se da un panorama más detallado sobre trabajos previos relacionados, y se mencionan algunas tareas periféricas al proyecto que debieron ser realizadas.

Parte I

Marco teórico

3. Cadenas de Markov

Las cadenas de Markov fueron propuestas por Andréi Márkov a comienzos del siglo XX. Las cadenas de Markov permiten modelar procesos aleatorios que presentan un cierto grado de memoria[19].

3.1. Cadena de Markov de tiempo discreto

Sea $\boldsymbol{x} = [x_1, x_2, \ldots, x_n]$ una secuencia de variables aleatorias extraídas de un alfabeto finito $S = \{s_1, \ldots, s_M\}$. Aplicando la regla de la cadena de Bayes se puede expresar la probabilidad conjunta como:

$$P(x_1, \ldots, x_n) = P(x_1) \cdot \prod_{i=2}^{n} P(x_i | x_1, x_2, \ldots, x_{i-1}) \tag{3.1}$$

En base a esto, se plantea la **suposición de Markov**: En un proceso de Markov de orden N, la probabilidad de una variable aleatoria en un determinado instante de tiempo depende únicamente de los N valores pasados del proceso. Esto implica que toda la historia del proceso aleatorio hasta un determinado instante, puede ser contenida dentro de esos N valores previos. De acuerdo con esta suposición, y para el caso de un proceso de primer orden, la expresión en 3.1 se puede reducir como:

$$\boxed{P(x_i | x_1, x_2, \ldots, x_{i-1}) = P(x_i | x_{i-1})} \rightarrow P(x_1, \ldots, x_n) = P(x_1) \cdot \prod_{i=2}^{n} P(x_i | x_{i-1}) \tag{3.2}$$

Las variables aleatorias x_i pueden incluso ser asociadas a estados, cuyas transiciones están regidas por la probabilidad de transición $P(q_t = s_i | q_{t-1} = s_j)$ donde q_t es el estado del sistema en el instante actual, que puede tomar alguno de los M posibles estados derivados de S. Gracias a esto es posible definir una matriz de transiciones:

$$a_{ij} = P(q_t = s_i | q_{t-1} = s_j) \rightarrow \begin{cases} 1 \leq i \\ 1 \leq j \leq M \\ a_{ij} \geq 0 \\ \sum_{j=1}^{M} a_{ij} = 1 \end{cases}$$

Un ejemplo que puede ilustrar esto es el caso de un sistema que presenta tres posibles estados $\{s_1, s_2, s_3\}$ con matriz de transición A, y probabilidades iniciales $\boldsymbol{r} = \begin{bmatrix} r_1 & r_2 & r_3 \end{bmatrix}^T$. Un posible problema a plantear podría ser el de encontrar la probabilidad de que ocurra una determinada secuencia (e.g. $\boldsymbol{s} = [s_3, s_2, s_1, s_1, s_1, s_3]$), la cual se obtiene fácilmente haciendo:

$$\begin{aligned} P(s_3, s_2, s_1, s_1, s_1, s_3) =& P(s_3) \cdot P(s_2 | s_3) \cdot P(s_1 | s_2) \cdot P(s_1 | s_1) \cdot P(s_1 | s_1) \cdot P(s_3 | s_1) \\ =& r_3 \cdot a_{32} \cdot a_{21} \cdot a_{11} \cdot a_{11} \cdot a_{13} \end{aligned}$$

Otra pregunta interesante que es posible responder resulta: Dado que el sistema se encuentra en un determinado estado conocido, ¿cuál es la probabilidad de que permanezca en dicho estado durante d intervalos de tiempo?. La observación esperada resulta:

$$\boldsymbol{s} = \left[\underbrace{s_i, s_i, \ldots, s_i, s_i}_{d}, s_j \neq s_i \right]$$

por lo que su probabilidad se puede calcular como:

$$P\left(\underbrace{s_i, s_i, \ldots, s_i, s_i}_{d}, s_j \neq s_i | q_1 = s_i\right) = \overbrace{P\left(s_i|s_i\right) \cdot \cdots \cdot P\left(s_i|s_i\right)}^{d} \cdot P\left(s_j|s_i\right)$$
$$= \overbrace{a_{ii} \cdot \cdots \cdot a_{ii}}^{d} \cdot a_{ij}$$
$$= a_{ii}^{d}\left(1 - a_{ii}\right) = p_i\left(d\right)$$

Se puede ver entonces que la duración del estado s_i se distribuye de manera exponencial. Por otra parte, calculando la esperanza de dicha distribución se observa que la duración esperada de un determinado estado resulta:

$$\mathbb{E}\left[d_i\right] = \sum_{d=1}^{\infty} d \cdot p_i\left(d\right) = \left(1 - a_{ii}\right) \sum_{d=1}^{\infty} d \cdot a_{ii}^{d} = \frac{a_{ii}}{\left(1 - a_{ii}\right)^2} \cdot \left(1 - a_{ii}\right) = \frac{a_{ii}}{1 - a_{ii}}$$

Los procesos descritos en esta sección se caracterizan por tener estados que pueden ser observados de manera directa. Esto generalmente no ocurre, y por ende se debe recurrir a indicadores que permiten observar de manera indirecta los estados reales del sistema.

3.2. Modelos ocultos de Markov (HMM)

A diferencia de las cadenas de Markov convencionales, en este caso las observaciones son una función aleatoria que depende del estado real, de manera que se trata de un proceso aleatorio embebido dentro de otro proceso aleatorio.

Un proceso que puede ser modelado mediante HMM es el de la extracción de bolas de colores de distintas urnas. Supóngase que se tienen N urnas con bolas de distintos colores en ellas. En cada instante de tiempo se elige una urna de manera aleatoria[1] y se extrae una bola de algún determinado color, pero sin dar a conocer de qué urna fue extraída. En este caso se tienen las observaciones de los distintos colores de las bolas, pero no se conocen los estados reales que son las distintas urnas.

Los elementos que componen un HMM se resumen en:

- El alfabeto de las observaciones $O = \{o_1, \ldots, o_M\}$.
- El conjunto de estados posibles $S = \{s_1, \ldots, s_N\}$.
- Las probabilidades de transición $A = \{a_{ij}\}$, donde $P\left(q_t = s_i | q_{t-1} = s_j\right)$. Se debe tener en cuenta que $\sum_{j=1}^{N} a_{ij} = 1$.
- Las probabilidades de observación $B = \{b_i\left(k\right)\}$, donde $b_i\left(k\right)$ es la probabilidad de emitir el símbolo o_k estando en el estado s_i:

  $$b_i\left(k\right) = P\left(x_t = o_k | q_t = s_i\right)$$

  donde x_t simboliza la observación en el estado actual. Se debe tener en cuenta que $\sum_{k=1}^{M} b_i\left(k\right) = 1$.
- Probabilidades iniciales $r = \{r_i\}$, con $r_i = P\left(q_1 = s_i\right)$. Se debe tener en cuenta que $\sum_{i=1}^{N} r_i = 1$.

Por conveniencia, en general todos estos elementos se suelen agrupar en $\Phi = (A, B, r)$.

Para los HMM, además de aplicar la suposición de Markov de la ecuación 3.2, se plantea la **suposición de independencia de salida** que establece que la probabilidad de observar un determinado símbolo en el instante t depende únicamente del estado actual q_t, y es condicionalmente independiente de las observaciones pasadas:

$$\boxed{P\left(x_t | x_1, x_2, \ldots x_{t-1}, q_1, q_2, \ldots, q_t\right) = P\left(x_t | q_t\right) \rightarrow \begin{cases} x_i \in O \\ q_i \in S \end{cases}} \tag{3.3}$$

En base a estas definiciones, se pueden plantear tres problemas de interés a resolver con estos modelos.

[1] Siguiendo la distribución de las distintas transiciones.

- ***Evaluation problem**:* Dado un modelo Φ y una secuencia de observaciones $\boldsymbol{x} = (x_1, \ldots, x_T)$, encontrar la probabilidad de que dichas observaciones hayan sido generadas por dicho modelo, es decir $P(\boldsymbol{x}|\Phi)$.
- ***Decoding problem***: Dado un modelo Φ y una secuencia de observaciones $\boldsymbol{x} = (x_1, \ldots, x_T)$, encontrar cuál es la secuencia de estados $\boldsymbol{s} = (q_1, \ldots, q_T)$ más probable que explica dichas observaciones.
- ***Learning problem***: Dado un modelo Φ y una secuencia de observaciones $\boldsymbol{x} = (x_1, \ldots, x_T)$, ajustar el modelo $\hat{\Phi}$ de manera de maximizar la probabilidad conjunta $P\left(\boldsymbol{x}|\hat{\Phi}\right)$.

3.2.1. Solución al *evaluation problem*

Se busca encontrar la manera de calcular $P(x_1, x_2, \ldots, x_T|\Phi)$. Esta probabilidad se puede reescribir como:

$$P(\boldsymbol{x}|\Phi) = \sum_{\forall \boldsymbol{s}} P(\boldsymbol{s}|\Phi)\, P(\boldsymbol{x}|\boldsymbol{s}, \Phi)$$

Aplicando la suposición de Markov de la expresión 3.2 el primer término se puede expresar como:

$$P(\boldsymbol{s}|\Phi) = P(q_1|\Phi) \prod_{t=2}^{T} P(q_t|q_{t-1}, \Phi) = r_{q_1} \cdot a_{q_1 q_2} \cdots a_{q_{T-1} q_T}$$

Aplicando la suposición de independencia de salida 3.3 el segundo término resulta:

$$P(\boldsymbol{x}|\boldsymbol{s}, \Phi) = \prod_{t=1}^{T} P(x_t|q_t, \Phi) = \prod_{t=1}^{T} b_{q_t}(x_t) = b_{q_1}(x_1) \cdots b_{q_T}(x_T)$$

La probabilidad deseada entonces queda:

$$\begin{aligned} P(\boldsymbol{x}|\Phi) &= \sum_{\forall \boldsymbol{s}} r_{q_1} \cdot a_{q_1 q_2} \cdots a_{q_{T-1} q_T} \cdot b_{q_1}(x_1) \cdots b_{q_T}(x_T) \\ &= \sum_{\forall \boldsymbol{s}} (r_{q_1} b_{q_1}(x_1)) \cdot (a_{q_1 q_2} \cdot b_{q_2}(x_2)) \cdots \left(a_{q_{T-1} q_T} \cdot b_{q_T}(x_T)\right) \end{aligned}$$

Si este problema se quisiera resolver por fuerza bruta se debería proceder de la siguiente manera:

- Se elije una secuencia de estados $\boldsymbol{s}_1 = (q_1, \ldots, q_T)$.
- Se comienza la secuencia con q_1 con probabilidad r_{q_1}.
- Se realizan las transiciones de acuerdo con a_{q_{t-1}, q_t} y se generan las observaciones con probabilidad $b_{q_t}(x_t)$ hasta llegar a $t = T$.
- Se repite el proceso para las demás secuencias de estados s_i y se suman los resultados.

Se puede ver que esta definición requiere sumar para todas las posibles secuencias de estados $\boldsymbol{s}$, lo cual resulta en una operación $\mathcal{O}\left(N^T\right)$, siendo una tarea casi imposible de realizar de manera eficiente. Es por esta razón que se utiliza un algoritmo conocido como ***forward-backward***, que se describe a continuación.

Algoritmo *forward-backward*

Se define la variable ***forward*** $\alpha_t(i)$ como la probabilidad de observar las observaciones parciales hasta el instante t y estar en el estado s_i en el instante t dado el modelo:

$$\boxed{\alpha_t(i) = P(x_1, x_2, \ldots, x_t, q_t = s_i|\Phi)} \tag{3.4}$$

Esta variable puede ser calculada de forma iterativa realizando el siguiente procedimiento:

- Se inicializa $\alpha_1(i) = r_i b_i(x_1)$ con $1 \leq i \leq N$.

- Este paso es conocido como el de **inducción**:

$$\alpha_{t+1}(j) = \left(\sum_{i=1}^{N} \alpha_t(i)\, a_{ij}\right) b_j(x_{t+1}) \rightarrow \begin{cases} 1 \leq t \leq T-1 \\ 1 \leq j \leq N \end{cases}$$

  Dado que $\alpha_t(i)$ es la probabilidad de estar en el estado s_i y de haber observado la secuencia $(x_1, \ldots, x_t)$ en el instante t, entonces el producto $\alpha_t(i)\, a_{ij}$ es la probabilidad de haber observado la secuencia $(x_1, \ldots, x_t)$ y de haber alcanzado el estado s_j en el instante $t+1$ partiendo de s_i en t. Sumando estas probabilidades para los N posibles estados en el instante t, se obtiene la probabilidad de haber observado la secuencia $(x_1, \ldots, x_t)$ y de estar en el estado s_j en el instante $t+1$ (ver figura 3.1). Finalmente, multiplicando por la probabilidad de emisión $b_j(x_{t+1})$ se obtiene la variable *forward* para el instante siguiente. Este cálculo se debe repetir para todo instante t de la secuencia y para los distintos estados.

- Finalmente si se llega al estado final:

$$P(\boldsymbol{x}|\Phi) = \sum_{i=1}^{N} \alpha_T(i)$$

Se puede ver que mediante este método es posible resolver el *evaluation problem*, pero con la ventaja de que la complejidad computacional requerida en este caso es $\mathcal{O}\left(N^2 T\right)$.

De manera similar se define la variable ***backward*** $\beta_t(i)$ como la probabilidad de observar las observaciones parciales desde el instante $t+1$ hasta T, dado el estado s_i en el instante t:

$$\boxed{\beta_t(i) = P(x_{t+1}, x_{t+2}, \ldots, x_T|\Phi, q_t = s_i)} \tag{3.5}$$

De la misma manera que para α_t, la variable *backward* puede ser calculada de manera iterativa:

- Se inicializa $\beta_T(i) = 1$ con $1 \leq i \leq N$. El valor se elige de manera **arbitraria** como la unidad.

- El paso de inducción en este caso resulta:

$$\beta_t(i) = \sum_{j=1}^{N} a_{ij} b_j(x_{t+1})\, \beta_{t+1}(j) \rightarrow \begin{cases} t = T-1, T-2, \ldots, 1 \\ 1 \leq j \leq N \end{cases}$$

  La interpretación de este paso es similar al del caso anterior (ver figura 3.1), con la diferencia que debe comenzarse desde el final de la secuencia.

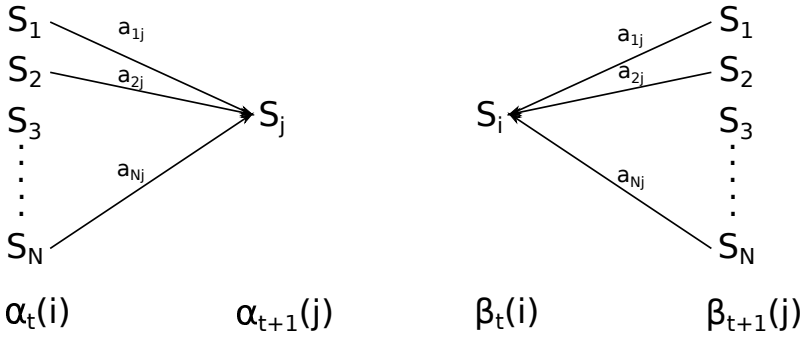


Figura 3.1: Interpretación gráfica para el cálculo de las variables *forward* y *backward*. Se muestran los caminos para llegar al estado s_j desde los N posibles estados en el instante anterior y en el siguiente.

3.2.2. Solución al *decoding problem*

Para encontrar la secuencia de estados que mejor explica las observaciones, es necesario definir primero algún criterio de optimalidad. Para implementar esta solución se define la probabilidad de estar en el estado s_i en el

instante t, dado el modelo y la secuencia de observaciones:

$$\boxed{\gamma_t(i) = P(q_t = s_i|\boldsymbol{x}, \Phi) \qquad \sum_{i=1}^{N} \gamma_t(i) = 1} \tag{3.6}$$

Esta probabilidad puede ser expresada a partir de las variables *forward-backward* definidas en la sección 3.2.1:

$$\gamma_t(i) = \frac{\alpha_t(i)\,\beta_t(i)}{P(\boldsymbol{x}|\Phi)} = \frac{\alpha_t(i)\,\beta_t(i)}{\sum_{j=1}^{N} \alpha_t(j)\,\beta_t(j)}$$

Conociendo los valores de $\gamma_t(i)$ para todo $1 \leq i \leq N$ y $1 \leq t \leq T$, es decir la probabilidad de estar en cada uno de los estados para todo instante de tiempo, una posible elección de secuencia óptima podría ser la que cumple que $q_t = \text{argmax}_{1 \leq i \leq N}(\gamma_t(i))$. Este criterio consiste en elegir de manera independiente en cada instante de tiempo aquel estado que resulta más probable.

Si bien este criterio resulta muy común para ciertas aplicaciones, otro posible criterio es el conocido como **algoritmo de Viterbi** [38].

Algoritmo de Viterbi

Este algoritmo puede ser considerado como una modificación del algoritmo *forward*. En lugar de sumar las probabilidades de todos los caminos que llegan a un mismo estado, este algoritmo elige y almacena el mejor camino.

El criterio para encontrar la mejor secuencia de estados es:

$$\delta_t(i) = \max_{q_1,\ldots,q_{t-1}} P(q_1, q_2, \ldots, q_t = s_i, x_1, x_2, \ldots, x_t|\Phi)$$

Se puede ver que $\delta_t(i)$ define la secuencia de máxima probabilidad hasta el instante t que pasa por el estado s_i, tomando en cuenta las observaciones hasta dicho instante. Para poder realizar el cálculo de manera eficiente, nuevamente se puede plantear de forma recursiva haciendo:

$$\delta_{t+1}(j) = \left(\max_i \delta_t(i)\,a_{ij}\right) \cdot b_j(x_{t+1})$$

Los pasos a seguir para resolver este algoritmo entonces resultan:

- Inicialización:
$$\begin{cases} \delta_1(i) = r_i b_i(x_1) & 1 \leq i \leq N \\ \psi_1(i) = 0 \end{cases}$$
- Recursión:
$$\begin{cases} \delta_t(j) = (\max_{1 \leq i \leq N} \delta_{t-1}(i)\,a_{ij}) \cdot b_j(x_t) \\ \psi_t(j) = \text{argmax}_{1 \leq i \leq N} \delta_{t-1}(i)\,a_{ij} \end{cases} \qquad 2 \leq t \leq T$$
- Finalización:
$$\begin{cases} P^* = \max_{1 \leq i \leq N} \delta_T(i) \\ q_T^* = \text{argmax}_{1 \leq i \leq N} \delta_T(i) \end{cases}$$
- Se recupera la secuencia óptima:
$$q_t^* = \psi_{t+1}\left(q_{t+1}^*\right) \qquad t = T-1, T-2, \ldots, 1$$

3.2.3. Solución al *learning problem*

Éste constituye el problema más difícil de resolver ya que no se conocen métodos analíticos que permitan resolverlo de manera cerrada. Sí existen métodos iterativos, como el de Baum-Welch [28] que permiten resolver este problema.

Para poder desarrollar el método de Baum-Welch es necesario definir a la probabilidad de encontrarse en el estado s_i en el instante t y de estar en el estado s_j en el instante $t+1$, dados el modelo y las observaciones:

$$\boxed{\xi_t(i,j) = P(q_t = s_i, q_{t+1} = s_j | \boldsymbol{x}, \Phi)}$$

Esta probabilidad puede ser expresada en función de las variables *forward-backward* como:

$$\xi_t(i,j) = \frac{\alpha_t(i)\, a_{ij} b_j(x_{t+1})\, \beta_{t+1}(j)}{P(\boldsymbol{x}|\Phi)} = \frac{\alpha_t(i)\, a_{ij} b_j(x_{t+1})\, \beta_{t+1}(j)}{\sum_{i=1}^{N}\sum_{j=1}^{N} \alpha_t(i)\, a_{ij} b_j(x_{t+1})\, \beta_{t+1}(j)}$$

En la figura 3.2 se tiene una interpretación gráfica de este resultado. Por otra parte, esta probabilidad también puede ser relacionada con $\gamma_t(i)$ sumando para todos los posibles estados en $t+1$:

$$\gamma_t(i) = \sum_{j=1}^{N} \xi_t(i,j)$$

Sumando $\gamma_t(i)$ a lo largo del tiempo (para $1 \leq t \leq T-1$) se obtiene un resultado que puede ser interpretado como esperanza de la cantidad de veces que el estado s_i es visitado. De igual manera, la suma de $\xi_t(i,j)$ a lo largo del tiempo (para $1 \leq t \leq T-1$), se puede interpretar como la esperanza de la cantidad de transiciones del estado s_i al estado s_j.

Usando estos conceptos, las fórmulas de reestimación de los parámetros del modelo se pueden escribir como:

$$\begin{cases} \hat{r}_i = \mathbb{E}\left[\text{Cantidad de veces en el estado } s_i \text{ en } t=1\right] = \gamma_1(i) \\ \hat{a}_{ij} = \frac{\mathbb{E}[\text{Cantidad de transiciones de } s_i \text{ a } s_j]}{\mathbb{E}[\text{Cantidad de transiciones de } s_i]} = \frac{\sum_{t=1}^{T-1} \xi_t(i,j)}{\sum_{t=1}^{T-1} \gamma_t(i)} \\ \hat{b}_j(k) = \frac{\mathbb{E}[\text{Cantidad de veces en el estado } s_j \text{ observando } o_k\ ]}{\mathbb{E}[\text{Cantidad de veces en el estado } s_j]} = \frac{\sum_{t=1,\, x_t = o_k}^{T} \gamma_t(j)}{\sum_{t=1}^{T} \gamma_t(i)} \end{cases}$$

De acuerdo con el modelo de Baum-Welch [28], se puede demostrar que el nuevo modelo $\hat{\Phi} = \left(\hat{A}, \hat{B}, \hat{r}\right)$ resulta en un modelo más probable en el sentido de $P\left(\boldsymbol{x}|\hat{\Phi}\right) \geq P(\boldsymbol{x}|\Phi)$.

Este método de reestimación puede ser derivado también de la siguiente función auxiliar introducida por Baum:

$$Q\left(\Phi, \hat{\Phi}\right) = \sum_{\boldsymbol{s}} P(\boldsymbol{s}|\boldsymbol{x}, \Phi) \log\left(P\left(\boldsymbol{s}, \boldsymbol{x}|\hat{\Phi}\right)\right) = \sum_{\boldsymbol{s}} \frac{P(\boldsymbol{s}, \boldsymbol{x}|\Phi)}{P(\boldsymbol{x}|\Phi)} \log\left(P\left(\boldsymbol{s}, \boldsymbol{x}|\hat{\Phi}\right)\right)$$

para la cual está demostrado que al ser maximizada, se está maximizando también el likelihood.

Este método puede ser pensado como la aplicación del algoritmo EM (*Expectation-Maximization*) *[7]* donde el paso E es el cálculo de la función auxiliar $Q\left(\Phi, \hat{\Phi}\right)$ y el paso M es la maximización en función de $\hat{\Phi}$.

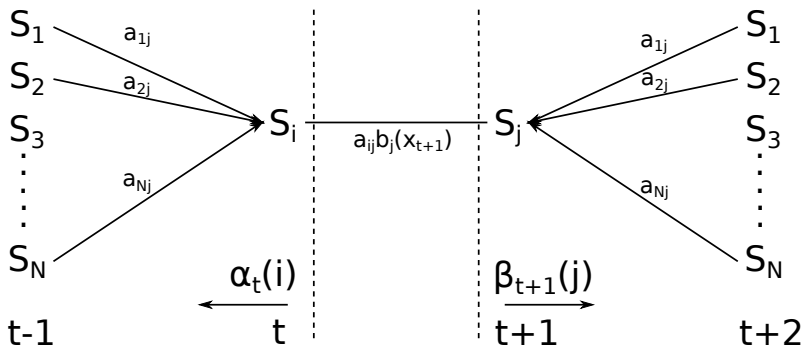


Figura 3.2: Interpretación gráfica para el cálculo de las variables de $\xi_t(i,j)$.

4. Procesamiento de las señales de habla

Un paso muy importante para trabajar con redes neuronales es el de acondicionar correctamente los datos que se desean analizar. Si bien es deseable que la red sea capaz de aprender la mejor representación de manera automática, esto puede provocar que los tiempos de entrenamiento requeridos y la complejidad del modelo se vuelvan prohibitivos.

Para trabajar con señales de habla un primer paso podría ser el de intentar utilizar los audios de manera directa. En este caso la entrada estaría compuesta por señales $\boldsymbol{x} \in \mathbb{R}^{1\times T}$, y la red neuronal sería la encargada de extraer la información relevante. Algunas de las desventajas de este método son:

- La señal útil se encuentra distorsionada por el ruido.
- Las redes neuronales mejoran su rendimiento cuando se trata de problemas de mayor dimensionalidad.
- Las señales de entrada pueden ser muy largas ralentizando el proceso de aprendizaje.

Por estas razones es que se suele utilizar otro tipo de representaciones para trabajar con señales de habla, las cuales se presentarán a continuación.

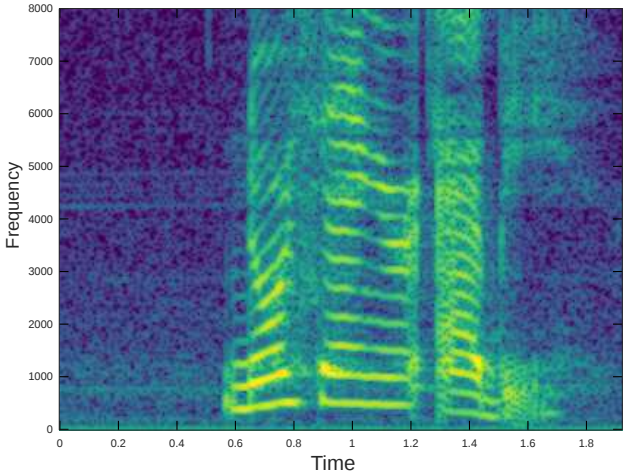


Figura 4.1: Ejemplo de un espectrograma.

4.1. Espectrogramas

Una manera de favorecer la separación entre la señal útil y el ruido es pasando al dominio de la frecuencia, ya que generalmente el ruido se encuentra ubicado en frecuencias superiores.

Un espectrograma es un gráfico que muestra el contenido armónico de una señal en función del tiempo. Suele ser presentado en mapas de color como el de la figura 4.1, donde los ejes se corresponden con el tiempo y la frecuencia.

Para obtener un espectrograma se debe proceder de la siguiente manera:

- Se define un tamaño de ventana y de solapamiento. En general para este tipo de problemas se utilizan ventanas de 25 ms y un solapamiento de 10 ms.
- Por cada ventana de audio se calcula su FFT:

$$X(k) = \sum_{n=0}^{N-1} x(n)\, e^{-j2\frac{\pi k n}{N}} \qquad k = 0, \ldots, N-1$$

- La FFT de cada ventana se coloca de manera vertical una al lado de la otra, formando la imagen de la figura 4.1.

A pesar de que aumentar la dimensionalidad de la representación favorece el aprendizaje, utilizar espectrogramas presenta el siguiente problema: Una misma señal enunciada en distintos tonos de voz tendrá representaciones diferentes. Esto es muy común cuando se analizan audios de hombres y mujeres, ya que las mujeres suelen tener un tono de voz más agudo. Esto puede observase en la figura 4.2, donde se ve que el espectro correspondiente al hombre comienza a una frecuencia menor que el de la mujer, y se extiende en un rango menor de frecuencias.

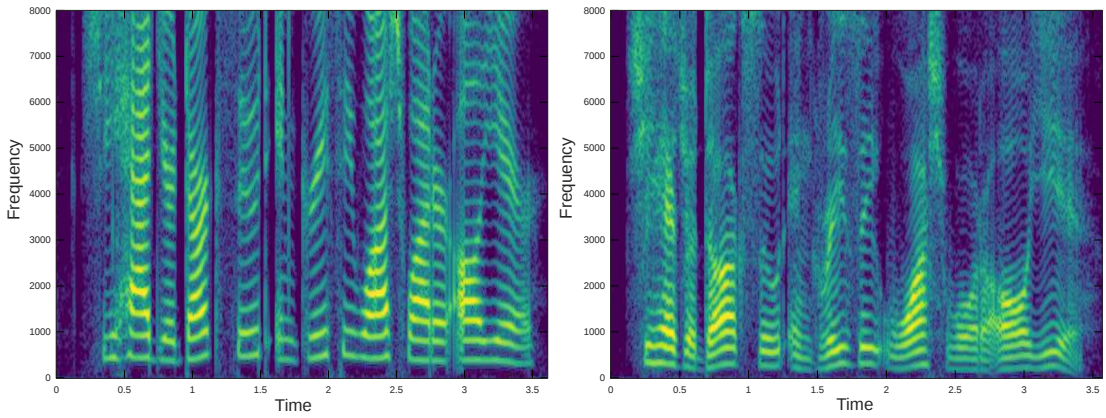


Figura 4.2: Comparación de los espectrogramas de audios correspondientes a una mujer (izquierda) y un hombre (derecha). Los audios utilizados fueron "SA1_FKSR0.WAV" y "SA1_MDHL0.WAV" obtenidos de la base de datos de habla TIMIT [10].

4.2. Coeficientes MFCC

Los MFCC (*Mel Frequency Cepstral Coefficients*) son coeficientes que representan las señales de habla basados en la percepción auditiva humana, permitiendo extraer características relevantes de la señal dejando de lado cosas como el volumen, tono, etc. Por esta razón, estos coeficientes suelen ser la opción más conveniente para el análisis de señales de habla.

La principal ventaja del uso de los coeficientes MFCC es que utilizan la escala no lineal Mel, la cual aproxima el comportamiento del sistema auditivo humano, donde la distribución de frecuencias resulta logarítmica.

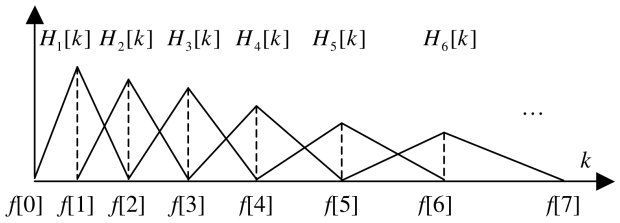


Figura 4.3: Banco de filtros en la escala Mel.

La obtención de los coeficientes MFCC se realiza de la siguiente manera:

- Se ventanea la señal de igual manera que cuando se realiza un espectograma. Generalmente se utilizan ventanas de 25 ms con un solapamiento de 10 ms.
- A cada ventana se le aplica la FFT:

$$X(k) = \sum_{n=0}^{N-1} x(n)\, e^{-j2\frac{\pi kn}{N}} \qquad k = 0, \ldots, N-1$$

- Se define el banco de filtros en la escala Mel. Estos filtros calculan el espectro promedio alrededor de cada frecuencia central, utilizando anchos de banda cada vez mayores (ver figura 4.3). Una posible definición para dichos filtros es:

$$H_m[k] = \begin{cases} 0 & k < f[m-1] \\ \frac{2(k-f[m-1])}{(f[m+1]-f[m-1])(f[m]-f[m-1])} & f[m-1] \leq k < f[m] \\ \frac{2(f[m+1]+k)}{(f[m+1]-f[m-1])(f[m]-f[m-1])} & f[m] \leq k < f[m+1] \\ 0 & k > f[m+1] \end{cases}$$

- Se calcula el logaritmo de la energía de salida de cada uno de los filtros:

$$S[m] = \ln\left(\sum_{k=0}^{N-1} |X[k]|^2 H_m[k]\right) \qquad 0 \leq m < M$$

- Finalmente se aplica la Transformada Coseno Discreta para encontrar los coeficientes:

$$\boxed{c[n] = \sum_{m=0}^{M-1} S[m] \cos\left(\frac{\pi n}{M}\left(m + \frac{1}{2}\right)\right) \qquad 0 \leq n < M}$$

El resultado de esta operación es una matriz que tiene M *features* por cada ventana de la señal, es decir que su resultado es muy similar al de un espectograma (ver figura 4.4). Generalmente en los trabajos de reconocimiento de habla suelen utilizarse entre 13 y 26 coeficientes MFCC, ya que dichas cantidades han demostrado ser las que arrojan mejores resultados. Es interesante notar que a pesar de utilizar muchos menos *features* que los espectogramas (las FFT suelen calcularse con 512 o 1024 puntos), los coeficientes MFCC generalmente llevaron a mejores resultados, lo cual ejemplifica el hecho de que una mayor dimensionalidad no siempre es mejor.

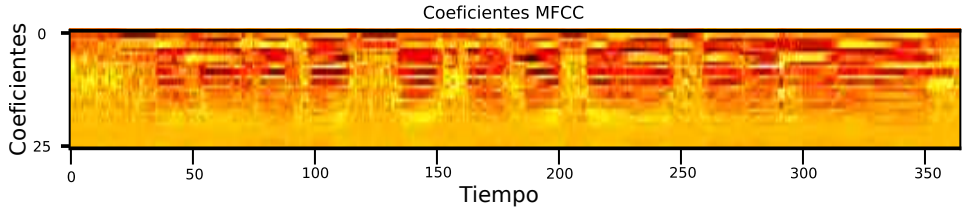


Figura 4.4: Ejemplo de una matriz de coeficientes MFCC.

5. Redes neuronales

5.1. Introducción

El estudio de las redes neuronales comenzó en la década del 40' gracias a McCulloch y Pitts [24], quienes establecieron el primer modelo matemático de una neurona. Sin embargo, no fue sino hasta los últimos años que éstas comenzaron a tomar mayor relevancia. En sus inicios, las redes neuronales fueron concebidas con el objetivo de emular las neuronas reales, y de esta manera recrear el proceso de aprendizaje que se desarrolla en el cerebro humano. Si bien hoy en día siguen existiendo ramas de investigación orientadas a dicho objetivo, se ha encontrado que las redes neuronales, cuando no se las restringe a modelos biológicos, incluso resultan más efectivas en ciertas tareas que los seres humanos. Gracias al incremento en la cantidad de datos de entrenamiento disponibles y a la aparición de placas de video capaces de paralelizar ciertas operaciones, se ha visto en los últimos años que las redes neuronales han comenzado a ser aplicadas a nuevas actividades, muchas de las cuales se encuentran fuera del mundo académico. Es por esto que hoy en día son reales ciertas cosas que antes pudieron ser consideradas ciencia ficción, como lo son autos autónomos, sistemas de diagnóstico médico, sistemas capaces de jugar videojuegos, e incluso sistemas de búsqueda de exoplanetas utilizados por la NASA.

Las redes neuronales representan un subconjunto dentro del conjunto de los algoritmos de *machine learning* o aprendizaje automático, y al igual que los algoritmos de aprendizaje tradicionales, las redes neuronales pueden ser de aprendizaje supervisado o no supervisado:

- **Aprendizaje supervisado**: Los datos de entrenamiento consisten en una dupla que muestra la relación entre las características relevantes o *features* de entrada a analizar y las etiquetas correspondientes a dichos *features*. Este tipo de algoritmos se suelen utilizar en tareas de clasificación o regresión.

- **Aprendizaje no supervisado**: Los datos de entrenamiento contienen únicamente los *features* de entrada a analizar, y la red debe encargarse de aprender su estructura. Este tipo de aprendizaje es utilizado en problemas de *clustering* o en casos en que se desea aprender la distribución de probabilidad que generó los datos de entrada.

Existen también técnicas de aprendizaje semi-supervisado, donde se tienen combinaciones entre ambos tipos de aprendizajes. Por otra parte, dependiendo del tipo de problema que se desee resolver, existen distintas estructuras de redes que permiten aprovechar al máximo distintas características presentes en los datos, algunas de las cuales se detallarán más adelante.

5.2. Arquitecturas comunes

5.2.1. Perceptrón

Como se mencionó anteriormente, el estudio de las redes neuronales comenzó con la creación del primer modelo de neurona por McCulloch y Pitts [24]. Una neurona se encuentra compuesta principalmente por tres elementos básicos, como se puede observar en la figura 5.1:

- Un conjunto de conexiones sinápticas caracterizadas por pesos. Los pesos se suelen denotar como w_{kj}, donde k indica el número de neuronas y j indica el índice de la entrada a dicha neurona. Además de los pesos se añade un término de *bias* b_k, el cual puede ser introducido también como un peso w_{k0} asignado a una entrada extra $x_0 = 1$.
- Un bloque sumador que se encarga de sumar todas las entradas multiplicadas por sus respectivos pesos.
- Una función de activación que se encarga de limitar las posibles salidas de la neurona.

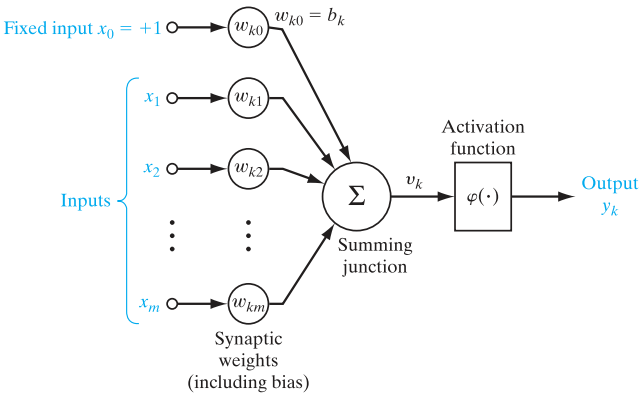


Figura 5.1: Modelo de perceptrón obtenido de [16].

El modelo matemático de esta neurona resulta:

$$y_k = \begin{cases} 1 & \sum_{j=1}^{m} w \cdot x_j \geq b_k \\ 0 & \text{otro caso} \end{cases}$$

Donde el *bias* b_k es un valor fijo, y los pesos w son idénticos para cada una de las entradas. Por otra parte, se puede ver que en este caso la función de activación es un escalón o *threshold* $\varphi(v) = \mathbf{1}\{v \geq 0\}$. Este modelo permitía resolver problemas similares a los de las compuertas lógicas, teniendo el principal problema de ser incapaz de resolver problemas como el de la compuerta XOR, el cual es un problema no linealmente separable.

En 1949 Donald Hebb propuso la conocida **regla de Hebb** [17] que establece:

> "*Cuando el axón de una célula A se encuentra lo suficientemente cerca para excitar a la célula B de forma tal que participa del proceso de activación de ésta, se producen procesos de crecimiento en una o ambas células de manera tal que la eficiencia de A para provocar que B sea disparada se ve incrementada.*"

Esto significa que dos neuronas que disparan juntas reforzarán sus conexiones, mientras que dos neuronas que no lo hacen debilitarán dichas conexiones.

En base a este concepto, en 1958 Rosenblatt dio origen al **perceptrón** [29]. La principal diferencia con el modelo anterior se encuentra en que los pesos y el *bias* dejaron de ser los mismos para cada una de las entradas, ya que se deben reforzar aquellas conexiones que disparan juntas. El modelo matemático del perceptrón entonces se transformó en:

$$\boxed{y_k = \varphi\left(\sum_{j=0}^{m} w_{kj} \cdot x_j\right) \rightarrow \begin{cases} x_0 = 1 \\ w_{k0} = b_k \end{cases}}$$

En la figura 5.1 se tiene una representación de dicho modelo.

Funciones de activación

Mediante este modelo es posible realizar tareas tanto de regresión como de clasificación, siempre que se elijan las funciones de activación adecuadas. Algunas de las funciones de activación más conocidas son:

- **Función lineal**: Se trata de $\varphi(v) = v$, por lo que toma valores entre $-\infty$ y $+\infty$.
- **Función escalón**: La función se puede denotar mediante la función indicadora: $\varphi(x) = \mathbf{1}\{x \geq \theta\}$. Puede ser utilizada para tareas de clasificación de dos clases, y era la utilizada por el modelo clásico de neurona.
- **Función sigmoidea**: Matemáticamente esta función es:

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

  Se puede ver que su salida varía entre 0 y 1, y lo hace de manera suave. Una ventaja de esta función es que su derivada puede ser escrita a partir de ella misma:

  $$\sigma'(x) = \frac{e^{-x}}{(1 + e^{-x})^2} = \frac{1}{1 + e^{-x}}\left(1 - \frac{1}{1 + e^{-x}}\right) = \sigma(x)(1 - \sigma(x))$$

  Esta propiedad permite que algoritmos como el de *backpropagation* puedan ser calculados de manera eficiente.
- **Función tanh**: Esta función toma valores en el intervalo $(-1, 1)$, es suave y su derivada también puede ser obtenida a partir de ella misma:

  $$\varphi(x) = \tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \rightarrow \varphi'(x) = 1 - \varphi(x)^2$$

- **Función lineal rectificada**: Se trata de la función de activación lineal, a la cual se le incorpora una alinealidad:

  $$\varphi(x) = \text{máx}(x, 0)$$

  Esta función es de las más utilizadas en la actualidad ya que su derivada es fácil de calcular, es rápida, y principalmente porque ha demostrado empíricamente que en muchos casos resulta mejor que las demás.
- **Función softmax**: Esta función es la más utilizada cuando se trata de tareas de clasificación, ya que su salida puede ser interpretada como una distribución de probabilidad. Se la suele utilizar para la clasificación entre k clases:

  $$\varphi : \mathbb{R}^k \rightarrow [0, 1]^k \qquad \varphi(x)_j = \frac{e^{x_j}}{\sum_{i=1}^{k} e^{x_i}}$$

Se puede ver que la mayoría de estas funciones resultan alineales. El problema de utilizar funciones de activación lineales es que la salida de la red también resultará lineal (será una combinación lineal de las entradas), lo cual no permite explorar todo el sub-espacio de posibles funciones. Es por esto que se suelen utilizar funciones no lineales, al menos en alguna etapa de la estructura.

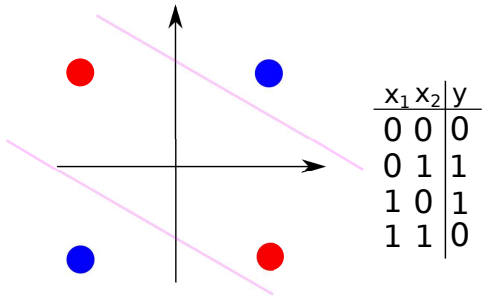
5.2.2. Perceptrón multicapa

En la sección anterior se presentó al perceptrón como unidad básica con la cual era posible realizar tareas sencillas de clasificación o regresión. El ejemplo más sencillo donde este módulo unitario falla es el de la compuerta XOR, que como se puede ver en la figura 5.2 no se trata de un problema linealmente separable. Mediante la utilización de un único perceptrón y dos entradas se puede escribir:

$$v = w_1 x_1 + w_2 x_2 + b \rightarrow y = \begin{cases} 1 & v \geq 0 \\ 0 & v < 0 \end{cases}$$

Se puede ver que dicho clasificador consiste en una recta que intenta separar las clases, pero como se presenta en la figura 5.2, no hay manera de realizarlo mediante una única recta. Es por esta razón que se introduce el modelo de **perceptrón multicapa** el cual simplemente consiste en la agrupación de varios perceptrones, tanto en la misma capa como en diferentes capas, como se presenta en la figura 5.3.

Mediante un perceptrón multicapa el problema de la XOR podría plantearse con una capa oculta de dos neuronas. Una de las neuronas de la capa oculta crearía un clasificador encargado de separar la clase azul de arriba a la derecha de las otras tres muestras, mientras que la otra neurona de la capa oculta separaría la clase azul de abajo a la izquierda. La salida de ambas neuronas se ingresa en una tercera neurona que decide a qué clase corresponde: Si la muestra está entre ambas curvas se trata de la clase roja, mientras que en caso contrario se trata de la clase azul (ver figura 5.2).



x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

Figura 5.2: Ejemplo de una compuerta XOR.

Como se mostró en el ejemplo anterior, el hecho de agregar más capas a la red puede ser interpretado como una abstracción del problema. Al incrementar la cantidad de parámetros, lo que se está haciendo es llevar el problema a un subespacio de mayor dimensionalidad, donde un problema no linealmente separable puede ser transformado en otro que sí lo es.

Cada neurona del perceptrón multicapa se encuentra conectada a todas las neuronas de la capa anterior, y por ende las capas suelen ser conocidas como *dense layers*. La salida de cada capa se puede denotar de manera matricial como:

$$v_k^l = \sum_{j=0}^{m} w_{kj}^l \cdot y_j^{l-1} \rightarrow \boldsymbol{v}^l = \boldsymbol{W}^l \cdot \boldsymbol{y}^{l-1} \rightarrow \boldsymbol{y}^l = \varphi^l\left(\boldsymbol{v}^l\right)$$

donde l indica la capa o *layer* que se está analizando. La salida total de la red entonces será una composición de funciones de la forma:

$$\boldsymbol{y}^L = \varphi^L\left(\boldsymbol{W}^L \cdot \boldsymbol{y}^{L-1}\right) = \varphi^L\left(\boldsymbol{W}^L \cdot \varphi^{L-1}\left(\boldsymbol{W}^{L-1} \cdot \boldsymbol{y}^{L-2}\right)\right) = \varphi^L\left(\boldsymbol{W}^L \cdot \varphi^{L-1}\left(\cdots \boldsymbol{W}^1 \cdot \boldsymbol{x}\right)\right)$$

Se puede ver en esta expresión que si las funciones de activación resultan lineales, a pesar de tener distintas capas la salida resultaría una combinación lineal de las entradas.

De acuerdo con [16] el teorema de aproximación universal establece que:

TEOREMA DE APROXIMACIÓN UNIVERSAL

Sea $\varphi(\cdot)$ una función continua no constante, acotada y monótonamente creciente. Sea I_{m_0} el hipercubo m_0-dimensional definido por $[0,1]^{m_0}$. El subspacio de funciones continuas se denota como $\mathcal{C}(I_{m_0})$. Dada cualquier función $f \in \mathcal{C}(I_{m_0})$ y $\varepsilon > 0$, existe un entero m_1 y un conjunto de constantes reales α_i, b_i, w_{ij} con $i \in \{1, \ldots, m_1\}$ y $j \in \{1, \ldots, m_0\}$ tal que es posible definir

$$F(x_1, \ldots, x_{m_0}) = \sum_{i=1}^{m_1} \alpha_i \cdot \varphi\left(\sum_{j=1}^{m_0} w_{ij} x_j + b_i\right)$$

como una versión aproximada de $f(\cdot)$ para todo $x_1, \ldots, x_{m_0}$, es decir

$$|F(x_1, \ldots, x_{m_0}) - f(x_1, \ldots, x_{m_0})| < \varepsilon$$

Gracias a este teorema se puede observar que mediante un perceptrón multicapa de una única capa oculta, es posible aproximar cualquier tipo de función con un error arbitrario, sin importar qué tan complicada ésta sea. Sin embargo, este teorema indica la existencia de una aproximación, pero no establece ninguna regla para encontrarla, ni tampoco qué tan alcanzable es dicha solución computacionalmente: por ejemplo podría ocurrir que se requieran millones de neuronas en la capa intermedia, haciendo que sea un problema difícil de calcular. Es por esta razón

que suelen utilizarse más capas en las redes neuronales, de forma de poder reproducir funciones más complejas con una menor cantidad de parámetros.

Es importante notar que en este teorema se establece la necesidad de funciones de activación no lineales, lo cual ya había sido mencionado anteriormente.

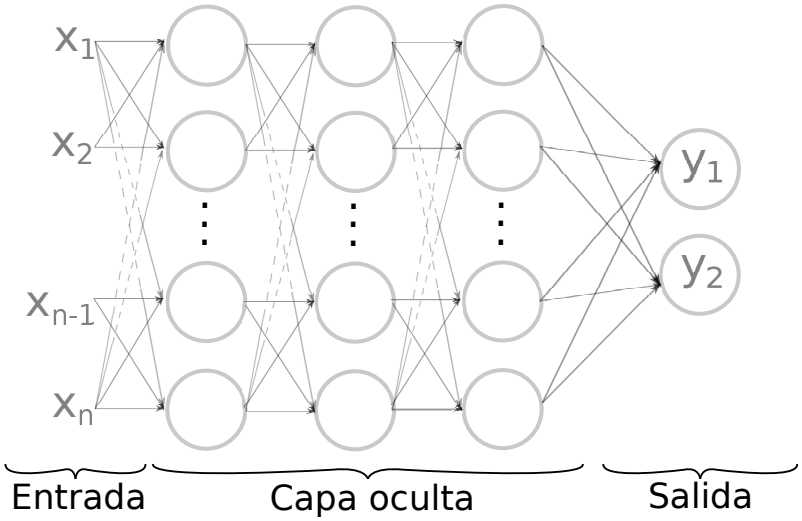


Figura 5.3: Estructura del perceptrón multicapa.

5.2.3. Redes convolucionales

El perceptrón multicapa permite relacionar cada entrada con todas las demás, lo cual facilita la búsqueda de características o *features* dentro de los datos de entrada que no pueden ser observados a simple vista. Por otra parte, al incrementar su profundidad se aumenta su nivel de abstracción, permitiendo la búsqueda de patrones más complejos (e.g. una red de análisis de imágenes intentando detectar si la persona presente en ella se encuentra sonriendo).

Sin embargo, éste tipo de redes no aprovecha la estructura de determinados problemas, ya que asume que todas las entradas se encuentran correlacionadas de alguna manera con las demás. Es por esto que surgieron las redes convolucionales, las cuales permiten analizar a los datos de entrada por regiones, relacionando entradas que se encuentran cercanas[2]. Es por esto que suelen ser utilizadas para el análisis de imágenes, donde la intensidad de cada píxel depende generalmente de los píxeles cercanos.

Este tipo de red fue propuesta en 1988 por Kunihiko Fukushima [9] pero no fue sino hasta 1998 que ganaron popularidad, ya que no se tenían recursos computacionales suficientes ni estaban del todo establecidos los algoritmos de gradiente descendente.

El principio de funcionamiento de las redes convolucionales, como su nombre lo indica, está basado en una operación de convolución. En lugar de tener una gran cantidad de neuronas, se definen unas submatrices conocidas como ***kernels***, cuyas dimensiones suelen ser de 5×5, 3×3 o 1×1[3]. El proceso se puede resumir de la siguiente manera:

1. Se aplica el *kernel* de $n \times n$ al primer conjunto de muestras. Para esto se multiplica elemento por elemento contra una submatriz también de tamaño $n \times n$ de la entrada.

2. A la matriz de tamaño $n \times n$ que se obtiene como resultado, se le aplica una transformación, conocida como ***pooling***, donde se reduce la dimensionalidad de acuerdo con algún criterio (se retiene el máximo de la submatriz, el mínimo, el promedio, etc.).

3. El resultado de la capa de *pooling* es agregado a la matriz de salida en la posición que se corresponde con el desplazamiento del *kernel*.

4. Se desplaza el *kernel* y se repite el proceso.

[2]Los datos de entrada suelen estar agrupados en vectores, matrices o demás estructuras similares. Se suele utilizar el término *cercanas* para indicar que sus índices dentro de la estructura de datos son similares.

[3]Los *kernels* de tamaño 1×1 hoy en día se encuentran muy utilizados en las conocidas ***residual networks***.

Se puede ver que el mecanismo descrito tiene una gran similitud con una convolución, donde el *kernel* barre la matriz de entrada y presenta a su salida una nueva matriz de menor dimensión, pero donde la información fue destacada por sobre la redundancia. En general en las redes convolucionales se suelen aplicar varios *kernels* diferentes a una misma imagen, por lo que a la salida de cada capa se tiene una nueva matriz de menor dimensión por cada *kernel* aplicado.

Mediante el diseño de los distintos *kernels* es posible identificar distintos patrones en las imágenes de entrada: líneas verticales, líneas horizontales, esquinas, etc. Sin embargo, dado que en general no se conoce cuál es la mejor representación de la información disponible, los *kernels* suelen ser inicializados de manera aleatoria y entrenados junto con el resto de la red, nuevamente permitiendo que la red encuentre la representación más conveniente para resolver el problema.

Dado que la salida de las redes convolucionales es una matriz, es necesario agregar perceptrones multicapa para cumplir con las tareas de regresión o clasificación. Dado que el entrenamiento se realiza con toda la red completa, las capas convolucionales logran aprender representaciones más fácilmente separables por el perceptrón multicapa de salida. Las redes convolucionales han demostrado ser altamente eficaces en tareas como las de clasificación e identificación de imágenes.

Algunas de las principales ventajas de esta estructura son:

- Presentan muchos menos pesos que los necesarios para un perceptrón multicapa, por lo que resultan más fáciles de optimizar. El hecho de utilizar el mismo *kernel* para toda la matriz (imagen) de entrada se lo suele conocer como **compartición de pesos**.
- Suelen ser más robustas ante el problema de gradiente evanescente.

En la figura 5.4 se tiene un diagrama de una red convolucional de dos capas, con una etapa de clasificación a la salida.

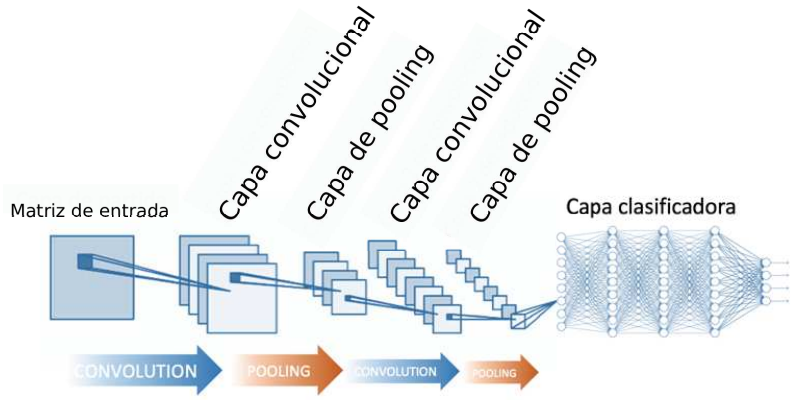


Figura 5.4: Diagrama en bloques de una red convolucional utilizada para tareas de clasificación.

5.2.4. Redes recurrentes

Así como existen las redes convolucionales capaces de aprovechar la estructura de la información en las imágenes, existen por otro lado las redes recurrentes capaces de introducir memoria en el aprendizaje, lo cual es una cualidad muy conveniente para el análisis de secuencias de datos. Mediante estas redes es posible analizar secuencias correlacionadas temporalmente, y por esa razón es que resulta ser la estructura más utilizada para reconocimiento de habla.

La estructura de las redes recurrentes consiste en un ciclo que recorre la secuencia de entrada, donde se van pesando tanto las entradas nuevas como las pasadas. Esto se ejemplifica en la figura 5.5, donde se observa que el flujo de información recurrente puede ser visualizado como una secuencia de copias de la misma red a la que se van ingresando las entradas en distintos instantes.

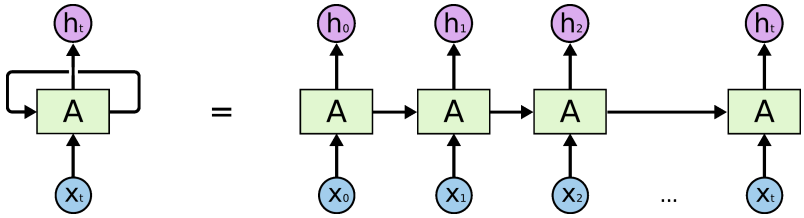


Figura 5.5: Esquema de una red recurrente desdoblada en el tiempo. La imagen fue obtenida de [26].

Se puede ver también en dicha figura que la información es transferida instante a instante, lo cual permite que este tipo de redes sean capaces de capturar dependencias a largo plazo (*long-term dependencies*). Sin embargo, se ha observado que a medida que las dependencias temporales se encuentran más alejadas, las redes recurrentes convencionales (conocidas también como *vanilla*) resultan difíciles de ser entrenadas [4]. Otro gran problema que presentan este tipo de redes, es que son muy difíciles de entrenar, ya que debido a su estructura recurrente la propagación del gradiente para actualizar sus parámetros puede llevar a problemas como *vanishing* o *exploding gradient*[4]:

$$\frac{\partial L_t}{\partial h_{t-k}} \propto \left(W_h^T\right)^k \rightarrow \begin{cases} L_t: \text{ Función de costo en } t \\ h_t: \text{ Vector de memoria de la RNN} \\ W_h: \text{ Matriz de pesos de la RNN} \end{cases}$$

Se puede ver entonces que al incrementar la longitud de la secuencia, la derivada de la función de costo puede tanto crecer como disminuir de manera exponencial. Es por estas razones que surgieron nuevas estructuras recurrentes que hoy en día suelen ser las más utilizadas.

Redes LSTM

En 1997 se introdujeron las redes recurrentes conocidas como ***Long Short-Term Memory (LSTM)*** [18], las cuales hoy en día resultan ser las más utilizadas. Una de las principales razones por las que las *vanilla RNN (VRNN)* tienen dificultad de aprender dependencias *long-term* es que recuerdan toda la información pasada. Es por esto que las redes LSTM introducen la posibilidad de olvidar información pasada, haciendo que sea más eficiente la correlación solo con información relevante.

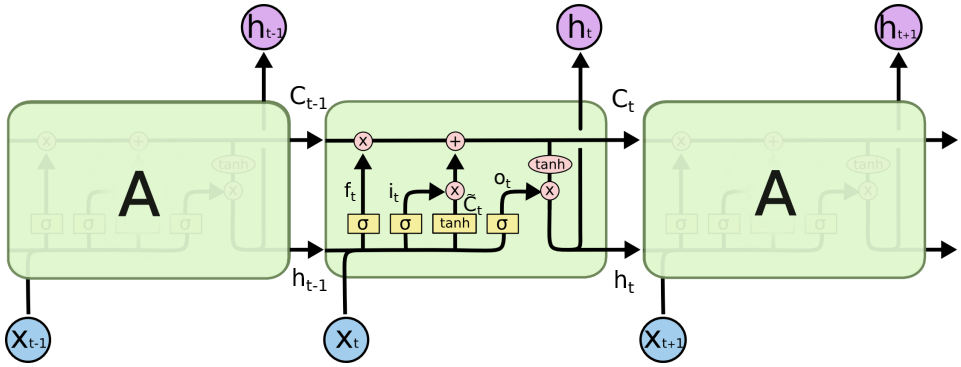


Figura 5.6: Celda LSTM desdoblada. La imagen fue obtenida de [26].

Como se observa en la figura 5.6 las redes LSTM están compuestas por distintas capas, las cuales se pueden desarrollar de la siguiente manera:

- El **estado de la celda** se denota como $\boldsymbol{C}_t$, el cual depende de los estados pasados, las salidas pasadas y la nueva información de entrada.
- La **salida de la celda** se denota como $\boldsymbol{h}_t$, la cual nuevamente depende de la información pasada y la nueva.

[4] Estos conceptos serán explicados más en detalle en secciones posteriores.

- Para favorecer el aprendizaje de dependencias *long-term* se agrega una etapa conocida como ***forget gate*** f_t. Consiste en una subred neuronal que tiene como entrada a la salida del instante pasado $\boldsymbol{h}_{t-1}$ junto con la entrada actual $\boldsymbol{x}_t$, y presenta una función de activación sigmoidea que genera una salida en el intervalo $[0, 1]$:

$$\boldsymbol{f}_t = \sigma\left(W_f \cdot [\boldsymbol{h}_{t-1}, \boldsymbol{x}_t] + \boldsymbol{b}_f\right) \qquad f_t^i \in [0, 1]$$

  Al multiplicar $\boldsymbol{f}_t$ con el estado anterior $\boldsymbol{C}_{t-1}$, lo que se está haciendo es determinar la cantidad de información del estado anterior que se va a retener para la predicción actual. Por lo tanto, cuando $f_t^i = 1$ significa que se almacenará el estado de forma completa, mientras que cuando $f_t^i = 0$ significa que el estado pasado será olvidado.

- El siguiente paso es determinar qué nueva información se debe resaltar. Para esto se crea una nueva subred encargada de determinar cuánto peso se le dará a la nueva información $\boldsymbol{i}_t$ y otra encargada de introducir dicha información $\tilde{\boldsymbol{C}}_t$:

$$\begin{cases} \boldsymbol{i}_t = \sigma\left(W_i \cdot [\boldsymbol{h}_{t-1}, \boldsymbol{x}_t] + \boldsymbol{b}_i\right) & i_t^i \in [0, 1] \\ \tilde{\boldsymbol{C}}_t = \tanh\left(W_C \cdot [\boldsymbol{h}_{t-1}, \boldsymbol{x}_t] + \boldsymbol{b}_C\right) \end{cases}$$

- El valor obtenido de la actualización debe ser sumado al del estado previo, por lo que la salida resulta[5]:

$$\boldsymbol{C}_t = \underbrace{\boldsymbol{f}_t \cdot \boldsymbol{C}_{t-1}}_{\text{Inf. recordada}} + \underbrace{\boldsymbol{i}_t \cdot \tilde{\boldsymbol{C}}_t}_{\text{Nueva inf.}}$$

  Durante el entrenamiento se actualizarán los pesos y *bias* $(W_f, \boldsymbol{b}_f, W_i, \boldsymbol{b}_i, W_C, \boldsymbol{b}_C)$ de manera que la red recurrente logre aprender a pesar la nueva información con respecto a la antigua (siempre dependiendo del valor de la entrada).

- Finalmente la salida de la celda $\boldsymbol{h}_t$ se obtiene pesando el nuevo estado $\boldsymbol{C}_t$ mediante una nueva subred que utiliza la salida anterior y la entrada actual:

$$\begin{cases} \boldsymbol{o}_t = \sigma\left(W_o \cdot [\boldsymbol{h}_{t-1}, \boldsymbol{x}_t] + \boldsymbol{b}_o\right) & o_t^i \in [0, 1] \\ \boldsymbol{h}_t = \boldsymbol{o}_t \cdot \tanh\left(\boldsymbol{C}_t\right) \end{cases}$$

Se puede ver que las redes LSTM están compuestas por subredes encargadas de aprender la manera de pesar la información nueva con respecto a la antigua. Existen distintas variantes de este tipo de redes, donde se introducen distintas funciones de activación, nuevas interconexiones entre las subredes, etc., pero en todas el principio de funcionamiento resulta el mismo.

Otro tipo de redes recurrentes muy utilizadas son las conocidas como ***Gated Recurrent Unit (GRU)***, cuya estructura es muy similar a las LSTM con la diferencia de que el estado y el *forgate gate* se encuentran fusionados.

Redes bidireccionales

Hasta ahora se describieron redes recurrentes capaces de correlacionar la nueva información con la información pasada. Sin embargo, en tareas de reconocimiento del habla por ejemplo, la información futura puede ser también relevante para predicciones actuales. Es por esto que en 1997 se introdujeron las redes recurrentes bidireccionales [34], que básicamente se encuentran compuestas por dos redes recurrentes convencionales cuyos flujos de información circulan en sentidos opuestos. Esto se encuentra ejemplificado en la figura

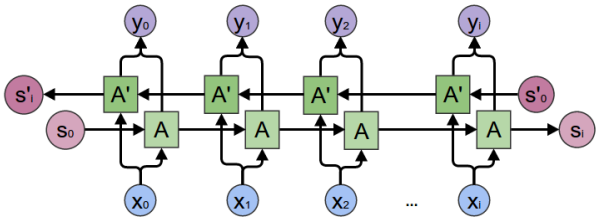


Figura 5.7: Diagrama de una red recurrente bidireccional desdoblada. La imagen fue obtenida de [26].

[5] El producto entre vectores se realiza componente a componente.

5.7, donde se observa que una red se desdobla desde el comienzo de la secuencia hasta el final, mientras la otra lo hace desde el final hasta el inicio.

La principal desventaja de este tipo de redes es que requieren conocer previamente la secuencia de manera completa, lo cual obliga a agregar sistemas de segmentación de la información en casos donde ésta llega de manera continua. Sin embargo, se ha comprobado que las redes bidireccionales resultan ser más efectivas que las convencionales. Una razón de esto podría ser que al analizar las secuencias de manera completa, se tiene un mayor entendimiento del contexto completo de la secuencia, haciendo que las predicciones sean más coherentes: por ejemplo en tareas de NLP (*Natural Language Processing*), dos oraciones que comienzan igual pueden tener connotaciones completamente diferentes, lo cual solo puede ser sabido si se tiene información futura.

5.3. Funciones de costo

Para poder entrenar una red neuronal es necesario establecer una métrica que permita evaluar su funcionamiento, lo cual comúnmente es conocido como **función de costo** o **función de pérdida**. Por otra parte, como se presentará más adelante, la actualización del modelo de la red (es decir de sus pesos) se realiza mediante algoritmos que buscan encontrar aquellas modificaciones que se mueven en el sentido descendente del gradiente buscando alcanzar el mínimo de la función de costo.

Las funciones de costo permiten medir qué tan malo resulta el modelo ajustando la relación entre los *features* de entrada y las salidas deseadas, y es por esta razón que se busca minimizarla. Estas funciones no solo dependen de los datos de entrenamiento (*features* y etiquetas), sino que también dependen fuertemente de la estructura del modelo, es decir que matemáticamente se pueden expresar como:

$$\text{Función de costo} \rightarrow J\left(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{w}, \boldsymbol{b}\right) \rightarrow \begin{cases} \boldsymbol{x}: \ \textit{Features} \\ \boldsymbol{y}: \ \text{Etiquetas} \\ \boldsymbol{w}: \ \text{Pesos} \\ \boldsymbol{b}: \ \textit{Bias} \end{cases}$$

A continuación se presentarán algunas de las funciones de costo más utilizadas.

5.3.1. Error cuadrático medio

El error cuadrático medio se lo utiliza en casos de **regresión** donde se intenta aproximar una función. Esta función consiste básicamente en el promedio del cuadrado del error entre las predicciones y los valores reales:

$$\boxed{J\left(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{\theta}\right) = \frac{1}{N} \sum_{i=1}^{N} \left(h\left(\boldsymbol{x}_i, \boldsymbol{\theta}\right) - \boldsymbol{y}_i\right)^2}$$

donde $\boldsymbol{\theta}$ son los parámetros de la red: pesos y *bias*.

5.3.2. *Cross-entropy*

A diferencia del caso anterior, esta función se utiliza en problemas de **clasificación**, donde se tienen k salidas para k clases mutuamente excluyentes, y cada salida puede ser interpretada como la probabilidad de que dicha clase sea la correcta.

Claude Shannon en 1948 dio vida al concepto de **teoría de la información** [35], donde se trabajó sobre el problema de transmisión eficiente de mensajes entre una fuente y un receptor.

La unidad de medida establecida resultó el *bit*, y se demostró que en una transmisión no todos los bits resultan útiles, ya que muchos son redundantes, se producen errores, etc. Cuando se trabaja con problemas de clasificación donde todas las clases son equiprobables, es fácil obtener la cantidad de bits necesarios para codificar la información de ese problema:

$$\begin{cases} N: \ \text{Cantidad de clases} \\ b: \ \text{Cantidad de bits} \end{cases} \rightarrow \begin{cases} N = 2^b \\ b = \log_2\left(N\right) \end{cases}$$

Sin embargo, cuando las probabilidades no resultan iguales, resulta evidente que es más eficiente codificar aquellos eventos más probables con menos bits, mientras que a los eventos raros se los puede codificar con más bits. Siendo y_i la probabilidad del evento i-ésimo, la cantidad de bits dedicados para dicho evento se puede obtener haciendo $b = \log_2\left(\frac{1}{y_i}\right) = -\log_2(y_i)$[6]. Finalmente la cantidad de información promedio que se observará se puede obtener tomando la esperanza de la cantidad de información de cada una de las clases:

$$\boxed{H(\boldsymbol{y}) = -\sum_{i=1}^{N} y_i \log_2(y_i)}$$

La expresión hallada es conocida como la **entropía**, y da una idea de la cantidad de información contenida en una variable. Se puede ver que esta expresión alcanza su máximo cuando todas las clases son equiprobables.

Utilizando esta estrategia es posible asignar representaciones óptimas a cada una de las clases, dependiendo de su probabilidad de ocurrencia. Sin embargo, la distribución de las clases es lo que se desea aprender mediante la red neuronal, por lo que la cantidad de bits asignados a la clase i-ésima no se obtendrán a partir de la verdadera distribución $\boldsymbol{y}$, sino que será a partir de la distribución estimada $\hat{\boldsymbol{y}}$ como $\hat{b}_i = -\log_2(\hat{y}_i)$. Dado que las observaciones de las clases sí ocurren de acuerdo con la distribución real $\boldsymbol{y}$, se define a la **cross-entropy** como:

$$\boxed{H(\boldsymbol{y}, \hat{\boldsymbol{y}}) = -\sum_{i=1}^{N} y_i \log_2(\hat{y}_i)}$$

Se puede ver que si la distribución estimada coincide con la real, la cross-entropy es igual que la entropía, mientras que si son diferentes la cross-entropy siempre resultará mayor. La diferencia entre estas dos magnitides es conocida como la **distancia de Kullback-Leibler** $D_{KL}(\boldsymbol{y}||\hat{\boldsymbol{y}})$, e indica la cantidad en bits de exceso utilizados en la codificación por estar usando una distribución aproximada en lugar de la real:

$$D_{KL}(\boldsymbol{y}||\hat{\boldsymbol{y}}) = H(\boldsymbol{y}, \hat{\boldsymbol{y}}) - H(\boldsymbol{y})$$

Para aplicar esta función de costo a problemas de clasificación de redes neuronales se procede de la siguiente manera:

- Se codifica a las etiquetas del conjunto de entrenamiento en un formato conocido como *one hot*: Para N clases se utilizan vectores de dimensión N, que tienen un uno en la clase correcta, y ceros en las demás clases. Esta codificación representa a la distribución de probabilidades real $\boldsymbol{y}$.
- Para N clases la red neuronal debe tener N salidas, las cuales deben representar una distribución de probabilidad (es decir que se debe utilizar una función de activación *softmax*). La predicción de la red entonces es la distribución estimada $\hat{\boldsymbol{y}}$.
- Utilizando ambas distribuciones se computa la cross-entropy.

Se puede ver entonces que minimizar la cross-entropy pretende aprender la verdadera distribución de los datos.

5.4. *Backpropagation*

Si bien existen muchas estrategias de optimización, generalmente se encuentran basadas en lo mismo: **gradiente descendente**. Como se mencionó previamente, para entrenar una red neuronal se debe establecer una función de costo que dependerá de la estructura de la red, y que al minimizarla se estará aproximando al modelo que se desea aprender. La manera más comúnmente utilizada para minimizarla es realizando modificaciones en el modelo de la red en el sentido en que el gradiente se hace negativo (ver figura 5.8) de forma tal de acercarse a un mínimo[7]. Esto se realiza de manera iterativa, donde la velocidad de modificación de los pesos se regula con un parámetro conocido como *learning rate* γ:

$$\boldsymbol{w}' = \boldsymbol{w} - \gamma \nabla J(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{w}, \boldsymbol{b})$$

[6] Se puede ver que para el caso equiprobable se tiene que $y_i = \frac{1}{N}$, dando como resultado la expresión anterior.
[7] Si bien es deseable alcanzar el mínimo global, generalmente existen mínimos locales.

Sin embargo, una red neuronal puede estar compuesta por miles de neuronas, haciendo que la cantidad de parámetros sobre los que se debe calcular el gradiente sea enorme. Es por esta razón que se introduce el algoritmo conocido como **backpropagation** para poder realizar esta serie de actualizaciones de manera iterativa.

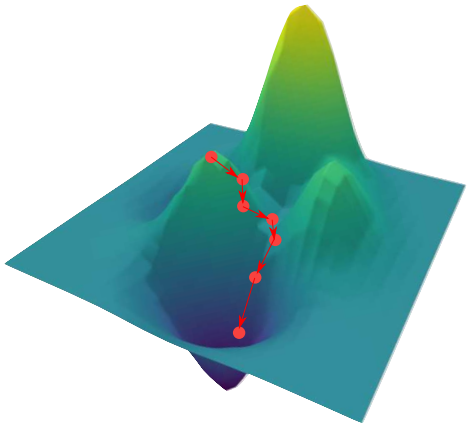


Figura 5.8: Ejemplo del funcionamiento de las estrategias de gradiente descendente.

El objetivo de este algoritmo es el de obtener un método de cálculo de las derivadas parciales de la función de costo en función de los parámetros de la red, sin la necesidad de realizar las derivadas completas.

Para poder desarrollar el algoritmo de *backpropagation* se debe establecer la siguiente nomenclatura, la cual se encuentra ejemplificada en la figura 5.9:

- Se define w^l_{jk} como el peso que conecta la k-ésima neurona en $(l-1)$-ésima capa con la j-ésima neurona en la l-ésima capa.
- Se define b^l_j como el *bias* de la j-ésima neurona en la l-ésima capa.
- Se define a^l_j como la salida de la j-ésima neurona en la l-ésima capa.
- Se define z^l_j como la entrada pesada (*weighted input*) de la j-ésima neurona en la l-ésima capa.
- Se define δ^l_j como el error de la j-ésima neurona en la l-ésima capa. El error se define como[8]:

$$\delta^l_j = \frac{\partial J}{\partial z^l_j}$$

  donde se puede ver que el error será grande en zonas donde la pendiente de la función de costo es pronunciada, mientras que se anula en los mínimos del costo.

De acuerdo con esta nomenclatura, la salida de la j-ésima neurona de la capa l se obtiene como:

$$a^l_j = \sigma\left(\underbrace{\sum_k w^l_{jk} a^{l-1}_k + b^l_j}_{z^l_j}\right) = \sigma\left(z^l_j\right) \rightarrow \boldsymbol{a}^l = \sigma\left(\underbrace{\boldsymbol{w}^l \cdot \boldsymbol{a}^{l-1} + \boldsymbol{b}^l}_{\boldsymbol{z}^l}\right) = \sigma\left(\boldsymbol{z}^l\right)$$

donde la segunda expresión se encuentra en su formato matricial.

[8] El error podría ser definido de manera diferente, pero se optó por utilizar la convención presentada en [25] ya que simplifica los cálculos.

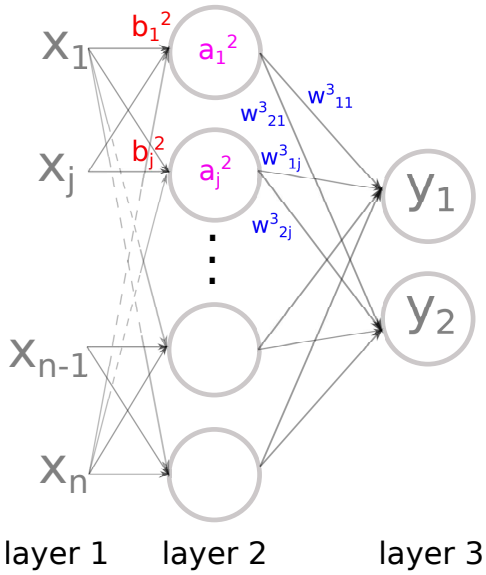


Figura 5.9: Esquema de un MLP con una capa oculta.

El objetivo es encontrar cómo evoluciona la función de costo al introducir modificaciones en los pesos de la red. Para esto, Nielsen [25] establece cuatro ecuaciones fundamentales:

1. El error en la capa de salida: El objetivo es encontrar una expresión que relacione el error en la última capa L, en función de las salidas de la última capa. Para esto se debe aplicar la regla de la cadena, donde se suman las derivadas parciales con respecto a las salidas:

$$\delta_j^L = \frac{\partial J}{\partial z_j^L} = \sum_k \frac{\partial J}{\partial z_j^L} \cdot \frac{\partial a_k^L}{\partial a_k^L} = \sum_k \frac{\partial J}{\partial a_k^L} \cdot \frac{\partial a_k^L}{\partial z_j^L}$$

Dado que la derivada $\frac{\partial a_k^L}{\partial z_j^L}$ se anula para $k \neq j$, la expresión resulta:

$$\boxed{\delta_j^L = \frac{\partial J}{\partial a_j^L} \cdot \frac{\partial a_j^L}{\partial z_j^L} = \frac{\partial J}{\partial a_j^L} \cdot \frac{\partial \sigma\left(z_j^L\right)}{\partial z_j^L} = \frac{\partial J}{\partial a_j^l} \cdot \sigma'\left(z_j^L\right)} \tag{5.1}$$

Se puede ver que aparece el término de la derivada de la función de activación, lo cual explica porqué es deseable utilizar funciones de activación cuyas derivadas son simples de calcular, como se mencionó en la sección 5.2.1.

Esta expresión también puede formularse en su formato matricial, haciendo uso del producto de Hadamard[9]:

$$\boldsymbol{\delta}^L = \nabla_a \boldsymbol{J} \odot \sigma'\left(\boldsymbol{z}^L\right)$$

2. El error de la capa l en función del error de la capa $l+1$: En este caso se busca encontrar una expresión del error en una capa en función de la siguiente, para lo cual se aplica nuevamente la regla de la cadena, pero esta vez se realiza en función de todas las neuronas de la siguiente capa, es decir de todas las conexiones de salida de la j-ésima neurona de la capa l (ver figura 5.9):

$$\delta_j^l = \frac{\partial J}{\partial z_j^l} = \sum_k \frac{\partial J}{\partial z_j^l} \cdot \frac{\partial z_k^{l+1}}{\partial z_k^{l+1}} = \sum_k \underbrace{\frac{\partial J}{\partial z_k^{l+1}}}_{\delta_k^{l+1}} \cdot \frac{\partial z_k^{l+1}}{\partial z_j^l} = \sum_k \delta_k^{l+1} \cdot \frac{\partial z_k^{l+1}}{\partial z_j^l}$$

De esta manera se obtuvo una expresión que da el error en la capa l como una función de todos los errores

[9] El producto de Hadamard $\odot$ es un producto vectorial que se realiza componente a componente: $(s \odot t)_j = s_j \cdot t_j$.

en la capa $l+1$. El segundo término se puede reducir como:

$$\frac{\partial z_k^{l+1}}{\partial z_j^l} = \frac{\partial}{\partial z_j^l}\left(\sum_n w_{kn}^{l+1} a_n^l + b_k^{l+1}\right) = \frac{\partial}{\partial z_j^l}\left(\sum_n w_{kn}^{l+1}\sigma\left(z_n^l\right) + b_k^{l+1}\right) = \sum_n w_{kn}^{l+1}\frac{\partial}{\partial z_j^l}\sigma\left(z_n^l\right) = w_{kj}^{l+1}\sigma'\left(z_j^l\right)$$

donde nuevamente se anularon todos los términos donde $j \neq n$. La expresión final entonces resulta:

$$\boxed{\delta_j^l = \sum_k \delta_k^{l+1} \cdot w_{kj}^{l+1} \cdot \sigma'\left(z_j^l\right)} \tag{5.2}$$

Se puede ver que se trata de la suma de los errores de todas las neuronas de la capa $l+1$ ponderados por el peso que tienen sus conexiones con la neurona j de la capa previa, y también por un término que indica la variación debida a la función de activación.

La expresión vectorial de esta ecuación es:

$$\boldsymbol{\delta}^l = \left(\left(\boldsymbol{w}^{l+1}\right)^T \boldsymbol{\delta}^{l+1}\right) \odot \sigma'\left(\boldsymbol{z}^l\right)$$

3. La variación del costo con respecto al *bias*: Nuevamente se puede aplicar la regla de la cadena, introduciendo la derivada sobre todas las salidas:

$$\frac{\partial J}{\partial b_j^l} = \sum_k \frac{\partial J}{\partial b_j^l} \cdot \frac{\partial z_k^l}{\partial z_k^l} = \sum_k \underbrace{\frac{\partial J}{\partial z_k^l}}_{\delta_k^l} \cdot \frac{\partial z_k^l}{\partial b_j^l} = \sum_k \delta_k^l \cdot \frac{\partial}{\partial b_j^l}\left(\sum_n w_{kn}^l a_n^{l-1} + b_k^l\right)$$

Se puede ver que la derivada solo es distinta de cero cuando $j = k$, es decir que la expresión resulta:

$$\boxed{\frac{\partial J}{\partial b_j^l} = \sum_k \delta_k^l \cdot \frac{\partial b_k^l}{\partial b_j^l} = \delta_j^l} \tag{5.3}$$

De esta forma, se obtiene que la variación del costo con respecto al *bias* coincide con la variación del costo con respecto a la salida z_j^l.

4. La variación del costo con respecto a los pesos: Se debe proceder de la misma manera que para el caso del *bias*:

$$\frac{\partial J}{\partial w_{jk}^l} = \sum_n \frac{\partial J}{\partial z_n^l} \cdot \frac{\partial z_n^l}{\partial w_{jk}^l} = \sum_n \delta_n^l \cdot \frac{\partial z_n^l}{\partial w_{jk}^l} = \sum_n \delta_n^l \cdot \frac{\partial}{\partial w_{jk}^l}\left(\sum_i w_{ni}^l a_i^{l-1} + b_n^l\right) \rightarrow \boxed{\frac{\partial J}{\partial w_{jk}^l} = \delta_j^l a_k^{l-1}} \tag{5.4}$$

Nuevamente se obtiene que la variación depende del error previamente calculado, pero en este caso se encuentra pesado por la activación de la capa previa.

Una vez establecidas las ecuaciones fundamentales, el algoritmo de *backpropagation* se puede desarrollar de la siguiente manera:

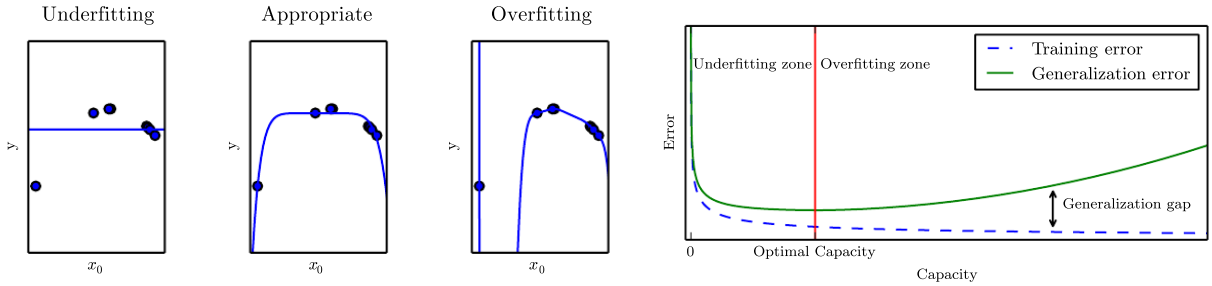
1. Se realiza una ejecución *forward*, es decir que se introducen las entradas $\boldsymbol{x}$ y se calculan las activaciones de todas las neuronas hasta llegar a la salida de la red.
2. Se calcula el error de salida δ_j^L de acuerdo con la ecuación 5.1.
3. Utilizando la ecuación 5.2 se propagan de atrás hacia adelante los errores calculados.
4. Una vez propagados los errores por toda la red, los gradientes de la función de costo se pueden obtener utilizando las ecuaciones 5.3 y 5.4.

5. Finalmente se deben actualizar todos los pesos y *bias* de la red de acuerdo con las siguientes expresiones:

$$\begin{cases} w^l_{jk} \to w^l_{jk} - \gamma \frac{\partial J}{\partial w^l_{jk}} \\ b^l_j \to b^l_j - \gamma \frac{\partial J}{\partial b^l_j} \end{cases}$$

5.5. Regularización

Una de las principales reglas que se deben cumplir al entrenar algoritmos de *machine learning* es que el sistema a entrenar jamás debe observar las muestras de testeo o validación durante la etapa de entrenamiento. Lo que se busca no es encontrar aquel algoritmo que mejor ajusta los datos de entrenamiento, sino aquel que logra realizar predicciones sobre nuevos elementos de la mejor manera. Se define entonces a la **generalización** como la habilidad de responder apropiadamente bajo nuevos escenarios no observados durante la etapa de entrenamiento.



(a) Ejemplos de los problemas de *overfitting* y *underfitting* (tomados de [14]). El caso de *underfitting* puede ser pensado como un polinomio incapaz de ajustar los puntos debido a su bajo orden, mientras que el problema de *overfitting* se corresponde con el de un polinomio de un orden lo suficientemente alto como para colocar sus raíces en todas las muestras de entrenamiento.

(b) Error en función de la capacidad (tomado de [14]). Se puede ver que al incrementar la capacidad de la red el sistema tiende a ajustar de manera exacta las muestras de entrenamiento, no así a las muestras de validación, y por ende el error de generalización crece.

Figura 5.10: Ejemplificación del problema de generalización (tomado de [14]).

Para que un sistema generalice correctamente es necesario lograr que éste aprenda o aproxime el modelo generador de esos datos, evitando que el sistema simplemente asocie entradas con salidas sin aprender sus verdaderas relaciones (es decir, que aprenda de memoria). Esto permite introducir una característica conocida como **capacidad**, la cual define la habilidad del sistema de aproximar una gran variedad de funciones. En general, cuanto más compleja sea una red neuronal (es decir cuantos más parámetros tenga) más fácil le resultará aproximar funciones complejas. Un ejemplo de esto fue presentado en la sección 5.2.2, donde fue necesario utilizar tres perceptrones en lugar de uno para resolver un problema no linealmente separable.

Dada la capacidad de una red neuronal, existen dos posibles problemas:

- ***Underfitting***: Este problema ocurre cuando la capacidad de una red neuronal no es lo suficientemente grande como para ajustar las muestras de entrenamiento. En el caso de la XOR de la sección 5.2.2, si se hubiera utilizado un único perceptrón, jamás se habría logrado obtener error nulo sobre las muestras de entrenamiento, sin importar la cantidad de muestras o el tiempo de entrenamiento utilizado. Generalmente este problema se resuelve incrementando la capacidad de la red.
  La generalización en este caso suele ser baja debido a que no se logró ni siquiera aprender las muestras de entrenamiento.

- ***Overfitting***: Por definición este problema ocurre cuando se tiene un error de entrenamiento bajo pero un error de generalización alto. En este caso la capacidad de la red es lo suficientemente alta como para aprender de memoria las muestras de entrenamiento. Este problema también suele ocurrir cuando se tienen pocas muestras de entrenamiento, ya que la red puede resultar incapaz de extraer la información necesaria para aproximar el modelo.
  Si bien a veces es posible solucionar el problema de *overfitting* aumentando la cantidad de muestras (o en

algunos casos reduciendo la capacidad de la red), existen técnicas conocidas como **regularización** que permiten mejorar los resultados.

Ambos problemas se encuentran ejemplificados en la figura 5.10.

Para introducir el concepto de regularización, es importante presentar el siguiente teorema:

***No Free Lunch Theorem* (NFL)**

Este teorema fue demostrado por David Wolpert en 1996 y establece que **cualquier** algoritmo de clasificación tiene el mismo error de generalización, cuando se lo promedia sobre **todas** las posibles distribuciones.

Este teorema entonces implica que no es posible establecer un único algoritmo para todos los posibles problemas, sino que deben diseñarse soluciones particulares para cada problema en particular. Es por esta razón que la solución al problema de generalización no se resume únicamente a aumentar la cantidad de muestras de entrenamiento, sino que deben tomarse en consideración características específicas del problema a resolver para ajustar el modelo a dicho problema.

La regularización entonces se define como *cualquier modificación que se introduce al algoritmo para reducir el error de generalización, pero no el de entrenamiento* [14]. Generalmente la regularización se introduce como un término extra en la función de costo, donde se introducen limitaciones sobre qué tipo de funciones resultan válidas:

- **Regularización de pesos**: La función de regularización resulta $\Omega(\boldsymbol{w}) = ||\boldsymbol{w}||$. Al introducir este término a la función de costo, no solo se está minimizando el costo original, sino que se está minimizando la norma de los pesos de la red. La ventaja de esto es que se evita que existan neuronas con altas activaciones y neuronas con bajas activaciones, de manera que puede decirse que el "conocimiento se encuentra repartido". Generalmente se utilizan dos tipos de variantes de esta regularización:
  - **Regularización L1**: En este caso la función de costo suele ser

    $$J_T = J + \lambda \cdot \sum ||\boldsymbol{w}||$$

    donde λ determina cuánto peso se le da a la regularización. En general esta regularización se utiliza cuando se tienen parámetros irrelevantes y se quiere llevarlos a cero.
  - **Regularización L2**: En este caso la función de costo suele ser

    $$J_T = J + \lambda \cdot \sum ||\boldsymbol{w}||^2$$

    Esta regularización se utiliza cuando se quiere evitar que determinadas neuronas se activen fuertemente a determinadas entradas. Mediante este método se logra que los pesos sean pequeños y que la información se encuentre repartida entre más neuronas.
- ***Dropout***: Esta técnica consiste en eliminar de manera aleatoria un conjunto de neuronas de una capa de la red, durante una iteración de entrenamiento. Luego de dicha iteración, las neuronas son restablecidas y el proceso se repite con otro grupo aleatorio de neuronas. Esto permite reducir la correlación entre neuronas, haciendo que éstas deban ponderar mejor todas sus entradas en lugar de depender de una única entrada. Un hiperparámetro de este método es la probabilidad de que una neurona sea eliminada en cada iteración.
- ***Data augmentation***: Como se mencionó previamente, tener un conjunto pequeño de datos de entrenamiento evita alcanzar buena generalización. Una forma de solucionarlo es incrementando de manera artificial el conjunto de entrenamiento introduciendo ruido en las muestras. En problemas de reconocimiento de habla, además de agregar ruidos gaussianos o aleatorios, se suelen introducir sonidos de fondo.
- ***Batch normalization***: Es una técnica de optimización pero que introduce efectos de regularización. Consiste en un proceso de reparametrización donde se toman las salidas de las neuronas de una capa (sin su respectiva función de activación), se calcula la media y la varianza del *batch*, y se las normaliza:

  $$\boldsymbol{z}' = \frac{\boldsymbol{z} - \boldsymbol{\mu}}{\boldsymbol{\sigma}}$$

La utilización de esta técnica introduce las siguientes ventajas:

- Se normaliza la distribución de los datos. Esto permite reducir los efectos adversos cuando la variabilidad en las muestras de entrada es grande, donde puede haber muestras que fueron tomadas en ambientes más ruidosos que otras.
- Al utilizar algoritmos basados en gradiente descendente, la actualización de los pesos se realiza en base a la derivada de la función de costo. Sin embargo, en dicho cálculo solo se tienen en cuenta las variaciones de primer orden, descartando la influencia que pueden tener las derivadas de orden superior. Esto puede provocar que determinadas actualizaciones realizadas en el sentido del gradiente descendente, resulten en un incremento del costo, lo cual se encuentra ejemplificado en la figura 5.11.
  Si bien esto puede ser evitado ajustando el *learning rate*, existen funciones de costo que presentan zonas de mucha variabilidad y otras donde resultan más suaves, lo cual imposibilita la determinación de un paso óptimo. Dado que mediante *batch normalization* se realiza una normalización, dichos efectos de orden superior se ven reducidos, y por ende permite la utilización de un *learning rate* más elevado reduciendo el riesgo de que ocurran situaciones como las de la figura 5.11.

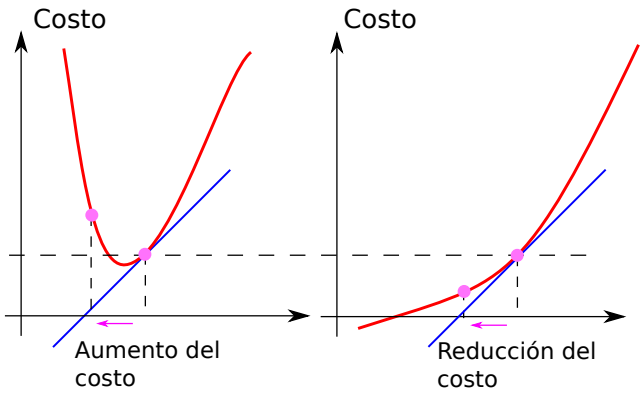


Figura 5.11: Comparación de la evolución del costo al utilizar un mismo valor de *learning rate*, cuando se tienen funciones de costo con distinta influencia de las derivadas de orden superior.

Parte II

Arquitecturas bajo estudio

En las secciones previas se introdujeron los conceptos necesarios para abordar las técnicas implementadas en este trabajo. A continuación se presenta el marco teórico de las arquitecturas aquí estudiadas.

6. Redes CTC (*Connectionist Temporal Classification*)

Este algoritmo fue introducido en 2006 por Alex Graves [12]. Previo a la utilización de este método, los sistemas basados en el reconocimiento de secuencias requerían que las secuencias de entrenamiento se encontraran alineadas con las secuencias de etiquetas. Esto no solo requiere mucho tiempo de preprocesamiento de los datos, sino que en muchos casos puede ser una tarea muy difícil de realizar correctamente (e.g. el etiquetado de texto manuscrito en cursiva).

6.1. Descripción

El algoritmo CTC permite definir una función de costo para entrenar redes neuronales cuyos datos de entrada no se encuentran alineados. El problema que se intenta resolver es encontrar la manera de mapear una secuencia de entrada $\boldsymbol{x} \in \mathbb{R}^{m \times T}$ con m *features* por cada instante de tiempo[10] $t \in T$, a una secuencia de salida $\boldsymbol{z} \in \mathbb{R}^{n \times U}$ con n *features* y longitud $u \in U$, **bajo la restricción de que $\boldsymbol{U} \leq \boldsymbol{T}$**. Dado que en general U y T no resultan iguales, no existe una manera *a priori* de alinear ambas secuencias. En la figura 6.1 se tiene un ejemplo del problema que se debe resolver mediante esta técnica.

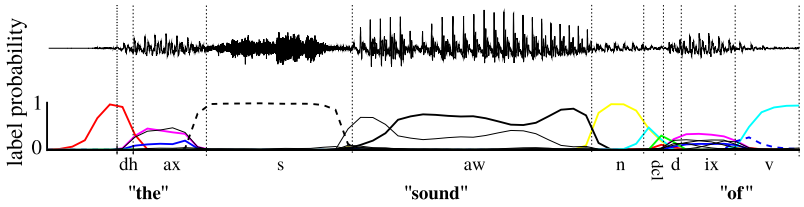


Figura 6.1: Visualización del problema de alineación entre las señales de habla y las transcripciones (obtenido de [12]).

Sea L el alfabeto finito de *labels* o etiquetas, se espera obtener a la salida del sistema una secuencia de símbolos pertenecientes a L que minimicen el error con respecto a la secuencia verdadera. Debido a la naturaleza del problema se suelen utilizar redes recurrentes cuyas salidas tiene una función de activación **softmax**, de manera que en cada instante de tiempo se obtiene la probabilidad de cada uno de los símbolos incluidos en L, formando así un clasificador que opera sobre secuencias.

Sin embargo, la salida de una red recurrente presenta el mismo tamaño en la dimensión temporal que la entrada, por lo que requeriría que las secuencias de entrenamiento estuvieran alineadas. Por esta razón, se introduce un nuevo símbolo llamado ***blank*** mediante el cual se representarán las transiciones de un símbolo a otro. El nuevo modelo de red entonces operará sobre el alfabeto $L' = L \cup \{blank\}$ y deberá aprender a emitir el símbolo *blank* para especificar transiciones entre símbolos.

Se define entonces a $\boldsymbol{y} = \mathcal{N}_w(\boldsymbol{x})$, con $\mathcal{N}_w : \mathbb{R}^{m \times T} \to \mathbb{R}^{n \times T}$, como la salida de la red neuronal, donde y_k^t es interpretada como la probabilidad de observar la etiqueta k en el instante t. Asumiendo que las salidas en distintos instantes de tiempo son condicionalmente independientes dado el estado interno de la red, la distribución de las secuencias π[11] definidas sobre el alfabeto L' se puede definir como el producto de las probabilidades de cada símbolo para todo instante de tiempo:

$$\boxed{p(\boldsymbol{\pi}|\boldsymbol{x}) = \prod_{t=1}^{T} y_{\pi_t}^t \qquad \forall \boldsymbol{\pi} \in (L')^T} \tag{6.1}$$

[10] Esta dimensión no necesariamente tiene que ser el tiempo. Lo importante es remarcar que se trata de secuencias de datos.
[11] Estas secuencias suelen ser denominadas como ***paths***.

donde $\forall \boldsymbol{\pi} \in (L')^T$ denota a todas aquellas secuencias de símbolos incluidos en L' que tienen una longitud T.

Una vez obtenida la secuencia de salida $\boldsymbol{y}$ de longitud T se la debe alinear con la secuencia real. Para esto se define una función $\mathcal{B}$ del tipo *many-to-one* mediante la cual se mapea la secuencia de salida $\boldsymbol{y}$ a otra de menor longitud mediante el siguiente criterio:

- Se eliminan todas las repeticiones de etiquetas. Un ejemplo de esto puede ser: $\mathcal{B}(aaaabbbb) = \mathcal{B}(aaab) = \ldots = ab$.
- Si dos etiquetas iguales se encuentran separadas por un *blank*, entonces se consideran símbolos diferentes. Un ejemplo de esto puede ser: $\mathcal{B}(a - ab-) = \mathcal{B}(-aa - aaabbb - -) = \mathcal{B}(-a - a - b-) = aab$.

De esta manera, la red deberá aprender a emitir versiones de las secuencias verdaderas donde aquellos sonidos largos son traducidos a secuencias de símbolos repetidos, los cuales serán luego descartados y no se tendrán en cuenta a la hora de medir el rendimiento del sistema.

Aplicando esta técnica entonces es posible encontrar la función *likelihood* que mide la probabilidad de una determinada secuencia dadas las muestras de entrada. Para lograrlo, se debe sumar la probabilidad de todas aquellas secuencias a las que al aplicarle la transformación $\mathcal{B}$, resultan mapeadas a la secuencia real. La expresión para el *likelihood* entonces resulta:

$$\boxed{p(\boldsymbol{l}|\boldsymbol{x}) = \sum_{\boldsymbol{\pi} \in \mathcal{B}^{-1}(\boldsymbol{l})} p(\boldsymbol{\pi}|\boldsymbol{x})} \tag{6.2}$$

Se puede ver que para encontrar una solución a 6.2 es necesario conocer todos los *paths* $\boldsymbol{\pi}$ tales que $\mathcal{B}(\boldsymbol{\pi}) = \boldsymbol{l}$, lo cual puede ser un conjunto extremadamente grande. Si bien esta tarea presenta una alta complejidad, puede ser resuelta mediante el algoritmo ***forward-backward*** [28] presentado en la sección 3.2.1.

6.1.1. Algoritmo *forward-backward* para CTC

La variable *forward* definida en la ecuación 3.4 se reescribe en este caso como:

$$\alpha_t(s) = \sum_{\pi \in N^T : \mathcal{B}(\pi_{1:t}) = \boldsymbol{l}_{1:s}} \prod_{t'=1}^{t} y_{\pi_{t'}}^{t'}$$

Esta expresión puede ser interpretada de la siguiente manera:

- Se define $\pi \in N^T$ como al conjunto de secuencias de longitud T a las cuales en caso de aplicar la función de mapeo $\mathcal{B}$ a la subsecuencia de longitud t, se obtiene la secuencia de salida $\boldsymbol{l}_{1:s}$. Esto significa que se tomarán todos aquellos *paths* cuyos primeros t valores sean mapeados a los primeros s valores en la secuencia real.
- El término correspondiente a la productoria representa a la probabilidad $p(\boldsymbol{\pi}_{1:t}|\boldsymbol{x})$. De esta manera, para cada *path* en N^T se obtiene la probabilidad para los primeros t símbolos.

De acuerdo con esto, entonces, la expresión de $\alpha_t(s)$ consiste en la suma de las probabilidades de cada *path* cuyos primeros t símbolos son mapeados a los primeros s símbolos de la salida.

El siguiente paso consiste en encontrar la representación recursiva de $\alpha_t(s)$ para reducir la complejidad del cálculo de $p(\boldsymbol{l}|\boldsymbol{x})$. Dado que las secuencias verdaderas no contienen el símbolo *blank* pero éste es una clase válida dentro del modelo de la red neuronal, es necesario modificar la secuencia verdadera. Lo que se hace entonces es crear una nueva secuencia $\boldsymbol{l}'$ que se obtiene agregando un *blank* al comienzo y al final de $\boldsymbol{l}$, así como también entre cada uno de los símbolos. Por ejemplo si $\boldsymbol{l} = \{a, b, c\}$, la secuencia transformada será $\boldsymbol{l}' = \{-, a, -, b, -, c, -\}$. Se puede ver que la longitud de la secuencia transformada es $|\boldsymbol{l}'| = 2|\boldsymbol{l}| + 1$.

Una vez armada la secuencia $\boldsymbol{l}'$, la recursividad *forward* puede visualizarse utilizando un diagrama como el de la figura 6.2.

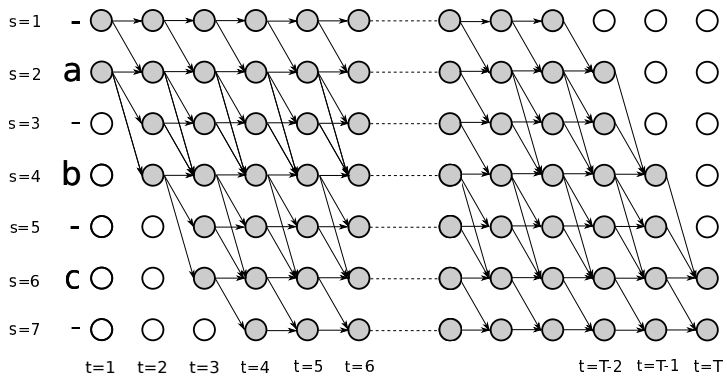


Figura 6.2: Diagrama para visualizar el armado de la expresión recursiva de los $\alpha_t(s)$.

Siguiendo el diagrama de la figura 6.2, el primer paso consiste en inicializar la recursión tomando como posibles símbolos válidos al *blank* y al primer símbolo de $\boldsymbol{l}$. Por lo tanto, la inicialización resulta:

$$\begin{cases} \alpha_1(1) = y_b^1 & \text{Probabilidad del } \textit{blank} \\ \alpha_1(2) = y_{l_1}^1 & \text{Probabilidad del primer símbolo} \\ \alpha_1(s) = 0 & 2 < s \leq |\boldsymbol{l}'| \end{cases}$$

Para avanzar al siguiente paso de la recursión se deben analizar únicamente aquellos *paths* que resultan mapeados a la secuencia real. Es por esto que en la figura 6.3a solo se admiten las transiciones en rojo:

- Si en el instante anterior se estaba en un *blank,* solo se admite una transición al siguiente símbolo o a un nuevo *blank*.
- Si en el instante anterior se estaba en un símbolo, se admite una repetición del símbolo, un *blank*, o una transición a un símbolo **diferente**.

Se puede ver que hay dos maneras de llegar a un *blank* mientras que hay tres maneras de llegar a un símbolo. Por lo tanto, la recursión deberá dividirse en dos casos:

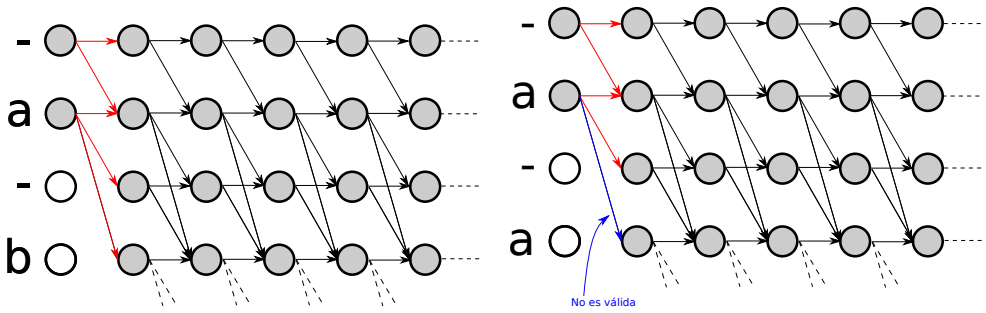
- En el caso de que $\boldsymbol{l}'_s = \textit{blank}$ se deben sumar las probabilidades de no haber cambiado el símbolo anterior (es decir que se permanece en un *blank*) más la probabilidad de haber realizado una transición a partir de $s-1$ (que siempre es diferente de *blank*):

  $$\tilde{\alpha}_t(s) = \alpha_{t-1}(s) + \alpha_{t-1}(s-1)$$

  Esta probabilidad debe ser pesada por la probabilidad de encontrarse en el símbolo s en el instante t: $\alpha_t(s) = \tilde{\alpha}_t(s)\, y_{l'_s}^t$.

- En el caso en que s sea un símbolo diferente de *blank* se debe agregar un posible camino más: una transición entre dos símbolos diferentes también es válida. Por lo tanto, se deben sumar la probabilidad de permanecer en el mismo símbolo, la de provenir de un *blank* (es decir de $s-1$) y la probabilidad de provenir del símbolo anterior (es decir de $s-2$):

  $$\tilde{\alpha}_t(s) = \alpha_{t-1}(s) + \alpha_{t-1}(s-1) + \alpha_{t-1}(s-2) \rightarrow \alpha_t(s) = \tilde{\alpha}_t(s)\, y_{l'_s}^t$$



(a) Visualización del segundo paso de la recursión.

(b) Caso en que se tienen símbolos repetidos.

Figura 6.3: Ampliación del diagrama de la figura 6.2.

Sin embargo, se puede ver que en estas expresiones no se encuentra incluido el caso en que se tenga una transición entre símbolos iguales como sucede en la figura 6.3b. Se puede ver que la transición azul se corresponde con el *path* $\{a, a\}$ que será mapeado a $\mathcal{B}(aa) = a$, y por ende no es un resultado válido. Por esta razón, estos casos son tratados como el caso en que $\boldsymbol{l}'_s = blank$, admitiendo solo transiciones desde un *blank* o a partir de sí mismos.

Agrupando los tres casos la regla recursiva para la variable *forward* resulta:

$$\boxed{\alpha_t(s) = \begin{cases} y^t_{l'_s} \cdot (\alpha_{t-1}(s) + \alpha_{t-1}(s-1)) & (\boldsymbol{l}'_s = blank) \cup (\boldsymbol{l}'_s = \boldsymbol{l}'_{s-2}) \\ y^t_{l'_s} \cdot (\alpha_{t-1}(s) + \alpha_{t-1}(s-1) + \alpha_{t-1}(s-2)) & \text{otro caso} \end{cases}}$$

Finalmente la probabilidad de la secuencia completa se obtiene llegando al final de la recursión:

$$\boxed{p(\boldsymbol{l}|\boldsymbol{x}) = \alpha_T\left(|\boldsymbol{l}'|\right) + \alpha_T\left(|\boldsymbol{l}'| - 1\right)} \tag{6.3}$$

Para obtener la variable *backward* debe procederse de igual manera, pero en este caso comenzando de atrás hacia adelante:

$$\beta_t(s) = \sum_{\pi \in N^T : \mathcal{B}(\pi_{t:T}) = \boldsymbol{l}_{s:|\boldsymbol{l}|}} \prod_{t'=t}^{T} y^{t'}_{\pi_{t'}}$$

Las condiciones iniciales se plantean admitiendo que en el último instante de tiempo se pueda observar un *blank* o un símbolo (ver figura 6.2):

$$\boldsymbol{\beta}_T = \begin{cases} \beta_T\left(|\boldsymbol{l}'|\right) = y^T_b \\ \beta_T\left(|\boldsymbol{l}'| - 1\right) = y^T_{\boldsymbol{l}_{|\boldsymbol{l}|}} \\ \beta_T(s) = 0 \qquad\qquad s < |\boldsymbol{l}'| - 1 \end{cases}$$

Siguiendo el mismo razonamiento que para el caso *forward*, las fórmulas de la recursión *backward* resultan:

$$\boxed{\beta_t(s) = \begin{cases} y^t_{l'_s} \cdot (\beta_{t+1}(s) + \beta_{t+1}(s+1)) & (\boldsymbol{l}'_s = blank) \cup (\boldsymbol{l}'_s = \boldsymbol{l}'_{s+2}) \\ y^t_{l'_s} \cdot (\beta_{t+1}(s) + \beta_{t+1}(s+1) + \beta_{t+1}(s+2)) & \text{Otro caso} \end{cases}}$$

Recordando la ecuación 3.6, el producto de las variables *forward* y *backward* resulta en la probabilidad de todos los caminos que pasan por el símbolo s en el instante t. Utilizando la notación introducida en esta sección, se puede expresar de forma matemática como:

$$\alpha_t(s)\,\beta_t(s) = \sum_{\pi \in \mathcal{B}^{-1}(\boldsymbol{l}):\, \pi_t = \boldsymbol{l}_s} y^t_{\boldsymbol{l}_s} \prod_{t=1}^{T} y^t_{\pi_t} = \sum_{\pi \in \mathcal{B}^{-1}(\boldsymbol{l}):\, \pi_t = \boldsymbol{l}_s} y^t_{\boldsymbol{l}_s} \cdot p(\pi|\boldsymbol{x})$$

El término de la productoria es el que se puede ver en la ecuación 6.1, y define la probabilidad de una determinada secuencia. Por otra parte, dicha probabilidad es multiplicada por la probabilidad de que ocurra el símbolo s en el instante t. Finalmente se suman todos los posibles paths π que al aplicarles la función de mapeo $\mathcal{B}$ son transformados

en $\boldsymbol{l}$ y que en pasan por el símbolo $\boldsymbol{l}_s$ en el instante t. Nuevamente se puede ver que se trata de una tarea que requiere gran complejidad computacional, ya que se deben conocer todos las posibles secuencias.

Dado que la probabilidad $y^t_{\boldsymbol{l}_s}$ no depende de la sumatoria, se puede despejar:

$$\alpha_t(s)\,\beta_t(s) = y^t_{l_s} \sum_{\pi\in\mathcal{B}^{-1}(\boldsymbol{l}):\,\pi_t=\boldsymbol{l}_s} p(\pi|\boldsymbol{x}) \rightarrow \boxed{\frac{\alpha_t(s)\,\beta_t(s)}{y^t_{\boldsymbol{l}_s}} = \sum_{\pi\in\mathcal{B}^{-1}(\boldsymbol{l}):\,\pi_t=\boldsymbol{l}_s} p(\pi|\boldsymbol{x})}$$

Se puede ver que la expresión obtenida es similar a la de la ecuación 6.2, pero con la restricción de que es la probabilidad debida a todas las secuencias que pasan por el símbolo $\boldsymbol{l}_s$ en el instante t. Por esta razón, se deben sumar para cada instante de tiempo las probabilidades para cada posible s:

$$\boxed{p(\boldsymbol{l}|\boldsymbol{x}) = \sum_{t=1}^{T}\sum_{s=1}^{|\boldsymbol{l}|}\frac{\alpha_t(s)\,\beta_t(s)}{y^t_{l_s}}}$$

Se obtuvo entonces una expresión para el *likelihood* de manera recursiva a partir de las variables *forward* y *backward*.

6.2. Maximización del *likelihood*

La función de costo para redes CTC se encuentra basada en la maximización del *likelihood* (o minimización del *log-likelihood*) sobre las muestras de entrenamiento. Siendo $\boldsymbol{S} = \{(\boldsymbol{x},\boldsymbol{l})_i\}\rfloor_{i=1}^{N}$ el conjunto de secuencias de entrenamiento y $\mathcal{N}_w$ la función que representa a la red neuronal con su salida softmax, la función de costo que se desea minimizar resulta:

$$O^{ML}(\boldsymbol{S},\mathcal{N}_w) = -\sum_{(\boldsymbol{x},\boldsymbol{l})\in S}\ln\left(p(\boldsymbol{l}|\boldsymbol{x})\right)$$

El entrenamiento de la red neuronal se realiza mediante el algoritmo de *backpropagation* por lo que debe derivarse la función de costo. Dado que las muestras del conjunto de entrenamiento fueron tomadas de manera independiente, es posible analizar a las muestras por separado en lugar de todo el conjunto:

$$\frac{\partial O^{ML}(\{(\boldsymbol{x},\boldsymbol{l})\},\mathcal{N}_w)}{\partial y^t_k} = -\frac{\partial\ln\left(p(\boldsymbol{l}|\boldsymbol{x})\right)}{\partial y^t_k}$$

Para poder aplicar la derivada respecto a y^t_k, es necesario analizar únicamente aquellas secuencias que pasan por el símbolo k en el instante t. Sin embargo, se debe recordar que varios símbolos en $\boldsymbol{l}'$ pueden ser mapeados al mismo símbolo en $\boldsymbol{l}$ (por ejemplo las secuencias $\boldsymbol{l}' = \{aaa;\, a--;\ldots\}$ son transoformadas en $\boldsymbol{l} = \{a\}$), y por ende se deben sumar las derivadas de todos aquellos casos. Esto significa que la derivada de la función de *likelihood* con respecto a la salida de la clase k-ésima en el instante t se obtiene como la suma de las derivadas sobre todos los *paths* en $\boldsymbol{l}'$ que pasan por la clase k en el instante t. Para expresar esto matemáticamente, se puede introducir una función $lab(\boldsymbol{l}',k) = \{s:\, l'_s = k\}$ que cuenta la cantidad de repeticiones de dicha clase. Finalmente, es posible obtener la expresión que permite aplicar el algoritmo de backpropagation maximizando el *likelihood*:

$$\boxed{\frac{\partial p(\boldsymbol{l}|\boldsymbol{x})}{\partial y^t_k} = -\frac{1}{\left(y^t_k\right)^2}\sum_{s\in lab(\boldsymbol{l}',k)}\alpha_t(s)\,\beta_t(s) \rightarrow \frac{\partial\ln\left(p(\boldsymbol{l}|\boldsymbol{x})\right)}{\partial y^t_k} = \frac{1}{p(\boldsymbol{l}|\boldsymbol{x})}\frac{\partial p(\boldsymbol{l}|\boldsymbol{x})}{\partial y^t_k}}$$

6.3. Decodificación

En la sección previa se explicó el procedimiento de entrenamiento de una red neuronal basada en CTC, implementando una función de costo que permite maximizar el *likelihood* entre una secuencia predicha de caracteres, a la cual se le aplica la función de mapeo $\mathcal{B}(\cdot)$, y la secuencia real $\boldsymbol{l}$ de caracteres de entrenamiento. Para esto se utilizaron métodos como el algoritmo de *forward-backward* para evitar el cálculo por fuerza bruta que requería sumar el *likelihood* de todas las posibles secuencias con respecto a las predicciones de la red.

Una vez resuelto el problema de entrenamiento, se debe proceder a resolver el problema de decodificación, donde es necesario encontrar la secuencia de caracteres más probable. En la sección 3.2.2 se presentó el algoritmo

de Viterbi, pero a continuación se presentarán dos métodos que son más utilizados en este tipo de tareas con redes neuronales.

6.3.1. *Greedy decoding*

Este algoritmo también suele ser conocido como *best path decoding*. Se basa en el hecho de que al elegir el carácter más probable en cada intervalo de tiempo, la secuencia completa resulta la más probable:

$$\hat{z}_t = \mathcal{B}\left(\arg\max_k \left(\boldsymbol{y}^t\right)\right)$$

La ventaja de este algoritmo es que es extremadamente veloz, ya que solo requiere encontrar el máximo en cada instante de tiempo y luego eliminar los *blanks* y las repeticiones. La complejidad computacional entonces resulta $\mathcal{O}(T \cdot M)$, donde T es el largo de la secuencia temporal y M la cantidad de caracteres en el alfabeto.

6.3.2. *Beam search decoding*

A pesar de las ventajas que presenta *greedy decoding*, existen ciertas situaciones donde no resulta el más eficiente. Un claro ejemplo de esto puede establecerse a partir del diagrama de la figura 6.4. Si se aplicara *greedy decoding* la secuencia de salida sería $\{-,-\}$ ya que resultan los símbolos más probables, dando una secuencia de probabilidad de $0{,}8 \cdot 0{,}6 = 0{,}48$. Sin embargo, las secuencias $\{aa, a-, -a\}$ son mapeadas al símbolo $\mathcal{B}(\{aa, a-, -a\}) = a$, dando una probabilidad de $0{,}2 \cdot 0{,}4 + 0{,}2 \cdot 0{,}6 + 0{,}8 \cdot 0{,}4 = 0{,}52$.

Por lo tanto, debido a que $\mathcal{B}(\cdot)$ es una función del tipo *many-to-one* pueden aparecer secuencias más probables que la obtenida por el algoritmo anterior si es que se analizan varias posibles secuencias en simultaneo.

El algoritmo de *beam search* lo que hace es crear candidatos de texto, llamados ***beams***, a los que les asigna un determinado puntaje. En cada instante de tiempo lo que se hace es almacenar los BW (*beam width*) *beams* más probables[12]. Luego, al pasar al siguiente instante de tiempo, cada uno de estos *beams* es extendido por las probabilidades de cada uno de los caracteres, y el proceso es repetido hasta el final de la secuencia, donde se retorna la secuencia de *beams* más probable.

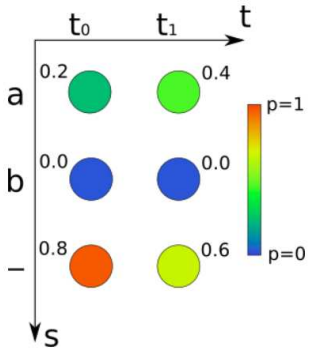


Figura 6.4: Ejemplo de la salida de una red neuronal que utiliza CTC (extraído de [32]).

La ventaja de este método entonces es que permite recorrer varias secuencias en simultaneo, y de esta manera solucionar el problema encontrado con *greedy decoder*. Otra gran ventaja de este método es que permite introducir un modelo de lenguaje. La complejidad temporal del algoritmo resulta $\mathcal{O}(T \cdot BW \cdot C \cdot \log(BW \cdot C))$, donde T es el largo de la secuencia, BW es la cantidad de *beams*, y C es la cantidad de caracteres.

En la figura 6.5 se presenta un ejemplo de este algoritmo, donde se puede ver que en el primer instante de tiempo se eligen como *beams* los caracteres $\{a, -\}$. Luego cada *beam* es extendido por los demás caracteres, y se eligen los más probables para continuar con el proceso. Al encontrar secuencias que resultan iguales, se unifican los *beams* y se suman las probabilidades.

[12] El caso en que $BW = 1$ se obtiene como resultado el algoritmo de *greedy decoder* ya que se está analizando sólo la opción más probable. Sin embargo, las implementaciones de *greedy decoders* suelen ser más eficientes que la utilización de *beam search* con $BW = 1$.

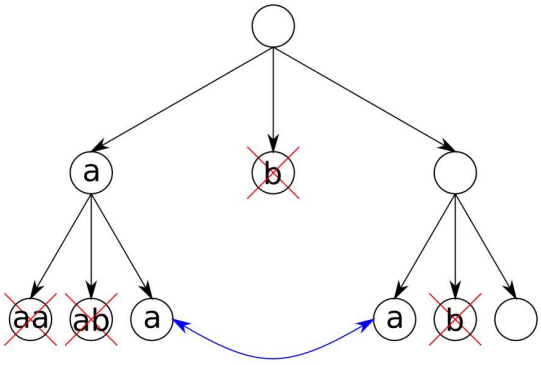


Figura 6.5: Ejemplo del funcionamiento del algoritmo de *beam search* con $BW = 2$.

6.4. Métrica de error

Además de la utilización de funciones de costo, es necesario definir una función de *accuracy* para poder medir el error de clasificación en unidades que se encuentren mejor relacionadas con el problema en cuestión. En los problemas del tipo *sequence-to-sequence* suele utilizarse una métrica conocida como distancia de Levenshtein o *edit distance*, la cual define a la distancia entre dos secuencias como la mínima cantidad de operaciones (inserción, borrado y substitución) que se deben realizar sobre una de las secuencias para obtener la otra:

$$Edit\,distance = Insertions + Deletions + Substitutions$$

Al normalizar esta distancia por el largo de la secuencia objetivo se obtiene una nueva medida conocida como ***Label Error Rate (LER)***, la cual tiene una interpretación similar al *accuracy* en los problemas de clasificación convencionales, y que permite conocer un porcentaje de similitud de las secuencias predichas con las reales.

Si bien esta métrica es representativa del error a nivel caracter, la interpretación de una sentencia viene dada por las palabras predichas. Por esta razón, los trabajos en esta área suelen medir el error a nivel palabra[13], utilizando una medida conocida como ***Word Error Rate (WER)***, el cual es la *edit distance* de palabras entre la secuencia objetivo y la real. Para poder obtener esta medida se debe realizar lo siguiente:

- Se realiza una predicción a nivel caracter, obteniendo una lista de símbolos.
- Se busca en dicha lista los símbolos correspondientes al espacio, y se separa la lista original en sublistas separadas por dicho caracter.
- Cada sublista representa una palabra. Se calcula la *edit distance* entre dichas palabras.
- Se normaliza la distancia por la cantidad de palabras, y así se obtiene como resultado el WER.

Es importante remarcar que un caracter incorrecto dentro de una palabra hace que dicha palabra sea considerada errónea, y debido a que la cantidad de palabras por oración suele ser menor que la cantidad de caracteres por oración, en general el WER resulta superior al LER. Una manera de mejorar esto es mediante la utilización de un modelo de lenguaje, de manera de que las palabras predichas sean siempre palabras válidas.

Dado que en este trabajo no se utilizó ningún modelo de lenguaje, se decidió utilizar ambas métricas al mismo tiempo. De esta forma, se pueden analizar los siguientes posibles resultados:

- LER bajo y WER alto: Los errores a nivel caracter se encuentran distribuidos a lo largo de las oraciones, lo cual puede provocar cambios en la interpretación (cambios de sexo, de tiempo, etc.).
- LER alto y WER bajo: Los errores a nivel caracter se encuentran concentrados dentro de pocas palabras. Esto puede estar indicando que dichas palabras no aparecieron demasiado durante el entrenamiento. Si el WER es lo suficientemente bajo, la interpretación de la oración no se ve comprometido.
- LER bajo y WER bajo: El sistema está funcionando correctamente.

[13] En conjunto con modelos de lenguaje.

- LER alto y WER alto: El sistema no está funcionando correctamente.

Dependiendo de los resultados observados, es posible tomar decisiones sobre posibles modificaciones en los hiperparámetros utilizados.

7. Redes LAS (*Listen, Attend and Spell*)

Como se presentó en la sección anterior, el algoritmo CTC permite entrenar modelos de secuencias no alineadas. Sin embargo, esta estrategia asume que las etiquetas predichas son condicionalmente independientes de todas las demás, lo cual es una hipótesis no utilizada para este nuevo método. Las redes neuronales LAS fueron presentadas en 2015 en [5], y demostraron no solo alcanzar mejores resultados que las redes convencionales, sino que también mejoraron las predicciones obtenidas con redes basadas en CTC.

7.1. Descripción

Sea $\boldsymbol{x} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T)$ una secuencia de *features* de entrada y sea $y = (<\text{sos}>, y_1, \ldots, y_U, <\text{eos}>)$ la secuencia de etiquetas objetivo. Al igual que para el caso de CTC, el objetivo es encontrar un sistema que transforme secuencias de habla en secuencias de texto, es decir una transformación del tipo $\mathcal{T} : \mathbb{R}^{m \times T} \rightarrow \mathbb{R}^{n \times U}$. En este trabajo de tésis se decidió no utilizar símbolos de puntuación ni números, por lo que los símbolos de salida resultan únicamente caracteres. Sin embargo, para trabajar con este tipo de redes es necesario introducir dos nuevos símbolos que permitirán indicar el inicio y fin de las secuencias: el *token* de START_OF_SEQUENCE<sos> y el de END_OF_SEQUENCE <eos>.

Una de las principales ventajas de esta técnica es que no asume independencia condicional para las salidas en distintos instantes de tiempo. Por lo tanto, el *likelihood* en este caso resulta:

$$p\left(\boldsymbol{y}|\boldsymbol{x}\right) = \prod_i p\left(y_i|\boldsymbol{x}, y_{<i}\right) \tag{7.1}$$

El método propuesto en el trabajo [5] se encuentra basado en dos etapas, las cuales serán descritas a continuación.

7.1.1. Primer etapa: *Listen*

Uno de los principales problemas que presentan los sistemas conversores de habla a texto es la gran diferencia de longitudes entre la entrada y la salida. Por esta razón, en este sistema se introduce una etapa de reducción de longitudes:

$$\boldsymbol{h} = \text{Listen}\left(\boldsymbol{x}\right) \qquad \text{Listen} : \mathbb{R}^{m \times T} \rightarrow \mathbb{R}^{n \times U} \qquad U \leq T$$

Esta reducción en la longitud de las secuencias se logra mediante la utilización de redes recurrentes bidireccionales agrupadas en una estructura piramidal de manera de reducir la longitud a la mitad en cada una de las etapas. La salida en el instante i en la j-ésima capa se la puede denotar como:

$$h_i^j = pRNN\left(h_{i-1}^j, \left[h_{2i}^{j-1}, h_{2i+1}^{j-1}\right]\right)$$

Se puede ver que en esta red se están relacionando la salida de la capa j-ésima en el instante anterior h_{i-1}^j, junto con la concatenación de salidas consecutivas en la capa previa[14]. La implementación de esta etapa consiste en concatenar los *features* de instantes sucesivos, de manera que la dimensión temporal se reduzca a la mitad, mientras que la cantidad de *features* de salida se duplica. En la figura 7.1 se tiene un diagrama que representa esta etapa.

[14] Dado que la longitud se reduce a la mitad, el valor en el instante i en la capa j proviene del valor en el instante $2i$ en la capa $j-1$.

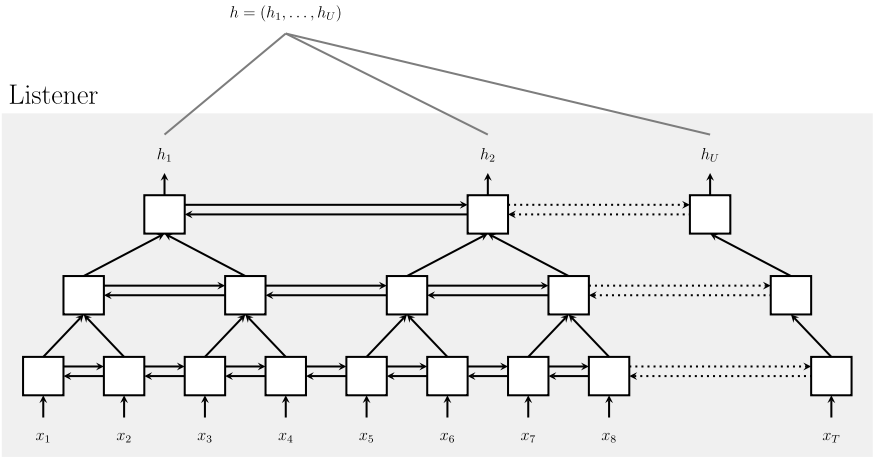


Figura 7.1: Diagrama en bloques del módulo *listener* (obtenido de [5]).

7.1.2. Segunda etapa: *Attend and Spell*

El mecanismo de atención (*attention mechanism*) fue propuesto inicialmente en [3], y permite realizar traducciones del tipo *sequence-to-sequence* donde se buscan partes relevantes de las secuencias para ser resaltadas, es decir "prestarles atención". Previo a este método, los métodos de traducción consistían en la utilización de estructuras *enconder-decoder* que comprimían toda la información en secuencias de largo fijo, mientras que utilizando este mecanismo de atención la tarea se realiza de una manera similar a la forma en que los humanos realizamos traducciones, focalizándonos en las partes importantes.

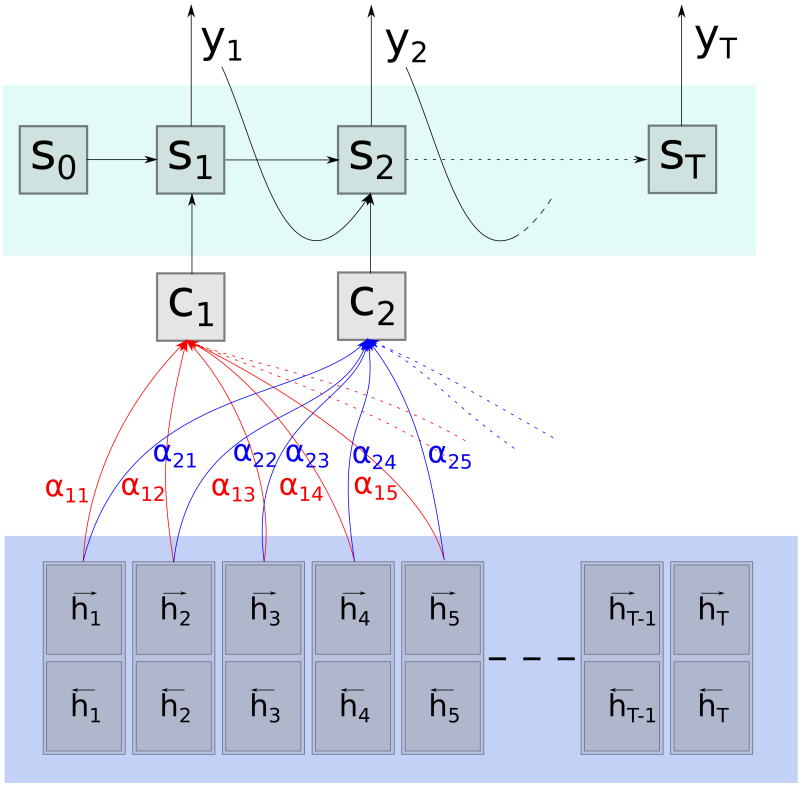


Figura 7.2: Diagrama en bloques del módulo de atención basado en el diagrama de [3].

En la figura 7.2 se tiene esquematizado el mecanismo de atención. El funcionamiento de este mecanismo es similar a las estructuras del tipo *encoder-decoder* para traducción de secuencias: Se tiene una red recurrente bidireccional de la cual se extraen los coeficientes *forward* y *backward*, y se tiene otra red recurrente que utiliza las predicciones previas y la nueva información para realizar una nueva predicción.

Para simplificar la notación, los coeficientes de la recursión *forward* $\overrightarrow{\boldsymbol{h}}_i$ y los coeficientes de la recursión *backward* $\overleftarrow{\boldsymbol{h}}_i$ son concatenados en un único vector $\boldsymbol{h}_i = \left[\overrightarrow{\boldsymbol{h}}_i, \overleftarrow{\boldsymbol{h}}_i\right]$, el cual será el vector de *features* para el instante i-ésimo.

A partir de estos coeficientes, se define el **vector de contexto** $\boldsymbol{c}_i$ el cual es el encargado de comprimir la información de los distintos instantes de tiempo para realizar nuevas predicciones. El vector de contexto se encuentra compuesto por un promedio ponderado de las distintas entradas, el cual se obtiene mediante los pesos α_{ij}[15] que indican cuánta atención se le debe prestar a la entrada $\boldsymbol{h}_j$ para obtener el contexto de la salida i-ésima:

$$\boxed{\boldsymbol{c}_i = \sum_{j=1}^{T} \alpha_{ij} \boldsymbol{h}_j}$$

Se puede ver que los pesos α_{ij} deben cumplir que $\sum_{j=1}^{T} \alpha_{ij} = 1$, y que cuanto mayor sea el valor del α_{ij} mayor será el aporte de $\boldsymbol{h}_j$ en la predicción del instante i-ésimo.

El siguiente paso consiste en utilizar los vectores de contexto para computar las predicciones. Esto se realiza utilizando una nueva red recurrente que utiliza como entrada los vectores de contexto para ajustar su estado interno $\boldsymbol{s}_i$ (el cual es propagado a las iteraciones posteriores como se observa en la figura 7.2) y cuya salida son las predicciones deseadas.

El paso faltante corresponde a la obtención de los coeficientes α_{ij}, lo cual suele ser realizado mediante una red neuronal. Es de esperar que los coeficientes que indican cuánta atención debe prestarse a la entrada $\boldsymbol{h}_j$ en el instante i sea una función de la entrada $\boldsymbol{h}_j$ y del estado en el instante previo. Se utiliza entonces un perceptrón multicapa con salida *softmax*:

$$e_{ij} = a\left(\boldsymbol{h}_j,\, \boldsymbol{s}_{i-1}\right) \rightarrow \alpha_{ij} = \frac{\exp\left(e_{ij}\right)}{\sum_{k=1}^{T} \exp\left(e_{ik}\right)} = \operatorname{softmax}\left(e_{ij}\right) \tag{7.2}$$

La red neuronal $a\left(\cdot\right)$ se la conoce como **modelo de alineación**.

Finalmente, una vez obtenidas las distribuciones del contexto y de los estados, se debe aplicar el modelo de *speller* $g\left(\cdot\right)$ que se encargará de generar las etiquetas de salida correspondientes (en este caso caracteres), que consiste básicamente en un perceptrón multicapa que traduce los *features* de salida del mecanismo de atención en una distribución de probabilidades sobre los posibles caracteres obteniendo así una estimación de $p\left(y_i|\boldsymbol{x}, y_{<i}\right)$.

[15] Conocidos también como *attention weights*.

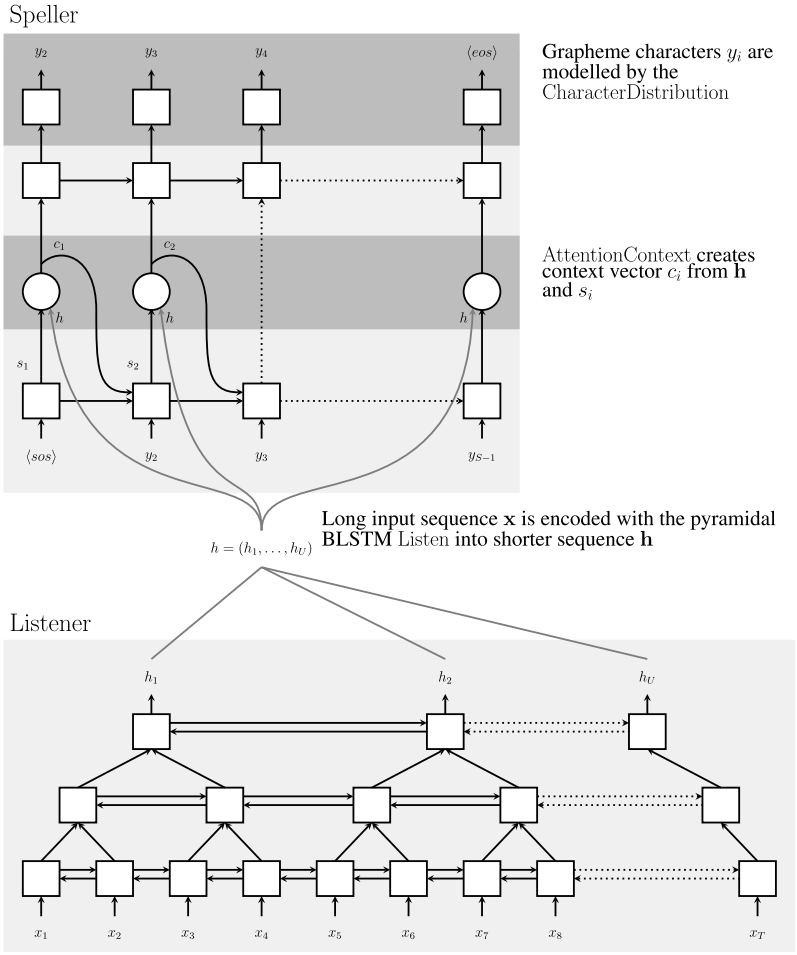


Figura 7.3: Diagrama completo de una red LAS (obtenido de [5]).

Este mecanismo permite pesar cada una de las salidas y_t por cada una de las entradas. Si bien de esta forma se logra analizar el contexto completo de la secuencia, aparece el problema de que se deben calcular $\mathcal{O}\left(n^2\right)$ coeficientes, lo cual puede ser prohibitivo para ciertos problemas. Esto parece contradecir el hecho de que el objetivo era prestar atención unicamente a las partes relevantes de la entrada, pero el problema es que para decidir qué es relevante se debe primero observar toda la secuencia.

7.2. Hard attention

El modelo de atención presentado en la sección anterior suele ser conocido como *soft attention* ya que todos los vectores de contexto aportan a las predicciones en los distintos instantes de tiempo, es decir que la atención se reparte de manera suave entre los estados. Sin embargo, existe otro modelo de atención conocido como *hard attention* que presta atención solo a un subgrupo de entradas. Esto permite reducir la complejidad computacional y evita una baja del rendimiento al trabajar con secuencias más largas, pero con la desventaja de que la operación no resulta derivable. Sin embargo, Luong *et al.* [21] introdujo una técnica que permite reducir este problema.

Mientras que en el modelo de *soft attention* se puede entender que el vector de contexto se obtiene como la esperanza de los estados $\boldsymbol{h}_j$, en *hard attention* se realiza un proceso de muestreo donde se seleccionan aquellas entradas que serán utilizadas de acuerdo a la probabilidad obtenida de los α_{ij}, y se descartan las demás. En este caso el contexto se puede calcular de la siguiente manera:

$$\boldsymbol{c}_i = \sum_{j=1}^{T} \delta_{ij} \boldsymbol{h}_j$$

donde δ_{ij} puede considerarse como la función $\operatorname{argmax}(\alpha_{ij})$, es decir que en la sumatoria sólo sobrevive uno de los vectores de contexto. Esta implementación muchas veces es reemplazada por una multinomial para facilitar el cómputo.

Las diferencias entre los mecanismos soft y *hard* son más evidentes en los casos en que se trabaja con imágenes, lo cual puede observarse en la figura 7.4.

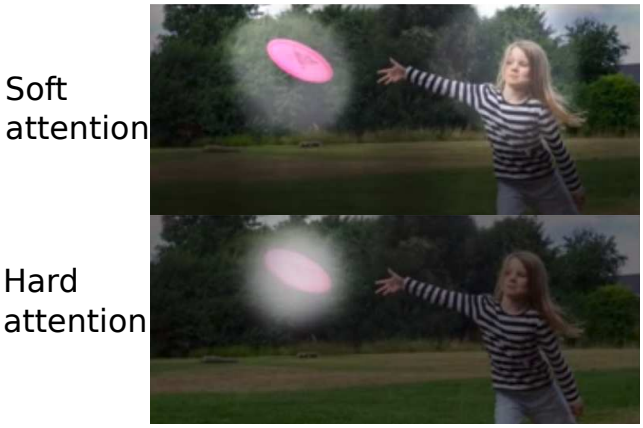


Figura 7.4: Ejemplo de la diferencia entre *soft* y *hard attention*. Se puede ver que en el caso *soft* se le presta atención a varias regiones de la imagen, mientras que en el caso *hard* sólo se ilumina una parte.

7.3. Entrenamiento

Como se mencionó previamente, la salida de esta red resulta en una matriz que contiene la probabilidad de cada una de las clases para cada instante de tiempo. Por lo tanto, es posible utilizar la función de costo *cross-entropy* presentada en la sección 5.3.2, pero aplicada a lo largo del eje temporal de la salida. Esto significa que se deben tomar la secuencia predicha y la real, calcular la *cross-entropy* entre los pares que tienen el mismo índice, y luego realizar una suma de todos los costos parciales.

Por otra parte, debido a que la salida de esta arquitectura es una matriz que contiene las probabilidades de cada símbolo para cada instante de tiempo, es posible utilizar los mismos métodos de decodificación que para la red CTC, con la diferencia de que ahora se debe utilizar el símbolo `<eos>` como indicador de se finalizó con la predicción.

Parte III

Implementaciones

En la sección previa se expuso la base teórica que permite la implementación de las distintas arquitecturas. En esta sección se describen los detalles importantes a tener en cuenta al implementar el código de los diferentes modelos de redes neuronales utilizados en esta tesis.

8. Entorno

Uno de los principales problemas enfrentados en este trabajo fue encontrar la manera de entrenar los distintos modelos:

- Las redes utilizadas son altamente profundas.
- Las redes recurrentes funcionan de manera iterativa, lo cual requiere más tiempo de cómputo que las demás arquitecturas.
- La cantidad de ejemplos necesarios es elevada, lo cual requiere tenerlos montados en memoria[16] e incrementa los tiempos de cómputo. Por otra parte, las muestras de entrenamiento en general se procesan en *batchs* lo cual requiere paralelizar las operaciones, es decir que se hace necesaria la utilización de una placa de video (GPU).

Es por esta razón que se utilizaron los siguientes entornos:

- **Computadora personal**:
  - GPU: NVIDIA GTX940MX (2GB).
  - RAM: 8GB.
  - OS: Ubuntu 17.10
  - Versión de TensorFlow: 1.13.1
  - Versión de Python: 3.6

  Con esta computadora fue posible realizar pruebas a pequeña escala.
- **Google Colaboratory**: Afortunadamente Google ofrece un servicio gratuito de máquina virtual con un entorno de Python basado en Jupyter. Permite hacer uso de sus recursos durante 12 h continuas, obligando a restaurar la sesión una vez transcurrido dicho intervalo.
  - GPU: Tesla K80 (24GB).
  - RAM: 13GB.
  - Versión de TensorFlow: 1.14.0
  - Versión de Python: 3.6

  Mediante este entorno es posible también vincular con Google Drive, lo cual permite tener almacenados los *datasets* sin tener que regenerarlos cada vez que se restaure la sesión. Lo mismo ocurre con los pesos de las redes entrenadas, los cuales deben ser descargados antes de que se reinicie el entorno para evitar perderlos.

Para montar el sistema de archivos de Google Drive en Google Colaboratory se debe añadir la siguiente celda:

[16] Existen alternativas para solucionar esto. Es posible utilizar una estructura de archivos conocida como *TFRecords* (`https://www.tensorflow.org/guide/datasets`), la cual permite introducir al grafo de la red las operaciones de lectura de los archivos. De esta manera, los ejemplos son leídos directamente de los archivos sin la necesidad de montarlos todos en memoria RAM. La desventaja de esto es que la lectura de ejemplos resulta más lenta.

```
# Se monta el sistema de archivos de Google Drive. Requiere autenticación.
from google.colab import drive
drive.mount('/content/drive/')

# Se agrega la ruta al repositorio de manera de hacer uso de los distintos módulos
# de Python de forma directa. Esto permite importar bibliotecas propias de la misma
# manera que en una computadora personal.
import sys
sys.path.append('drive/My Drive/Tesis/repo')
```

En esta tesis se desarrollaron módulos propios que deben ser importados para poder ejecutar distintas funcionalidades. Es por esto que es importante la configuración de las rutas al proyecto[17].

9. Configuración global

Las redes implementadas presentan parámetros que se encuentran compartidos, por lo que éstos fueron implementados de tal manera de reducir la cantidad de código y de facilitar su uso.

La primera estructura consiste en la configuración del proyecto, donde simplemente se tienen almacenadas las rutas a todos los archivos necesarios: bases de datos, almacenamiento de los pesos y modelos, etc. Esto fue implementado en la clase llamada `ProjectData`:

```
class ProjectData:
    def __init__(self):
        # Files data
        self.SOURCE_DIR = 'data/'
        self.WAV_DIR = self.SOURCE_DIR + 'wav/'
        self.WAV_TRAIN_DIR = self.WAV_DIR + 'wav_train/'
        self.WAV_TEST_DIR = self.WAV_DIR + 'wav_test/'
        self.TRANSCRIPTION_DIR = self.SOURCE_DIR + 'transcription/'
        self.TRANSCRIPTION_TRAIN_DIR = self.TRANSCRIPTION_DIR + 'transcription_train/'
        self.TRANSCRIPTION_TEST_DIR = self.TRANSCRIPTION_DIR + 'transcription_test/'
        self.DATABASE_DIR = self.SOURCE_DIR

        self.OUT_DIR = 'out/'

        self.ZORZNET_CHECKPOINT_PATH = self.OUT_DIR + 'zorznet/' + 'checkpoint/model.ckpt'
        self.ZORZNET_MODEL_PATH = self.OUT_DIR + 'zorznet/' + 'model/model'
        self.ZORZNET_TENSORBOARD_PATH = self.OUT_DIR + 'zorznet/' + 'tensorboard/'

        self.LAS_NET_CHECKPOINT_PATH = self.OUT_DIR + 'las_net/' + 'checkpoint/model.ckpt'
        self.LAS_NET_MODEL_PATH = self.OUT_DIR + 'las_net/' + 'model/model'
        self.LAS_NET_TENSORBOARD_PATH = self.OUT_DIR + 'las_net/' + 'tensorboard/'
```

Esta clase es utilizada para definir las rutas en todos los entrenamientos, de manera de que quede lo más parametrizado posible.

Otro estructura importante es la que permite la configuración de los hiperparámetros de las redes: cantidad de capas, cantidad de neuronas, etc. En este caso se definió la clase `NetworkDataInterface`:

```
class NetworkDataInterface:
    checkpoint_path: str = None
    model_path: str = None
    tensorboard_path: str = None

    num_features: int = None
    num_classes: int = None

    optimizer: str = None
    learning_rate: float = None
```

Los primeros tres atributos permiten determinar las rutas donde se almacenarán los archivos de salida de la red: el archivo del modelo contiene la estructura del grafo, el archivo del *checkpoint* contiene los pesos entrenados y el archivo de *Tensorboard* permite visualizar información relevante de la red. Los demás atributos son:

- `num_features`: Cantidad de *features* de la entrada.
- `num_classes`: Cantidad de símbolos (caracteres) de salida.

[17] Se asume en este caso que el repositorio fue clonado en una carpeta llamada `repo` en la ruta `drive/My Drive/Tesis/`.

- `optimizer`: Nombre del algoritmo de optimización que se desea utilizar (gradiente descendente, ADAM, etc.).
- `learning_rate`: Paso para utilizar con el optimizador.

En esta clase entonces están definidos todos los hiperparámetros que están compartidos entre todos los tipos de redes. Dado que cada red tiene una estructura diferente, se establecieron clases específicas para cada una, las cuales heredan las propiedades de `NetworkDataInterface`.

Si bien inicialmente se armaron clases para crear los métodos de entrenamiento, validación y testeo de cada uno de los modelos (entre otros), debido a la gran cantidad de datos se decidió utilizar la herramienta de *TensorFlow* que se presenta en la siguiente sección[18].

9.1. *TensorFlow Estimators*

La biblioteca conocida como `tf.estimator` es una API de *TensorFlow* [6] que consiste en una interfaz de alto nivel que facilita el armado de nuevos modelos, sin la necesidad de preocuparse por los métodos de entrenamiento y validación, así como tampoco de la inserción de datos al modelo. Por otra parte, esta biblioteca facilita la ejecución de modelos en CPUs, GPUs o TPUs sin la necesidad de modificar los modelos de las redes, únicamente modificando los parámetros de configuración del entorno[19].

Para incorporar un nuevo modelo a esta biblioteca se deben seguir los siguientes pasos:

- Se debe crear una función generadora de datos. Como se describirá en la próxima sección, esto es posible combinarlo con la biblioteca `tf.dataset` de manera de ingresar los datos a los distintos modelos de una manera estandarizada. Un ejemplo para esta función puede ser:

```
def data_input_fn(filenames, batch_size, parse_fn, shuffle_buffer, num_epochs):
    dataset = tf.data.TFRecordDataset(filenames)
    dataset = dataset.map(parse_fn)
    dataset = dataset.batch(batch_size=batch_size)
    dataset = dataset.shuffle(shuffle_buffer).repeat(num_epochs)
    iterator = dataset.make_one_shot_iterator()
    next_element = iterator.get_next()
    feature, label = next_element
    return feature, label
```

  Esta función recibe una lista de archivos que contienen el *dataset*, el tamaño del *batch*, una función que conoce la estructura con que se encuentran almacenados los datos en el *dataset*, el tamaño del *buffer* con el que se mezclarán las muestras y la cantidad de iteraciones que se desean realizar, y devuelve los tensores con las entradas y salidas de la red. Esta función es añadida al grafo de la red, de manera que es ejecutada junto con las demás operaciones, evitando tener que leer muestras y procesarlas de manera separada.

- Se crea una función que recibe las entradas de la red y retorna su salida, es decir una función que define el modelo de la red:

```
def model_fn(
    features, # Los features obtenidos de data_input_fn()
    labels,   # Los labels obtenidos de data_input_fn()
    mode,     # El modo de ejecución de la red: tf.estimator.ModeKeys.TRAIN, tf.estimator.ModeKeys
        .EVAL, tf.estimator.ModeKeys.TEST
    params  # Diccionario con parámetros de configuración adicional
):
```

  La función que define el modelo debe retornar un objeto del tipo `tf.estimator.EstimatorSpec()`, donde se definen distintas operaciones dependiendo del modo en que se esté ejecutando la red.

- Una vez definida la operación de ingreso de datos y definido el modelo de la red, la biblioteca ofrece los métodos de entrenamiento, validación y testeo de manera automática. Para esto se debe crear el modelo:

[18] Los métodos creados previamente se encuentran disponibles en el repositorio de este trabajo.

[19] Si bien esto es cierto para la mayoría de los casos, en este proyecto no fue posible configurar el uso de TPUs debido a que se encontraron conflictos con el uso de redes recurrentes con secuencias de tamaño variable. Si se hubiera logrado, los tiempos de entrenamiento podrían haberse reducido casi en un 50 %.

```
model = tf.estimator.Estimator(
    model_fn=model_fn,      # Función que define el modelo de la red
    params=params,          # Diccionario con parámetros de configuración adicional
    config=config           # Configuración de la ejecución (ruta de guardado de pesos, guardado de
        checkpoints, etc.)
)
```

Una vez creado el modelo, para ejecutar cualquiera de los tres métodos enunciados previamente se debe hacer:

```
# Entrenamiento
model.train(
    input_fn=lambda: data_input_fn(
        filenames=train_files,
        batch_size=batch_size,
        parse_fn=parse_fn,
        shuffle_buffer=shuffle,
        num_epochs=train_epochs
    )
)

# Validacion
model.evaluate(
    input_fn=lambda: data_input_fn(
        filenames=val_files,
        batch_size=batch_size,
        parse_fn=parse_fn,
        shuffle_buffer=shuffle
    )
)

# Testeo
model.predict(
    input_fn=lambda: data_input_fn(
        filenames=test_files,
        batch_size=1,
        parse_fn=parse_fn
    )
)
```

Se puede ver entonces que esta biblioteca facilita el uso de las redes, ya que sólo es necesario realizar el armado de los modelos y el procesamiento de los datos, sin tener que armar la lógica de ejecución, guardado, etc.

10. Generación de datos

Uno de los procesos más importantes para trabajar con redes neuronales está relacionado con el correcto acondicionamiento de los datos de entrenamiento. Si bien es posible delegar en el entrenamiento de las redes el proceso de encontrar la mejor representación posible de los datos de entrada, utilizar una representación que facilita la extracción de información permite reducir los tiempos de entrenamiento y mejorar los resultados.

10.1. Generación de *features*

Durante el desarrollo de este trabajo y el de los trabajos previos [39, 40] se observó que los resultados obtenidos al utilizar coeficientes MFCC en lugar de espectrogramas fueron superiores, y es por esta razón que se optó en este trabajo utilizar dicha técnica. Para el cálculo de estos coeficientes se utilizó la biblioteca `python_speech_features` [22]:

```
import python_speech_features as features

def mfcc(self, audio, fs: float, winlen: float, winstep: float, numcep: int,
         nfilt: int, nfft: int, lowfreq, highfreq,
         preemph: float, winfunc) -> np.ndarray:
    return features.mfcc(audio, samplerate=fs, winlen=winlen, winstep=winstep,
                         numcep=numcep, nfilt=nfilt, nfft=nfft, lowfreq=lowfreq,
                         highfreq=highfreq, preemph=preemph, winfunc=winfunc)
```

Mediante esta biblioteca es posible transformar los audios en matrices de tamaño `[T, num_ceps]`.

Otra representación analizada fue la utilizada por el proyecto **DeepSpeech** [15]. En esta representación lo que se hace es calcular los coeficientes MFCC de la manera tradicional, pero se realiza un proceso de agrupación

de muestras, de manera que en cada instante de tiempo se tengan más *features*, es decir que se incrementa la dimensionalidad de la entrada.

```
import python_speech_features as features

def deep_speech_mfcc(self, audio, fs: float, winlen: float, winstep: float,
                     numcep: int, nfilt: int, nfft: int, lowfreq, highfreq,
                     preemph: float, winfunc, num_context: int) -> np.ndarray:

    features = self.mfcc(audio, fs=fs, winlen=winlen, winstep=winstep,
                         numcep=numcep, nfilt=nfilt, nfft=nfft,
                         lowfreq=lowfreq, highfreq=highfreq,
                         preemph=preemph, winfunc=winfunc)

    # We only keep every second feature (BiRNN stride = 2)
    features = features[::2]

    num_strides = len(features)

    empty_context = np.zeros((num_context, numcep), dtype=features.dtype)

    features = np.concatenate((empty_context, features, empty_context))

    window_size = 2 * num_context + 1
    strided_features = np.lib.stride_tricks.as_strided(
        features,
        (num_strides, window_size, numcep),
        (features.strides[0], features.strides[0], features.strides[1]),
        writeable=False)

    strided_features = np.reshape(strided_features, [num_strides, -1])

    # Copy the strided array so that we can write to it safely
    strided_features = np.copy(strided_features)
    strided_features = (strided_features - np.mean(strided_features)) / np.std(strided_features)

    return strided_features
```

El proceso de obtención de estos *features* se puede describir de la siguiente manera:

- Primero se deben calcular los coeficientes MFCC de la señal de audio obteniendo una matriz de tamaño `[T, num_ceps]`, donde `T` es el tiempo y `num_ceps` es la cantidad de coeficientes MFCC.
- Para reducir el tamaño de las matrices de *features* se descartan los índices temporales impares, de manera que la matriz de *features* ahora es de tamaño `[T/2, num_ceps]`.
- Se agregan al principio y al final de la matriz de *features* dos submatrices de ceros de tamaño `[num_context, num_ceps]`, de manera que ahora el tamaño de la matriz de *features* resulta `[T/2 + 2*num_context, num_ceps]`. El motivo de esto es que se realizará un proceso de ventaneo, y estas matrices permiten utilizar ventanas de igual tamaño sin afectar el resultado. Dicho ventaneo permite agrupar conjuntos de *features* de distintos instantes de tiempo, y la variable `num_context` indica el tamaño de la ventana.
- Mediante el método `np.lib.stride_tricks.as_strided()` se realiza el siguiente proceso:
  - Se crea una ventana de tamaño `2*num_context+1` [20].
  - Se aplica la ventana empezando por el indice `num_context+1`. Por lo tanto, en la primer ventana se incorporan los primeros `num_context` ceros, y los siguientes valores son los de la matriz de MFCCs. A la salida de esta ventana se obtiene entonces una matriz de tamaño `[2*num_context+1, num_ceps]`.
  - La ventana se desplaza una posición y se repite el proceso. El desplazamiento se realiza hasta alcanzar el último índice previo a la submatriz concatenada al final, es decir hasta el índice `num_context+T/2`.
  - A la salida se obtienen entonces `T/2` matrices de tamaño `[2*num_context+1, num_ceps]`.
- Al listado de matrices se le aplica un *reshape* de manera de transformar cada matriz en un vector, por lo que finalmente la matriz de *features* presenta ahora un tamaño `[T/2, (2*num_context+1)*num_ceps]`.

[20] `num_context(pasado) + 1(presente) + num_context(futuro)`

- Finalmente se realiza un proceso de normalización, donde se resta la media y se divide por la varianza.

Se puede ver que mediante este proceso se incrementa fuertemente la cantidad de *features* por instante de tiempo, introduciendo información de los coeficientes MFCC de instantes pasados y futuros. De esta manera, la correlación temporal realizada por la red recurrente se ve complementada con las capas densas que extraen la información en cada instante de tiempo.

En la figura 10.1 se tiene un ejemplo de una matriz de *features* obtenida mediante este mecanismo. Se puede observar cómo la cantidad de *features* resulta mayor que la longitud temporal de la entrada. Por otra parte, se hace evidente también cómo se encuentran apilados los coeficientes MFCC de manera desplazada.

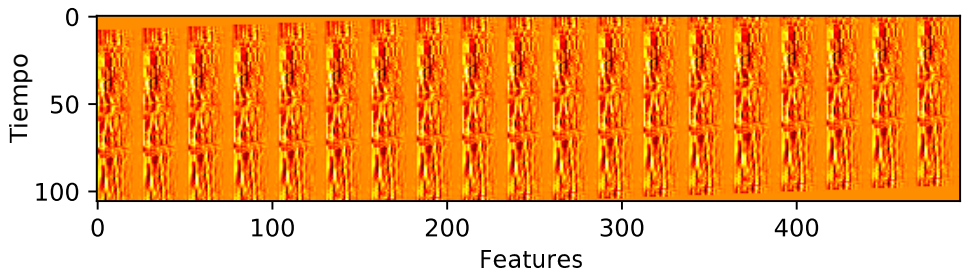


Figura 10.1: Ejemplo de matriz de *features* calculada mediante el método utilizado por DeepSpeech.

Para la generación de *features* se definió la clase `FeatureConfig`, la cual contiene las posibles configuraciones para la extracción de *features* extraídos. Los posibles argumentos son:

- `feature_type`: Es un objeto del tipo `str` que indica el tipo de *feature* a extraer. Los posibles valores son:
  - `'mfcc'`: Coeficientes MFCC.
  - `'deep_speech_mfcc'`: Coeficientes MFCC pero procesados mediante el algoritmo utilizado por DeepSpeech.
  - `'spec'`: Espectograma.
  - `'log_spec'`: Espectograma en escala logarítmica.
- `nfft`: Cantidad de puntos utilizados en la FFT.
- `winlen`: Tamaño de la ventana en milisegundos.
- `winstride`: Desplazamiento de la ventana en milisegundos.
- `preemph`: Factor de *preemphasis* para el filtro de entrada.
- `num_filters`: Cantidad de filtros del banco de filtros.
- `num_ceps`: Cantidad de coeficientes MFCC.
- `mfcc_window`: Tipo de ventana a aplicar durante el ventaneo.
- `lowfreq`: Frecuencia de corte inferior del banco de filtros.
- `highfreq`: Frecuencia de corte superior del banco de filtros.
- `num_context`: Tamaño del contexto utilizado cuando se utiliza *features* del tipo `deep_speech_mfcc`.

10.2. TFRecords

Debido a que el entrenamiento de redes neuronales requiere grandes cantidades de datos, y que generalmente los usuarios comunes no tienen a su disposición computadoras con la capacidad de montar dichos datos en memoria, TensorFlow introdujo un *pipeline* que implementa operaciones de lectura de archivos en el formato especial

tfrecords, basado en los mecanismos de serialización de datos de Google conocido como *Protocol Buffers* (`https://developers.google.com/protocol-buffers/`).

Mediante esta herramienta es posible serializar las muestras de entrenamiento en archivos, para luego deserializarlas durante la ejecución de la red sin la necesidad de tener todas las muestras montadas en memoria.

10.2.1. Generación de archivos

Para generar estos archivos se definió un método dentro de la clase `Dataset` diseñada para este trabajo, que se encarga de transformar de una representación a otra. Los datos almacenados dentro de estos archivos son:

- Matrices de *features*.
- Vectores de *labels*.
- Longitud temporal de los *features*.
- Longitud temporal de los *labels*.

Estos datos deben ser transformados a su correspondiente objeto para poder ser serializado. Las opciones disponibles son: `tf.train.FloatList`, `tf.train.Int64List`, `tf.train.BytesList`, etc. Finalmente, se crea un un objeto del tipo `tf.train.SequenceExample()`, el cual permite almacenar secuencias de longitud variable, se lo serializa a un `String`, y se lo almacena en el archivo deseado.

A continuación se presenta el método utilizado:

```
def to_tfrecords(self, filename: str):

    writer = tf.python_io.TFRecordWriter(filename)

    for item in self.__database:
        feature_len, num_features = np.shape(item.item_feature.feature)
        target_len = len(item.label.to_index())

        feats_list = [tf.train.Feature(float_list=tf.train.FloatList(value=frame))
                      for frame in item.item_feature.feature]
        feat_dict = {"feature": tf.train.FeatureList(feature=feats_list)}
        sequence_feats = tf.train.FeatureLists(feature_list=feat_dict)

        # Context features for the entire sequence
        feat_len = tf.train.Feature(int64_list=tf.train.Int64List(value=[feature_len]))
        target_len = tf.train.Feature(int64_list=tf.train.Int64List(value=[target_len]))
        target = tf.train.Feature(int64_list=tf.train.Int64List(value=item.label.to_index()))

        context_feats = tf.train.Features(feature={"feat_len": feat_len,
                                                   "target_len": target_len,
                                                   "target": target})

        example = tf.train.SequenceExample(context=context_feats,
                                           feature_lists=sequence_feats)

        writer.write(example.SerializeToString())

    writer.close()
```

10.2.2. Lectura de datos

Para poder alimentar a la red neuronal con los datos almacenados en los archivos *tfrecords*, es necesario indicarle a los diferentes métodos internos de TensorFlow la manera de hacerlo. El primer paso es crear un objeto del tipo `tf.data.Dataset`, donde ya se encuentran definidas ciertas operaciones requeridas para poder utilizar los datos almacenados. Los pasos a seguir entonces resultan:

- Creación del objeto `tf.data.Dataset` mediante la función `tf.data.TFRecordDataset()` la cual recibe el listado de archivos *tfrecords*.
- Se define una función de deserialización, la cual se encarga de leer cada ejemplo almacenado en el objeto `dataset` y transformarlo en la representación necesaria por la red.

- Se define el tamaño del *batch*. Dado que se trabaja con secuencias de tamaño variable, se utiliza el método `padded_batch()`, el cual requiere como argumentos los tamaños de cada uno de los elementos serializados, y los valores que se van a utilizar como *padding* dentro del *batch*.

- En caso de que se desee que las muestras sean tomadas de manera aleatoria dentro del *dataset*, se debe definir la operación de `shuffle()`. Un argumento importante a definir es el tamaño del `shuffle_buffer`. En cada iteración, por cada muestra del *batch*, se reserva un espacio para `shuffle_buffer` muestras sobre las que se elegirá de manera aleatoria el ejemplo a utilizar en la siguiente iteración. Si este parámetro es igual a la unidad, no se tiene un comportamiento aleatorio, mientras que si es igual a la cantidad de muestras dentro del conjunto de entrenamiento se estarán mezclando todas las muestras por igual. El problema que se tiene al incrementar este parámetro es que se requiere un mayor uso de memoria.

El segmento de código utilizado para realizar esto se presenta a continuación:

```
def data_input_fn(files_list, map_fn, batch_size, num_features, label_pad, shuffle_buffer=None):
    # Se crea el objeto tf.data.Dataset a partir de la lista de archivo
    # que se desean leer
    dataset = tf.data.TFRecordDataset(files_list)
    # Se define la función encargada de parsear cada muestra
    dataset = dataset.map(map_fn)
    # Se define el tamaño de un batch. Se utiliza padded_batch para poder
    # trabajar con secuencias de distinto tamaño
    dataset = dataset.padded_batch(
        batch_size=batch_size,
        padded_shapes=((None, num_features), [None], (), ()),
        padding_values=(
            tf.constant(value=0, dtype=tf.float32),
            tf.constant(value=label_pad, dtype=tf.int64),
            tf.constant(value=0, dtype=tf.int64),
            tf.constant(value=0, dtype=tf.int64),)
    )
    # Se define un shuffle_buffer para mezclar las muestras en cada operación
    # de lectura
    if shuffle_buffer is not None:
        dataset = dataset.shuffle(shuffle_buffer)

    return dataset
```

Los métodos que restan definir son los encargados de deserializar cada una de las muestras. En estos métodos se debe especificar el tipo de dato que se está intentando deserializar:

- Los elementos cuya longitud será constante dentro del *batch*, se deben deserializar mediante `tf.FixedLenFeature()`.

- Los elementos cuya longitud es variable dentro del *batch*, se deben deserializar mediante `tf.VarLenFeature()`.

Estas funciones deben recibir como argumento un elemento almacenado en el dataset, y deben retornar los tensores correspondientes a cada una de los elementos que se quieran leer[21]. Dado que en este trabajo se utilizaron distintos tipos de datos, fue necesario definir dos métodos de serialización: uno de los métodos retorna cada elemento como un `tf.SparseTensor()` mientras que el otro los retorna en formato `tf.Tensor()`. A continuación se presentan dichas funciones:

```
@staticmethod
def tfrecord_parse_sparse_fn(example_proto):
    context_features = {
        "feat_len": tf.FixedLenFeature([], dtype=tf.int64),
        "target_len": tf.FixedLenFeature([], dtype=tf.int64),
        "target": tf.VarLenFeature(dtype=tf.int64)          }
    sequence_features = {
        "feature": tf.VarLenFeature(dtype=tf.float32),
    }
    # Parse the example (returns a dictionary of tensors)
    context_parsed, sequence_parsed = tf.parse_single_sequence_example(
        serialized=example_proto,
        context_features=context_features,
        sequence_features=sequence_features
    )
```

[21] Se debe tener en cuenta que las dimensiones especificadas en `padded_batch()` deben coincidir con los elementos retornados por la función de deserialización.

```
    return sequence_parsed["feature"], context_parsed["target"], context_parsed["feat_len"],
        context_parsed["target_len"]

@staticmethod
def tfrecord_parse_dense_fn(example_proto):
    sparse_feature, sparse_target, feat_len, target_len = Database.tfrecord_parse_sparse_fn(
        example_proto)
    feature = tf.sparse.to_dense(sparse_feature)
    target = tf.sparse.to_dense(sparse_target)
    return feature, target, feat_len, target_len
```

Una vez definida la estructura de deserialización de las muestras, simplemente deben conectarse las operaciones de deserialización con la entrada de las redes, de manera que la operación de lectura de los archivos quede dentro de las operaciones realizadas por el grafo de TensorFlow.

11. ZorzNet

Este modelo se encuentra basado en la red utilizada en el trabajo previo [40]. En la figura 11.1 se presenta un diagrama reducido del grafo de la red, cuyas características principales son las siguientes:

- Las entradas de la red están determinadas por los siguientes *placeholders*:
  - `input_features`: En esta entrada se introducen las matrices de *features*. Dado que los audios son de diferente largo, es necesario completar con ceros para poder agruparlos en una única matriz. Las dimensiones son `[batch_size,T,num_features]`, donde `T` es el largo máximo dentro del *batch*.
  - `input_labels`: En esta entrada se introducen las transcripciones codificadas como índices. En este caso los vectores fueron completados con la etiqueta `-1` para poder agrupar todas las muestras dentro del *batch*. La dimensión de esta entrada es `[batch_size,U]`, donde `U` es la longitud máxima dentro del *batch*.
  - `seq_len`: Si bien los elementos dentro del *batch* fueron completados para que tengan una misma longitud, la redundancia introducida debe ser ignorada durante el entrenamiento. Por esta razón se agrega una entrada que contiene las longitudes originales de los *features* de entrada. Se trata de un escalar por cada elemento del *batch* por lo que la dimensión es `[batch_size]`.
- La primera etapa de la red está compuesta por un perceptrón multicapa llamada `dense_layer_1`. En esta etapa se introducen las matrices de *features* de manera que la dimensión sobre la que la red aplica transformaciones sea la de los *features*, resultando a la salida en matrices de tamaño `[batch_size,T,num_features_2]`. El objetivo de esta capa es el de permitirle a la red encontrar la mejor representación de los *features*, y no restringirse a métodos elegidos manualmente. Por ejemplo, en este trabajo de tesis se comprobó que al utilizar matrices de coeficientes MFCC los resultados eran mejores que al utilizar espectrogramas, lo cual da la idea de que tal vez exista alguna otra representación superior a ambas que puede ser aprendida.
- La siguiente etapa está compuesta por una red recurrente llamada `RNN_cell`. En esta red recurrente se aprenderán las relaciones temporales entre *features*, ya que se correlacionarán los *features* para distintos instantes de tiempo. Esta capa puede ser bidireccional o unidireccional, dependiendo de los parámetros de configuración.
- A la salida de la red recurrente se tiene nuevamente un perceptrón multicapa llamado `dense_layer_2`. Al igual que antes, el objetivo es permitirle a la red encontrar una representación más fácil de clasificar.
- Finalmente se tiene una capa llamada `dense_output` que tiene `num_clases` salidas y que funciona como clasificador. Es por esta razón que su función de activación es la *softmax*.
- A cada capa de la red se le agregó la posibilidad de aplicar *dropout* y *batch normalization*, de manera de mejorar la generalización.
- La precisión de la red se midió de acuerdo con el LER durante el entrenamiento, y se incorporó el WER durante la etapa de validación.

Los hiperparámetros posibles para esta red son:

- `num_classes`: Cantidad de clases de salida, es decir la cantidad de caracteres que se desean representar.
- `num_features`: Cantidad de *features* de entrada.
- `noise_stddev`: Varianza del ruido blanco aditivo.
- `num_reduce_by_half`: Esta parámetro indica cuántas veces debe muestraerse por la mitad la matriz de entrada. Cuando este parámetro vale cero la entrada no se modifica, cuando vale uno la entrada se divide por la mitad, cuando vale dos se divide dos veces por la mitad, y así sucesivamente.
- `dense_layer_1`:
  - `num_dense_layers_1`: Cantidad de capas densas para la primer capa.
  - `num_units_1`: Cantidad de perceptrones por capa.
  - `dense_activations_1`: Funciones de activación para cada capa.
  - `batch_normalization_1`: *Flag* que indica si se debe utilizar *batch normalization*.
  - `batch_normalization`_trainable_1: *Flag* que indica si la capa de *batch normalization* es entrenable.
  - `keep_prob_1`: Probabilidad de que una neurona no sea eliminada por *dropout*.
  - `kernel_init_1`: Función de inicialización para los pesos.
  - bias_init_1: Función de inicialización para el bías.
- Red recurrente:
  - `is_bidirectional`: *Flag* que indica si la red recurrente es bidireccional.
  - `num_cell_units`: Cantidad de celdas LSTM.
  - `cell_activation`: Función de activación de la celda.
  - `keep_prob_rnn`: Probabilidad de que una neurona no sea eliminada por *dropout*.
  - `rnn_batch_normalization`: *Flag* que indica si se debe utilizar *batch normalization*.
- `dense_layer_2`:
  - `num_dense_layers_2`: Cantidad de capas densas para la primer capa.
  - `num_units_2`: Cantidad de perceptrones por capa.
  - `dense_activations_2`: Funciones de activación para cada capa.
  - `batch_normalization_2`: *Flag* que indica si se debe utilizar *batch normalization*.
  - `batch_normalization`_trainable_2: *Flag* que indica si la capa de *batch normalization* es entrenable.
  - `keep_prob_2`: Probabilidad de que una neurona no sea eliminada por *dropout*.
  - `kernel_init_2`: Función de inicialización para los pesos.
  - bias_init_2: Función de inicialización para el bías.
- `dense_regularizer`: Coeficiente de regularización de los pesos de las capas densas.
- `rnn_regularizer`: Coeficiente de regularización de los pesos de las capas recurrentes.
- `beam_width`: Cantidad de *beams* utilizados para la decodificación. Si vale cero se utiliza *greedy decoder*, mientras que si es mayor a cero se utiliza *beam search*.
- `learning_rate`: Coeficiente de aprendizaje.
- *Learning rate decay:*
  - `use_learning_rate_decay`: *Flag* que indica si se debe utilizar *learning rate decay*.

- `learning_rate_decay_steps`: Cantidad de iteraciones sobre las que se reduce el *learning rate*.
- `learning_rate_decay`: Factor por el que se reduce el *learning rate*.

- `clip_gradient`: Factor de *clip gradient*.
- `optimizer`: Función de optimización. Las posibilidades son `'rms'`, `'adam'`, `'momentum'` y `'sgd'`.

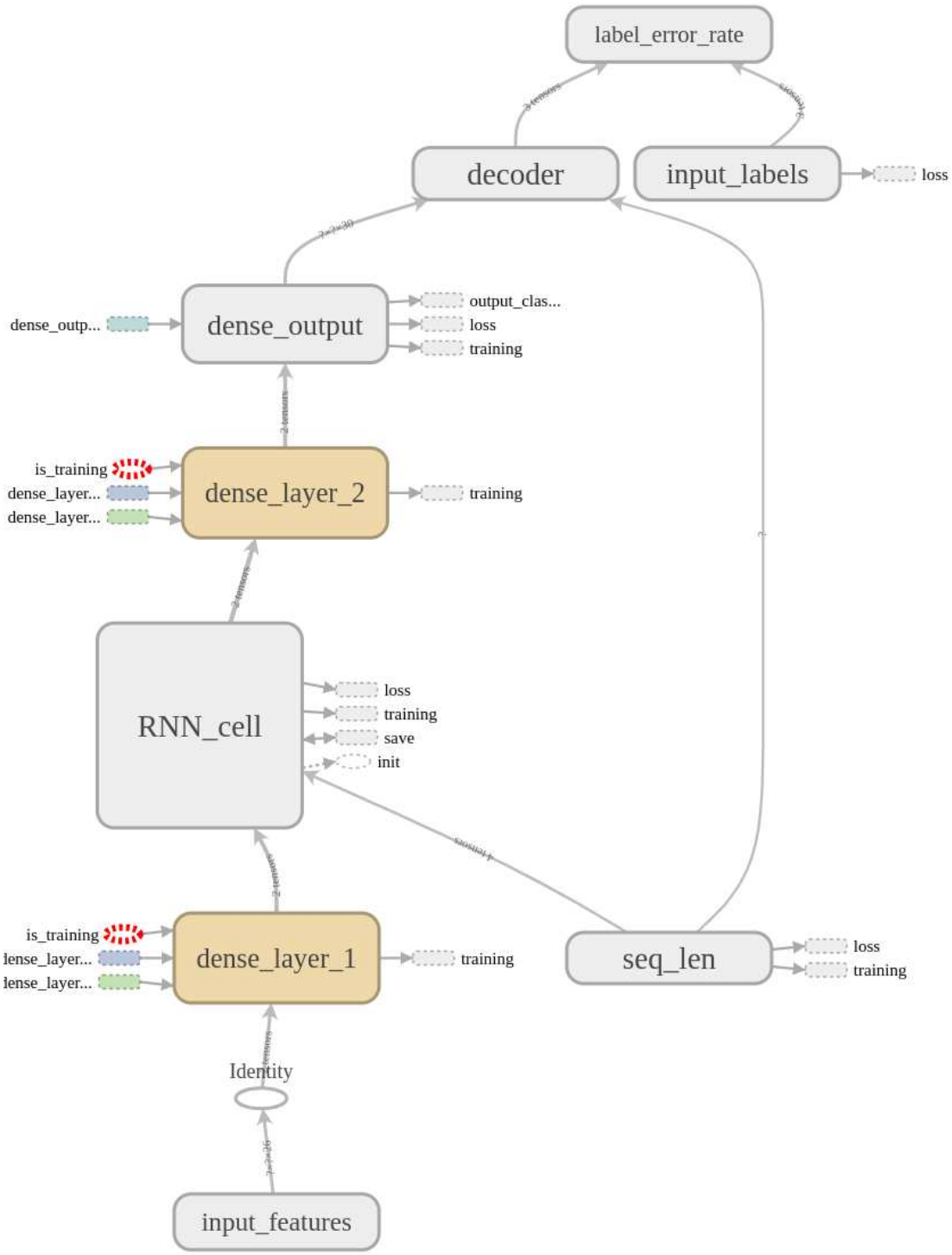


Figura 11.1: Grafo reducido de la red ZorzNet obtenido mediante TensorBoard.

La red hasta aquí planteada es capaz de traducir matrices de *features* de tamaño `[T,num_features]` en matrices

de probabilidad de etiquetas de tamaño `[T,num_classes]`. Sin embargo, las secuencias de etiquetas de entrenamiento no se encuentran alineadas con los audios, y es por esto que se debió incorporar CTC al modelo. Para esto se necesitan tres módulos ya implementados dentro de las operaciones de *TensorFlow*:

- Función de costo: La función de costo introducida en 6.2 se encuentra implementada como:

```
tf.nn.ctc_loss(
    labels,
    inputs,
    sequence_length,
    preprocess_collapse_repeated=False,
    ctc_merge_repeated=True,
    ignore_longer_outputs_than_inputs=False,
    time_major=True
)
```

  Los argumentos de esta función son:

  - `labels`: Secuencia de etiquetas de las transcripciones. Cada etiqueta debe ser convertida a un índice dentro de un diccionario (e.g. 'a'→1, 'b'→2, etc.) donde se agrega un símbolo más que se corresponde al *blank*. El índice de esta etiqueta es agregado como último elemento de los símbolos permitidos.
  - `inputs`: Matriz de probabilidades de etiquetas por cada instante de tiempo. Una observación importante es que **la función de costo CTC provista por TensorFlow aplica de manera interna la función *softmax***. Esto obliga a la capa anterior a utilizar un función de activación lineal, ya que en caso contrario se estaría aplicando una activación del tipo $\text{softmax}(\text{softmax}(x))$, lo cual no lleva a buenos resultados. Debido a esto, la salida de la capa de clasificación se dividió en dos: una con activación lineal para utilizarla en la función de costo, y otra con activación *softmax* para utilizarla durante la decodificación.
  - `sequence_length`: Longitud de cada una de las secuencias de *features* dentro del *batch*. Dado que cada secuencia de entrada debe ser completada con ceros para poder agruparlas dentro de un *batch*, es necesario que el algoritmo sepa cuándo detenerse sin iterar sobre valores agregados de manera artificial.
  - `preprocess_collapse_repeated`: En caso de ser `True`, aquellas etiquetas continuas repetidas serán unificadas en una única etiqueta, previo a la aplicación del algoritmo CTC.
  - `ctc_merge_repeated`: Como se mencionó en la sección 6.1 a la salida del decodificador se unifican las etiquetas repetidas que no estén separadas por un *blank*. Si esta variable es `False` se desactiva este comportamiento.
  - `ignore_longer_outputs_than_inputs`: Como se describió previamente, el algoritmo CTC sólo puede ser aplicado si las secuencias de entrada son más largas que las de salida. Si esta variable es `True` entonces cada vez que se deba medir el costo de una secuencia cuya entrada es mas corta que la salida, ésta será ignorada y se propagará un gradiente nulo. En caso de que se utilice `False` es necesario eliminar todas las secuencias problemáticas, ya que el algoritmo genera una excepción durante la ejecución.
  - `time_major`: Si es `True` entonces los *features* de entrada tienen la forma `[T,batch_size,num_features]`, mientras que si es `False` entonces deben ser `[batch_size,T,num_features]`.

- Función de decodificación: Se tienen dos alternativas:

  - *Greedy decoder*: Los argumentos tienen la misma interpretación que en la función de costo

```
tf.nn.ctc_greedy_decoder(
    inputs,
    sequence_length,
    merge_repeated=True
)
```

  - *Beam search decoder*: Los argumentos tienen la misma interpretación que en la función de costo

```
tf.nn.ctc_beam_search_decoder(
    inputs,
    sequence_length,
    beam_width=100,
    top_paths=1,
    merge_repeated=True
)
```

Para el caso en que `beam_width=1` se tiene un algoritmo equivalente a `ctc_greedy_decoder`, pero resulta menos eficiente que utilizar la función anterior.

Para aplicar estas funciones se procedió de la siguiente manera:

- Se crearon dos salidas diferentes de la capa `dense_output`: una con una función de activación lineal y otra con *softmax*.
- A la entrada de la función de costo `tf.nn.ctc_loss` se utilizó la salida con la activación lineal, mientras que para los decodificadores se utilizó la salida *softmax*.
- La función de costo global entonces es la suma de `tf.nn.ctc_loss` más un término de regularización que aplica la norma L2 a los pesos de los perceptrones multicapa de la red.

Siguiendo esta estructura fue posible obtener una red capaz de aprender la alineación entre los audios y las etiquetas, así como también de realizar predicciones sobre secuencias no vistas durante el entrenamiento.

11.1. Problemas con *batch normalization*

Las redes aquí presentadas utilizan *batch normalization*, ya que se ha comprobado que acelera la convergencia durante el entrenamiento y mejora la generalización. Para esto se utiliza la siguiente función provista por TensorFlow:

```
tf.layers.batch_normalization(
    inputs,
    training,
    trainable,
    ...,
)
```

Entre todos los argumentos que admite esta función, interesan principalmente los siguientes dos:

- `training`: Se trata de un argumento *booleano* que indica si se está en etapa de entrenamiento o de inferencia. Durante la etapa de entrenamiento, se utiliza la fórmula presentada en la sección 5.5 usando la media y la varianza del *batch* de muestras, mientras que durante la etapa de inferencia se utilizan la media y la varianza aprendidas durante el entrenamiento.
- `trainable`: Se trata de un argumento *booleano* que indica si las variables internas de la media y la varianza son entrenables o no. Esto evita que los datos estadísticos utilizados en esta capa se vean modificados durante el entrenamiento, en aquellos casos donde se desee mantenerlos fijos. Esto es muy común cuando se utiliza una red pre-entrenada para realizar una tarea diferente a la original. En estos casos, generalmente se suele entrenar la última capa de la red y se mantienen fijos los pesos de las demás capas.

De acuerdo con esto, se puede observar que durante la etapa de entrenamiento la media y la varianza internas de esta capa no son utilizadas, pero en caso de que `trainable=True`, dichas variables deben ser actualizadas junto con los demás parámetros de la red. El problema es que la operación de actualización de TensorFlow se realiza sobre el grafo de cómputo, el cual no contiene a las variables de la media y la varianza por no estar siendo utilizadas. De acuerdo con la documentación, la actualización de estas variables se encuentra en `tf.GraphKeys.UPDATE_OPS`, por lo que se debe actualizar la operación de entrenamiento de la siguiente manera:

```
update_ops = tf.get_collection(tf.GraphKeys.UPDATE_OPS)
with tf.control_dependencies(update_ops):
    train_op = optimizer.minimize(loss)
```

Sin embargo, este mecanismo presentó algunos problemas. Al utilizar una capa con *batch normalization* configurada como entrenable a la salida de una red recurrente, el cómputo del grafo queda congelado. Según se encontró en varios foros esto se trata de un problema de la biblioteca TensorFlow, ya que la operación de actualización debe realizarse antes de la operación de entrenamiento, pero debido a retardos en los cómputos, no se detecta que dicha operación haya finalizado. Por lo tanto, se debió modificar esta operación según el comentario `https://github.`

com/tensorflow/tensorflow/issues/19568#issuecomment-416393783, donde se fuerza a esperar la finalización de las actualizaciones:

```
loss = tf.tuple([loss], control_inputs=tf.get_collection(tf.GraphKeys.UPDATE_OPS))[0]
grads_and_vars = optimizer.compute_gradients(loss)
train_op = optimizer.apply_gradients(grads_and_vars, global_step)
```

Utilizando esta modificación fue posible entrenar los parámetros internos de las capas de *batch normalization* sin importar su ubicación dentro del grafo.

12. Redes LAS

En la figura 12.1 se presenta el grafo genérico utilizado para este modelo. Las características principales de este modelo son:

- Las entradas para entrenar este modelo son las siguientes:
  - `input_features`: Al igual que en la red anterior, la entrada de *features* se corresponde con matrices de tamaño `[batch_size, T, num_features]`.
  - `targets`: En este caso además de incluir los símbolos que se desean aprender, es necesario agregar al comienzo y al final dos nuevas etiquetas que indican el inicio de una secuencia (`<sos>`) y el final (`<eos>`) respectivamente. A diferencia de la red CTC, las predicciones se generarán de manera dinámica sin requerir una alineación, por lo que es necesario emitir el símbolo de `<eos>` para indicar que la predicción ha finalizado.
  - `input_features_length` y `targets_length`: A diferencia de la red CTC, en este caso también se requiere introducir la longitud de la secuencia de salida, y no solo la de entrada.

  Es importante aclarar que todos los elementos dentro del *batch* deben ser rellenados con redundancia para poder agruparlos en sus respectivos tensores.
- `dense_layer_1`: Nuevamente se agrega un perceptrón multicapa cuyo objetivo es el de ajustar la representación de los *features* en la entrada a una nueva que puede resultar mejor para este problema.
- `listener`: Como se presentó previamente, consiste en una red recurrente bidireccional piramidal, donde cada capa que la compone reduce la dimensión temporal a la mitad.
- `dense_layer_2`: Se trata de un nuevo perceptrón multicapa cuyo objetivo es correlacionar *features*.
- `attention`: Este módulo es el que contiene el mecanismo de atención, pudiéndose elegir entre *soft* y *hard attention*. Como se explicó en la sección 7.1.2, se trata de una red recurrente que introduce las variables de contexto, y así asignar un peso a las salidas en distintos instantes de tiempo.
- `speller`: Está compuesto por el decodificador: *greedy* o *beam search*. La salida del mecanismo de atención está compuesta por las probabilidades de cada etiqueta en dicho instante de tiempo[22], por lo que es necesario decodificar esas probabilidades.
- Nuevamente se utilizó el LER durante el entrenamiento, y se incorporó el WER durante la etapa de validación.

Para el entrenamiento es necesario definir una función de costo. En este caso se utilizó la función de costo que calcula la *cross-entropy* para secuencias, lo cual fue explicado en las secciones previas:

```
loss = tf.contrib.seq2seq.sequence_loss(
    logits,
    targets,
    target_weights
)
```

Los argumentos para esta función de costo son:

[22] El tiempo en este caso no coincide con el de los *features* de entrada, sino con la indexación de las predicciones de la red.

- `logits`: Se trata de un tensor de tamaño `[batch_size, U, num_clases]` que contiene las predicciones realizadas por el mecanismo de atención, siendo `U` el largo de la predicción, `num_clases` la cantidad de símbolos de salida y `batch_size` es la cantidad de muestras que se procesan en paralelo.
- `targets`: Las etiquetas reales que se desean predecir.
- `target_weights`: Este parámetro permite asignar un peso a cada una de las predicciones de la secuencia[23]. Dado que se introduce redundancia para que todas las secuencias del *batch* sean del mismo largo, es necesario indicarle de alguna manera al mecanismo de atención que las muestras que está procesando corresponden a dicha redundancia. En este caso se le asignó un peso igual a la unidad para aquellos instantes de tiempo válidos, mientras que se asignó un peso nulo a las redundancias agregadas de manera artificial, de manera que puedan ser ignoradas a la hora de realizar predicciones.

Al igual que en la red CTC, se incluyó a la función de costo un término de regularización L2 para los pesos de los perceptrones multicapa, nuevamente dejando de lado los pesos de las redes recurrentes.

Finalmente, para la decodificación se definieron dos posibles métodos:

- *Greedy decoder*:

```
def greedy_decoder(inputs, embedding, start_token, end_token, initial_state, projection_layer,
    batch_size):
    start_token = tf.fill([batch_size], start_token)    # Identificador de <sos>
    helper = tf.contrib.seq2seq.GreedyEmbeddingHelper(embedding, start_token, end_token)
    decoder = tf.contrib.seq2seq.BasicDecoder(inputs, helper, initial_state, output_layer=
        projection_layer)
    return decoder
```

  Los parámetros de esta función son:

  - `start_token` es el identificador de `<sos>` definido previamente. Se lo debe apilar para convertirlo en un tensor en lugar de ser un escalar.
  - El `helper` es la clase que implementa el algoritmo de *greedy decoding*. Para esto recibe una función `embedding` que recibe un tensor de identificadores y retorna su representación. Esto permite combinar el sistema con representaciones diferentes a la *one hot* utilizada en este trabajo.
  - El `decoder` recibe la salida del mecanismo de atención, el `helper`, un estado inicial y una capa de perceptrones para aplicar a su salida, y retorna un decodificador que debe ser ejecutado de la siguiente manera para obtener sus predicciones:

```
decoder_outputs, final_context_state, final_sequence_length = tf.contrib.seq2seq.
    dynamic_decode(
    decoder,
    maximum_iterations=maximum_iterations
)
```

- *Beam search decoder*:

```
def beam_search_decoder(input_cell, embedding, start_token, end_token, initial_state, beam_width,
    projection_layer, batch_size):
    start_token = tf.fill([tf.div(batch_size, beam_width)], start_token)
    decoder = tf.contrib.seq2seq.BeamSearchDecoder(
        cell=input_cell,
        embedding=embedding,
        start_tokens=start_token,
        end_token=end_token,
        initial_state=initial_state,
        beam_width=beam_width,
        output_layer=projection_layer)
    return decoder
```

  La interpretación de los argumentos para este decodificador es similar al del caso anterior.

[23] No se debe confundir con el peso asignado por el mecanismo de atención, ya que eso se aplica de manera automática.

A diferencia de la red anterior, donde se aplica el algoritmo de CTC a los *logits* obtenidos a la salida para decodificar la secuencia predicha, los modelos de atención (y en general los modelos del tipo *sequence-to-sequence*) requieren un módulo de decodificación diferente. Si bien en esta sección se mencionaron los decodifcadores `GreedyEmbeddingHelper` y `BeamSearchDecoder`, éstos solo sirven para la etapa de inferencia. Durante el entrenamiento TensorFlow ofrece dos posibles mecanismos:

- `tf.contrib.seq2seq.TrainingHelper`: Al realizar la decodificación, en lugar de utilizar la salida del instante anterior como entrada para la predicción actual, se utiliza el verdadero valor de la salida del instante anterior (por eso esto solo puede ser utilizado durante el entrenamiento, ya que durante la validación no está disponible el *ground truth*). Esto genera un problema conocido como ***Exposure Problem***, el cual provoca que durante el entrenamiento la red no aprenda de sus propias salidas, sino que lo hace directamente de su secuencia objetivo, lo cual genera que durante la inferencia, donde la secuencia objetivo no es conocida, no se logren buenos resultados. Para solucionar este problema, existe el siguiente mecanismo.

- `tf.contrib.seq2seq.ScheduledEmbeddingTrainingHelper`: A diferencia del caso anterior, este método recibe un argumento llamado `sampling_probability` que define la probabilidad de que se tome como entrada para el instante actual el valor de la secuencia objetivo (como en el método anterior) o se tome a partir de la salida del instante anterior. Esto permite que la red sea expuesta a sus propias salidas durante el entrenamiento, mejorando su capacidad de realizar buenas predicciones en la etapa de inferencia.

Dependiendo del valor de `sampling_probability` definido en los hiperparámetros de la red, se utilizará un mecanismo u otro: Si la probabilidad es nula, entonces se utilizan únicamente los valores de *ground truth* (usando `TrainingHelper`), mientras que si es distinta de cero se utiliza `ScheduledEmbeddingTrainingHelper`, donde se escoge con probabilidad $(1 - \texttt{sampling\_prob})$ los valores de *ground truth* y con probabilidad `sampling_prob` las salidas del instante anterior.

Los posibles hiperparámetros para esta red entonces son:

- `num_classes`: Cantidad de clases de salida, es decir la cantidad de caracteres que se desean representar más los símbolos de `<sos>` y `<eos>`.

- `num_features`: Cantidad de *features* de entrada.

- `num_embeddings`: Tamaño del *embedding* de las transcripciones. Esto fue agregado pero no se probó su correcto funcionamiento.

- `sos_id`: Índice asignado al caracter `<sos>`.

- `eos_id`: Índice asignado al caracter `<eos>`.

- `noise_stddev`: Varianza del ruido blanco aditivo.

- `num_reduce_by_half`: Esta parámetro indica cuántas veces debe muestrearse por la mitad la matriz de entrada. Cuando este parámetro vale cero la entrada no se modifica, cuando vale uno la entrada se divide por la mitad, cuando vale dos se divide dos veces por la mitad, y así sucesivamente.

- `dense_layer_1`:

  - `num_dense_layers_1`: Cantidad de capas densas para la primer capa.
  - `num_units_1`: Cantidad de perceptrones por capa.
  - `dense_activations_1`: Funciones de activación para cada capa.
  - `batch_normalization_1`: *Flag* que indica si se debe utilizar *batch normalization*.
  - `batch_normalization`_trainable_1: *Flag* que indica si la capa de *batch normalization* es entrenable.
  - `keep_prob_1`: Probabilidad de que una neurona no sea eliminada por *dropout*.
  - `kernel_init_1`: Función de inicialización para los pesos.
  - bias_init_1: Función de inicialización para el bías.

- *Listener*:
  - `listener_num_layers`: Cantidad de capas del *listener*. Se debe tener en cuenta que cada capa reduce la entrada por la mitad.
  - `listener_num_units`: Cantidad de perceptrones por capa del *listener*.
  - `listener_activation_list`: Funciones de activación para cada capa del *listener*.
  - `listener_keep_prob_list`: Probabilidades de que una neurona no sea eliminada por *dropout*.
- `dense_layer_2`:
  - `num_dense_layers_2`: Cantidad de capas densas para la primer capa.
  - `num_units_2`: Cantidad de perceptrones por capa.
  - `dense_activations_2`: Funciones de activación para cada capa.
  - `batch_normalization_2`: *Flag* que indica si se debe utilizar *batch normalization*.
  - `batch_normalization_`trainable_2: *Flag* que indica si la capa de *batch normalization* es entrenable.
  - `keep_prob_2`: Probabilidad de que una neurona no sea eliminada por *dropout*.
  - `kernel_init_2`: Función de inicialización para los pesos.
  - bias_init_2: Función de inicialización para el bías.
- Mecanismo de atención:
  - `attention_type`: Tipo de mecanismo de atención. Las posibilidades son `'luong'` y `'bahdanau'`.
  - `attention_num_layers`: Cantidad de capas del mecanismo de atención.
  - `attention_size`: Tamaño de salida del mecanismo de atención.
  - `attention_units`: Cantidad de neuronas por capa del mecanismo de atención.
  - `attention_activation`: Funciones de activación por cada capa.
  - `attention_keep_prob`: Probabilidad de que una neurona no sea eliminada por *dropout*.
- `kernel_regularizer`: Coeficiente de regularización de los pesos de las capas densas.
- `beam_width`: Cantidad de *beams* utilizados para la decodificación. Si vale cero se utiliza *greedy decoder*, mientras que si es mayor a cero se utiliza *beam search*.
- `sampling_probability`: Probabilidad utilizada en el `TrainingHelper` o el `ScheduledEmbeddingTrainingHelper`.
- `learning_rate`: Coeficiente de aprendizaje.
- *learning rate decay:*
  - `use_learning_rate_decay`: *Flag* que indica si se debe utilizar *learning rate decay*.
  - `learning_rate_decay_steps`: Cantidad de iteraciones sobre las que se reduce el *learning rate*.
  - `learning_rate_decay`: Factor por el que se reduce el *learning rate*.
- `clip_gradient`: Factor de *clip gradient*.
- `optimizer`: Función de optimización. Las posibilidades son `'rms'`, `'adam'`, `'momentum'` y `'sgd'`.

Se puede ver que muchos de estos hiperparámetros coinciden con los del modelo anterior, por lo que se pudieron reutilizar varias estructuras de código.

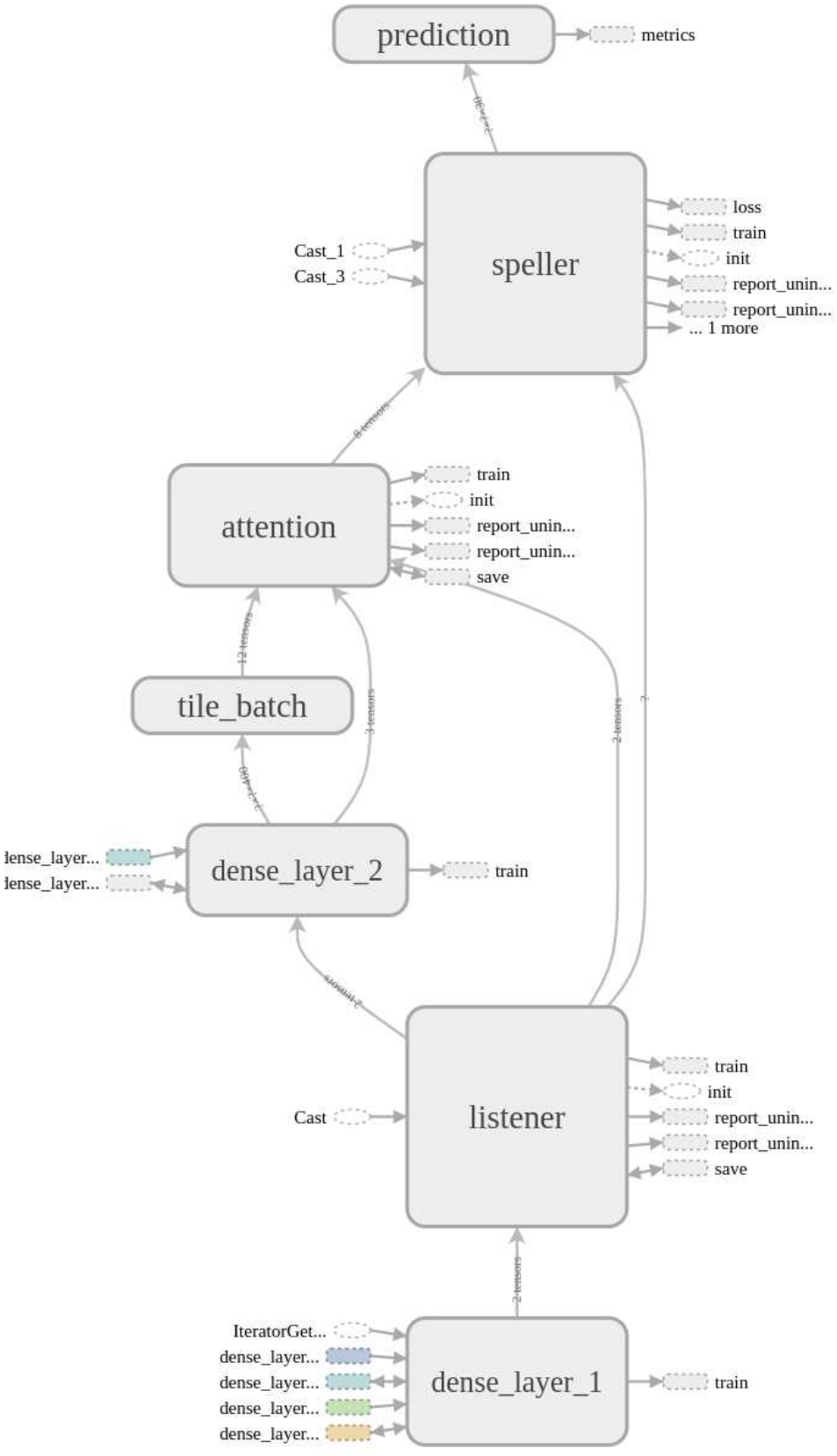


Figura 12.1: Grafo reducido de la red LAS obtenido mediante TensorBoard.

Parte IV

Resultados y discusión

13. Resultados

Durante este trabajo se debieron ejecutar muchos modelos distintos de redes para poder encontrar un conjunto de hiperparámetros que permitieran alcanzar errores aceptables. Dado que los tiempos de entrenamiento son elevados se debieron realizar pruebas a pequeña escala (con menos datos) para analizar el impacto de la modificación de cada hiperparámetro. Esto permite tener una idea aproximada de la influencia de cada uno, pero no necesariamente asegura que sean los mejores.

En esta sección se describirán los resultados obtenidos junto con su correspondiente análisis.

13.1. Generación de datos

Como se mencionó en la sección 10.1 se observaron mejores resultados al utilizar coeficientes MFCC en vez de espectogramas. Si bien estas pruebas fueron realizadas con pocos datos, todos los modelos planteados convergieron más rápidamente y con un error menor al utilizar coeficientes MFCC.

Una vez definida la utilización de los coeficientes MFCC se realizaron pruebas similares para comparar los resultados entre los coeficientes convencionales y la representación utilizada por DeepSpeech. En este caso, el incremento de dimensionalidad de DeepSpeech no solo redujo el error, sino que también se redujo la cantidad de iteraciones requeridas. Sin embargo, este método introdujo dos desventajas:

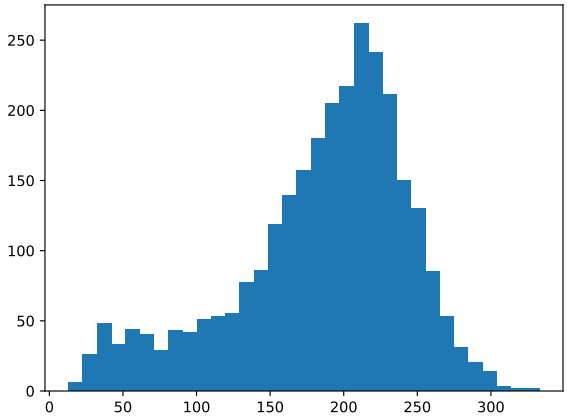
- El tiempo de cómputo por iteración se incrementó. Sin embargo, dado que se requieren menos iteraciones, el tiempo de entrenamiento resulta similar al necesario cuando se utilizan directamente los coeficientes.
- La complejidad computacional se incrementó. Dado que se aumentó la dimensionalidad de la representación, se requiere un mayor uso de memoria, haciendo prohibitivo montar en memoria todas las muestras. Esto fue solucionado mediante la utilización de archivos *tfrecords*.

Dados estos resultados, se decidió utilizar la siguiente configuración de *features*:

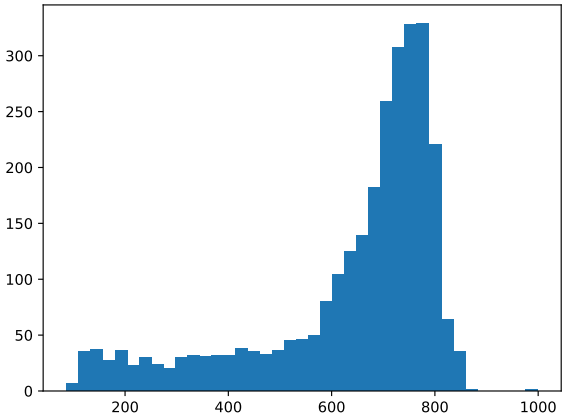
```
feature_config = FeatureConfig()
feature_config.feature_type = 'deep_speech_mfcc'    # 'mfcc', 'spec', 'log_spec', 'deep_speech_mfcc'
feature_config.nfft = 1024
feature_config.winlen = 20      # 20ms
feature_config.winstride = 10   # 10ms
feature_config.preemph = 0.98
feature_config.num_filters = 40
feature_config.num_ceps = 26
feature_config.mfcc_window = np.hanning
feature_config.lowfreq = 0
feature_config.highfreq = None  # Se usa fs/2
feature_config.num_context = 9
```

Mediante esta configuración se obtuvieron matrices de tamaño `[(2*num_context+1)*num_ceps, T]=[494, T]`.

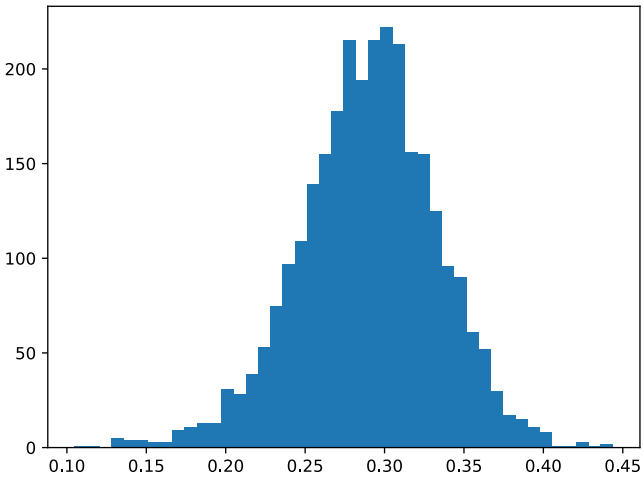
Un aspecto importante a analizar es la distribución de las longitudes de los audios y las transcripciones, ya que un factor de comparación entre los distintos modelos es la capacidad de reconocer dependencias de largo plazo. En la figura 13.1 se presentan histogramas de las relaciones de longitudes entre la entrada y la salida de la red. En la figura 13.1c se puede ver que todas las matrices de *features* generadas tienen al menos el doble de la longitud que la entrada, y que la gran mayoría de los audios resulta tres veces más largo que la entrada. Por lo tanto, no es posible aplicar modelos que reduzcan la longitud de entrada más de dos veces sin que se comprometan las estrategias de alineación: CTC requiere que la longitud de entrada sea mayor que la de la salida, y el modelo de atención no es capaz de emitir salidas más largas que su entrada, por lo que tampoco es capaz de alinear correctamente en ese caso.



(a) Histograma de la distribución de las longitudes de las transcripciones.



(b) Histograma de la distribución de las longitudes de los *features*.



(c) Histograma de la distribución del *ratio* entre la longitud de las transcripciones y la longitud de los *features*.

Figura 13.1: Propiedades de las muestras utilizadas para generar el conjunto de entrenamiento. Los datos se obtuvieron utilizando 100 horas de LibriSpeech.

Es importante remarcar que la distribución de longitudes de los datos utilizados no se encuentra distribuida sobre un amplio rango de longitudes, lo cual explica algunos de los resultados que serán presentados a continuación.

13.2. ZorzNet

Luego de realizar distintas pruebas a pequeña escala, los hiperparámetros utilizados para el entrenamiento completo fueron:

```
network_data.num_classes = ClassicLabel.num_classes - 1 # a-z
network_data.num_features = 494 # Cantidad de features de entrada
network_data.noise_stddev = 0.1 # Varianza del ruido blanco aditivo
network_data.num_reduce_by_half = 0 # Cantidad de veces que se reduce la entrada por la mitad

network_data.num_dense_layers_1 = 1
network_data.num_units_1 = [400] * network_data.num_dense_layers_1
network_data.dense_activations_1 = [tf.nn.relu] * network_data.num_dense_layers_1
network_data.batch_normalization_1 = True
network_data.batch_normalization_trainable_1 = True
network_data.keep_prob_1 = [0.6] * network_data.num_dense_layers_1
network_data.kernel_init_1 = [tf.truncated_normal_initializer(mean=0, stddev=0.1)] * network_data.
    num_dense_layers_1
network_data.bias_init_1 = [tf.zeros_initializer()] * network_data.num_dense_layers_1

network_data.is_bidirectional = True
network_data.num_cell_units = [512] * 2
network_data.cell_activation = [tf.nn.tanh] * 2
network_data.keep_prob_rnn = None
network_data.rnn_batch_normalization = False
```

```
network_data.num_dense_layers_2 = 1
network_data.num_units_2 = [150] * network_data.num_dense_layers_2
network_data.dense_activations_2 = [tf.nn.relu] * network_data.num_dense_layers_2
network_data.batch_normalization_2 = True
network_data.batch_normalization_trainable_2 = True
network_data.keep_prob_2 = [0.6] * network_data.num_dense_layers_2
network_data.kernel_init_2 = [tf.truncated_normal_initializer(mean=0, stddev=0.1)] * network_data.
    num_dense_layers_2
network_data.bias_init_2 = [tf.zeros_initializer()] * network_data.num_dense_layers_2

network_data.dense_regularizer = 0.5
network_data.rnn_regularizer = 0.0

network_data.beam_width = 0     # 0 -> greedy_decoder, >0 -> beam_search

network_data.learning_rate = 0.001
network_data.use_learning_rate_decay = True
network_data.learning_rate_decay_steps = 1000
network_data.learning_rate_decay = 0.98

network_data.clip_gradient = 5
network_data.optimizer = 'adam'       # 'rms', 'adam', 'momentum', 'sgd'
network_data.momentum = None
```

Algunas observaciones sobre estos hiperparámetros elegidos:

- Se añadió ruido blanco a los *features* de entrada para simular un incremento artificial de los datos de entrenamiento. Debido a los tiempos de entrenamiento, no se repitieron las pruebas sin ruido aditivo, por lo que no se pudo comprobar la efectividad de este parámetro.
- Se incorporó una variable que indica cuántas veces debe reducirse la entrada por la mitad antes de ingresar a la red. Luego de varias pruebas, se comprobó que los resultados eran mejores si se mantenía la relación de longitudes originales entre la entrada y la salida[24].
- Si bien uno esperaría que la decodificación *beam search* explorara mejor las posibles secuencias, se observó que *greedy decoder* arrojó mejores resultados (además de que se redujeron considerablemente los tiempos de validación). Esto puede deberse a un error en la implementación, pero dado que los resultados con *greedy decoder* fueron lo suficientemente buenos, no se intentó encontrar la verdadera causa.
- No se realizaron suficientes pruebas para evaluar la efectividad de *learning rate decay*, por lo que se eligió un valor razonable de acuerdo a la cantidad de iteraciones necesarias para observar reducciones en el error.

Algunas conclusiones preliminares obtenidas con los resultados son:

- Se demostró que las funciones de activación **ReLU** son muy eficientes y rápidas. Sin embargo, cuando fueron utilizadas en la red recurrente los resultados no fueron buenos. Una posible explicación es que la función ReLU solo arroja valores positivos, lo cual podría estar limitando la propagación de la información al desdoblar la red recurrente.
- La utilización de *dropout* en la red recurrente arrojó resultados adversos, por lo que solo se aplicó a los perceptrones multicapa. Esto se debe a un error de implementación, ya que en cada instante de tiempo (al desdoblar la red recurrente) se elige un nuevo conjunto de neuronas para descartar, evitando que la salida correspondiente a la memoria se propague correctamente. La solución a esto consiste en descartar las mismas neuronas en cada instante de tiempo, lo cual no fue implementado en este trabajo.
- Al utilizar ***gradient clipping*** se observó no solo una mejoría en los resultados, sino también un gran incremento en la velocidad de convergencia. Esto comprueba la utilidad de esta técnica en redes altamente profundas, principalmente aquellas que utilizan redes recurrentes.
- El aumento de la cantidad de capas densas no mejoró los resultados, a pesar de que en la mayoría de los casos suele ser cierto. Una posible razón es que se haya incurrido en un problema de gradiente evanescente que no pudo ser resuelto mediante las técnicas de regularización aplicadas.

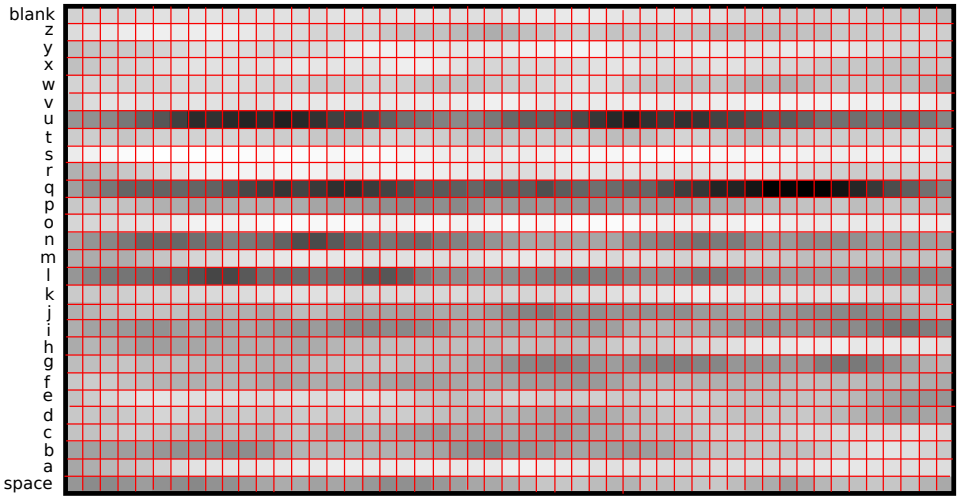
[24] Durante la generación de las matrices de *features* se descartan la mitad de las muestras, por lo que este parámetro se aplica sobre entradas ya submuestreadas.

- La regularización solo se realizó sobre los pesos de los perceptrones multicapa, dejando de lado los pesos de las redes recurrentes. Si bien en principio arrojaba peores resultados, es posible que tal vez las redes recurrentes regularizadas requirieran mucho más tiempo de entrenamiento. Sin embargo, como se observó buena generalización, no se profundizó en este tema.

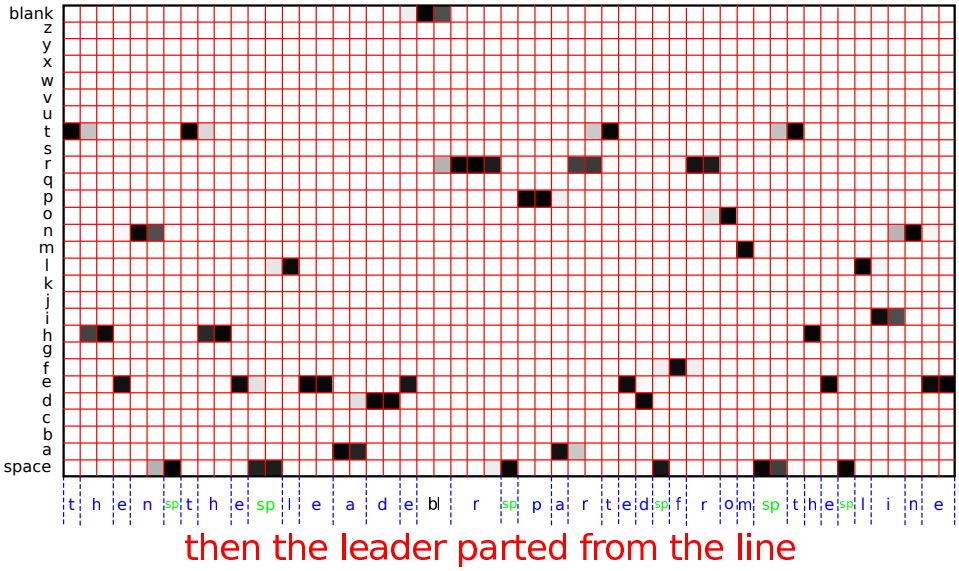
Al iniciar el entrenamiento de la red, ésta no tiene ningún tipo de conocimiento sobre la distribución de los datos, y por lo tanto las salidas pueden considerarse completamente aleatorias. Por ejemplo, en la figura 13.2a se observa que la distribución de los caracteres no sigue ningún patrón en especial, y los valores obtenidos provienen únicamente de la inicialización aleatoria de los pesos. Sin embargo, a medida que se van realizando iteraciones sobre los datos, las distribuciones de los caracteres a la salida comienzan a parecerse a la real. Esto se puede observar en la figura 13.2b, donde se presentan las probabilidades predichas para cada instante de tiempo, luego de que la red fuera entrenada. Para obtener dichos resultados se entrenó a la red con pocos audios, y se realizó una validación con uno de los audios utilizados durante el entrenamiento, y es por esta razón que la predicción no presentó errores. Por otra parte, la longitud de la entrada fue reducida por la mitad[25], de manera que la diferencia de longitudes entre la entrada y la salida no fue considerable. Es interesante remarcar cómo algunos símbolos tienen probabilidades similares de ser los correctos, lo cual de alguna manera muestra que las transiciones entre caracteres no son abruptas, y que la alineación de secuencias "a mano" no siempre resulta trivial.

Un análisis similar se puede hacer con los resultados de la figura 13.2c. En este caso el proceso de entrenamiento fue igual al descrito anteriormente, pero la longitud de entrada no fue reducida, por lo que la matriz de entrada es el doble de la anterior. Dado que se utilizaron pocas muestras, la red fue capaz de aprenderlas de memoria, por lo que la alineación aprendida no necesariamente resulta ser la real. Si la alineación aprendida fuera la correcta, al duplicar la cantidad de muestras, se esperaría observar que la duración de cada clase se incremente. Sin embargo, queda claro en la figura 13.2c que ésto no ocurre, y es por eso que aparecen resultados inesperados, como por ejemplo que la duración del caracter h resulta considerable frente a las demás clases.

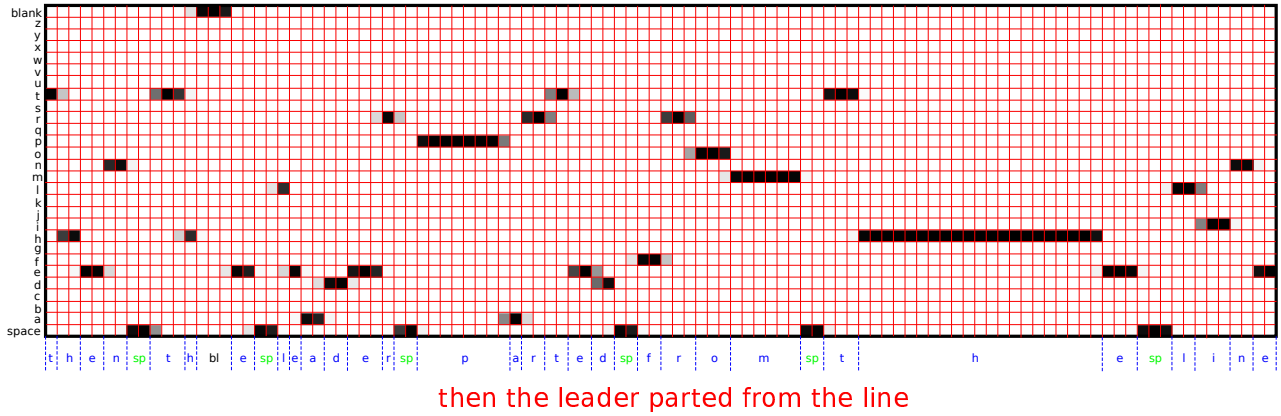
[25] Además del muestreo realizado en la generación de los *features*.



(a) Distribución de los caracteres para cada instante de tiempo cuando la red no se encuentra entrenada.



(b) Distribución de los caracteres para cada instante de tiempo luego de entrenar correctamente la red.



(c) Distribución de los caracteres para cada instante de tiempo luego de entrenar correctamente la red. A diferencia de la figura 13.2b, donde la entrada se la redujo dos veces a la mitad, en este caso sólo se descartaron la mitad de las muestras.

Figura 13.2: Matrices de alineación de salida de la red ZorzNet. El eje vertical representa los caracteres, el eje horizontal representa el tiempo y la intensidad representa la probabilidad de que el caracter sea el correcto.

Para poder observar una correcta alineación (o al menos más cercana a la real) entre la entrada y la salida, es necesario que la red aprenda la verdadera estructura de los datos, lo cual se logra incrementando la generalización. En la figura 13.3 se tienen los resultados obtenidos al utilizar la misma red anterior, pero en este caso entrenada

con 100 horas de audios de LibriSpeech y validada con el mismo audio anterior, pero sin que éste fuera parte del conjunto de entrenamiento.

Una primer observación a los resultados de la figura 13.3 es que aparece al inicio y al final el símbolo *blank* indicando los silencios en el audio. Por otra parte, el *blank* se encuentra mucho más presente que en la figura 13.2c, principalmente entre transiciones. Esto puede estar indicando que ante la dificultad de encontrar un punto específico donde se pasa de un caracter al siguiente, a la red le resulta menos costoso utilizar el símbolo *blank* para representar esas *uterancias* que no se corresponden a un caracter en particular. Este comportamiento no se observaba en el caso anterior, ya que la muestra había sido aprendida de memoria, y no hizo falta encontrar una representación genérica que pudiera ser aplicada a todos los audios de igual manera.

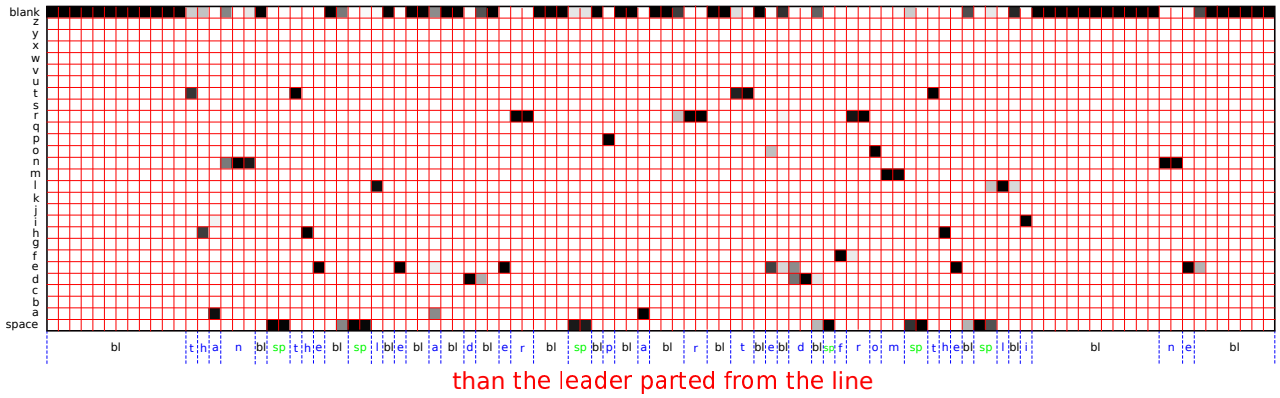


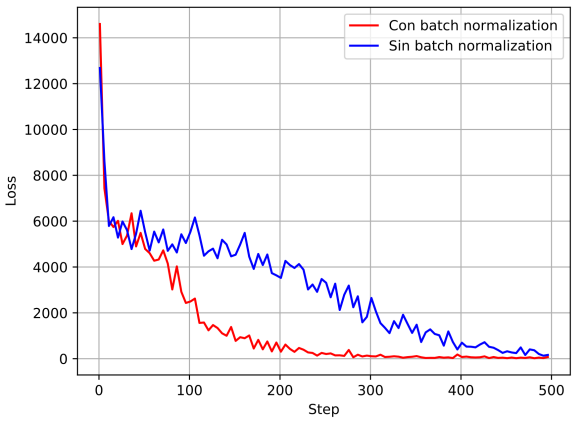
Figura 13.3: Distribución de los caracteres para cada instante de tiempo cuando la red fue entrenada con 100 horas de audios de LibriSpeech.

Otro factor interesante para analizar es la ventaja de la utilización de *batch normalization*. Para esto se utilizó la misma red presentada previamente entrenada con cien audios del *set* de entrenamiento, pero en un caso se la entrenó con *batch normalization* y en el otro no. Los resultados se encuentran en la figura 13.4. Las redes fueron entrenadas por 500 iteraciones hasta que éstas alcanzaron hacer *overfitting* sobre los datos de entrada.

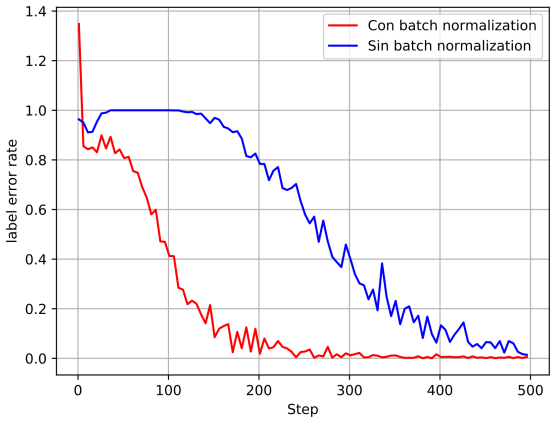
En dichas figuras se puede observar cómo la velocidad de convergencia de la red que utiliza *batch normalization* es muy superior a la otra. Luego de cincuenta iteraciones se puede observar cómo la red con *batch normalization* comienza a aprender a reproducir correctamente, logrando un error casi nulo luego de doscientas iteraciones. Por el contrario, la red sin este mecanismo comenzó a aprender los datos a partir de las doscientas iteraciones, habiendo pasado por un período donde el costo se reducía levemente pero sin mejorar el error de predicción.

Este fenómeno se hace mucho más evidente al aumentar la cantidad de audios. La cantidad de audios utilizada en esta prueba es tal que la red es capaz de aprenderlos de memoria, pero al aumentar el tamaño del *set* de entrenamiento, la red se encuentra obligada a encontrar una representación común entre los audios, lo cual es mucho más difícil de hacer cuando las muestras no están normalizadas. Esta prueba no pudo ser realizada de manera completa, ya que para observar que una red que no utiliza *batch normalization* converja se debe entrenar durante mucho más tiempo que a una que sí lo utiliza[26]. Sin embargo, sí pudo observarse que luego de varios días de entrenamiento con las 100 h de LibriSpeech no se produjeron reducciones considerables del costo que indicaran que la red estuviera realmente aprendiendo.

[26] Se debe recordar que cada entrenamiento de la red ZorzNet toma alrededor de cinco días hasta alcanzar un resultado medianamente interesante.



(a) Costo obtenido en función de las iteraciones.



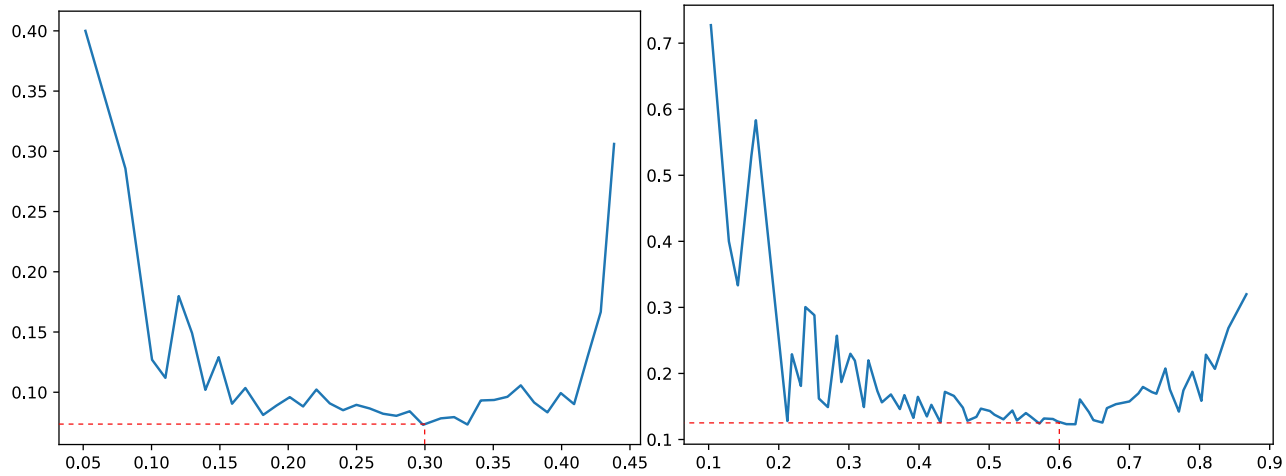
(b) LER obtenido en función de las iteraciones.

Figura 13.4: Comparación entre dos redes ZorzNet, una con *bath normalization* y la otra no, al ser entrenada por 500 *epochs* con cien audios obtenidos de LibriSpeech.

Un último factor importante a tener en cuenta es la capacidad de aprender dependencias de largo plazo. En la figura 13.5 se presenta el error en función de la relación entre la longitud de entrada y la de salida. En dicha figura se observa una zona donde el error no se modifica considerablemente, mientras que en los extremos el error se incrementa. La principal razón de esto se tiene en la figura 13.1, donde se muestra que la mayoría de los audios presentan un *ratio* de aproximadamente 0,3, habiendo pocos ejemplos en los extremos, demostrando que el conjunto de datos de entrenamiento no es el más adecuado para realizar este tipo de comparaciones. A pesar de esto, puede observarse en la figura 13.5a que aquellos audios que tienen un ratio cercano a 0,3, es decir que están en el centro de la campana de la figura 13.1, tienen un error cercano al 5 %. Sin embargo, al analizar los resultados de la imagen 13.5b, donde los audios fueron submuestreados por la mitad nuevamente y el centro de la campana ahora se encuentra en el *ratio* de 0,6, el error pasa a superar el 10 %. Una posible explicación para esto podría ser que al reducirse la relación de longitudes, la red se encuentra cada vez más obligada a alinear cada caracter con un único vector de *features*, dejando poco lugar para aquellas transiciones donde no es clara la diferencia[27].

Igualmente para poder tener resultados concluyentes sobre este tipo de pruebas, es necesario que la distribución de las longitudes en el *set* de entrenamiento sea más uniforme, de manera que el error en función de la longitud sea únicamente una función dependiente del modelo de red utilizado.

[27] Asumiendo que la pérdida de información por el descarte de muestras no es demasiado influyente.



(a) LER en función del *ratio* entre la longitud de los *features* y las transcripciones al reducir la longitud de los *features* a la mitad.

(b) LER en función del *ratio* entre la longitud de los *features* y las transcripciones al reducir la longitud de los *features* a dos veces por la mitad.

Figura 13.5: LER obtenido con ZorzNet sobre el *set* de testeo de LibriSpeech, en función del *ratio* entre la longitud de los *features* y de las transcripciones.

13.3. LASNet

Los hiperparámetros en este caso fueron:

```
network_data.num_classes = LASLabel.num_classes # a-z + <sos> + <eos>
network_data.num_features = 494

network_data.num_embeddings = 0
network_data.sos_id = LASLabel.SOS_INDEX
network_data.eos_id = LASLabel.EOS_INDEX
network_data.noise_stddev = 0.1
network_data.num_reduce_by_half = 0

network_data.beam_width = 0

network_data.num_dense_layers_1 = 2
network_data.num_units_1 = [400] * network_data.num_dense_layers_1
network_data.dense_activations_1 = [tf.nn.relu] * network_data.num_dense_layers_1
network_data.batch_normalization_1 = True
network_data.batch_normalization_trainable_1 = True
network_data.keep_prob_1 = [0.8] * network_data.num_dense_layers_1
network_data.kernel_init_1 = [tf.truncated_normal_initializer(mean=0, stddev=0.1)] * network_data.
    num_dense_layers_1
network_data.bias_init_1 = [tf.zeros_initializer()] * network_data.num_dense_layers_1

network_data.listener_num_layers = 1
network_data.listener_num_units = [512] * network_data.listener_num_layers
network_data.listener_activation_list = [tf.nn.tanh] * network_data.listener_num_layers
network_data.listener_keep_prob_list = [0.9] * network_data.listener_num_layers

network_data.num_dense_layers_2 = 0
network_data.num_units_2 = [400]
network_data.dense_activations_2 = [tf.nn.tanh] * network_data.num_dense_layers_2
network_data.batch_normalization_2 = True
network_data.batch_normalization_trainable_2 = True
network_data.keep_prob_2 = [0.8] * network_data.num_dense_layers_2
network_data.kernel_init_2 = [tf.truncated_normal_initializer(mean=0, stddev=0.1)] * network_data.
    num_dense_layers_2
network_data.bias_init_2 = [tf.zeros_initializer()] * network_data.num_dense_layers_2

network_data.attention_type = 'luong'        # 'luong', 'bahdanau'
network_data.attention_num_layers = 1
network_data.attention_size = None
network_data.attention_units = 512
network_data.attention_activation = tf.nn.tanh
network_data.attention_keep_prob = 0.9

network_data.kernel_regularizer = 0.0
```

```
network_data.sampling_probability = 0.1

network_data.learning_rate = 0.001
network_data.use_learning_rate_decay = True
network_data.learning_rate_decay_steps = 8000
network_data.learning_rate_decay = 0.98

network_data.clip_gradient = 0
network_data.optimizer = 'adam'       # 'rms', 'adam', 'momentum', 'sgd'
network_data.momentum = None
```

Algunos comentarios sobre los hiperparámetros elegidos:

- Nuevamente se agregó ruido blanco a la entrada sin comprobar su verdadera efectividad.
- Se utilizó una única capa para el *listener*, es decir que la longitud temporal de la entrada fue reducida a la mitad. Dado que el *listener* está compuesto por redes recurrentes, nuevamente se utilizó la función de activación `tanh`, permitiendo que se tomen tanto valores positivos como negativos.
- Existen dos posibilidades para el mecanismo de atención: Bahdanau o *soft attention*, y Luong o *hard attention*. En este trabajo no pudieron hacerse muchas pruebas con el mecanismo de *soft attention* ya que éste presenta una complejidad computacional mucho mayor, casi imposibilitando su uso en la computadora personal, y obligando a utilizar un *batch size* pequeño en Google Colab. Es por esta razón que se optó por trabajar únicamente con el mecanismo de Luong.
- El valor de `sampling_probability` fue de 0,1, ya que al tomar factores superiores se tuvieron problemas de *exploding gradient*. Luego de un largo período de entrenamiento es posible incrementar este factor, ya que las predicciones de instantes previos son más confiables, pero se optó por mantener fijo este valor porque se observó una correcta generalización.
- Nuevamente se utilizó *learning rate decay* sin una verdadera comprobación de su utilidad.

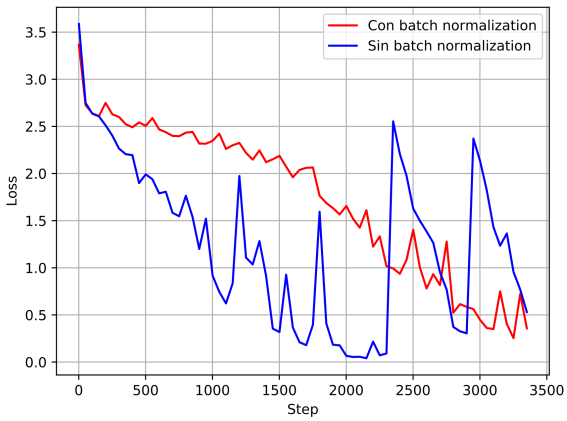
Algunas conclusiones preliminares entonces son:

- No se observaron diferencias considerables al agregar un perceptrón multicapa entre el *listener* y la capa de atención. Por lo tanto, debido a que de esta manera se reducen los tiempos de cómputo, se decidió puentear esta capa (al indicar que el número de capas es cero, automáticamente se conecta la entrada con la salida). Esto puede estar relacionado con la existencia de un problema de gradiente evanescente, lo cual también fue observado en la red anterior al incrementar la profundidad.
- La utilización de `clip_gradient` no dio buenos resultados. Esto podría deberse a que los valores utilizados fueron siempre demasiado bajos, haciendo que la actualización de pesos se volviera muy lenta. Sin embargo, debido a los tiempos de entrenamiento requeridos, no se continuó entrenando para confirmar que ésta fuera la razón.
- A pesar de no aplicar ningún término de regularización sobre los pesos, se observó una buena generalización. Esto puede deberse a que el mecanismo de atención elige internamente la información relevante, evitando que se aprenda de la redundancia.

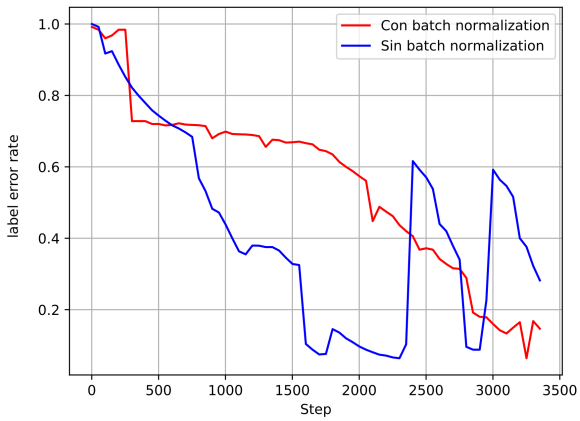
Uno de los detalles importantes sobre esta red es que requiere mucho tiempo de entrenamiento. No solo se requirió más tiempo de ejecución por cada *batch*, sino que además se requirieron muchas más iteraciones hasta comenzar a ver una disminución del costo. A modo de comparación, se puede observar en la figura 13.6 que a pesar de entrenar con un único audio, se requirieron aproximadamente 1500 iteraciones antes de comenzar a ver una reducción considerable en el error, mientras que en la red CTC se requirió menos de un tercio de éstas.

En la figura 13.6 también se tiene una comparación entre los resultados obtenidos con y sin *batch normalization*. Dado que la prueba se realizó con un único audio, la utilización de *batch normalization* no tiene demasiado sentido, pero debido a los tiempos que requieren estas pruebas, no se pudo realizar una comparación con más audios. Sin embargo, se puede observar que al utilizar *batch normalization* la convergencia resulta mucho más suave. Incluso en el caso sin *batch normalization* se observa cómo luego de acercarse a una condición de *overfitting* con

error cercano a cero se produjeron grandes saltos en el costo, indicando una mayor inestabilidad. Una posible explicación para esto puede ser la siguiente: Se trata de una red altamente profunda que combina redes recurrentes, perceptrones multicapa, etc. Generalmente las distintas arquitecturas convergen a velocidades diferentes, por lo que el *learning rate* utilizado tiene efectos diferentes sobre cada una de ellas, pudiendo ser grande para algunas y pequeño para otras, y como se mencionó previamente este efecto puede ser reducido utilizando *batch normalization*. Por otra parte, se trata de una red altamente profunda con una gran predisposición a tener problemas de *vanishing gradient*, provocando que no todas las capas puedan entrenarse de igual manera. Por lo tanto, en una red de estas características sin *batch normalization*, al acercarse a una zona de bajo error (donde se encuentra un mínimo y el gradiente es pequeño), puede ocurrir que determinadas capas dejen de poder actualizarse, y a su vez aquellas capas que sí se actualizan lo estén haciendo con un *learning rate* alto, provocando que la red se aleje rápidamente del mínimo alcanzado previamente.



(a) Evolución del costo de la red LAS al entrenar con un único audio de LibriSpeech.



(b) Evolución del LER de la red LAS al entrenar con un único audio de LibriSpeech.

Figura 13.6: Comparación entre dos redes LASNet, una con *bath normalization* y la otra no, al ser entrenadas con un único audio de LibriSpeech.

Al igual que en el caso anterior se graficó el error en función del *ratio* entre la longitud de entrada y la salida en la figura 13.7. Es importante tener en cuenta que el *listener* de la red LAS reduce a la mitad la longitud de la entrada. Si se analiza el funcionamiento de la red LAS completa, se deben comparar las figuras 13.7 y 13.5a, donde se hace evidente que para una misma relación de longitudes, el error en la red CTC resulta superior. Por otra parte, si se analiza únicamente el funcionamiento de la función de costo CTC y la capa de atención de la red LAS, se deben comparar las figuras 13.7 y 13.5b, donde en este caso la red LAS presenta un error menor.

Para realizar una verdadera comparación del error en función de las longitudes, es necesario graficar el error de la red LAS para distintos valores de *ratios*, pero debido a los largos tiempos de entrenamiento que ésto requiere, esta experiencia no fue realizada.

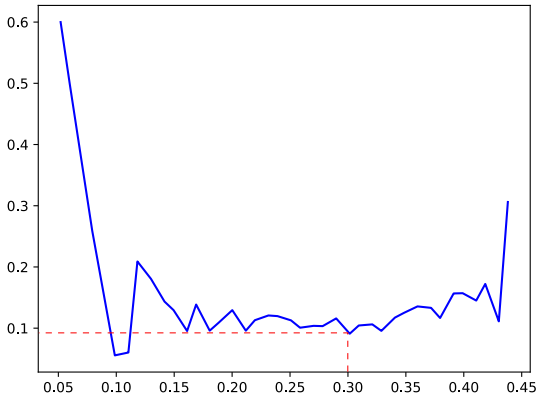


Figura 13.7: LER obtenido con LASNet sobre el conjunto de *testeo* de LibriSpeech, en función del *ratio* entre la longitud de los *features* y de las transcripciones. La longitud de los audios de entrada fue reducida a la mitad, de la misma manera que en la figura 13.5a.

Un ejemplo de alineación para este modelo se encuentra representado en la figura 13.8. A diferencia de los resultados obtenidos en la figura 13.2, la salida decodificada en esta red no utiliza un símbolo *blank* ni unifica caracteres iguales, y es por esta razón que en cada instante de tiempo se tiene una predicción válida. Por otra parte, a diferencia de la red CTC, la longitud temporal de la salida no coincide con la longitud de la entrada. Parte del aprendizaje de la red es aprender a emitir el símbolo `<eos>` para indicar que la predicción fue finalizada. Es por esta razón que en este caso no es posible estimar una posible alineación entre la entrada y la salida donde se observe en qué instante ocurrió cada caracter[28].

Algo interesante para remarcar sobre los resultados de la figura 13.8 es el error cometido al emitir el caracter *w* en lugar de la *l*. En dicho instante de tiempo se observa que los dos caracteres con mayor probabilidad son justamente la *w* y la *l*, pero debido a que se está utilizando la decodificación *greedy* sin modelo de lenguaje, se termina eligiendo un caracter que debiera ser poco probable, pero que la red entiende que es el ganador.

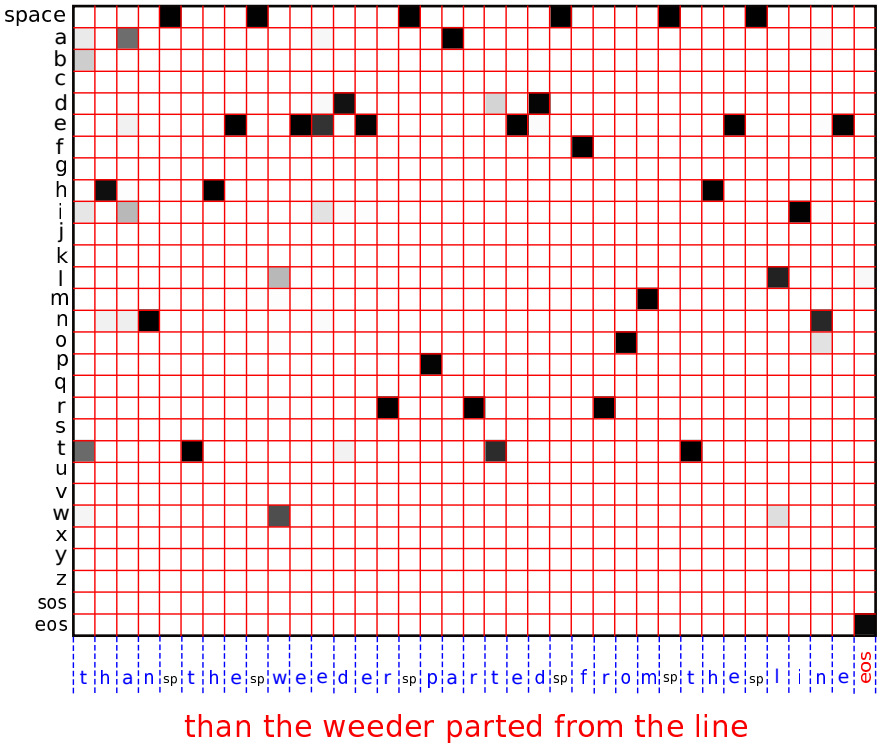


Figura 13.8: Distribución de los caracteres para cada instante de tiempo para la red LASNet.

[28]Las funciones de TensorFlow que realizan la decodificación no permiten el acceso a la predición completa, solo a la decodificación. Es por esta razón que este dato no puede ser conocido.

Otra manera de visualizar la evolución de las predicciones de la red es analizando los pesos α_{ij} definidos en la ecuación 7.2. En la figura 13.9 se tienen los resultados para esta prueba. Como se mencionó previamente, los coeficientes α_{ij} indican el peso de la entrada j-ésima para estimar el contexto de la predicción i-ésima. Por lo tanto, sería esperable que en la figura 13.9 se observara que los pesos asignados al instante actual fueran mucho mayores que los asignados al resto de los instantes de tiempo. Sin embargo, se puede observar que en varios casos los pesos asignados a los últimos instantes de tiempo resultaron considerables. Si bien esto podría estar indicando una correlación entre distintos instantes debida a cambios en la entonación o énfasis, probablemente en este caso se deba principalmente a que la red no ha sido entrenada lo suficiente como para aprender completamente la estructura.

En esta figura es también posible observar cómo se alinea la predicción con la entrada, salvo que no se tiene información sobre la probabilidad de cada clase en cada instante de tiempo, sino sólo a qué parte de la entrada se le presta atención para estimar cada una de ellas.

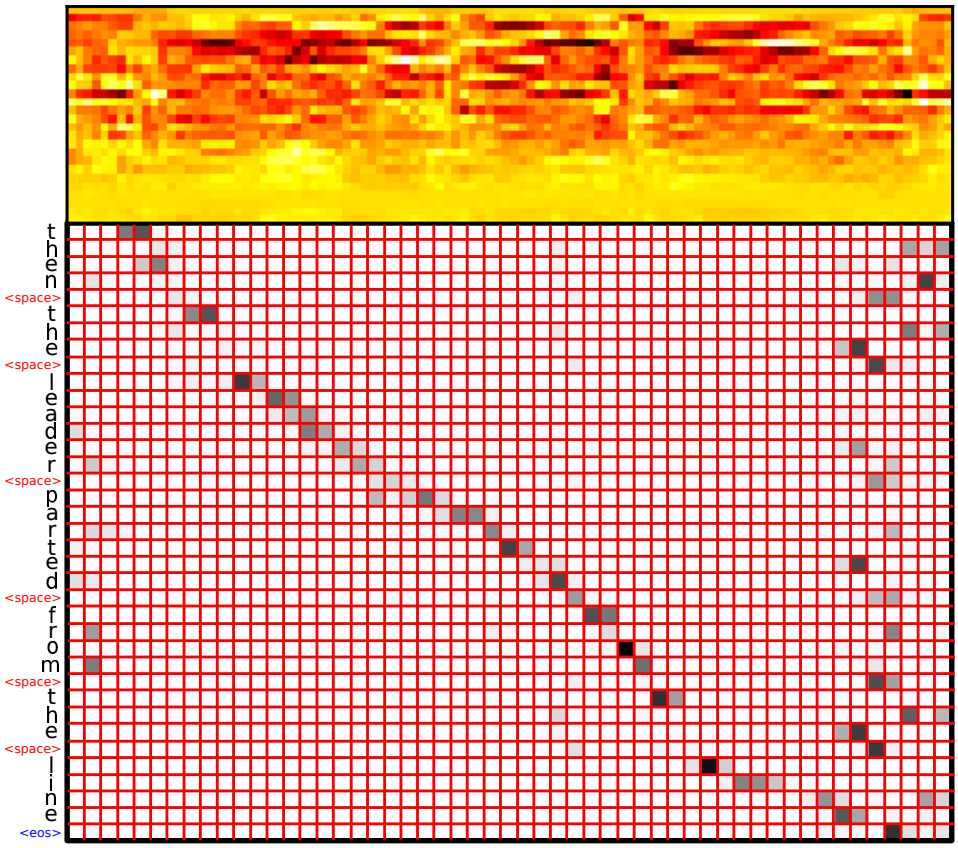


Figura 13.9: Historial de alineación obtenido con la red LASNet. En el eje horizontal se tiene el tiempo, en el vertical se tiene la transcripción, y la intensidad representa el nivel de atención que se le presta a cada entrada para realizar la predicción de cada uno de los caracteres de salida.

13.4. Comparación de resultados

Los principales problemas encontrados en este trabajo fueron la velocidad y el tiempo de entrenamiento requeridos para lograr buenos resultados. En la tabla 13.1 se tiene un resumen de los tiempos de ejecución de las distintas redes bajo distintas circunstancias.

Los resultados correspondientes a la red CTC+RNN_BN se corresponden con la red ZorzNet que utiliza *batch normalization* en la RNN. Según se encontró en distintos foros, la biblioteca de TensorFlow contiene un *bug* conocido que provoca que la actualización, cuando se utiliza *batch normalization* en una red recurrente, tome mucho tiempo. Dado que la aplicación de *batch normalization* en las capas densas demostró ser muy eficaz, es esperable que su utilización en las capas recurrentes también lo sea. Sin embargo, debido a la gran cantidad de tiempo que requiere cada iteración, ésto no pudo ser comprobado.

En el cuadro también se puede observar que el tamaño del *batch* utilizado para la red LASNet con el mecanismo de *soft attention* resulta menor que en las demás pruebas. Esto se debe a que el tamaño de la red excedía la capacidad de cómputo del entorno, por lo que se redujo el *batch size* hasta que pudiera ser ejecutada.

Al analizar los tiempos de ejecución necesarios para observar al menos una vez a cada muestra de entrenamiento, se hace evidente que la elección de un nuevo hiperparámetro requiere grandes períodos de espera hasta ver nuevos resultados, por lo que no es posible asegurar que los hiperparámetros utilizados en este trabajo sean los que permitan el mejor resultado posible.

Tipo de red	GPU	*Batch size*	*Train* (1 *batch*)	*Validation* (1 *batch*)	Duración de 1 *epoch*
CTC	Tesla K80	50	20,7s	4,4s	12420s = 3,45h
CTC+RNN_BN	Tesla K80	30	90s	6,5s	90000s = 25h
LAS (Luong)	Tesla K80	50	25s	3,8s	15000s ≈ 4,16h
LAS (Bahdanau)	Tesla K80	10	10,6s	3,4s	318000s ≈ 88h

Cuadro 13.1: Comparación de tiempos de ejecución utilizando los hiperparámetros que dieron los mejores resultados, al ejecutar las distintas redes en Google Colaboratory con la placa Tesla K80 sobre las 100 horas de audios de LibriSpeech.

Finalmente, en la tabla 13.2 se tienen los resultados obtenidos para las distintas redes. Una primera observación a estos resultados es la influencia que tiene la cantidad de datos utilizados durante el entrenamiento. Se puede ver que en el trabajo previo [40] entrenado con la base de datos TIMIT, que contiene cerca de 5000 audios, se obtuvo un error muy superior al obtenido con la nueva[29] red CTC entrenada con 100 horas de muestras (aproximadamente 30000 audios). Esto también queda en evidencia al comparar con los resultados obtenidos utilizando la misma red pero entrenando con más horas de audios. Es importante aclarar que debido a los largos períodos de entrenamiento que requieren estas redes, al incrementar la cantidad de muestras no se realizaron entrenamientos tan prolongados como se debiera haber hecho, incluso no se observó que se hubiera llegado a un estancamiento en la reducción del costo en ninguna de las arquitecturas.

Durante el entrenamiento se observó que la red CTC logra una gran reducción del error durante las primeras iteraciones, pero luego comienza a reducirse muy lentamente. Por ejemplo, cuando se realizó el entrenamiento con 100 horas de LibriSpeech, se alcanzó un error cercano al 50 % en la primer hora de entrenamiento, pero luego se requirieron aproximadamente 10 días hasta alcanzar el error final del 25,1 %. Con la red LAS (Luong) se observó el comportamiento opuesto, ya que durante los primeros días el error permaneció por arriba del 90 %, y luego de sobrepasar las 100 iteraciones se redujo rápidamente. Es importante aclarar que la red LAS requiere más iteraciones (y por ende más tiempo) de entrenamiento para alcanzar errores similares.

Tipo de red	Dataset	LER (train)	LER (val)	WER	LM
ZorzNet	TIMIT [40]	10.2 %	31.8 %	-	-
	100hs LS	3.5 %	8.6 %	25.1 %	-
	200hs LS	4.5 %	8.4 %	25 %	-
	500hs LS	2.5 %	**7.2 %**	**21.7 %**	-
LASNet	100hs LS	-	11.1 %	27.0 %	-
	200hs LS	4.5 %	8.6 %	21.0 %	-
	500hs LS	4.5 %	**7.2 %**	**17.9 %**	-
LAS [5]	Google voice 2000hs	-	-	16.2 %	-
		-	-	12.6 %	LM+Rescoring
		-	-	**10.3 %**	LM+sampling+rescoring
CTC [12]	TIMIT 5hs	-	31.4 %	-	-
CTC [11]	81hs WSJ	-	9.2 %	30.1 %	-
		-	-	10.4 %	Bigrama
		-	-	**8.7 %**	Trigrama
DeepSpeech [15]	300hs SWB	-	-	25.9 %	5-grama
	2300hs SWB+FSH	-	-	16 %	5-grama
	5000hs	-	-	**11.8 %**	5-grama

Cuadro 13.2: Comparación de los resultados obtenidos al ejecutar las distintas redes en Google Colaboratory con la placa Tesla K80. Los resultados reportados para la red **ZorzNet** y la red **LASNet** son los correspondientes a los modelos implementados en este trabajo, mientras que los demás resultados son los de otros trabajos de referencia similares.

[29] Si bien los hiperparámetros no son los mismos y se incluyeron nuevas técnicas de regularización y optimización, la estructura sigue siendo comparable. Por otra parte, el error obtenido en [40] es muy similar al de [12], por lo que es probable que no se pudieran lograr resultados mejores.

En cuanto a la red CTC, se puede ver que en [11] se utilizaron 81 horas de audios del Wall Street Journal, y los resultados son comparables con los de la red ZorzNet entrenada con 100 horas de LibriSpeech. Una diferencia importante entre estos modelos es que la red en [11] utiliza cinco capas recurrentes de 500 unidades cada una, mientras que en este trabajo se utilizaron dos capas recurrentes de 512 unidades, ya que el tamaño de la red era demasiado grande para ser ejecutado con los recursos disponibles.

Si bien los errores obtenidos fueron comparables, se observa que simplemente agregando un modelo de lenguaje el error se reduce considerablemente, lo cual era esperable debido a la naturaleza de la métrica de error. Cuando se utiliza como métrica el WER, que exista un caracter erróneo invalida la palabra, y debido a que en general la proporción de cantidad de palabras por oración es menor a la de caracteres por oración, el WER suele ser más alto. Al utilizar un modelo de lenguaje que busca palabras similares dentro de un diccionario, los errores debidos a un caracter erróneo se reducen, y si además se utiliza un modelo de n-gramas también se evita que por culpa de un cambio en un caracter se elija una palabra errónea que debiera ser de baja probabilidad en el contexto de esa oración. El ejemplo más evidente se da con el trabajo [11], donde se utiliza un conjunto de entrenamiento pequeño[30], y con la simple introducción de un modelo de lenguaje se logran alcanzar errores que son aproximadamente la mitad de los alcanzados por los modelos implementados en este trabajo.

Otro resultado comparable es el presentado por DeepSpeech al entrenar con 300 horas de audios. Se puede ver que a pesar de utilizar 5-gramas en la decodificación, el error reportado es mayor que los alcanzados por ZorzNet, incluso utilizando una menor cantidad de datos.

En la figura 13.10 se muestran algunos ejemplos de predicciones realizadas con ZorzNet. En la última de las predicciones se observa que se eliminó un caracter pasando de *excess* a *ecess*, dando como resultado una palabra inexistente, lo cual es un ejemplo de un caso que puede ser resuelto mediante la utilización de un diccionario. Por otra parte, en la primera de las predicciones se contraen las palabras *on the* en *only*, dando como resultado palabras válidas pero donde la oración carece de sentido. Este es el tipo de problemas que puede resolverse utilizando un modelo de lenguaje que introduzca la probabilidad de que ocurra determinada palabra dadas las anteriores.

En relación con esto, se creó un diccionario de palabras con todas aquellas presentes en el *set* de entrenamiento para refinar los resultados. A cada oración predicha se la separó por los espacios, y a cada palabra resultante se la comparó con todas las palabras del diccionario hasta encontrar aquella que minimizara la distancia. Además de ser un método extremadamente lento no se logró mejorar el error, ya que al tratarse de una búsqueda lineal, cuando se tienen dos palabras a una misma distancia, en muchos casos no se elige la correcta. Para mejorar esto se decidió realizar una prueba aplicando un modelo de lenguaje real. El modelo de lenguaje utilizado se encuentra disponible en `https://github.com/githubharald/CTCWordBeamSearch`, donde se define un algoritmo *beam search* implementado en TensorFlow, que permite incorporar un *corpus*, un listado de palabras válidas y definir una estrategia de *scoring* (n-gramas, *forecast*, etc.) para mejorar la decodificación de los *logits* predichos por la red. Sin embargo, no se lograron mejorías significativas, y dado que gran parte del código de esta biblioteca se encuentra compilado, no es posible saber si se trata de un problema de implementación o si fue incorporado de manera incorrecta en este trabajo. Debido a esto, se continuó trabajando sin utilizar ningún tipo de modelo de lenguaje.

Es importante destacar de la figura 13.10 que al analizar la pronunciación de las palabras donde ocurrieron errores, se observa que la pronunciación entre la predicción errónea y la real no resulta tan diferente, lo cual muestra que la red está aprendiendo una estructura de pronunciación que podría ser mejorada agregando ruidos y nuevos acentos.

[30] A pesar de ser aumentado de manera artificial.

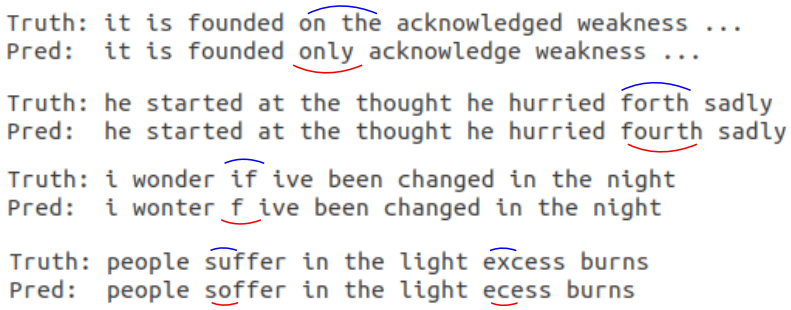


Figura 13.10: Ejemplos de predicciones realizadas sobre el conjunto de testeo de LibriSpeech con la red ZorzNet.

Observaciones similares pueden hacerse a la red LAS. La red utilizada en [5] está compuesta por un *listener* de tres niveles y un *speller* de dos capas recurrentes con 512 unidades cada una, lo cual es una estructura similar a la de este trabajo. Sin embargo, la red fue entrenada con 2000 horas de audios (aumentados artificialmente mediante distintos tipos de ruidos) durante más de dos semanas, mientras que en este trabajo se utilizó menos de un cuarto de muestras de entrenamiento (dejando de lado el incremento artificial por el ruido aditivo) y se entrenó durante aproximadamente diez días. Esta diferencia se condice con la reducción en el error observada en la tabla 13.2 al incrementar la cantidad de ejemplos de entrenamiento. Por otra parte, nuevamente se observa una mejoría considerable en los resultados al introducir un modelo de lenguaje.

Como se mencionó previamente, esta red resultó mucho más difícil de entrenar que la red CTC. No solo los tiempos de entrenamiento fueron mayores, sino que además se requirieron varios días hasta observar una baja considerable del costo. Algunas de las posibles explicaciones para esto pueden ser:

- Se tienen redes recurrentes en distintos niveles de la red, y dado que no fue posible aplicar *batch normalization* a las celdas LSTM, el *learning rate* podría no ser el óptimo para cada una de ellas. Esto podría estar provocando que aprendan a velocidades distintas, y que el costo se reduzca muy lentamente.
- Se trata de una red altamente profunda, donde pueden aparecer problemas de *vanishing gradient*. Esto puede estar generando que la actualización de las primeras capas sea muy lenta.

A pesar de estas desventajas frente a la red CTC, se puede observar que el error de validación de la red LAS resulta menor. También se puede ver en la tabla 13.2 que el aumento de la cantidad de muestras de entrenamiento tiene más impacto en la generalización de la red LAS que en el de la red CTC, lo cual puede estar indicando que la red CTC se encuentra cerca del mínimo alcanzable con los hiperparámetros elegidos.

Se puede observar también que el LER de ambas redes resulta igual (al entrenar con 500 h de LibriSpeech), pero que el WER de la red LAS es bastante menor. Esto está indicando que los errores a nivel caracter de la red CTC se encuentran más distribuidos a lo largo de toda la oración, generando una mayor proporción de palabras incorrectas, mientras que la red LAS tiene una mayor tendencia a cometer errores dentro de una misma palabra.

Un último detalle importante es que en este trabajo se utilizó *greedy decoder* en ambas redes. Sin embargo, tanto en [5] como en [11] utilizaron *beam search* como decodificador, donde también mostraron que un incremento en la cantidad de *paths* significaba una mejora en el error. Como se mencionó en la sección 13.2, el algoritmo *beam search* no produjo una mejoría en los resultados, por lo que esta conclusión no pudo ser comprobada.

Algo interesante de analizar es la generalización de los modelos utilizados sobre muestras de audios correspondientes a bases de datos completamente distintas. En la tabla 13.3 se compararon los errores obtenidos con ambos modelos, al ser entrenados con LibriSpeech pero validados con TIMIT. Como se puede observar, los resultados difieren ampliamente de los de la tabla 13.2, lo cual está indicando que las redes no solo aprendieron una estructura de lenguaje que les permitió generalizar sobre muestras jamás observadas, sino que también aprendieron características particulares del tipo de dataset:

Tipo de red	LER	WER
ZorzNet	15.2 %	43.4 %
LASNet	17.1 %	43.9 %

Cuadro 13.3: Comparación de los resultados al validar sobre TIMIT habiendo entrenado con 500 horas de audios de LibriSpeech.

nivel de ruido, velocidad y calidad de pronunciación, diversidad de acentos y personas, etc. Esto nuevamente indica que es necesario que el *set* de entrenamiento sea representativo de la tarea a realizar, y que una mala elección de dicho conjunto de muestras puede llevar a resultados pobres a pesar de tener la arquitectura correcta.

14. Conclusiones

En este trabajo se analizaron dos de las técnicas más utilizadas en la actualidad para la resolución de problemas del tipo *sequence-to-sequence*, en particular para la conversión de habla a texto. A pesar de que en la comunidad suele utilizarse el WER como métrica de error para este tipo de problemas, dado que en este trabajo no se utilizó un modelo de lenguaje, dicha métrica resulta de algún modo injusta ante los resultados obtenidos. Sin embargo, debido a que en general los trabajos en esta área no reportan el LER, fue necesario medir el WER para comparar resultados.

Al comparar con trabajos de estructuras similares, y que hoy en día se consideran el estado del arte, se observó que los resultados obtenidos son cercanos a los niveles de error alcanzados por éstos, y que con algunas modificaciones podrían llegar a ser igualados. De acuerdo con lo discutido previamente, los factores más importantes para lograr esto son:

- **La cantidad de datos de entrenamiento**. Tanto en este trabajo como en los de referencia, se observó una mejora en la generalización al aumentar la cantidad de muestras de entrenamiento. Debido a esto, se puede inferir que incrementando las muestras de entrenamiento sería posible alcanzar niveles de error mucho más cercanos a lo que hoy se considera el estado del arte con estas arquitecturas.
  La mayoría de los trabajos analizados no solo utilizó una mayor cantidad de datos, sino que también aumentó su conjunto de entrenamiento de manera artificial mediante la introducción de ruido: ruido blanco, ruido ambiente, etc. En dichos trabajos demostraron que su utilización permite que los sistemas sean mas resistentes ante situaciones reales. Por esta razón, se incorporó ruido blanco en este trabajo, pero debido a los tiempos de entrenamiento no se pudo comprobar su verdadera efectividad.

- **El tiempo de entrenamiento**. Google Colaboratory resultó ser una herramienta muy útil y práctica para el entrenamiento de redes neuronales, aunque se encontraron algunos problemas:

  - Si bien las sesiones de entrenamiento pueden durar hasta doce horas antes de ser reiniciadas, se observó que luego de varios días de entrenamiento estas sesiones comenzaron a ser más cortas, incluso llegando al punto de no poder reiniciar la sesión durante otras doce horas. Esto provocó que en lugar de entrenar las redes 24 horas al día, solo se lo hiciera aproximadamente 6 horas al día, incrementando los tiempos brutos de entrenamiento.
  - Si bien es posible tener acceso a la carpeta de Google Drive desde Google Colaboratory, existen problemas cuando se intentan importar demasiados archivos. En este trabajo se dividió la base de entrenamiento en archivos *tfrecords* de 1GB, dando como resultado cerca de 100 archivos. Al intentar importar todos estos archivos desde Google Colaboratory, luego de unos minutos la sesión era finalizada, imposibilitando su uso durante varias horas. La solución fue entrenar con menos archivos, elegidos aleatoriamente cada vez, de manera de observar todo el *set* completo.

- **La utilización de un modelo de lenguaje**. Como se muestra en el cuadro 13.2, todos los trabajos de referencia reducen ampliamente su error al incorporar un modelo de lenguaje. Por lo tanto, es posible asumir que dados los resultados obtenidos mediante los modelos aquí implementados, sería posible alcanzar niveles de error similares si se introdujera un modelo de lenguaje. Debido a la complejidad de armar un modelo propio, y a que los modelos de lenguaje obtenidos de bibliotecas libres no produjeron resultados mejores, se optó por dejar de lado este camino.

- **El preprocesamiento de los datos**. Muchos de los trabajos utilizados como referencia, incorporaron nuevas técnicas de preprocesamiento de los *features* de entrada para mejorar sus resultados. En este trabajo no se utilizaron dichas técnicas, pero sí se logró mejorar los resultados al modificar la representación de entrada de

los coeficientes MFCC tradicionales por la representación de DeepSpeech, lo cual se condice con los resultados de dichos trabajos.

Una posible mejora a futuro sería la de profundizar sobre estos temas, ya que han demostrado ser influyentes en los trabajos de referencia.

Otras observaciones a tener en cuenta a la hora de trabajar con este tipo de modelos son:

- **Los datos de entrenamiento deben ser representativos del problema que se desea resolver**. Por ejemplo en este trabajo no se pudieron sacar conclusiones sobre el aprendizaje de dependencias de largo plazo, ya que la distribución de longitudes influyó fuertemente en las relaciones aprendidas por la red. Lo mismo ocurrió al validar con TIMIT las redes entrenadas con LibriSpeech, ya que los acentos, longitudes de audios, métodos de grabación, etc. fueron diferentes.
- **La utilización de técnicas de regularización**, principalmente *batch normalization*, demostró ser de gran ayuda no solo para reducir el error de generalización, sino también para incrementar la velocidad de convergencia.
- **Los problemas de gradiente evanescente o explosivo deben ser tenidos siempre en cuenta**. Si bien el aumento de la profundidad de una red suele llevar a mejores resultados, ésto debe estar acompañado de la utilización de técnicas de regularización y normalización para evitar este tipo de problemas.
- **La representación de los datos**. Esto no solo permite facilitar el proceso de entrenamiento, sino que además permite reducir la complejidad de la red. Por ejemplo en la red [5] se utilizan los coeficientes MFCC de manera directa, pero lo compensan aumentando el tamaño del *listener*. Esto provoca que se deban aprender más parámetros y que los tiempos de ejecución sean más elevados. Si bien en determinados casos ésto es útil, se debe analizar esta relación de compromiso.
- **La elección del entorno de trabajo**. En este caso se optó por utilizar TensorFlow ya que se trata de una biblioteca creada por Google, y que hoy en día ya se considera una de las más utilizadas. Sin embargo, todavía se encuentra en pleno desarrollo y en algunos casos con escasa documentación.

Otra mejora importante que puede ser introducida, principalmente a la red CTC, es el reemplazo de las redes recurrentes por mecanismos de atención del tipo *Multi-Head Attention*. En el trabajo [37] se presentan los mecanismos de atención de este tipo, que en lugar de utilizar el contexto de una red recurrente como entrada al mecanismo de atención, utiliza un perceptrón multicapa, lo cual permite un gran incremento en la velocidad de entrenamiento y la eliminación de la red recurrente. Esta estrategia fue utilizada en combinación con el algoritmo CTC en el trabajo [31] donde se lograron resultados altamente competitivos. En el repositorio de este trabajo [41] se encuentra una implementación de este modelo, aunque debido a problemas de implementación que no pudieron ser resueltos, los resultados obtenidos estuvieron muy alejados de los aceptables.

En este trabajo de tesis se pudo realizar una implementación satisfactoria de dos arquitecturas diferentes en TensorFlow, alcanzando resultados cercanos al estado del arte. Se observó que ambas arquitecturas analizadas presentan resultados similares en cuanto a la métrica de error, siendo la red LAS la que logró el mejor desempeño. Sin embargo, la red CTC presenta ventajas computacionales: menor tiempo de ejecución, menos cantidad de parámetros, más rápida de entrenar, etc. No pudieron obtenerse conclusiones en cuanto a la capacidad de aprender dependencias de largo plazo debido a que los datos de entrenamiento no son los adecuados para este tipo de análisis.

Se analizó la influencia de la representación de los datos, así como también la necesidad de que el conjunto de entrenamiento sea numeroso y representativo del problema en cuestión. En relación a esto, se exploraron distintas estrategias de regularización, que demostraron no solo mejorar los resultados, sino también reducir los tiempos de entrenamiento.

Es importante destacar que este trabajo de tesis fue realizado mediante el uso de herramientas libres, gratuitas y colaborativas, y a pesar de las limitaciones que esto trajo, se alcanzaron resultados similares a los de otros proyectos relevantes en el área.

Parte V

Apéndices

A. Trabajos previos

Previo a este trabajo, se realizaron otros proyectos relacionados con este temática. El primero fue el trabajo práctico final para la materia de Redes Neuronales (86.54) [39], el cual consistió en el armado de un sistema capaz de reconocer los dígitos del 0 al 9 a partir de señales de habla. La red neuronal utilizaba como entrada las matrices de coeficientes MFCC, las cuales eran ingresadas a dos capas convolucionales junto con una capa oculta, para finalmente llegar a una capa de 10 neuronas encargada de clasificar el dígito correcto. Dado que se trató de un primer acercamiento a la utilización de redes neuronales, este trabajo presentaba las siguientes limitaciones:

- Se utilizó la biblioteca ***TFLearn*** [2] en lugar de ***TensorFlow*** [1], ya que la primera presentaba ya implementados internamente muchos métodos de *TensorFlow,* facilitando su uso. La desventaja de esto es que no se tuvo tanta libertad para definir los modelos.
- La cantidad de muestras utilizadas fue pequeña. Si bien se utilizó la base de datos TIDIGITS [23], sólo se utilizó una porción de ésta, lo cual pudo haber limitado los resultados obtenidos.
- No se utilizaron métodos de regularización, lo cual pudo haber mejorado los resultados.
- Los audios fueron recortados a una misma longitud y aquellos más cortos que la longitud deseada fueron completados con ceros. Esto evitó tener que trabajar con secuencias de tamaño variable a costa de la pérdida de información[31].

A pesar de estas limitaciones se obtuvo una precisión de validación del 96,4 %, lo cual es suficientemente bueno para determinadas tareas.

El siguiente proyecto se realizó en el marco del trabajo práctico final de la materia Procesamiento del Habla (86.53) [40]. En este caso se planteó un conversor de habla a texto basado en redes neuronales, similar al implementado en este trabajo. Dada la experiencia previa y al estudio de nuevos conceptos, este proyecto incorporó lo siguiente:

- En este caso se utilizó *TensorFlow*, de manera que fue necesario implementar de manera manual muchas de las operaciones necesarias. A pesar del incremento en la complejidad, permitió incorporar nuevos conceptos y dio más libertad para optimizar los distintos modelos.
- Se incorporaron técnicas de regularización (*dropout*, regularización de pesos, *batch normalization*, etc.) que demostraron ser altamente efectivas.
- Se utilizó la base de datos TIMIT [10], la cual está compuesta por aproximadamente 6000 audios de personas enunciado distintas frases. A pesar de usarla de forma completa, tampoco consiste en una base de datos considerablemente grande.
- En este caso se utilizaron los audios sin recortar, de manera que fue necesario utilizar redes recurrentes para trabajar con las secuencias de longitud variable.

La complejidad de este segundo trabajo resultó mayor, ya que no solo se debió trabajar con más datos y de longitud variable, sino que se requirió mucho más esfuerzo en la calibración. Sin embargo, se obtuvo un LER del 31,5 % que es un valor cercano al que se obtuvieron en otros trabajos similares.

Este segundo reconocedor se basó en la utilización del algoritmo CTC, el cual es uno de los métodos centrales en este trabajo, y permitió la reutilización del código de algunos modelos y la oportunidad de comenzar a analizar ciertos conceptos importantes.

[31] Los recortes se realizaron de manera automática, lo cual en algunos casos podría haber producido la pérdida de partes de segmentos relevantes en la señal de habla.

B. Procesamientos complementarios

B.1. Conversor de archivos de audio

Los archivos de audio `.wav` proporcionados por TIMIT [10] y LibriSpeech [27] se encuentran en un formato diferente al que se suele utilizar. Estos audios presentan encabezados del formato SPHERE en lugar de NIST, que es el utilizado generalmente por los reproductores de audio y bibliotecas de Python. Para esto fue necesario entonces utilizar el conversor **sph2pipe_v2.5** de [36], con el cual se escribió el siguiente *script*:

```
#!/bin/bash
# This scripts allows to transform the Sphere NIST WAV format to normal WAV format

if [ "$#" -eq 1 ]
then
    search_dir=$1
    this_dir=FixWav
    echo $this_dir

    cd ..
    for file in "$search_dir"/*
    do
        echo "$file"
        ./"$this_dir"/sph2pipe -t : -f rif "$file" out.wav
        mv out.wav "$file"
    done
else
    echo "Not enough arguments"
    echo "Usage: ./FixWav.sh <source_dir>"
    exit 1
fi
```

Para utilizar este *script* se debe tener en la misma carpeta el software de [36] previamente descargado, y se debe ejecutar en la consola: `./FixWav.sh <source_dir>`, donde `<source_dir>` es la carpeta que contiene todos los audios.

B.2. Copiado de archivos de TIMIT

Los audios y transcripciones de TIMIT se encuentran separados en carpetas de acuerdo a sus características. Para generar los *datasets* compatibles con este trabajo, se optó por unificar todos estos archivos en una única carpeta, identificándolos con diferentes nombres para evitar problemas de repetición. Para automatizar el copiado de archivos en una única carpeta se utilizó el siguiente *script*:

```
#!/bin/bash
# This scripts takes all files in the TIMIT database and copies them to another folder
results_dir=results

if [ "$#" -eq 1 ]
then
    mkdir "$results_dir"
    mkdir "$results_dir/TRAIN"
    mkdir "$results_dir/TEST"

    search_dir=$1
    for data_dir in TEST TRAIN
    do
        for d in $(find "$search_dir/$data_dir/" -maxdepth 2 -type d)
        do
            for file in "$d"/*
            do
                extension="${file##*.}"
                filename="${file%.*}"
                folder="$(cut -d'/' -f11 <<<${filename})"

                cp "$file" "${filename}_${folder}.${extension}"
                mv "${filename}_${folder}.${extension}" "$results_dir/$data_dir/"
                echo "$0: '${file}'"
            done
        done
    done
else
```

```
    echo "Not enough arguments"
    echo "Usage: ./CopyTimit.sh <source_dir>"
    exit 1
fi
```

B.3. Copiado de archivos de LibriSpeech

Al igual que se realizó con TIMIT, se creó un script capaz de copiar todos los archivos estructurados según el *dataset* de LibriSpeech en una única carpeta, para poder ser utilizados en este proyecto:

```
#!/bin/bash
# This scripts takes all files in the Librispeech database and copies them to another folder

if [ "$#" -eq 1 ]
then
    search_dir=$1
    results_dir=$2
    for d in $(find "$search_dir/" -maxdepth 3 -type d)
    do
        for file in "$d"/*
        do
            extension="${file##*.}"
            if [ "$extension" == "flac" ];
            then
                filename="${file##*/}"
                echo "${filename}"
                cp $file "$results_dir/$filename"
            fi
        done
    done
else
    echo "Not enough arguments"
    echo "Usage: ./CopyLibrispeech.sh <source_dir> <results_dir>"
    exit 1
fi
```

B.4. Conversor flac2wav

La base de datos utilizada en este trabajo fue LibriSpeech [27]. Los audios en este caso se encuentran en el formato `.flac`, por lo que deben ser convertidos a `.wav` para poder utilizarlos de la misma manera que a los demás *datasets*. Para lograr esto se utilizó el siguiente *script*:

```
#!/bin/bash
# This scripts allows to convert the audio files from .flac to .wav

if [ "$#" -eq 1 ]
then
    search_dir=$1

    for file in "$search_dir"/*
    do
        filename=$(basename $file .flac)
        echo "$filename.wav"

        ffmpeg -i $file "$search_dir/$filename.wav"

    done
else
    echo "Not enough arguments"
    echo "Usage: ./flac2wav.sh <source_dir>"
    exit 1
fi
```

Referencias

[1] Abadi, Martín; Agarwal, Ashish; Barham, Paul; Brevdo, Eugene; Chen, Zhifeng; Citro, Craig; Corrado, Greg S.; Davis, Andy; Dean, Jeffrey; Devin, Matthieu; Ghemawat, Sanjay; Goodfellow, Ian; Harp, Andrew; Irving, Geoffrey; Isard, Michael; Jozefowicz, Rafal; Jia, Yangqing; Kaiser, Lukasz; Kudlur, Manjunath; Levenberg, Josh; Mané, Dan; Schuster, Mike; Monga, Rajat; Moore, Sherry; Murray, Derek; Olah, Chris; Shlens, Jonathon; Steiner, Benoit; Sutskever, Ilya; Talwar, Kunal; Tucker, Paul; Vanhoucke, Vincent; Vasudevan, Vijay; Viégas, Fernanda; Vinyals, Oriol; Warden, Pete; Wattenberg, Martin; Wicke, Martin; Yu, Yuan; Zheng, Xiaoqiang; "TensorFlow: Large-scale machine learning on heterogeneous systems", 2015. Software available from `https://www.tensorflow.org/`.

[2] Aymeric, Damien; and others; Biblioteca: TFLearn, `https://github.com/tflearn/tflearn`.

[3] Bahdanau, Dzmitry; Cho, Kyunghyun; Bengio, Yoshua: "Neural Machine Translation by Jointly Learning to Align and Translate". In International Conference on Learning Representations, 2015.

[4] Bengio, Yoshua; Simard, Patrice; Fasconi, Paolo: "Learning long-term dependencies with gradient descent is difficult", IEEE transactions on neural networks, Vol. 5, NO. 2, March 1994.

[5] Chan, W.; Jaitly, N.; Le, Q. V.; Vinyals, O.: "Listen, attend and spell", CoRR, vol. abs/1508.01211, 2015.

[6] Cheng, Heng-Tze; Haque, Zakaria; Hong, Lichan; Ispir, Mustafa; Mewald, Clemens; Polosukhin, Illia; Roumpos, Georgios; Sculley, D.; Smith, Jamie; Soergel, David; Tang, Yuan; Tucker, Philipp; Wicke, Martin; Xia, Cassandra; Xie, Jianwe: "TensorFlow Estimators: Managing Simplicity vs. Flexibility in High-Level Machine Learning Frameworks", 2017. Software available from `https://www.tensorflow.org/guide/estimators`.

[7] Dempster, A. P.; Laird, N. M.; Rubin, D. B. "Maximum Likelihood from Incomplete Data via the EM Algorithm". Journal of the Royal Statistical Society, Series B. 39 (1): 1-38. (1977).

[8] Deng, J.; Berg, A.; Satheesh, S.; Su, H.; Khosla, A.; Fei-Fei, L. ILSVRC-2012, 2012. `http://www.image-net.org/challenges/LSVRC/2012/`.

[9] Fukushima, Kunihiko. "Neocognitron: A hierarchical neural network capable of visual pattern recognition." Neural networks 1.2 (1988): 119-130.

[10] Garofolo, J. S.; Lamel, L. F.; Fisher, W. M.; Fiscus, J. G.; Pallett, D. S.; Dahlgren, N. L.: "DARPA TIMIT acoustic phonetic continuous speech corpus CDROM," 1993.

[11] Graves, Alex; Navdeep Jaitly. "Towards End-To-End Speech Recognition with Recurrent Neural Networks.", ICML 2014.

[12] Graves, Alex; Fernández, Santiago; Gomez, Faustino; Schmidhuber, Jürgen , "Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks", in Proceedings of the 23rd international conference on Machine learning - ICML 2006.

[13] Graves, Alex; M. Liwicki; S. Fernandez; R. Bertolami; H. Bunke; J. Schmidhuber, "A novel connectionist system for unconstrained handwriting recognition", IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 31, no. 5, pp. 855–868, 2009.

[14] Goodfellow, Ian; Bengio, Yoshua; Courville, Aaron, "Deep Learning", MIT Press, 2016, `http://www.deeplearningbook.org`.

[15] Hannun, Awni; Case, Carl; Casper, Jared; Catanzaro, Bryan; Diamos, Greg; Elsen, Erich; Prenger, Ryan; Satheesh, Sanjeev; Sengupta, Shubho; Coates, Adam; Ng, Andrew: "Deep Speech: Scaling up end-to-end speech recognition", 2014, `https://arxiv.org/abs/1412.5567`, `https://github.com/mozilla/DeepSpeech`

[16] Haykin, Simon: "Neural networks and learning machines", Third edition. McMaster University, Hamilton, Ontario, Canada.

[17] Hebb, D. O.: “The Organization of Behavior”. Wiley, New York, 1949.

[18] Hochreiter, Sepp; Schmidhuber, Jürgen: “Long short-term memory”, Neural Computation 9(8):1735-1780, 1997.

[19] Huang, Xuedong; Acero, Alex; Hon, Hsiao-Wuen: “Spoken Language Processing: A Guide to Theory, Algorithm and System Development”, Prentice Hall PTR (2001).

[20] Hwang, Kyuyeon; Sung, Wonyong: “Character-level incremental speech recognition with recurrent neural networks,” in IEEE International Conference on Acoustics, Speech and Signal Processing. IEEE, 2016, pp. 5335–5339.

[21] Luong, Minh-Thang; Pham, Hieu; Manning, Christopher: “Effective Approaches to Attention-based Neural Machine Translation”, Stanford University, 2015.

[22] Lyons, James: python_speech_features: common speech features for ASR, 2013, `https://github.com/jameslyons/python_speech_features`

[23] Leonard, Gary R. (1984): “A Database for speaker-independent digit recognition”. IEEE International Conference on Acoustics, Speech, Signal Processing (ICASSP), vol. 9. 9. 328 - 331. 10.1109/ICASSP.1984.1172716.

[24] McCulloch, W.S., Pitts, W.: “A logical calculus of the ideas immanent in nervous activity”. Bulletin of Mathematical Biophysics (1943) 5: 115.

[25] Nielsen, Michael: "Neural Networks and Deep Learning", Determination Press, 2015, `http://neuralnetworksanddeeplearning.com/`.

[26] Olah, Chirstopher: “Understanding LSTM Networks”, 27 de Agosto de 2015, `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`.

[27] Panayotov, Vassil; Chen, Guoguo; Povey, Daniel; Khudanpur, Sanjeev:”LibriSpeech: an ASR corpus based on public domain audio books”, ICASSP 2015.

[28] Rabiner, Lawrence R.: “A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition”. Proceedings of the IEEE, 77 (2), p. 257-286, February 1989.

[29] Rosenblatt, F.: “The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain”. Cornell Aeronautical Laboratory, Psychological Review, 65, 386-408, (1958).

[30] Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986b): “Learning internal representations by error propagation”. In D. E. Rumelhart and J. L. McClelland, editors, Parallel Distributed Processing, volume 1, chapter 8, pages 318–362. MIT Press, Cambridge.

[31] Salazar, Julian; Kirchhoff, Katrin; Huang, Zhiheng: “Self-attention networks for Connectionist Temporal Classification in Speech Recognition”, 2019.

[32] Scheidl Harald, ”Beam Search Decoding in CTC trained Neural Networks”: `https://towardsdatascience.com/beam-search-decoding-in-ctc-trained-neural-networks-5a889a3d85a7`

[33] Scheidl, Harald; Fiel, Stefan; Sablatnig, Robert, ”Word Beam Search: A Connectionist Temporal Classification Decoding Algorithm”, 16th International Conference on Frontiers in Handwriting Recognition (ICFHR 2018), Niagara Falls, New York, USA, 2018, page 253-258.

[34] Schuster, Mike, and Kuldip K. Paliwal. "Bidirectional recurrent neural networks." Signal Processing, IEEE Transactions on 45.11 (1997): 2673-2681.2.

[35] Shannon, C. E.: “A Mathematical Theory of Communication”, Bell System Technical Journal, 1948.

[36] SPHERE Conversion Tools, University of Pennsylvania: `https://www.ldc.upenn.edu/language-resources/tools/sphere-conversion-tools`.

[37] Vaswani, Ashish; Shazeer, Noam; Parmar,Niki; Uszkoreit, Jakob; Jones, Llion; Gomez, Aidan N.; Kaiser, Lukasz; Polosukhin, Illia: "Attention Is All You Need", 2017.

[38] Viterbi, A. J., "Error bounds for convolutional codes and an asymptotically optimum decoding algorithm", IEEE Trans. Inform. Theory, vol. IT-13, pp. 260-269, April 1967.

[39] Zorzano, Nicolás: "Reconocedor de dígitos con redes neuronales", Facultad de Ingeniería de la Universidad de Buenos Aires, 9 de Febrero de 2018, `https://github.com/nicozorza/spoken-digits-recognition`.

[40] Zorzano, Nicolás: "Reconocimiento de habla mediante redes neuronales", Facultad de Ingeniería de la Universidad de Buenos Aires, 22 de Septiembre de 2018, `https://github.com/nicozorza/speech-recognition`.

[41] Zorzano, Nicolás: "Diseño e implementación de un sistema end-to-end para la conversión de habla a texto mediante redes neuronales", Facultad de Ingeniería de la Universidad de Buenos Aires, `https://github.com/nicozorza/speech-to-text`.

[42] Zweig, Geoffrey; Yu, Chengzhu; Droppo, Jasha; Stolcke, Andreas: "Advances in all-neural speech recognition". ICASSP 2017: 4805-4809