UNIVERSIDAD NACIONAL DE LA PLATA

FACULTAD DE INFORMÁTICA

TESINA DE LICENCIATURA

TÍTULO: Generación de texto estructurado en español para batallas de freestyle utilizando modelos de Deep Learning
AUTORES: Dal Bianco, Pedro Alejandro
DIRECTOR: Dr. Ronchetti Franco
CODIRECTOR:
ASESOR PROFESIONAL:
CARRERA: Licenciatura en Informática

Resumen

Con la presencia cada vez mayor de la inteligencia artificial en gran variedad de áreas diferentes, el uso de técnicas de aprendizaje automático y deep learning para propósitos creativos también ha aumentado significativamente los últimos años. Este tipo de trabajos dentro del área de procesamiento de lenguaje natural típicamente toman la forma de modelos neuronales generadores de ficción o líricas, encontrándose en la mayoría de los casos en idioma inglés y sin una posibilidad de adaptarse fácilmente a otros idiomas. En este trabajo se desarrolló un sistema generador de texto que se enmarca en el estilo y estructura del subgénero del rap conocido como batallas de freestyle, que además del componente puramente creativo o artístico incluye un factor competitivo entre sus ejecutantes.

Palabras Clave

Tesina de grado, Aprendizaje Profundo, Procesamiento de Lenguaje Natural, Generación de Lenguaje Natural, Generación de texto estructurado.

Trabajos Realizados

En primer lugar se definieron dos bases de datos de textos, una primera con letras de canciones del género de rap y similares y una segunda compuesta por transcripciones de batallas de freestyle exclusivamente dado que no se encontraron al momento bases de datos públicas de este dominio. Sobre estos datos se experimentaron distintas técnicas de preprocesamiento y representación y se utilizaron para entrenar un modelo de lenguaje neuronal para la generación de texto que se corresponda con estos géneros. Por último, se desarrolló un sistema de generación de freestyle que incluye algoritmos que trabajan sobre el resultado del modelo de lenguaje para garantizar que el texto generado respete la estructura y la rima requerida por el género.

Conclusiones

El modelo de lenguaje desarrollado logró efectivamente captar el estilo del género a partir del entrenamiento sobre las bases de datos generadas, pero no así la correcta estructuración de este en estrofas de cuatro versos o el correspondiente uso de la rima. Sin embargo, a partir de algoritmos que garantizaron que el texto generado cumpliera con estas dos características fue posible generar texto que se corresponda no solo con el estilo si no también con la estructura correspondiente al género de las batallas de freestyle.

Trabajos Futuros

Se identifican como posibles trabajos futuros la implementación de un sistema que, en concordancia con el género de las batallas de *freestyle* sea capaz no solo de generar texto que del género si no que también pueda responder a una estrofa producida por un potencial contrincante; profundizar en la utilización de técnicas de *few-shot learning* y estudiar su aplicabilidad al problema desarrollado; experimentar con el uso de otras arquitecturas de redes neuronales para definir el modelo de lenguaje a utilizar e incrementar la cantidad de datos disponibles para el entrenamiento ya sea a partir de la recolección de nuevas transcripciones o a través del uso de *data augentation*.

Fecha de la presentación: *Febrero de 2021*

Generación de texto estructurado en español para batallas de freestyle utilizando modelos de Deep Learning

Director: Dr. Ronchetti Franco

Alumno: Dal Bianco, Pedro Alejandro

Facultad de Informática, Universidad Nacional de La Plata

Para obtener el grado de Licenciado en Informática

Febrero 2021

Índice general

1. Introducción **5**
- 1.1. Resumen . . . 5
- 1.2. Motivación . . . 5
- 1.3. Objetivos . . . 7
- 1.4. Organización del documento . . . 7

2. Deep Learning **9**
- 2.1. Modelos lineales . . . 10
  - 2.1.1. Clasificación binaria . . . 11
  - 2.1.2. Clasificación multiclase . . . 12
- 2.2. Función de error . . . 14
  - 2.2.1. Entropía Cruzada Categórica . . . 14
- 2.3. Descenso de Gradiente . . . 15
  - 2.3.1. Sobreajuste . . . 17
- 2.4. Redes neuronales . . . 18
  - 2.4.1. Funciones de activación . . . 21
  - 2.4.2. Entrenamiento de redes neuronales - *Back-propagation* . . . 21
  - 2.4.3. Dropout . . . 22
  - 2.4.4. Transfer Learning . . . 23
- 2.5. Redes neuronales recurrentes . . . 23
  - 2.5.1. RNNs bidireccionales . . . 24
  - 2.5.2. Arquitecturas con compuertas . . . 25

3. Procesamiento de Lenguaje Natural **29**
- 3.1. Preprocesamiento . . . 30
  - 3.1.1. Tokenización . . . 31
  - 3.1.2. Byte Pair Encoding . . . 33
- 3.2. Representación de documentos . . . 34
  - 3.2.1. Representación con Características Estáticas . . . 34
  - 3.2.2. Representación con Características Dinámicas . . . 35
  - 3.2.3. Representación con Características Aprendidas: *Word Embeddings* . . . 36
  - 3.2.4. Obtención de vectores de *Embeddings* . . . 37

3.3. Modelos de Lenguaje - Generación de Lenguaje Natural 42
3.3.1. Modelos de Lenguaje de n-gramas 43
3.3.2. Modelos de Lenguaje Neuronal con ventana fija 44
3.3.3. Modelos de Lenguaje con Redes Neuronales Recurrentes 45
3.4. Generación de texto estructurado 47
3.4.1. Batallas de *freestyle* 49
3.4.2. Métricas de evaluación 50
3.4.3. Trabajos relacionados 51

4. Desarrollo Realizado **55**
4.1. Generación de Base de Datos 55
4.2. Preprocesamiento y análisis de los datos 58
4.2.1. Análisis de vocabulario 58
4.2.2. Tokenización 60
4.3. Modelo de generación de texto 69
4.4. Entrenamiento del modelo de lenguaje 70
4.5. Generación de texto estructurado 73
4.5.1. Organización del texto en estrofas 74
4.5.2. Generación de texto bajo estructura de rima 74
4.5.3. Algoritmo de generación 82

5. Conclusiones y trabajos futuros **85**
5.1. Conclusiones generales 85
5.2. Lineas de trabajo futuras 86

Bibliografía **89**

A. Muestras de texto generado **95**

Capítulo 1

Introducción

1.1. Resumen

Con la presencia cada vez mayor de la inteligencia artificial en gran variedad de áreas diferentes, el uso de técnicas de aprendizaje automático y *deep learning* para propósitos creativos también ha aumentado significativamente los últimos años. Este tipo de trabajos dentro del área de procesamiento de lenguaje natural típicamente toman la forma de modelos neuronales generadores de ficción o líricas, encontrándose en la mayoría de los casos en idioma inglés y sin una posibilidad de adaptarse fácilmente a otros idiomas. En este trabajo se desarrolló un sistema generador de texto que se enmarca en el estilo y estructura del subgénero del rap conocido como batallas de *freestyle*, que además del componente puramente creativo o artístico incluye un factor competitivo entre sus ejecutantes.

La totalidad del desarrollo consistió, en primer lugar, en la generación de dos bases de datos, la primera compuesta por transcripciones de batallas de *freestyle* (no se encontró al momento de realización de este trabajo una base de datos pública que contenga este tipo de material) y una segunda que contiene letras de canciones de géneros similares; la definición de un modelo de lenguaje neuronal que fue entrenado con el texto contenido en las bases de datos y finalmente algoritmos que utilizan el modelo entrenado para la generación de texto que se corresponda con el estilo buscado respetando la rima y la métrica correspondientes. El modelo utilizado mostró ser capaz de captar el estilo del género en particular, incluidas particularidades como la utilización de de algunas expresiones en inglés o vocablos característicos de este. En conjunto con los algoritmos de generación desarrollados se logró generar texto realista en español acorde al género mencionado y que además respeta la estructura de estrofas y la rima.

1.2. Motivación

El procesamiento de lenguaje natural y más específicamente la generación de lenguaje natural es un área que ha resultado de interés desde hace ya tiempo. En un principio las técnicas más

utilizadas fueron el uso de plantillas (*templates*) o similares, sin embargo, el surgimiento de técnicas de *deep dearning* aplicadas a este campo en particular ha permitido un gran avance en el estado actual del arte [Deng and Liu, 2018]. Hoy en día, además, la disponibilidad de grandes bases de datos permiten la generación de modelos muy robustos en lo respecta a generación de lenguaje natural. GPT-3 [Brown et al., 2020], un modelo reciente entrenado con una versión de la base de datos Common Crawl[1] que consta de alrededor de un billón de palabras, es un claro ejemplo de este fenómeno. Las pruebas del artículo original han obtenido muy buenos resultados en el campo de la generación de texto entrenando primero sobre grandes bases de datos como el mencionado y luego realizando *fine-tuning* sobre textos específicos del estilo que se busca generar, y esto se puede ver, por ejemplo, en [Branwen, 2020] donde se utiliza dicho modelo para la generación de distintos géneros particulares como poesía, diálogos o textos humorísticos, entre otros.

La generación de textos estructurados, como la de poesía o lírica, entre otros, combina la necesidad de flexibilidad que brinda el uso de técnicas de *deep dearning* con reglas específicas acerca de la métrica, la acentuación y la rima del texto generado. Ejemplos de esto se pueden encontrar en [Ghazvininejad et al., 2016] que combina redes LSTM para la generación de texto poético con máquinas de estados finitos que aceptan el texto que cumple con la estructura deseada o [Lau et al., 2018] que utiliza *embeddings* y redes LSTM para la generación de varias líneas candidatas para formar sonetos y luego elige la que se mejor adapta a la estructura deseada en función de la métrica y la rima.

Si bien es posible encontrar trabajos que tratan la generación de texto poético en otros idiomas además del inglés, como [Yi et al., 2017] donde se utilizan redes neuronales recurrentes entrenadas sobre poesía clásica china o [Zugarini et al., 2019] que entrena un modelo basado en sílabas con poemas en italiano de Dante Alighieri, el único trabajo que se encontró que utiliza el lenguaje español es [Ghazvininejad et al., 2016], referenciado anteriormente. Si bien este último trabajo se enfoca principalmente en la generación de texto en inglés, en su última sección muestra una prueba de generación de texto en español utilizando el modelo desarrollado entrenado con un banco de letras de canciones en español y texto extraído de Wikipedia para la generación de poesía.

Otros trabajos que tratan la generación de textos de géneros similares al *freestyle* se ven en [Potash et al., 2015] donde se generan letras de rap en inglés, sin embargo las batallas de *freestyle* le agregan un componente por el que resulta aún de mayor interés para la generación automática de texto: el objetivo de estas es que el texto sea improvisado en tiempo real en respuesta a la rima de un competidor. Es decir que además de la generación de texto en sí, y que respete la rima y la métrica adecuada, es necesario que esto sea realizado en un tiempo casi inmediato, característica que puede proporcionar un modelo de generación automática.

[1] https://commoncrawl.org/the-data/

1.3. Objetivos

El objetivo de esta tesina es desarrollar un modelo generador de texto estructurado que logre generar texto en español que respete las características utilizadas en el marco de competencias de *freestyle*, como son la rima y la métrica, utilizando técnicas de *deep learning*. Se presentan como objetivos específicos:

1. Generar bases de datos que contengan las transcripciones de batallas de freestyle y permitan el entrenamiento de modelos de Generación de Lenguaje Natural (NLG, por sus siglas en inglés).
2. Estudiar, implementar y comparar distintas técnicas de procesamiento de texto para el entrenamiento de modelos de *deep learning*, tales como el uso de *word embeddings* (representación de palabras como vectores de números reales) o diferentes formas de tokenización.
3. Estudiar los modelos neuronales utilizados para la generación de texto, particularmente las redes neuronales recurrentes (RNN) tales como las LSTM, y definir una arquitectura para un modelo de lenguaje que se entrene luego con las bases de datos generadas.
4. Implementar un sistema que utilize el modelo de lenguaje entrenado para la generación de texto que respondan a las formas y el estilo de los utilizados en batallas de *freestyle*. Además, se utilizarán técnicas que permitan garantizar que el texto generado cumpla con la estructura y rima acorde.

1.4. Organización del documento

La presente tesina se encuentra estructurada de la siguiente forma:

En el capítulo 2 se presenta el marco teórico en lo que refiere a aprendizaje automático y aprendizaje profundo necesario para llevar a cabo el desarrollo de este trabajo. Se describe el concepto de modelo de aprendizaje automático y en particular las redes neuronales, su entrenamiento y como es posible utilizarlas para tareas específicas como la clasificación binaria o multiclase. Además, se describen algunas arquitecturas particulares de redes comúnmente utilizadas para el procesamiento de secuencias de texto tales como las redes neuronales recurrentes y más específicamente las LSTM.

En el capítulo 3 se profundiza en el area de procesamiento de lenguaje natural y se desarrollan conceptos claves en este campo tales como la tokenización, la representación de documentos y el caso específico de los *word embeddings*, el uso de modelos de lenguaje y la problemática específica de trabajar con texto estructurado, más puntualmente con el género de las batallas de *freestyle*.

En el capítulo 4 se presenta el desarrollo realizado y el proceso con el que se llevó a cabo. Dentro de este se encuentran la confección de dos bases de datos de textos enmarcados en los

géneros de rap y de *freestyle* respectivamente, el análisis de los textos obtenidos y pruebas con diferentes formas de tokenización para finalmente el entrenamiento de un modelo de lenguaje neuronal que se utiliza para la generación de este tipo de texto. Luego, se describe también el programa desarrollado para garantizar que el texto generado cumpla con la estructura y rima correspondiente al género.

Por último, en el capítulo 5 se presentan las conclusiones obtenidas y se listan posibles lineas de trabajo a futuro para continuar con lo desarrollado en esta tesina.

Capítulo 2

Deep Learning

La inteligencia artificial (AI por sus siglas en inglés) es un campo dentro de las ciencias de la computación que busca entender los principios que hacen posible el comportamiento inteligente, ya sea en sistemas naturales o artificiales. Esto se hace bajo la hipótesis de que el razonamiento es un proceso computable [Poole et al., 1998]. El verdadero desafío de la inteligencia artificial se encontró en resolver problemas que para un humano pudieran resultar fáciles de llevar a cabo pero difíciles de describir formalmente, tales como reconocer palabras a partir de la escucha o identificar un rostro en una imagen [Goodfellow et al., 2016].

El aprendizaje automático o *machine learning* (ML) es una rama de la inteligencia artificial que busca que en lugar de que una computadora actúe en función de instrucciones explicitadas previamente por un humano, esta aprenda la forma correcta de realizar una tarea a partir de un conjunto de datos. Esto da lugar a un nuevo paradigma para la inteligencia artificial: en la inteligencia artificial tradicional un humano ingresa reglas (un programa) y un conjunto de datos a ser procesados para finalmente obtener respuestas al procesamiento de los datos, mientras que en lo que refiere a aprendizaje automático se ingresarán los datos en conjunto con las respuestas esperadas de su procesamiento y se espera obtener las reglas (el programa) que permitan realizar dicho procesamiento. Estas reglas obtenidas pueden luego ser aplicadas a nuevos datos para obtener respuestas originales [Chollet et al., 2018]. La figura 2.1 busca ilustrar la diferencia descrita entre estos paradigmas.

Entonces, un sistema de aprendizaje automático es "entrenado" en lugar de explícitamente programado. Este proceso de entrenamiento se lleva a cabo presentándole al sistema distintos ejemplos que resulten relevantes para la tarea a realizar en busca de que este sea capaz de encontrar una estructura estadística en estos que le permita obtener reglas para automatizar dicha tarea [Chollet et al., 2018].

El llamado aprendizaje profundo o *deep learning* es una subrama del aprendizaje automático que pone énfasis en aprender *capas* de representaciones significativas de los datos de forma sucesiva. Todas estas capas de representaciones se aprenden, al igual que en el aprendizaje automático, a partir de datos durante el entrenamiento. En esta subrama, dichas representaciones

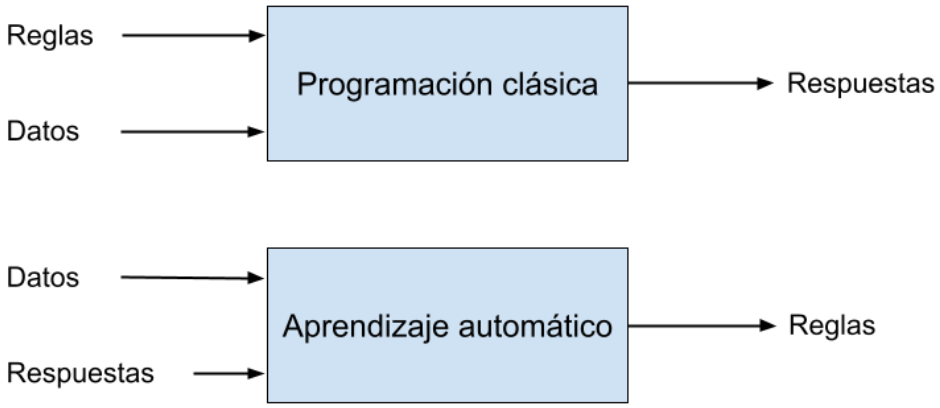


Figura 2.1: Diferencia entre paradigma de inteligencia artificial clásico y el aprendizaje automático.

son aprendidas a través de modelos conocidos como redes neuronales, estructuradas en capas literales "apiladas" una encima de la otra [Chollet et al., 2018]. Estos modelos se desarrollan de forma más extensa en la sección 2.4. La figura 2.2 ilustra a través de un diagrama de Venn como se relacionan los conceptos de inteligencia artificial, aprendizaje automático y aprendizaje profundo descritos.

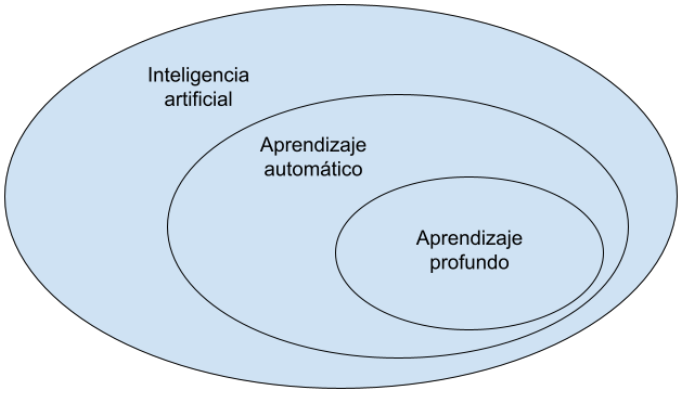


Figura 2.2: Diagrama de Venn que relaciona los conceptos de inteligencia artificial, aprendizaje automático y aprendizaje profundo.

2.1. Modelos lineales

Como se mencionó anteriormente, a través del aprendizaje automático se busca obtener a partir de datos que constituyan la entrada y salida esperada para una tarea específica, un programa capaz de realizar dicha tarea. Dado que el obtener el programa buscado entre el conjunto de

la totalidad de los posibles programas (o posibles funciones) resulta un problema muy difícil es necesario restringirse a familias específicas de programas (o funciones) conocidas como clases de hipótesis. Al restringirse a una clase de hipótesis en particular se establece lo que se conoce como un sesgo inductivo, un conjunto de asunciones acerca de la forma que va a tener la solución, lo que facilita el proceso de encontrarla. Además, la clase de hipótesis determina que es lo que puede y no puede ser representado por el programa resultante [Goldberg and Hirst, 2017] [Mohri et al., 2018].

Una de las clases de hipótesis más comunes dentro del aprendizaje automático es la de las funciones lineales de muchas dimensiones. Estas son, dadas una cantidad de entradas d_e y una cantidad de salidas d_s, las funciones de la forma:

$$\begin{gathered} f(\boldsymbol{x}) = \boldsymbol{x}\boldsymbol{W} + \boldsymbol{b} \\ \boldsymbol{x} \in \mathbb{R}^{d_e}, \boldsymbol{W} \in \mathbb{R}^{d_e \times d_s}, \boldsymbol{b} \in \mathbb{R}^{d_s} \end{gathered} \tag{2.1}$$

En este caso, el vector $\boldsymbol{x}$ es la entrada de la función mientras que la matriz $\boldsymbol{W}$ y el vector $\boldsymbol{b}$ son los llamados parámetros. El entrenamiento de un modelo dentro de esta clase de hipótesis consistirá entonces en ajustar los valores de los parámetros $\boldsymbol{W}$ y $\boldsymbol{b}$ para que la función se comporte de la forma esperada para un conjunto de datos compuesto por valores de entrada (x_1, x_2, ..., x_n) y sus correspondientes valores de salida esperados (y_1, y_2, ..., y_n).

Si bien la clase de hipótesis de las funciones lineales esta restringida a representar exclusivamente relaciones lineales, modelos dentro de esta clase han sido los principales enfoques en lo que refiere a procesamiento de lenguaje natural estadístico y sirven como unidades básicas en la construcción de modelos no lineales más poderosos como lo son las redes neuronales (desarrolladas en la sección 2.4) [Goldberg and Hirst, 2017]. Además, resultan útiles para introducir conceptos utilizados también en modelos más complejos de aprendizaje profundo. Por estas razones es que se ha decidido incluirlos en este capítulo.

2.1.1. Clasificación binaria

Los problemas llamados de clasificación binaria consisten en determinar para una entrada si pertenece a una de dos clases particulares, típicamente representadas con los valores 0 y 1 o llamadas también clase negativa y positiva respectivamente. Un ejemplo de este tipo de problema puede serlo la clasificación de reseñas de un producto entre positivas y negativas, para lo que se necesitaría un modelo capaz de tomar como entrada una representación adecuada de cada reseña (las formas de representación de textos para procesamiento del lenguaje natural se desarrollan en el capítulo 3) y cuya salida fuera un 0 o un 1 representando la clase de pertenencia de la reseña.

Dado que para cada entrada se corresponde una única salida perteneciente al conjunto $\{0, 1\}$, para modelar este tipo de problemas se utiliza una versión restringida de la ecuación 2.1 donde $d_s = 1$, donde $\boldsymbol{w}$ es un vector y b un escalar:

$$f(\boldsymbol{x}) = \boldsymbol{x}\boldsymbol{w} + b \tag{2.2}$$

Entonces, para un problema de clasificación binaria, la salida del modelo debe indicar si su correspondiente entrada pertenece o no a una clase determinada, representadas por los valores 0 y 1. Sin embargo, la salida del modelo definido en 2.2 se encuentra en el rango $[-\infty, +\infty]$. Resulta necesario entonces a partir de este obtener uno de clasificación binaria, lo que es posible definiendo un valor U conocido típicamente como umbral de decisión y redefiniendo el modelo de la siguiente manera:

$$f(\boldsymbol{x}) = \begin{cases} 0 & \text{si } f(\boldsymbol{x}) = \boldsymbol{x}\boldsymbol{w} + b \leq U \\ 1 & \text{si } f(\boldsymbol{x}) = \boldsymbol{x}\boldsymbol{w} + b > U \end{cases} \tag{2.3}$$

De esta forma es posible entrenar un modelo lineal que permita clasificar nuevos valores de entrada en una de las clases del conjunto $\{0, 1\}$. Si se puede separar el conjunto de datos del problema efectivamente usando un modelo lineal entonces el conjunto de datos es considerado linealmente separable. Esto se puede ver como que existe un hiperplano que separa los datos del conjunto entre los que pertenecen a cada clase de forma perfecta [Mohri et al., 2018].

Modelo lineal logístico - Función Sigmoidea

El modelo de clasificación binaria descrito permite clasificar efectivamente una entrada entre las clases del conjunto $\{0, 1\}$, pero no brinda información acerca de la confianza de dicha clasificación. Un modelo que predice la probabilidad de un evento de suceder o no (o la probabilidad de pertenencia o no a una clase) se lo conoce como modelo logístico. Una posible forma de obtener un modelo logístico para utilizar en un problema de clasificación binaria es a través de la composición de la función correspondiente a un modelo lineal (ecuación 2.2) con la función Sigmoidea:

$$\sigma(f(\boldsymbol{x})) = \frac{1}{1 + e^{-\boldsymbol{x}\boldsymbol{w}+b}} \tag{2.4}$$

En la figura 2.3 se puede ver el gráfico de la función Sigmoidea cuya imagen es el intervalo $[0, 1]$. Cuando se utiliza este modelo en conjunto con una función de error adecuada (se desarrolla en la sección 2.2) es posible interpretar la salida del modelo $\sigma(f(x))$ para una entrada x dada como la probabilidad de pertenencia de x a la clase 1 (también llamada clase positiva). También se obtiene que $1-\sigma(f(x))$ se considera la probabilidad de pertenencia de x a la clase 0 (o negativa).

2.1.2. Clasificación multiclase

Si bien existen casos de problemas que se pueden representar como de clasificación binaria, son comunes también aquellos que tienen una naturaleza de clasificación multiclase [Goldberg and Hirst, 2017]

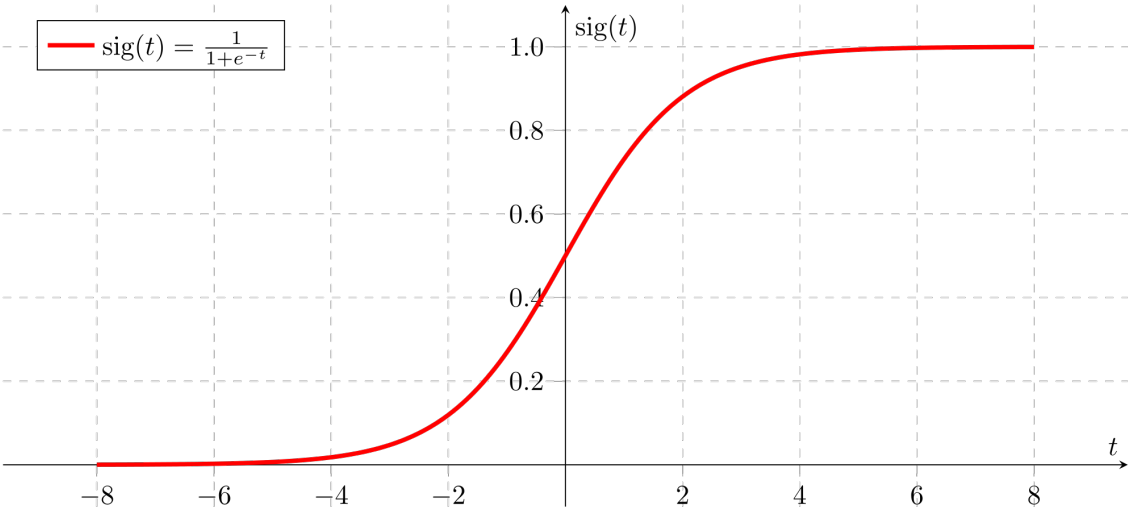


Figura 2.3: Gráfica de la función Sigmoidea.

[Mohri et al., 2018]. Para un problema de clasificación multiclase con k clases distintas, a cada entrada se le asocia una etiqueta del conjunto $\{0, 1, ..., k\}$ que representa la clase a la que pertenece. Un ejemplo de este tipo de problemas es la clasificación de un artículo periodístico en distintas categorías (política, deportes, etc.) o dada una secuencia de palabras cual es la palabra que debiera ir a continuación donde cada posible palabra representa una clase distinta en la que el modelo puede clasificar la entrada (este último ejemplo se encuentra desarrollado con más detalle en la sección 3.3).

Una posible forma de implementar un clasificador multiclase para k clases a partir del modelo lineal descrito en la ecuación 2.1 es definir la dimensión de salida del modelo $d_s = k$. Entonces $\boldsymbol{W} \in \mathbb{R}^{d_e \times k}$ y cada una de sus columnas serán los pesos utilizados para predecir la pertenencia de una entrada a una de las k clases en particular. La salida del modelo ahora pertenece al conjunto $\mathbb{R}^k$, y se puede considerar el índice del valor más alto del vector como la clase asignada por el modelo para una entrada particular. Entonces, un posible modelo de clasificación multiclase puede tener la forma:

$$f(\boldsymbol{x}) = argmax(\boldsymbol{x}\boldsymbol{W} + \boldsymbol{b}) \tag{2.5}$$

Este no es el único enfoque posible para modelar problemas de clasificación multiclase, pero desarrollar la totalidad de estos queda por fuera del alcance de este trabajo. Es posible encontrar un buen acercamiento a estos en [Allwein et al., 2000].

Modelo lineal logístico - Función Softmax

Así como en clasificación binaria se utiliza la función Sigmoidea para a partir del modelo lineal obtener una estimación de probabilidades, de forma análoga se utiliza en clasificación multiclase

la función Softmax:

$$softmax(\boldsymbol{x}_i)) = \frac{e^{\boldsymbol{x}_i}}{\sum_j e^{\boldsymbol{x}_j}} \tag{2.6}$$

Esta transformación fuerza que los valores del vector resultante sean positivos y que la suma de estos sea igual a 1, lo que permite interpretarlos como una distribución de probabilidades. El modelo utilizado para predicción multiclase resulta entonces en la aplicación de la función Softmax al resultado del modelo definido en 2.5:

$$f(\boldsymbol{x}) = softmax(\boldsymbol{x}\boldsymbol{W} + \boldsymbol{b}) \tag{2.7}$$

2.2. Función de error

Como se mencionó anteriormente, un algoritmo de aprendizaje automático recibe como entrada un conjunto de entrenamiento que se puede definir como un par de vectores $\boldsymbol{x}$ e $\boldsymbol{y}$ de tamaño n que representan ejemplos de entrenamiento y su correspondiente valor esperado respectivamente. El objetivo de este algoritmo es el de encontrar una función f que mapee de forma precisa los ejemplos con el correspondiente valor esperado, es decir que sus predicciones $\boldsymbol{y'} = f(\boldsymbol{x})$ sobre los ejemplos del conjunto de entrenamiento $\boldsymbol{x}$ sean correctos.

Para poder obtener una medición formal de que tan certeras son las predicciones de un modelo se define una función de error o *loss function* $L(y', y)$, que asigna un valor numérico a una predicción y' en función de su proximidad con el valor esperado y. Dado un conjunto de entrenamiento determinado, es posible fijar sus valores para calcular el error total del modelo en función de sus parámetros, comúnmente denominados Θ (en los modelos lineales vistos estos son $\boldsymbol{W}$ y $\boldsymbol{b}$). Entonces, el error total $\mathcal{L}$ de un modelo f con parámetros Θ para un conjunto de prueba $\boldsymbol{x}$, $\boldsymbol{y}$ se define como:

$$\mathcal{L}(\Theta) = \frac{1}{n}\sum_{i=1}^{n} L(f(x_i), y_i) \tag{2.8}$$

El objetivo entonces del algoritmo de aprendizaje automático será encontrar los valores de parámetros Θ que minimicen el error total del modelo $\mathcal{L}$.

2.2.1. Entropía Cruzada Categórica

Para los problemas de clasificación multiclase es común la utilización de la función conocida como Entropía Cruzada Categórica (*categorical cross-entropy*) como función de error ya que permite medir la disimilitud entre dos distribuciones de probabilidades [Goldberg and Hirst, 2017], por lo que resulta relevante en el desarrollo de este trabajo. Al ser usada como función de error

para este tipo de problemas, se utiliza para medir la similitud entre la distribución de probabilidades generada por el modelo para una entrada respecto a su clase real. Para poder interpretar la clase real de un ejemplo como una distribución de probabilidades se utiliza la llamada representación *one-hot*, que consiste en, para un problema con k clases distintas, representar a cada etiqueta de una clase i como un vector de dimensión k con el valor 1 en la posición i y el valor 0 en las posiciones restantes.

La definición general para la Entropía Cruzada Categórica es la siguiente:

$$L(\boldsymbol{y'}, \boldsymbol{y}) = -\sum_i \boldsymbol{y}_i log(\boldsymbol{y}'_i) \tag{2.9}$$

Sin embargo, cuando la distribución de probabilidades real tiene la forma de un vector *one-hot* esta se puede simplificar a:

$$L(\boldsymbol{y'}, \boldsymbol{y}) = -log(\boldsymbol{y}'_t) \tag{2.10}$$

Siendo t el índice de la clase real para un ejemplo dado. Esta función penalizará a las distribuciones de probabilidad que más difieran en la probabilidad asignada a la clase real (y por ende, como el total de probabilidades suma 1, también diferirán más en las probabilidades asignadas a las demás posibles clases).

2.3. Descenso de Gradiente

Existen distintos métodos para encontrar los valores de los parámetros Θ que minimicen el error total del modelo. Es posible pensar en primer lugar en soluciones analíticas para modelos sencillos como los lineales ya descritos, sin embargo estos métodos suelen ser computacionalmente muy costosos a medida que crece la cantidad de parámetros y no son generalizables a otros modelos más complejos. Es por esto que uno de los métodos más ampliamente utilizado hoy en día en algoritmos de aprendizaje automático en general y particularmente en algoritmos de aprendizaje profundo es el conocido como descenso de gradiente [Goodfellow et al., 2016].

El descenso de gradiente es un método efectivo para entrenar modelos de aprendizaje automático que recibe una función f con sus respectivos parámetros Θ, una función de error L y un conjunto de entrenamiento $\boldsymbol{x}$ e $\boldsymbol{y}$ y buscará establecer los valores de Θ para minimizar el error total del modelo [Goldberg and Hirst, 2017] [Montavon et al., 2012]. Este presenta las ventajas de ser un algoritmo iterativo, generalizable (se puede utilizar para otros modelos además de los lineales, como Máquinas de Vectores de Soporte o redes neuronales) y más rápido que los métodos analíticos cuando se utilizan grandes cantidades de datos.

Este método consiste en iterativamente calcular el vector gradiente de la función de error L respecto a los parámetros Θ habiendo fijado los valores del conjunto de entrenamiento. Luego, se actualizan los valores de Θ sustrayéndole el valor del vector gradiente calculado en el punto

Θ multiplicado por un valor α conocido como factor de aprendizaje o *learning rate*, lo que equivale a desplazarse sobre L en la dirección de máximo decrecimiento en ese punto. Esto se ilustra, para un $\Theta \in \mathbb{R}^2$ en la figura 2.4. El factor de aprendizaje es un valor escalar positivo que determina que tan grandes serán las actualizaciones de los parámetros Θ o los saltos en el desplazamiento respecto a la función de error L. Un valor del factor de aprendizaje muy bajo implica que se necesitarán más pasos para llegar a converger en un mínimo de L, pero un valor muy alto podría llevar al algoritmo a divergir. Esto se ilustra en la figura 2.5 para un $\Theta \in \mathbb{R}$. No hay un valor ideal definido para la selección del factor de aprendizaje, son prácticas comunas probar con diferentes valores y evaluar el desempeño del entrenamiento o disminuir el valor de este a lo largo de las iteraciones [Goodfellow et al., 2016] [Goldberg and Hirst, 2017]. Por último, el criterio de corte para el algoritmo suele definirse por una cantidad fija de iteraciones sobre el conjunto de entrenamiento o el alcanzar un valor determinado de error total del modelo.

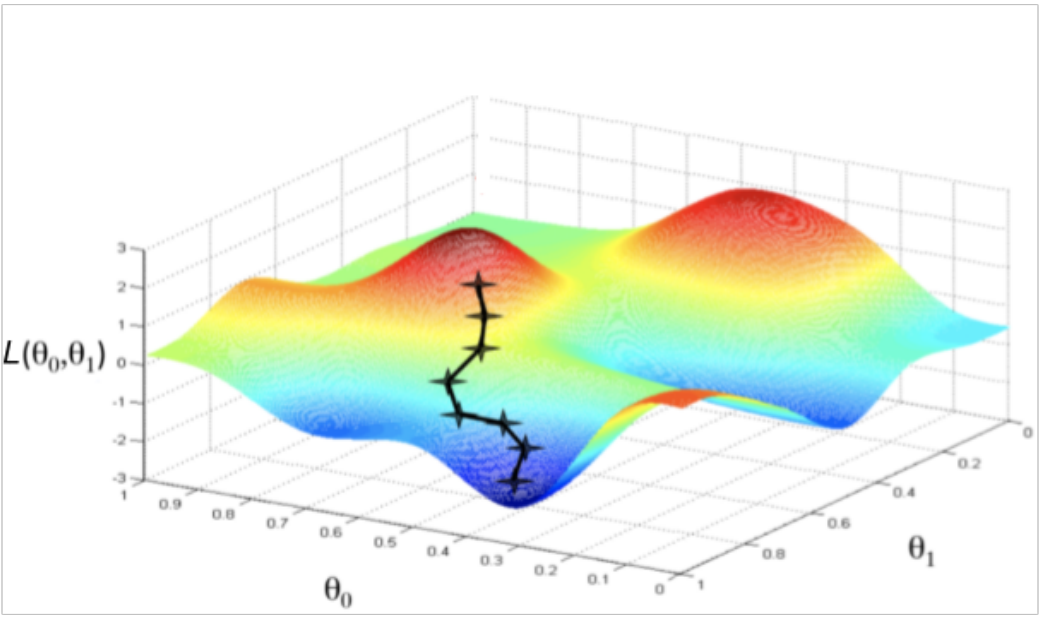
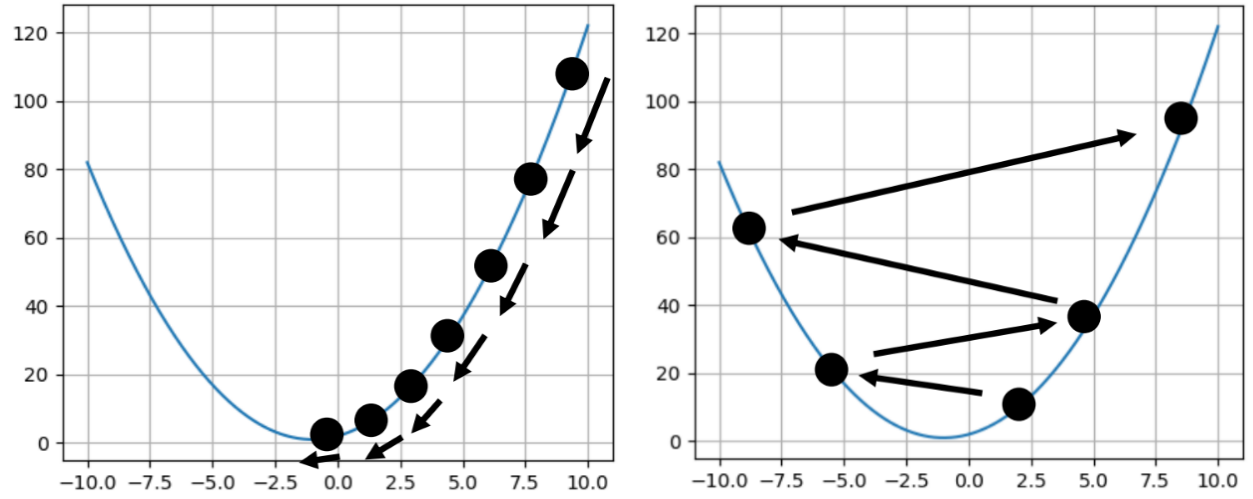


Figura 2.4: Ilustración del proceso de descenso de gradiente en un espacio tridimensional con $\Theta \in \mathbb{R}^2$.

El algoritmo 1 muestra el pseudocódigo correspondiente al descenso de gradiente, más específicamente de una versión de este conocida como descenso de gradiente estocástica por mini lotes. La diferencia entre ambas versiones es que al momento de calcular los gradientes, en el descenso de gradiente tradicional se tiene en cuenta todo el conjunto de entrenamiento, mientras que en la versión aquí presentada se calculan en función de subconjuntos de este llamados mini lotes que se obtienen de forma aleatoria en cada iteración. Esto permite trabajar con conjuntos de datos de mayor tamaño los cuales resultaría poco práctico procesar en su totalidad para cada



(a) Con un factor de aprendizaje pequeño se alcanzará un mínimo en mas pasos.

(b) Con un factor de aprendizaje grande el descenso de gradiente puede divergir.

Figura 2.5: Gráfico del error en función de un parámetro Θ a lo largo de iteraciones del descenso de gradiente con factores de aprendizaje mas chico (2.5a) y mas grande (2.5b)

iteración.

Algoritmo 1: Entrenamiento a través de descenso de gradiente estocástico

Data:

- Función $f(\boldsymbol{x}, \Theta)$ con sus respectivos parámetros Θ.
- Conjunto de entrenamiento compuesto por los vectores $\boldsymbol{x}$ e $\boldsymbol{y}$ de entradas y su correspondiente valor esperado.
- Función de error L

while *No se cumpla el criterio para finalizar* **do**

Tomar de forma aleatoria un mini lote (*minibatches*) de m ejemplos del conjunto de entrenamiento;

Computar el estimador del vector gradiente: $\hat{\boldsymbol{g}} \leftarrow \frac{1}{m} \nabla_\Theta \sum_i L(f(x_i, \Theta), y_i)$;

Actualizar los parámetros: $\Theta \leftarrow \Theta - \alpha \hat{\boldsymbol{g}}$;

end

2.3.1. Sobreajuste

Uno de los factores más relevantes dentro del aprendizaje automático es la relación entre la optimización y la generalización. La optimización se refiere a como se ajusta un modelo a un conjunto de datos sobre el cual se entrena, mientras que la generalización se refiere a que tan bien se desempeña el modelo entrenado sobre datos que no haya visto antes [Chollet et al., 2018]. Es una práctica común, para evaluar estos dos factores la separación del conjunto de datos en dos subconjuntos: uno utilizado como conjunto de entrenamiento sobre el cual se ejecutará el

algoritmo de entrenamiento correspondiente como lo puede ser el descenso de gradiente y otro conjunto de prueba que se utilizará para evaluar el desempeño del modelo en datos sobre los que no haya entrenado.

Típicamente, en una primera etapa de entrenamiento la optimización y la generalización se encuentran correlacionadas: a medida que disminuye el error sobre el conjunto de entrenamiento también lo hace sobre el de prueba. En esta etapa se dice que el modelo está subajustando (*underfitting*): el modelo aún no es capaz de modelar todos los patrones relevantes en los datos disponibles. Sin embargo, a medida que avanza el entrenamiento es común que la capacidad de generalización del modelo (evaluada como su desempeño sobre el conjunto de prueba) se estanque o incluso empiece a disminuir mientras el desempeño sobre el conjunto de entrenamiento continúa mejorando. En este punto se considera que el modelo está sobreajustando (*overfitting*): esta aprendiendo patrones específicos de los datos presentes en el conjunto de entrenamiento y que resultan engañosos o irrelevantes para el análisis de datos por fuera de este [Chollet et al., 2018].

Una primer solución para este problema es simplemente la obtención de más datos: naturalmente un modelo entrenado sobre una mayor cantidad de datos será capaz de generalizar mejor. En caso de no ser posible, otra opción es la de utilizar un modelo más sencillo, lo que genera que sea capaz de identificar una menor cantidad de patrones del conjunto de entrenamiento y forzará al proceso de optimización a enfocarse en los más relevantes, lo que le da una mejor posibilidad de generalizar correctamente. Otro método para evitar el sobreajuste se conoce como regularización. Este método consiste en forzar los pesos de la red a tomar valores pequeños, lo que resulta en una distribución más regular de estos y traduciéndose también en un modelo más simple. Para lograr esto se agrega a la función de error utilizada para el entrenamiento un costo asociado a tener pesos con valores grandes en la red. Las formas más comunes de regularización se conocen como L1 y L2 y se diferencian en la forma en la que se computa dicho costo: en la primera se obtiene a partir del valor absoluto de los pesos de la red mientras que en la segunda se obtiene a partir del cuadrado de estos.

2.4. Redes neuronales

Como ya se mencionó en la sección 2.1, los modelos lineales de aprendizaje automático permiten representar exclusivamente relaciones lineales. En caso de buscar representar relaciones más complejas resulta necesario utilizar modelos no lineales, como es el caso de las redes neuronales. Estas pueden pensarse como la composición de modelos lineales con funciones no lineales llamadas funciones de activación y la red neuronal más simple, conocida como Perceptrón es equivalente al modelo lineal descrito en la ecuación 2.1:

$$\begin{gathered} NN_{\text{Perceptrón}}(\boldsymbol{x}) = \boldsymbol{x}\boldsymbol{W} + \boldsymbol{b} \\ \boldsymbol{x} \in \mathbb{R}^{d_e}, \boldsymbol{W} \in \mathbb{R}^{d_e \times d_s}, \boldsymbol{b} \in \mathbb{R}^{d_s} \end{gathered} \tag{2.11}$$

Una forma común de representar redes neuronales es a través de un grafo donde los nodos llamados neuronas son unidades de cómputo que reciben uno o más valores escalares como entradas y otro como salida. Cada neurona computa el producto de cada una de sus entradas por un respectivo valor llamado peso, representado por las aristas de la red y que se corresponden con los parámetros del modelo (los valores que se buscan optimizar), luego suma[1] los valores obtenidos y les aplica una función no lineal cuyo resultado es la salida de la neurona. La figura 2.6 muestra una vista del Perceptrón descrito en la ecuación 2.11 representado de esta manera.

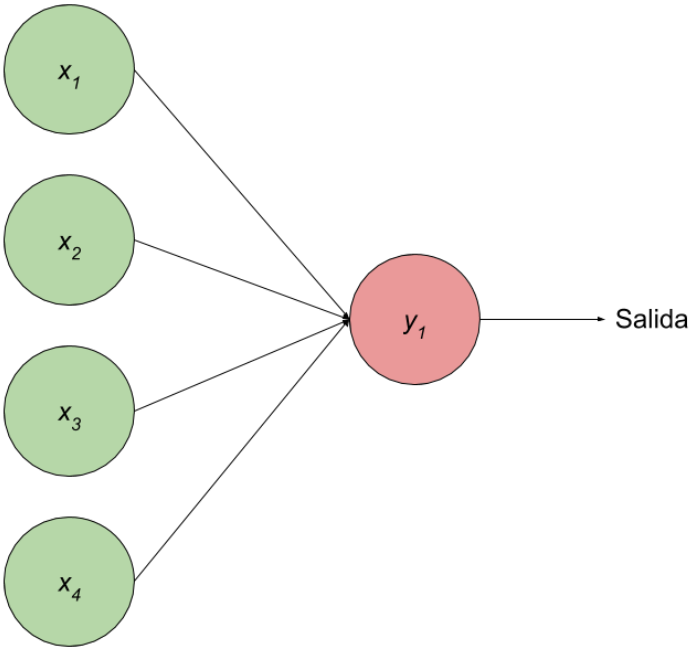


Figura 2.6: Representación de un Perceptrón en forma de grafo con $d_e = 4$ y $d_s = 4$.

Como se puede ver en la figura 2.11, las neuronas se organizan en capas que reflejan el flujo de la información y se corresponden en este caso con la salida de cada transformación lineal. Las capas que resultan efectivamente de transformaciones lineales se las conoce como capas totalmente conectadas (*fully-conected layers*) o capas densas, pero no son la única arquitectura posible, existen otros tipos de capas como las llamadas Convolucionales o de *Pooling*, entre otras, que quedan por fuera del alcance de este trabajo. La primera capa representa los valores de entrada al modelo, la última la salida de este y las capas intermedias se denominan capas ocultas. La cantidad de capas del modelo es conocida comúnmente como profundidad de la red [Goldberg and Hirst, 2017]. Modelos como el descrito, donde la información fluye desde la capa

[1]Si bien la suma es la operación más común, existen casos de la utilización de otras funciones tales como el máximo.

de entrada hacia la de salida a través de las capas intermedias sin ningún tipo de retroalimentación se los denomina redes neuronales *feed-forward* [Goodfellow et al., 2016]. Cuando una red incorpora conexiones de retroalimentación se la considera una red neuronal recurrente y estas son descriptas con mas detalle en la sección 2.5.

Sin embargo, el Perceptrón se sigue correspondiendo con un modelo lineal. Para poder obtener modelos no lineales resulta necesario incluir otra capa oculta que incorpore, precisamente, una función no lineal, dando lugar al llamado Perceptrón Multicapa con una capa oculta. Este tipo de red neuronal, ilustrado en forma de grafo en la figura 2.7, tiene la forma:

$$\begin{gathered} NN_{\text{PMC1}}(\boldsymbol{x}) = g(\boldsymbol{x}\boldsymbol{W_1} + \boldsymbol{b_1})\boldsymbol{W_2} + \boldsymbol{b_2} \\ \boldsymbol{x} \in \mathbb{R}^{d_e}, \boldsymbol{W_1} \in \mathbb{R}^{d_e \times d_1}, \boldsymbol{b_1} \in \mathbb{R}^{d_1}, \boldsymbol{W_2} \in \mathbb{R}^{d_1 \times d_s}, \boldsymbol{b_2} \in \mathbb{R}^{d_s} \end{gathered} \tag{2.12}$$

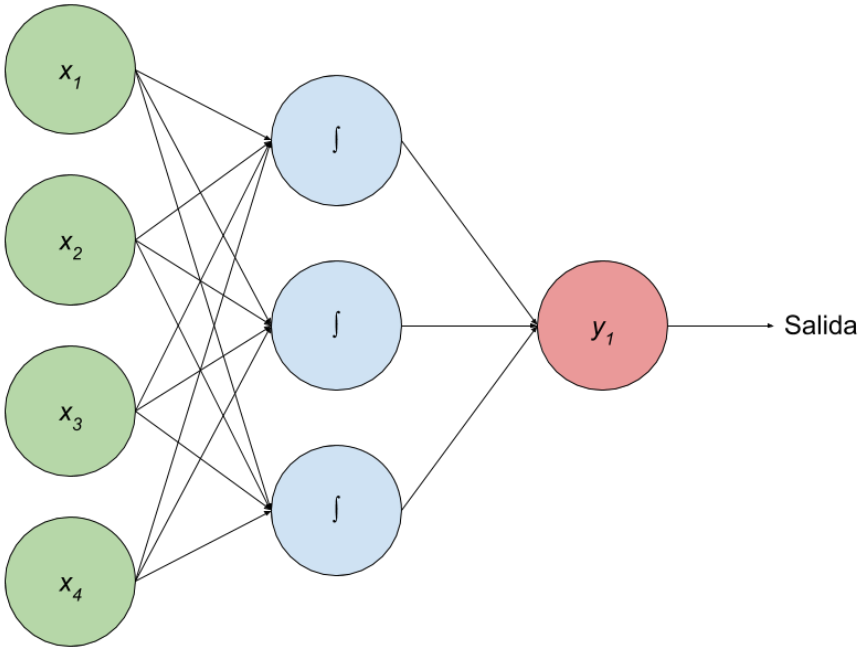


Figura 2.7: Representación de un Perceptrón Multicapa en forma de grafo con $d_e = 4$ y $d_s = 4$ y una capa oculta de dimensión 3.

Entonces, para describir una red neuronal se debe especificar la cantidad de capas en conjunto con su respectiva dimensión, la dimensión de entrada d_e y la dimensión de salida d_s. Cuando $d_s = 1$ la salida será un escalar y se puede utilizar la red para modelar problemas de regresión o clasificación binaria de la misma forma que los modelos lineales como se describió en la sección 2.1. Así también, cuando $d_s = k$ es posible modelar un problema de clasificación de

k clases. La diferencia con los modelos lineales es que una red neuronal tiene, en términos de poder de representación, la capacidad de un aproximador universal. Esto quiere decir que, en teoría, es capaz de aproximar todas las funciones continuas de un subconjunto cerrado de $\mathbb{R}^n$ [Hornik et al., 1989] [Cybenko, 1989].

Aspectos de una red neuronal tales como la cantidad de capas o sus respectivas cantidades de neuronas entran dentro de lo que se conoce como hiperparámetros de la red. Estos son parametros que se utilizan para controlar el proceso de aprendizaje de la red y no pueden ser aprendidos, si no que deben ser definidos manualmente. Otros ejemplos de estos lo son el factor de aprendizaje o el tamaño de los mini lotes utilizados para el descenso de gradiente.

2.4.1. Funciones de activación

Existen distintas funciones no lineales típicamente usadas como funciones de activación en redes neuronales. Que función utilizar depende mayormente de pruebas empíricas para un problema en particular [Goldberg and Hirst, 2017]. Algunas de las más utilizadas son la función Sigmoidea (descrita en la sección 2.1.1), la función Tangente Hiperbólica y la función ReLU (*Rectifier Linear Unit*).

La función Tangente Hiperbólica se define como:

$$tanh(x) = \frac{e^{2x} - 1}{e^{2x} - 1} \tag{2.13}$$

Su gráfica se puede ver en la figura 2.8b y transforma su entrada x en un valor dentro del intervalo $[-1, 1]$.

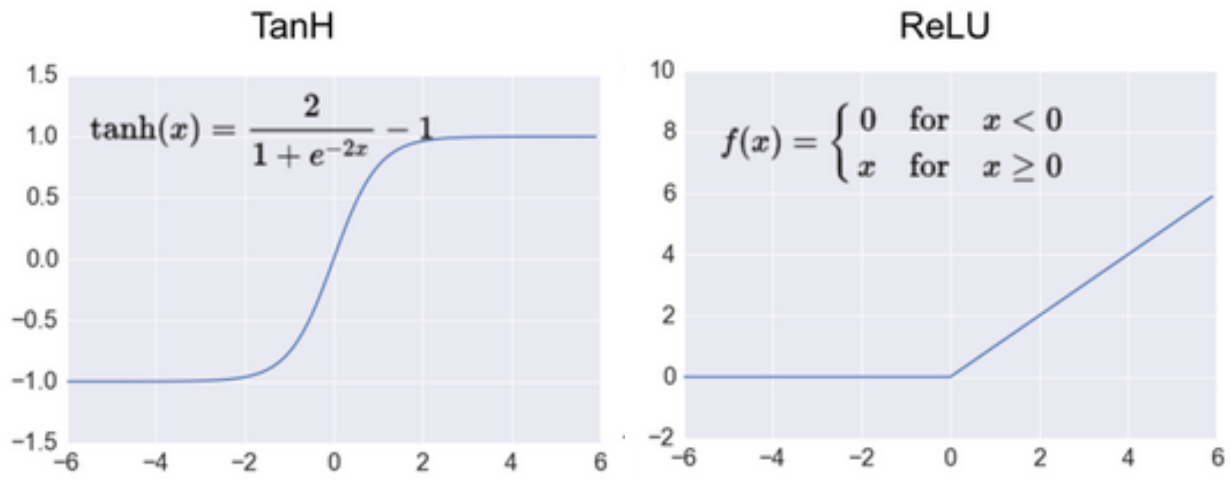
La función ReLU [Glorot et al., 2011] se define de la siguiente manera:

$$ReLU(x) = max(0, x) = \begin{cases} 0 & x < 0 \\ x & casocontrario \end{cases} \tag{2.14}$$

Esta función pesar de ser sencilla ha demostrado lograr un gran desempeño en distintas tareas y tanto esta como la la función Tangente Hiperbólica (cuyas gráficas se observan en la figura 2.8) superan ampliamente el rendimiento de la función Sigmoidea.

2.4.2. Entrenamiento de redes neuronales - *Back-propagation*

El entrenamiento de redes neuronales en función de un conjunto de entrenamiento se lleva a cabo, como ya se mencionó, utilizando el algoritmo de descenso de descenso de gradiente presentado en la sección 2.3. También se utilizan típicamente las mismas funciones de error utilizadas para los modelos lineales descritas en la sección 2.2. La principal diferencia en el entrenamiento de las redes neuronales respecto a los modelos lineales refiere a la forma en que,



(a) (b)

Figura 2.8: Gráfico de las funciones TanH (2.8a) y ReLU (2.8b)

durante la ejecución del algoritmo de descenso de gradiente, se calculan y evalúan los gradientes de cada uno de los parámetros de la red.

El algoritmo conocido como *Back-propagation* [Rumelhart et al., 1986a] permite llevar a cabo este proceso de calcular y evaluar los gradientes respecto a los parámetros de la red de una forma simple y computacionalmente poco costosa, a diferencia de otros métodos analíticos. Queda por fuera del alcance de este trabajo detallar profundamente el funcionamiento de este algoritmo, pero a grandes rasgos, utiliza la regla de la cadena del cálculo (donde si una variable z depende de una variable y que a su vez depende de una variable x, entonces $\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$) para computar sucesivamente los gradientes de las capas de la red en un orden específico que le permite ejecutarse de una forma altamente eficiente [Goodfellow et al., 2016].

2.4.3. Dropout

La técnica conocida como Dropout es una de las formas más conocidas y más usadas de regularización en redes neuronales para prevenir el sobreajuste, discutido en la sección 2.3.1 [Chollet et al., 2018]. Esta, aplicada a una capa de la red, consiste en aleatoriamente durante el entrenamiento "apagar" ciertas neuronas de la capa fijando sus valores de salida en cero. La proporción de neuronas a apagar es un hiperparámetro que se debe definir, pero suelen ser valores comunes un 2 %, 3 % o 5 %. Luego, durante la etapa de evaluación no se apaga ninguna neurona, por lo que los valores de la capa se deben escalar por un factor igual a la proporción de neuronas apagadas durante el entrenamiento para compensar el hecho de que habrá más unidades activas que durante el entrenamiento. Esta técnica evita que las predicciones de la red dependan de los pesos de neuronas específicas. Además, el ruido introducido en la red debido a la aleatoriedad de las neuronas apagadas se traduce en que se descarten patrones

casuales que la red memorizaría en caso de no estar este ruido presente [Chollet et al., 2018] [Goldberg and Hirst, 2017].

2.4.4. Transfer Learning

El uso de modelos neuronales medianamente complejos en casos donde la cantidad de datos con los que se cuenta son escasos suele llevar a que el modelo generalice de forma pobre y sobreajuste sobre los datos disponibles. Para afrontar esta problemática sin recurrir a modelos más sencillos es común el uso de una técnica conocida dentro del campo del aprendizaje profundo como *transfer learning*. Esta consiste en partir de un modelo neuronal ya entrenado sobre un conjunto de datos que no se corresponda exactamente con el del problema original pero si presente un formato y características similares y sea de un tamaño significativamente mayor para luego, congelando algunos de los valores de los pesos de las capas resultantes del entrenamiento o disminuyendo el factor de aprendizaje entrenar de nuevo el modelo esta vez sí con el conjunto de datos correspondiente al problema específico. Esta última etapa de entrenamiento sobre el conjunto de datos específico de mayor tamaño se conoce como *fine-tuning*. Se espera que el entrenamiento sobre el conjunto de datos de mayor tamaño le permita al modelo extraer características más generales que le resulten útiles luego al momento de trabajar con los datos específicos del problema [Zhang et al., 2020]. Esta forma de entrenamiento, además de disminuir el sobreajuste en el entrenamiento de modelos complejos con conjuntos de datos más pequeños permite que, dado que los pesos de la red estarán en valores cercanos a los óptimos, se alcanzará una convergencia más rápida evitando caer en mínimos locales, facilitando el entrenamiento en esta etapa de *fine-tuning*. Es una técnica ampliamente utilizada en distintas áreas del aprendizaje profundo, y en el campo de procesamiento de lenguaje natural especificamente es comunmente utilizada para el entrenamiento de vectores de representación de las palabras conocidos como *word embeddings* sobre los que se profundiza luego en la seccion 3.2.3.

2.5. Redes neuronales recurrentes

Las redes neuronales recurrentes (o RNNs, por sus siglas en inglés) [Rumelhart et al., 1986b] son una familia de redes neuronales especializada en el procesamiento de datos secuenciales. Además, la mayoría de estas redes tienen la capacidad de procesar entradas de longitud variable, a diferencia de las redes *feed-forward* que solo permitían entradas de tamaño fijo [Goodfellow et al., 2016].

Al momento de procesar una entrada x_i que forma parte de una secuencia x_1, x_2, ..., x_n, las RNNs tienen acceso a los valores de las capas ocultas de la red resultantes de haber procesado las entradas anteriores de la secuencia. Los valores de las capas ocultas al procesar una entrada x_i son conocidos como **estado** de la red (típicamente definido como h, en referencia a que es el valor de las capas ocultas o *hidden layers*). El estado de la red se puede definir entonces recursivamente como se muestra en la ecuación 2.15 y así como cualquier función puede ser representada como una red neuronal *feed-forward*, cualquier función que involucre recurrencia

puede ser representada como una RNN [Goodfellow et al., 2016].

$$\boldsymbol{h}^t = f(\boldsymbol{h}^{t-1}, \boldsymbol{x}^t; \Theta) \tag{2.15}$$

Una forma común de representar a las redes neuronales recurrentes se conoce como forma de grafo desplegado. Esta consiste en representar cada elemento de la red varias veces, una vez por cada paso temporal. Entonces los elementos de la red se representarán tantas veces como el largo que tenga la secuencia a procesar. Un ejemplo de una RNN representada de esta forma se puede ver en la figura 2.9.

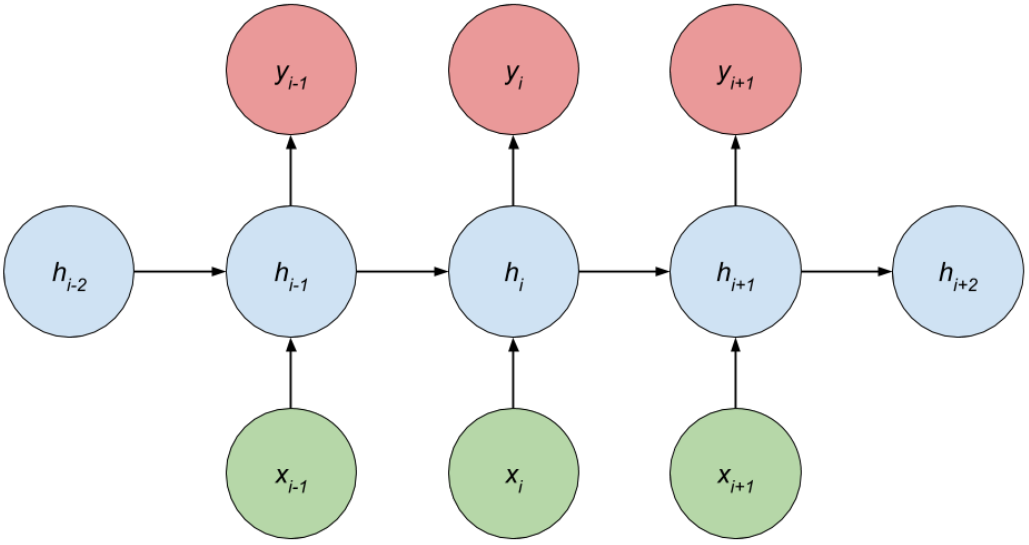


Figura 2.9: Representación de una RNN en forma de grafo desplegado.

El entrenamiento de estas redes neuronales se lleva a cabo de forma similar que en las redes *feed-forward*, aplicando el algoritmo de descenso de gradiente con las funciones de error ya descritas. Sin embargo, el error y los gradientes se calculan sobre el grafo desplegado de la RNN a lo largo del procesamiento de la totalidad de cada secuencia. Esta variante del algoritmo de *back-propagation* se conoce como *back-propagation* a través del tiempo o *back-propagation through time* (BPTT) [Werbos, 1990].

2.5.1. RNNs bidireccionales

Las RNNs presentadas hasta el momento consideraban únicamente información de las entradas pasadas x_1, x_2, ..., x_{n-1} al procesar una entrada x_n, sin embargo, en muchos casos puede resultar necesario tener en cuenta la totalidad de la secuencia. Ejemplos de este tipo de problemas puede ser el reconocimiento del habla o de dígitos escritos, donde la información acerca de fonemas o

dígitos a continuación (tanto como los previos) del que se está procesando puede permitir una mejor clasificación de este por ejemplo sirviendo para desambiguar entre dos posibles términos.

Las Redes Neuronales Recurrentes Bidireccionales se crearon justamente para abordar esta necesidad [Schuster and Paliwal, 1997] y han resultado muy exitosas [Graves, 2012] en aplicaciones donde esta surge, tales como el reconocimiento de escritura a mano ([Graves et al., 2008]), reconocimiento del habla ([Graves and Schmidhuber, 2005]; [Graves et al., 2013]) y bioinformática ([Baldi et al., 1999]).

A grandes rasgos una RNN bidireccional se compone de dos subredes recurrentes, una que procesará la secuencia en orden desde el comienzo y otra que la procesará de forma inversa desde el final de esta. De esta forma es que estos modelos permiten computar una representación que, para un valor x_t de una secuencia, depende tanto de información pasada como futura, pero resulta más sensible a los valores cercanos a dicho valor. Esto sin necesidad de utilizar una ventana de tamaño fijo al rededor de este valor, como resultaría necesario en caso de estar utilizando una red neuronal *feed-forward* [Goodfellow et al., 2016].

2.5.2. Arquitecturas con compuertas

El entrenamiento de RNNs presenta un problema en particular conocido como el problema de desvanecimiento de gradiente (*vanishing gradient*) [Pascanu et al., 2013]. Este consiste en que los gradientes, en los pasos mas tardíos del procesamiento de una secuencia disminuyen rápidamente dentro del proceso de *back-propagation*, lo que causa que muchas veces los valores son demasiado pequeños al momento de alcanzar los valores correspondientes al principio de la secuencia. Esto resulta en que sea dificil para la red captar dependencias de largo plazo dentro de las secuencias [Goldberg and Hirst, 2017].

Considerando a la RNN como un dispositivo de cómputo de propósito general, se puede pensar cada aplicación del modelo lee una nueva entrada x_i y una memoria representada por el estado de la red en ese instante s_{i-1} y opera con ambos de alguna forma que resulta en un nuevo estado s_i. Viéndolo de esta forma, es posible considerar que un problema de las RNN es que no tiene control sobre el acceso a la memoria. En cada paso esta lee y escribe la totalidad de la memoria disponible.

Las llamadas arquitecturas con compuertas (*gated architectures*) son una familia de RNNs que utilizan las llamadas compuertas (*gates*) para controlar el acceso a la memoria. Estas compuertas son, básicamente, un vector binario con el que se realiza un producto punto entre este y la entrada y la memoria, respectivamente. Esto le permite a la red discriminar entre cuales de los valores de la entrada serán tenidos en cuenta y cuales descartados, así como también que valores de la memoria se almacenarán y cuales serán sobrescritos. Más detalladamente, dada una compuerta $\boldsymbol{g} \in 0, 1^n$, una entrada $\boldsymbol{x} \in \mathbb{R}^n$ y una memoria $\boldsymbol{s} \in \mathbb{R}^n$, se define la nueva memoria $\boldsymbol{s'} = \boldsymbol{g} \cdot \boldsymbol{x} + (1 - \boldsymbol{g}) \cdot \boldsymbol{s}$. Entonces, $\boldsymbol{s'}$ almacenará los valores de la entrada que la compuerta permita (en cuya posición haya un 1) sobrescribiendo los valores que hubiera en la memoria en esa posición y descartará los valores cuya posición en la compuerta contenga un 0,

manteniendo el valor almacenado en la memoria.

Los vectores correspondientes a las compuertas pueden ser utilizados como un componente más de la red neuronal que utilice para controlar el acceso a la memoria, es decir, que la red pueda aprender cuando resulta importante retener u olvidar determinada información obtenida en un estado pasado. Para esto, el comportamiento de estas compuertas no puede ser estático, si no que debe poder aprenderse en conjunto con el resto de los parámetros de la red. Se requiere entonces, que estos vectores también sean diferenciables y así poder utilizar correctamente el algoritmo de *back-propagation*. Por esta razón es que en lugar de utilizar vectores binarios para representar las compuertas se utilizan vectores de números reales que luego son pasados a través de una función Sigmoidea. La cercanía al 1 o al 0 de cada valor resultante indicará si deja pasar o no el valor de la entrada que le corresponda.

LSTM

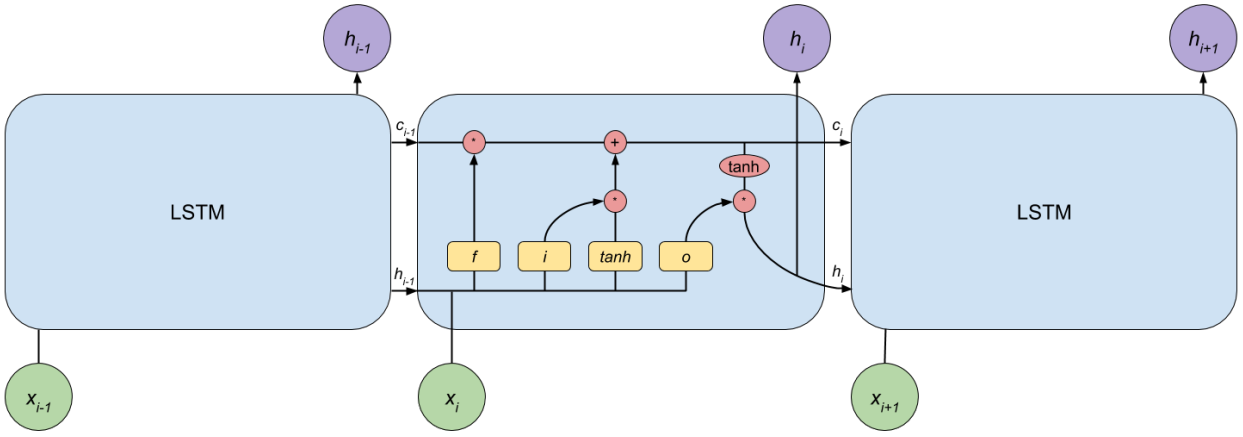


Figura 2.10: Gráfico de una red LSTM en forma de grafo desplegado.

La arquitectura de redes neuronales conocida como *Long Short-Term Memory* (LSTM), introducida por primera vez en [Hochreiter and Schmidhuber, 1997a], es un tipo de RNN con compuertas diseñada específicamente para resolver el problema del desvanecimiento de gradiente. A grandes rasgos, esta arquitectura divide el vector del estado en dos vectores, uno denominado "celdas de memoria" $\boldsymbol{c}$ y otro correspondiente al estado oculto $\boldsymbol{h}$, también conocido como memorioa de trabajo. Las celdas de memoria se utilizan para preservar la memoria y los gradientes a lo largo del tiempo, por lo que son controladas por diferentes compuertas. Estas compuertas son conocidas como de entrada (*input*), de olvido (*forget*) y de salida (*output*) y sus valores se obtienen a través de computar una combinación lineales del vector de entrada actual $\boldsymbol{x_i}$ con el estado oculto anterior $\boldsymbol{h_{i-1}}$ y pasar esta a través de una función Sigmoidea. La compuerta de olvido f determina cuanto de la memoria previa se debe mantener $(\boldsymbol{f} \cdot \boldsymbol{c_{i-1}})$ mientras que la compuerta de entrada determina cuanto de cada nueva entrada debe ser tenido en cuenta $(\boldsymbol{i} \cdot tanh(\boldsymbol{x_j}\boldsymbol{W}^x + \boldsymbol{h_{j-1}}\boldsymbol{W}^x))$. Finalmente, la compuerta de salida determina que

valores de las mencionadas "celdas de memoria" deben ser tenidas en cuenta como memoria de trabajo para el procesamiento de la siguiente entrada ($\boldsymbol{h_j} = \boldsymbol{o} \cdot tanh(\boldsymbol{c_j})$). Una visualización gráfica de la arquitectura descrita se puede observar en la figura 2.10.

Las redes LSTM son hoy en día la forma más exitosa de arquitectura de redes neuronales recurrentes [Goldberg and Hirst, 2017]. Han demostrado poseer la capacidad de aprender dependencias a largo plazo más fácilmente que otras arquitecturas, en principio sobre conjuntos de datos diseñados específicamente para este tipo de tareas ([Bengio et al., 1994], [Hochreiter and Schmidhuber, 1997b], [Hochreiter et al., 2001]) y luego también en otros desafiantes problemas de procesamiento de secuencias en los que lograron un desempeño que alcanzó el estado del arte ([Graves, 2012], [Graves et al., 2013], [Sutskever et al., 2014]).

Capítulo 3

Procesamiento de Lenguaje Natural

El procesamiento de lenguaje natural (NLP, por sus siglas en inglés) es una rama de las Ciencias de la Computación y de la Inteligencia Artificial que cubre un conjunto de técnicas para analizar y representar textos que ocurran naturalmente a uno o más niveles lingüísticos con el propósito de alcanzar escalas de procesamiento similares al humano para un rango de tareas o aplicaciones [Liddy, 2001]. Es posible desarrollar con mayor profundidad algunos de los puntos de esta definición. Por "textos que ocurran naturalmente" podemos referirnos a texto de cualquier idioma, género o modo con la condición de que sea utilizado por humanos para comunicarse entre sí y no generado específicamente para el propósito del análisis. Además, considerando que la definición refiere como objetivo del NLP a un rango de tareas o aplicaciones señala que este no se lo considera un fin en si mismo, si no el medio para concretar otros fines específicos, como lo pueden ser la traducción de textos, generación de resúmenes, respuesta a preguntas, interacción a través de diálogos o, el que será el eje de este trabajo, la generación de lenguaje natural, entre otros [Liddy, 2001].

Los primeros trabajos en NLP surgieron en los años 50 con acercamientos desde la inteligencia artificial simbólica o las reglas definidas a mano (*hand-written rules*). Estos enfoques si bien resultaban útiles para un análisis sintáctico de los textos, se mostraron insuficientes para tratar con el significado, es decir, la semántica de estos y para lidiar con características propias del lenguaje natural, como las ambigüedades o los textos cuya prosa no respeta necesariamente las reglas sintácticas del idioma, pero aún así resultan comprensibles para el lector humano [Nadkarni et al., 2011]. Por otro lado, se ha acrecentado la disponibilidad cada vez mayor de grandes cantidades de texto electrónico debido en gran parte al surgimiento de Internet que es, sin dudas, la principal fuente de este texto hoy en día [Bird et al., 2009]. Estas dos razones fueron las principales impulsoras de que desde los años 80 en adelante haya habido un giro cada vez más pronunciado al uso de métodos estadísticos para NLP entre los que se encuentran las diferentes formas de aprendizaje automático [Liddy, 2001][Nadkarni et al., 2011].

El proceso de descubrimiento de conocimiento (KDD por sus siglas en inglés) a partir de textos, en el cual se pueden enmarcar varios de los procesos descritos en las secciones siguientes, se

tomó como referencia para ordenar los contenidos de este capítulo. Dicho proceso consta, en primer lugar, de una etapa de Preparación de los datos, en la cual a partir de los datos iniciales se obtiene una vista minable de estos. Luego, una segunda etapa de Minería de datos donde se detectan los patrones que resulten relevantes en la vista minable obtenida y por último una etapa de evaluación, interpretación o visualización de los datos obtenidos. En la figura 3.1 se ilustran las etapas mencionadas y se listan algunos de los procesos desarrollados en este trabajo que se pueden enmarcar en sus etapas.

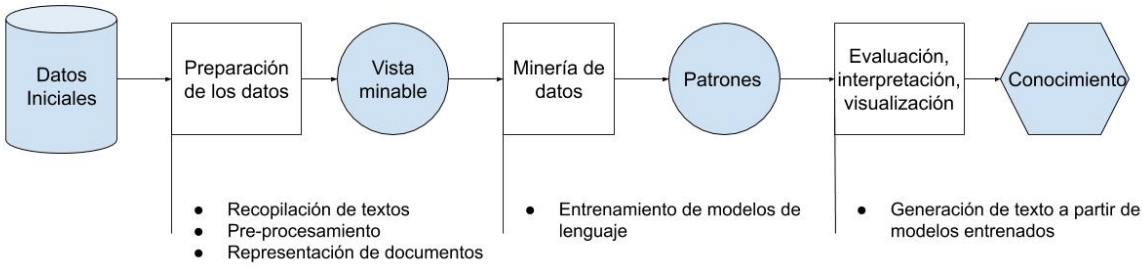


Figura 3.1: Etapas del proceso de descubrimiento de conocimiento (KDD).

En este capítulo se desarrollan algunas de las tareas principales específicas del campo de NLP dentro de un proceso de KDD y las técnicas más utilizadas para llevarlas a cabo, junto con algunos puntos específicos del área que conciernen a este trabajo en particular. En la sección 3.1 se tratan técnicas de preprocesamiento de texto como la tokenización (subsección 3.1.1) o el algoritmo de Byte Pair Encoding (subsección 3.1.2). Luego, en la sección 3.2 se describen distintos acercamientos para la representación de textos, como lo son la representación por atributos estáticos (subsección 3.2.1), dinámicos (subsección 3.2.2) o aprendidos (subsección 3.2.3) y el caso especifico de los *Word Embeddings* y como se lleva a cabo su entrenamiento en la subsección 3.2.4. En la sección 3.3 se define Modelo de Lenguaje, se detallan algunos de los más utilizados, como lo son los modelos de n-gramas, los modelos neuronales de ventana fija y los modelos neuronales recurrentes (subsecciones 3.3.1, 3.3.2 y 3.3.3) y se ilustra su uso para la generación de lenguaje natural. Por último, en la sección 3.4 se profundiza acerca del problema de la generación de texto estructurado y en la subsección 3.4.1 se describe el dominio específico de este trabajo: las Batallas de Freestyle y sus características particulares para trabajar la generación de texto sobre este.

3.1. Preprocesamiento

Se agrupan en la etapa de preprocesamiento del texto a las tareas encargadas de convertir el texto a un formato que resulte analizable para un propósito específico. La elección de una combinación correcta de estas tareas puede resultar en una mejora significativa en el resultado de un trabajo de NLP [Uysal and Gunal, 2014].

En esta sección se desarrolla una de las principales tareas de preprocesamiento, necesaria en todo trabajo de NLP, que es la tokenización o partición del texto (subsección 3.1.1) y luego un algoritmo que permite llevar a cabo una forma específica de tokenización conocido como Byte Pair Encoding (subsección 3.1.2).

Es posible nombrar también, como tareas que pertenecen a esta etapa, el filtrado de caracteres o de palabras, como lo pueden ser la remoción de signos de puntuación o de palabras en función de una baja frecuencia, la normalización (por ejemplo, la conversión de letras mayúsculas a minúsculas), la lematización, es decir, la conversión de palabras a sus formas básicas (como lo puede ser la conversión de verbos a su infinitivo) o el truncado de palabras. Además, suelen resultar necesarias tareas de limpieza sobre el texto tales como la detección y remoción de errores ortográficos o de caracteres inválidos.

3.1.1. Tokenización

La tokenización o partición es un proceso que separa un texto crudo en mínimas unidades lingüísticas identificables, llamadas *tokens*. Dichos tokens son considerados entonces unidades indivisibles a la hora de su procesamiento. El principal desafío en lo que respecta a tokenización es encontrar una forma de separación que permita extraer información útil a partir de cada token [Jurafsky and Martin, 2018]. A continuación, en las subsecciones siguientes, se describen los tres posibles enfoques de tokenización: a nivel de palabras, de caracteres y de subpalabras.

Tokenización a nivel de palabras

La tokenización a nivel de palabras es uno de los enfoques más típicos y sencillos de implementar. Este consiste en utilizar como separadores el caracter de espacio en blanco y los signos de puntuación. Este puede computarse a través de algoritmos determinísticos basados en expresiones regulares que se compilan en autómatas de estado finito muy eficientes [Jurafsky and Martin, 2018].

Sin embargo, este enfoque trae consigo ciertos problemas que hacen que su utilización muchas veces no sea la óptima:

- La tokenización a nivel de palabras en la mayoría de los casos implica que el tamaño del vocabulario, es decir, la cantidad total de tokens utilizados, será significativamente grande si se pretende incluir todas las distintas palabras que se encuentren en la base de datos. Esto puede conllevar un uso más deficiente de la memoria y una baja de la *performance* de los modelos que trabajen sobre este tipo de tokenización.

  Por ejemplo, modelos del estado del arte que utilizan este enfoque como Transformer XL [Dai et al., 2019] cuenta con un vocabulario de 267.735 tokens.

- Al entrenar un modelo a partir de un texto que fue tokenizado a nivel de palabras, en principio dicho modelo no será capaz de procesar nuevos ejemplos con palabras que no figuren en su vocabulario. Para tratar este problema se suelen incorporar al diccionario

del modelo un token que representa una palabra desconocida, pero esto se traduce en que no le será posible obtener información de dichas palabras.

- Otro problema que surge de la tokenización a nivel de palabras es la división de elementos los cuales resultaría útil que fueran un único token. Este es el caso de nombres propios, como lo puede ser "Buenos Aires", que nos brinda mucha más información acerca de a que se refiere que los tokens "buenos" y "aires" por separado.
- Por último, en este tipo de tokenización los errores de escritura toman aún mas relevancia: un token "eror" no guardará en principio ningún tipo de relación con el token "error", por lo que los tokens contengan este tipo de errores resultan inútiles para un modelo entrenado a partir de texto tokenizado de esta manera.

Tokenización a nivel de caracteres

En busca de solucionar varios de los problemas mencionados anteriormente es que se presenta la tokenización a nivel de caracteres. Esta forma de tokenización inmediatamente resulta en un vocabulario mucho más reducido: por ejemplo basta con 27 tokens para representar los caracteres del español, más los utilizados para representar caracteres especiales, aunque es más común encontrar este tipo de tokenización en idiomas como el chino, donde un caracter es una unidad semántica razonable [Jurafsky and Martin, 2018]. Esta forma de tokenización presenta la ventaja de que un modelo entrenado a nivel de caracteres será capaz de procesar cualquier nueva entrada, ya que no encontrará ningún token que no pertenezca al vocabulario definido originalmente. Sin embargo, retomando el caso de idiomas como el español o el inglés, trabajar con este tipo de tokenización necesariamente implica la necesidad de trabajar con secuencias de tokens más largas para obtener información útil: una secuencia de cuatro palabras puede resultar suficiente para obtener una buena estimación de que palabra es la siguiente, mientras que una secuencia de cuatro caracteres puede ser incluso insuficiente para inferir de que palabra (o palabras) forman parte. Si bien se han logrado resultados muy interesantes utilizando este enfoque, como pueden verse en [Radford et al., 2017] o [Kalchbrenner et al., 2016], este no permite alcanzar uno de los objetivos iniciales de la tokenización, que es la de obtener unidades con valor semántico.

Tokenización a nivel de subpalabras

Por último, como una forma intermedia entre la tokenización a nivel de palabras y la tokenización a nivel de caracteres se encuentra la **tokenización a nivel de subpalabras**. Este tipo de tokenización genera tokens únicos para las palabras más comunes, y divide las palabras menos comunes en subpalabras con cierto valor semántico. Por ejemplo, palabras como "invertebrado" pueden descomponerse en los tokens "in", "vertebr" y "ado", a partir de los cuales es posible obtener una noción del significado de la palabra. Esta forma de tokenización es hoy en día la principal utilizada en los modelos del estado del arte en lo que respecta a generación de lenguaje natural, como GPT-3 [Brown et al., 2020] o BERT [Devlin et al., 2018] y algunos de los algoritmos mas utilizados para lograrlo son WordPiece, Unigram LM, o Byte Pair Encoding,

el utilizado en este trabajo y que será desarrollado a continuación en la subsección 3.1.2, entre otros.

3.1.2. Byte Pair Encoding

El algoritmo conocido como Byte Pair Encoding (se traduce como Codificación de a Pares de Bytes) es un algoritmo de compresión de datos descrito originalmente en [Gage, 1994]. Si bien este no era su propósito original, su uso para la tokenización de textos en tareas de NLP se encuentra ampliamente extendido y desde que se lo propuso para este tipo de tareas en [Sennrich et al., 2015] se ha convertido, junto con la tokenización a nivel de subpalabras, en la norma en los modelos del estado del arte en lo que respecta a NLP, como los ya mencionados GPT-3 o BERT.

El algoritmo consiste en una primera etapa de inicialización del vocabulario donde cada palabra se *tokeniza* a nivel de caracteres y se le adjunta un token especial que indica el final de la palabra (típicamente denominado '</w>'). Luego, se genera un vocabulario en forma de diccionario, donde a cada palabra *tokenizada* se le asocia su cantidad de ocurrencias en el texto. Un ejemplo muy simplificado de posible diccionario generado de la forma descrita es:

{"c a n t a r </w>": 4, "a r b o l </w>": 3, "v o l a r </w>": 3}

En el caso de este ejemplo, el diccionario se obtuvo a partir de un texto que contiene la palabra "cantar" cuatro veces y las palabras "arbol" y "volar" tres veces. El algoritmo continúa entonces de forma iterativa, donde en cada iteración se cuenta la frecuencia de cada par de tokens consecutivos, y se los une en un solo token. En el caso del ejemplo, con $4 + 3 + 3 = 10$ apariciones, el par mas frecuente es el compuesto por los tokens 'a' y 'r', por lo que estos se unen en un nuevo token 'ar' y el diccionario queda con la siguiente forma:

{"c a n t ar </w>": 4, "ar b o l </w>": 3, "v o l ar </w>": 3}

En la iteración siguiente se encuentra como par mas frecuente a los tokens 'ar' y '</w>', por lo que se concatenan en el token 'ar</w>'. En este ejemplo podemos ver la necesidad de la utilización de un token que indique el final de una palabra, ya que el token 'ar</w>' es probable que nos indique la presencia de un verbo en infinitivo, que difiere mucho de lo que podemos inferir de la presencia del token 'ar' en "arbol". Entonces, tras unir los tokens mencionados, el diccionario pasa a tener la forma:

{"c a n t ar</w>": 4, "ar b o l </w>": 3, "v o l ar</w>": 3}

Al llegar a la sexta iteración, se formaron los tokens "cant" y "ol" y el diccionario presenta la siguiente forma:

{"cant ar</w>": 4, "ar b ol </w>": 3, "v ol ar</w>": 3}

Esta forma ya puede ser considerada una tokenización a nivel de subpalabras útil para un problema determinado, aunque lógicamente una base de datos del mundo real presentará una mayor cantidad de palabras y de apariciones de cada una que la presentada en este ejemplo. También

es posible observar que luego de doce iteraciones cada palabra ya estará representada por un token único (el diccionario estará compuesto por los tokens "cantar</w>", "arbol</w>" y "volar</w>"), por lo que no será posible realizar otra unión de tokens.

La cantidad de total iteraciones es posible definirla utilizando un valor fijo o teniendo en cuenta la cantidad de tokens de los que se desea que esté compuesto el vocabulario. Respecto a dicha cantidad, luego de cada iteración hay tres escenarios posibles: el número de tokens puede decrementarse en uno como en la primera iteración del ejemplo, puede aumentar en uno como en la segunda iteración del mismo, o mantenerse igual. En la práctica, a medida que la cantidad de iteraciones se incrementa típicamente la cantidad de tokens se incrementa en un principio, para luego decrementarse.

3.2. Representación de documentos

La extracción de características es una etapa importante para todo problema de aprendizaje automático. Esta tiene una influencia directa en los resultados que se obtengan, más allá del modelo que se utilice. En lo que refiere a NLP, esta extracción de características consiste en convertir texto crudo o legible a una forma numérica adecuada para un modelo de aprendizaje automático y es conocida como Representación de Texto [Vajjala et al., 2020].

Es posible clasificar las distintas formas de representación de texto en tres categorías: las que utilizan características estáticas, las que utilizan características dinámicas o las que utilizan características aprendidas. En esta sección se describen cada una de estas categorías y se presentan ejemplos de formas de representación que se encuentren dentro de cada una [Jurafsky and Martin, 2018].

Resulta necesario mencionar que los ejemplos de extracción de características y formas de representación mencionados a lo largo de esta sección hablan principalmente de palabras cuando en realidad aplican a tokens (que pueden ser tanto palabras como subpalabras o caracteres). Esto se debe a que estos métodos de representación son, en muchos casos, previos a que se popularice la utilización de formas tokenización que no sean en palabras, pero aplican igualmente a los otros niveles de tokenización ya descritos en la sección 3.1.1.

3.2.1. Representación con Características Estáticas

La representación de texto con características estáticas consiste en utilizar características elegidas previamente al procesamiento de los documentos, típicamente por un grupo de expertos en el dominio del problema específico para representar el documento en función de dichas características, también llamadas *features*. Ejemplos de estas características son la cantidad de palabras, longitud promedio de palabras, cantidad promedio de cierto tipo de palabras o características propias de un dominio, como puede serlo la cantidad de citas por párrafo en un texto académico, entre otras.

Este tipo de representaciones puede resultar útil en problemas de clasificación de texto donde se

cuenta con información experta acerca de que características resultan relevantes para clasificar los textos correctamente. Ejemplos de trabajos donde se utilizó este enfoque se pueden ver en [Ferretti et al., 2018] o [Hassen et al., 2017]. En el primero se busca identificar artículos de Wikipedia de calidad deficiente utilizando características estáticas, tales como la cantidad de texto sin referencias, mientras que en el segundo se busca clasificar software malicioso en función de la frecuencia de ciertos códigos de operación o mnemónicos.

3.2.2. Representación con Características Dinámicas

Las representaciones con características dinámicas, son las que utilizan características que surgen como parte del procesamiento de los datos. Típicamente, para estas formas de representación se genera en primer lugar un vocabulario que contenga la totalidad de tokens del texto y luego se lo representa en función de dicho vocabulario [Jurafsky and Martin, 2018].

Un ejemplo de una representación de este tipo es la llamada Bolsa de Palabras (o BOW, por sus siglas en inglés), para la cual en primer lugar, como ya se mencionó, se genera un vocabulario a partir de los textos y luego cada texto se representará como una tabla donde se indica, para cada palabra del vocabulario, si está presente en el texto y opcionalmente su cantidad de apariciones. Esta forma de representación no tiene en cuenta el orden en el que están dispuestas las palabras en cada texto y es común encontrarla en problemas de clasificación y mas específicamente de *sentiment analysis* (análisis del sentimiento) de un texto [Jurafsky and Martin, 2018], como es el caso de identificar si una reseña de un producto es positiva o negativa. Un ejemplo de la aplicación de este tipo de representación sobre reseñas de un producto se ilustra en la figura 3.2.

"Muy bueno, cumple con lo esperado."
"Funciona bien y el precio es bueno."
"Desastre. Muy caro y funciona mal."

	a	bien	bueno	caro	con	cumple	desastre	es	el	esperado	funciona	lo	mal	muy	precio	y
d1	0	0	1	0	1	1	0	0	0	1	0	1	0	1	0	0
d2	0	1	1	0	0	0	0	1	1	0	0	0	0	0	1	1
d3	0	0	0	1	0	0	1	0	0	0	1	0	1	1	0	1

Figura 3.2: Ejemplo de la representación en forma de bolsa de palabras de tres textos que corresponden a reseñas de un producto.

Otras formas de representación de un texto utilizando características dinámicas incluyen la indexación del vocabulario total del texto y la posterior representación de cada palabra como su correspondiente índice en el vocabulario. Entonces, por ejemplo, si se quisiera representar la frase "Hay un perro y un gato en casa" de esta manera, , el vocabulario será el conjunto $V = \{Hay, un, perro, y, gato, en, casa\}$, y a cada palabra le corresponderá un índice $1 \leq i \leq |V| = 7$. La frase entonces queda representada por la lista $[1, 2, 3, 4, 2, 5, 6, 7]$.

La representación utilizando índices suele no ser óptima para modelos de texto, ya que puede llevar a inferencias equivocadas como, en el caso del ejemplo, que las palabras "y" y "perro"

tienen más en común que las palabras "perro" y "gato", por estar sus índices más próximos, lo que no es necesariamente cierto. También es posible que un modelo que buscara predecir la siguiente palabra en un texto, ante una posibilidad muy alta de que la palabra fuera "perro" o "gato", optara por la palabra "y", por ser su índice (4) un intermedio de los anteriores (3 y 5).

Como solución a los problemas mencionados es común para la representación de texto la llamada codificación *one-hot*. Esta consiste en representar cada palabra como un vector del tamaño del vocabulario que contenga un 1 en la posición correspondiente al índice de la palabra y un 0 en el resto de las posiciones del vector. Si se quisiera codificar la frase del ejemplo anterior utilizando dicha representación, entonces las palabras se deben representar de la siguiente manera:

$Hay = [1, 0, 0, 0, 0, 0, 0]$, $un = [0, 1, 0, 0, 0, 0, 0]$, $perro = [0, 0, 1, 0, 0, 0, 0]$, $y = [0, 0, 0, 1, 0, 0, 0]$, $gato = [0, 0, 0, 0, 1, 0, 0]$, $en = [0, 0, 0, 0, 0, 1, 0]$, $casa = [0, 0, 0, 0, 0, 0, 1]$.

Lógicamente, un texto representado de esta forma requerirá de más espacio de almacenamiento que si se utilizaran solo los índices de las palabras, lo que puede resultar un problema cuando se trabaja con vocabularios de gran tamaño, pero se eliminan los problemas surgidos al utilizar la codificación anterior. Es más, resulta importante notar que representación *one-hot* no guarda ningún tipo de valor semántico de las palabras representadas [Goldberg and Hirst, 2017]. En el ejemplo mostrado, "perro" y "gato" son tan distintos como "perro" y "casa", lo que no necesariamente se corresponde con el significado de la palabra.

3.2.3. Representación con Características Aprendidas: *Word Embeddings*

Por último, las llamadas representaciones aprendidas presentan hoy en día uno de los mayores saltos en lo que refiere a la representación de palabras para modelos más recientes de aprendizaje profundo en el NLP [Goldberg and Hirst, 2017]. Esta idea consiste en extender el uso del aprendizaje automático, usualmente utilizado en la etapa de minería de datos, a la etapa de representación, para aprender a representar las palabras en función de sus contextos. La forma principal de este tipo de representación de características son los llamados *Word Embeddings*. En esta subsección se desarrolla el concepto de *embeddings* y cuales son las ventajas que brinda este tipo de representación asumiendo que ya se cuenta con vectores de *embeddings* adecuados, mientras que en la subseccción 3.2.4 se describe uno de los métodos utilizados para obtener dichos vectores.

La utilización de *Word Embeddings* consiste en representar a cada palabra como un vector denso de longitud fija de números reales y donde palabras con significados similares tengan representaciones similares, lo que se logra a partir de conocer el contexto de cada una, es decir, las palabras que aparecen frecuentemente junto a ella [Goldberg and Hirst, 2017].

Esta forma de representación presenta muchas ventajas respecto a enfoques anteriores, como la representación *one-hot*. Las primeras son computacionales: para obtener una representación *one-hot* de una palabra la dimensión del vector de representación será igual al tamaño total de vocabulario, que es un valor significativamente grande en general. A diferencia de esto, el

tamaño del respectivo vector de *embedding* de una palabra es un valor arbitrario fijo, que si bien su tamaño ideal es aún un problema abierto, es generalmente mucho menor al tamaño del vocabulario (300 es uno de los valores más comunes utilizado como dimensión de los vectores de *embeddings* en varios estudios) [Yin and Shen, 2018]. Además, las redes neuronales suelen no funcionar tan bien con vectores dispersos de muy alta dimensión [Goldberg and Hirst, 2017].

Luego, como se vio en el ejemplo de la representación *one-hot*, sabemos que esta no permite obtener ninguna información semántica de las palabras representadas: dados dos vectores *one-hot* que representen dos palabras distintas, no es posible conocer como estas se encuentran relacionadas. A diferencia de esto, la representación a través de *embeddings* permite conocer que tan cercanas semánticamente son dos palabras a través de la llamada Similitud de Coseno. Utilizando *embeddings* bidimensionales, esto se puede interpretar gráficamente observando que el grado del ángulo entre los vectores de las palabras con significados similares será cercano a 0, mientras que será mayor para palabras cuya semántica no esté relacionada. Un posible ejemplo de esto se ve en la figura 3.3. Resulta importante mencionar que si bien cuando se trata con *embeddings* de mayor dimensionalidad se pierde la capacidad de graficarlos, la Similitud de Coseno sigue funcionando de la misma manera.

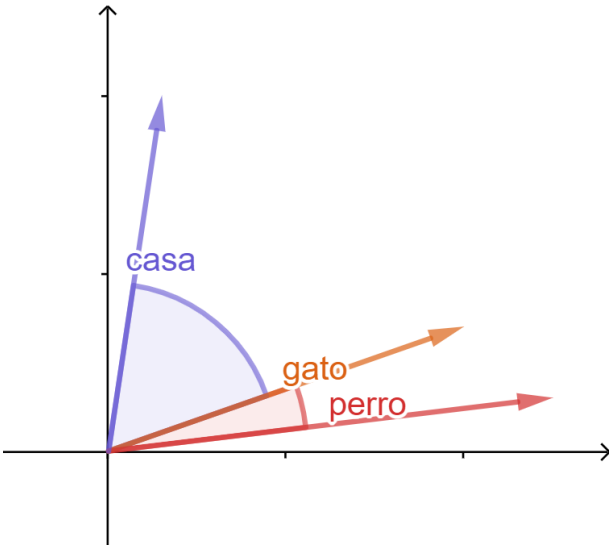


Figura 3.3: Ejemplo de posibles vectores que representan las palabras “perro”, “gato” y “casa” donde se puede ver como el ángulo entre las primeras dos es mucho menor que entre ellas y “casa”. Los valores para este ejemplo son meramente ilustrativos, no fueron obtenidos a partir de un algoritmo para calcular *embeddings*.

3.2.4. Obtención de vectores de *Embeddings*

En esta subsección se describe el proceso de obtención de los vectores de *embeddings* adecuados para la representación de cada palabra (o token) de un vocabulario a partir de un texto presentado por primera vez en [Mikolov et al., 2013]. Este proceso consta, a grandes razgos, de

dos etapas: en la primera etapa se conforma un conjunto de datos adecuado a partir del texto, para lo cual se utilizan los conceptos de Ventana Deslizante, *Skipgram* y Muestreo Negativo, descritos a continuación. Luego, en la segunda etapa, se aprenden efectivamente los vectores de *embeddings* a partir del conjunto de datos generado.

Ventana deslizante

Como ya se mencionó, los *embeddings* buscan captar información acerca del contexto de una palabra. Para generar los datos que puedan captar ese contexto y a partir de los cuales se realizará el entrenamiento, se ejecuta sobre el texto lo que se conoce como un algoritmo de **ventana deslizante** de tamaño N. Este define que para formar el conjunto de datos se debe iterar sobre el texto tomando inicialmente las palabras $w_0, w_1, ..., w_{n-1}$, desplazando la ventana una posición, y tomando luego las palabras $w_1, w_2, ..., w_n$ hasta haber procesado el texto por completo. Típicamente, estos datos se usan para entrenar modelos de lenguaje, es decir modelos que computan la probabilidad de ocurrencia de una enésima palabra en función de las $N-1$ palabras anteriores (en la sección 3.3 se desarrolla con mayor profundidad acerca de modelos de lenguaje). Por esto es que las cadenas obtenidas a partir del uso de una ventana deslizante se suelen separar en pares que contienen en su primera posición las primeras $N-1$ palabras de la cadena que se utilizan como entrada para el modelo y en la segunda la enésima palabra, que representa el objetivo a predecir por dicho modelo. Un ejemplo de un conjunto de datos generado a través de este algoritmo sobre un texto se puede observar en la figura 3.4.

Texto de entrada: “Muchas gracias por todo, yo improviso de este modo”

Ventana deslizante ejecutándose sobre el texto

muchas	gracias	por	todo	yo	improviso	de	...
muchas	gracias	por	todo	yo	improviso	de	...
muchas	gracias	por	todo	yo	improviso	de	...
muchas	gracias	por	todo	yo	improviso	de	...
muchas	gracias	por	todo	yo	improviso	de	...

Conjunto de datos generado

entrada 1	entrada 2	objetivo
muchas	gracias	por
gracias	por	todo
por	todo	yo
todo	yo	improviso
yo	improviso	de

Figura 3.4: Conjunto de datos generado a partir de aplicar una ventana deslizante de tamaño 3 sobre un texto dado.

Es interesante notar que el tamaño de ventana es un hiperparámetro que influye en lo que finalmente representará a los *embeddings* entrenados. Tamaños de ventana pequeños, de entre 2 o 15 palabras, suelen captar similitudes funcionales de las palabras. Es decir, llevan a

que las palabras cuyos vectores sean más similares sean las se puedan utilizar en los mismos contextos o que se puedan considerar intercambiables teniendo en cuenta las palabras que las rodean [Goldberg and Hirst, 2017]. Ejemplo de esto suelen ser los antónimos, como las palabras "bueno" y "malo" que suelen estar rodeadas en lo inmediato por las mismas palabras: "El plato estaba muy bueno/malo", "Siempre fui bueno/malo para los deportes".

Skipgram

Como se mencionó en la subsección anterior, los conjuntos de datos utilizados para el entrenamiento de modelos de lenguaje consisten de pares que relacionan las palabras $N-1$, $N-2$, ..., con la enésima palabra siguiente. Sin embargo, la arquitectura conocida como **skipgram**, utilizada para el entrenamiento de *embeddings*, define una estructura distinta para el conjunto de datos generado. En primer lugar, se utiliza una ventana deslizante de tamaño N pero con la particularidad de que además de tener en cuenta las $N-1$ palabras previas como entrada para la predicción de la enésima palabra, se contemplan también las $N-1$ palabras que se encuentren a continuación de esta. Luego, a diferencia del formato anterior de los datos, donde se relacionaban todas las palabras anteriores con la palabra que se busca predecir, ahora se relacionará a esta con cada una de ellas por separado. Un ejemplo de su aplicación se puede ver en la figura 3.5.

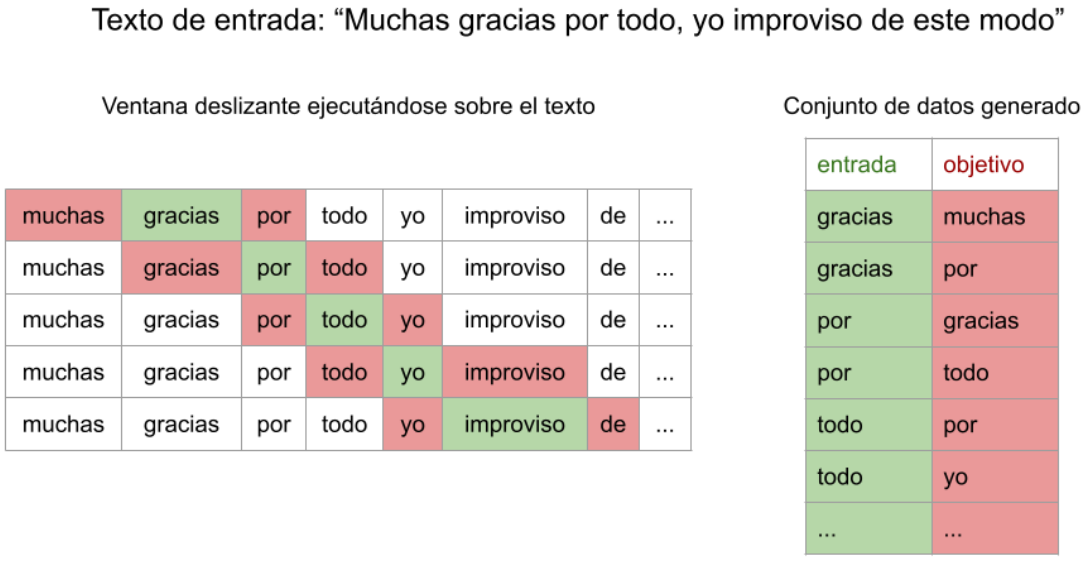


Figura 3.5: Conjunto de datos generado a partir de aplicar una ventana deslizante de tamaño 2 sobre un texto dado utilizando la arquitectura skipgram.

Una última modificación en la estructura del conjunto de datos se da a razón de convertir el problema a la forma de un problema de clasificación binaria, lo que resulta en que sea mas

simple y rápido de calcular. Para esto, en lugar de modelar los datos con la forma de una palabra como entrada y otra como salida, se estructuran en la forma de pares de palabras, a las que les corresponderá un 1 como salida en caso de estar relacionadas o un 0 en caso de no estarlo. La figura 3.6 ilustra en un ejemplo la aplicación de la modificación descrita a un conjunto de datos dado.

Conjunto de datos en forma de pares de palabras

entrada	objetivo
gracias	muchas
gracias	por
por	gracias
por	todo
todo	por
todo	yo
...	...

Conjunto de datos en forma de clasificación binaria

entrada	entrada	objetivo
gracias	muchas	1
gracias	por	1
por	gracias	1
por	todo	1
todo	por	1
todo	yo	1
...	...	...

Figura 3.6: Ejemplo de conjunto de datos en formato de pares de palabras y el respectivo conjunto en formato de clasificación binaria.

Muestreo negativo

Resulta necesario notar que a partir de los procesos descritos en las subsecciones anteriores se obtuvo un conjunto que contiene únicamente pares de palabras relacionadas entre sí. Por esto es que es probable que un modelo entrenado sobre ese conjunto aprenda a inferir que para cualquier par de palabras que reciba como entrada, estas se encuentran relacionadas, lo que lógicamente es un error.

Una solución a este problema es la de utilizar el llamado **muestreo negativo**, que consiste en incluir en el conjunto de datos muestras de palabras que no se encuentren relacionadas. Para llevar esto a cabo es posible tomar cada ejemplo del conjunto de datos y generar nuevos donde se relacione la primer palabra del par con palabras aleatorias del vocabulario con las que no se encuentren previamente relacionadas, a los que les corresponderá un 0 como salida. Esta idea se basó en lo propuesto en [Gutmann and Hyvärinen, 2010] y en conjunto con los procesos mencionados en las subsecciones anteriores se conocen como **Skipgram con Muestreo Negativo** (Skipgram with Negative Sampling o SGNS). En la figura 3.7 se ilustra un ejemplo de

un conjunto de datos obtenido a partir de un texto utilizando SGNS.

Figura 3.7: Conjunto de datos generado utilizando SGNS sobre un texto.

Aprendizaje de *embeddings*

Una vez definida la estructura del conjunto de datos a utilizar, se procede a desarrollar el proceso de entrenamiento de los *embeddings* a partir de este. En primer lugar es necesario definir un tamaño de vocabulario V, es decir, la cantidad de palabras (más específicamente tokens) para las que se busca entrenan vectores de *embeddings*, y el tamaño E para dichos vectores (como se mencionó anteriormente, un valor común para esto es de 300, pero es posible considerar otros valores). Se define entonces un modelo neuronal que buscará predecir para un par de palabras si estas se encuentran o no relacionadas y que contará con dos matrices: una de *embeddings* y otra llamada de Contexto, ambas de tamaño VxE. A través del entrenamiento de dicho modelo se buscará ajustar la matriz de *embeddings* para que sirva como una representación adecuada.

El entrenamiento es un proceso iterativo donde para cada iteración se tienen en cuenta del conjunto de datos un ejemplo positivo de una palabra de entrada X y sus respectivos ejemplos negativos. Es decir que se toman un par de palabras X, Y que se encuentren relacionadas en el conjunto de datos, y una cantidad N (que depende de la cantidad de muestras negativas por muestra positiva que se hayan insertado) de pares de palabras X, W, X, Z ..., que no estén relacionados (les corresponderá un 0 en el conjunto de datos).

La predicción del modelo respecto a si las palabras de cada uno de los pares tomados están relacionadas se corresponde a la aplicación de una función sigmoidea sobre el producto punto

entre el vector de *embeddings* de la palabra de entrada X $(E(X))$, con los vectores de contexto de cada una de las palabras de los pares tomados respectivamente $(C(w_1), C(w_2), ..., C(w_n))$. Se considerará el error del modelo a la diferencia entre el resultado esperado para cada par de palabras (1 para el ejemplo positivo o 0 para los negativos) y el resultado de su predicción para ese par. Dicho error se buscará minimizar actualizando los parámetros del modelo, es decir, los valores de la matriz de *embeddings*. Una iteración de este proceso de entrenamiento se ilustra en la figura 3.8.

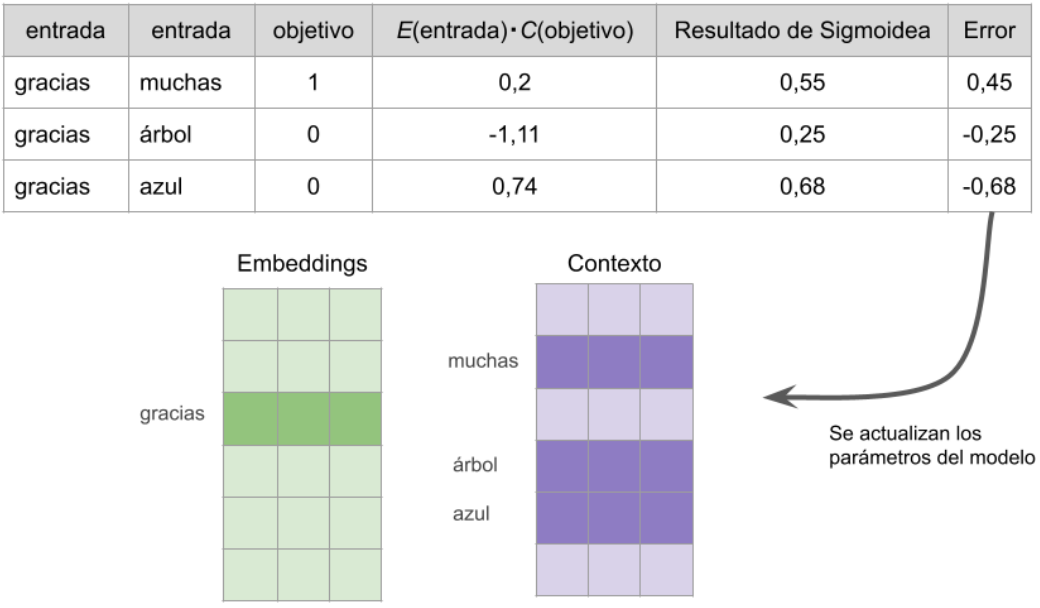


Figura 3.8: Iteración del proceso de entrenamiento descrito.

Una vez avanzado el entrenamiento del modelo y al alcanzar un error total mínimo (o lo necesariamente bajo) se descarta la matriz de contexto y se espera que la matriz de *embeddings* del modelo contenga en sus filas vectores que se correspondan a una adecuada representación de cada token del vocabulario.

3.3. Modelos de Lenguaje - Generación de Lenguaje Natural

Se conoce como Modelo de Lenguaje a un sistema que dada una secuencia de palabras $x^{(1)}$, $x^{(2)}$, ..., $x^{(n)}$ es capaz de determinar la distribución de probabilidad para la siguiente palabra $x^{(n+1)}$:

$$P(x^{(n+1)}|x^{(1)}, ..., x^{(n)}) \tag{3.1}$$

donde cada una de las palabras x^i puede ser cualquier token del vocabulario $V = \{w_1, ..., w_{|V|}\}$ [Goldberg and Hirst, 2017].

Estos modelos hoy en día se pueden encontrar en sistemas como los teclados predictivos o en las sugerencias en los buscadores, como también en los sistemas de generación de texto. Un sistema generador de texto construido a partir de un modelo de lenguaje funciona, en general, de la siguiente manera:

1. Se define una sentencia inicial $S = x^{(1)}, x^{(2)}, ..., x^{(n)}$, comúnmente llamada semilla o *seed*.
2. Se utiliza S como entrada del modelo de lenguaje, que devuelve una distribución de probabilidades para todos los elementos del vocabulario indicando sus respectivas probabilidades de ser el siguiente token de S.
3. Se actualiza S agregándole el token del vocabulario con mayor probabilidad de ser el siguiente token de S.
4. Se repite desde el paso 2, pero esta vez con los últimos n elementos de S. Esto se repite una cantidad fija de veces, o hasta que S alcance una longitud deseada.
5. S contendrá el texto generado a partir de la sencencia semilla.

Una aclaración respecto al punto 3 es el hecho de en muchos casos no se toma para continuar el texto generado el token que estrictamente presente la mayor probabilidad, si no que se toma un token al azar a partir de la distribución de probabilidades generada por el modelo [Goldberg and Hirst, 2017]. A esta distribución de probabilidades se la suele además cocientar por un valor conocido como temperatura para aumentar o disminuir las distancias entre las probabilidades de cada token y así controlar la "azarosidad" del modelo. Una menor temperatura implica que las diferencias entre las probabilidades serán mayores, por lo que en este caso será aún más factible que se seleccionen tokens con probabilidad inicialmente alta y también será mas raro que se seleccionen tokens con probabilidades bajas, resultando en un sistema de generación más conservador. Por otro lado una temperatura alta suaviza las diferencias entre las posibilidades de cada token, resultando en un sistema que utilice más frecuentemente tokens menos probables, generando texto con mayor diversidad pero más propenso a equivocarse [Hinton et al., 2015].

A lo largo de esta sección se detallan algunos de los modelos de lenguaje más utilizados como lo son los modelos de lenguaje de n-gramas (subsección 3.3.1), los modelos neuronales con ventana fija (subsección 3.3.2) y los modelos neuronales recurrentes (subsección 3.3.3).

3.3.1. Modelos de Lenguaje de n-gramas

Los llamados modelos de n-gramas son los modelos de lenguaje más sencillos. Estos surgen de la idea de obtener la probabilidad de la enésima palabra en una cadena $P(w_1, w_2, ..., w_n)$ utilizando simplemente la regla de la cadena en probabilidades:

$$\begin{aligned} P(w_1^n) &= P(w_1)P(w_2|w_1)P(w_3|w_1^2)...P(w_n|w_1^n - 1) \\ &= \prod_{k=1}^{n} P(w_k|w_1^{k-1}) \end{aligned} \tag{3.2}$$

Sin embargo, la cantidad de posibles sentencias que se deben tener en cuenta si se quieren considerar todas las palabras previas a una dada para calcular su probabilidad es demasiado grande y seguramente muy variada, por lo que nunca será posible tener en cuenta los datos suficientes para poder realizar estimaciones confiables. Es por esto que estos modelos proponen la utilización de n-gramas. Un n-grama no es mas que una secuencia de n palabras, que son las que estos modelos utilizan para para predecir la palabra $n + 1$. Uno de los ejemplos más comunes de este tipo de modelos son los llamados modelos de Markov, dado que la asunción de que la probabilidad de ocurrencia de una palabra está determinada únicamente por un número n fijo de palabras anteriores se conoce como asunción de Markov. Entonces, la probabilidad de ocurrencia de una palabra m dada todas las anteriores se asume similar al de su ocurrencia dadas únicamente las n palabras anteriores:

$$P(w_m|w_1^{m-1}) \approx P(w_m|w_{m-n+1}^{n-1}) \tag{3.3}$$

En general, estos modelos resultan modelos insuficientes del lenguaje principalmente debido a que suelen existir en él dependencias de mucha distancia, que no son posibles de tener en cuenta trabajando con un tamaño de n-grama fijo. Además, son modelos costosos en términos de almacenamiento: dado un texto con un vocabulario V los posibles n-gramas sobre dicho texto serán $|V|^n$. Si bien no todos los n-gramas ocurrirán o serán válidos, la cantidad a tener en cuenta crece al menos multiplicativamente al aumentar el tamaño de los n-gramas. También, el trabajar con tamaños de n-gramas grandes implicará, por el tamaño en general significativo del vocabulario y la naturaleza misma del lenguaje natural que las estadísticas obtenidas a partir de estos sean dispersas [Goldberg and Hirst, 2017].

3.3.2. Modelos de Lenguaje Neuronal con ventana fija

Estos modelos utilizan una arquitectura de red neuronal (descrita con mas detalle en el el capítulo 2) para realizar la predicción propia de un modelo de lenguaje: dada una secuencia de palabras $x^{(1)}, x^{(2)}, ..., x^{(n)}$ obtener la palabra $x^{(n+1)}$, con la salvedad de que, al igual que los modelos de n-gramas, solo pueden contemplar una cantidad fija N de palabras.

Típicamente estos modelos toman como entrada una representación *one-hot* de las palabras, por lo que cuentan con una capa inicial de dimensión NxV, siendo V el tamaño del vocabulario. Esta se encuentra inmediatamente conectada a una capa de *embeddings* que le asigna a cada palabra su vector de *embeddings* correspondiente, que puede haber sido entrenado previamente o entrenarse junto al resto de la red. Luego tendrá una o más capas ocultas y por último, a

la capa de salida de tamaño V se le aplicará una función *softmax* para obtener la predicción del modelo acerca de las probabilidades de ocurrencia de cada palabra del vocabulario. Una representación gráfica de este tipo de modelos se puede observar en la figura 3.9.

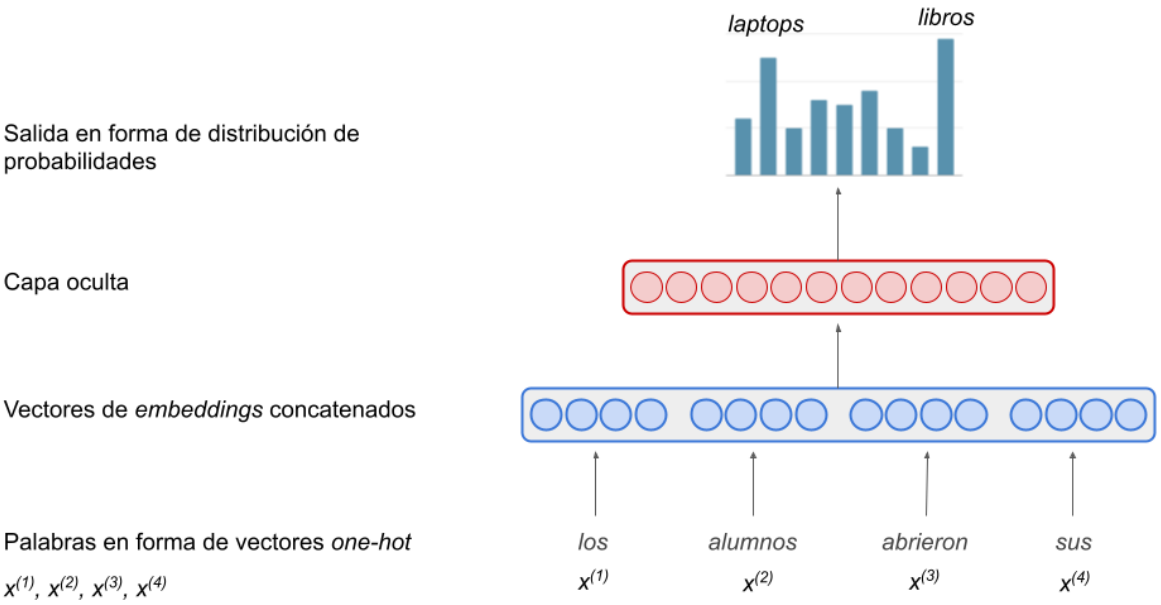


Figura 3.9: Ejemplo de arquitectura de red neuronal correspondiente al modelo de lenguaje descrito.

Las principales ventajas de este modelo respecto a los modelos de n-gramas tienen que ver con que este no tiene la necesidad de almacenar todos los n-gramas para realizar una predicción. Estos resultan necesarios únicamente durante la etapa de entrenamiento del modelo, lo que reduce o elimina los problemas relacionados con el almacenamiento y la dispersión de datos.

Sin embargo, persisten las limitaciones respecto al tamaño fijo de ventana: al igual que los modelos anteriores, este no será capaz de tener en cuenta dependencias entre palabras que se encuentren a una distancia mayor a N, que para los tamaños típicos de ventana utilizados, dichas dependencias suelen abundar en el lenguaje natural.

3.3.3. Modelos de Lenguaje con Redes Neuronales Recurrentes

Las Redes Neuonales Recurrentes (RNN por sus siglas en inglés) son una familia de redes neuronales que, entre otras características, tienen la capacidad de procesar entradas de cualquier longitud, lo que soluciona uno de los principales problemas de los modelos mencionados anteriormente para tareas de NLP para poder lidiar con dependencias entre palabras a grandes distancias. Su arquitectura se especifica de forma más detallada en el capítulo 2 pero el factor que diferencia a este tipo de redes de las arquitecturas típicas es que al momento de procesar una

secuencia la salida depende no solo del elemento n de esta, si no también del resultado de haber procesado los elementos $n-1$, $n-2$, ... Dicho resultado suele corresponderse, en su versión más básica, con el estado oculto de la red al momento de procesar cada elemento. Esta forma de procesamiento secuencial de los elementos, que en el caso del NLP se trata de tokens, permite capturar la naturaleza secuencial inherentemente presente en el lenguaje [Young et al., 2018]. En la figura 3.10 se muestra de forma gráfica un ejemplo de esta arquitectura.

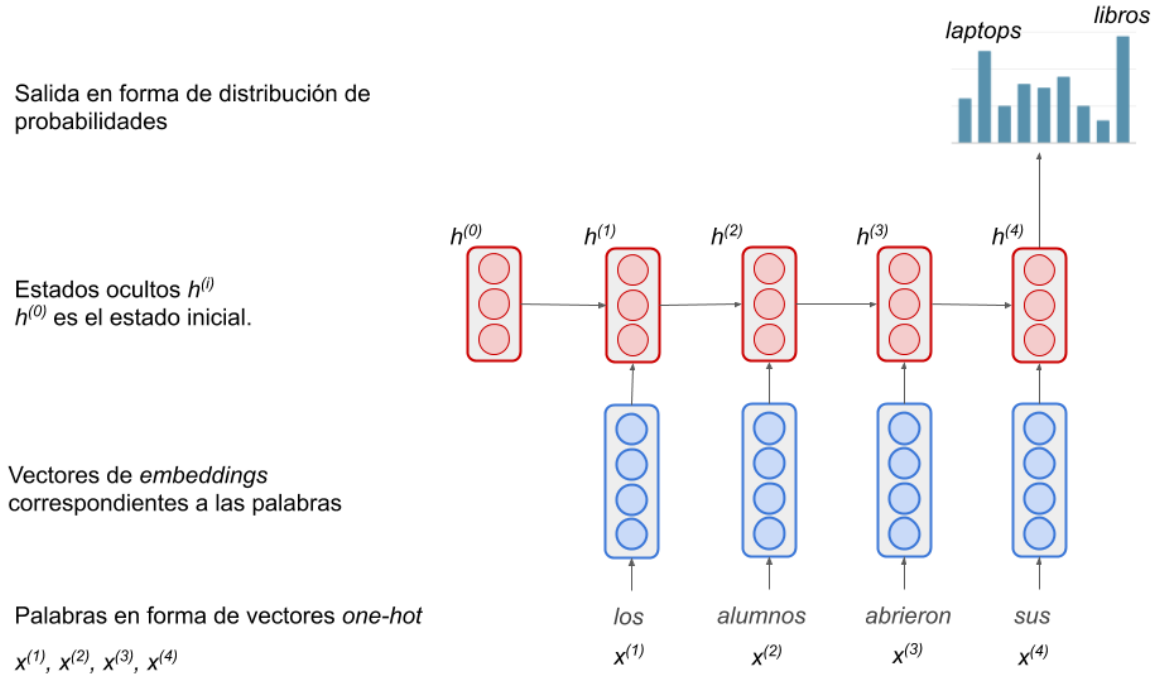


Figura 3.10: Ejemplo de modelo de lenguaje utilizando una arquitectura de red neuronal recurrente.

Las principales ventajas de este tipo de modelos, como se mencionó anteriormente, es la posibilidad de utilizar entradas de longitud variable y que tienen la capacidad, en teoría, de utilizar información de las palabras procesadas muchos pasos atrás. En la práctica, muchas veces los modelos mas simples no pueden discriminar correctamente la información relevante que se encuentra a grandes distancias de una palabra procesada, y en sus versiones más simples sufren lo que se conoce como desvanecimiento de gradiente (*vanishing gradient*), es decir que gradiente toma valores muy pequeños a medida que se propaga hacia atrás impidiendo un entrenamiento eficiente [Young et al., 2018]. Para superar estos problemas es que se han desarrollado arquitecturas específicas dentro de las RNN que buscan solucionar este tipo de problemas como son las llamadas de *Long Short-Term Memory*, que también se encuentran desarrolladas con mayor detalle en el capítulo 2. Como desventaja de este tipo de redes se debe mencionar el hecho de que el entrenamiento de redes recurrentes resulta computacionalmente mas costoso que para los modelos neuronales anteriores.

3.4. Generación de texto estructurado

Si bien hasta ahora se habló de modelos capaces de generar texto, estos por si solos muchas veces pueden no resultar suficientes para captar cuando un texto debe respetar una estructura específica, como es el caso de la generación de versos en el contexto de una poesía o en el caso del presente trabajo, de textos que se adecúen al género *freestyle*. Esta estructura refiere a características que debe poseer el texto, propias de su género, como lo pueden ser la cantidad de sílabas que debe tener una linea, una determinada forma de acentuación de las palabras, la cantidad de lineas por verso o la estructura de rima esperada para cada uno.

Algunas de estas características estructurales resultan mas factibles de lograr a través del entrenamiento de modelos de lenguaje. En lo que refiere al largo de las lineas y de los versos, es común la inclusión de tokens especiales que indican el fin de una linea y el fin de un verso para que luego sea el mismo modelo el que aprenda cuando debe insertar uno de estos tokens en el texto generado, como se puede ver por ejemplo en [Potash et al., 2015] o [Zugarini et al., 2019]. Sin embargo, muchas veces se opta por fijar la cantidad y el tamaño de las lineas de un verso y que los modelos de lenguaje sean responsables únicamente de generar los tokens que vayan a formar parte de dicha estructura. Un ejemplo de esto se ve en [Yi et al., 2017], donde el modelo de lenguaje genera una linea a partir de la linea anterior, y se determina anteriormente que cada verso estará compuesto por cuatro lineas. También en [Ghazvininejad et al., 2016] se ve como dada una estructura de rima fija, y por ende también una cantidad fija de lineas, genera posibles lineas candidatas y utiliza una RNN para definir cual de estas es la que mejor se adecua a lo que se busca generar.

Otras de las características mencionadas al comienzo de la sección son más difíciles de captar con modelos de lenguaje, como pueden ser la rima y la métrica. Por esta razón, dichas características suelen ser incorporadas a través de reglas fijas que afectan explícitamente el texto generado para que este sea acorde a la estructura deseada, lo que puede afectar negativamente en su significado [Yi et al., 2017].

En primer lugar, la rima se define como la igualdad completa (en caso de ser una rima consonante) o parcial (en caso de ser una rima asonante) entre los sonidos de dos o más palabras. La generalidad de los textos poéticos, como también el caso puntual de las letras de las batallas de *freestyle* están compuestos por estrofas, cuyas lineas, conocidas como versos, deben seguir una estructura particular dada por qué versos finalizan con palabras que riman entre ellas. Estas estructuras se describen típicamente utilizando letras mayúsculas que representan cada verso, y los versos representados con la misma letra se corresponden con los que finalicen con palabras que rimen entre sí. Entonces una estructura de rima descrita como AAAA, refiere a que todos sus cuatro versos finalizan con palabras que riman entre sí, mientras que una estructura ABAB representa una estrofa donde el primer verso rima con el tercero y el segundo con el cuarto. Además de las mencionadas, otras estructuras típicas son ABBA o AABB. En la figura 3.11 se muestran algunos ejemplos de versos con sus respectivas estructuras de rimas.

Como ya se mencionó, típicamente es difícil que un modelo de lenguaje sea capaz de tener en

Aquí me pongo a cantar	**A**	*Pido a los Santos del Cielo*	**A**
al compás de la vigüela,	**B**	*que ayuden mi pensamiento,*	**B**
que el hombre que lo desvela	**B**	*les pido en este momento*	**B**
una pena estraordinaria,	**C**	*que voy a cantar mi historia*	**C**
como la ave solitaria	**C**	*me refresquen la memoria,*	**C**
con el cantar se consuela.	**B**	*y aclaren mi entendimiento.*	**B**

Figura 3.11: Versos extraídos del poema Martín Fierro con una estructura de rima ABBCCB.

cuenta esta característica. Es posible que esto se deba al hecho de que en caso de la tokenización a nivel de palabras o subpalabras, le resulta imposible al modelo identificar qué tokens riman entre sí, ya que la representación de estos no contempla las letras que los componen. Además, identificar los patrones de rima descritos requiere que el modelo sea capaz de identificar dependencias a mediana o larga distancia entre las palabras finales de los versos que riman. Por estas razones es que en el caso de buscar generar texto que cumpla con cierta estructura de rima es común en primer lugar fijar esta estructura para luego adecuar a esta el texto generado.

Para lograr esto, primero se define un patrón de rimas como los mencionados (AAAA, ABAB, etc.). En caso de que el género que se busque generar no defina un patrón de rimas específico (como se ve en [Yi et al., 2017], que busca generar una forma particular de poesía china que utiliza un patrón de rima ya definido), este se suele elegir de forma aleatoria para cada estrofa. Luego, se busca adaptar el texto generado a dichas estructuras, para lo que es necesario modificar, como mínimo, la última palabra generada de cada verso, lo que presenta dos problemas:

- En caso de no estar fijo el largo de cada verso, no es posible conocer cuando el modelo generará la última palabra de cada uno.
- Si se modifica la última palabra de un verso para que se adapte a una estructura de rima en particular, es probable que eso ocasione daños en el valor semántico del verso, ya que esta última palabra modificada no necesariamente será adecuada para encontrarse a continuación de sus predecesoras en dicho verso, lo que da lugar a la generación de textos incoherentes.

Por estas razones, para generar texto que cumpla con patrones específicos de rimas suele ser útil la generación inversa del texto, es decir, que dada una secuencia de palabras $x^{(1)}$, $x^{(2)}$, ..., $x^{(n)}$ se busca predecir cual será la palabra anterior x^0 y a partir de esto generar un texto. De esta forma es posible, en primer lugar conocer de antemano cual será la última palabra de cada linea y luego poder modificar esta para que efectivamente se adapte a la estructura buscada, rimando con las palabras necesarias de los versos siguientes. Entonces, el modelo generará a partir de esta última palabra sus palabras antecesoras en el verso, obteniendo un verso más

coherente. Tanto en [Yi et al., 2017] como en [Lau et al., 2018] se pueden encontrar ejemplos trabajos donde se utiliza este recurso.

Hasta ahora en este apartado no se hizo mención al problema específico de identificar si dos palabras efectivamente riman. Si bien esto puede variar de acuerdo al idioma, típicamente para afrontar este problema trabajos que generen texto en idioma inglés utilizan diccionarios de rimas, disponibles como librerías en varios lenguajes de programación, cosa que no ocurre para la generación de texto en español, idioma para el cual no se ha encontrado literatura que describa posibles formas de llevar a cabo esta tarea en específico.

3.4.1. Batallas de *freestyle*

El *freestyle* es un subgénero del rap que se lleva a cabo de forma improvisada. Una de sus formas más conocidas son las Batallas de *Freestyle*, donde los ejecutantes, llamados *freestylers* o MCs compiten entre sí midiendo quien es capaz de ejecutar los mejores versos [Edwards, 2009]. Si bien los primeros registros de Batallas de *freestyle* datan de alrededor de la década de los 80 en Estados Unidos, es posible trazar un paralelismo entre estas y el contrapunto, una forma de payada típica de la cultura gauchesca del Río de la Plata donde, de la misma manera, los ejecutantes se miden en su capacidad de improvisar versos y de la cual se tienen registros desde principios del siglo XIX.

Hoy en día este género ha crecido mucho y particularmente en países de habla hispana. Desde el 2005 se disputa a nivel internacional la llamada Red Bull Batalla de los Gallos, que reúne a los máximos exponentes del *freestyle* en español y en estos últimos años se han establecido ligas nacionales en España (2017), Argentina (2018), México (2019), Chile (2019) y Perú (2020). El formato de estas si bien a variado a lo largo del tiempo es el mismo para todas: los participantes se miden a lo largo de seis encuentros o rounds cada uno con reglas particulares, y un jurado evaluará cada estrofa con un puntaje de entre 0 a 4 puntos en función de la calidad de la rima y cuanto se adecúe a las reglas del round en particular y hasta dos puntos extra por otros factores tales como puesta en escena. Algunos de los formatos clásicos de round son los llamados "easy mode" o "hard mode", donde cada *freestyler* tiene un minuto para elaborar estrofas que incluyan palabras que se generan aleatoriamente cada 10 y 5 segundos respectivamente, o el conocido como "deluxe", que consiste en 160 segundos donde cada participante elabora una estrofa con temática libre y el adversario responde de inmediato con otra estrofa, entre otros.

Además de las dificultades propias de la generación de texto estructurado mencionadas en la sección 3.4, la generación de texto para batallas de *freestyle* en español cuenta con dificultades particulares que no se encuentran presentes en otros géneros:

- Las bases de datos que contienen transcripciones de batallas de *freestyle*, al estar compuestas de letras improvisadas en el momento, son mucho menos frecuentes que, por ejemplo, las que contienen poemas o letras de canciones de otros géneros ya que en estos últimos, el texto se suele publicar junto con la obra. Las batallas de *freestyle* por su parte están pensadas para ser vistas en vivo y están disponibles en su mayoría en formato audiovisual.

- La generación de texto en español implica que muchas de las soluciones implementadas en el marco de generación de lenguaje natural en inglés no pueden trasladarse a este trabajo fácilmente. Este es el caso, por ejemplo, de la representación de palabras usando *embeddings* pre-entrenados sobre texto en inglés, o la utilización de diccionarios de rimas en inglés para la identificación de estas.
- El hecho de que sea un subgénero surgido inicialmente en inglés resulta en que se incluyan muchos anglicismos, es decir, palabras de dicho idioma dentro de un *freestyle* en español, además de deformaciones o "españolizaciones" de dichas palabras u otras pertenecientes a la jerga específica de la comunidad. Esto implica que ni siquiera muchas de las soluciones implementadas para la generación de texto en español puedan trasladarse fácilmente a este trabajo:
  - En el caso de querer utilizar *embeddings* pre-entrenados, es probable que si fueron realizados en base a palabras, no contemplen todos los casos mencionados de palabras por fuera del español tradicional.
  - El diseño de un algoritmo que identifique rimas en español lo hará teniendo en cuenta las vocales de la palabra, ya que la fonética de la palabra se corresponde con su escritura. Al incluir palabras en inglés esto deja de suceder. Un ejemplo de esto es la palabra en inglés *beat*, muy utilizada en el contexto de las batallas de *freestyle* y que se pronuncia "bit", por lo que no podemos esperar que el algoritmo mencionado identifique correctamente una rima para esta palabra.
- Según el formato de round tomado como referencia, la generación de texto para este género en particular implica que cada estrofa generada debe responder a una estrofa anterior o bien utilizar una palabra específica. El como formular una respuesta adecuada o que tener en cuenta de la estrofa a la que se responde, así como la mejor forma de incluir una palabra particular no resultan problemas triviales.

Así como el género presenta las dificultades particulares mencionadas, el hecho de ser un género competitivo en conjunto con el que se lleve a cabo utilizando la improvisación en tiempo real constituyen motivaciones por las cuales resulta interesante el desarrollo de un sistema capaz de generar texto de acuerdo a este.

3.4.2. Métricas de evaluación

Las métricas utilizadas para evaluar modelos de lenguaje difieren significativamente de las utilizadas para modelos típicos de Machine Learning, ya que típicamente no es posible determinar de una forma binaria o al menos numérica qué tan correctamente un modelo eligió la siguiente palabra de una frase. Por esta razón es que la generación de lenguaje natural continúa siendo un campo donde la evaluación por parte de expertos representa un factor clave.

De cualquier manera, han surgido diferentes métodos que buscan evaluar de forma automática modelos de generación de lenguaje, siendo uno de los más populares hoy en día el conocido

como BLEU, presentado por primera vez en [Papineni et al., 2002]. Este método fue pensado inicialmente para evaluar modelos dedicados a la traducción de texto, por lo que contempla que para cada texto generado existe un texto de referencia, en el caso de las traducciones la traducción exacta, al que el texto generado debe ser por lo menos similar.

Sin embargo, este no es el caso de la generación de texto con fines artísticos como lo son versos poéticos o en el caso de este trabajo, del estilo de *freestyle* donde no contamos con un texto de referencia, si no que se busca generar texto nuevo, distinto a cualquiera que esté contenido en el conjunto de datos. En este tipo de trabajos se suele tener aún mas en cuenta el factor de evaluación humano, como se ve en [Zugarini et al., 2019], donde se buscan generar versos similares a los del libro "La Divina Comedia" y evalúan los resultados en función de expertos que tratan de distinguir versos originales de los generados. También en el ya mencionado [Yi et al., 2017], donde se entrena a un modelo para generar poesía típica china, se realiza una evaluación con expertos que puntúan los versos generados en función de distintos criterios como la fluidez, coherencia, entre otros.

3.4.3. Trabajos relacionados

En esta sección se analizan trabajos relacionados al presente en lo que refiere a la generación de texto estructurado, revisando trabajos recientes que se enmarquen en este área. En primer lugar, resulta necesario mencionar que el estado del arte actual en lo que refiere a generación de lenguaje natural lo alcanza el modelo GPT-3, presentado en [Brown et al., 2020]. Este es un modelo general del lenguaje, es decir que no fue diseñado para resolver una tarea específica (como pudiera ser la traducción de textos), si no que es posible interactuar con el modelo a través de una consola de texto en la que se le describe, utilizando lenguaje natural y a lo sumo algunos ejemplos, cual es la tarea que se espera que realice. Así, por ejemplo, fue posible especificarle que generara artículos de noticias los cuales resultaron difíciles de distinguir para evaluadores humanos de artículos reales [Brown et al., 2020]. También es posible encontrar en trabajos como [Branwen, 2020] el uso de este modelo para la generación de textos estructurados tales como diálogos o poesía. La figura 3.12 muestra un poema generado de esta forma. Para estos casos el modelo fue capaz de reproducir de forma aceptable su estructura, aunque no necesariamente respetando un esquema de rima. Resulta importante mencionar que GPT-3 fue entrenado sobre la base de datos Common Crawl[1], una base de datos que cuenta con alrededor de un billón de palabras, mucho mayo que cualquier base de datos de texto estructurado disponible hoy en día. Dado que al momento de desarrollo de este trabajo el acceso al modelo no es libre no se ha podido evaluar su capacidad para el dominio específico de este trabajo como lo son las batallas de *freestlye*.

A continuación, se describen trabajos que desarrollen específicamente la generación de texto estructurado. Entre estos es posible listar a [Lau et al., 2018], donde se busca generar texto poético, más específicamente en formato de soneto en inglés. Para esto, el texto generado debe consistir de cuatro lineas que se ajusten a un esquema de rima específico y a un esquema de

[1] https://commoncrawl.org/the-data/

"*Rime of the Ancient Google,*" from Guillaume de Salluste Du Bartas's "*Divine Weeks and Works*" (1578–1584)

Google
google in Google,
Google has gone to God.
You cannot conceive it or guess it,
For all of a sudden it hast happened.
Come, come, whoever you are,
Oh come now,
Come, come, whoever you are,
Hasten, hasten!

Figura 3.12: Poema extraído de [Brown et al., 2020] generado por el modelo GPT-3.

acentuación llamado pentámetro. En primer lugar, se diseñó un modelo de lenguaje compuesto por una red neuronal recurrente el cual se entrenó con sonetos extraídos de base de datos del Proyecto Gutenberg y otro modelo llamado Modelo de Pentámetro, que puntúa una linea en función de cuanto se ajusta a dicho esquema de acentuación. El modelo de lenguaje fue entrenado para realizar generación inversa del texto (proceso descrito en la sección 3.4).Luego, el proceso de generación consiste en utilizar el modelo de lenguaje para generar diez lineas candidatas, de las cuales se utilizará la que resulta más adecuada según el modelo de Pentámetro. A continuación se selecciona un esquema de rimas al azar (AAAA, ABAB, etc.) y se utiliza el modelo de lenguaje para generar las últimas palabras de cada linea que, en caso de no corresponderse con el esquema de rima buscado no se tienen en cuenta y se genera una nueva palabra. A partir de las palabras finales que cumplen con el esquema buscado, el modelo de lenguaje completa la generación de los versos. Los poemas generados con este modelo fueron presentados a evaluadores expertos que si bien valoraron positivamente su métrica y rima, incluso mejor que sonetos escritos por humanos, les resultó fácil distinguir entre los poemas generados por el modelo presentado y los utilizados para su entrenamiento. Esto se debió principalmente a la "falta de impacto emocional y legibilidad" de los primeros [Lau et al., 2018].

En idiomas distintos del inglés es posible encontrar trabajos como [Yi et al., 2017], en el cual se utilizan modelos de redes neuronales recurrentes para generar poesía clásica china, más específicamente *quatrains* o cuartetos. Estos están compuestos de cuatro versos, cada uno compuesto por cinco o siete caracteres chinos y donde el segundo, el cuarto y opcionalmente el primero deben rimar. En este caso, utilizaron tres modelos distintos, todos compuestos por redes neuronales recurrentes y entrenados a partir de 398.391 poemas clásicos chinos. El primero, llamado Word Poetry Module (WPM) fue entrenado para generar un verso a partir de una palabra; el segundo, Sentence Poetry Module (SPM) busca generar un verso a partir de otro verso y el tercero, Context Poetry Module (CPM) tiene el objetivo de generar un verso a partir de varios anteriores, y es el que se busca que capte los patrones de rima de los cuartetos. El proceso de

generación consiste en utilizar una palabra ingresada por el usuario como entrada para el WPM que se encarga de generar la primera linea del cuarteto, que luego es utilizada como entrada del SPM para generar la segunda linea. Por último, la primer y segunda linea se utilizan como entradas del CPM para generar la tercera, y esta junto con la primera de nuevo se ingresan al CPM para generar la cuarta y última linea. Los poemas generados con este modelo fueron sometidos a evaluación humana donde expertos debían puntuarlos junto con poemas escritos por humanos en función de criterios tales como coherencia, significado, fluidez o características poéticas. La evaluación obtuvo resultados positivos: si bien los versos generados por el modelo obtuvieron en general menor puntuación que los escritos por humanos, la distancia no fue tan significativa, y el modelo obtuvo mucho mejor puntaje que otros generados por modelos propuestos anteriormente. En la figura 3.13 se incluye un par de poemas generados por este modelo extraídos de [Yi et al., 2017].

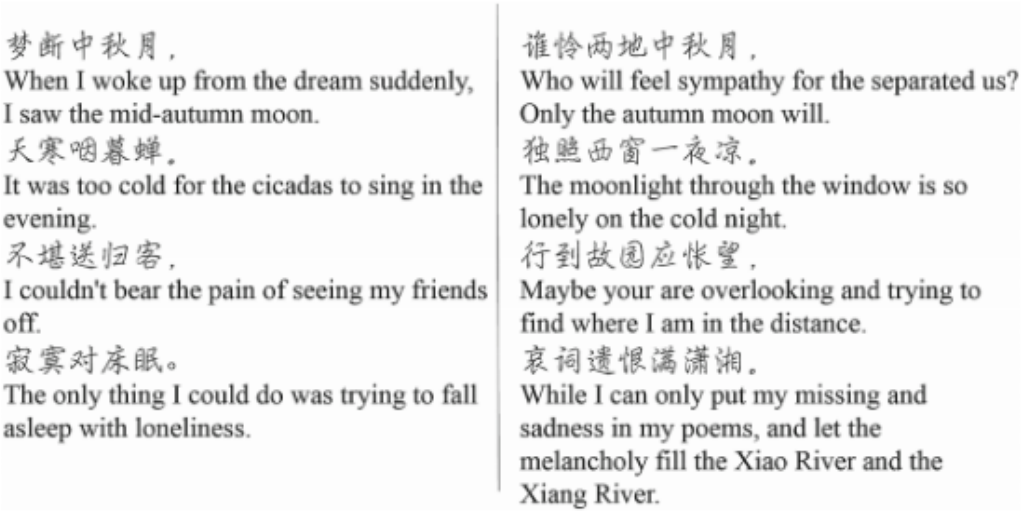


Figura 3.13: Figura extraída de [Yi et al., 2017] con un par de versos generados por el modelo presentado en dicho trabajo.

En [Zugarini et al., 2019] se busca generar poesía en italiano replicando el trabajo del autor de La Divina Comedia Dante Alighieri. Para esto entrenaron un modelo neuronal recurrente, más específicamente una red LSTM, que trabaja con tokenización a nivel de sílabas, agregando además de tokens especiales para representar el comienzo y final de un terceto (conjunto de tres versos). En primer lugar, preentrenaron el modelo utilizando grandes conjuntos de textos públicos en italiano en busca de que el modelo aprendiera la sintaxis y gramática del idioma para luego entrenarlo utilizando específicamente la Divina Comedia. El proceso de generación para cada terceto consiste en generar sílabas a partir de un token de comienzo de terceto hasta que se genere un token de final de terceto o se alcance el límite de 75 sílabas. Así se generan 2.000 tercetos los cuales se puntúan en función de características conocidas del autor, como lo son la cantidad de sílabas: los tercetos en particular están compuestos de tres endecasílabos,

es decir, versos de once sílabas; el esquema de rimas que debiera ser encadenado (ABA) y la utilización de palabras que se encuentren dentro del vocabulario de la divina comedia. Al igual que en otros de los trabajos ya mencionados, el texto generado se sometió a evaluación humana que buscaba distinguir los tercetos generados por el modelo de otros de la autoría de Dante. En este caso, los tercetos generados fueron percibidos como escritos por humanos alrededor de la mitad de las veces (56,25 %) que los realmente escritos por Dante.

El único trabajo que se encontró donde se considera la generación de texto estructurado en español es [Ghazvininejad et al., 2016], que si bien está desarrollado en inglés, se menciona que el enfoque utilizado es trasladable a otros idiomas, entre ellos el español. Dicho trabajo presenta un sistema capaz de generar poesía con una temática particular ingresada por el usuario. Para esto, se compuso un vocabulario al que además a cada palabra de este se la relacionó con su patrón de acentuación (que indica cuales de sus sílabas están acentuadas). Luego, dada una temática particular se obtiene una lista de 1.000 palabras relacionadas con esta utilizando el modelo word2vec [Mikolov et al., 2013] entrenado sobre el primer millar de millón de caracteres de Wikipedia y se separan las palabras en clases de rimas, es decir, conjuntos de palabras que riman entre sí. Se seleccionan entonces un par de rimas de forma aleatoria entre los posibles pares y se utiliza un autómata finito que aceptará todas las posibles secuencias de palabras del vocabulario que se ajusten a la estructura deseada y finalicen con las palabras de los pares de rimas obtenidos. Si bien todas las secuencias cumplen con la estructura de estrofa buscada, la mayoría de estas no tienen sentido a nivel semántico. Entonces, para seleccionar los versos que efectivamente tienen sentido se utiliza un modelo de lenguaje compuesto por una red neuronal recurrente entrenada sobre 94.882 letras de canciones en inglés que se utiliza para puntuar cada uno de los posibles caminos.

Por último, no se han encontrado trabajos que traten la generación de texto específicamente para batallas de *freestyle*, pero si en [Potash et al., 2015] se presenta un modelo de lenguaje compuesto por redes neuronales LSTM para la generación de letras de rap. Este fue entrenado a partir de 219 canciones con al menos 175 palabras en cada una. En este caso no se aplicaron ningún tipo de restricciones específicas respecto al esquema de rima o métrica que debía seguir el texto generado. El modelo define el largo de los versos y de las estrofas a través de la generación de tokens especiales para indicar ambas cosas respectivamente. En el trabajo se evalúa que se logró generar de forma efectiva nuevas letras que repliquen el estilo de diferentes artistas.

De los distintos trabajos listados se puede notar que si bien se han desarrollado varios trabajos en lo que respecta a la generación de texto estructurado utilizando modelos neuronales, la mayoría de estos apuntan a la generación de texto en inglés o a géneros clásicos de los que se dispone de gran cantidad de texto como son los casos de [Zugarini et al., 2019] y [Yi et al., 2017], a diferencia del trabajo actual que se centra en la generación en español y de un género con escasas cantidades de texto disponible hoy en día. Además, en todos los casos listados el fin de los proyecto se centra en replicar efectivamente un género artístico, no se deben tener en cuenta otros elementos que sí resulta necesarios considerarlos a la hora de batallas de *freestyle* como la necesidad de responder a un verso y en ultima instancia la posibilidad de medirse ante otro ejecutante.

Capítulo 4

Desarrollo Realizado

En este capítulo se describe el desarrollo realizado a lo largo de este trabajo, que consiste en un sistema de generación de lírica que se enmarca dentro del género *freestyle*, respetando su estructura y su rima. Los principales aportes de este trabajo son los siguientes:

- Dos bases de datos de textos, una primera que contiene letras de canciones del género de rap y otros similares y una segunda compuesta por transcripciones de batallas de *freestyle* exclusivamente, descritas en la sección 4.1.
- Un modelo de lenguaje neuronal entrenado con las bases de datos listadas para la generación de texto de estos géneros, detallado en la sección 4.3.
- El sistema de generación de líricas de *freestyle* que incluye algoritmos que trabajan sobre el resultado del modelo de lenguaje para garantizar que el texto generado respete la estructura y la rima requerida por el género, presentados en la sección 4.5.

Además, en la sección 4.2 se describe el trabajo de preprocesamiento necesario sobre los datos obtenidos para poder usarlos como entrada del modelo neuronal utilizado en conjunto con tareas de análisis del contenido de dichos datos.

4.1. Generación de Base de Datos

Dado que las competencias de *freestyle* son un fenómeno relativamente reciente (la primer competencia internacional oficial en español se realizó en el año 2005), sumado a la naturaleza de improvisación del género y en conjunto con que su ejecución suele estar acompañada de una puesta en escena de la que forma parte la gestualidad, pronunciación e interacción con el público, resulta esperable que la gran mayoría del material disponible hoy en día de este género se encuentre en formato de audio o más comúnmente audiovisual. Si bien existe gran cantidad de este tipo de material (grabaciones de audio o video de batallas de *freestyle*) al que se puede acceder a través de internet, es muy poco el material que se encuentra disponible en formato de

texto, como lo pueden ser transcripciones, necesario para entrenar modelos de lenguaje capaces de generar el tipo de texto buscado.

Siendo que en principio no se disponía de ninguna base de datos de texto que contenga efectivamente material del dominio buscado y como se mencionó en el párrafo anterior, al momento de realización de este trabajo no se pudo encontrar dicho material en internet de forma abundante, se consideró apropiado utilizar la técnica de *fine-tuning* preentrenando el modelo sobre material similar que sea de más fácil acceso en gran cantidad. Es por esto entonces, que para el desarrollo de este trabajo se generaron dos bases de datos: una primera que contiene letras de canciones de rap o géneros similares y una segunda que consiste únicamente de transcripciones de batallas de *freestyle*. A través del entrenamiento sobre la primera el modelo será capaz de obtener nociones no solo acerca de la estructura general del lenguaje español si no también acerca de cuestiones más específicas como la estructuración de un texto en estrofas o posiblemente la rima. Además, si bien estas no se corresponden exactamente con transcripciones de batallas de *freestyle*, al ser letras de géneros similares comparten en gran medida aspectos tales como el vocabulario o las rimas utilizadas, por lo que se consideró que los patrones aprendidos respecto a estos aspectos también pueden resultar de gran utilidad. Sin embargo, también hay que tener en cuenta que existen aspectos donde estas difieren de la lírica utilizada en una batalla de *freestyle*. Por ejemplo, típicamente las líricas de una canción se estructuran en distintas partes tales como estrofa o estribillo, repitiendo algunas de estas varias veces, cosa que no sucede dentro de las batallas de *freestyle* donde al ser las líricas improvisadas, no se repetirá, en teoría, ninguna estrofa. Esta primer base de datos se conformó a partir de letras obtenidas realizando *web scrapping* sobre las páginas `www.musica.com` y `www.letras.com` en los perfiles de los artistas más relevantes de estos géneros. Finalmente, la base de datos generada cuenta con 833 letras de canciones distintas, todas en idioma español, que contienen 54364 lineas y una totalidad de 381834 palabras. En la figura 4.1 se muestran 2 estrofas extraídas de esta base de datos.

Porque camino sobrado, sobrado
No se si es por como rimo, o es la forma en que mi estilo se mantiene parado
He conocido a la traición disfrazada de un amigo, así que estoy preparado
Se quien no estuvo en las malas y hoy que vivo de las buenas no te quiero a mi lado

El que no estuvo en las malas que hoy no llame ni pa' fiestas
Dani dónde está que no contesta?
Y dónde estabas vos cuando tocaba pa' 40?
Podrás llamar mañana? Discúlpame las molestias

Figura 4.1: Porción de la letra de la canción "A mi lado" del rapero Dani, incluída en la base de datos que contiene canciones del género.

La segunda base de datos consiste únicamente de transcripciones de batallas de *freestyle* y es la que se utilizó para el entrenamiento definitivo de los modelos de lenguaje luego de preentrenarlos con la base de datos ya descrita. Los datos utilizados para generar esta base de datos

resultaron mucho más difíciles de obtener, por lo que la cantidad de texto que contiene es significativamente menor al que se encuentra la base anterior. La única fuente de transcripciones de batallas de *freestyle* que se encontró al momento de realizar este trabajo fue una *wiki*[1] dedicada exclusivamente a transcribir líricas de este género y que cuenta con 23 transcripciones, de las cuales varias se encuentran incompletas. Se obtuvieron estas transcripciones también utilizando técnicas de *web scrapping* y se completaron manualmente a partir de material audiovisual de las respectivas batallas encontrado en YouTube. Se agregaron además otras transcripciones obtenidas también de forma manual a partir de material audiovisual de otras batallas encontradas también en esta plataforma para completar la base de datos con una totalidad de 36 transcripciones compuestas por 8355 lineas y 68763 palabras. En la figura 4.2 se muestran 4 estrofas extraídas de esta base de datos.

Quiero que escuches muy bien los ritmos que te tiro cada vez que te lo rimo de una forma atentamente
Para que quieras mirarme a la cara, no se compara con los ritmos que le tiro tengo la sangre caliente
Que mi doble tempo dice que no se entiende?
Pero que está perdiendo lo entendió perfectamente

Porque te aseguro que yo lo mato
Cada vez que vengo con las rimas agresivas te terminan te asesinan te aseguro que lo mato
Habla de fecha vietnamita el novato
Pero después soy yo el que lo copia a papo

Todo lo que digas, sabés lo que pasa? Es que son tonterías
Sabes que vengo con la palabra y lo reviento, las manos arriba pero de todo el evento
Yo soy un código y con eso estoy contento
Porque me odian todos pero no niegan que tengo talento

Te topaste a la que es-es-es tu peor pesadilla
Cuando cua-cua-cuando me dice que a mí me gana le dejó las cosas claras y le perforo las costillas
Este rapero dice que brilla, pero cuando rima contra mi queda a la orilla
Dice que muy bueno que es como una maravilla, yo no soy el trueno pero a vos te di vuelta la tortilla

Figura 4.2: Porción de la transcripción de una batalla de *freestyle* entre los raperos Wos y Cacha en el marco del torneo conocido como Freestyle Master Series (FMS) y que se obtuvo originalmente a través de YouTube[2].

La tabla 4.1 muestra una descripción general de las dos bases de datos, las cuales se encuentran públicas y pueden ser accedidas dentro del repositorio de la presente tesina [3,4].

[1] https://batallas-de-rap-lyrics.fandom.com/es/wiki/Batallas_de_Rap_Lyrics_Wiki
[2] https://www.youtube.com/watch?v=gLdhy2FoaVk&t=1368s&ab_channel=UrbanRoosters (minuto 19:45)
[3] https://github.com/midusi/freestyle_generator/tree/master/raw/hip_hop_lyrics
[4] https://github.com/midusi/freestyle_generator/tree/master/raw/freestyle_lyrics

	BD con líricas de canciones	BD con batallas de *freestyle*
Total de documentos	833	36
Total de lineas	54364	8355
Total de palabras	381834	68763

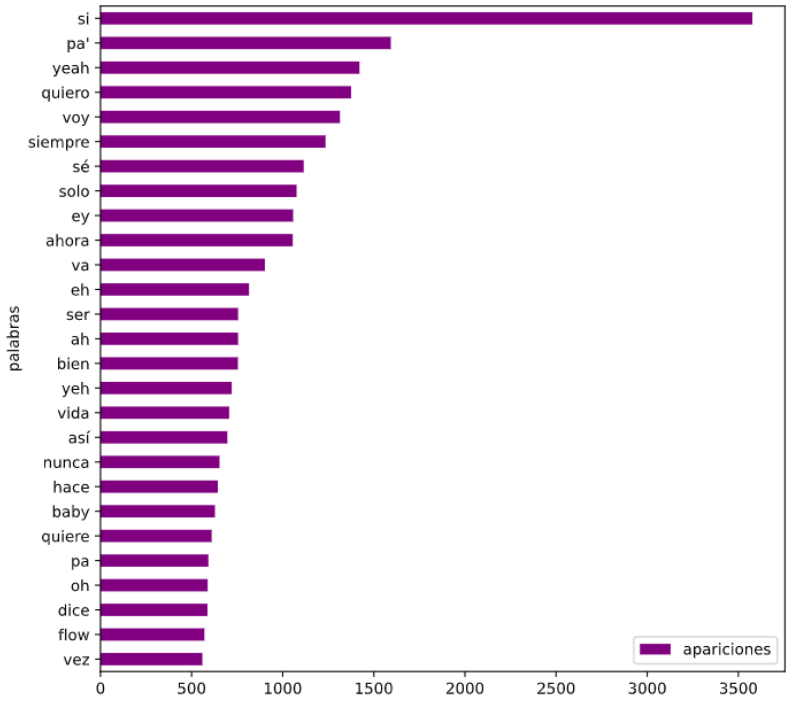
Tabla 4.1: Resumen del contenido de las dos bases de datos generadas.

4.2. Preprocesamiento y análisis de los datos

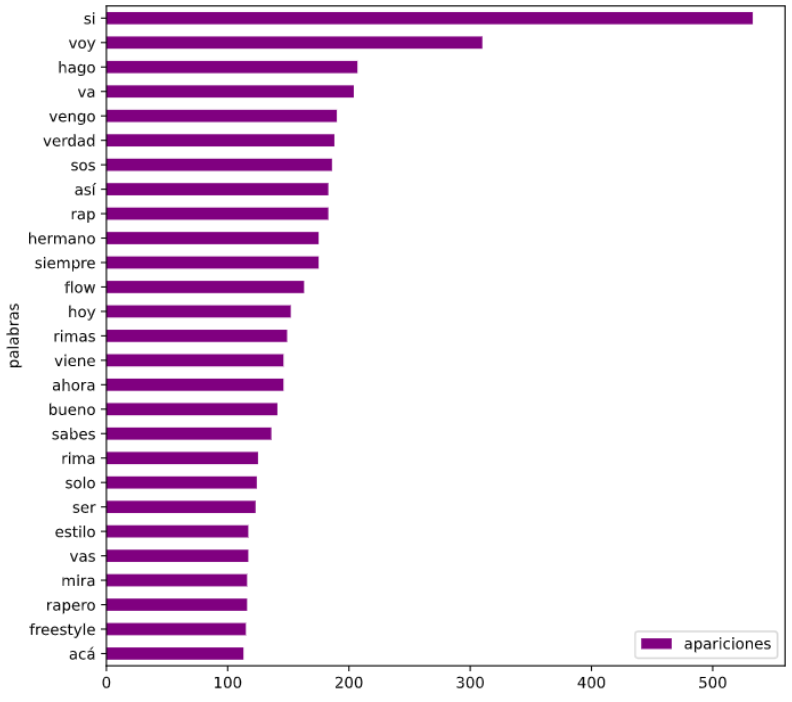
Previo a utilizar el texto obtenido en el conjunto de datos para el entrenamiento del modelo se realizaron sobre este tareas de preprocesamiento y análisis. En primer lugar, se debieron depurar los textos removiendo expresiones que no formaban parte de la letra, tales como indicaciones de la repetición de algún verso ("x2", "bis"), de una parte específica de la canción ("coro:") o del cantante de un verso en particular. Luego se hizo un análisis de las palabras presentes en el texto para validar algunos de los aspectos de las batallas de *freestyle* descritos en la sección 3.4.1 y que se describe a continuación en la subsección siguiente. Por último, se probaron distintos métodos de tokenización y se analizó su viabilidad para los conjuntos de datos disponibles.

4.2.1. Análisis de vocabulario

El análisis de vocabulario realizado consistió, en primer lugar, en descartar las llamadas palabras vacías (conocidas en inglés como *stop words*) [Luhn, 1960]. Estas son palabras tales como pronombres, artículos o preposiciones que suelen ser las más frecuentes en cualquier texto de un idioma en particular y no suelen ser tenidas en cuenta para trabajos de NLP [Rajaraman and Ullman, 2011]. Luego, se realizó una tokenización a nivel de palabras y se contabilizaron los tokens con más apariciones tanto en la totalidad de los conjuntos de datos como en el específico de transcripciones de batallas de *freestyle*. A partir de esto se generaron dos histogramas recortados correspondientes a cada uno de los conjuntos de datos que se pueden observar en las figuras 4.3a y 4.3b respectivamente. Estas listas permiten observar que tal como se había mencionado en la sección 3.4.1, el uso de palabras en inglés (tales como *flow*, *freestyle* o *baby*), deformaciones de ciertas palabras (así como "pa'" se usa en reemplazo de la palabra "para") y la inclusión de onomatopeyas (como es el caso de "yeah" o "ey", entre otras) es un factor muy presente tanto en las letras del género como en las batallas de *freestyle* específicamente. Esto necesariamente debe ser tenido en cuenta al momento de considerar cómo se busca generar rimas: la fonética de las palabras en inglés se debe analizar de forma distinta a la de las palabras en español; y al momento de utilizar técnicas de *transfer learning* como puede ser el uso de embeddings preentrenados, ya que estos probablemente no consideren los tokens correspondientes a las deformaciones de palabras ni a las onomatopeyas.



(a)



(b)

Figura 4.3: Palabras con mayor cantidad de apariciones sobre la totalidad del conjunto de datos (4.3a) y exclusivamente sobre las transcripciones de *freestyle* (4.3b)

4.2.2. Tokenización

Se consideraron tres formas posibles de tokenización, una a nivel de palabras, y dos a nivel de subpalabras de las cuales una fue a nivel de sílabas y otra utilizando el algoritmo de Byte Pair Encoding, descrito en la sección 3.1.2. No se tuvo en cuenta la tokenización a nivel caracteres por varios motivos: primero, resulta más complejo construir versos que rimen fijando caracteres al final de este que hacerlo fijando palabras, luego, al no ser unidades semánticas válidas no resulta de interés el uso de *embeddings* y por ende la utilización de *transfer learning* a través de estos. Por último, también se consideró que si bien ha habido trabajos que utilizaron este tipo de tokenización, la gran mayoría de los trabajos de NLP actualmente se inclinan por otros tipos.

Tokenización a nivel de palabras

La tokenización a nivel de palabras fue una de las primeras opciones por una cuestión de facilidad de implementación. Si bien se pudo realizar correctamente y se obtuvieron, sobre el total del conjunto de datos, 28424 tokens (palabras) distintos, se presentó el problema de que muchos de estos aparecen con muy poca frecuencia en dicho conjunto de datos. Del total de tokens distintos 12919 de estos aparecen una única vez, lo que equivale a un 45,45 % del total y si se consideran los que aparecen hasta 3 veces el número asciende a 19777, un 69,58 %. Si se evalúa esta situación en el conjunto de transcripciones de batallas de *freestyle* exclusivamente, el problema es aún mayor. Este conjunto de datos cuenta con 8681 tokens (palabras) distintos de los cuales 4723 (un 54,41 % del total) aparecen una única vez y 6847 de estos (un 78,87 %) aparecen a lo sumo 3 veces.

La situación observada respecto a la cantidad de tokens resulta en un problema significativo ya que es probable que el modelo de lenguaje utilizado no pueda obtener información útil a partir del conjunto de datos acerca de cómo representar y cómo utilizar las palabras representadas por estos tokens poco frecuentes, lo que implica que una parte significativa del conjunto de datos no pueda aprovecharse. De hecho, es común en trabajos del área de NLP el no tener en cuenta para la generación de los conjuntos de entrenamiento del modelo los tokens que aparecen con una frecuencia menor a un umbral determinado, lo que permite reducir el tamaño del vocabulario y descartar elementos de los que posiblemente representen información inútil. Sin embargo, con el porcentaje de tokens únicos presentes en el conjunto de datos mencionado, el descartarlos significa dejar de lado una porción significativa de los conjuntos de entrenamiento finales: si se busca generar, por ejemplo, pares de entrenamiento compuestos por sentencias de un largo de 5 tokens que se utilicen para predecir un sexto token objetivo, sobre el total del conjunto de datos es posible generar 579297 pares distintos, pero si se descartaran los que contienen tokens que aparecen una única vez eso resulta en descartar 69051 pares, un 11.92 % del total, mientras que si se descartan los que contienen tokens que aparecen a lo sumo 3 veces esto implica ignorar 145401 pares, un 25.10 % de la cantidad de pares de entrenamiento original. Si se tiene en cuenta exclusivamente la base de datos compuesta por transcripciones de batallas de *freestyle* se pueden generar 89360 pares con la forma descrita, de los que se descartarían 24960 (un 27.93 %) al ignorar los que contengan tokens que aparezcan una única vez y 45755 (un

51.20 %) si se hiciera esto con los que aparecen a lo sumo 3 veces, más de la mitad del conjunto de entrenamiento obtenido. La tabla 4.2 resume las observaciones mencionadas respecto al problema con los tokens de baja frecuencia al usar este tipo de tokenización y estas permiten concluir que la tokenización a nivel de palabras podría no ser ideal para este problema con este conjunto de datos en particular.

	BD con líricas de canciones		BD con batallas de *freestyle*	
	Cantidad	Porcentaje	Cantidad	Porcentaje
Total de tokens	28424	100 %	8681	100 %
Tokens que aparecen una única vez	12919	45,45 %	4723	54,41 %
Tokens que aparecen a lo sumo 3 veces	19777	69,58 %	6847	78,87 %
Total de pares de entrenamiento con secuencias de largo 5	579297	100 %	89360	100 %
Pares descartados si poseen tokens con frecuencia 1	69051	11,92 %	24960	27,93 %
Pares descartados si poseen tokens con frecuencia menor o igual a 3	145401	25,10 %	45755	51,20 %

Tabla 4.2: Resumen de las observaciones realizadas respecto a la cantidad de tokens con bajas frecuencias (de a lo sumo 1 o 3) y cantidad de pares de entrenamiento generados que los incluyen al utilizar tokenización a nivel de palabras.

Algunas de las posibles razones que se consideran pueden causar este problema de gran cantidad de palabras con poca frecuencia de aparición son:

- En primer lugar, el tamaño de los conjuntos de datos utilizados. Si bien se puede considerar que la base de datos que contiene la totalidad de las canciones del género contiene una cantidad considerable de estas, al momento de compararlo con los conjuntos de datos utilizados para el entrenamiento de otros modelos de lenguaje de propósito general estos suelen estar compuestos de una cantidad de texto considerablemente mayor, como lo puede ser la totalidad de Wikipedia o gran cantidad de texto periodístico accesible a través de internet. Respecto a la base de datos exclusiva de batallas de *freestyle*, como ya se mencionó anteriormente, obtener los datos no resultó un aspecto trivial, lo que significó una base de datos significativamente menor. En cualquiera de los casos, se espera que al aumentar el tamaño del conjunto de datos habría una mejora en este problema puntual.
- Otro punto a tener en cuenta es el de posibles errores de ortografía o distintas formas de escribir ciertas expresiones u onomatopeyas. En el listado de palabras más utilizadas mostrado en la sección 4.2.1 se puede observar la presencia tanto del token “yeah” como de “yeh”, que podrían en algunos casos estar refiriéndose a la misma expresión. También se puede ver la utilización muy marcada del token “pa'”, en reemplazo de la palabra “para”,

pero esta palabra también es utilizada en muchos de los textos e incluso en algunos se puede encontrar el token "pa" (sin el apóstrofe final) utilizado con el mismo significado. Esto resulta en que la frecuencia total de la palabra "para" quede repartida entre tres tokens distintos.

- Por último, también es necesario considerar la misma naturaleza del género. Puntualmente en las batallas de *freestyle* uno de los desafíos en particular consiste en incluir en los versos improvisados una palabra dada. Esta puede que justamente sea una palabra ajena al contexto de la batalla y no sea utilizada frecuentemente para que resulte más complejo para el ejecutante incluirla. Entonces esto también podría resultar potencialmente en la aparición de tokens con una frecuencia muy baja a lo largo de todo el conjunto de datos, que se corresponden con estas palabras incluidas forzosamente durante una ejecución.

Tokenización a nivel de sílabas

Una segunda forma de tokenización que se utilizó fue a nivel de sílabas. Esta forma de tokenización no complejizaba tanto la selección de tokens que rimen entre sí como podría hacerlo la tokenización a nivel caracteres y permitió obtener una mejora respecto a la problemática de los tokens con poca frecuencia, sin embargo, no significó una solución al problema. Luego de someter la totalidad del conjunto de datos a este tipo de tokenización se obtuvieron 4807 tokens distintos, de los cuales 1170 aparece tan solo una vez (el 24,34 % del total) y 2071 aparecen a lo sumo 3 veces (el 43,06 %). Al utilizar únicamente el conjunto de datos con transcripciones de batallas de *freestyle* se obtienen 1911 tokens distintos con 479 de estos apareciendo una única vez entre los textos (un 25,07 %) y 882 apareciendo como máximo 3 veces (un 46,15 %). En ambos conjuntos de datos se ve una mejora de los porcentajes de tokens con baja frecuencia respecto a la tokenización a nivel de palabras, pero los valores aún son significativamente altos.

Respecto a la cantidad de pares que es posible generar descartando estos tokens los números si mejoran significativamente: si se generan estos pares de la misma forma que la que se describió en la sección anterior sobre el total de los datos con una ventana deslizante de tamaño 5 es posible generar 1351634 secuencias, lógicamente, una cantidad mucho mayor a la observada al tokenizar al nivel de palabras (más del doble). De estas secuencias 6936 contienen tokens que aparecen una única vez y el 18523 contienen tokens que aparecen a lo sumo 3 veces, un 0,51 % y un 1,37 % del total, respectivamente. Al realizar este mismo análisis ahora sobre la base de datos que contiene exclusivamente transcripciones se obtienen 213538 pares posibles, de los cuales 2698 (un 1,26 %) contienen tokens que aparecen una única vez y 7915 (un 3,71 %) contienen tokens con frecuencia de a lo sumo 3. Si bien los números obtenidos muestran una mejora significativa, hay que considerar que al estar utilizando tokenización a nivel de sílabas el tamaño de las secuencias a considerar para predecir el token siguiente debiera ser mayor al utilizado cuando los tokens representan palabras completas, ya que en este caso cada uno brinda menos información. Teniendo en cuenta que el total de tokens obtenidos al tokenizar el total del conjunto de datos a nivel sílaba es 2,33 veces más grande que el obtenido al tokenizarlo a nivel de palabra, se puede considerar que cada token que representa a una palabra se traduce, en promedio, en 2,33 tokens a nivel de sílaba (esto no es exacto ya que hay que considerar factores

como que para la tokenización a nivel de sílaba se utilizan tokens especiales que no son necesarios cuando se tokeniza a nivel de palabras, como el del espacio en blanco '\ '). Entonces, resulta más adecuado para la comparación analizar la utilización de un tamaño de secuencia que represente la misma información que una secuencia de 5 tokens a nivel de palabras: $5 * 2,33 \approx 12$. Al utilizar secuencias de este largo es posible, sobre el total del conjunto de datos, generar 1351627 pares de entrenamiento, de los que 14703 (un 1,09 %) contiene tokens con una única aparición y 38548 (un 2,85 %) contiene tokens con a lo sumo tres apariciones, mientras que al trabajar exclusivamente sobre las transcripciones de batallas de *freestyle* 213533 pares con 4858 de estos conteniendo tokens con una frecuencia de 1 y 13986 que contienen tokens con frecuencia de a lo sumo 3 (el 2,28 % y 6,55 % del total, respectivamente). Observando estos resultados, al utilizar este tipo de tokenización si parece una decisión más adecuada el no tener en cuenta estos tokens de baja frecuencia ni las secuencias que los contienen para generar los pares de entrenamiento del modelo. La tabla 4.3 resume las observaciones realizadas respecto a este tipo de tokenización.

	BD con líricas de canciones		BD con batallas de *freestyle*	
	Cantidad	Porcentaje	Cantidad	Porcentaje
Total de tokens	4647	100 %	1877	100 %
Tokens que aparecen una única vez	1208	26,00 %	465	24,77 %
Tokens que aparecen a lo sumo 3 veces	2086	44,89 %	858	45,71 %
Total de pares de entrenamiento con secuencias de largo 5	1351634	100 %	213538	100 %
Pares descartados si poseen tokens con frecuencia 1	6936	0,51 %	2698	1,26 %
Pares descartados si poseen tokens con frecuencia menor o igual a 3	18523	1,37 %	7915	3,71 %
Total de pares de entrenamiento con secuencias de largo 12	1351627	100 %	213533	100 %
Pares descartados si poseen tokens con frecuencia 1	14703	1,09 %	4858	2,85 %
Pares descartados si poseen tokens con frecuencia menor o igual a 3	38548	2,28 %	13986	6,55 %

Tabla 4.3: Resumen de las observaciones realizadas respecto a la cantidad de tokens con bajas frecuencias (de a lo sumo 1 o 3) y cantidad de pares de entrenamiento generados que los incluyen al utilizar tokenización a nivel de sílabas.

Tokenización utilizando BPE

Por último se utilizó el algoritmo de Byte Pair Encoding (sección 3.1.2), que permite tokenizar un texto identificando cuales son los tokens que aparecen con mayor frecuencia, por lo que se consideró que podía resultar de utilidad teniendo en cuenta el problema presentado en las dos formas anteriores de tokenización. Al utilizar este algoritmo se debe definir un tamaño de vocabulario deseado y a través de este se busca obtener una cantidad total de tokens distintos lo más cercana posible a dicho tamaño. Se probaron distintos tamaños de vocabulario: 20000 (valor cercano a la cantidad de tokens obtenida al tokenizar a nivel de palabras), 10000, 5000 (valor cercano a la cantidad de tokens obtenida al tokenizar a nivel de sílabas), 3000 y 1000. En todos los casos se logró una mejora significativa en lo que refiere a la cantidad de tokens con baja frecuencia, aunque esta es más notoria cuando se utiliza un tamaño de vocabulario menor. Esto último resulta esperable ya que con un menor tamaño de vocabulario los tokens suelen ser más pequeños, es decir, más cercanos al nivel de caracter, mientras que al usar un vocabulario más grande estos se vuelven más complejos y se suelen corresponder con palabras completas. En la figura 4.4 se ilustra un ejemplo de como resultan estos distintos niveles de tokenización para una frase extraída del conjunto de datos. Se utilizó además un token especial "[SEP]" indicando el final de cada verso, con el objetivo de que el modelo luego sea capaz de aprender cuándo finalizar cada uno sin utilizar una longitud fija.

"En la improvisación
Lamentablemente yo muerdo como león"

Tamaño de vocabulario = 1000:
['_en', '_la', '_improvis', 'a', 'ción', '[SEP]', '_la', 'm', 'enta', 'ble', 'mente', '_yo', '_muer', 'do', '_como', '_le', 'ón', '[SEP]']

Tamaño de vocabulario = 5000:
['_en', '_la', '_improvisa', 'ción', '[SEP]', '_lamenta', 'ble', 'mente', '_yo', '_muer', 'do', '_como', '_le', 'ón', '[SEP]']

Tamaño de vocabulario = 20000:
['_en', '_la', '_improvisación', '[SEP]', '_lamentablemente', '_yo', '_muer', 'do', '_como', '_león', '[SEP]']

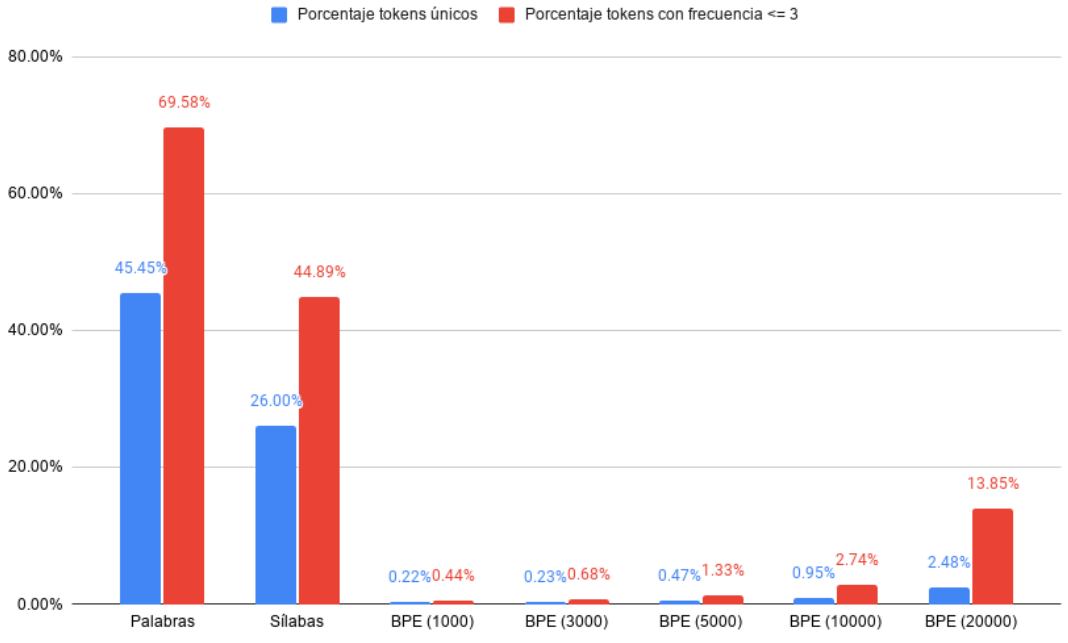
Figura 4.4: Ejemplo de tokenización sobre una frase extraída del conjunto de datos utilizando BPE con distintos tamaños de vocabulario.

Para la evaluación de este algoritmo, en primer lugar, se ejecutó sobre la totalidad del conjunto de datos con cada uno de los diferentes tamaños de vocabulario mencionados. Esto resulto en 5 tokenizadores distintos, cada uno con su conjunto de tokens que se corresponden con un respectivo tamaño de vocabulario. Luego, con cada uno de los tokenizadores y conjuntos de tokens definidos se evaluó la cantidad de tokens que aparecen una única vez y los que aparecen a lo sumo 3 veces sobre el total del conjunto de datos y sobre las transcripciones de letras de batallas respectivamente. Dado entonces que los tokens fueron definidos sobre la totalidad del conjunto de datos, además de por las razones ya mencionadas anteriormente, el que la cantidad

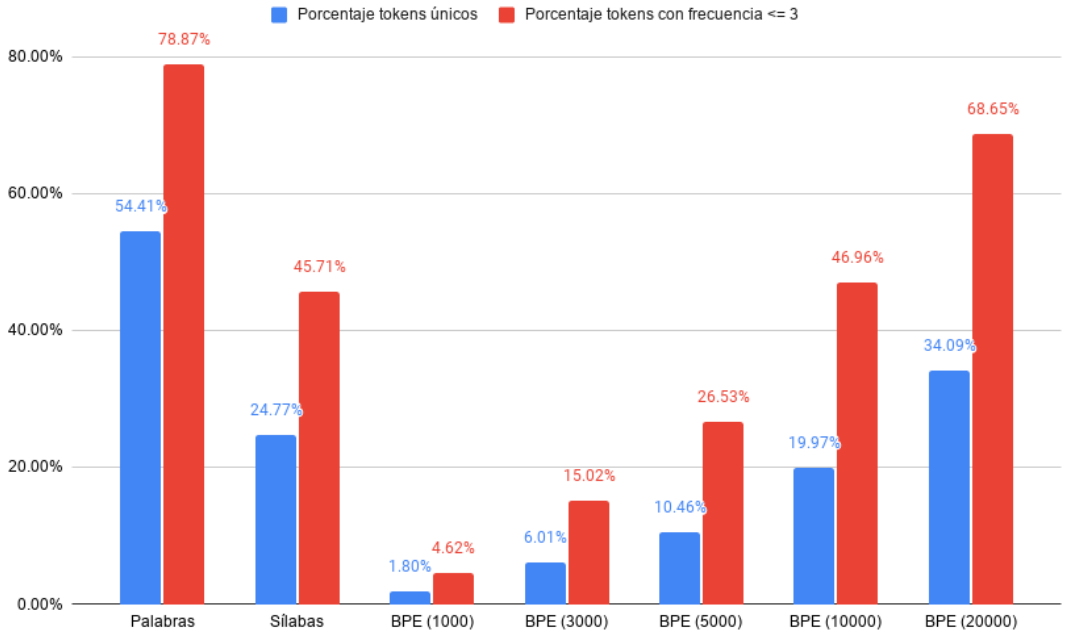
de tokens únicos aumente al evaluar este algoritmo únicamente sobre las transcripciones de batallas de *freestyle* resulta razonable.

En las figuras 4.5a y 4.5b se muestra el porcentaje de tokens que aparecen una única vez y los que aparecen a lo sumo 3 veces en la totalidad del conjunto de datos y sobre las transcripciones de batallas de *freestyle* respectivamente para los distintos niveles de tokenización y la aplicación de BPE con distintos tamaños de vocabulario. Como se mencionó anteriormente y resulta esperable, los porcentajes más bajos se obtuvieron al utilizar BPE con tamaños de vocabulario pequeños, mientras que a medida que se aumenta dicho tamaño los valores se acercan más a los obtenidos al tokenizar a nivel palabras o sílabas, sobre todo al evaluarlos sobre la base de datos que contiene exclusivamente las transcripciones de batallas de *freestyle*.

Para realizar el análisis sobre los pares de entrenamiento que se pueden generar utilizando los distintos tamaños de vocabulario bajo esta forma de tokenización se utilizó el mismo criterio que para la tokenización a nivel de sílabas: se utilizaron distintos tamaños de secuencias en función de que tanto mayor sea la cantidad de tokens resultantes al tokenizar el texto con este tamaño de vocabulario respecto a la cantidad de tokens al tokenizarlo a nivel de palabra, para la que se utilizó un tamaño de secuencia de 5. Los resultados de este análisis para cada uno de los tamaños de vocabulario utilizados se pueden observar en las tablas 4.4a y 4.4b para la totalidad del conjunto de datos y las transcripciones de batallas de *freestyle* exclusivamente. Finalmente, las figuras 4.6a y 4.6b ilustran los porcentajes de secuencias que contienen tanto tokens únicos como tokens con frecuencia menor a 3 para ambos conjuntos de datos respectivamente. Como era de esperarse y de forma análoga a lo que sucede en el caso del porcentaje de tokens con baja frecuencia, los porcentajes más bajos respecto a la cantidad de pares de entrenamiento que contienen tokens de baja frecuencia se obtuvieron al tokenizar el texto utilizando BPE con tamaños de vocabulario pequeños. Sin embargo, en esta estadística la tokenización a nivel de sílabas permitió también obtener buenos resultados: los porcentajes al utilizar esta técnica fueron similares a los obtenidos al utilizar BPE con un vocabulario de tamaño 20000 al compararlo sobre el total del conjunto de datos y significativamente mejores que los obtenidos al utilizar BPE con vocabularios mayores que 5000 sobre el conjunto de datos que contiene únicamente transcripciones de batallas de *freestyle*.



(a)



(b)

Figura 4.5: Porcentaje de tokens que aparecen una única vez y tokens con a lo sumo 3 apariciones sobre el total de tokens distintos en la totalidad del conjunto de datos (4.5a) y exclusivamente sobre las transcripciones de *freestyle* (4.5b) al aplicar tokenización en distintos niveles.

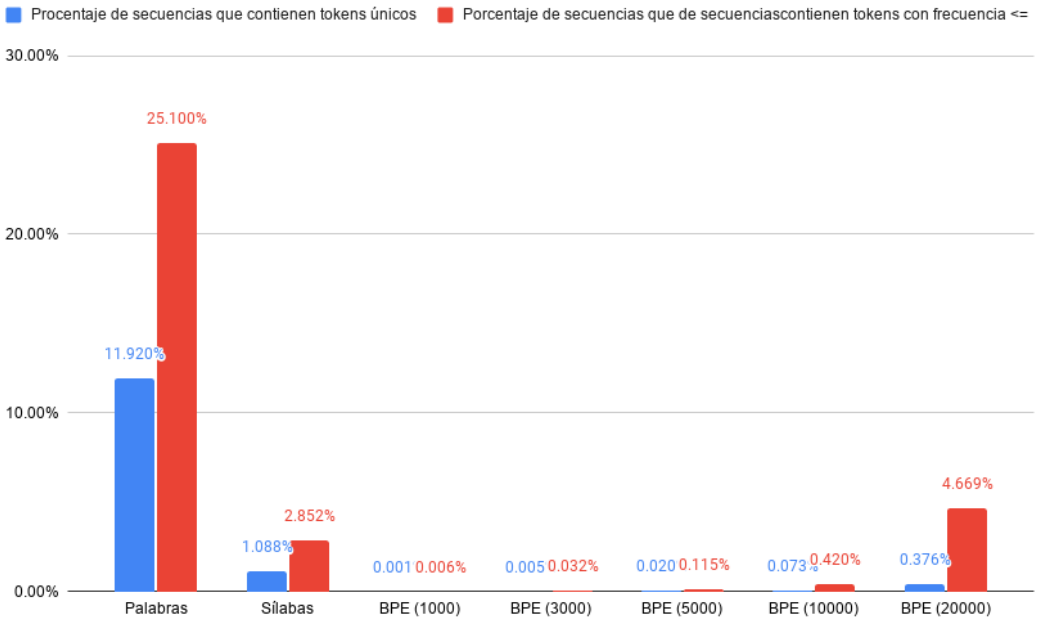
Tamaño de vocabulario	1000	3000	5000	10000	20000
Relación entre el tamaño de la tokenización del texto y la tokenización a nivel de palabras	151,82 %	125,28 %	116,47 %	106,69 %	99,48 %
Largos de secuencia	8	6	6	5	5
Total pares generados	879505	725769	674686	618032	576289
Pares que contienen tokens únicos	13	39	137	454	2164
Pares que contienen tokens con frecuencia <= 3	49	235	776	2595	26909
Porcentaje de pares que contienen tokens únicos	0.001 %	0.005 %	0.020 %	0.073 %	0.376 %
Porcentaje de pares que contienen tokens con frecuencia <= 3	0.006 %	0.032 %	0.115 %	0.420 %	4,669 %

(a)

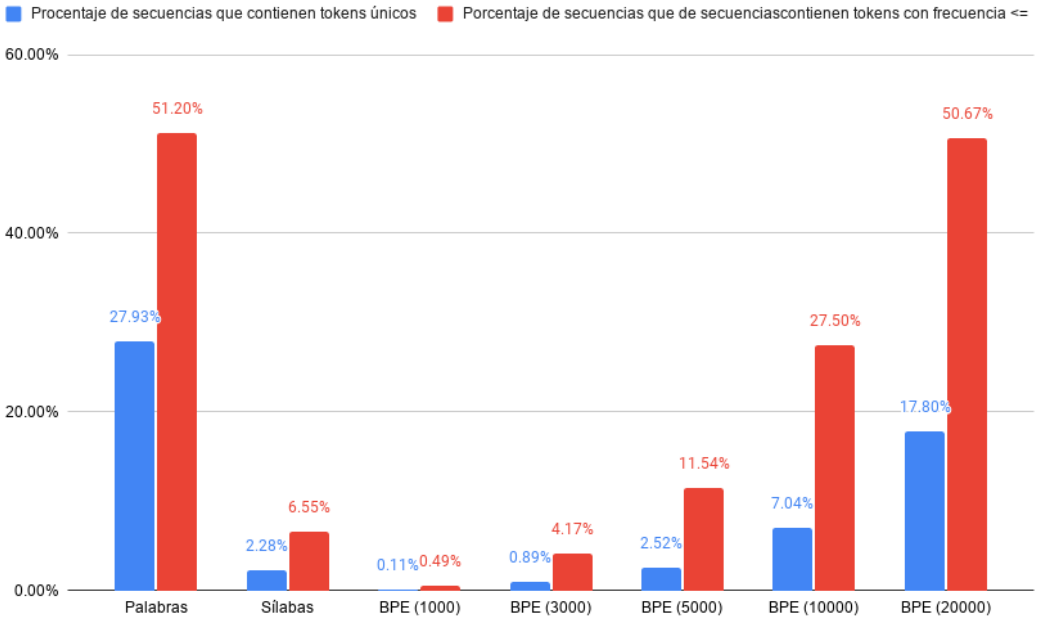
Tamaño de vocabulario	1000	3000	5000	10000	20000
Relación entre el tamaño de la tokenización del texto y la tokenización a nivel de palabras	151,82 %	125,28 %	116,47 %	106,69 %	99,48 %
Largos de secuencia	8	6	6	5	5
Total pares generados	134277	111851	104418	96406	89694
Pares que contienen tokens únicos	144	994	2633	6790	15970
Pares que contienen tokens con frecuencia <= 3	661	4662	12045	26513	45444
Porcentaje de pares que contienen tokens únicos	0,11 %	0,89 %	2,52 %	7,04 %	17,80 %
Porcentaje de pares que contienen tokens con frecuencia <= 3	0,49 %	4,17 %	11,54 %	27,50 %	50,67 %

(b)

Tabla 4.4: Resumen del análisis respecto al problema con los tokens de baja frecuencia en la cantidad de pares de entrenamiento posibles al utilizar tokenización con BPE y diferentes tamaños de vocabulario sobre la totalidad del conjunto de datos (4.4a) y sobre las transcripciones de batallas de *freestyle* (4.4b).



(a)



(b)

Figura 4.6: Porcentaje de pares de entrenamiento que contienen tokens que aparecen una única vez y tokens con a lo sumo 3 apariciones sobre el total de pares de entrenamiento en la totalidad del conjunto de datos (4.6a) y exclusivamente sobre las transcripciones de *freestyle* (4.6b) al aplicar tokenización en distintos niveles.

4.3. Modelo de generación de texto

Para la generación de texto se utilizó un modelo de lenguaje que consiste en una red neuronal recurrente compuesta por varias capas que utilizan conceptos desarrollados en los capítulos 2 y 3. En primer lugar, la red recibe como entrada un vector de enteros que se corresponden con los índices de una secuencia de tokens del vocabulario. El largo de la secuencia es un hiperparámetro configurable de la red y se realizaron experimentos utilizando distintos valores de este en combinación con distintas formas de tokenización descritos en las secciones siguientes. Dicha secuencia es entonces procesada como entrada de una capa de *embeddings*, que mapea cada índice con sus respectivos vectores de *embeddings* (descritos con más detalle en la sección 3.2.3) de dimensión 300, otro hiperparámetro de la red definido por ser dicho valor utilizado como un valor estándar en otros trabajos de NLP. Los valores de dichos vectores se ajustan durante el entrenamiento de la red neuronal a la vez que el resto de los pesos de la red.

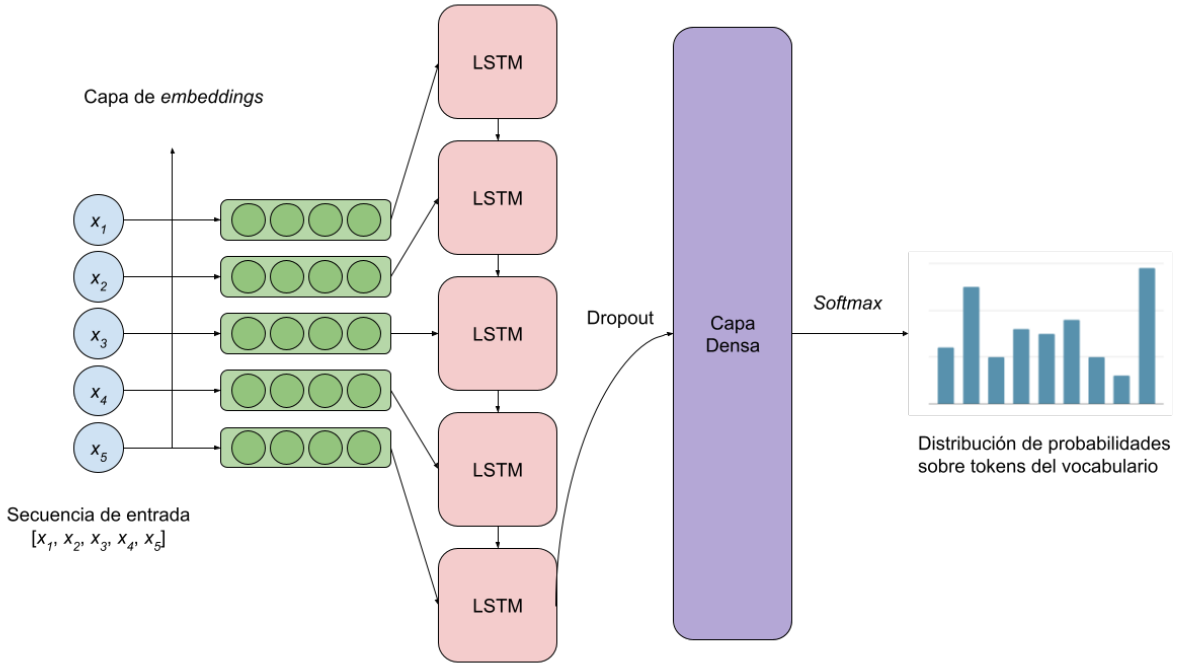


Figura 4.7: Representación gráfica del modelo utilizado para un tamaño de secuencias de 5.

Luego, la secuencia de vectores resultante es utilizada como entrada de una capa LSTM bidireccional con 256 unidades, cuya arquitectura se detalla en la seccion 2.5.2. Se consideró importante que la capa sea bidireccional ya que, como se desarrolla más adelante en la sección 4.5, para entrenar el modelo se utilizan secuencias de texto en orden inverso a como se encuentran naturalmente. Entonces, las unidades LSTM de la capa bidireccional que procesen la secuencia inversa serán las que en realidad sean capaces de captar las dependencias en el orden correcto del texto. A continuación se utilizó una capa conocida como Dilución o *Dropout* descrita en

2.4.3 para evitar el sobreajuste del modelo, que en lo que respecta a modelos de generación de texto se traducirá en generar únicamente texto similar al ya existente en el conjunto de datos, en lugar de generalizar correctamente para la generación de nuevos textos.

Por último, la capa de salida del modelo es una capa densa que debe poseer tantas neuronas como el tamaño del vocabulario, por lo que para diferentes experimentos con tamaños de vocabularios distintos se deberá generar un modelo acorde a cada uno. Sobre esta capa se utiliza como función de activación la función *softmax* (detallada en la seccion 2.1.2) que permite intepretar la salida de la red como una distribución de probabilidades entre los distintos tokens del vocabulario. En la figura 4.7 se puede ver una representación grafica del modelo descrito.

4.4. Entrenamiento del modelo de lenguaje

Utilizando el modelo de lenguaje descrito se realizaron experimentos con distintos tipos de tokenización y largos de secuencia: a nivel de palabras con secuencias de 5 tokens de largo, a nivel de sílabas con 8 secuencias de tamaño 8 y utilizando BPE con vocabulario de tamaño 3000 y 5000 con secuencias de tamaño 6 y 5 respectivamente. No se utilizó BPE con tamaños de vocabulario mayores a 5000 ya que, como se mostro en la sección 4.2.2, en estos tamaños de vocabulario vuelve a significar un problema la presencia de tokens con baja frecuencia. Tampoco se utilizó un tamaño de vocabulario de 1000 ya que las tokenizaciones obtenidas con este tamaño resultaban en tokens demasiado sencillos, muy cercanos al nivel de caracteres como también se mostró en la sección 4.2.2.

Para el entrenamiento del modelo en cada experimento se generaron dos conjuntos de datos, uno con cada una de las bases de datos disponibles. Para esto se tokenizó la totalidad de cada una de las bases de datos y se reemplazó cada token por su correspondiente índice. Luego, se ejecutó sobre dichas secuencias de índices un algoritmo de ventana deslizante para generar todos los pares de entrenamiento x, y donde x es un vector correspondiente a una secuencia de índices con el cual se debe predecir la palabra y, representada de forma *one-hot* en un vector del tamaño del total del vocabulario. De este conjunto generado se utilizó el 80 % como conjunto de entrenamiento y el 20 % como conjunto de prueba. Luego, se preentrenó el modelo sobre el conjunto de datos generado a partir del total de las canciones y por último se realizó *fine-tuning* entrenándolo sobre el conjunto compuesto exclusivamente por batallas de *freestyle*. La función de error utilizada durante el entrenamiento del modelo fue la Entropía Cruzada Categórica (sección 2.2.1) y se utilizaron lotes de datos de tamaño 256.

Los entrenamientos se llevaron a cabo en parte sobre una computadora local y utilizando tambíen la plataforma Colaboratory de Google[5], que permite la ejecución remota de modelos de *deep learning* sobre *hardware* de gran potencia. El hardware disponible localmente fue una GPU NVIDIA GTX 1050 TI que cuenta con 768 cores y 8 GB de memoria, y una CPU Intel Core i5-8400 de 6 cores, también con 8 GB de memoria y sobre la que se ejecuta un sistema operativo Windows 10, mientras que el hardware accesible a través de Google Colaboratory

[5] https://colab.research.google.com/

consiste en una GPU Nvidia K80 de 12 GB o una Nvidia T4 de 16 GB (varía en cada ejecución según disponibilidad) y una CPU Intel(R) Xeon(R) que cuenta con 2 cores y 12 GB de memoria RAM. Lógicamente, por las características del *harware* el entrenamiento de modelos sobre la plataforma Collaboratory resulta significativamente más rápido que sobre el equipo local, sin embargo dicha plataforma presenta límites de tiempo para el uso de la totalidad de sus recursos de *hardware*, razón por la cual se utilizaron ambas opciones.

La tabla 4.5 presenta una comparativa de la exactitud y tiempo de entrenamiento de los modelos entrenados con diferentes parámetros de tokenización correspondientes a cada experimento. Respecto al análisis de los tiempos requeridos se debe tener en cuenta que el entrenamiento de un experimento llevado a cabo en la plataforma Google Colab, como ya se lo mencionó, será necesariamente mucho menor al de otro llevado a cabo localmente, por lo que se debe tener en cuenta ese factor al momento de comparar esta característica entre dos experimentos llevados a cabo en plataformas distintas. Sin embargo resulta interesante notar que el tiempo requerido para el entrenamiento del modelo utilizado es significativo, aún disponiendo localmente de *hardware* específico como lo es una GPU. Esto se debe considerar al momento de llevar a cabo un entrenamiento dado que, como se mencionó anteriormente, si bien el uso de plataformas tales como Colaboratory pueden facilitar y acelerar significativamente el proceso, estas poseen un límite para el uso de sus recursos.

Respecto a la eficacia obtenida en cada uno de los experimentos, es posible notar en primer lugar que la utilización de *fine-tuning* resultó efectiva para todos los experimentos realizados: el preentrenamiento sobre la base de datos que contiene canciones de rap y géneros similares permitió que se alcancen, con una cantidad de épocas de entrenamiento mucho menor altos niveles de eficacia sobre la base de datos que compuesta únicamente por transcripciones de *freestyle*. El preentrenamiento consistió en 40 épocas para los experimentos que utilizaron tokenización con BPE con vocabularios de tamaño 3000 y 5000, en las que alcanzaron una eficacia de 85,98 % y 82,58 % sobre el conjunto de entrenamiento, respectivamente. Para el modelo entrenado sobre una tokenización a nivel de sílabas se requirieron más épocas para alcanzar niveles de eficacia similares, se realizaron 80 épocas con las que alcanzo un 78,14 % de eficacia. Por último, el modelo entrenado bajo tokenización a nivel de palabras fue el que obtuvo una mayor eficacia entrenado con una cantidad de épocas algo mayor que la utilizada para los experimentos con tokenización utilizando BPE: alcanzó una eficacia del 95,35 % luego de 50 épocas. Luego del preentrenamiento, para todos los experimentos realizados bastaron entre un cuarto o un quinto de la cantidad de épocas de entrenamiento sobre el conjunto de transcripciones de *freestyle* para alcanzar una eficacia incluso mayor a la lograda en el entrenamiento sobre este otro conjunto de datos. Los modelos de los experimentos donde se utilizo BPE con tamaño de vocabulario de 3000 y 5000 alcanzaron una eficacia del 96,28 % y 95,89 % luego de 10 y 12 épocas respectivamente. En el experimento que utilizó tokenización en base a sílabas se obtuvo luego de 20 épocas de entrenamiento una eficacia del 85,65 % y por último, el modelo entrenado bajo una tokenización a nivel de palabras alcanzó una eficacia del 97,71 % tras entrenar solo por 10 épocas.

Resulta también llamativa la diferencia entre la eficacia obtenida en el conjunto de entrena-

miento y el conjunto de prueba para todos los experimentos realizados. Esta diferencia está presente en todos los experimentos, en mayor medida en el entrenamiento sobre el conjunto de transcripciones de *freestyle* respecto al preentrenamiento sobre el conjunto de canciones de géneros similares. Este comportamiento es típicamente asociado al sobreajuste (*overfitting*) del modelo, no obstante hay que tener en cuenta que la situación descrita en la subsección 4.2 acerca de, en el caso de la tokenización a nivel de palabras, la existencia de una gran cantidad de tokens con poca frecuencia de aparición en el conjunto de datos puede ser también una causa directa de esto: es probable que varios de los tokens únicos o de baja frecuencia formen parte de las secuencias que componen el conjunto de prueba. Esto resultará en que el modelo no pueda aprender acerca del uso de esos tokens a partir del conjunto de entrenamiento. En el caso de la tokenización a nivel de subpalabras, estas palabras de baja frecuencia se corresponderán a varios tokens que componen secuencias que aparecerán una baja cantidad de veces, resultando probablemente en el mismo fenómeno. De cualquier manera, se cree que una mayor cantidad de datos de entrenamiento resulta necesario para poder lograr una mejora en estos resultados.

Nivel de tokenización	Palabras	Sílabas	BPE(3000)	BPE(5000)
Tamaño de secuencia	5	8	6	5
Plataforma utilizada	Colaboratory	Colaboratory	Local	Local
Entrenamiento sobre conjunto de canciones de géneros de rap o similares				
Cantidad de épocas	50	80	40	40
Tiempo promedio de cada época (s)	215	154	1912	1399
Tiempo total de entrenamiento (h)	2:59	3:25	21:25	15:32
Eficacia (*accuracy*) sobre conjunto de entrenamiento	95,35 %	78,14 %	85,98 %	82,58 %
Eficacia (*accuracy*) sobre conjunto de prueba	42,42 %	60,89 %	57,28 %	52,06 %
Entrenamiento sobre el conjunto de transcripciones de *freestyle*				
Cantidad de épocas	10	20	12	10
Tiempo promedio de cada época (s)	41	21	292	216
Tiempo total de entrenamiento (h)	0:06	0:07	0:58	0:36
Eficacia (*accuracy*) sobre conjunto de entrenamiento	97,71 %	85,65 %	96,28 %	95,89 %
Eficacia (*accuracy*) sobre conjunto de prueba	40,31 %	59,81 %	55,51 %	51,12 %

Tabla 4.5: Comparativa de la exactitud y tiempo de entrenamiento de los modelos entrenados con diferentes parámetros de tokenización.

4.5. Generación de texto estructurado

Una vez entrenado el modelo, este se puede utilizar como un sistema generador de texto tal como se describe en la sección 3.3, a partir de un texto inicial S llamado semilla se repite tantas veces como tokens a generar: se ingresa S como secuencia de entrada a la red, se obtiene el siguiente token de forma aleatoria a partir de la distribución de probabilidades generado por el modelo y se concatena a la secuencia S. Los primeros experimentos de generación de texto fueron llevados a cabo de esta forma y si bien el modelo entrenado logra así generar texto que replica efectivamente el estilo de las letras presentes en las bases de datos, hay algunos factores que no es capaz de captar correctamente para plasmar en el texto generado como lo son la rima y la cantidad de versos de cada estrofa. En la figura 4.8 se puede observar una porción de texto generado de la forma descrita en el cual es posible ver lo anteriormente enunciado:

- Si bien el modelo aprendió de cierta forma la organización del texto en estrofas generando cada cierta cantidad de tokens dos saltos de linea consecutivos, estos aparecen de forma irregular dando lugar a estrofas de diferente largo (en el caso mostrado se generaron estrofas con 3, 4 y 2 versos respectivamente), cuando estas debieran estar fijas en 4 versos.
- Si bien se puede dar eventualmente, no se garantiza que siempre las estrofas generadas cumplan con una estructura de rima. Para el texto mostrado, por ejemplo se puede encontrar una cierta estructura de rima en la segunda estrofa generada (que tiene la forma ABBB), pero claramente no es el caso para la tercera, cuando es algo que debiera suceder para todas.

Eso es verdad, es es que esto está difícil
Pero ey, yo te pateo cuando vengo te noqueo
Porque soy un teólogo, un psicólogo y porque vengo con empeño

Aparte, yo sé que tengo nivel
Mientras este se cree pablo neruda
Así que viene con su pluma
Pero yo hablo con la mente desnuda

Sé quien soy, ven como juego?
Ves lo que hizo este ingenio, tené cuidado

Figura 4.8: Porción de texto generada con una temperatura de 1.5 utilizando el modelo entrenado con texto tokenizado utilizando BPE con tamaño de vocabulario 5000 y tamaño de secuencia 5. El texto se modificó para capitalizar la primer letra de cada verso, ya que el modelo fue entrenado únicamente con caracteres en minúscula.

Además, analizando el caso puntual de la segunda estrofa generada se encontraron en el conjunto de datos de entrenamiento dos estrofas distintas que utilizan palabras y frases similares a la generada, las cuales se pueden observar en las figuras 4.9a y 4.9b. Esto permite suponer que

el modelo identificó que estas palabras se encontraban comúnmente próximas entre si, pero no implica que se pueda generalizar a que haya aprendido patrones de rimas.

Yo me siento pablo neruda
Porque hablo con la mente desnuda
A el todavía le quedan dudas
De que a mis punchlines los escribo en plumas

(a)

Eso es lo pasa, me viene el stuart con dudas
No se compra un "tua", así que viene con su pluma
Se cree pablo neruda
Pero yo traje certeza, mientras este trae dudas

(b)

Figura 4.9: Estrofas extraídas del conjunto de datos de entrenamiento que presentan similitudes en las palabras y su uso respecto a la segunda estrofa generada en la figura 4.8.

Teniendo en cuenta las observaciones realizadas y siendo que estos son aspectos que pueden validarse correctamente sobre el texto a medida que se genera es que se decidió modificar el algoritmo de generación para garantizar que el texto cumpla con estos por fuera del modelo neuronal.

4.5.1. Organización del texto en estrofas

La primera modificación que se realizó fue con la finalidad de garantizar que las estrofas contengan la cantidad de versos adecuados. En cualquiera de las formas de tokenización definidas, implementar este control resultó trivial gracias a la utilización de tokens que indican el final de cada verso (representados por el token "[SEP]" en la tokenización utilizando BPE y el token de salto de linea "\n" en las otras formas de tokenización). Esto permite identificar cuando el modelo completa la generación de una estrofa simplemente contabilizando la cantidad de tokens de fin de verso generados hasta que esta llegue a 4. Entonces, se modificó el algoritmo de generación de texto para que en lugar de que este permita definir una cantidad arbitraria de tokens a generar, únicamente permita definir una cantidad de estrofas las cuales se contabilizan de la manera descrita. En la figura 4.10 se ven ejemplos de texto generado luego de esta modificación utilizando los mismos hiperparámetros que los ya descritos para la obtención del texto de la figura 4.8. Se puede ver claramente como el texto generado se encuentra ahora estructurado en estrofas, particularmente se tomaron 3 de las generadas. Sin embargo, como era de esperarse, la aparición de versos que rimen continúa siendo una situación esporádica ya que no se garantiza que las estrofas se adecúen a una estructura de rima en particular.

4.5.2. Generación de texto bajo estructura de rima

Respecto al garantizar que cada uno de los versos cumpla con la estructura de rima correspondiente, este no resulta un problema tan trivial como el de que las estrofas cumplan con la cantidad de versos correspondientes. Se identificaron 4 razones particulares que ilustran la dificultad de este problema:

1. En primer lugar, identificar cuándo dos palabras riman no es una tarea tan trivial como contabilizar un token en particular. Es necesario contemplar varios casos posibles de

En cada rima, soy de los mejores
Y le doy gracias a las batallas
Por hacerme ser alguien inteligente
Que sabe pararse de frente

Sabes que pasa? Que no tenés fluidez
Te lo aclaro, mi hermano
Nadie confía en vos
Nadie quiere saber de tu improvisación

Y bueno de verdad, lo mato en la improvisación
Te falta aprender que tengo creación
Porque no tenés el nivel
Te subes a mi montaña rusa de este parque de diversión

Figura 4.10: Texto generado luego de la modificación del algoritmo de generación para que este se agrupe adecuadamente en estrofas.

acentuación de cada una de las palabras en conjunto con reglas particulares de la lengua para identificar sus respectivas vocales tónicas, a partir de las cuales se tendrán en cuenta el resto de los caracteres para analizar el tipo de rima. Esto se complejiza aún más cuando se debe considerar la aparición de palabras en inglés o expresiones u onomatopeyas que no siguen las reglas correspondientes de acentuación. Por ejemplo, la expresión "yeah", tomada del inglés debiera escribirse en español como "yea", ya que caso contrario debería poseer una tilde en la e por ser una palabra grave que no termina en n, s o vocal. Sin embargo, esto no sucede y típicamente se encuentra escrita con hache al final y sin tilde al igual que en inglés.

2. Luego, para modificar el texto generado para que se ajuste a una estructura de rima en particular es necesario trabajar sobre la generación de la última palabra de cada verso. Esto conlleva la dificultad de que, al no tener una cantidad de palabras/sílabas fijas por verso y elegirse la palabra a continuación de forma aleatoria sobre la distribución de probabilidades predicha por el modelo, no es posible identificar cuál será esta última palabra hasta ya generada tanto la palabra como el token siguiente, el cual debiera corresponderse con el token de fin de verso ("[SEP]" o "\n").

3. Además, si se busca ajustarse a una estructura de rima modificando la última palabra de un verso luego de haber generado las anteriores puede dar lugar a la pérdida de coherencia en el texto generado. Tomando como ejemplo un verso extraído del conjunto de datos, si el modelo hubiera generado la frase "yo muerdo como león" pero por la estructura de rima generada se debiera ajustar a que rime con un verso en particular con una terminación distinta, el reemplazar la última palabra por otra exclusivamente para que cumpla con la estructura de rima probablemente generaría que el verso final no resultara coherente. Esto se debe a que el contexto en que se encuentra esta última palabra limita en gran

medida cuales son las posibles candidatas: en el caso del ejemplo, debieran ser "animales capaces de morder". Aún si se tuvieran en cuenta las predicciones del modelo, que es lo que permite considerar el contexto de una palabra para predecir la siguiente, además de que la palabra rime (por ejemplo, eligiendo la palabra que rime únicamente de entre los tokens que tienen una alta probabilidad de ser los siguientes, o tomando el vector de predicciones generado y aumentando la posibilidad de elegir las palabras que cumplen con la estructura de rima), no es posible garantizar que siempre se pueda encontrar una palabra que se adapte al contexto y también rime correctamente.

4. Por último, si bien en el caso de utilizar tokenización a nivel de palabras y disponiendo de una forma de computar si dos palabras efectivamente riman, resulta relativamente fácil realizar un análisis de la probabilidad de aparición de una palabra que rime con otra en particular (simplemente analizando el vector de salida del modelo de lenguaje en la posición que se corresponda con la palabra buscada), esto se complejiza mucho más al utilizar tokenización a nivel de subpalabras. Tanto trabajando a nivel de sílabas como utilizando BPE, en la mayoría de los casos no bastará con observar un token en particular de los que se dispone de su probabilidad de aparición en el vector de salida del modelo para saber si este será parte de una palabra que rime con la deseada. Para esto, se deberán analizar las predicciones de uno o más tokens siguientes a futuro lo que se traduce en un costo computacional muy alto.

En las siguientes subsecciones, se desarrollan las herramientas desarrolladas y las modificaciones realizadas al algoritmo de generación para superar las dificultades enumeradas y efectivamente lograr que el texto generado se adapte a una estructura de rima deseada.

Identificación de palabras que rimen

El primer punto a tratar para poder llevar a cabo la generación de texto bajo una determinada estructura de rima es la identificación de la presencia o no de rima, asonante o consonante, entre dos palabras. Como ya se mencionó en la sección 3.4, la mayoría de los trabajos de dominios similares en lengua inglesa utilizan diccionarios de rimas que le permiten identificar esto. Sin embargo, no se encontraron al momento que se realizó este trabajo de herramientas similares para la lengua española. Por esto, para la identificación de rimas se desarrolló un *script* que permite discriminar la sílaba y vocal tónica de una palabra, y a partir de esto reconocer entre dos palabras si existe una rima consonante (coinciden todas sus letras a partir de la vocal tónica) o asonante (coinciden en sus vocales a partir de la vocal tónica). Sin embargo, como se menciona en la sección 4.2.1, varias de las expresiones utilizadas en el género son directamente palabras en inglés o expresiones que no se ajustan a las reglas de acentuación del español, por lo que el *script* desarrollado inicialmente no será capaz de identificar correctamente las rimas en las que intervengan estos vocablos. Para poder subsanar este problema se definió un diccionario que a cada una de las expresiones en inglés encontradas más frecuentemente en los textos les asigna su correspondiente castellanización, es decir, una adaptación de la palabra a las normas idiomáticas del castellano (por ejemplo, para *flow* le asigna "flou" o para *freestyle* "fristail"). Estas castellanizaciones permiten, para estas palabras en inglés, identificar tanto cuando riman

entre ellas como con otra palabra en español. La figura 4.11 muestra un recorte con las primeras diez entradas definidas en el diccionario.

```
"flow": "flóu",
"freestyle": "fristail",
"disney": "dísney",
"baby": "beibi",
"destroy": "distróy",
"yeah": "yea",
"hardcore": "hárdcor",
"street": "estrit",
"beat": "bit",
"hater": "jeiter"
```

Figura 4.11: Recorte con las primeras diez entradas definidas en el diccionario de rimas.

Entonces finalmente, para identificar si dos palabras del vocabulario riman se chequea en primer lugar si alguna de ellas o ambas poseen una entrada en el diccionario de castellanizaciones que les corresponda y en caso afirmativo se la o las reemplaza por su respectivo valor y luego si se ejecuta el *script* de identificación de rimas con las palabras resultantes. Tanto el *script* descrito como el diccionario de castellanizaciones definido se encuentran disponibles en el repositorio del proyecto[6].

Generación invertida de texto

Disponiendo de una forma de identificar cuando dos palabras rimas se realizó una primera modificación al algoritmo de generación de texto en pos de generar un verso que rime con una palabra en particular:

- Sobre un modelo entrenado con texto tokenizado a nivel de palabras (para evitar el problema descrito en el punto 4) se almacenan las distribuciones de probabilidades producidas por el modelo para los últimos dos tokens generados; entonces, cuando se genere el token que indica el final del verso se dispondrá también de la distribución de probabilidades a partir de la cual se generó el token correspondiente a la última palabra del verso.
- Se modifica esta distribución de probabilidades multiplicando por un factor mayor a 1 a las probabilidades correspondientes a tokens que rimen con la palabra deseada (se pueden utilizar distintos factores según sea una rima consonante o asonante) y se vuelve a computar la función *softmax* sobre el resultado para que se corresponda con una nueva distribución de probabilidades

[6] https://github.com/midusi/freestyle_generator/blob/master/src/rima.py

- Se genera una nueva palabra final para el verso que reemplazará a la anterior, esta vez a partir de la nueva distribución de probabilidades generada, que da prioridad a las palabras que rimen con la palabra deseada.

La figura 4.12 muestra tres estrofas generadas de la forma descrita sobre las cuales se marca en negrita las palabras que se debieron ajustar para que esta se adapte a un esquema de rima en particular. Además muestra debajo de cada estrofa, para cada palabra que fue reemplazada por otra para ajustarse a dicho esquema, la palabra original que se correspondía a esa posición. Se ilustran tres estrofas todas con un esquema de rimas AAAA y estos sirven para verificar lo enunciado en el punto 3: si bien los versos generados de esta forma ahora cumplen con la estructura de rima definida, las palabras utilizadas como reemplazo de las palabras finales de cada verso quedan en general fuera del contexto de este. Esto se puede apreciar para prácticamente todas estas palabras: en el último verso de la primera estrofa mostrada la frase "no vas a ganar" resulta mucho más coherente que la frase "no vas a afeitar" y más aún dentro del contexto de una batalla de *freestyle*. Lo mismo aplica en el último verso de la segunda estrofa, donde "Yo doy la clase, te siento en el pupitre" resulta mucho mas coherente que la frase final: "Yo doy la clase, te siento en el **mall**"; o en la segunda frase de la tercer estrofa mostrada, de nuevo la frase "voy a secuestrar completa a Afrodita" resulta más adecuada que "voy a secuestrar completa a **imaginas**".

Una posible solución entonces a los problemas descritos hasta ahora al intentar generar texto que rime es la generación inversa del texto, es decir, en lugar de entrenar el modelo para que dada una secuencia x_i, x_{i+1}, ..., x_{i+n} este prediga la siguiente palabra x_{i+n+1}, entrenarlo para que prediga la palabra anterior x_{i-1}. Esta forma de generación, utilizada en trabajos con dominios similares tales como [Yi et al., 2017] y [Lau et al., 2018], soluciona los dos problemas mencionados en los puntos 2 y 3: permite conocer exactamente cuando la siguiente palabra a predecir es la última palabra de un verso (luego de la aparición de un token "[SEP]") y permite redefinir la palabra sin el riesgo de generar incoherencias dentro del verso en el se encuentra, ya que este se generará luego a partir de la palabra elegida.

Lógicamente, para llevar a cabo este tipo de generación se deberá modificar la forma en que se generan los conjuntos de datos utilizados para el entrenamiento del modelo. Si bien se utilizará el mismo algoritmo de ventana deslizante sobre el texto tokenizado para generar los pares x, y, ahora $x = [p_{i+n}, p_{i+n-1}, ..., p_{i+1}]$ representará una secuencia cuyo orden se encuentra invertido respecto al orden en el texto original e $y = p_i$ representará la palabra anterior a dicha secuencia. La figura 4.13 ilustra como quedan formados los pares de entrenamiento a partir de un texto de esta forma.

Como se mencionó en la sección 4.3, es por esta razón que se consideró importante que la capa LSTM del modelo fuera bidireccional, ya que al estar las secuencias de los pares de entrenamiento invertidas respecto al orden natural del lenguaje, es posible que al procesarlas de esta forma no se captaran dependencias secuenciales del texto. Al ser la capa bidireccional se analizaran las secuencias también de forma inversa, lo que se corresponderá con el orden real de los tokens en el texto y será capaz entonces de captar dichas dependencias.

Que pueda hacerlo en un momento como este ya ***apremia***
Que vas a hacer? Es normal que yo lo ***bacterias***
Claro que esta batalla ya la ***bacteria***
No tenés lo mismo, no vas a ***afeitar***

("adorne" → "**bacterias**"), ("regaló" → "**bacteria**"), ("ganar" → "**afeitar**")

Te doy, te gano, aquí te voy a ***dar***
Con las rimas que suelto ***ja'***
Por escribir esos ***ah***
Yo doy la clase, te siento en el ***mall***

("épicas" → "**ja'**"), ("rumores" → "**ah**"), ("pupitre" → "**mall**")

Se cubren los oídos, como por ***dinamita***
Pero yo voy a secuestrar completa a ***imaginas***
Resentido! Y la tengo aunque ni siquiera lo ***cristalina***
No, no, sólo ven que no quedó ***imaginar***

("Afrodita" → "**imaginas**"), ("veas" → "**cristalina**"), ("ninguno" → "**imaginar**")

Figura 4.12: Ejemplos de estrofas generadas de la forma descrita con un modelo entrenado con tokens a nivel de palabra, con un largo de secuencia 5 y temperatura 1.

La figura 4.14 muestra, de la misma forma que la figura 4.12, la aplicación del algoritmo de generación descrito al comienzo de la sección pero sobre texto generado de forma invertida. Se puede observar ahora que las palabras que se debieron ajustar para que la estrofa cumpla con la estructura de rima ya no representan problemas de coherencia para los versos y de hecho resultan mucho más acordes al verso que las palabras previas al reemplazo, lo que tiene sentido ya que el verso fue construido luego de reemplazar la palabra y partiendo de esta. Si se puede observar que las palabras que fueron reemplazadas en varios casos guardaban una relación más estrecha con el verso siguiente en la estrofa (que fue generado inmediatamente antes que la palabra): en el tercer verso de la primera estrofa se reemplaza la palabra romana, que guardaba relación directa con la temática del verso siguiente, por demanda, que si bien se ajusta correctamente al verso en el que se encuentra, no guarda relación con el verso siguiente, el cual fue generado previo a la palabra. Otro caso donde se da esta situación se puede observar en el segundo verso de la segunda estrofa, al reemplazar "muerto" por "territorio".

Cómputo de probabilidades a nivel de subpalabras y diccionario de rimas

Si bien resulta útil al utilizar modelos entrenados con texto tokenizado a nivel de palabras, no es posible aplicar el algoritmo descrito en la sección anterior al utilizar otros niveles de

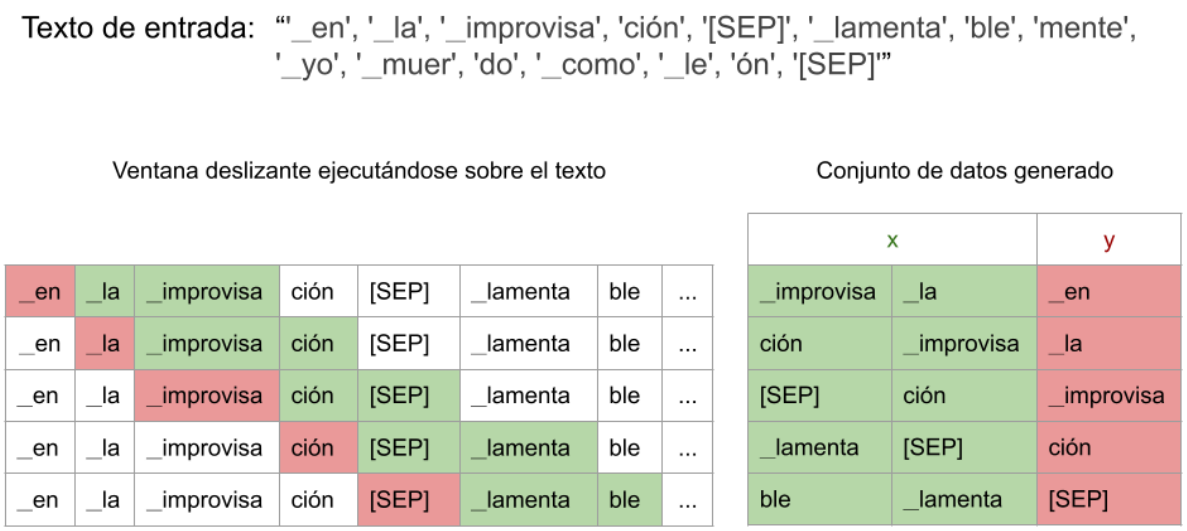


Figura 4.13: Ejecución de algoritmo de ventana deslizante de tamaño 3 para la generación de un conjunto de datos invertido respecto al orden del texto.

tokenización a nivel de subpalabras como lo son la tokenización por sílabas o utilizando BPE por lo ya descrito en el punto 4: un único token correspondiente a una subpalabra no brinda la suficiente información respecto a si la palabra de la que formará parte rima o no con otra palabra dada, aún siendo el último token que la compone.

Para poder obtener una versión de dicho algoritmo que funcione a nivel de subpalabras se generaron el conjunto de todas las palabras distintas presentes en ambas bases de datos y un diccionario de rimas, el cuál se generó computando para cada palabra del conjunto mencionado todas las otras palabras de dicho conjunto que rima con ella y el tipo de la rima (consonante o asonante). Lógicamente, la generación de dicho diccionario de rimas resulta computacionalmente muy costosa ya que esta es de orden cuadrático sobre el total de palabras en los conjuntos de datos que cuenta con 28424 palabras distintas. Sin embargo, esto se lleva a cabo una única vez y permite acelerar la obtención de palabras que rimen con una dada durante la generación del texto.

Luego, la versión modificada del algoritmo para generar un verso que rime con una palabra dada, teniendo en cuenta que se utiliza generación invertida del texto, funciona de la siguiente manera:

- En primer lugar se obtendrán a través del diccionario de rimas las palabras que riman con la palabra dada, y para cada una de estas palabras p:
  - Se tokeniza la palabra p de acuerdo al nivel de tokenización que se esté utilizando,

Y seguro que se acabaron las ***entradas***
Porque mi faceta te hace ser un cometa, no cometas más ***erratas***
Yo represento a la gente que ***demanda***
Este es el imperio romano, bienvenido a ***esparta***

("peligro" → "**entradas**"), ("chaval" → "**erratas**"), ("romano" → "**demanda**")

Vengo con rap, rap en ***episodio***
Represento a este chile hermano, ***territorio***
Y cuando yo ya arranco tu te ***mueres***
La verdad que tengo el don y un hijo de ***nueve***

("latina" → "**episodio**"), ("muerto" → "**territorio**"), ("encontrar" → "**mueres**")

Ha vuelto este ritmo que parece ***grato***
Hoy hoy día para mí todos esos rivales son ***malos***
Saben que lo hago yo lo hago cuando ***hablo***
Oh, de verdad, aprovecha de aprenderte el ***año***

("lindo" → "**grato**"), ("lenguaje" → "**malos**"), ("lejos" → "**hablo**")

Figura 4.14: Ejemplos de estrofas generadas de la forma descrita con un modelo entrenado con tokens a nivel de palabra, con un largo de secuencia 5 y temperatura 1, entrenado con las secuencias de texto invertidas.

resultando en la secuencia $[t_1, t_2, ..., t_n]$.

- Se obtiene la probabilidad de generación de t_n ($P(gen(t_n))$) a partir de la distribución de probabilidades producida por el modelo al procesar la secuencia previa s. Luego, para obtener la probabilidad de generación de cada uno de los tokens t_i habiéndose generado el token t_{i+1} ($P(gen(t_i)|gen(t_{i+1}))$), se agrega al final de la secuencia s el token t_{i+1} y se da esta como entrada al modelo. La probabilidad mencionada se corresponde con la ubicación del token t_i en la distribución de probabilidades producida por el modelo.

- Se considera la posibilidad de generación de esta palabra p como:

$$P(gen(p)) = P(gen(t_1)|gen(t_2) \cap ... \cap gen(t_n)) \tag{4.1}$$

Y por la regla de la cadena de la probabilidad:

$$P(gen(p)) = \prod_{k=1}^{n} P(gen(t_k)| \bigcap_{j=n}^{k-1} gen(t_j)) \tag{4.2}$$

- Se computa $P(gen(p))$ y se almacena en un diccionario utilizando p como clave.

- Habiéndose generado un diccionario que contiene para cada una de las palabras que rima con la palabra dada su correspondiente probabilidad de generación, se computa una distribución de probabilidades entre estas utilizando la función *softmax*. Luego, se elige una de estas palabras de forma aleatoria en función de la distribución de probabilidades generada. La palabra elegida efectivamente rima con la palabra deseada ya que fue elegida de entre las devueltas por el diccionario de rimas.

4.5.3. Algoritmo de generación

Finalmente, teniendo en cuenta las consideraciones descritas en las subsecciones anteriores, el proceso de generación de una estrofa que se ajuste a un determinado esquema de rima consiste en:

- Ingresar un texto que se utiliza como "semilla" para el modelo. Dicho texto se tokeniza utilizando los tokens definidos al momento que se ejecutó el algoritmo de BPE sobre la totalidad del texto y en caso de encontrar tokens que no están presentes en el vocabulario se les asigna un token desconocido ("[UNK]").
- Definir una estructura de rima de forma aleatoria entre algunas de las estructuras más típicamente utilizadas, para este trabajo se tomaron las estructuras AAAA, AABB, ABAB y ABBA.
- Utilizar la tokenización de la semilla como entrada para el modelo y a partir de las predicciones de este generar texto hasta que se obtenga un token "[SEP]" indicando el final del verso.
- La siguiente palabra a predecir será la ultima palabra del verso anterior, por lo que se debe garantizar que esta se ajuste a uno de los esquemas de rimas mencionados. En caso de que eso suceda y deba rimar con una palabra de un verso anterior se utilizará el algoritmo descrito en la subsección anterior para obtener una palabra que cumpla con la rima esperada.
- Esto se repetirá hasta completar los cuatro versos acordes al esquema de rima de la estrofa.

En la figura 4.15 se muestran algunas estrofas generadas usando el algoritmo descrito. En el anexo A del anexo se incluyen más muestras de texto generado.

De la esquina cómo verás más que ignoraban
¿Y dónde está todo eso con lo que soñabas
Hago la diferencia con mi poesía pero nada
Yo, yeah, entendí que todo importaba

La diferencia con el es que me gano el primer puesto
La fe mueve montañas mis montañas
Mi personalidad que me acompañaba
Mirame a mi ya me voy yendo

Yo sí te voy a subir el pulgar
Es caro, flow extranjero
Representado canotes a llorar
Vas a perderlo, negro, no retrocedemos

Mientras yo escribía vos la criticaste
Que si matan, que si roncan, si te tocan los parto como street fighter
Quiero verte bailar con los diamantes
No de versace na' nadie lo hace

Figura 4.15: Ejemplos de estrofas generadas de la forma descrita con un modelo entrenado sobre un texto tokenizado utilizando BPE con un tamaño de vocabulario de 5000, con un largo de secuencia 5 y temperatura 1, entrenado con las secuencias de texto invertidas. Se pueden observar una primera y última estrofa generadas bajo la estructura AAAA, mientras que la segunda y tercera siguen las estructuras ABBA y ABAB respectivamente

Capítulo 5

Conclusiones y trabajos futuros

En este capítulo se exponen las principales conclusiones obtenidas durante el desarrollo de la tesina. Luego, se presentan posibles líneas de investigación futuras en pos de extender los temas estudiados en esta tesina.

5.1. Conclusiones generales

A lo largo de este trabajo se estudiaron las técnicas existentes para la generación de lenguaje utilizando aprendizaje automático y más específicamente *deep learning*. Se desarrollaron los conceptos esenciales acerca de estos métodos que hoy en día son los que alcanzan el estado del arte en lo que refiere a tareas de NLP a través del uso de arquitecturas recurrentes tales como redes LSTM, también descritas en esta tesina.

Luego, se profundizó sobre ideas fundamentales en lo que refiere a NLP tales como tokenización y algoritmos específicos para llevarla a cabo como lo es el de Byte Pair Encoding, técnicas de representación de documentos como los *word embeddings* y como se obtienen, la definición de modelo de lenguaje y puntualmente de modelo de lenguaje neuronal recurrente y como se pueden utilizar estos para la generación de un tipo de texto en particular. Además se estudiaron los factores y dificultades a tener en cuenta a la hora de generar texto que debiera corresponderse con una estructura en particular como cierta métrica o rima. Teniendo en cuenta estos, se realizó un estudio sobre distintos trabajos cuyos problemas también se enmarcaban dentro de la generación de texto estructurado, la mayoría de los cuales se encontraron en idioma inglés, y se analizaron las soluciones que presentaban para sortear las dificultades presentadas. Por último, también se estudiaron las dificultades específicas que surgen al trabajar con un género como el *freestyle* en español como lo fueron la inclusión de expresiones en inglés, vocablos propios de la jerga del género o ciertas onomatopeyas que no respetan las reglas de acentuación del idioma español.

Utilizando los conceptos estudiados se logró entonces desarrollar un sistema generador de texto

que, a través de un modelo de lenguaje neuronal y algoritmos de generación de texto que garantizan que este respete la estructura y rima correspondiente, genere líricas que se enmarcan dentro del género de las batallas de *freestyle*. Para esto fue necesaria la confección de una base de datos de texto que contiene específicamente transcripciones de batallas de *freestyle* ya que no se encontró al momento de realización de esta tesina otra base de datos pública que contenga este tipo de material. Además también se generó otra base de datos de mayor tamaño que contiene letras de canciones de géneros similares y que fue utilizada para preentrenar el modelo de lenguaje utilizado.

Para llevar a cabo esto, se realizaron en primer lugar tareas de preprocesamiento sobre el texto que permitieron observar que en casos como el del presente trabajo donde los datos disponibles para entrenar un modelo de lenguaje son en principio escasos y con un vocabulario muy diverso la tokenización a nivel de palabras puede no ser ideal. Esto se concluyó a partir de observar que, en este caso, gran cantidad de los tokens (palabras) que componen el texto aparecen en este con muy baja frecuencia. En este caso la tokenización a nivel de subpalabras mostró ser una forma más adecuada: la tokenización a nivel de sílabas efectivamente redujo la cantidad de tokens con baja frecuencia en el texto tokenizado, pero continuaba representando una porción significativa del total. El algoritmo de Byte Pair Encoding con tamaños de vocabulario pequeños (1000, 3000 o en menor medida 5000) fue el método de tokenización más eficiente para solucionar esta problemática, resultando en que la cantidad de tokens con menos de tres apariciones en el total del conjunto de datos ronde el 1 % del total.

Respecto al entrenamiento del modelo se ha podido confirmar la efectividad del uso de la técnica de *fine-tuning* sobre las dos bases de datos generadas: luego de un pre entrenamiento sobre la base de datos de mayor tamaño se pudieron lograr porcentajes de precisión de alrededor del 90 % sobre la base de datos que contiene transcripciones de batallas de *freestyle* con una cantidad de épocas de entrenamiento significativamente menor.

Por último se concluye que el modelo de lenguaje utilizado logró efectivamente captar el estilo del género a partir del entrenamiento sobre las bases de datos desarrolladas, pero no así la correcta estructuración de este en estrofas de cuatro versos o el correcto uso de la rima. Sin embargo, a partir de algoritmos que garantizaron que el texto generado cumpliera con estas dos características fue posible generar texto que se corresponda no solo con el estilo si no también con la estructura correspondiente al género de las batallas de *freestyle*.

5.2. Lineas de trabajo futuras

Se identifican como posibles lineas de trabajo a futuro para la continuación de lo presentado en esta tesina:

- La implementación de un sistema que, en concordancia con el género de las batallas de *freestyle* sea capaz no solo de generar texto que se enmarque dentro del género si no que también sea capaz de responder a una estrofa producida por un potencial contrincante. Para esto sería necesario que el sistema evalúe para una estrofa dada cuales son los factores

clave a tener en cuenta para elaborar una respuesta que resulte adecuada y coherente no solo internamente si no para con la estrofa a la cual responde.

- Profundizar en la utilización de técnicas conocidas como de aprendizaje con pocos ejemplos o *few-shot learning* y estudiar su aplicabilidad al problema desarrollado en esta tesina en particular. Estas permiten que modelos de aprendizaje generalicen a partir de pocos ejemplos de entrenamiento, lo que resulta adecuado siendo que la base de datos disponible que contiene exclusivamente transcripciones de batallas de *freestyle* es relativamente pequeña. Entre estas técnicas se incluye, por ejemplo, el *fine-tuning* utilizado en este trabajo.
- Experimentar con el uso de otras arquitecturas de redes neuronales para definir el modelo de lenguaje a utilizar. Un ejemplo de esto puede ser la conocida como *Transformer*, una arquitectura de red neuronal recurrente que ha sido muy utilizada recientemente dentro de trabajos del área de NLP y puntualmente en los que se tratan problemas conocidos como "de secuencia a secuencia" (o *sequence to sequence*), tales como la traducción o el resumen de textos.
- Por último, aumentar la cantidad de datos disponibles para el entrenamiento de los modelos. Esto se puede llevar a cabo en primer lugar simplemente ampliando el tamaño de las bases de datos desarrolladas a partir de la recolección de nuevas transcripciones o canciones de géneros similares así como también estudiando la posibilidad de la utilización de técnicas de *data-augmentation* para generar más datos a partir de los que ya se encuentran disponibles.

Bibliografía

[Allwein et al., 2000] Allwein, E. L., Schapire, R. E., and Singer, Y. (2000). Reducing multiclass to binary: A unifying approach for margin classifiers. *Journal of machine learning research*, 1(Dec):113–141.

[Baldi et al., 1999] Baldi, P., Brunak, S., Frasconi, P., Soda, G., and Pollastri, G. (1999). Exploiting the past and the future in protein secondary structure prediction. *Bioinformatics*, 15(11):937–946.

[Bengio et al., 1994] Bengio, Y., Simard, P., and Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166.

[Bird et al., 2009] Bird, S., Klein, E., and Loper, E. (2009). *Natural Language Processing with Python*. O'Reilly Media, Inc., 1st edition.

[Branwen, 2020] Branwen, G. (2020). Gpt-3 creative fiction.

[Brown et al., 2020] Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. (2020). Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*.

[Chollet et al., 2018] Chollet, F. et al. (2018). *Deep learning with Python*, volume 361. Manning New York.

[Cybenko, 1989] Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314.

[Dai et al., 2019] Dai, Z., Yang, Z., Yang, Y., Carbonell, J., Le, Q. V., and Salakhutdinov, R. (2019). Transformer-xl: Attentive language models beyond a fixed-length context. *arXiv preprint arXiv:1901.02860*.

[Deng and Liu, 2018] Deng, L. and Liu, Y. (2018). *Deep learning in natural language processing*. Springer.

[Devlin et al., 2018] Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2018). Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

[Edwards, 2009] Edwards, P. (2009). *How to rap*. Chicago Review Press.

[Ferretti et al., 2018] Ferretti, E., Cagnina, L., Paiz, V., Donne, S. D., Zacagnini, R., and Errecalde, M. (2018). Quality flaw prediction in spanish wikipedia: A case of study with verifiability flaws. *Information Processing & Management*, 54(6):1169 – 1181.

[Gage, 1994] Gage, P. (1994). A new algorithm for data compression. *C Users Journal*, 12(2):23–38.

[Ghazvininejad et al., 2016] Ghazvininejad, M., Shi, X., Choi, Y., and Knight, K. (2016). Generating topical poetry. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 1183–1191.

[Glorot et al., 2011] Glorot, X., Bordes, A., and Bengio, Y. (2011). Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 315–323. JMLR Workshop and Conference Proceedings.

[Goldberg and Hirst, 2017] Goldberg, Y. and Hirst, G. (2017). Neural network methods in natural language processing. morgan & claypool publishers(2017). *9781627052986 (zitiert auf Seite 69)*.

[Goodfellow et al., 2016] Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press. http://www.deeplearningbook.org.

[Graves, 2012] Graves, A. (2012). Supervised sequence labelling. In *Supervised sequence labelling with recurrent neural networks*, pages 5–13. Springer.

[Graves et al., 2008] Graves, A., Liwicki, M., Fernández, S., Bertolami, R., Bunke, H., and Schmidhuber, J. (2008). A novel connectionist system for unconstrained handwriting recognition. *IEEE transactions on pattern analysis and machine intelligence*, 31(5):855–868.

[Graves et al., 2013] Graves, A., Mohamed, A.-r., and Hinton, G. (2013). Speech recognition with deep recurrent neural networks. In *2013 IEEE international conference on acoustics, speech and signal processing*, pages 6645–6649. Ieee.

[Graves and Schmidhuber, 2005] Graves, A. and Schmidhuber, J. (2005). Framewise phoneme classification with bidirectional lstm and other neural network architectures. *Neural networks*, 18(5-6):602–610.

[Gutmann and Hyvärinen, 2010] Gutmann, M. and Hyvärinen, A. (2010). Noise-contrastive estimation: A new estimation principle for unnormalized statistical models. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pages 297–304.

[Hassen et al., 2017] Hassen, M., Carvalho, M. M., and Chan, P. K. (2017). Malware classification using static analysis based features. In *2017 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 1–7. IEEE.

[Hinton et al., 2015] Hinton, G., Vinyals, O., and Dean, J. (2015). Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*.

[Hochreiter et al., 2001] Hochreiter, S., Bengio, Y., Frasconi, P., Schmidhuber, J., et al. (2001). Gradient flow in recurrent nets: the difficulty of learning long-term dependencies.

[Hochreiter and Schmidhuber, 1997a] Hochreiter, S. and Schmidhuber, J. (1997a). Long short-term memory. *Neural computation*, 9(8):1735–1780.

[Hochreiter and Schmidhuber, 1997b] Hochreiter, S. and Schmidhuber, J. (1997b). Lstm can solve hard long time lag problems. *Advances in neural information processing systems*, pages 473–479.

[Hornik et al., 1989] Hornik, K., Stinchcombe, M., and White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366.

[Jurafsky and Martin, 2018] Jurafsky, D. and Martin, J. H. (2018). Speech and language processing (draft). *Chapter A: Hidden Markov Models (Draft of September 11, 2018). Retrieved March*, 19:2019.

[Kalchbrenner et al., 2016] Kalchbrenner, N., Espeholt, L., Simonyan, K., Oord, A. v. d., Graves, A., and Kavukcuoglu, K. (2016). Neural machine translation in linear time. *arXiv preprint arXiv:1610.10099*.

[Lau et al., 2018] Lau, J. H., Cohn, T., Baldwin, T., Brooke, J., and Hammond, A. (2018). Deep-speare: A joint neural model of poetic language, meter and rhyme. *arXiv preprint arXiv:1807.03491*.

[Liddy, 2001] Liddy, E. D. (2001). Natural language processing.

[Luhn, 1960] Luhn, H. P. (1960). Key word-in-context index for technical literature (kwic index). *American documentation*, 11(4):288–295.

[Mikolov et al., 2013] Mikolov, T., Chen, K., Corrado, G., and Dean, J. (2013). Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*.

[Mohri et al., 2018] Mohri, M., Rostamizadeh, A., and Talwalkar, A. (2018). *Foundations of machine learning*. MIT press.

[Montavon et al., 2012] Montavon, G., Orr, G., and Müller, K.-R. (2012). *Neural networks: tricks of the trade*, volume 7700. springer.

[Nadkarni et al., 2011] Nadkarni, P. M., Ohno-Machado, L., and Chapman, W. W. (2011). Natural language processing: an introduction. *Journal of the American Medical Informatics Association*, 18(5):544–551.

[Papineni et al., 2002] Papineni, K., Roukos, S., Ward, T., and Zhu, W.-J. (2002). Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318.

[Pascanu et al., 2013] Pascanu, R., Mikolov, T., and Bengio, Y. (2013). On the difficulty of training recurrent neural networks. In *International conference on machine learning*, pages 1310–1318. PMLR.

[Poole et al., 1998] Poole, D., Mackworth, A., and Goebel, R. (1998). Computational intelligence.

[Potash et al., 2015] Potash, P., Romanov, A., and Rumshisky, A. (2015). Ghostwriter: Using an lstm for automatic rap lyric generation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1919–1924.

[Radford et al., 2017] Radford, A., Jozefowicz, R., and Sutskever, I. (2017). Learning to generate reviews and discovering sentiment (2017). *arXiv preprint arXiv:1704.01444*.

[Rajaraman and Ullman, 2011] Rajaraman, A. and Ullman, J. D. (2011). Mining of massive datasets: Data mining (ch01). *Min. Massive Datasets*, 18:114–142.

[Rumelhart et al., 1986a] Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986a). Learning representations by back-propagating errors. *nature*, 323(6088):533–536.

[Rumelhart et al., 1986b] Rumelhart, D. E., Smolensky, P., McClelland, J. L., and Hinton, G. (1986b). Sequential thought processes in pdp models. *Parallel distributed processing: explorations in the microstructures of cognition*, 2:3–57.

[Schuster and Paliwal, 1997] Schuster, M. and Paliwal, K. K. (1997). Bidirectional recurrent neural networks. *IEEE transactions on Signal Processing*, 45(11):2673–2681.

[Sennrich et al., 2015] Sennrich, R., Haddow, B., and Birch, A. (2015). Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909*.

[Sutskever et al., 2014] Sutskever, I., Vinyals, O., and Le, Q. V. (2014). Sequence to sequence learning with neural networks. *arXiv preprint arXiv:1409.3215*.

[Uysal and Gunal, 2014] Uysal, A. K. and Gunal, S. (2014). The impact of preprocessing on text classification. *Information Processing & Management*, 50(1):104–112.

[Vajjala et al., 2020] Vajjala, S., Majumder, B., Gupta, A., and Surana, H. (2020). *Practical Natural Language Processing*. O'Reilly Media, Inc.

[Werbos, 1990] Werbos, P. J. (1990). Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560.

[Yi et al., 2017] Yi, X., Li, R., and Sun, M. (2017). Generating chinese classical poems with rnn encoder-decoder. In *Chinese Computational Linguistics and Natural Language Processing Based on Naturally Annotated Big Data*, pages 211–223. Springer.

[Yin and Shen, 2018] Yin, Z. and Shen, Y. (2018). On the dimensionality of word embedding. In *Advances in Neural Information Processing Systems*, pages 887–898.

[Young et al., 2018] Young, T., Hazarika, D., Poria, S., and Cambria, E. (2018). Recent trends in deep learning based natural language processing. *ieee Computational intelligenCe magazine*, 13(3):55–75.

[Zhang et al., 2020] Zhang, A., Lipton, Z. C., Li, M., and Smola, A. J. (2020). *Dive into Deep Learning*. https://d2l.ai.

[Zugarini et al., 2019] Zugarini, A., Melacci, S., and Maggini, M. (2019). Neural poetry: Learning to generate poems using syllables. In *International Conference on Artificial Neural Networks*, pages 313–325. Springer.

Apéndice A

Muestras de texto generado

Porque en tu barrio no tengo pa' montarte
Negro velando como un melón
Te mato con frases, te doy el desenlace
Tengo las rimas en la improvisación

De que me habrían
Como, y los que veo colores me atraía
Todos sabe lo que entendía
Pobre porquería pobre porquería

Ella pasa la pistola metia en el cazador
No estoy roncando con viento, tus plans, es ke lo merezca esa cara de vencedor
Voy subiendo el desibelio tú te pareces al criterioh
Después se te hace un amor

Ya se la promesa
Y haber preferido ser mc que la universidad
Cuando no entendes que lo hacemo' a los blandos ladrones de mirar
Por que más te vas si sean

Y yo amartando, fracaso' me pidió disculparme
Venga con main interesada
Se que vos conmigo no resucitadas
Miradas ya no se hacen

Ah, yeah, los negro llegó en el imperio lo deje
El impero con nosotros, represente
Acá la verdad no ofende
Pues yo quiero tu gente

Yo no soy así véntilo la país que los problemas resolvemos

Ve tus jugadas, tus movimientos
El sador yo soy elegante salido del fuego
Yo tengo rap y no es solo mi atuendo

Hoy sigo de pie a pesar de todo, por el que está, por el que va, con esto que te quemo
Vos rapero siguen salvando rimas triunfadas
Este idiota lo intenta, al final no me dice nada
Obviamente hermano, juro que a mi ya me voy yendo

Desde un día paladarreados
Ya no siento nada en cada frente de mi' desesione'
Por esto es ser hiphoper
Eh, eh, eh, eh esto es ser hiphoper

De mi gran estrenada
Acuérdate de mí yo lo copio
Lo siento si rapeas mucho pero nunca dices nada
Si no te acordás, pero es muy obvio

Se que por subir involucra
Dime, dime qué pasó entre tú y yo, eh wo
Para que vean que voy un paseo, a mi melodía y la mía la modulas
Como un sudor fresco recorriéndole la nuca

Pero es el tipo de corrido
Dejo mi vida en cada renglón
Te falta aprender que tengo creación
En tu piel oscuras que has tenido

Así que solamente, ey, yo permanezco
Quiero billete y la perno que yo quiero
Que quiero que se oiga desde montevideo
A vernos por encontrarnos los pelos

De tu mente me la tomo como vino
Yo ya no me hablaba
Ves mi pistola, ella e' mi bala
Y esto sucede ahora mismo

Por mi ganas de amar por mis ganas de amar
Es una bola de billar
Y ya ven que aquí fluyo un montón?
Que pasó? perdes la improvisación?

Años criticando no me bajaron de la carrera, te pensas que soy el apuro de tu vacio
Y y para ser dos tengo miles de familias

Ninguno de verdad, yo le daría el dinero pa' siria
Te dejo la equis, porque todavía no saben que es tanta, sale del escenario

Se pesa esta durmiendo
Creía que se pobe pero igual estoy latiendo
Mi rima es el estilo porque todos saben bien lo que quiero
Irme a mi ya me voy yendo

Y vos con tus rimas plásticas
A mí no me interesa que le quieras contar, a mí no me interesa es la pellan
Me toca partir la liga, porque yo soy el que revienta la temática
Námate que un sí, pero a ver igual sin estrellas

Esto es de verdad yo no lo invento
Puro real me dice que la imita
Vos no lo viviste para contar el cuento
Ser hip hop no es subirte a mi tarima

A ver si están con la cara rota
No no es mi bull igual hay que decir que es uno de los suspiros
Yo no preparo contenido
Y es que el loco va a decir que no va a batallar para el tuyo es el problema de trueno ni-dula, es un banditito

El flow mío aprieta, claro lo haré otra cosa que ignorarlos
El último y empezaron
Porque ya sabe muy asaltando
Y no me partiste un carajo

Yo sí te voy a enseñar, yo salió el mote
Seguí un hermano, de la batalla lo soltaste en el bombo un pariante' y rockers
Entonces no pudiste con los otros vatos
Tranquilamente me perdí, yo me pongo hardcore

Me diste la gorra te muestro lo que es la placa de este weon
Eso es así vo', eso es rey de revueltos
Así lo asimilo digo todo lo que quiero
Si quiero irme yo ya me voy yendo

Que ahora improviso de que quiere rapear juntos
Ya representando ya la sangre, con calculo
fal tan dos, tranquilo boludo
Que rapea, te golpea donde antes no hubo

Rilo, rapera, escuché cabida es mi flow y roble
Tengo rap y no vendo ni un atuendo

Dice usted con mi flow me lo muevelo
Pues que cree posible, pero solo son sensaciones

Que ya estés acá arriba es por que ya han fallado
Así que puedo hacerlo no acaparo, en estilo no me comparo
Soy un warrior, no sé si no lo has notado
Vengo del barrio pesado donde los raperos han rapeado

No me podés, porque sobrame mis trampas
Y al final es que la envidia mata ah
Te doy ventaja, no le das a la caja, sos la migaja
En serio se lo quitas a la grada

De la esquina cómo verás más que ignoraban
¿Y dónde está todo eso con lo que soñabas
Hago la diferencia con mi poesía pero nada
Yo, yeah, entendí que todo importaba

Dame un rompe portón, es el acertijo
Pero no tiene talento es como un codiguito
Dice que criticó y no ha ocupado el concepto pero ahora lo invertimos
Pero esto te pasa porque no sos vos mismo

Representamos a chile, peleamos como espartantes en la canción del pastis
Porque cuando yo entro siempre reviento el bombo
Ey, y al piso, piso, el piso, trazo, te lo creo, creo casi
Oh dem, yo no estoy jugando al fortnite contra este detrozo

Por eso yo sí que tengo reconocimiento
La respuesta puesta o la sudaba
Este idiota lo intenta, al final no me dice nada
Obviamente hermano, juro que no es un invento

Cuando agarró el micrófono me siento un cocodrilo en la tarima pero ya hay más dilatadas
Es que hay el que sabes que prefiero
Sé que se ha confundido en todas, donde que yo me esperaba
Tu problema no es mi andia del hardcore perro

Me tira el comienzo de mi carrera y el tiene el de vuelta
Este es un beat de lo que sea yo aunque gane aunque pierda
Saben de esta mal y yo lo huelo
Riquelme ya te la clavó al ángulo en el último momento

Irte pa' siria, ahí conoce las bombas
Pero yo fluyo en el bombo y lo normal es que tú te escondas?
Ay perdón, miama a ser una leyenda como el

Y todos saben que va a sucecer

Yo soy más que un muerto, más que un llorón
No es nada creo que esto ya lo hartó
No le pegue al patrón
No soy, ni jimmy neutron, soy el auto mas bueno cuando subo al renglón

Yo sí te voy a subir el pulgar
Es caro, flow extranjero
Representado canotes a llorar
Vas a perderlo, negro, no retrocedemos