Escuela Técnica Superior de
Ingeniería Informática

TRABAJO FIN DE GRADO

Generación de Haikus con Deep Learning

Realizado por

José Manuel Gata Fernández

Para la obtención del título de

Grado en Ingeniería Informática - Ingeniería del Software

Dirigido por

Juan Antonio Nepomuceno Chamorro

Realizado en el departamento de

Lenguajes y Sistemas Informáticos

Convocatoria de Junio, curso 2020/21

Agradecimientos

En primer lugar, me gustaría agradecer a mi tutor D. Juan Antonio Nepomuceno Chamorro por su afable supervisión y apoyo en el trabajo realizado.

Quisiera agradecer al colectivo docente de la Universidad de Sevilla por haber hecho posible mi formación gracias a sus enseñanzas. En especial, agradecer a D. Jorge García Gutiérrez, quien me contagió la pasión por la programación desde el principio.

En segundo lugar, agradecer a mi familia. A mis padres, por haberme apoyado siempre en todas mis decisiones y por haberme enseñado que el trabajo siempre tiene sus frutos.

A mi hermana menor, Anabel, por haber sido una fuente de inspiración y calma cuando más lo necesitaba y por haber creído siempre en mí.

Por último, agradecer a mis compañeros y amigos, porque sin ellos esta etapa no la podría recordar con los buenos momentos que hemos pasado juntos. En especial, agradecer a Francisco Alé Palacios, Curro, y a Pedro Biedma Fresno, por haber defendido conmigo cada proyecto a lo largo de todos estos años.

Resumen

El presente trabajo “Generación de Haikus con Deep Learning” explora el campo de la generación de texto creativo mediante el uso de inteligencia artificial. El principal objetivo de esta investigación es tener una primera toma de contacto con el Deep Learning y lograr comprender su funcionamiento, desarrollando un generador de haikus mediante el uso de redes neuronales recurrentes.

Para llevar a cabo esta investigación, se ha realizado una revisión de artículos de inteligencia artificial que exploran el problema de la generación de haikus y también se ha realizado un estudio del libro “Deep Learning con Python”, de François Chollet.

Este documento muestra el estudio realizado para el cumplimiento de los objetivos designados. Comienza con el estudio del dominio del problema y del estado del arte, como primera interacción con el mismo. Una vez se comprende el funcionamiento de las redes neuronales y las bases del Deep Learning, se desarrollan diversos notebooks para poner en práctica los conocimientos teóricos y, posteriormente, se desarrollan tres modelos que tratan de dar solución al problema de la generación de haikus.

Sobre los generadores realizados, concluimos que el mejor resultado obtenido fue en el tercero, haciendo uso de una red neuronal recurrente LSTM con sendas entradas y salidas para cada uno de los versos, realizando generación de texto a nivel de carácter. El texto generado resultó en determinados casos coherente, mostrando una estructura sintáctica correcta y manteniendo el esquema métrico 5-7-5 de un haiku.

Palabras clave: Generación, haikus, Deep Learning, inteligencia artificial, redes neuronales recursivas, LSTM

Abstract

The current study “Haikus Generation with Deep Learning” explores the field of creative text generation through the use of artificial intelligence. The main objective of this research is to have a first contact with Deep Learning and to understand its functioning, developing a haikus generator using recurrent neural networks.

To carry out this research, it has been necessary to review artificial intelligence articles that explore the problem of haiku generation and the study of the book “Deep Learning with Python”, by François Chollet, has also been carried out.

This document shows the study made in order to fulfil the designated objectives. It begins with the study of the domain of the problem and the state of the art, as the first interaction with it. Once the functioning of neural networks and the bases of Deep Learning had been understood, various notebooks were developed to put the theoretical knowledge into practice and, subsequently, three models were developed to try to solve the haikus generation problem.

Regarding the generators created, we conclude that the best result obtained was in the third, using a LSTM recurrent neural network with inputs and outputs for each of the verses, generating text at character level. The generated text was consistent in certain cases, showing a correct syntactic structure and maintaining the 5-7-5 metric scheme of a haiku.

Keywords: Generation, haikus, Deep Learning, artificial intelligence, recurrent neural networks, LSTM

Material de apoyo

En el siguiente enlace podemos encontrar un repositorio GitHub con el conjunto de notebooks realizados para este trabajo.

- **Enlace GitHub:**

https://github.com/josgatfer/Generacion-de-haikus-con-Deep-Learning

- **Código QR:**

Índice general

Índice de figuras

Índice de extractos de código

Índice de cuadros

Índice de tablas

Capítulo 1. Problema planteado

En el presente documento se desarrollará el estudio en profundidad del Deep Learning. La investigación se ha realizado para la aplicación de lo aprendido en el problema de la generación automática de haikus.

1.1 Análisis de antecedentes y aportación realizada

Para la implementación de la aplicación, me basaré en el artículo: "*Generating Haiku with Deep Learning*" [1]. En dicho artículo el autor explica el estudio del dominio del problema y el desarrollo de la aplicación, especificando los problemas con los que se encontró y su forma de resolverlos.

1.2 Objetivos

El objetivo de este proyecto es el aprendizaje de Deep Learning mediante el estudio del libro "*Deep Learning con Python",* de François Chollet, y la posterior aplicación de lo aprendido en el problema de la generación de haikus con Deep Learning.

El estudio del libro se centrará en los capítulos que hablan sobre el Deep Learning, en especial, para su aplicación en la generación de textos y secuencias.

No es objetivo del proyecto el correcto funcionamiento del generador de haikus.

1.3 Metodología

Para la realización del proyecto se pondrán en uso los conocimientos adquiridos sobre metodologías ágiles, consistentes en la integración continua de cambios en el proyecto de manera incremental. Para el desarrollo de la aplicación se utilizará Github como sistema de control de versiones en la nube. En el repositorio en la nube se realizará el versionado del código fuente de los *notebooks* de Jupyter y de la memoria.

Se llevarán a cabo reuniones con el tutor, Juan Antonio Nepomuceno, sin una periodicidad establecida para la comprobación del estado del proyecto. El proyecto, además de estas reuniones, contempla la realización de una reunión de inicio del proyecto en la que se explica el alcance y objetivos del proyecto y una reunión de cierre del proyecto, en la que se revisará el trabajo realizado.

1.4 Análisis temporal y de costes de desarrollo

El proyecto constará de varias fases, las cuales se especifican en la siguiente tabla:

Fase	Inicio	Fin
Fase 1 - Planificación	24/02/2021	03/03/2021
Fase 2 - Estudio teórico	04/03/2021	19/03/2021
Fase 3 - Estudio del problema	19/03/2021	02/04/2021
Fase 4 - Desarrollo de la solución	03/04/2021	01/05/2021
Fase 5 - Experimentación	02/05/2021	30/05/2021
Fase 6 - Cierre del proyecto	31/05/2021	20/06/2021

Tabla 1. Planificación temporal del proyecto.

Cada fase tendrá distintas tareas implícitas que producirán un incremento al final de la fase. Las tareas y sus incrementos esperados se definen a continuación:

Fase 1 – Planificación

- **Tareas**
  - **Tarea 1.1** - Definición de la planificación temporal del proyecto
  - **Tarea 1.2** - Establecimiento de la metodología de trabajo
  - **Tarea 1.3** - Estudio de costes del proyecto
  - **Tarea 1.4** - Desarrollo de un plan de contingencia
- **Incremento** - Documentación de la planificación en la memoria

Fase 2 – Estudio teórico

- **Tareas**
  - **Tarea 2.1** - Introducción al Deep Learning
  - **Tarea 2.2** - Lectura de los capítulos 1, 2, 3 y 6 de *"Deep Learning con Python"*
  - **Tarea 2.3** - Búsqueda de información complementaria sobre Deep Learning aplicado a procesamiento de texto
  - **Tarea 2.4** - Estudio de las herramientas
- **Incremento** - Documentación del estudio teórico en la memoria

Fase 3 – Estudio del problema

- **Tareas**
  - **Tarea 3.1** - Búsqueda de artículos y experimentos de referencia: *Papers with code*
  - **Tarea 3.2** - Búsqueda y estudio de datasets
  - **Tarea 3.3** - Diseño teórico de la solución
- **Incremento** - Material escrito de la investigación del problema

Fase 4 – Desarrollo de la solución

- **Tareas**
  - **Tarea 4.1** - Preparación del entorno de desarrollo
  - **Tarea 4.2** - Desarrollo de la solución
  - **Tarea 4.3** - Estudio de posibles variaciones de la solución y posible desarrollo de las mismas
- **Incremento** - Memoria del desarrollo de la solución

Fase 5 – Experimentación

- **Tareas**
  - **Tarea 5.1** - Diseño y documentación de los experimentos
  - **Tarea 5.2** - Desarrollo de los experimentos y documentación de los resultados
  - **Tarea 5.3** - Análisis de los resultados y conclusiones
- **Incremento** - Documento de la experimentación

Fase 6 – Cierre del proyecto

- **Tareas**
    - **Tarea 6.1** - Revisión y corrección de errores
    - **Tarea 6.2** - Preparación de la presentación
- **Incremento** - Documento de la memoria finalizado y presentación

1.5 Costes

Para la realización del trabajo, se estima que semanalmente se invertirán 20 horas. Teniendo en cuenta que se ha planificado trabajo para 16 semanas, el total de horas invertidas será de unas 320 horas.

El sueldo mensual establecido por la dirección general de trabajo para un perfil de investigador en el colectivo de ámbito estatal para los centros de educación universitaria e investigación es de 1692,94€ de 40 horas semanales. Esto hace que cada hora de trabajo tenga un coste de 42,32€.

Teniendo en cuenta el dato anterior, aplicando el coste por hora de 42,32€ a las 320 horas estimadas, se concluye que el coste estimado para la realización de este proyecto es de 13542,40€.

Los costes de software son nulos, dado que se utilizarán librerías y entornos de desarrollo gratuitos.

El coste de hardware es igualmente nulo.

1.6 Plan de contingencia

En caso de que se encontrasen problemas durante el desarrollo del proyecto cuya solución no fuese factible en el periodo de tiempo establecido para el proyecto, se aplicaría el siguiente plan de contingencia:

- El estudio del Deep Learning, poniendo especial atención en su aplicación a secuencias de texto, se mantiene.
- El dominio del problema cambiaría, en lugar de generación de haikus con Deep Learning, el problema a resolver sería la generación de textos utilizando redes LSTM con Keras.

Con el fin de llevar esto a cabo, se ha realizado un breve estudio de la información disponible en la web y en concreto, se ha encontrado un reto de Kaggle que sería de ayuda para la replicación de la solución a este problema. También se ha encontrado un dataset para concretar aún más el dominio del problema, la generación de titulares de noticias.

1.7 Ejecución de la planificación

Durante el transcurso del proyecto logramos seguir la planificación correctamente hasta la tercera fase. En ella nos damos cuenta de que algo no está llevándose a cabo como se esperaba. Debieron hacerse algunos ajustes a lo largo de la ejecución de la misma para poder completar en tiempo el proyecto.

Emplazamos a leer el Capítulo 6, en el que se explica cómo se llevó a cabo finalmente la planificación del trabajo.

Capítulo 2. CONTEXTO DEL PROBLEMA

2.1 Haikus

El haiku es un género poético originario de Japón que en las últimas décadas, gracias a la globalización, está siendo conocido en todo el planeta. A continuación explicaremos en detalle qué es un haiku y qué lo diferencia de cualquier otro poema, apoyándonos en la explicación dada en [2]. – Todos los haikus mostrados en este capítulo han sido extraídos de [3] –

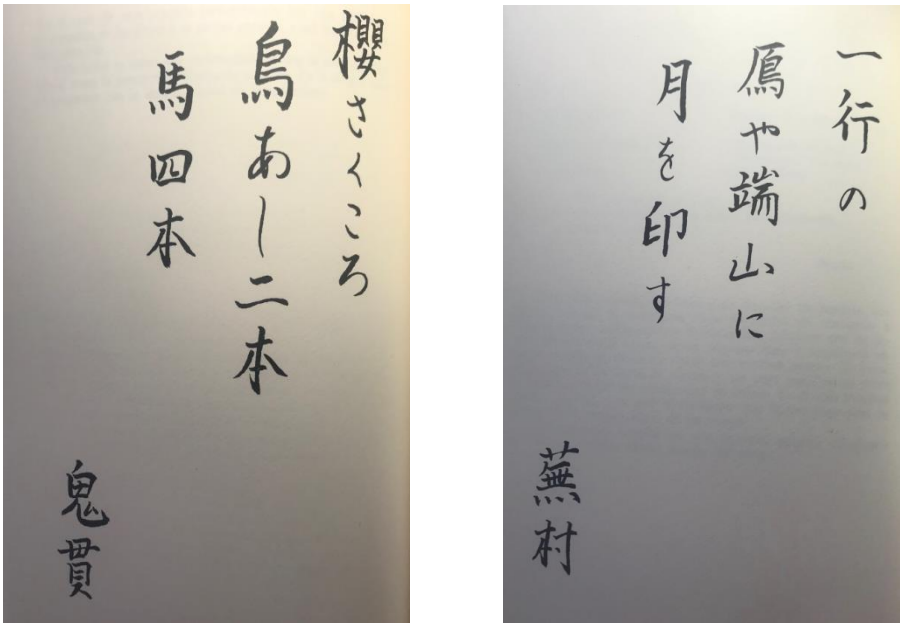


Figura 1. Haikus escritos a mano en japonés.

Los haikus son poemas de tres versos sin rima, con estructura 5-7-5 en sus sílabas. Aunque éste es el estándar, existe cierta flexibilidad en el número de sílabas de los versos. En realidad, tradicionalmente los haikus no se miden en sílabas, sino en *moras*. Una *mora* es una unidad fonética más breve que una sílaba. Esto permite que las 17 sílabas que constituirían un haiku en japonés puedan tener alguna sílaba menos en otros idiomas.

Los haikus, además de mantener esta estructura tan característica, deben poseer otros rasgos. Uno de ellos, tradicionalmente, es el *kigo*, palabra o expresión introducida en el poema que hace intuir la época del año a la que se refiere el poema. El *kigo* no tiene por qué ser explícito, mencionando un mes o una estación, algunas expresiones que pueden ser consideradas *kigo* son referentes a elementos que están presentes en ésta época del año, como podría ser *"cerezos en flor"* para referirse a la primavera.

No todos los haikus han de poseer un *kigo*, pero la tradición dicta que así debe ser para que sea considerado perteneciente a este género.

Por lo general, los haikus plasman escenas de la naturaleza o de la vida cotidiana. Suelen incluir animales, plantas, paisajes o fenómenos naturales. El protagonista del poema suele ser el ambiente, no la persona que habla. Esto no quiere decir que el poeta no pueda expresar sus sentimientos, sino que debe hacerlo siempre para ensalzar el entorno.

Haru / no / umi	*Primavera / (= posesivo) / mar*
hinemosu / notari	*todo el día / ondulando*
notari / kana	*ondulando / (admiración)*

El mar en primavera
con su vaivén de olas
sin fin todo el día.

Buson

El haiku debe transmitir la impresión que ha causado la contemplación de algo. Esto puede ser belleza, armonía, serenidad, fugacidad, melancolía… y, como dijimos anteriormente, la expresión debe mostrarla la escena, más que el poeta. [2]

La escritura de un haiku ha de ser sencilla y natural. "Un haiku es un dedo que apunta a la luna, pero si el dedo está ensortijado, el lector se fijará en el dedo y no en la luna" [2]. No ha de recargarse el poema con figuras literarias que distraigan de la lectura. Las palabras no deben llamar la atención por si solas. No obstante, nada impide utilizar, por ejemplo, una metáfora, siempre que la función que desempeña sea ayudar a la descripción de la escena o a la transmisión de la impresión poética.

Sakura / saku / koro	*Cerezo / florecer / época*
tori / ashi / nihon	*pájaro / pata / dos piezas*
uma / shihon	*caballo / cuatro piezas*

Cuando florece el cerezo,
dos patas para el pájaro,
cuatro para el caballo.

Onitsura

Capítulo 3. ESTADO DEL ARTE

3.1 Redes neuronales

Las redes neuronales pertenecen al campo de la Inteligencia Artificial. Su nacimiento y su nombre se deben al intento de replicación del funcionamiento de un cerebro humano en un ordenador.

La esencia es que unos datos entran en un cerebro, son procesados y éste devuelve una respuesta. [4]

La unidad básica de una red neuronal es la neurona simple o perceptrón. La información es introducida en esta unidad, es procesada y genera una salida. En concreto, un perceptrón tiene varias entradas con un peso cada una. Si la suma de las entradas supera un umbral la salida será uno, si no, será cero. [5]

Así pues, basándonos en [6] podríamos concretar que la neurona simple está formada por:

- Entradas. Datos que se introducen en la neurona.

- Sesgo. Un valor constante que se le suma a la entrada de la función de activación de la neurona. Normalmente es 1.

- Pesos. Valores que modifican el valor introducido en una entrada. Estos pesos se ajustarán a medida que se lleve a cabo el entrenamiento de la red neuronal y son la razón de que un modelo sea capaz de predecir correctamente un objetivo para problemas concretos. Explicaremos esto en más profundidad a continuación.

- Función de activación. Se aplica al sumatorio de todas las entradas una vez han sido tratadas por sus pesos. El resultado constituye la salida de la neurona. Esta función permite estimar relaciones no lineales entre los datos. Existen muchas funciones de activación, tales como las lineales, logísticas o hiperbólicas.

- Salida. Resultado de la función de activación.

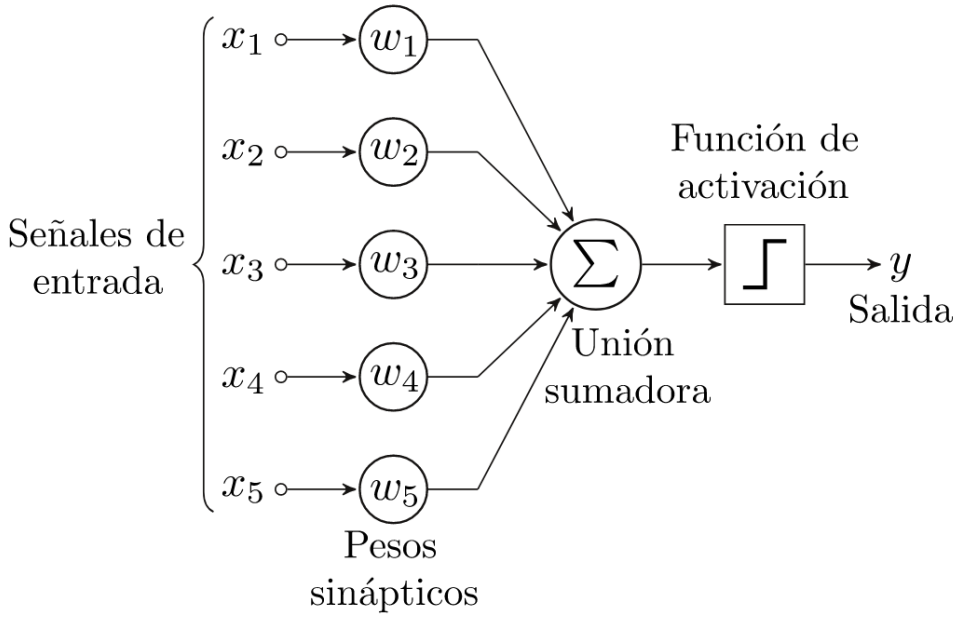


Figura 2. Arquitectura del perceptrón.

Ahora sabemos, en esencia, cómo es la unidad básica de una red neuronal. Las neuronas pueden unir sus salidas a entradas de otras neuronas, creando enlaces que modificarán el estado de activación de las neuronas adyacentes. Esto, unido a su entrenamiento, hará que las redes neuronales resulten más útiles para resolver problemas en los que las características sean difíciles de expresar con la programación convencional. [6]

La forma en que se unen las neuronas es jerárquica, formando niveles. La profundidad en niveles podrá crecer de manera indefinida hasta que finalmente se concluya devolviendo la salida final.

Así pues, podemos distinguir distintos tipos de capas en las redes neuronales según su jerarquía.

- Capas de entrada. Son las capas iniciales de una red neuronal.
- Capas intermedias u ocultas.
- Capas de salida. Son las capas finales que devuelven el resultado final de la red neuronal.

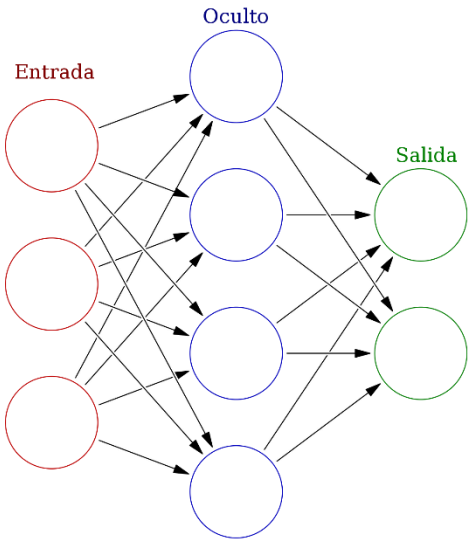


Figura 3. Arquitectura simple de una red neuronal.

3.2 Funcionamiento de las redes neuronales

Las redes neuronales son modelos que no están diseñados con el fin de ajustar pares (entrada, salida), sino de aumentar el conocimiento estructural de los datos disponibles y de futuros datos que provengan del mismo contexto del problema. [7]

Para conseguir su objetivo, las redes neuronales tratan de asignar unas entradas a determinados objetivos. Esto lo hacen mediante la observación de varios ejemplos de entradas y salidas que les mostremos. [1]

Los datos de ejemplo que utilizamos para mostrar a la red neuronal el contexto del problema son los datos de entrenamiento.

Los datos de entrenamiento, como dijimos antes, sirven para ajustar las salidas que proporciona la red en función de determinadas entradas. El proceso de ajuste es llamado entrenamiento.

En el entrenamiento se modifican los pesos de cada una de las neuronas de manera que las funciones de activación devuelvan valores óptimos para conseguir la salida esperada. En este contexto, “aprender” significa lograr una combinación de valores para los pesos de todas las capas de manera que el modelo entrenado asocie correctamente a los valores de entrada sus objetivos asociados. [1]

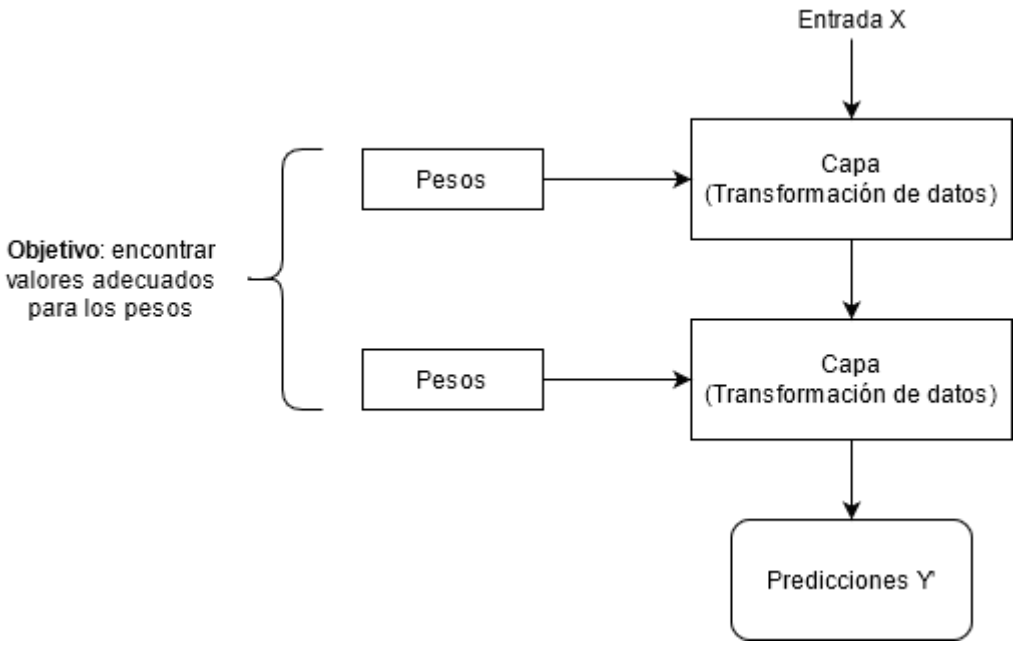


Figura 4. Objetivo del entrenamiento.

La cuestión es que esta tarea se complica conforme aumenta el número de parámetros que deben modificarse, sobre todo si tenemos en cuenta que modificar el valor de uno de los pesos afectará al comportamiento del resto.

Para ser capaces de controlar algo, lo primero es ser capaces de medirlo. Para ello es necesario hacer uso de una *"función de pérdida*", también llamada "*función objetivo*". Esta función toma las predicciones calculadas por la red y las esperadas y calcula una distancia, lo que especifica cómo de bien ha funcionado la red para ese ejemplo. [1]

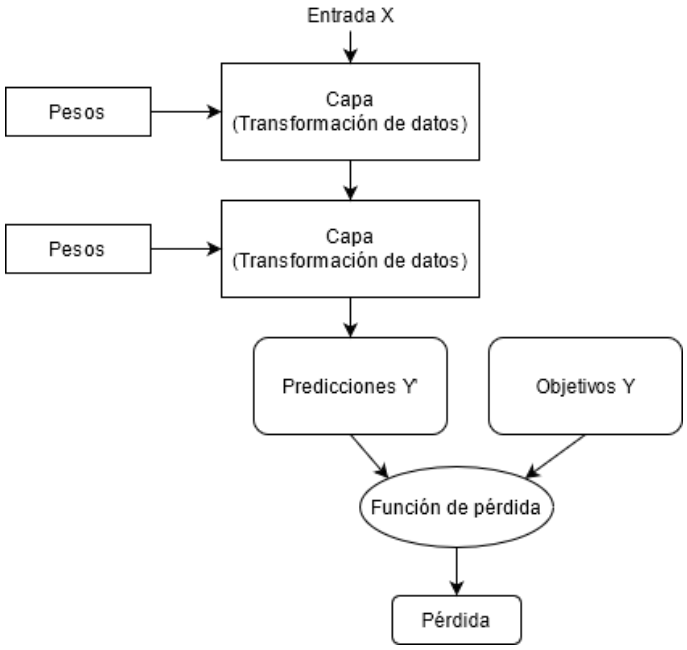


Figura 5. Entrenamiento con cálculo de pérdida.

Una vez tenemos este valor, lo emplearemos como señal de retroalimentación para el ajuste de los pesos en una dirección en que éste valor de pérdida se reduzca. Así aparece la figura del "*optimizador*", que implementa el algoritmo de "*retropropagación*".

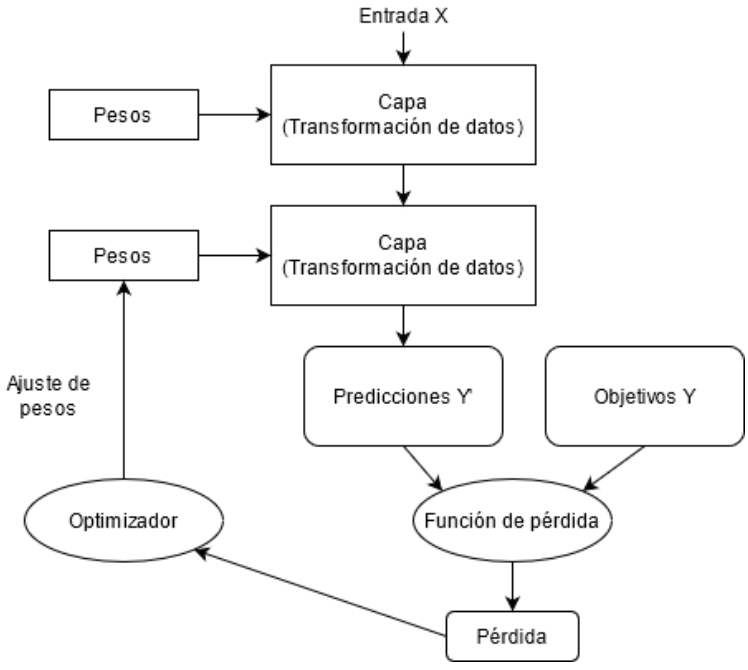


Figura 6. Entrenamiento con optimizador.

En principio, es frecuente el uso de pesos aleatorios iniciales en las redes. Luego, mediante el algoritmo de "*retropropagación"* se actualizan. Esto puede repetirse varias iteraciones de

manera que se crea un "bucle de entrenamiento", el cual, tras producirse suficientes veces produce valores para los pesos que minimizan la función de pérdida.

A continuación explicaremos cómo funciona el algoritmo de "*retropropagación*" más en profundidad. Es importante comprender bien este algoritmo, dado que es la pieza clave del *Deep Learning*.

Para empezar, explicaremos conceptos básicos matemáticos que son necesarios para comprender correctamente el funcionamiento de este algoritmo.

- **Derivada**. Dada una función diferenciable – que se puede derivar - *f(x),* su derivada *f'(x)* representa la aproximación lineal local de *f* que se produce en esos puntos. Es decir, la derivada de *f(p), f'(p)* representa la pendiente *a* de la recta tangente a *f* en el punto *p*. El valor de *a* determinará cuánto afecta un cambio en *x* en torno a *p*. Si *a* es negativa, un pequeño cambio positivo en *x* resultará en un descenso de *f(x)*, si *a* es positiva, un pequeño cambio positivo en *x* será un aumento de *f(x)* (como muestra la figura). [1]

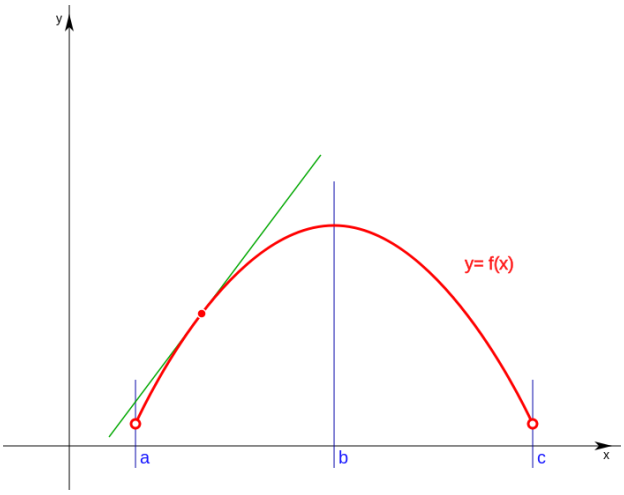


Figura 7. Derivada de f(x) en un punto p.

- **Gradiente**. El gradiente es, explicándolo brevemente, la generalización del concepto de derivadas a funciones con entradas multidimensionales. Dada una entrada mutidimensional *W*, la derivada de *f* en el punto *W0*, *f(W0)* es un *gradiente tensorial* – el equivalente en este caso a la pendiente para funciones lineales - con la forma de *W*, donde cada gradiente de los coeficientes *f(W0)* indica la dirección y magnitud con la que debe cambiar cada coeficiente para minimizar la función de pérdida, *f*. [1]

  El gradiente, como podemos ver, describe la curvatura de *f(W)* en torno a *W0.* Podemos reducir el valor de *f(W)* moviendo *W* en la dirección opuesta al gradiente. Por ejemplo:

$$W_1 = W_0 - paso * gradiente(f)(W_0)$$

Como podemos ver, es necesario establecer un pequeño "*paso"* como factor de escalada, el cual es necesario para no alejarnos demasiado de *W0* ya que el gradiente solo aproxima la curvatura a *W0* cuando estamos cerca de él. [1]

- **Descenso de Gradiente Estocástico**. Este algoritmo es el que hace posible que se puedan conseguir los valores mínimos de una función de pérdida. Lo ideal sería encontrar la solución de la ecuación:

$$gradiente(f)(W) = 0$$

Sin embargo, esto actualmente resulta imposible para la gran mayoría de problemas en los que necesitaríamos lidiar con decenas de millones de parámetros. Es por ello que se desarrolló el algoritmo de *Descenso de Gradiente Estocástico*. El algoritmo, apoyándonos en [1] para explicarlo, funciona de la siguiente manera:

1. Dibujamos un lote de muestras de entrenamiento *x* y los correspondientes objetivos *y.*
2. Ejecutamos la red sobre *x* para obtener las predicciones *y_pred*.
3. Computamos la pérdida de la red en el lote para medir el desajuste entre *y_pred* e *y.*
4. Computamos el gradiente de la pérdida respecto a los parámetros de la red (una propagación hacia atrás)
5. Movemos ligeramente los parámetros en dirección opuesta al gradiente, por ejemplo, $W -= paso * gradiente$, reduciendo así un poco la pérdida en el lote.

De esta forma, aplicando el algoritmo repetidas veces – y normalmente con distintos lotes – podemos ser capaces de alcanzar eventualmente un valor mínimo en la función de pérdida. Es muy importante tener en cuenta el valor del "*paso*", también conocido como "*learning rate"*, dado que un valor grande llevaría a ubicaciones aleatorias de la curva y un valor excesivamente pequeño podría hacer que nos estancáramos en un mínimo local.

Existen muchas variantes de este algoritmo, diferenciándose la mayoría en el hecho de tener en cuenta otros factores como podrían ser el "momento". Estas variantes son conocidas como "optimizadores".

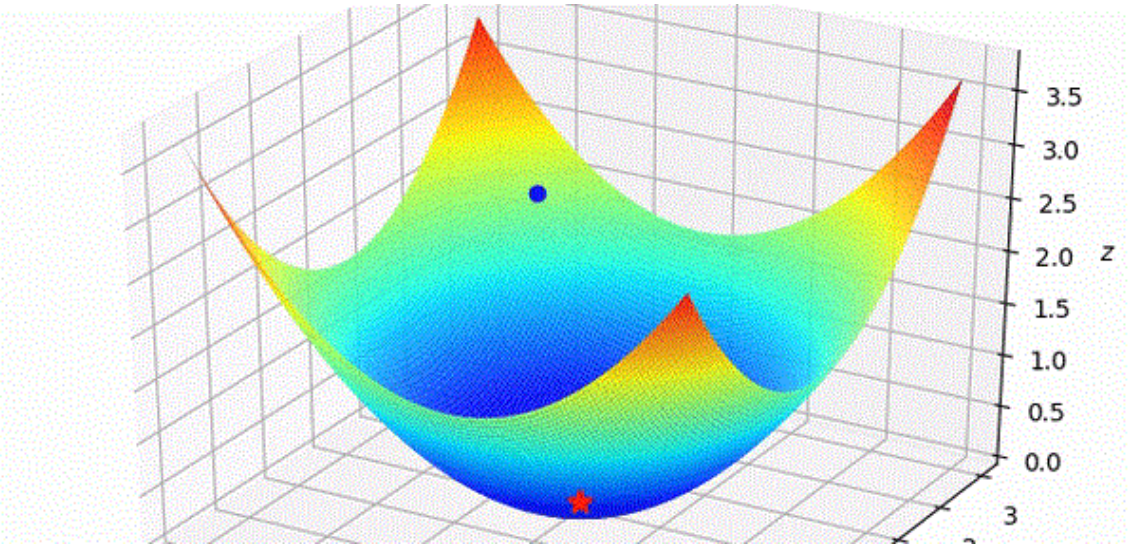


Figura 8. Descenso de gradiente.

Una vez hemos explicado los conceptos básicos anteriores, podemos explicar el algoritmo de "***retropropagación***".

En la práctica, una función de red neuronal consiste en varias operaciones con tensores encadenadas, cada una de ellas con una derivada conocida simple. Por ejemplo, esta sería una red *f* formada por tres operaciones con tensores *a, b* y *c* y con matrices de peso *W1, W2* y *W3*:

$$f(W1, W2, W3) = a\Big(W1, b\big(W2, c(W3)\big)\Big)$$

Si aplicamos la regla de la cadena:

$$f\big(g(x)\big) = f'\big(g(x)\big) * g'(x)$$

Podríamos conseguir computar los valores de gradiente de la red neuronal en cada una de sus capas, propagando hacia atrás los valores. A esto se le conoce como "*retropropagación*". Comienza con el último valor de pérdida y va hacia atrás pasando por cada capa hasta llegar a la inferior, computándose en el proceso la contribución de cada parámetro al valor de pérdida. [1]

Esta es la base de las redes neuronales y del *Deep Learning*. Existen diferentes funciones de pérdida, cada una usada en un tipo de problema distinto, al igual que funciones de activación, que controlan la salida de cada una de las neuronas de una capa.

De estas últimas podemos destacar las siguientes:

- La función **sigmoide**. Transforma valores de rangos infinitos en una probabilidad entre 0 y 1.
- La función **softmax**. Asigna probabilidades decimales a cada clase en un problema de clases múltiples. [8]
- La función **ReLu**. Su nombre abrevia "*rectified linear unit*", puede definirse como la función *f(x) = max(0, x)*, es decir, cualquier valor negativo se convertiría en 0, el resto se

mantiene tal como está. Cabe destacar que el 0 es el umbral por defecto y más usado, pero puede cambiarse.

3.3 Deep Learning

El *Deep Learning* es un campo del *Machine Learning*. Este campo de la Inteligencia Artificial se dedica a la investigación y aplicación de técnicas que le permiten a un ordenador aprender sin ser explícitamente programadas. Existen muchos algoritmos de aprendizaje automático, entre los que se encuentran el aprendizaje por refuerzo, los algoritmos genéticos, las reglas de asociación, los árboles de decisión, los algoritmos de agrupamiento o de *clustering*, las máquinas de vectores y las redes neuronales.

El *Deep Learning* hace uso de redes neuronales para lograr el aprendizaje de sucesivas capas de representaciones cada vez más significativas de los datos. El "profundo" en *Deep Learning* hace referencia a la cantidad de capas de representaciones que se utilizan en el modelo. Por lo general suele hacer uso de decenas e incluso cientos de capas. Cada una de ellas aprende automáticamente a medida que se entrena al modelo con datos. [6]

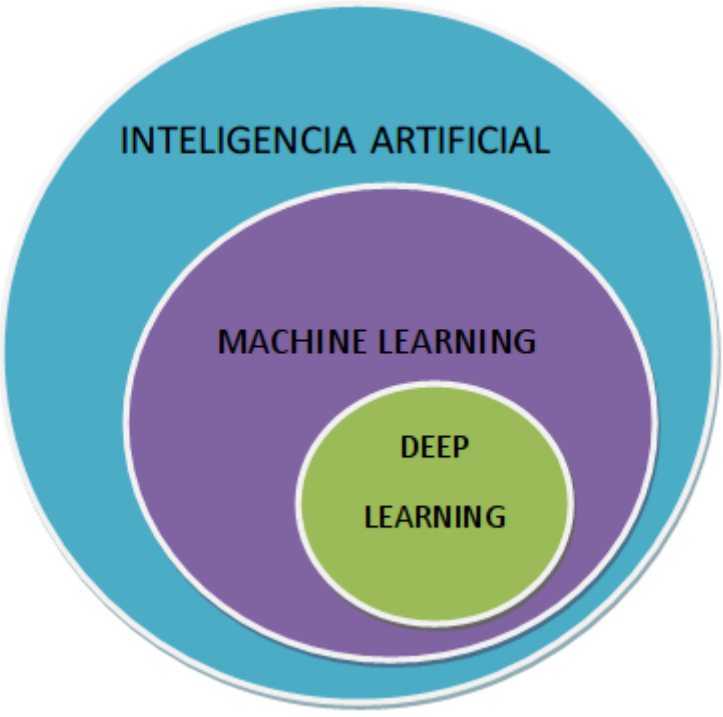


Figura 9. Organización de campos de la Inteligencia Artificial.

Haciendo uso de los algoritmos explicados en el anterior capítulo, este campo de la Inteligencia Artificial se encuentra actualmente en un periodo de auge y aún no se sabe a ciencia cierta hasta dónde alcanzarán los descubrimientos y avances en él.

A pesar de que en el anterior epígrafe hemos explicado las bases de las redes neuronales y su funcionamiento, para el contexto del problema debemos explicar algunos conceptos que se detallarán a continuación.

3.4 Redes neuronales recurrentes

Hasta ahora hemos hablado de redes neuronales que no tienen memoria. La entrada que reciben es procesada y devuelven una salida. Si necesitáramos tratar secuencias temporales con este concepto, necesitaríamos mostrarle la serie completa a la red en un vector grande.

Una red neuronal recurrente (RNR) procesa secuencias iterando por los elementos de las mismas y manteniendo un estado que contiene información relativa a lo que ha visto hasta el momento. Es una red neuronal con un bucle interno. El estado se reinicia entre procesamientos de dos secuencias independientes diferentes. Este tipo de redes procesa la información haciendo uso de una única entrada en la red, pero varios pasos, recorriendo los elementos de la secuencia. [1]

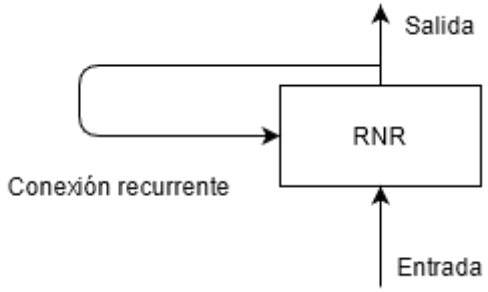


Figura 10. Red neuronal recurrente.

Para explicarlo de manera resumida, una RNR es un bucle *for* que reutiliza cantidades computadas durante la iteración anterior del bucle. A continuación mostramos gráficamente el funcionamiento de una RNR sencilla para una sola secuencia con varios pasos temporales. [1]

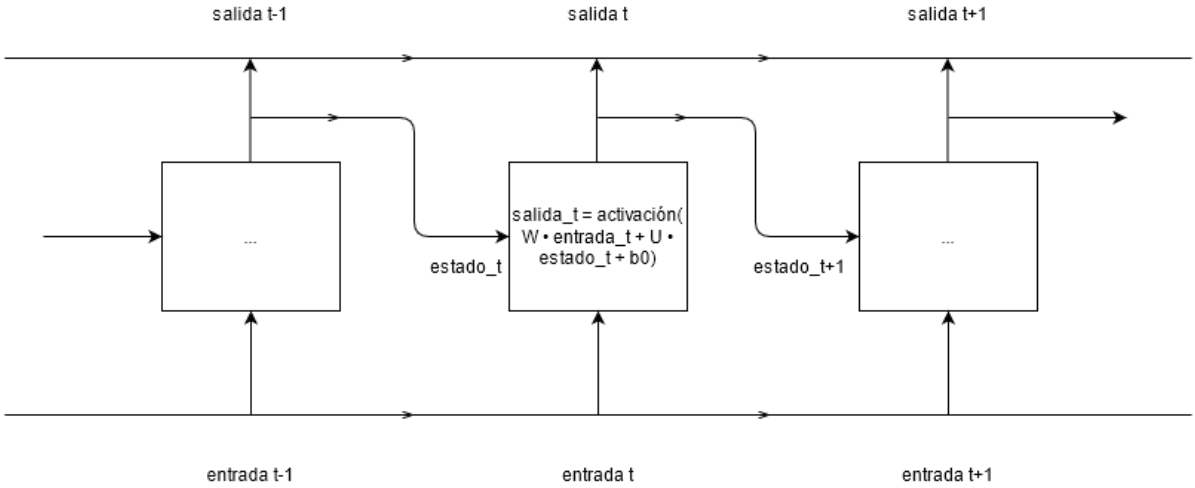


Figura 11. RNR sencilla desplegada con el tiempo.

En ***Keras*** existen varias capas recurrentes de las que podemos hacer uso. La más importante para nosotros – dado que es la capa que usaremos en nuestros experimentos – será la capa ***LSTM***.

LSTM , *Long-Short-Term Memory*, es un algoritmo desarrollado por Hochreiter y Schmidhuber en 1997. La principal diferencia con la RNR sencilla que explicamos anteriormente es que es capaz de transmitir información por varios pasos de tiempo. La idea detrás de esto es mitigar el desvanecimiento de información gradual que se produce durante el procesamiento. [1]

Vamos a añadir al diagrama anterior un flujo de datos adicional que lleve información por los pasos de tiempo. Llamaremos a sus valores en distintos pasos de tiempo *C* t, donde *C* significa *carry* (transportar). La información se combinará con la conexión de entrada y la conexión recurrente mediante un producto escalar y afectará al estado que se enviará al siguiente paso de tiempo. [1]

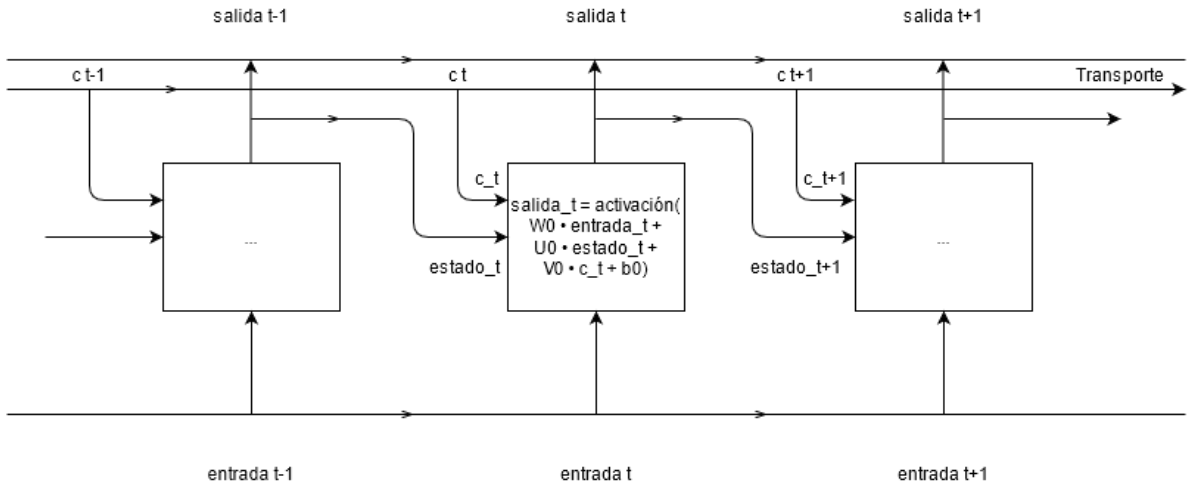


Figura 12. Paso de RNR sencilla a LSTM añadiendo pista de transporte.

Ahora necesitamos especificar cómo se computa el valor del flujo de datos de transporte. Se realizan tres transformaciones de la misma forma que se realizaban para una célula de la RNR sencilla.

Cada una de las transformaciones tiene sus propias matrices de peso, que indexaremos con las letras *i, f* y *k.* [1] Ahora tenemos:

$$salida_t = activación(estado_t \bullet U_0 + entrada_t \bullet W_0 + C_t \bullet V_0 + b_0)$$

$$i_t = activación(estado_t \bullet U_i + entrada_t \bullet W_i + b_i)$$

$$f_t = activación\big(estado_t \bullet U_f + entrada_t \bullet W_f + b_f\big)$$

$$k_t = activación(estado_t \bullet U_k + entrada_t \bullet W_k + b_k)$$

De manera que ahora el diagrama quedaría de esta forma:

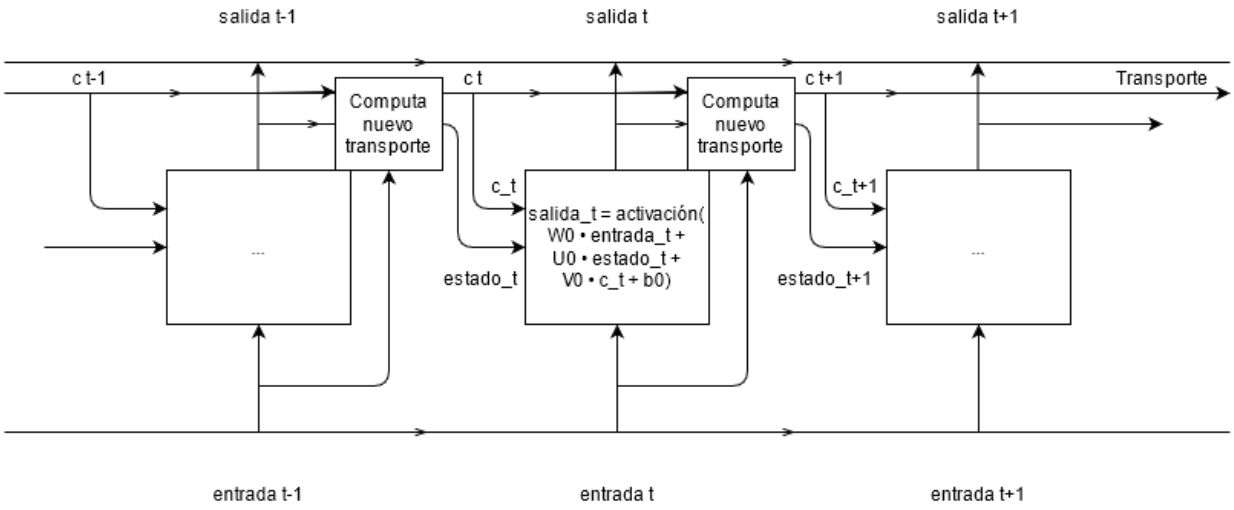


Figura 13. Anatomía de una LSTM.

Realmente no podemos saber a ciencia cierta qué hace cada una de las operaciones anteriores. Lo que hacen estas operaciones está determinado en realidad por los pesos que las parametrizan, que a su vez están siendo aprendidos en cada ronda de entrenamiento, por lo que se hace imposible otorgar un propósito específico a cada operación. Lo que hemos hecho puede interpretarse mejor como un conjunto de "limitaciones" en la investigación, más que como un "diseño" en un sentido ingeniero. En resumen, no es necesario entender nada sobre la arquitectura específica de una célula LSTM, lo importante es comprender lo que realiza: deja pasar información para reinyectarla después, tratando de combatir el problema del desvanecimiento del gradiente. [1]

3.5 Redes recurrentes generativas

El desarrollo del algoritmo LSTM abrió las puertas a un mundo de experimentación en el campo de las redes neuronales para el tratamiento de datos secuenciales. Una de las vertientes de estos experimentos fue la generación de datos secuenciales.

Normalmente en *Deep Learning* se entrena una red para predecir algo como consecuencia de unas entradas dadas. En el caso de los problemas de multiclase, normalmente se predice una distribución de probabilidad de que suceda cada una de las clases. Esto, trasladado a texto, podría ser la probabilidad de que una frase continúe con un carácter en concreto. A esto se le llama "modelo de lenguaje". Un modelo de lenguaje capta el "espacio latente" del lenguaje, su estructura estadística.

Una vez tenemos un modelo de lenguaje, podemos crear nuevas secuencias. Primero será necesario introducir una primera cadena de texto, que serán los "*datos condicionantes*". Una vez hecho esto podemos comenzar a generar caracteres a partir de esta cadena, añadirlos al final y continuar generando con la nueva cadena.

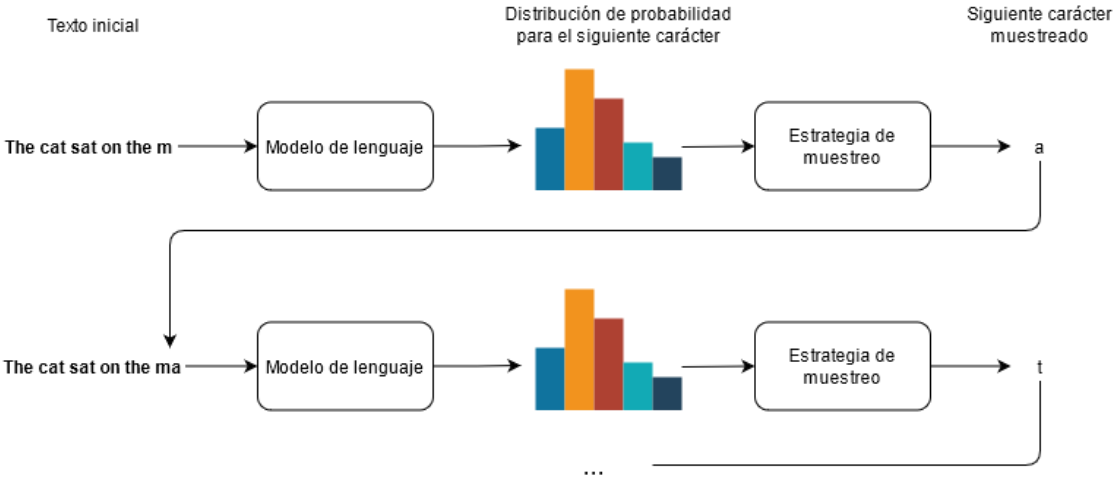


Figura 14. Proceso de generación de texto carácter por carácter con un modelo de lenguaje.

Es importante escoger una estrategia de muestreo adecuada. La intuición nos haría pensar que el "muestreo avaricioso", que consiste en escoger siempre el carácter más probable, sería la mejor opción, pero no suele ser así. El resultado de esto suele ser la generación de texto predecible e incoherente.

La aproximación más usada es el "muestreo estocástico", que introduce cierta aleatoriedad haciendo uso de un parámetro llamado *"temperatura estocástica*". En la práctica, lo que hace este valor es alterar la distribución de probabilidad, cambiando sus pesos. A temperatura más baja, los caracteres obtenidos son más predecibles, a temperatura más alta, los caracteres se vuelven más sorprendentes y desestructurados.

Poniendo en práctica lo anterior – siguiendo el esquema visto en [1] – podemos generar texto con un modelo entrenado, una cadena de texto inicial y escogiendo un valor de temperatura repitiendo el siguiente proceso:

1. Extraer del modelo la distribución de probabilidad del siguiente carácter, teniendo en cuenta el texto generado hasta ahora.
2. Cambiar los pesos de la distribución en función de la temperatura determinada.
3. Muestrear el siguiente carácter al azar según la distribución con los pesos modificados.
4. Añadir el carácter al final del texto disponible

Siguiendo los pasos anteriores podemos generar cualquier tipo de texto. El texto generado dependerá sobre todo de los datos con los que se haya entrenado la red neuronal.

3.6 Preprocesamiento de texto

Para llevar a cabo este proyecto necesitaremos entrar brevemente en el campo del Procesamiento del Lenguaje Natural para aprender a tratar las secuencias de texto, de manera que puedan ser utilizadas adecuadamente.

Las redes neuronales solo trabajan con números. Esto, cuando trabajamos con texto, nos obliga a realizar un tratamiento de los datos de manera que puedan ser usados por el modelo. A esta etapa anterior al entrenamiento se la conoce como preprocesamiento.

Antes de utilizar un dataset para entrenar un modelo, hay numerosas acciones que podemos llevar a cabo para intentar mejorar el rendimiento que tendrá. El filtrado de muestras para descartar las muestras de peor calidad podría ser una de ellas.

En esta sección nos centraremos en las conversiones de cadenas de texto a tensores que puedan ser utilizados por una red neuronal.

3.6.1 Codificación one-hot

La codificación one-hot es la forma más básica y usual de convertir un *token* en un vector. Consiste en asociar un índice entero único con cada palabra y luego convertir ese índice entero *i* en un vector binario de tamaño *N* (el tamaño del vocabulario); el vector es todo ceros, salvo la entrada *i,* que es un 1. [1]

Esta codificación puede ser utilizada tanto a nivel de palabra como a nivel de carácter.

En un vocabulario de 27 caracteres ordenado alfabéticamente, la letra *a* estaría codificada de la siguiente manera:

$$[1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ ]$$

Esta forma de procesar el texto resulta útil cuando el tamaño del vocabulario no es demasiado extenso. Si los vocabularios fuesen de gran magnitud, estaríamos derrochando espacio. Como podemos ver de todos los escalares en el vector, solamente uno es significativo, lo cual nos podría hacer pensar que estamos utilizando realmente $\frac{1}{N}$ del tamaño que ocupamos. Esto realmente no es cierto, puesto que la posición del escalar "significativo" es lo que realmente importa, por tanto, el resto de ceros cobran importancia.

Sin embargo, eso no evita que para vocabularios de gran tamaño este tipo de codificación sea muy exigente en cuanto a espacio, por lo que es necesario hacer uso de otra técnica en ese caso.

3.6.2 Embeddings de palabras

Los vectores obtenidos mediante one-hot son binarios, dispersos y con alta dimensionalidad (mismas dimensiones que número de palabras en el vocabulario). Los *embeddings* de palabras son vectores de punto flotante densos de baja dimensionalidad. Los *embeddings* de palabras suelen tener 256 dimensiones, 512 dimensiones o 1024 dimensiones al tratar con vocabularios muy amplios. Por otro lado, one-hot suele conducir a vectores de 20000 dimensiones o más, si los vocabularios son de 20000 palabras o más. Los embeddings de palabras recogen mucha más información en muchas menos dimensiones. [1]

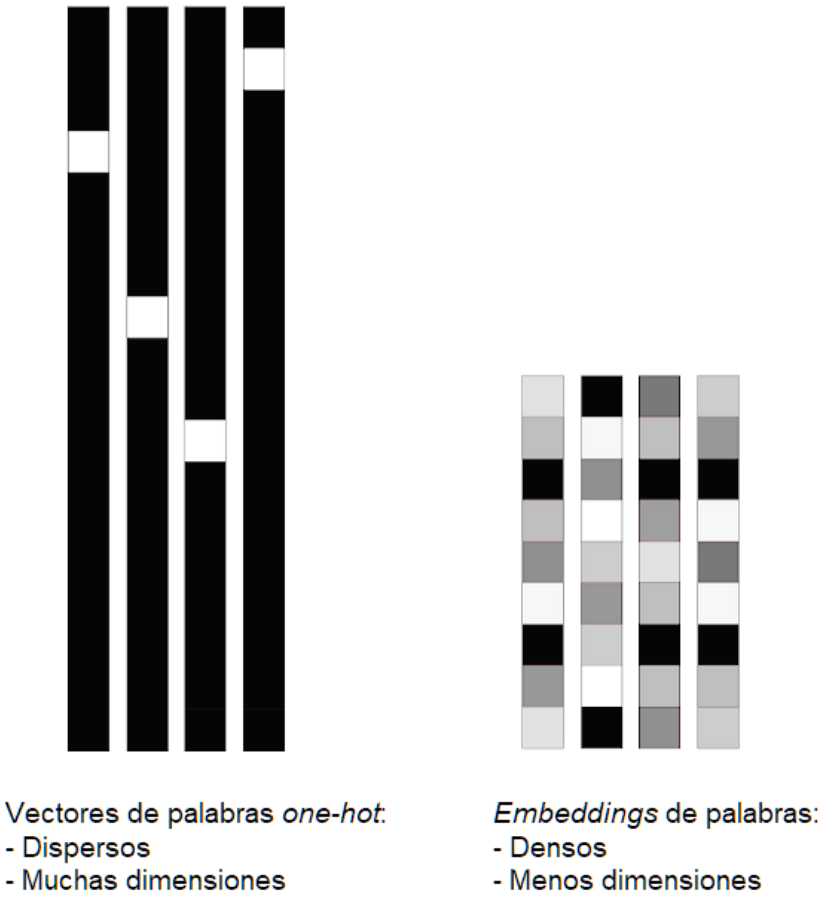


Figura 15. Comparación de vectores one-hot y embeddings de palabras.

Existen dos formas de conseguir *embeddings* de palabras. Pueden aprenderse para el contexto del problema que se necesite o pueden cargarse en un modelo *embeddings* precomputados para una tarea distinta a la que tratamos de resolver. A estos últimos se les llama "preentrenados".

Los *embeddings* de palabras pretenden asignar lenguaje humano a un espacio geométrico. Por ejemplo, en un espacio de *embedding* razonable, los sinónimos deberían estar en vectores de palabras similares. En general, la distancia entre los vectores de las palabras debería relacionarse con la distancia semántica entre las palabras asociadas (las palabras con significados diferentes alejadas entre sí y las relacionadas más cerca). Además de la distancia, también intervendrían las direcciones.

Explicaremos esto de una manera más visual – tal y como se explica en [1] –.

Veremos en un espacio bidimensional cómo podrían representarse las palabras "lobo", "perro", "tigre" y "gato". El mismo vector que no permite pasar de "perro" a "lobo" nos permitiría pasar de "gato" a "tigre". El vector que une estas palabras podría ser "de doméstico a salvaje". También podríamos encontrar otro vector que nos permitiría ir de "perro" a "gato" y de "lobo" a "tigre", haciendo referencia a una relación "de cánido a felino".

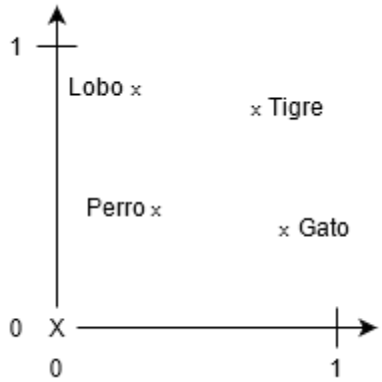


Figura 16. Ejemplo de juguete de un espacio de embedding de palabras.

En espacios reales, los embeddings suelen representar miles de vectores como estos, en muchas más dimensiones. Los más habituales podrían ser los de transformaciones geométricas de "género" y "plural".

Para aprender *embeddings* de palabras específicos para el problema que estemos tratando necesitamos tokenizar las palabras, asignarles un número único que luego podamos utilizar para "traducirlas" de vuelta. En ***Keras*** se puede hacer uso de una capa ***Embedding*** para llevar a cabo esta tarea. Esta capa funciona como un diccionario que asigna índices enteros (las palabras) a vectores densos. Toma enteros como entrada, los busca en un diccionario interno y devuelve los vectores asociados. Estos vectores se ajustan gradualmente mediante la retropropagación. [1]

Conforme el entrenamiento vaya pasando, los *embeddings* de palabras se irán refinando. Los *embeddings* específicos son más efectivos cuando hay conjuntos extensos de datos.

Hacer uso de *embeddings* de palabras preentrenados es sencillo usando ***Keras***, solo debemos preocuparnos de asociar los *tokens* que hemos establecido con los *tokens* de los *embeddings* preentrenados. Luego, necesitaremos cargar los pesos en una capa de ***Embedding*** que dejemos congelada para que no siga entrenándose. En contraposición con los *embeddings* específicos, los preentrenados resultan efectivos en problemas donde hay escasez de datos de entrenamiento.

En el próximo capítulo profundizaremos y llevaremos a cabo ejemplos prácticos de estas técnicas.

Capítulo 4. CASOS DE ESTUDIO

4.1 Clasificadores de críticas cinematográficas de IMDB

Para tomar contacto por primera vez con las redes neuronales y ***Keras***, trataremos de replicar los clasificadores de críticas de IMDB tal y como se explican en el capítulo 6 del libro *"Deep Learning con Python"*, de François Chollet [1]. Los modelos que se estudiarán se centran en el uso de embeddings de palabras.

En primer lugar, trataremos de replicar el primer clasificador que hace uso de embeddings de palabras específicos, los cuales establecen relaciones entre las palabras conforme avanza el entrenamiento del modelo.

El dataset que se va a utilizar para el primer experimento es uno preinstalado en el módulo ***datasets*** de ***Keras***. Las críticas en este conjunto de datos pueden ser limitadas por palabras. Este límite de palabras hará que se tomen las que más veces aparecen en la crítica. Las críticas pueden tener como valores objetivo 0 o 1, dependiendo de si es negativa o positiva, respectivamente. Además, las palabras de las críticas ya están tokenizadas y procesadas como números, de manera que pueden ser introducidas directamente en la red neuronal.

Importamos en primer lugar el dataset y las herramientas de preprocesamiento de ***Keras***.

```
from keras.datasets import imdb
from keras import preprocessing
```

Código 1. Librerías importadas para el primer caso de estudio

Ahora, tomaremos las 10000 palabras más habituales de cada crítica y las limitaremos a las primeras 20. También almacenaremos los valores objetivo.

```
max_palabras = 10000 #Número de palabras más habituales que se utilizarán, en nuestro caso, 10000
maxlen = 20          #Número de palabras tras las que se cortará la crítica

(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words = max_palabras)

x_train = preprocessing.sequence.pad_sequences(x_train, maxlen = maxlen)
x_test = preprocessing.sequence.pad_sequences(x_test, maxlen = maxlen)
```

Código 2. Preprocesamiento de datos de IMDB

A continuación crearemos el modelo. Comenzaremos con una capa de ***Embedding***, a la cual especificaremos que el vocabulario tiene 10000 palabras. También estableceremos que los embeddings serán de 8 dimensiones y que las frases serán de longitud ***maxlen***.

Tras esto, necesitaremos aplanar la salida de la capa anterior, por ello haremos uso de una capa ***Flatten***.

El paso anterior es necesario para que la salida de la capa de ***Embedding*** pueda ser utilizada por una última capa ***Dense*** con activación ***sigmoide***, de manera que dé como resultados 0 o 1.

Como optimizador haremos uso de ***RMSProp***, estableceremos la función de pérdida ***binary_crossentropy*** y monitorizaremos la precisión a lo largo de las épocas.

Entrenaremos el modelo durante 10 épocas en lotes de 32 muestras, reservando en cada época el 20% de las muestras para realizar validación.

```
from keras.models import Sequential
from keras.layers import Flatten, Dense, Embedding

model = Sequential()
model.add(Embedding(10000, 8, input_length = maxlen)) #Tras esta capa el tensor resultante
                                                      #es de la forma (muestras, maxlen, 8)

model.add(Flatten()) #Esta capa aplana el tensor anterior, quedando con la siguiente forma:
                     #(muestras, maxlen * 8)

model.add(Dense(1, activation = 'sigmoid')) #Este es el clasificador, con una sola neurona
                                            #que dará como resultado 0 o 1
model.compile(optimizer = 'rmsprop', loss = 'binary_crossentropy', metrics = ['acc'])
model.summary()

history = model.fit(x_train, y_train, epochs = 10, batch_size = 32, validation_split = 0.2)
```

Código 3. Modelo clasificador de críticas de IMDB con embeddings de palabras específicos.

```
Train on 20000 samples, validate on 5000 samples
Epoch 1/10
20000/20000 [==============================] - 2s 117us/step - loss: 0.6771 - acc: 0.6042 - val_loss: 0.6394 - val_acc: 0.6
904
Epoch 2/10
20000/20000 [==============================] - 2s 81us/step - loss: 0.5647 - acc: 0.7404 - val_loss: 0.5434 - val_acc: 0.72
42
Epoch 3/10
20000/20000 [==============================] - 2s 85us/step - loss: 0.4747 - acc: 0.7822 - val_loss: 0.5081 - val_acc: 0.74
30
Epoch 4/10
20000/20000 [==============================] - 2s 79us/step - loss: 0.4268 - acc: 0.8091 - val_loss: 0.4979 - val_acc: 0.74
90
Epoch 5/10
20000/20000 [==============================] - 2s 84us/step - loss: 0.3941 - acc: 0.8262 - val_loss: 0.4965 - val_acc: 0.75
34
Epoch 6/10
20000/20000 [==============================] - 2s 87us/step - loss: 0.3673 - acc: 0.8400 - val_loss: 0.4991 - val_acc: 0.75
60
Epoch 7/10
20000/20000 [==============================] - 2s 89us/step - loss: 0.3430 - acc: 0.8554 - val_loss: 0.5055 - val_acc: 0.75
28
Epoch 8/10
20000/20000 [==============================] - 2s 87us/step - loss: 0.3210 - acc: 0.8663 - val_loss: 0.5133 - val_acc: 0.74
98
Epoch 9/10
20000/20000 [==============================] - 2s 88us/step - loss: 0.3004 - acc: 0.8787 - val_loss: 0.5209 - val_acc: 0.74
82
Epoch 10/10
20000/20000 [==============================] - 2s 84us/step - loss: 0.2806 - acc: 0.8888 - val_loss: 0.5321 - val_acc: 0.74
60
```

Figura 17. Salida por consola. Entrenamiento del modelo de embeddings específicos para clasificación de críticas de IMDB.

Como podemos ver, obtenemos alrededor de un 75% de precisión en la validación, lo cual no está mal, pero es mejorable.

Para tratar de mejorar la precisión obtenida, probaremos a hacer uso de embeddings de palabras preentrenados. En concreto, haremos uso de los embeddings de ***GloVe***.

Cambiaremos de dataset, tomando ahora los datos directamente desde archivos con formato **.txt** que se encontrarán en dos carpetas que los clasifican como "positivas" si están en la carpeta ***pos*** o "negativas" si se encuentran en la carpeta ***neg***. Leeremos los datos y almacenaremos por una parte el texto de la crítica y por otra parte la valoración de la misma. 0 si la crítica es negativa y 1 si es positiva.

```
import os

imdb_dir = 'E:/Universidad/TFG/IMDB/aclImdb'
train_dir = os.path.join(imdb_dir, 'train')

labels = []
texts = []

for label_type in ['neg', 'pos']:
    dir_name = os.path.join(train_dir, label_type)
    for fname in os.listdir(dir_name):
        if fname[-4:] == '.txt':
            f = open(os.path.join(dir_name, fname), encoding="utf8")
            texts.append(f.read())
            f.close()
            if label_type == 'neg':
                labels.append(0)
            else:
                labels.append(1)
```

Código 4. Carga de datos de IMDB.

A continuación llevaremos a cabo el preprocesamiento de estos datos. Vectorizaremos el texto y se realizaremos una división del conjunto de datos de entrenamiento y de validación. Los embeddings de palabras preentrenados son especialmente efectivos en problemas con pocos datos, por eso, se restringirá el conjunto de entrenamiento a 200 muestras. [1]

También se tokenizarán las palabras, de manera que se asocie un número único a cada una de ellas.

```
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
import numpy as np

maxlen = 100
training_samples = 200
validation_samples = 10000
max_words = 10000

tokenizer = Tokenizer(num_words = max_words)
tokenizer.fit_on_texts(texts)
sequences = tokenizer.texts_to_sequences(texts)

word_index = tokenizer.word_index
print('Se han encontrado %s tokens únicos.' % len(word_index))

data = pad_sequences(sequences, maxlen=maxlen)

labels = np.asarray(labels)
print('Forma del tensor de datos: ', data.shape)
print('Forma del tensor de etiquetas: ', labels.shape)

indices = np.arange(data.shape[0])   #Estas líneas mezclan las críticas, dado que primero aparecen todas las
np.random.shuffle(indices)           #críticas negativas y luego todas las positivas
data = data[indices]
labels = labels[indices]

x_train = data[:training_samples]
y_train = labels[:training_samples]
x_val = data[training_samples: training_samples + validation_samples]
y_val = labels[training_samples: training_samples + validation_samples]
```

Código 5. Tokenización de las críticas de IMDB.

```
Se han encontrado 88582 tokens únicos.
Forma del tensor de datos:  (25000, 100)
Forma del tensor de etiquetas:  (25000,)
```

Figura 18. Salida por consola. Tokenización de las críticas de IMDB.

Los embeddings de palabras que usaremos son los de GloVe, en concreto es el archivo comprimido **glove.6B.zip**, el cual podemos encontrar aquí: *https://nlp.stanford.edu/projects/glove*

Una vez descomprimido, usaremos el archivo ***"glove.6B.100d.txt"***, que contiene embeddings de 100 dimensiones para 400000 palabras.

Si observamos los archivos **.txt** tras descomprimir, podemos ver que la estructura que siguen es la palabra al principio de la línea, seguida de todos los datos de sus dimensiones. Leeremos el archivo y lo estructuraremos en un diccionario que asocie la palabra con los coeficientes de sus 100 dimensiones.

```
glove_dir = 'E:/Universidad/TFG/IMDB/glove6B'

embeddings_index = {}
f = open(os.path.join(glove_dir, 'glove.6B.100d.txt'), encoding="utf8")

for line in f:
    values = line.split()
    word = values[0]
    coefs = np.asarray(values[1:], dtype='float32')
    embeddings_index[word] = coefs
f.close()

print('Se han encontrado %s vectores de palabras.' % len(embeddings_index))
```

Código 6. Carga de embeddings preentrenados de GloVe.

```
Se han encontrado 400000 vectores de palabras.
```

Figura 19. Salida por consola. Carga de embeddings preentrenados de GloVe.

Ahora hay que adaptar la información para que se presente en una matriz de la forma (*max_words*, *embedding_dim*), de manera que pueda ser cargada en una capa ***Embedding***. [1]

Además, es necesario asociar el índice de palabras que generamos previamente para la tokenización con el índice de embeddings.

```
embedding_dim = 100

embedding_matrix = np.zeros((max_words, embedding_dim))
for word, i in word_index.items():                  #Se comprueba cada palabra del índice de palabras de la tokenización
    if i < max_words:                               #Si existe en el índice de embeddings se toman los coeficientes y se
        embedding_vector = embeddings_index.get(word) #colocan en la matriz siguiendo el orden del índice de tokens
        if embedding_vector is not None:
            embedding_matrix[i] = embedding_vector #Las palabras que no estén en el embedding index serán todo ceros
```

Código 7. Adaptación de los embeddings preentrenados de GloVe.

Tras esto ya habremos terminado de preparar los datos para el entrenamiento y podremos configurar el modelo.

Haremos uso de un modelo similar al del primer experimento. Tendrá una capa ***Embedding*** a la que se le especifique el número de palabras encontradas, las dimensiones que tienen los embeddings y la longitud de las secuencias de palabras. Tras esta, colocaremos una capa ***Flatten*** para que la salida pueda ser tratada por una capa ***Dense*** de 32 neuronas con función de activación ***relu***. Esta función de activación es un cálculo simple que devuelve el valor que se provee como entrada directamente o 0 en caso de que la entrada sea 0 o menor que 0. Por último, haremos uso de una última capa ***Dense*** con 1 neurona y función de activación ***sigmoide***, de manera que el resultado sea 0 o 1.

Estableceremos que los pesos de la capa de ***Embedding*** están predefinidos y que no han de ser entrenados.

```
model = Sequential()
model.add(Embedding(max_words, embedding_dim, input_length=maxlen))
model.add(Flatten())
model.add(Dense(32, activation = 'relu'))
model.add(Dense(1, activation = 'sigmoid'))
model.summary()

model.layers[0].set_weights([embedding_matrix])
model.layers[0].trainable = False
```

Código 8. Configuración del modelo de embeddings preentrenados.

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_2 (Embedding)      (None, 100, 100)          1000000
_________________________________________________________________
flatten_2 (Flatten)          (None, 10000)             0
_________________________________________________________________
dense_2 (Dense)              (None, 32)                320032
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 33
=================================================================
Total params: 1,320,065
Trainable params: 1,320,065
Non-trainable params: 0
_________________________________________________________________
```

Figura 20. Salida por consola. Configuración del modelo de embeddings preentrenados.

Entrenaremos el modelo haciendo uso del optimizador ***RMSProp*** y estableceremos como función de pérdida ***binary_crossentropy***. También monitorizaremos la precisión que se obtenga conforme pasen las épocas.

Entrenaremos con los mismos parámetros que en el anterior experimento, durante 10 épocas y con lotes de 32 muestras. Sin embargo, esta vez no realizaremos la validación de manera estocástica, especificaremos cuáles son las muestras destinadas a la validación.

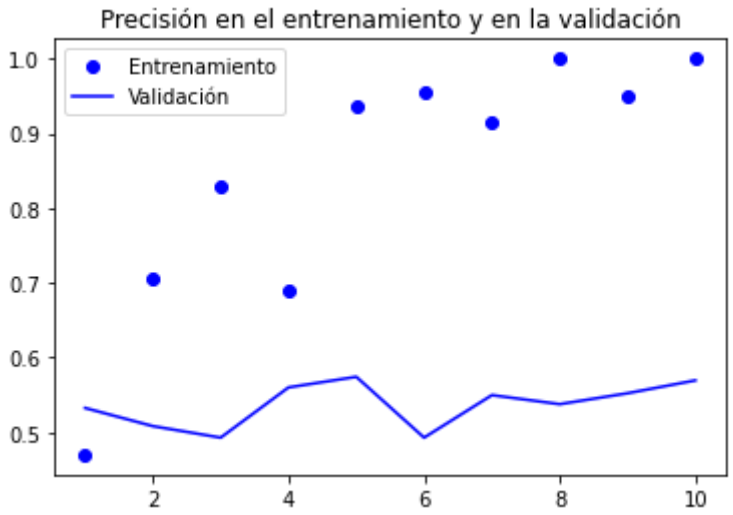
También almacenaremos los pesos del modelo en un archivo para el futuro.

```
model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['acc'])
history = model.fit(x_train, y_train, epochs = 10, batch_size = 32, validation_data = (x_val, y_val))
model.save_weights('pre_trained_glove_model.h5')
```

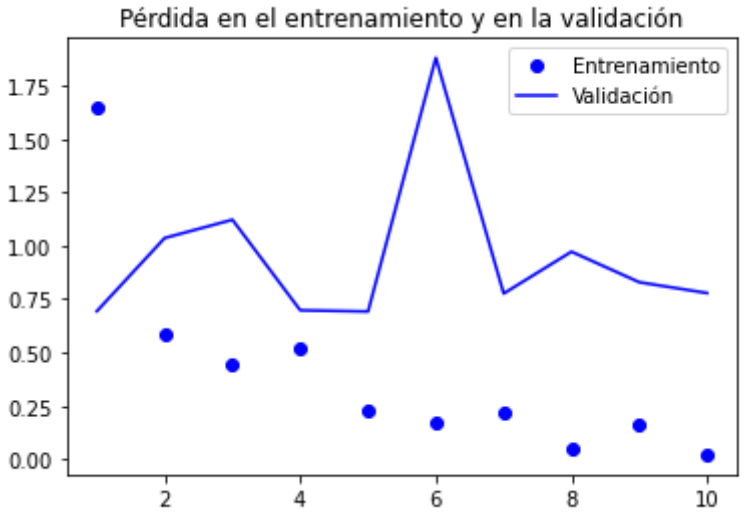
Mostraremos el resultado del entrenamiento en unas gráficas, las cuales serán dibujadas gracias al módulo ***pyplot*** de la librería ***matplotlib***.

```
import matplotlib.pyplot as plt

acc = history.history['acc']
val_acc = history.history['val_acc']
loss = history.history['loss']
val_loss = history.history['val_loss']

epochs = range(1, len(acc) + 1)

plt.plot(epochs, acc, 'bo', label = 'Entrenamiento')
plt.plot(epochs, val_acc, 'b', label = 'Validación')
plt.title('Precisión en el entrenamiento y en la validación')
plt.legend()

plt.figure()

plt.plot(epochs, loss, 'bo', label = 'Entrenamiento')
plt.plot(epochs, val_loss, 'b', label = 'Validación')
plt.title('Pérdida en el entrenamiento y en la validación')
plt.legend()

plt.show()
```

Código 9. Muestra por pantalla de gráficas de entrenamiento.



Cuadro 1. Precisión en el entrenamiento y en la validación del modelo de embeddings preentrenados.



Cuadro 2. Pérdida en el entrenamiento y en la validación del modelo de embeddings preentrenados.

Podemos comprobar que se está produciendo un gran sobreajuste, obteniendo en el entrenamiento una precisión del 100%, mientras que en la validación se obtiene alrededor del 55%. Esto es debido a que se están utilizando tan solo 200 muestras para el entrenamiento. [1]

Ahora probaremos a entrenar el modelo con embeddings específicos. Estos embeddings son mejores cuantos más datos haya, por lo que es de esperar que hacer uso de ellos con tan solo 200 muestras no vaya a dar buenos resultados. [1]

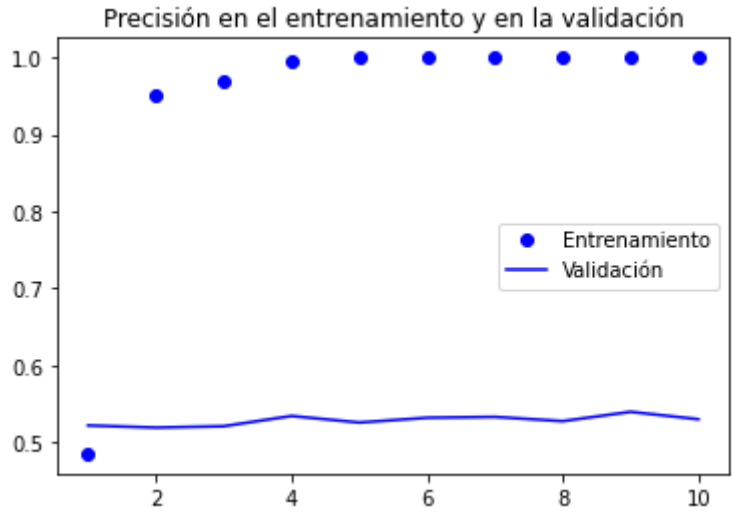
```
model = Sequential()
model.add(Embedding(max_words, embedding_dim, input_length=maxlen))
model.add(Flatten())
model.add(Dense(32, activation = 'relu'))
model.add(Dense(1, activation = 'sigmoid'))
model.summary()

model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['acc'])
history = model.fit(x_train, y_train, epochs = 10, batch_size = 32, validation_data = (x_val, y_val))
```

Código 10. Configuración de modelo de embeddings específicos para críticas de IMDB.

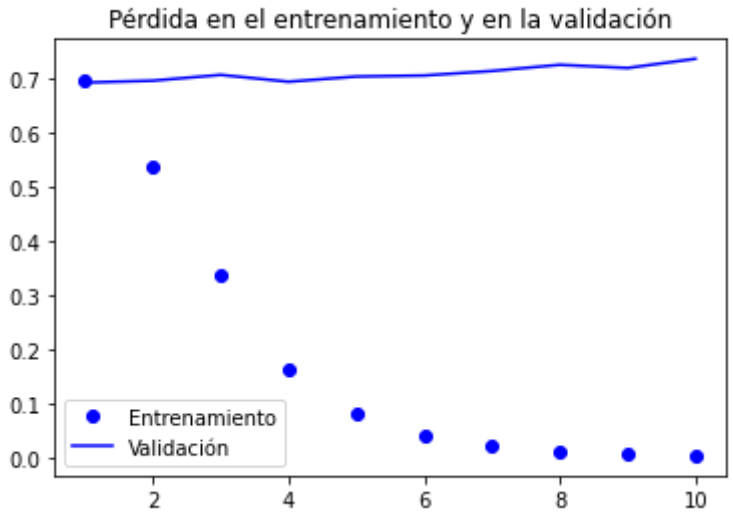
Mostramos los resultados de este entrenamiento.

```
import matplotlib.pyplot as plt

acc = history.history['acc']
val_acc = history.history['val_acc']
loss = history.history['loss']
val_loss = history.history['val_loss']

epochs = range(1, len(acc) + 1)

plt.plot(epochs, acc, 'bo', label = 'Entrenamiento')
plt.plot(epochs, val_acc, 'b', label = 'Validación')
plt.title('Precisión en el entrenamiento y en la validación')
plt.legend()

plt.figure()

plt.plot(epochs, loss, 'bo', label = 'Entrenamiento')
plt.plot(epochs, val_loss, 'b', label = 'Validación')
plt.title('Pérdida en el entrenamiento y en la validación')
plt.legend()

plt.show()
```

Código 11. Generación de gráficas de entrenamiento.



Cuadro 3. Precisión en el entrenamiento y en la validación del modelo de embeddings específicos.



Cuadro 4. Pérdida en el entrenamiento y en la validación del modelo de embeddings específicos.

Observamos un claro sobreajuste de nuevo en el entrenamiento. Probaremos a aumentar el número de muestras que usaremos para entrenar el modelo. Para ello, simplemente intercambiaremos los papeles de las muestras de entrenamiento y las de validación.

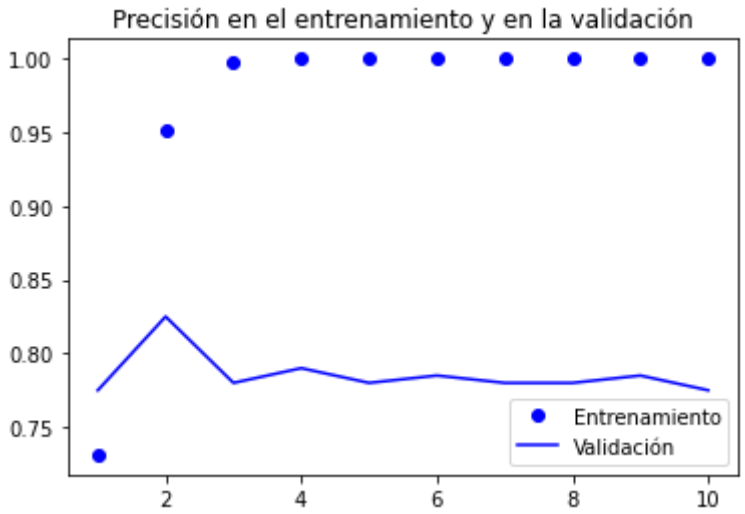
```
model = Sequential()
model.add(Embedding(max_words, embedding_dim, input_length=maxlen))
model.add(Flatten())
model.add(Dense(32, activation = 'relu'))
model.add(Dense(1, activation = 'sigmoid'))
model.summary()

model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['acc'])
history = model.fit(x_val, y_val, epochs = 10, batch_size = 32, validation_data = (x_train, y_train))
model.save_weights('embeddings_especificos_grande_model.h5')
```

Código 12. Preparación del modelo de embeddings específicos para mayor cantidad de muestras.

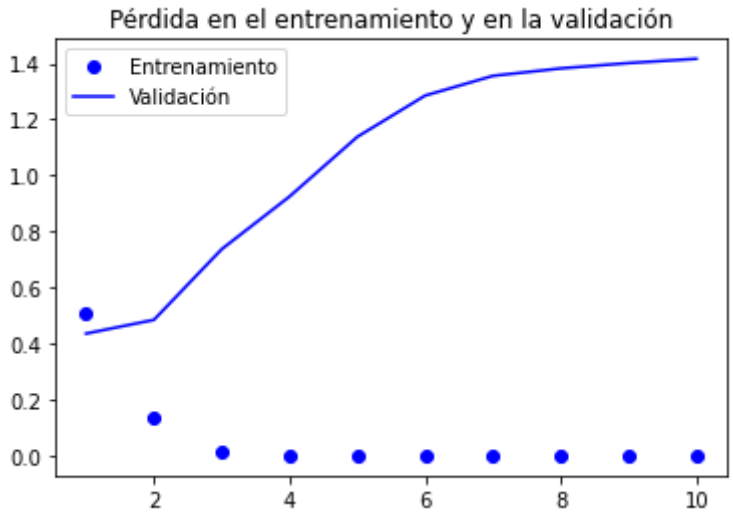
Mostraremos los resultados en gráficas de nuevo.

```
import matplotlib.pyplot as plt

acc = history.history['acc']
val_acc = history.history['val_acc']
loss = history.history['loss']
val_loss = history.history['val_loss']

epochs = range(1, len(acc) + 1)

plt.plot(epochs, acc, 'bo', label = 'Entrenamiento')
plt.plot(epochs, val_acc, 'b', label = 'Validación')
plt.title('Precisión en el entrenamiento y en la validación')
plt.legend()

plt.figure()

plt.plot(epochs, loss, 'bo', label = 'Entrenamiento')
plt.plot(epochs, val_loss, 'b', label = 'Validación')
plt.title('Pérdida en el entrenamiento y en la validación')
plt.legend()

plt.show()
```

Código 13. Preparación de gráficas de entrenamiento.



Cuadro 5. Precisión en el entrenamiento y en la validación del modelo de embeddings específicos con más muestras.



Cuadro 6. Pérdida en el entrenamiento y en la validación del modelo de embeddings específicos con más muestras

Como podemos ver, se produce también sobreajuste, pero la precisión en la validación llega a rondar el 80%, por lo que podemos comprobar que se obtienen mejores resultados cuantas más muestras se usan para embeddings específicos.

Para concluir este caso de estudio, evaluaremos el modelo de embeddings preentrenados de ***GloVe*** que entrenamos anteriormente. Para ello tomaremos datos de críticas que aún no han sido utilizados por ningún modelo y los prepararemos de la misma forma que hicimos anteriormente con el conjunto de datos de entrenamiento.

```
imdb_dir = 'E:/Universidad/TFG/IMDB/aclImdb'
test_dir = os.path.join(imdb_dir, 'test')

labels = []
texts = []

for label_type in ['neg', 'pos']:
    dir_name = os.path.join(test_dir, label_type)
    for fname in os.listdir(dir_name):
        if fname[-4:] == '.txt':
            f = open(os.path.join(dir_name, fname), encoding="utf8")
            texts.append(f.read())
            f.close()
            if label_type == 'neg':
                labels.append(0)
            else:
                labels.append(1)

sequences = tokenizer.texts_to_sequences(texts)
x_test = pad_sequences(sequences, maxlen = maxlen)
y_test = np.asarray(labels)
```

Código 14. Preparación de datos de evaluación de IMDB.

A continuación cargamos los pesos del modelo que almacenamos previamente en un fichero y lo evaluamos.

```
model.load_weights('pre_trained_glove_model.h5')
model.evaluate(x_test, y_test)
```

Código 15. Evaluación del modelo de embeddings preentrenados de GloVe.

```
25000/25000 [==============================] - 1s 45us/step

[0.7936463269805908, 0.564520001411438]
```

Figura 21. Salida por consola. Evaluación del modelo de embeddings preentrenados de GloVe.

Se consigue una precisión de alrededor del 56%, lo cual es un resultado bastante bajo, pero esperable. [1]

Ahora probaremos a evaluar el modelo de embeddings específico que fue entrenado con un conjunto grande de datos.

```
model.load_weights('embeddings_especificos_grande_model.h5')
model.evaluate(x_test, y_test)
```

Código 16. Evaluación del modelo de embeddings específicos con gran conjunto de muestras.

```
25000/25000 [==============================] - 1s 49us/step

[1.0465596101862191, 0.8226000070571899]
```

Figura 22. Salida por consola. Evaluación del modelo de embeddings específicos con gran conjunto de muestras.

Como podemos observar, este modelo obtiene un 82% de precisión en la evaluación, enfrentándose a datos que no ha visto nunca. Este resultado es bastante mejor que el anterior y nos sugiere que esta es una buena forma de tratar los problemas de análisis de sentimiento.

4.2 Generador de texto con LSTM

En este caso de estudio trataremos de replicar el ejemplo práctico de generación de textos basados en escritos de Nietzsche del capítulo 8 del libro *"Deep Learning con Python"*, de François Chollet.

Este experimento se basa en la generación de texto a nivel de carácter. Para este ejemplo haremos uso de uno de los textos que están incluidos en ***Keras***, en su módulo ***utils***. En concreto, será un texto de Nietzsche en inglés. La clave para la generación de textos es tener un texto o conjunto de ellos lo suficientemente grande como para que se pueda llegar a aprender algún modelo de estilo de redacción. [1]

```
import keras
import numpy as np

path = keras.utils.get_file('nietzsche.txt', origin='https://s3.amazonaws.com/text-datasets/nietzsche.txt')
text = open(path).read().lower()
print('Longitud del corpus:', len(text))
```

Código 17. Carga de textos de Nietzsche.

```
Downloading data from https://s3.amazonaws.com/text-datasets/nietzsche.txt
606208/600901 [==============================] - 1s 1us/step
Longitud del corpus: 600901
```

Figura 23. Salida por consola. Carga de textos de Nietzsche.

Ahora extraeremos secuencias que se solapen parcialmente. Estas secuencias tendrán longitud *maxlen* y serán tomadas cada *step* caracteres respecto al inicio de la anterior. Tras tomar las secuencias, les aplicaremos codificación **one-hot** y las colocaremos en una matriz 3D ***x*** con forma (*sequences*, *maxlen*, *unique_characters*). También prepararemos una matriz *y* que contenga los caracteres objetivo correspondientes. [1]

```
maxlen = 60
step = 3
sentences = []
next_chars = []

for i in range(0, len(text) - maxlen, step):
    sentences.append(text[i: i + maxlen])
    next_chars.append(text[i + maxlen])
print('Número de secuencias:', len(sentences))

#Se comprueban los caracteres únicos del texto.
chars = sorted(list(set(text)))
print('Caracteres únicos:', len(chars))
#Creamos un diccionario que mapee los caracteres únicos con su índice en 'chars'
char_indices = dict((char, chars.index(char)) for char in chars)

#Ahora realizamos codificación one-hot de los caracteres
print('Vectorización...')

x = np.zeros((len(sentences), maxlen, len(chars)), dtype=np.bool)
y = np.zeros((len(sentences), len(chars)), dtype=np.bool)

for i, sentence in enumerate(sentences):
    for t, char in enumerate(sentence):
        x[i, t, char_indices[char]] = 1
    y[i, char_indices[next_chars[i]]] = 1

print('\nEjemplo de muestra en X:\n')
print(x[0])
print('\nEjemplo de objetivo en Y:\n')
print(y[0])
```

Código 18. Preprocesamiento de muestras de textos de Nietzsche.

```
Número de secuencias: 200281
Caracteres únicos: 59
Vectorización...

Ejemplo de muestra en X:

[[False False False ... False False False]
 [False False False ... False False False]
 [False False False ... False False False]
 ...
 [False False False ... False False False]
 [False False False ... False False False]
 [False False False ... False False False]]

Ejemplo de objetivo en Y:

[False False False False False False False False False False False False
 False False False False False False False False False False False False
 False False False False False False False False False False False False
 False False False False False False False False  True False False False
 False False False False False False False False False False False]
```

Figura 24. Preprocesamiento de muestra de textos de Nietzsche.

Una vez hecho esto, ya tendremos el texto preparado para ser utilizado por la red neuronal. Comenzaremos configurando el modelo. Haremos uso de una capa ***LSTM*** seguida de un clasificador ***Dense*** con función ***softmax*** para todos los caracteres posibles.

```
from keras import layers

model = keras.models.Sequential()
model.add(layers.LSTM(128, input_shape=(maxlen, len(chars))))
model.add(layers.Dense(len(chars), activation='softmax'))
```

Código 19. Configuración del modelo generador de textos inspirados en Nietzsche.

Dado que los caracteres están codificados haciendo uso de **one-hot**, podemos utilizar **categorical_crossentropy** como función de pérdida. También haremos uso del optimizador ***RMSprop*** con un *learning rate* de 0.01.

```
optimizer = keras.optimizers.RMSprop(lr=0.01)
model.compile(loss='categorical_crossentropy', optimizer=optimizer)
```

Código 20. Monitorización del modelo generador de textos inspirados en Nietzsche.

A continuación definiremos la función de muestreo de caracteres. Esta función será usada para muestrear un carácter aleatorio teniendo en cuenta los pesos que devuelva la red neuronal en sus predicciones. También se tendrá en cuenta el valor de ***temperatura***, que determinará cuánta aleatoriedad se le aplicará a las predicciones del modelo. A más valor de ***temperatura***, más aleatorios serán los resultados. [1]

```
def sample(preds, temperature=1.0):
    preds = np.asarray(preds).astype('float64')
    preds = np.log(preds) / temperature
    exp_preds = np.exp(preds)
    preds = exp_preds / np.sum(exp_preds)
    probas = np.random.multinomial(1, preds, 1)
    return np.argmax(probas)
```

Código 21. Función "sample" para el generador de textos inspirados en Nietzsche.

Ahora vamos a entrenar el modelo. El siguiente bucle entrena y genera texto en cada época. En cada una de ellas generará texto con distintas temperaturas, de esta manera podremos comprobar la evolución del modelo y cómo afecta la temperatura en la generación de texto. [1]

```
import random
import sys

for epoch in range(1, 60):
    print('Época ', epoch)
    # Entrena al modelo para 1 época con los datos de entrenamiento
    model.fit(x, y,
              batch_size=128,
              epochs=1)

    # Selecciona una semilla de texto aleatoriamente
    start_index = random.randint(0, len(text) - maxlen - 1)
    generated_text = text[start_index: start_index + maxlen]
    print('--- Generando con la semill: "' + generated_text + '"')

    #Para cada época, genera texto con temperatura 0.2, 0.5, 1.0 y 1.2
    for temperature in [0.2, 0.5, 1.0, 1.2]:
        print('------ Temperatura:', temperature)
        sys.stdout.write(generated_text)

        # Generamos 400 caracteres cada vez
        for i in range(400):
            sampled = np.zeros((1, maxlen, len(chars)))
            for t, char in enumerate(generated_text):
                sampled[0, t, char_indices[char]] = 1.

            preds = model.predict(sampled, verbose=0)[0]
            next_index = sample(preds, temperature)
            next_char = chars[next_index]

            generated_text += next_char
            generated_text = generated_text[1:]

            sys.stdout.write(next_char)
            sys.stdout.flush()
        print()
```

Código 22. Entrenamiento del modelo generador de textos inspirados en Nietzsche.

A continuación veremos un ejemplo de los textos que se generaron durante el entrenamiento.

```
--- Generando con la semill: "l of women, in the marriage customs, in the
relations of old"
------ Temperatura: 0.2
l of women, in the marriage customs, in the
relations of old and the something the self-he still the considerate of the will the self-profers and with the considerate
of the consideration of the interpical and still the considerate of the considerate of the considerate of the considerate o
f the will the self-chand of the self the self-profers of the self-profers of the self-profering in the self-profertion of
the considerate of the self-man and the stringt
------ Temperatura: 0.5
rofertion of the considerate of the self-man and the stringtion of the fires to in its a profer it self-men and the worlds
he self har a that is the sufferent that the entitions that the desprofering and the standination of the man a thing and st
rongs and the man the herself believe and though of that our the interpian and them from the some the conseds the conscienc
e the conseas the chaistions
and in this he such is the dears in the contrance of the press
------ Temperatura: 1.0
d in this he such is the dears in the contrance of the pressing in the ansshange of the conspitice as huwad. he man probtio
n of
a then unself-men then whon onom besaritys in whine lely interpiness it as brock who
he eashers me
pinistly: huther howe rateral wife impustion is firghing in elsone they present is eligan; of there
howrubed, as the same a tho instinct, heres so
deas dopeningo beations of thener
to
a
presely in pation and natate dangerics of who
------ Temperatura: 1.2
f thener
to
a
presely in pation and natate dangerics of who sfob ; song
besire, becom and
was his maniffentkingr centermansery and deld, a callod fteei, cise him. headionnes wo ngtents, .

": somlakl
mis,ror, curtude, fro"suf
theseliagiently
is
unafoment, but"s there
forming they untan is adiery, they dorphe, if a preguring there, of
free re. he cons
ort howher, acone. the alselides have to dore ; besesse men a sho thach--assquatory atshables by thesy fr
```

Figura 25. Salida por consola. Textos generados inspirados en Nietzsche.

Como podemos ver, un valor bajo de temperatura resulta en un texto extremadamente predecible y repetitivo, pero todas las palabras son reales. Conforme subimos la temperatura, el texto se vuelve más creativo, creando a veces palabras totalmente inventadas pero que suenan verosímiles como "dorphe" o "huther".

Con temperaturas altas el texto se descompone y pierde el sentido casi por completo, creando cadenas semialeatorias de caracteres.

Sin duda, es importante escoger un buen valor de temperatura para conseguir tener cierta aleatoriedad sin que se pierda el sentido del texto. Para este ejemplo en concreto, parece que 0.5 podría ser una buena temperatura. [1]

Capítulo 5. GENERACIÓN DE HAIKUS

5.1 Definición del problema

El problema a resolver es la generación de haikus haciendo uso de un modelo de red neuronal generativo. Se trata de hacer uso de un modelo de red neuronal recurrente, útiles para aprendizaje profundo de secuencias y de textos, para conseguir que aprenda patrones textuales que caracterizan a un haiku.

Una vez sea entrenada la red, se tratarán de generar haikus para comprobar si realmente pueden haberse aprendido estructuras lingüísticas que conformen uno de estos poemas coherentemente.

Para ello, será necesario tener un dataset de haikus lo suficientemente grande y con una estructura uniforme, de manera que facilite el aprendizaje de patrones al modelo.

5.2 Medida de éxito

Nos encontramos ante un problema bastante abstracto, cuya medida de éxito resulta compleja de definir.

Para concretar una medida de éxito, mediante la cual podremos discernir el correcto aprendizaje y utilidad del generador, tendremos en cuenta la estructura del haiku generado, es decir, el número de sílabas que tenga cada verso; así como el porcentaje de aciertos (*accuracy*) que se obtenga en el entrenamiento.

Dado que los modelos desarrollados generarán poemas, también trataremos de generar haikus y comprobar qué estructuras mantienen, si es que mantienen alguna.

5.3 Protocolo de evaluación

Una vez sabemos todos los detalles del problema en profundidad, sabemos las dificultades que podremos encontrarnos. Para poder obtener datos de cómo está yendo el entrenamiento en cada momento, reservaremos en cada época un conjunto de muestras para realizar la validación.

Gracias a este método de validación sabremos al final de cada época la precisión en la predicción que se está consiguiendo en cada una de ellas. Esto nos facilitará la modificación de parámetros en el modelo para mejorar su rendimiento.

5.4 Organización del dataset

El dataset que será utilizado está formado por 144123 poemas en formato CSV. Entre ellos, 92536 son haikus que siguen el esquema 5-7-5. El resto del dataset ha sido completado con otros poemas de lengua inglesa que tienen distintos esquemas y rimas. Esto resulta problemático, ya que se corre el riesgo de hacer uso de un dataset con aproximadamente un 36% de su contenido con información errónea para la generación de haikus.

Además de lo anterior, los poemas poseen en algunos casos símbolos y palabras escritas en mayúscula.

Una vez explicada la estructura del dataset con el que se va a trabajar, comenzamos con el primer generador que realizaremos.

5.5 Primer generador

5.5.1 Definición

Este es el primer acercamiento que se realizará para tratar de solventar el problema. La idea principal es crear un primer generador de texto a nivel de carácter. Para ello se tratará de seguir el caso de estudio de generación de textos, a pesar de que este sea específico para textos en prosa.

5.5.2 Preprocesamiento del conjunto de datos

Para comenzar el experimento, es necesario tratar la información que será utilizada para el entrenamiento. Tomaremos los haikus del dataset y leeremos cada verso, uniéndolos entre sí separados con ***\n***.

Importamos las librerías necesarias para el primer generador.

```
import keras
from keras.preprocessing.sequence import pad_sequences
from keras.callbacks import LambdaCallback
from keras.models import Sequential
from keras.layers import Dense, Activation, LSTM
from keras.optimizers import RMSprop
import pandas as pd
import csv
import numpy as np
import string as string
import random
import sys
```

Código 23. Librerías importadas para el primer generador

Ahora, configuramos los parámetros más importantes para la configuración del Notebook.

En primer lugar, podemos modificar ***maxlen*** para cambiar el número de caracteres que tendrán las secuencias.

También podemos cambiar el número de caracteres que se saltan para empezar a tomar otra secuencia cambiando el valor de ***step***.

Por último, podemos establecer la ruta al dataset en nuestro equipo modificando ***haiku_path***.

```
# Configuración
#Número de caracteres de las secuencias
maxlen = 20
#Número de caracteres que se saltarán para empezar tomar otra secuencia
step = 5
#Ruta al archivo donde están almacenados los haikus
haiku_path = 'all_haiku.csv'
```

Código 24. Bloque de configuración del primer generador

A continuación, comenzaremos el preprocesamiento cargando los datos desde el fichero especificado en ***haiku_path***. Haciendo uso de la librería ***pandas*** leeremos las tres columnas que representan cada uno de los versos de los poemas, los uniremos en una sola cadena añadiendo ***\n*** al final de cada verso y, una vez hecho esto, limpiaremos el texto, transformando los

caracteres en minúscula y eliminando símbolos no necesarios. También, nos aseguraremos de que el texto esté codificado en **UTF-8**.

También se eliminarán espacios repetidos e irrelevantes que no aporten nada al poema. No hacer esto supondría que el modelo pudiera aprender que después de un espacio hay posibilidad de que venga uno o más espacios, lo que crearía versos vacíos en algunos casos.

```
sentences = []
next_chars = []

df = pd.read_csv(str(haiku_path), usecols=["0","1","2"])
texts = df[['0', '1', '2']].values
inputs = list()

#Bucle para unir los versos, limpiarlos de símbolos de puntuación y ponerlos en minúscula
for i in texts:
    aux = str(i[0]) + "\n" + str(i[1]) + "\n" + str(i[2])
    aux = "".join(v for v in aux if v not in string.punctuation and v not in string.digits).lower()
    aux = aux.encode("utf8").decode("ascii",'ignore')
    prechar = " "
    char_list = list(aux)
    #Eliminación de espacios repetidos
    for char in range(0,len(char_list)):
        if (prechar is " " or prechar is "\n") and char_list[char] is " ":
            char_list[char-1]=""
        prechar = char_list[char]
    aux = "".join(char_list)
    inputs.append(aux)
```

Código 25. Carga y limpieza de datos para el primer generador.

Una vez hecho esto, será necesario generar secuencias de caracteres de la misma longitud. El tamaño de estas cadenas será ***maxlen*** y se variará el índice del primer carácter de la secuencia tantas posiciones como dicte ***step***. También se tomarán los siguientes caracteres que aparecerán tras estas secuencias

```
#Generación de subsecuencias
for t in inputs:
    for i in range(0, len(t) - maxlen, step):
        sentences.append(t[i: i + maxlen])
        next_chars.append(t[i + maxlen])

print('Número de secuencias:', len(sentences))
print('Primeras dos secuencias tomadas:',sentences[:2])
print('Primeros caracteres objetivos tomados:',next_chars[:2])
```

Código 26. Generación de secuencias y caracteres objetivo del primer generador.

Estas son las dos primeras secuencias tomadas, junto con sus próximos caracteres:

```
Número de secuencias: 1384333
Primeras dos secuencias tomadas: ['fishing boats\ncolors', 'ng boats\ncolors of\nt']
Primeros caracteres objetivos tomados: [' ', 'h']
```

Figura 26. Salida por consola. Primeras secuencias y caracteres objetivos del primer generador.

Ahora que tenemos las secuencias junto con sus próximos caracteres, será necesario realizar codificación ***one-hot*** para introducir estos datos al modelo. Es por eso que tendremos que agrupar todos los caracteres únicos que existen en estas secuencias.

```
chars = []
for i in inputs:
    chars_in_text = sorted(list(set(i)))
    for c in chars_in_text:
        if c not in chars:
            chars.append(c)
chars.sort()
print('Caracteres encontrados en el conjunto de datos:', len(chars), '\n',chars)
```

Código 27. Agrupación de caracteres únicos para el primer generador.

Podemos ver el número de caracteres únicos que se han encontrado, listándolos.

```
Caracteres encontrados en el conjunto de datos: 28
 ['\n', ' ', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v',
'w', 'x', 'y', 'z']
```

Figura 27. Salida por consola. Caracteres únicos del primer generador.

Ahora aplicaremos codificación ***one-hot*** a las secuencias y a los caracteres objetivo.

```
char_indices = dict((c, i) for i, c in enumerate(chars))
indices_char = dict((i, c) for i, c in enumerate(chars))

x = np.zeros((len(sentences), maxlen, len(chars)), dtype=np.bool)
y = np.zeros((len(sentences), len(chars)), dtype=np.bool)
for i, sentence in enumerate(sentences):
    for t, char in enumerate(sentence):
        x[i, t, char_indices[char]] = 1
    y[i, char_indices[next_chars[i]]] = 1

print('Primera secuencia con codificación one-hot:\n')
print(x[0])
print('\nPrimer caracter objetivo con codificación one-hot:\n')
print(y[0])
```

Código 28. Codificación one-hot para el primer generador.

Ahora tendremos una matriz de la forma (*número de secuencias, maxlen, número de caracteres únicos*), el cual representa cada una de las secuencias codificadas.

También tendremos una matriz de la forma (*número de secuencias, número de caracteres únicos*), que corresponde a los caracteres objetivo codificados.

Para mostrar el resultado de la codificación, solo mostraremos el primero de los caracteres objetivo. El resto de caracteres y secuencias estarán en el mismo formato.

```
Primer caracter objetivo con codificación one-hot:

[False  True False False False False False False False False False False
 False False False False False False False False False False False False
 False False False False]
```

Figura 28. Primer carácter objetivo con codificación one-hot del primer generador.

Por último, separaremos un pequeño conjunto de estos datos, un 10%, y los reservaremos para utilizarlos en la evaluación del modelo. Estas muestras no serán utilizadas ni en el entrenamiento ni en la validación.

```
proporcion_evaluacion = 0.1
test_x = x[:int(len(x)*proporcion_evaluacion)]
x = x[int(len(x)*proporcion_evaluacion)+1:]
test_y = y[:int(len(y)*proporcion_evaluacion)]
y = y[int(len(y)*proporcion_evaluacion)+1:]
```

Código 29. Reserva de muestras para evaluación del primer generador.

Tras esto, ya tenemos preparados los datos para ser utilizados por el modelo, por lo que podemos pasar a configurarlo.

5.5.3 Configuración del modelo de red neuronal recurrente 1

El modelo de red neuronal de este primer generador será definido haciendo uso de la clase ***Sequential()*** de **keras**. Esta clase permite apilar capas de manera sencilla haciendo uso del método ***add()***.

Para este primer modelo utilizaremos una capa ***LSTM*** con 128 neuronas. Le especificaremos que la entrada será de la forma (*maxlen, número de caracteres únicos*). Sobre esta primera capa añadiremos una capa ***Dense*** con tantas neuronas como caracteres únicos hayan. Por último, agregaremos una capa ***Activation*** con función **Softmax**, de manera que los valores de salida sean una distribución de probabilidad que determinen cuánto de probable es que el carácter en un índice determinado sea el siguiente dada una secuencia. En el entrenamiento el modelo siempre tomará el carácter más probable como el predicho para calcular la precisión.

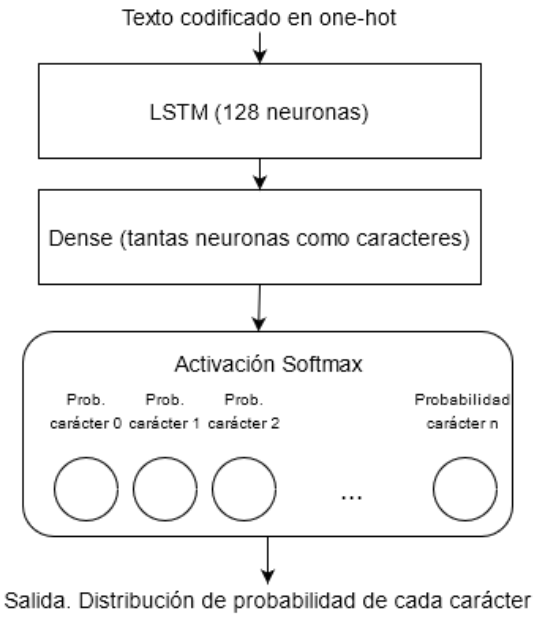


Figura 29. Arquitectura del primer modelo.

```
model = Sequential()
model.add(LSTM(128, input_shape=(maxlen, len(chars))))
model.add(Dense(len(chars)))
model.add(Activation('softmax'))
model.summary()
```

Código 30. Configuración del modelo del primer generador

```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_1 (LSTM)                (None, 128)               80384
_________________________________________________________________
dense_1 (Dense)              (None, 28)                3612
_________________________________________________________________
activation_1 (Activation)    (None, 28)                0
=================================================================
Total params: 83,996
Trainable params: 83,996
Non-trainable params: 0
_________________________________________________________________
```

Figura 30. Salida por consola. Resumen del modelo del primer generador

Ahora estableceremos algunas funciones para monitorizar y controlar el proceso que está siguiendo la red neuronal para entrenarse.

5.5.4 Monitorización del proceso de aprendizaje

Haremos uso del optimizador ***RMSprop*** con un ***learning rate*** de 0.01. Este se encargará de actualizar los pesos de la red neuronal, tratando de encontrar un mínimo global en la función de pérdida, es decir, tratando de conseguir la mejor configuración posible para el modelo establecido.

Tomaremos como función de pérdida ***categorical crossentropy***. Establecemos esto así puesto que el modelo devolverá un conjunto de varias categorías (los caracteres) en el que escoger una categoría excluye al resto de ser escogidas.

Como métrica del éxito que está teniendo el entrenamiento utilizaremos la ***categorical accuracy***, que nos dará información de la cantidad de aciertos de próximos caracteres que están habiendo al final de las épocas. De esta manera podremos intuir cómo está yendo el entrenamiento según se realiza.

```
optimizer = RMSprop(lr=0.01)
model.compile(loss='categorical_crossentropy', optimizer=optimizer, metrics=['categorical_accuracy'])
```

Código 31. Monitorización del proceso de aprendizaje del primer generador.

Además de lo anterior, que medirá cuantitativamente el éxito del progreso, añadiremos unas funciones para generar texto tras cada época con distintas temperaturas. De esta manera podremos ver cómo evoluciona el texto generado conforme avanza el entrenamiento.

En primer lugar definimos una función ***sample()***, que tomará las probabilidades de cada carácter, aplicará un valor ***temperature*** y devolverá el índice de un carácter.

```
def sample(preds, temperature=1.0):
    # Función auxiliar para tomar un índice partiendo de un array de probabilidades
    preds = np.asarray(preds).astype('float64')
    preds = np.log(preds) / temperature
    exp_preds = np.exp(preds)
    preds = exp_preds / np.sum(exp_preds)
    probas = np.random.multinomial(1, preds, 1)
    return np.argmax(probas)
```

Código 32. Función sample del primer generador.

Una vez hecho esto, definiremos la función ***on_epoch_end*** que se ejecute cada vez que termine una época e imprima por pantalla 200 caracteres para los valores de temperatura 0.2, 0.5, 1.0 y 1.2.

```
def on_epoch_end(epoch, logs):
    # Función invocada al final de cada época. Imprime por pantalla el texto generado.
    print()
    print('----- Generando texto tras la época: %d' % epoch)

    haiku_index = random.randint(0, len(inputs))
    haiku = inputs[haiku_index]
    start_index = random.randint(0, len(haiku) - maxlen - 1)
    for temperature in [0.2, 0.5, 1.0, 1.2]:
        print('----- Temperatura:', temperature)

        generated = ''
        sentence = haiku[start_index: start_index + maxlen]
        generated += sentence
        num_versos=0
        print('----- Generando con la semilla: "' + sentence + '"')
        sys.stdout.write(generated)
        for i in range(200):
            x_pred = np.zeros((1, maxlen, len(chars)))
            for t, char in enumerate(sentence):
                x_pred[0, t, char_indices[char]] = 1.

            preds = model.predict(x_pred, verbose=0)[0]
            next_index = sample(preds, temperature)
            next_char = indices_char[next_index]

            generated += next_char
            sentence = sentence[1:] + next_char

            sys.stdout.write(next_char)
            sys.stdout.flush()

        print()
print_callback = LambdaCallback(on_epoch_end=on_epoch_end)
```

Código 33. Función on_epoch_end del primer generador.

Ahora crearemos un ***ModelCheckpoint*** que almacene los valores de los pesos que más han reducido el valor de la función de pérdida al final de cada época.

```
from keras.callbacks import ModelCheckpoint

filepath = "weights.hdf5"
checkpoint = ModelCheckpoint(filepath, monitor='loss',
                             verbose=1, save_best_only=True,
                             mode='min')
```

Código 34. ModelCheckpoint del primer generador.

También añadiremos un ***Callback*** más, ***ReduceLROnPlateau***, que reducirá el *learning rate* conforme los valores de la función de pérdida dejen de mejorar. De esta manera los valores acabarán estabilizándose y la función de pérdida alcanzará un mínimo.

```
from keras.callbacks import ReduceLROnPlateau
reduce_lr = ReduceLROnPlateau(monitor='loss', factor=0.2,
                              patience=1, min_lr=0.001)
```

Código 35. ReduceLROnPlateau para el primer generador.

Añadiremos todas las funciones anteriores a un array de ***Callbacks*** que el modelo utilizará durante el entrenamiento.

```
callbacks = [print_callback, checkpoint, reduce_lr]
```

Código 36. Array 'callbacks' para el primer generador.

5.5.5 Entrenamiento del modelo definido

A continuación llamaremos a la función ***Fit()*** del modelo para entrenarlo. Le pasaremos como parámetros las secuencias de caracteres y los caracteres objetivo en codificación one-hot. Estableceremos el tamaño de los **lotes en 128 muestras**. Entrenaremos en **25 épocas** y realizaremos **validación** reservando el 20% de las muestras en cada época para ello. También introduciremos los ***Callbacks*** que definimos anteriormente.

```
history = model.fit(x, y, batch_size=128, epochs=25, validation_split = 0.2, callbacks=callbacks)
model.save_weights('generador_haikus1.h5')
```

Código 37. Entrenamiento del primer generador.

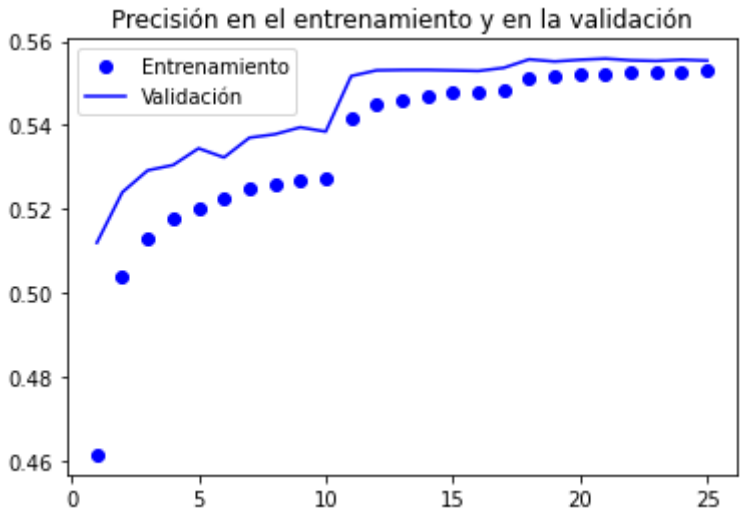
Representaremos los resultados del entrenamiento en unas gráficas. Para ello utilizaremos el módulo ***pyplot*** de la librería ***matplotlib*** y recogeremos los datos que fueron almacenados en ***history*** durante el entrenamiento.

La primera mostrará la precisión en la predicción del próximo carácter que ha tenido el modelo en el entrenamiento y en la validación de cada época.

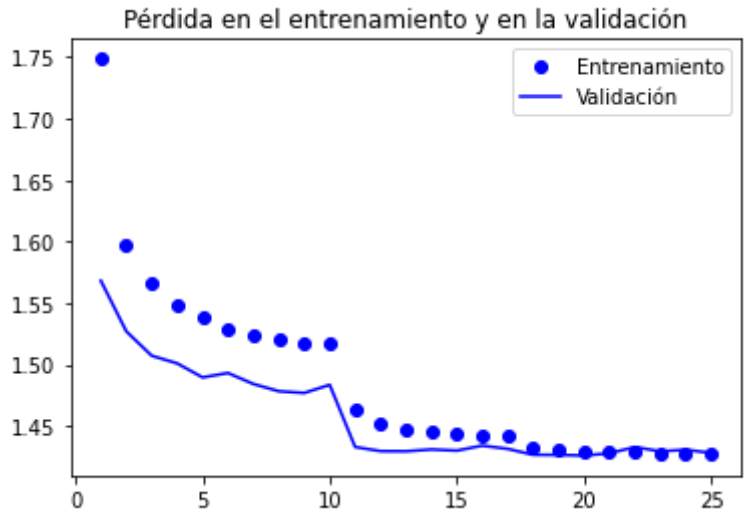
La segunda mostrará los valores de pérdida en el entrenamiento y en la validación conforme han avanzado las épocas.

```
import matplotlib.pyplot as plt

acc = history.history['categorical_accuracy']
val_acc = history.history['val_categorical_accuracy']
loss = history.history['loss']
val_loss = history.history['val_loss']

epochs = range(1, len(acc) + 1)

plt.plot(epochs, acc, 'bo', label = 'Entrenamiento')
plt.plot(epochs, val_acc, 'b', label = 'Validación')
plt.title('Precisión en el entrenamiento y en la validación')
plt.legend()

plt.figure()

plt.plot(epochs, loss, 'bo', label = 'Entrenamiento')
plt.plot(epochs, val_loss, 'b', label = 'Validación')
plt.title('Pérdida en el entrenamiento y en la validación')
plt.legend()

plt.show()
```

Código 38. Generación de gráficas de entrenamiento del primer generador.



Cuadro 7. Precisión en el entrenamiento y en la validación del primer generador.



Cuadro 8. Pérdida en el entrenamiento y en la validación del primer generador.

Como podemos observar en las gráficas, a partir de la época 11 el modelo alcanza un mínimo en la función de pérdida. El valor oscila levemente, pero se mantiene estable a partir de ahí, dando alrededor del 55% de precisión en el entrenamiento y algo menos, un 54%, de precisión en la validación. A continuación comprobaremos si la precisión y la pérdida se mantienen con valores que el modelo no ha visto nunca.

5.5.6 Evaluación del primer generador

Tras completar el entrenamiento, nos aseguraremos de que los resultados obtenidos estén representando el mínimo global de la función de pérdida, lo que significaría que el modelo ha alcanzado su máxima refinación posible.

Para ello utilizaremos los datos que separamos previamente en el preprocesamiento, reservándolos para utilizarlos en este momento. El modelo no ha tratado en ningún momento con estos datos, así que para él son totalmente nuevos.

```
results = model.evaluate(test_x, test_y)
print("Pérdida en la evaluación:", results[0])
print("Precisión en la evaluación:", results[1])
```

Código 39. Evaluación del primer generador.

```
138433/138433 [==============================] - 41s 295us/step
Pérdida en la evaluación: 1.6826399037634634
Precisión en la evaluación: 0.489977091550827
```

Figura 31. Salida por consola. Resultados de la evaluación del primer generador.

Como podemos observar, la precisión en la evaluación es algo inferior a la obtenida en el entrenamiento, lo cual sugiere que hemos conseguido ajustar los pesos de la red al máximo para

esta configuración y, además, nos deja ver que el modelo se ha sobreajustado a los datos de entrenamiento.

Ahora pasaremos a ver algunos textos que se han generado con este modelo.

En primer lugar, mostraremos un generador que escribe caracteres hasta que el texto contenga tres ***\n***. De esta manera se trata de simular que se escriban tres versos. El único problema es que en ocasiones el modelo no escribe estos caracteres nunca, por lo que este generador podría escribir indefinidamente.

Utilizaremos una frase predefinida de longitud ***maxlen***. También usaremos la función ***sample()*** que definimos anteriormente para tomar el próximo carácter. Utilizaremos una temperatura de 0.7.

```
temperatura = 0.7
#Frase que el modelo necesita para comenzar a generar, tiene que ser de longitud maxlen
sentence = "the eagle flies and\n"

generated = ''
num_versos = 0
model.load_weights('generador_haikus1.h5')

print('----- Generando con la semilla: "' + sentence + '"')
sys.stdout.write(generated)

while num_versos < 3:
    x_pred = np.zeros((1, maxlen, len(chars)))
    for t, char in enumerate(sentence):
        x_pred[0, t, char_indices[char]] = 1.

    preds = model.predict(x_pred, verbose=0)[0]
    next_index = sample(preds, temperatura)
    next_char = indices_char[next_index]
    if next_char == "\n":
        num_versos += 1
    generated += next_char
    sentence = sentence[1:] + next_char

    sys.stdout.write(next_char)
    sys.stdout.flush()
```

Código 40. Generador hasta tres \n. Primer generador.

```
----- Generando con la semilla: "the eagle flies and
"
saturday close to
the last fairages
with my dog year that was like strength
```

Figura 32. Salida por consola. Poema del generador de tres \n. Primer generador.

Como podemos ver, se generan palabras de la lengua inglesa correctamente, incluso parece que se han aprendido algunas estructuras sintácticas correctamente, pero el texto generado tiene poco o ningún sentido. Además, podemos ver que no está presente la estructura de un haiku.

En segundo lugar, probaremos a generar un número fijo de caracteres. En concreto, generaremos 200 caracteres. Utilizaremos una temperatura de 0.5.

```
temperatura = 0.5
sentence = "the eagle flies and\n"
print('----- Generando con la semilla: "' + sentence + '"')

for i in range(200):
            x_pred = np.zeros((1, maxlen, len(chars)))
            for t, char in enumerate(sentence):
                x_pred[0, t, char_indices[char]] = 1.

            preds = model.predict(x_pred, verbose=0)[0]
            next_index = sample(preds, temperatura)
            next_char = indices_char[next_index]

            generated += next_char
            sentence = sentence[1:] + next_char

            sys.stdout.write(next_char)
            sys.stdout.flush()
```

Código 41. Generador de 200 caracteres. Primer generador.

```
----- Generando con la semilla: "the eagle flies and
"
i wanna get the content of the same is the book for the and one
in my fun shadows soon is the parting the best real for passion in the only stars the same is the million with the story to
someone
i ne
```

Figura 33. Salida por consola. Generador de 200 caracteres. Primer generador.

Comprobando la salida que nos devuelve este generador, podemos ver que esta no es una buena forma de intentar generar poemas. En este caso, vemos que todas las palabras son reales, a excepción de la última, que fue cortada por el límite de caracteres. También podemos observar que se intenta mantener coherencia sintáctica entre las palabras, pero conforme se alarga la frase, el sentido de la misma empieza a desvanecerse.

5.5.7 Discusión del primer generador

A continuación explicaremos a grandes rasgos los distintos problemas y lecciones que se han extraído de esta primera versión.

En primer lugar, el método usado para la creación de secuencias de caracteres dejaba sin utilizar algunos caracteres al final de los poemas. Esto podría solucionarse utilizando técnicas de ***padding***, es decir, rellenando las secuencias que no lleguen a tener ***maxlen*** con caracteres superfluos para que pudieran ser utilizadas por la red.

También, aprendimos que el dataset requería de un mayor filtro, dado que no solo incluye haikus de estructura 5-7-5. Ha sido completado con algunos poemas ingleses con distinta estructura.

Por último, se comprobó que el modelo planteado no era el más adecuado para la generación de versos. En muchos casos se generaban textos sin saltos de línea. Además, no parecía que en ningún momento se hubiese aprendido que un haiku necesita tener estructura 5-7-5.

5.6 Segundo generador

5.6.1 Definición

En este segundo acercamiento trataremos de llevar a cabo un generador a nivel de palabra para solventar el problema. Para ello, se tratarán de utilizar los conocimientos adquiridos hasta el momento para completarlo.

Se hará uso de embeddings de palabras específicos, trataremos que se aprendan correctamente relaciones semánticas entre las palabras que conforman el corpus.

5.6.2 Preprocesamiento del conjunto de datos

Para empezar este experimento, necesitaremos preprocesar los datos de manera que en lugar de hacerlo a nivel de carácter, sea a nivel de palabra. Tomaremos todos los poemas que tenemos y uniremos sus versos. Por el momento, como estamos haciendo una prueba para tratar de hacer que aprenda patrones de lenguaje, no nos molestaremos en separar los versos con \n.

A continuación mostramos las librerías que serán utilizadas.

```
import keras
from keras.preprocessing.sequence import pad_sequences
from keras.preprocessing.text import Tokenizer, one_hot
from keras.callbacks import LambdaCallback
from keras.models import Sequential
from keras.layers import Dense, Activation, LSTM, Embedding
import pandas as pd
import csv
import numpy as np
import string as string
import random
import sys
```

Código 42. Librerías importadas para el segundo generador.

También, configuraremos los parámetros más importantes para el preprocesamiento. Como en el anterior experimento, éstos son la ruta al fichero con los haikus, la longitud de las secuencias y, por último, el número de palabras que se saltarán para tomar la siguiente secuencia.

```
haiku_path = "all_haiku.csv"
sequence_length = 4
sequence_step = 1
```

Código 43. Bloque de configuración del segundo generador.

Ahora leeremos el archivo con los haikus haciendo uso de la librería ***pandas***. Limpiaremos el texto, haciendo que esté en minúsculas y borrando dígitos y signos de puntuación. También eliminaremos espacios repetidos que puedan quedar. Uniremos los tres versos en una sola cadena.

```
df = pd.read_csv(str(haiku_path), usecols=["0","1","2"])
texts = df[['0', '1', '2']].values
corpus = []
for i in texts:
    aux = str(i[0]) + " " + str(i[1]) + " " + str(i[2])
    aux = "".join(v for v in aux if v not in string.punctuation and v not in string.digits).lower()
    aux = aux.encode("utf8").decode("ascii",'ignore')
    prechar = " "
    char_list = list(aux)
    #Eliminación de espacios repetidos
    for char in range(0,len(char_list)):
        if (prechar is " " or prechar is "\n") and char_list[char] is " ":
            char_list[char-1]=""
        prechar = char_list[char]
    aux = "".join(char_list)
    corpus.append(aux)
corpus[0:10]
```

Código 44. Carga y limpieza de los poemas para el segundo generador.

```
['fishing boats colors of the rainbow',
 'ash wednesday trying to remember my dream',
 'snowy morn pouring another cup of black coffee',
 'shortest day flames dance in the oven',
 'haze half the horse hidden behind the house',
 'low sun the lady in red on high heels',
 'advent the passing stranger farts',
 'tarn a bubble in the ice',
 'snowflakes new asphalt in the holes',
 'crystal night gusts of rain outside']
```

Figura 34. Salida por consola. Primeros poemas para el segundo generador.

Ahora, siguiendo el flujo de trabajo del anterior generador, recopilaremos todas las palabras únicas que existen en el texto y generaremos varias secuencias de palabras, almacenando la próxima palabra que le seguirá.

Además de lo anterior, codificaremos las palabras para que puedan ser utilizadas por la red neuronal. En esta ocasión, dado el gran número de palabras que hay, resulta imposible usar codificación one-hot, por lo que esta vez tokenizaremos las palabras transformándolas en enteros únicos que representen su posición en una lista. Para ello, haremos uso de la función ***one_hot()*** de ***Keras***, que permite introducir una longitud del vocabulario para asignar a cada palabra un número entero único.

```
word_list = []
for s in corpus:
    aux = s.split()
    word_list.extend(aux)
unique_words = sorted((set(word_list)))
print("Número de palabras únicas: " + str(len(unique_words)))

sequences, next_words = [], []
corpus = word_list
idx = 0
while idx + sequence_length + 1 < len(corpus):
    words = corpus[idx:idx+sequence_length]
    sequence=""
    for w in words:
        sequence += " " + w
    sequence = sequence.strip()
    nextword = corpus[idx+sequence_length]
    sequences.append(sequence)
    next_words.append(nextword)
    idx += sequence_step

embedded_sentences = [one_hot(sent, len(unique_words)) for sent in sequences]
embedded_next_words = [one_hot(next_word, len(unique_words))[0] for next_word in next_words]

embedded_sentences = np.asarray(embedded_sentences)
embedded_next_words = np.asarray(embedded_next_words)

print("Ejemplo de secuencia tokenizada: ",embedded_sentences[0] )
print("Ejemplo de próxima palabra tokenizada: ", embedded_next_words[0])
```

Código 45. División en secuencias y tokenización de los poemas para el segundo generador.

```
Número de palabras únicas: 51866
Ejemplo de secuencia tokenizada:  [20212 29361 45205 19848]
Ejemplo de próxima palabra tokenizada:  30122
```

Figura 35. Salida por consola. División en secuencias y tokenización de los poemas del segundo generador.

Como podemos ver, las secuencias tokenizadas tienen longitud ***maxlen*** y sus números van desde el 1 hasta el 51866. El 0 no se utiliza para indexar ningunas de las palabras.

Reservaremos un 10% de las muestras para llevar a cabo la evaluación del modelo tras el entrenamiento.

```
proporcion_evaluacion = 0.1
test_x = embedded_sentences[:int(len(embedded_sentences)*proporcion_evaluacion)]
x = embedded_sentences[int(len(embedded_sentences)*proporcion_evaluacion)+1:]
test_y = embedded_next_words[:int(len(embedded_next_words)*proporcion_evaluacion)]
y = embedded_next_words[int(len(embedded_next_words)*proporcion_evaluacion)+1:]
```

Código 46. Reserva de muestras para la evaluación del segundo generador.

Una vez realizadas estas preparaciones, podremos comenzar a configurar el modelo.

5.6.3 Configuración del modelo de red neuronal recurrente 2

En esta ocasión haremos uso de embeddings de palabras, por ello, necesitaremos utilizar en primer lugar una capa ***Embedding***. La configuraremos pasando como parámetros el tamaño del vocabulario que tenemos, le especificaremos también que queremos que se aprendan

embeddings de **100 dimensiones**. También le especificaremos que el 0 no es una palabra que registremos. Tras esta primera capa, haremos uso de una capa LSTM de **128 neuronas**. Le especificaremos que recibirá como entrada secuencias de ***maxlen*** palabras, siendo cada una de ella **1 solo número entero**. Por último, añadiremos una capa ***Dense*** con tantas neuronas como palabras únicas tengamos y, tras esta, una capa ***Activation*** haciendo uso de ***softmax***.

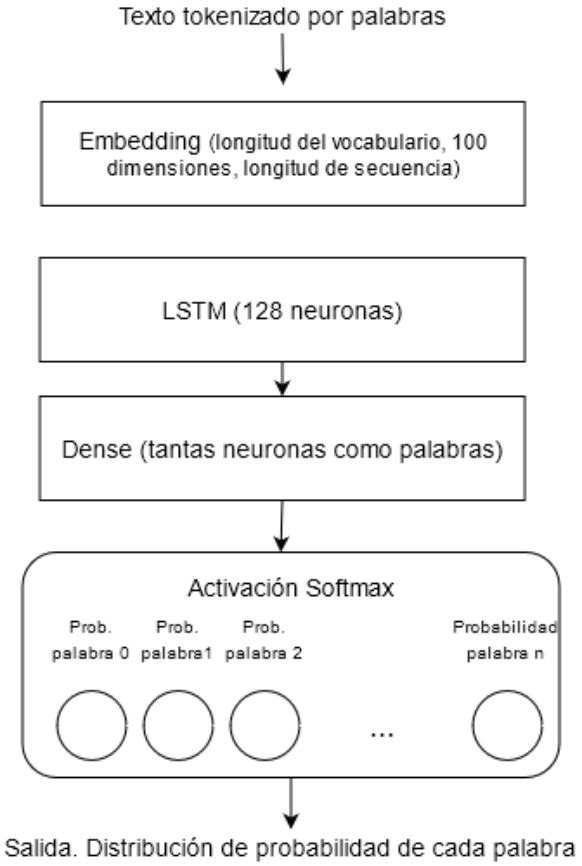


Figura 36. Arquitectura del segundo modelo.

```
model = Sequential()
model.add(Embedding(len(unique_words), 100, input_length=sequence_length, mask_zero = True))
model.add(LSTM(128, input_shape=(sequence_length, 1)))
model.add(Dense(len(unique_words)))
model.add(Activation('softmax'))
```

Código 47. Configuración del modelo del segundo generador.

```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_3 (Embedding)      (None, 4, 100)            5186600
_________________________________________________________________
lstm_3 (LSTM)                (None, 128)               117248
_________________________________________________________________
dense_3 (Dense)              (None, 51866)             6690714
_________________________________________________________________
activation_3 (Activation)    (None, 51866)             0
=================================================================
Total params: 11,994,562
Trainable params: 11,994,562
Non-trainable params: 0
_________________________________________________________________
```

Figura 37. Salida por consola. Resumen del modelo del segundo generador.

5.6.4 Monitorización del proceso de aprendizaje

Ahora prepararemos el modelo para ser entrenado. Para ello, esta vez usaremos el optimizador ***RMSProp*** con sus valores por defecto. Se encargará de ajustar los pesos de cada capa de manera que se mejore el rendimiento del modelo en la predicción de la próxima palabra.

También estableceremos como función de pérdida ***sparse_categorical_crossentropy***, dado que estamos ante un problema en el que existen varias categorías, en el que escoger una excluirá al resto. Además, escogemos esta función de pérdida porque el resultado es un entero cualquiera.

Por último, estableceremos como métrica la ***sparse_categorical_accuracy***, así podremos comprobar conforme pasen las épocas la precisión que se está obteniendo en la predicción de la próxima palabra.

```
model.compile(optimizer='rmsprop', loss='sparse_categorical_crossentropy', metrics=['sparse_categorical_accuracy'])
```

Código 48. Monitorización del entrenamiento del segundo modelo.

Una vez establecido esto, podremos pasar a entrenar el modelo.

5.6.5 Entrenamiento del modelo definido

Entrenaremos el modelo durante 25 épocas con lotes de 256 muestras. Realizaremos **validación** al final de cada época con el 20% de las muestras. Teniendo en cuenta el gran tamaño de vocabulario que tenemos y que hemos establecido 100 dimensiones para los embeddings, es de esperar que este entrenamiento tarde bastante tiempo en completarse.

```
history = model.fit(x, y, epochs = 25, batch_size = 256, validation_split = 0.2)
model.save_weights('generador_haikus2.h5')
```

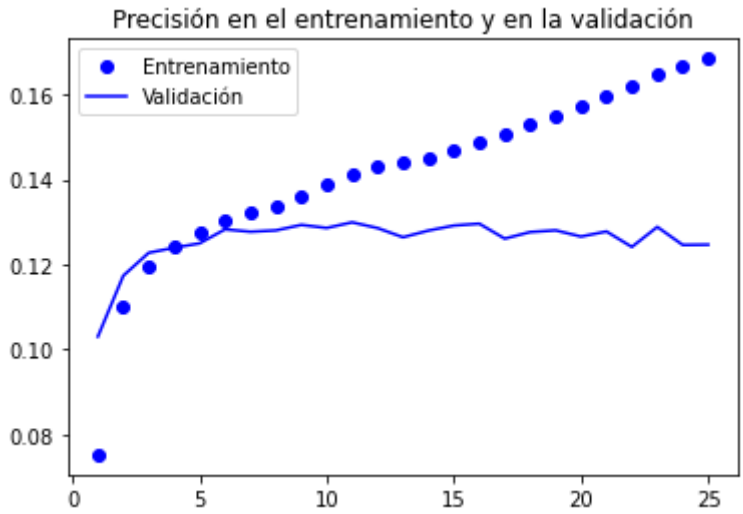
Código 49. Entrenamiento del segundo modelo.

Representaremos los datos obtenidos del entrenamiento mediante unas gráficas haciendo uso del módulo ***pyplot*** de ***matplotlib***. En la primera podremos comparar la precisión que ha obtenido el modelo en el entrenamiento y en la validación en cada una de las épocas.

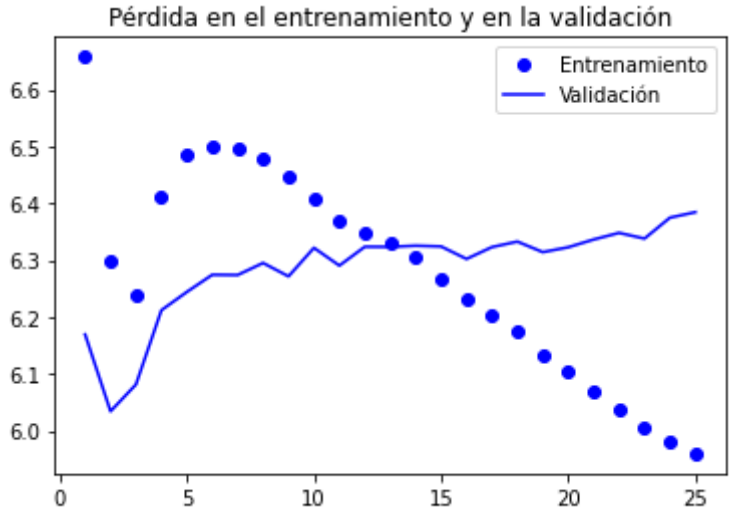
En la segunda podremos comprobar la evolución que ha habido en los valores de pérdida conforme pasaban las épocas. Tanto en el entrenamiento como en la validación.

```
import matplotlib.pyplot as plt

acc = history.history['sparse_categorical_accuracy']
val_acc = history.history['val_sparse_categorical_accuracy']
loss = history.history['loss']
val_loss = history.history['val_loss']

epochs = range(1, len(acc) + 1)

plt.plot(epochs, acc, 'bo', label = 'Entrenamiento')
plt.plot(epochs, val_acc, 'b', label = 'Validación')
plt.title('Precisión en el entrenamiento y en la validación')
plt.legend()

plt.figure()

plt.plot(epochs, loss, 'bo', label = 'Entrenamiento')
plt.plot(epochs, val_loss, 'b', label = 'Validación')
plt.title('Pérdida en el entrenamiento y en la validación')
plt.legend()

plt.show()
```

Código 50. Generación de gráficas de entrenamiento del segundo generador.



Cuadro 9. Precisión en el entrenamiento y en la validación del segundo generador.



Cuadro 10. Pérdida en el entrenamiento y en la validación del segundo generador.

Como podemos observar, además de los nefastos resultados en precisión y pérdida obtenidos, se ha producido sobreajuste a los datos de entrenamiento. Ello queda patente comprobando los valores de validación, que se mantienen estables a lo largo del tiempo, en contraste con los valores de entrenamiento, que van mejorando continuamente.

Ahora comprobaremos cómo se comporta el modelo con datos que no ha visto nunca.

5.6.6 Evaluación del segundo generador

Para llevar a cabo la evaluación del modelo, utilizaremos los datos que reservamos en el preprocesamiento de los datos. Estas muestras no han sido usadas para entrenar o validar el modelo en ningún momento, por lo que para él son totalmente nuevas.

Imprimiremos los resultados por pantalla para comprobarlos.

```
results = model.evaluate(test_x, test_y)
print("Pérdida en la evaluación:", results[0])
print("Precisión en la evaluación:", results[1])
```

Código 51. Evaluación del segundo generador.

```
188176/188176 [==============================] - 58s 308us/step
Pérdida en la evaluación: 8.345900101439096
Precisión en la evaluación: 0.08562728762626648
```

Figura 38. Salida por consola. Evaluación del segundo generador.

Los resultados muestran que el rendimiento del modelo es nefasto, con solo un 8,56% de precisión con datos que no ha visto nunca. El sobreajuste que vimos que se produjo en el entrenamiento no ayuda a este resultado, pero tratar de reducirlo no mejoraría significativamente el resultado que acabamos de obtener.

En esta ocasión no perderemos tiempo en tratar de generar poemas con este modelo, trataremos de llevar a cabo un tercer experimento que pueda obtener mejores resultados.

5.6.7 Discusión del segundo generador

Mientras se desarrollaba esta segunda versión se encontraron varios problemas, a los cuales se trató de encontrar una solución viable.

El primer problema es que, al ser un generador a nivel de palabra, la codificación **one-hot** resultaba ser muy pesada, requiriendo 909GB de espacio para poder ser usada. Para tratar de resolver este problema, se aplicó una capa de ***Embedding***, tratando que la red aprendiese a la vez las relaciones semánticas y, con el resto de capas, las relaciones sintácticas. Esto, a pesar de parecer una buena idea en la teoría, resultó ser fatal en la práctica, pues la capa de ***Embedding*** requiere de varias épocas para aprender relaciones entre palabras significativas, lo cual se acentúa con el gran tamaño del vocabulario del dataset.

```
encoded_docs = tokenizer.texts_to_matrix(sequences)
```

Código 52. Codificación one-hot para el segundo generador.

```
MemoryError: Unable to allocate 909. GiB for an array with shape (1881764, 64832) and data type float64
```

Figura 39. Salida por consola. Error de reserva de memoria para codificación one-hot para el segundo generador.

Sumado al problema anterior, no se han encontrado experimentos de referencia utilizando embeddings de palabras para generación de poemas en ***Keras***.

Tras llevar varios y largos intentos de entrenamiento obteniendo una precisión del 4,15%, se descubrió un error en el código. Se había configurado el modelo con activación ***softmax*** en la capa ***Dense*** y, también, en la capa ***Activation***, por lo que se estaba aplicando dos veces esta activación, llevando algunos casos a fallar. A pesar de solventar este fallo, la precisión no logró mejorar demasiado.

Una posible razón por la que no hayan funcionado demasiado bien los embeddings es la longitud de los versos. Los embeddings de palabras se suelen usar en textos largos en los que pueden tomarse secuencias de palabras más largas que en un poema de este tipo. Es posible que la longitud tan corta de las secuencias no favorezca la obtención de relaciones entre palabras.

Para tratar de mejorar este segundo modelo, podría aplicarse un mejor filtrado a los poemas, eliminando los que utilicen palabras extrañas o eliminando los que no tengan la estructura 5-7-5 de un haiku. También podría probar a usarse embeddings de palabras preentrenados.

5.8 Tercer generador

5.8.1 Definición

En este tercer acercamiento para tratar de resolver el problema trataremos de replicar el experimento que se lleva a cabo en el artículo "*Generating Haiku with Deep Learning (Part 1)*" de Jeremy Neiman. [9]

En este experimento, se trata de generar haikus a nivel de carácter haciendo uno de una red neuronal recurrente ***LSTM*** con tres líneas de entrenamiento paralelas. Mediante este

acercamiento se trata de conseguir que el modelo aprenda la estructura de un haiku y sea capaz de replicar estos poemas con precisión estructural.

5.8.2 Preprocesamiento del conjunto de datos

Primero, mostraremos el bloque de código con todas las librerías importadas para este Notebook.

```
from pathlib import Path
import re
import pandas as pd
import numpy as np
from keras.callbacks import ModelCheckpoint,  CSVLogger
from keras.layers import Add, Dense, Input, LSTM, Activation
from keras.models import Model, Sequential
from keras.preprocessing.text import Tokenizer
from keras.utils import np_utils
from keras.models import Sequential
from keras.optimizers import RMSprop
import inflect
import tensorflow as tf
from keras import backend as K
```

Código 53. Librerías importadas para el tercer generador.

Configuraremos la ruta al archivo que contiene el dataset.

```
root_path = Path('')
haiku_path = "all_haiku.csv"
```

Código 54. Configuración de la ruta del fichero de haikus para el tercer generador.

En esta ocasión trataremos de mejorar y exprimir al máximo el potencial del dataset que hemos estado utilizando en el resto de generadores.

En primer lugar, se preparará un diccionario de palabras y fonemas haciendo uso del diccionario fonético **CMU**, el cual puede descargarse desde http://www.speech.cs.cmu.edu/tools/lextool.html

Este diccionario almacena un conjunto de fonemas que pueden contar como una sílaba para distintas palabras.

También incluimos un fragmento de código para poder añadir palabras que el diccionario no tenga en cuenta, éstas deberán ser introducidas en el archivo "*custom.dict.txt*" siguiendo el formato del diccionario **CMU**.

Para este experimento no haremos uso de ésta funcionalidad, solo haremos uso de las palabras predeterminadas del diccionario.

```
# Carga los fonemas

# Diccionario estándar
WORDS = {}
with (root_path / 'cmudict.dict.txt').open('r') as f:
    for line in f.readlines():
        word, phonemes = line.strip().split(' ', 1)
        word = re.match(r'([^\(\)]*)(\(\d\))*', word).groups()[0]
        phonemes = phonemes.split(' ')
        syllables = sum([re.match(r'.*\d', p) is not None for p in phonemes])
        #print(word, phonemes, syllables)
        if word not in WORDS:
            WORDS[word] = []
        WORDS[word].append({
            'phonemes': phonemes,
            'syllables': syllables
        })

CUSTOM_WORDS = {}
vowels = ['AA', 'AE', 'AH', 'AO', 'AW', 'AX', 'AXR', 'AY', 'EH', 'ER', 'EY', 'IH', 'IX', 'IY', 'OW', 'OY', 'UH', 'UW', 'UX']
with (root_path / 'custom.dict.txt').open('r') as f:
    for line in f.readlines():
        try:
            word, phonemes = line.strip().split('\t', 1)
        except:
            print(line)
            continue
        word = re.match(r'([^\(\)]*)(\(\d\))*', word).groups()[0].lower()
        phonemes = phonemes.split(' ')
        syllables = sum([(p in vowels) for p in phonemes])

        if word not in CUSTOM_WORDS:
            CUSTOM_WORDS[word] = []
        CUSTOM_WORDS[word].append({
            'phonemes': phonemes,
            'syllables': syllables
        })
```

Código 55. Carga de datos del diccionario fonético CMU.

Una vez tenemos cargados los datos del diccionario de pronunciación, definiremos las funciones que leerán las líneas de los poemas, las limpiarán de símbolos y números y posteriormente contarán las sílabas que tienen.

```
inflect_engine = inflect.engine()

# Diccionario de palabras no encontradas, deberán buscarse los fonemas
NOT_FOUND = set()

def get_words(line):

    # Creamos una lista con las palabras del verso

    line = str(line)
    line = line.lower()
    # Reemplaza las números con su palabra escrita normalmente
    ws = []
    for word in line.split(' '):
        if re.search(r'\d', word):
            x = inflect_engine.number_to_words(word).replace('-', ' ')
            ws = ws + x.split(' ')
        else:
            ws.append(word)

    line = ' '.join(ws)

    words = []
    for word in line.split(' '):
        word = re.match(r'[\'"]*([\w\']*)[\'"]*(.*)', word).groups()[0]
        word = word.replace('_', '')
        words.append(word)

    return words

def count_non_standard_words(line):

    # Cuenta el número de palabras en el verso que no aparecen en el diccionario CMU.

    count = 0
    for word in get_words(line):
        if word and (word not in WORDS):
            count += 1
    return count
```

Código 56. Funciones auxiliares para contar el tercer generador.

En este caso, hacemos uso de la librería ***inflect*** para transformar dígitos a palabras.

```
def get_syllable_count(line):
    # Obtiene los posibles números de sílaba de la línea

    counts = [0]
    return_none = False
    for word in get_words(line):
        try:
            if word:
                if (word not in WORDS) and (word not in CUSTOM_WORDS):
                    word = word.strip('\'')

                if word in WORDS:
                    syllables = set(p['syllables'] for p in WORDS[word])
                else:
                    syllables = set(p['syllables'] for p in CUSTOM_WORDS[word])
                #print(syllables)
                new_counts = []
                for c in counts:
                    for s in syllables:
                        new_counts.append(c+s)

                counts = new_counts
        except:
            NOT_FOUND.add(word)
            return_none = True

    if return_none:
        return None

    return ','.join([str(i) for i in set(counts)])
```

Código 57. Función para contar sílabas del tercer generador.

A continuación leeremos los datos del dataset haciendo uso de la librería ***pandas***.

```
all_haikus = pd.read_csv(str(haiku_path), usecols=["0","1","2"])

all_haikus
```

Código 58. Carga de haikus del tercer generador.

Ahora aplicaremos las funciones que definimos anteriormente. Contaremos las sílabas de todos los versos y eliminaremos los poemas que tengan más de 3 palabras desconocidas.

También almacenaremos en "*unrecognized_words.txt*" las palabras que no se han podido reconocer. Para mejorar el funcionamiento de este modelo podrían añadirse éstas palabras al diccionario personalizable.

```
# Borra haikus con más de 3 palabras desconocidas
all_haikus['unknown_word_count'] = np.sum([all_haikus[str(i)].apply(count_non_standard_words) for i in range(3)], axis=0)
all_haikus = all_haikus[all_haikus['unknown_word_count'] < 3].copy()

for i in range(3):
    all_haikus['%s_syllables' % i] = all_haikus[str(i)].apply(get_syllable_count)

print("Palabras no reconocidas: ", len(NOT_FOUND))

with open('unrecognized_words.txt', 'w') as f:
    for w in NOT_FOUND:
        f.write(w)
        f.write('\n')

all_haikus
```

Código 59. Cuenta de palabras no reconocidas en los haikus.

```
Palabras no reconocidas:  5796
```

Figura 40. Salida por consola. Cuenta de palabras no reconocidas en los haikus.

Ahora haremos algo que no se lleva a cabo en el artículo. Debido a que nosotros tenemos un conjunto de haikus lo suficientemente grande, nos tomaremos la licencia de quedarnos solamente con los que tengan estructura 5-7-5.

```
# Filtramos el conjunto de poemas y nos quedamos con los que tengan estructura 5-7-5
corpus = []
all_haikus = all_haikus[all_haikus['0_syllables']=="5"]
all_haikus = all_haikus[all_haikus['1_syllables']=="7"]
all_haikus = all_haikus[all_haikus['2_syllables']=="5"]
df = all_haikus
```

Código 60. Filtrado de haikus con estructura 5-7-5.

Comenzamos a preparar los datos para el entrenamiento. Los inputs repetirán el primer carácter y los outputs se completarán añadiendo el próximo carácter, a excepción del último verso, que no lo hará.

```
# Tomamos el máximo tamaño de línea de todos los versos y eliminamos las muestras
# que sean más largas que el 99 percentil de longitud.

max_line_length = int(max([df['%s' % i].str.len().quantile(.99) for i in range(3)]))
df = df[
    (df['0'].str.len() <= max_line_length) &
    (df['1'].str.len() <= max_line_length) &
    (df['2'].str.len() <= max_line_length)
].copy()
df
# Aplicamos padding a los versos que no tienen el máximo tamaño de línea
# añadiendo \n al final de los mismos hasta llegar a este tamaño máximo
for i in range(3):
    # Para los inputs, duplicaremos el primer caracter
    df['%s_in' % i] = (df[str(i)].str[0] + df[str(i)]).str.pad(max_line_length+2, 'right', '\n')

    # Añadimos el primer carácter de la próxima línea si no es el último verso
    if i == 2: # Si es el último verso
        df['%s_out' % i] = df[str(i)].str.pad(max_line_length+2, 'right', '\n')
    else: # Si no es el último verso se añade el primer carácter de la próxima línea
        # Esto ayudará con el entrenamiento, de manera que la siguiente RNR tenga mejores probabilidades de
        # tomar el primer carácter correctamente.
        df['%s_out' % i] = (df[str(i)] + '\n' + df[str(i+1)].str[0]).str.pad(max_line_length+2, 'right', '\n')

max_line_length += 2
```

Código 61. Preparación de entradas y objetivos del tercer generador.

Comprobamos que las entradas y salidas serán, salvo las de la tercera línea, de esta forma:

0_in	0_out
wisiting the graves\n \n\n\n\n\n\n \n\n\n \n\n\n...	visiting the graves\ns \n\n\n\n\n\n \n\n\n \n\n\n...

Figura 41. Salida por consola. Ejemplo de formato de entradas y objetivos del tercer generador.

Ahora codificaremos los datos haciendo uso de codificación **one-hot** para que puedan ser utilizados por el modelo.

```
inputs = df[['0_in', '1_in', '2_in']].values

tokenizer = Tokenizer(filters='', char_level=True)
tokenizer.fit_on_texts(inputs.flatten())
n_tokens = len(tokenizer.word_counts) + 1
aux_X = [tokenizer.texts_to_sequences(inputs[:,i]) for i in range(3)]

# X es la entrada de cada línea en secuencias de caracteres codificados en one-hot
X = np_utils.to_categorical(np.array(aux_X), num_classes=n_tokens)

outputs = df[['0_out', '1_out', '2_out']].values

# Y es la entrada de cada línea en secuencias de caracteres codificados en one-hot
Y = np_utils.to_categorical([tokenizer.texts_to_sequences(outputs[:,i]) for i in range(3)], num_classes=n_tokens)

# X_syllables es la cuenta de sílabas de cada línea
X_syllables = df[['0_syllables', '1_syllables', '2_syllables']].values
```

Código 62. Codificación one-hot para el tercer generador.

Una vez hecho esto, estamos listos para preparar el modelo.

5.8.3 Configuración del modelo de red neuronal recurrente 3

Para este modelo, necesitaremos utilizar clases especiales haciendo uso de la API funcional de ***Keras***. Estas clases constituirán las distintas líneas de entrenamiento que se usarán en la capa ***LSTM***, de manera que puedan entrenarse simultáneamente.

Crearemos una clase ***TrainingLine*** que constituirá una de las líneas de entrenamiento. Después definiremos una función *"create_training_model"* que creará las líneas de entrenamiento en función de los parámetros que consideremos.

```
class TrainingLine:
    def __init__(self, name, previous_line, lstm, n_tokens):
        self.char_input = Input(shape=(None, n_tokens), name='char_input_%s' % name)

        self.syllable_input = Input(shape=(1,), name='syllable_input_%s' % name)
        self.syllable_dense = Dense(lstm.units, activation='relu', name='syllable_dense_%s' % name)
        self.syllable_dense_output = self.syllable_dense(self.syllable_input)

        #self.lstm = LSTM(latent_dim, return_state=True, return_sequences=True, name='lstm_%s' % name)

        #Si hay una linea previa, el estado inicial de ésta nueva será el que de como salida la anterior
        if previous_line:
            initial_state = [
                Add(name='add_h_%s' % name)([
                    previous_line.lstm_h,
                    self.syllable_dense_output
                ]),
                Add(name='add_c_%s' % name)([
                    previous_line.lstm_c,
                    self.syllable_dense_output
                ])
            ]
        else:
            initial_state = [self.syllable_dense_output, self.syllable_dense_output]

        self.lstm_out, self.lstm_h, self.lstm_c = lstm(self.char_input, initial_state=initial_state)

        self.output_dense = Dense(n_tokens, activation='softmax', name='output_%s' % name)
        self.output = self.output_dense(self.lstm_out)

def create_training_model(latent_dim, n_tokens):
    lstm = LSTM(latent_dim, return_state=True, return_sequences=True, name='lstm')
    lines = []
    inputs = []
    outputs = []
#Se crean todas las lineas de inputs, que se anclan a la capa LSTM que creamos previamente
    for i in range(3):
        previous_line = lines[-1] if lines else None
        lines.append(TrainingLine('line_%s' % i, previous_line, lstm, n_tokens))
        inputs += [lines[-1].char_input, lines[-1].syllable_input]
        outputs.append(lines[-1].output)

    training_model = Model(inputs, outputs)
    training_model.compile(optimizer='rmsprop', loss='categorical_crossentropy', metrics=['acc'])

    return training_model, lstm, lines, inputs, outputs
```

Código 63. Clases para la creación del modelo del tercer generador.

A continuación definiremos y entrenaremos el modelo haciendo uso de las clases y funciones que acabamos de crear.

La red consistirá en una capa ***LSTM*** de 2048 neuronas a la que se le pasarán 3 líneas de entrenamiento. En cada una de estas líneas se pasará un verso y las sílabas que tiene como estado. El número de sílabas, además, pasará por una capa ***Dense*** de tantas neuronas como tendrá la capa ***LSTM***, con activación ***relu***. Por último, cada línea terminará con una capa ***Dense*** con tantas neuronas como caracteres se estén utilizando y función de activación ***softmax***.

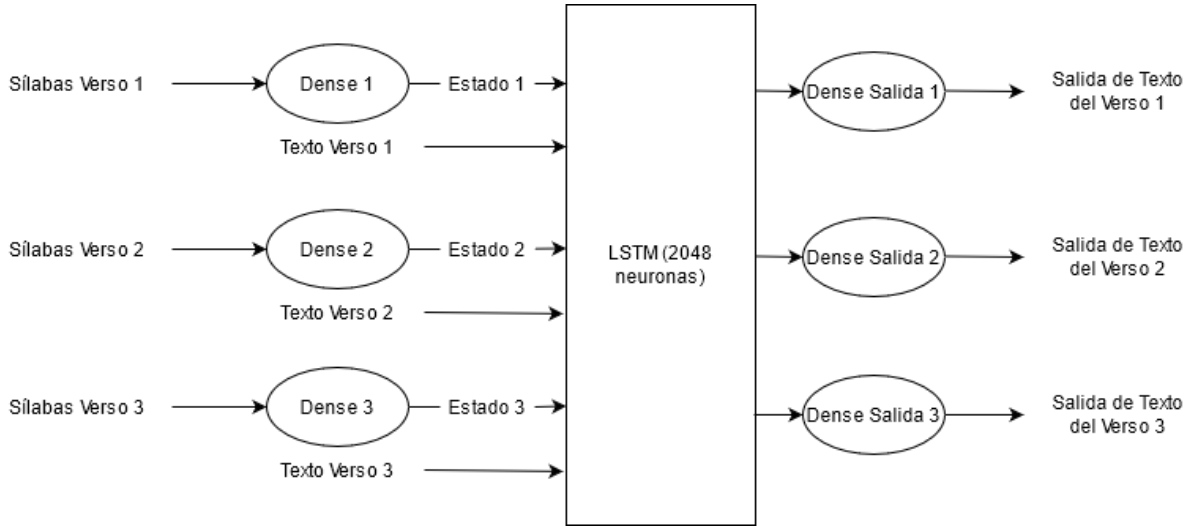


Figura 42. Arquitectura del tercer modelo.

```
tf_session = tf.Session()
K.set_session(tf_session)


output_dir = Path('output_%s' % "salida")
epochs = 10
X_syllables = np.array(X_syllables)
X_syllables = X_syllables.astype(np.float)
training_model, lstm, lines, inputs, outputs = create_training_model(2048, n_tokens)
```

Código 64. Configuración del modelo del tercer generador.

5.8.4 Monitorización del proceso de aprendizaje

En esta ocasión, monitorizaremos los resultados de cada época teniendo en cuenta la función de pérdida "***categorical_crossentropy***" y la precisión. Haremos uso del optimizador ***RMSprop***. Además, conforme pasen las épocas almacenaremos los pesos que mejores resultados hayan dado respecto a la pérdida haciendo uso de un ***ModelCheckpoint***. También almacenaremos en un archivo llamado "*training_log.csv*" los resultados del entrenamiento.

```
training_model.compile(optimizer='rmsprop', loss='categorical_crossentropy', metrics=['acc'])
```

Código 65. Configuración de la monitorización del tercer generador.

```
filepath = str(output_dir / ("%s-{epoch:02d}-{loss:.2f}-{val_loss:.2f}.hdf5" % 2048))
checkpoint = ModelCheckpoint(filepath, monitor='loss', verbose=1, save_best_only=True, mode='min')

csv_logger = CSVLogger(str(output_dir / 'training_log.csv'), append=True, separator=',')

callbacks_list = [checkpoint, csv_logger]
```

Código 66. Configuración de callbacks para el tercer generador.

Una vez configurados estos parámetros ya estamos listos para entrenar al modelo.

5.8.5 Entrenamiento del modelo definido

Entrenaremos la red durante 10 épocas con lotes de 256 muestras y reservando un 10% de las muestras para la validación. Es de esperar que este entrenamiento tarde bastante tiempo debido a la gran cantidad de parámetros internos que el modelo deberá ajustar.

```
history = training_model.fit([
    X[0], X_syllables[:,0],
    X[1], X_syllables[:,1],
    X[2], X_syllables[:,2]
], [Y[0], Y[1], Y[2]], batch_size=256, epochs=epochs, validation_split=.1, callbacks=callbacks_list)
```

Código 67. Entrenamiento del modelo del tercer generador.

A continuación se muestran los resultados de las últimas épocas del entrenamiento:

```
Epoch 00007: loss improved from 1.96168 to 1.85089, saving model to output_salida\2048-07-1.85-2.15.hdf5
Epoch 8/10
82574/82574 [==============================] - 304s 4ms/step - loss: 1.7404 - output_line_0_loss: 0.4454 - output_line_1_lo
ss: 0.8375 - output_line_2_loss: 0.4576 - output_line_0_acc: 0.8591 - output_line_1_acc: 0.7377 - output_line_2_acc: 0.8559
- val_loss: 2.1039 - val_output_line_0_loss: 0.5235 - val_output_line_1_loss: 0.9892 - val_output_line_2_loss: 0.5910 - val
_output_line_0_acc: 0.8392 - val_output_line_1_acc: 0.6979 - val_output_line_2_acc: 0.8201

Epoch 00008: loss improved from 1.85089 to 1.74040, saving model to output_salida\2048-08-1.74-2.10.hdf5
Epoch 9/10
82574/82574 [==============================] - 303s 4ms/step - loss: 1.6251 - output_line_0_loss: 0.4249 - output_line_1_lo
ss: 0.7834 - output_line_2_loss: 0.4168 - output_line_0_acc: 0.8652 - output_line_1_acc: 0.7544 - output_line_2_acc: 0.8686
- val_loss: 2.1423 - val_output_line_0_loss: 0.5286 - val_output_line_1_loss: 1.0065 - val_output_line_2_loss: 0.6071 - val
_output_line_0_acc: 0.8388 - val_output_line_1_acc: 0.6952 - val_output_line_2_acc: 0.8184

Epoch 00009: loss improved from 1.74040 to 1.62513, saving model to output_salida\2048-09-1.63-2.14.hdf5
Epoch 10/10
82574/82574 [==============================] - 302s 4ms/step - loss: 1.5078 - output_line_0_loss: 0.4064 - output_line_1_lo
ss: 0.7267 - output_line_2_loss: 0.3748 - output_line_0_acc: 0.8710 - output_line_1_acc: 0.7722 - output_line_2_acc: 0.8821
- val_loss: 2.1954 - val_output_line_0_loss: 0.5338 - val_output_line_1_loss: 1.0313 - val_output_line_2_loss: 0.6300 - val
_output_line_0_acc: 0.8386 - val_output_line_1_acc: 0.6941 - val_output_line_2_acc: 0.8163

Epoch 00010: loss improved from 1.62513 to 1.50784, saving model to output_salida\2048-10-1.51-2.20.hdf5
```

Figura 43. Salida por consola. Entrenamiento del modelo del tercer generador.

Como podemos observar, los resultados son bastante buenos, con una precisión en la validación de alrededor del 80% en el primer verso y en el último.

Sin embargo, el segundo verso obtiene alrededor de un 70% de precisión en la validación, algo que resulta llamativo.

5.8.6 Evaluación del tercer generador

En esta ocasión trataremos de generar poemas con distintas temperaturas y observaremos la estructura que se construye en los mismos.

Para esto, crearemos la función ***sample***, que se encargará de predecir el próximo carácter en base a unas probabilidades y un valor de temperatura.

También, en este caso, es necesario definir una clase ***GeneratorLine*** que represente las líneas de generador que se asocian a cada una de las ***TrainingLine*** del modelo.

Crearemos una clase ***Generator*** que será la que utilicemos para definir nuestro generador. Esta clase tendrá un método ***generate_haiku*** que recibirá como parámetros un *array* con las sílabas

que quisiéramos que tuviese el haiku, la temperatura a utilizar y el primer carácter utilizado si se desea.

Para generar los haikus se ejecutarán algunos métodos en la sesión de ***TensorFlow*** tales como ***feed_dict*** para poder introducir los datos según se generen al modelo.

```
def sample(preds, temperature=0.5):
    preds = np.asarray(preds).astype('float64')
    preds = np.log(preds) / temperature
    exp_preds = np.exp(preds)
    preds = exp_preds / np.sum(exp_preds)
    probas = np.random.multinomial(1, preds, 1)
    return np.argmax(probas)

class GeneratorLine:
    def __init__(self, name, training_line, lstm, n_tokens):
        self.char_input = Input(shape=(None, n_tokens), name='char_input_%s' % name)

        self.syllable_input = Input(shape=(1,), name='syllable_input_%s' % name)
        self.syllable_dense = Dense(lstm.units, activation='relu', name='syllable_dense_%s' % name)
        self.syllable_dense_output = self.syllable_dense(self.syllable_input)

        self.h_input = Input(shape=(lstm.units,), name='h_input_%s' % name)
        self.c_input = Input(shape=(lstm.units,), name='c_input_%s' % name)
        initial_state = [self.h_input, self.c_input]

        self.lstm = lstm

        self.lstm_out, self.lstm_h, self.lstm_c = self.lstm(self.char_input, initial_state=initial_state)

        self.output_dense = Dense(n_tokens, activation='softmax', name='output_%s' % name)
        self.output = self.output_dense(self.lstm_out)

        self.syllable_dense.set_weights(training_line.syllable_dense.get_weights())
        #self.lstm.set_weights(lstm.get_weights())
        self.output_dense.set_weights(training_line.output_dense.get_weights())
```

Código 68. Funciones auxiliares para el tercer generador (I).

```
class Generator:
    def __init__(self, lstm, lines, tf_session, tokenizer, n_tokens, max_line_length):
        self.tf_session = tf_session
        self.tokenizer = tokenizer
        self.n_tokens = n_tokens
        self.max_line_length = max_line_length

        self.lstm = LSTM(
            lstm.units, return_state=True, return_sequences=True,
            name='generator_lstm'
        )
        self.lines = [
            GeneratorLine(
                'generator_line_%s' % i,
                lines[i], self.lstm, self.n_tokens
            ) for i in range(3)
        ]
        self.lstm.set_weights(lstm.get_weights())
```

Código 69. Funciones auxiliares para el tercer generador (II).

```
def generate_haiku(self, syllables=[5, 7, 5], temperature=.1, first_char=None):
    output = []
    h = None
    c = None

    if first_char is None:
        first_char = chr(int(np.random.randint(ord('a'), ord('z')+1)))

    next_char = self.tokenizer.texts_to_sequences(first_char)[0][0]

    for i in range(3):
        line = self.lines[i]

        s = self.tf_session.run(
            line.syllable_dense_output,
            feed_dict={
                line.syllable_input: [[syllables[i]]]
            }
        )
        if h is None:
            h = s
            c = s
        else:
            h = h + s
            c = c + s

        line_output = [next_char]

        end = False
        next_char = None
        for i in range(self.max_line_length):
            char, h, c = self.tf_session.run(
                [line.output, line.lstm_h, line.lstm_c],
                feed_dict={
                    line.char_input: [[
                        np_utils.to_categorical(
                            line_output[-1],
                            num_classes=self.n_tokens
                        )
                    ]],
                    line.h_input: h,
                    line.c_input: c
                }
            )
```

Código 70. Función "generate_haiku" para el tercer generador (I).

```
            char = sample(char[0,0], temperature)
            if char == 1 and not end:
                end = True
            if char != 1 and end:
                next_char = char
                char = 1

            line_output.append(char)

        cleaned_text = self.tokenizer.sequences_to_texts([
            line_output
        ])[0].strip()[1:].replace(
            '  ', '\n'
        ).replace(' ', '').replace('\n', ' ')
        #print(line_output)
        print(cleaned_text)
        output.append(cleaned_text)

    return output
```

Código 71. Función "generate_haiku" para el tercer generador (II).

Ahora crearemos un modelo que servirá como placeholder para los pesos que obtuvimos durante el entrenamiento.

```
# Creamos un nuevo placeholder para el modelo
training_model, lstm, lines, inputs, outputs = create_training_model(2048, n_tokens)

# Cargamos los pesos que queramos, especificando el archivo que los guardó
training_model.load_weights(output_dir / '2048-10-1.51-2.20.hdf5')
```

Código 72. Creación de modelo placeholder y asignación de pesos.

Crearemos el generador con el modelo que acabamos de instanciar.

```
generator = Generator(lstm, lines, tf_session, tokenizer, n_tokens, max_line_length)
```

Código 73. Creación del tercer generador.

A continuación podemos ver qué es lo que se genera al hacer uso del método ***generate_haiku***, mostrando los valores numéricos que se obtienen antes de ser traducidos a palabras y filtrados para que sean legibles.

```
generator.generate_haiku(temperature = 0.1)
print()
```

Código 74. Generación de un haiku con el tercer generador.

```
[16, 16, 8, 6, 4, 3, 13, 2, 4, 10, 3, 2, 18, 6, 8, 13, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
united the wind
[2, 2, 5, 20, 2, 4, 10, 3, 2, 13, 3, 24, 6, 12, 2, 22, 16, 4, 2, 6, 2, 13, 5, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
f the devil but i do
[8, 8, 5, 4, 2, 10, 7, 24, 3, 2, 4, 10, 3, 2, 4, 6, 14, 3, 2, 4, 5, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
not have the time to
```

Figura 44. Salida por consola. Generación de un haiku con el tercer generador.

Para evaluar el generador, generaremos 50 haikus con distintas temperaturas, siempre con el valor por defecto de sílabas 5-7-5. Confiamos en que esta estructura se mantendrá.

En primer lugar se generarán 50 haikus con temperatura 0.1, después se generarán con temperatura 0.3 y, por último, con 1.

```
for i in range(50):
    generator.generate_haiku()
    print()
```

Código 75. Generación de 50 haikus con temperatura 0.1 en el tercer generador.

```
for i in range(50):
    generator.generate_haiku(temperature=.3)
    print()
```

Código 76. Generación de 50 haikus con temperatura 0.3 en el tercer generador.

```
for i in range(50):
    generator.generate_haiku(temperature=1)
    print()
```

Código 77. Generación de 50 haikus con temperatura 1.0 en el tercer generador.

Destacamos los siguientes haikus para cada una de las temperaturas establecidas, entre paréntesis completaremos las palabras de los primeros versos que lo necesiten:

Haiku destacado para temperatura 0.1

only the stream is

(t)he most beautiful person

in the world to me

Haiku destacado para temperatura 0.3

just say something that

(c)an happen to me I love

you and my brother

Haiku destacado para temperatura 1

no matter how much

(y)ou come through her is so much

happy for him if

Como podemos observar, sorprendentemente la mayoría de poemas mantienen el esquema 5-7-5, probablemente gracias a que se filtraron las muestras para que tuvieran esta estructura.

Observamos que no se generan correctamente las primeras letras de la segunda línea en un gran número de ocasiones, pero esto no es un problema demasiado grave, ya que pueden intuirse con facilidad.

Conforme aumenta la temperatura las palabras se vuelven más aleatorias e impredecibles, creando palabras que no existen en realidad pero que suenan creíbles. También, cuanto más se aumenta este valor menos sentido parecen tener los haikus.

Podemos ver que la estructura, en su mayoría, sintácticamente tiene sentido. Sin embargo vemos que en muchas ocasiones los poemas no llegan a tener demasiado sentido, pero sí que

podemos encontrar alguno que sorprende y podría hacer creer al lector que está escrito por un humano.

5.8.7 Discusión del tercer generador

Durante el desarrollo de este tercer generador se encontraron algunos problemas.

El más notable es que al utilizar la API funcional de ***Keras*** resultó complicado comprender y preparar el modelo.

Otra dificultad fue conseguir generar haikus tras entrenar el modelo, dado que era necesario hacer uso de ***TensorFlow***. Aunque esta versión fue creada siguiendo un artículo como referencia, se trató de adaptar el código de ***TensorFlow 1.14.0*** a versiones actuales, pero no hubo éxito en esta tarea.

El generador no escribe el primer carácter del segundo verso en un gran número de ocasiones. Esto podría ser debido a que el generador, cuando limpia el texto para mostrarlo por pantalla, elimina los primeros caracteres de los versos debido a que normalmente éstos se repiten.

A pesar de los problemas encontrados en el desarrollo de este experimento, podemos concluir que ha sido el que mayor éxito ha tenido, llegando a generar algunos poemas que podrían servir como medio auxiliar para la inspiración de un poeta.

Capítulo 6. DIFICULTADES TÉCNICAS

Durante la realización de este proyecto, surgieron algunas dificultades técnicas y problemas que hubo que solucionar para seguir adelante.

Uno de los grandes fallos que se cometieron fue la planificación inicial. Se planificó el trabajo de una forma que no se ajustó a la realidad. Debido a la subestimación de carga de trabajo por otras fuentes, y al desconocimiento de la metodología adecuada para la realización de un proyecto de esta índole, fue necesaria una replanificación según pasó el tiempo para conseguir lograr los objetivos propuestos. A continuación se muestra la planificación modificada, según los problemas que fueron surgiendo.

Fase 1 – Planificación

- **Tareas**
  - **Tarea 1.1** - Definición de la planificación temporal del proyecto
  - **Tarea 1.2** - Establecimiento de la metodología de trabajo
  - **Tarea 1.3** - Estudio de costes del proyecto
  - **Tarea 1.4** - Desarrollo de un plan de contingencia

Fase 2 – Estudio teórico

- **Tareas**
  - **Tarea 2.1** - Introducción al Deep Learning
  - **Tarea 2.2** - Lectura de los capítulos 1, 2, 3 y 6 de *"Deep Learning con Python"*
  - **Tarea 2.3** - Búsqueda de información complementaria sobre Deep Learning aplicado a procesamiento de texto
  - **Tarea 2.4** - Estudio de las herramientas
  - **Tarea 2.5** – Desarrollo de Notebooks de aprendizaje
- **Incremento** – Documentación de casos de estudio y estado del arte

Fase 3 – Desarrollo de la solución

- **Tareas**
  - **Tarea 3.1** - Búsqueda de artículos y experimentos de referencia: *Papers with code*
  - **Tarea 3.2** - Búsqueda y estudio de datasets
  - **Tarea 3.3** - Diseño teórico de la solución
  - **Tarea 3.4** - Preparación del entorno de desarrollo
  - **Tarea 3.5** – Desarrollo y pruebas de la solución
  - **Tarea 3.6** - Estudio de posibles variaciones de la solución y posible desarrollo de las mismas
- **Incremento** - Memoria del desarrollo de la solución

Fase 4 – Preparación de la memoria

- **Tareas**
  - **Tarea 4.1** – Redacción de la memoria en su conjunto
  - **Tarea 4.2** – Repaso de formato y revisión de la memoria

- **Incremento** - Documento de la memoria finalizado

Fase 5 – Cierre del proyecto

- **Tareas**
  - **Tarea 5.1** – Redacción de la presentación
  - **Tarea 5.2** - Preparación de la presentación
- **Incremento** – Presentación finalizada

Fase	Inicio	Fin
Fase 1 - Planificación	24/02/2021	03/03/2021
Fase 2 - Estudio teórico	04/03/2021	19/03/2021
Fase 3 - Desarrollo de la solución	20/03/2021	24/05/2021
Fase 4 – Preparación de la memoria	25/05/2021	20/06/2021
Fase 5 – Cierre del proyecto	21/06/2021	05/07/2021

Tabla 2. Replanificación de fases

Por otra parte, los costes no se vieron modificados. No se cumplieron las 20 horas semanales a rajatabla, invirtiendo algunas semanas más tiempo y otras menos, pero se cumplió el objetivo de 300 horas totales.

Una vez comenzado el desarrollo de la solución, se encontraron diversos problemas.

El entorno de trabajo configurado con ***Anaconda***, ***Jupyter Notebook***, ***Python 3.8*** y ***Keras*** con ***TensorFlow***, no estaba haciendo uso de la GPU para la ejecución de los Notebooks. Esto mermó la capacidad computacional que se tenía al comienzo del trabajo, aumentando drásticamente los tiempos de ejecución. Para solucionarlo se hizo uso temporalmente de ***Google Colab***, pero finalmente se decidió buscar una solución más cómoda e indefinida al problema.

La solución, tras varias horas de búsquedas e intentos fallidos, fue crear un entorno virtual nuevo en ***Anaconda*** en el que se hiciera uso de ***Python 3.7*** y versiones previas de ***Keras*** y ***TensorFlow***, poniendo énfasis en los paquetes ***keras-gpu*** y ***tensorflow-gpu*** que, por incompatibilidad de versiones, no estaban detectando la GPU. También fue necesario instalar los paquetes ***cudatoolkit*** y ***cudnn*** para la correcta ejecución de los modelos haciendo uso de la tarjeta gráfica. Además, debido a que el ordenador que se estaba utilizando tiene varios discos duros, hubo un pequeño fallo al instalar ***Visual Studio 2019*** en una ruta que no se estaba estableciendo correctamente en las variables de entorno del sistema, esto causaba algunos problemas a la hora de instalar las herramientas CUDA.

Avanzando en el desarrollo, en el tercer modelo, surgió otro problema grave. La versión de ***TensorFlow*** que se estaba utilizando en el artículo a seguir era una versión mayor por debajo de la que se estaba utilizando en el entorno de trabajo. En el artículo se hacía uso de ***Tensorflow 1.14.0*** y en el entorno virtual se estaba usando ***Tensorflow 2.4.1***. Este cambio provocó la necesidad de tener que refactorizar código y modernizarlo. Sin embargo, no se estudió en ningún momento la sintaxis de los programas escritos para ser ejecutados directamente por

Tensorflow, por lo que los pequeños fragmentos de código que hacían uso directo de esta librería resultaron, por una cuestión de inexperiencia, imposibles de adaptar.

Con el tiempo apremiando la solución encontrada a este problema fue crear otro entorno virtual en ***Anaconda*** haciendo uso de ***Python 3.6***, con ***Keras 2.3.1*** y ***TensorFlow 1.14.0***. De esta manera el código puede ser ejecutado sin necesidad de ser modificado.

Un problema que persistió a lo largo del desarrollo fue la falta de información para la generación de poemas con Deep Learning. La mayoría de artículos encontrados emulaban a otros, conduciendo en la mayoría de ocasiones a la misma resolución tecnológica, un generador a nivel de carácter. La falta de retos en ***Kaggle*** referentes a generación de poemas tampoco fue de ayuda. Esto provocó que el desarrollo del trabajo se ralentizase.

Capítulo 7. CONCLUSIONES Y TRABAJOS FUTUROS

El trabajo realizado en base al libro y los artículos citados en las referencias ha posibilitado adquirir un aprendizaje y conocimiento en profundidad del Deep Learning. Gracias a ellos ha sido posible conocer este campo de manera teórica y experimental.

El principal objetivo de este trabajo era el entendimiento y aprendizaje de los conceptos básicos del Deep Learning para aplicarlos en un problema concreto, la generación de haikus. En primer lugar, aprendimos las características principales que debía tener un haiku en el capítulo de "Contexto del problema". Tras esto, mediante la lectura comprensiva del libro "*Deep Learning con Python"*, de François Chollet, hemos sido capaces de dar los primeros pasos en este campo del Machine Learning, desarrollando unos notebooks de estudio que verifican la correcta asimilación de los conceptos básicos relacionados con el Deep Learning para secuencias de texto tales como el funcionamiento de una red neuronal y el preprocesamiento de datos de texto. Adicionalmente, además de estos conceptos básicos, aprendimos el funcionamiento de una red neuronal recurrente y cómo hacer uso de los modelos construidos con ellas para generar texto.

La segunda parte del objetivo principal, la aproximación al problema de la generación de haikus, ha sido realizada de tres maneras distintas, fruto de una evolución continua en el aprendizaje del contexto del problema durante el desarrollo de cada uno de los modelos.

El primer acercamiento al problema fue realizado aplicando los conocimientos de generación de texto en prosa a nivel de carácter. Los resultados obtenidos, como era de esperar, no fueron sorprendentes. El modelo es incapaz de mantener un esquema métrico de un haiku, aunque en ocasiones consiguió formar versos coherentes.

El segundo acercamiento comenzó una vez el primero fue refinado al máximo con los conocimientos que en ese momento se tenían. En este se trató de crear un modelo de generación de texto a nivel de palabra. Los resultados obtenidos en los entrenamientos, de bastante larga duración, no lograron pasar del 9% de precisión en la predicción de próximas palabras. Esto, unido a la frustración por el tiempo dedicado sin obtención de resultados favorables, hizo que este modelo se quedara en ese estado del desarrollo, sin llegar a generar texto en ningún momento. Se ha revisitado este modelo con el paso del tiempo para tratar de detectar algún fallo en su diseño o posible mejora, pero lo máximo que se consiguió fue una mejora de un 4.5% aproximadamente de la precisión en la predicción.

El tercer acercamiento comenzó a desarrollarse antes de completar el segundo. La idea tras este era hacer uso del modelo propuesto en el artículo de Jeremy Neiman [9]. Este modelo, debido a su complejidad, resultó complicado de entender e implementar. Fue necesario hacer uso de características avanzadas de Keras, como su API funcional, e incluso ejecutar código en la sesión de TensorFlow. A pesar de la problemática que rodeaba este modelo, fue el que mejores resultados tuvo en cuanto a estructura métrica y coherencia del texto generado, llegando a escribir texto en ocasiones, que podría haber sido escrito por un humano – uno que no sepa demasiado sobre haikus –. Este tercer modelo podría ser utilizado como herramienta auxiliar para obtener inspiración a la hora de escribir un haiku.

El desarrollo de la solución estuvo lleno de inconvenientes técnicos que fueron retrasándolo, tal y como se explica en el capítulo de “Dificultades técnicas”. Sin embargo, pese a esta problemática, ha sido posible dar una solución aceptable al problema de la generación de haikus.

De cara al futuro se realizarán modificaciones a cada uno de los modelos desarrollados. Pondremos el foco en aplicar el mismo cribado del dataset que se realizó en el tercer modelo y en buscar más formas de mejorarlo. Trataremos de utilizar lo aprendido en el segundo modelo y en el tercero para intentar desarrollar uno nuevo que divida las palabras en sílabas y tratar a éstas como propias palabras para ser usadas en una capa de Embedding. Dado que las sílabas en los poemas resultan ser lo más importante, quizás este acercamiento pueda ser el más acertado. Además de estas ideas, también entraremos más en profundidad en el campo del Procesamiento del Lenguaje Natural para apoyar el futuro trabajo.

El estudio explicado en este documento es una pequeña parte del Deep Learning. Este campo actualmente está en constante crecimiento y, en concreto su uso en secuencias de textos, es algo bastante reciente que aún está siendo investigado. Los resultados de distintos experimentos hacen que pensemos que esto solo es el principio y nos incitan a seguir trabajando no solo este campo, sino en todo el Deep Learning en su conjunto.

Referencias

[1] F. Chollet, Deep Learning con Python, Anaya, 2020.

[2] C. Sánchez Ruiz, «Qué es (y qué no es) un haiku,» [En línea]. Disponible en: https://www.tallerdeescritores.com/que-es-un-haiku. [Último acceso: 19 Junio 2021].

[3] F. Rodríguez-Izquierdo, El haiku japonés. Historia y traducción, Hiperión, 2010.

[4] J. D. Villanueva García, «Redes neuronales desde cero (I) – Introducción,» 23 Octubre 2020. [En línea]. Disponible en:

https://www.iartificial.net/redes-neuronales-desde-cero-i-introduccion/. [Último acceso: 17 Junio 2021].

[5] G. Julián, «Las redes neuronales: qué son y por qué están volviendo,» 21 Enero 2016. [En línea]. Disponible en:

https://www.xataka.com/robotica-e-ia/las-redes-neuronales-que-son-y-por-que-estan-volviendo. [Último acceso: 20 Junio 2021].

[6] R. E. López Briega, «Introducción al Deep Learning,» 13 Junio 2017. [En línea]. Disponible en:

https://relopezbriega.github.io/blog/2017/06/13/introduccion-al-deep-learning/. [Último acceso: 20 Junio 2021].

[7] F. Sancho Caparrini, «Aprendizaje Supervisado y No Supervisado,» 14 Diciembre 2020. [En línea]. Disponible en:

http://www.cs.us.es/~fsancho/?e=77. [Último acceso: 20 Junio 2021].

[8] Google, «Redes neuronales de clases múltiples: Softmax,» 10 Febrero 2020. [En línea]. Disponible en:

https://developers.google.com/machine-learning/crash-course/multi-class-neural-networks/softmax?hl=es. [Último acceso: 20 Junio 2021].

[9] J. Neiman, «Generating Haiku with Deep Learning (Part 1),» 29 Diciembre 2018. [En línea]. Disponible en:

https://towardsdatascience.com/generating-haiku-with-deep-learning-dbf5d18b4246. [Último acceso: 19 Junio 2021].

[10] R. Aguiar y K. Liao, «Autonomous Haiku Generation,» 20 Junio 2019. [En línea]. Disponible en:

https://paperswithcode.com/paper/autonomous-haiku-generation. [Último acceso: 26 Mayo 2021].

Glosario de términos

Lote: Conjunto de muestras que se utiliza para entrenar una red neuronal.

Tensor: Objeto matemático cuyos componentes pueden distribuirse a lo largo de distintas dimensiones. Cada dimensión contiene un subconjunto de las anteriores. Así, un tensor 2D está compuesto de tensores 1D, que a su vez están compuestos de tensores 0D. Existen tensores 0D (escalares), 1D (vectores), 2D (matrices 2D), 3D (matrices 3D)… etc.

Evaluación: En el ámbito del Machine Learning, consiste en la realización de pruebas con datos desconocidos para el modelo. Siempre es recomendable hacer uso de esta técnica para confirmar si los resultados del entrenamiento de un modelo han sufrido sobreajuste.

Validación: En el ámbito del Machine Learning, consiste en realizar pruebas con distintos datos a los que se han utilizado en el entrenamiento, obteniendo una estimación del funcionamiento del modelo con datos con los que no ha entrenado. La diferencia con la **evaluación** es que en la validación pueden contenerse datos que el modelo ya ha utilizado anteriormente.