Universidad Zaragoza

Facultad de Ciencias
Universidad Zaragoza

Máster Universitario en Modelización e Investigación Matemática, Estadística y Computacional 2020/2021

Trabajo Fin de Máster

Análisis de técnicas de Deep learning para la generación automática de Keywords

Ignacio Oscoz Villanueva

Directores
Jon Pey Pérez
Luis Mariano Esteban Escaño

10 de septiembre de 2021

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

Los cambios vertiginosos de las últimas décadas han transformado a la sociedad moderna industrial al grado de que ya es comúnmente aceptado que se vive en un nuevo tipo de sociedad y en una nueva era, la de la información. Esta era, también llamada era digital o era informática, designa al periodo en el que el movimiento de información se volvió más rápido que el movimiento físico, gracias a la creación y desarrollo de las tecnologías digitales de la información y la comunicación. La mayoría de esta información sin embargo, no se encuentra en forma de tablas numéricas, sino mas bien en formas textuales. Según [19], la industria estima que solo el 21 por-ciento de la información está estructurada. Por información estructurada nos referimos a aquella información donde los campos de datos se alinean uno al lado del otro en longitudes de registro fijas, con campos de datos específicos que aparecen en ubicaciones estáticas dentro de cada registro. Los datos no estructurados no contienen un formato de registro establecido; pueden tener cualquier forma o formato.

La creación de información de cada individuo es constante, ya sea por ejemplo en búsquedas de Google, donde la mayor parte de la información existe en forma textual, que es de naturaleza muy desestructurada. Ahora, para crear bits significativos de conocimiento a partir de esta información, es importante conocer los sistemas de Procesamiento del Lenguaje Natural (PLN) o en inglés *Natural Language Processing* (NLP). La PLN es el área de inteligencia artificial que se encarga de gestionar los lenguajes humanos. Es la técnica computacional en la que se representa y analiza el lenguaje automáticamente. Pese a ser un campo de investigación bastante nuevo, su investigación está creciendo a una velocidad muy alta. Y es que, los impresionantes resultados obtenidos con el aprendizaje profundo (o *deep learning* en inglés) en visión por computadora, reconocimiento de patrones y análisis de tráfico de red hicieron que los investigadores de PLN siguieran la misma tendencia.

Uno de los múltiples usos que se le puede dar a este nuevo campo es por

ejemplo establecer vínculos entre clientes potenciales y productos con fines de marketing. Los motores de búsqueda como Google, recuperan todos los documentos que contienen las palabras clave que especificamos. No hay valor añadido a los datos. La minería de textos lleva las cosas un paso más allá al extraer información precisa basada en mucho más que palabras clave. En su lugar, busca entidades o conceptos, relaciones, frases y/o oraciones. Intenta determinar el significado real basado en algoritmos de Procesamiento del Lenguaje Natural, que le permiten reconocer conceptos similares. Una búsqueda utilizando la minería de texto puede identificar hechos, relaciones e inferencias que no son del todo obvios.

1.1. Objetivos

Diferentes técnicas de inteligencia artificial han sido utilizadas como herramientas para automatizar la generación y gestión de campañas de marketing en la plataforma Google Ads. En esta plataforma, se realizan mas de 3 billones de búsquedas por día, generando una cantidad ingente de información. En este contexto, la minería de textos puede servir para la generación automática de palabras clave utilizando técnicas de *deep learning*.

En este trabajo se propone desarrollar una herramienta sistemática, basada esencialmente en redes neuronales, para la generación de *Keywords* (o palabras clave) relevantes. Existen distintas herramientas capaces de abordar este objetivo, y por eso el objetivo de este trabajo es implementar y determinar cuál resulta mas conveniente a través de su análisis. Con este fin, se ha realizado un estudio previo sobre el estado del arte para identificar las posibles alternativas que existen, y tratar de implementar algunas de ellas, con la intención de después analizar su funcionamiento. Para ello, se ha contado con la colaboración de la empresa Quarizmi, que además de proporcionar las bases de datos que se han utilizado, ha servido de guía para cumplir los objetivos pre-establecidos.

Para intentar cumplir dichas metas, se ha utilizado el lenguaje de programación *Python* en su versión *3.8*. Además se ha hecho uso de librerías como *Tensorflow* y *Keras*, dos librerías Open Source que nos permiten adentrarnos en el *Deep Learning* de forma sencilla.

1.2. Estructura de la memoria

Este trabajo pretende explicar, de forma precisa y descriptiva, la parte teórica empleada en el desarrollo del trabajo, así como los resultados obtenidos y las conclusiones finales. La memoria consta de cinco capítulos: el primer y actual capítulo se ha dedicado fundamentalmente a la introducción del trabajo y los

objetivos propuestos, el segundo a la representación de la información, el siguiente a los aspectos teóricos de las redes neuronales, otro a la explicación de las diferentes implementaciones realizadas, detallando sus resultados correspondientes, y un último a las conclusiones generales y trabajo futuro.

El segundo capítulo se centra en el campo de la minería de textos y sus principales etapas. La obtención y el preprocesado de los textos, así como la representación numérica de ellos, es fundamental para un correcto funcionamiento de modelos de aprendizaje automático.

El capítulo tres empieza explicando los detalles mas básicos de las redes neuronales. A medida que avanza el capítulo se van introduciendo conceptos más complejos para terminar explicando aquellos detalles que en el posterior capítulo acabaremos poniendo en práctica como las redes LSTM, las redes convolucionales y sus configuraciones de entrenamiento.

Como anticipaba, el siguiente apartado tratará sobre abordar los problemas y objetivos que nos hemos propuesto haciendo uso de los conceptos teóricos que se han ido desarrollando en los anteriores capítulos. Más concretamente se tratarán de entrenar dos modelos de redes LSTM (uno para la generación de *keywords* por caracteres y otro para la generación por palabras) y dos modelos de redes convolucionales (uno para la generación de *keywords* por caracteres y otro para la generación por palabras). Una vez entrenados los modelos se tratará de analizar críticamente los resultados obtenidos y de ver si se han conseguido cumplir los objetivos propuestos de generar palabras clave.

Para acabar, en el capítulo cinco se desarrollan las conclusiones generales de este Trabajo Fin de Máster y se valoran las líneas de trabajo futuro.

En el anexo se incluye parte del código de programación desarrollado para este trabajo. Más concretamente, se adjuntan los *script* principales de *Python* en los que se incluyen la implementación de los entrenamientos de las redes neuronales y la generación de textos.

Capítulo 2

Minería de textos

Tradicionalmente, el tratamiento de la información se ha realizado sobre bases de datos numéricos que vienen dados en forma totalmente estructurada. Sin embargo, como se ha mencionado anteriormente, la mayoría de la información de la que se dispone hoy en día se encuentra en formato textual, lo que es consecuencia directa del uso de Internet. Por poner un ejemplo, según un estudio de Domo [5], empresa encargada de conectar los datos con los usuarios para las empresas, en el año 2020, en un solo minuto, se estimaron que de media se produjeron al rededor de 4,5 de millones de búsquedas en Google, se enviaron casi 42 millones de mensajes en WhatsApp y se 'clickaron' a unos 140000 anuncios de Instagram.

El interés en tratar esta información que proviene de textos hace que el campo de la minería de textos se encuentre ahora mismo en auge. Se entiende por minería de textos la rama de la lingüística computacional cuyo objeto es la obtención de información que no se encuentra de forma explícita en un conjunto de textos. Esta busca extraer información útil e importante de formatos de documentos heterogéneos, como páginas web, correos electrónicos, publicaciones en redes sociales, artículos de revistas... Esto a menudo se hace identificando patrones dentro de los textos, como tendencias en el uso de palabras, estructura sintáctica... La gente suele hablar de "minería de texto y datos (MTD)" al mismo tiempo, pero estrictamente hablando, la minería de texto es una forma específica de minería de datos que se ocupa del texto.

La tecnología de minería de texto ahora se aplica por una amplia variedad de usuarios, desde organizaciones gubernamentales, instituciones de investigación y empresas para sus necesidades diarias. A continuación se muestran algunos ejemplos de uso en diferentes campos:

- Investigación: Un investigador humano necesita mucho tiempo para analizar y obtener información relevante. En algunos casos, esta información ni siquiera es posible obtenerla con su simple lectura. La minería de textos

permite a los investigadores encontrar más información y de una manera más rápida y eficiente.

- Negocios: Las grandes empresas utilizan la minería de textos para ayudar en la toma de decisiones y para responder rápidamente las consultas de los clientes.
- Seguridad: El análisis de extracción de texto de blogs y otras fuentes de texto en línea se utiliza para prevenir delitos en Internet y luchar contra el fraude.
- Diariamente: Los sitios web de correo electrónico utilizan la minería de texto para crear métodos de filtrado más fiables y eficaces. También se utiliza para fines de redes sociales al identificar las relaciones entre los usuarios y ciertos productos o para determinar las opiniones de los usuarios sobre temas particulares.

2.1. Etapas de minería de texto

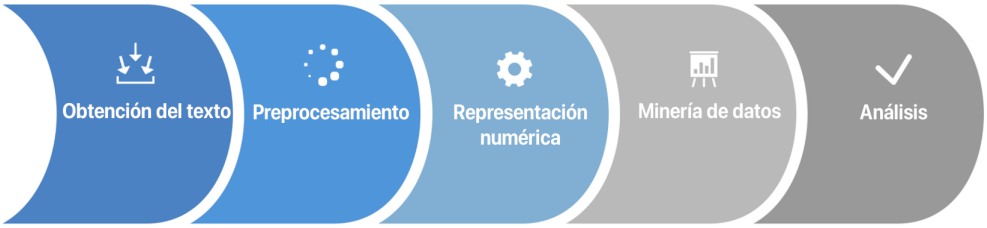


Figura 2.1: Etapas de un proceso de minería de textos

Un proceso de minería de textos consta de cinco etapas distintas [4], como se observa en la figura 2.1.

2.1.1. Obtención del texto

Recopilación de datos de diferentes recursos, como sitios web, correos electrónicos, comentarios de clientes, archivos de documentos. Dependiendo de la aplicación, este proceso puede ser completamente automatizado o guiado por el minero de texto. En tareas de aprendizaje supervisado, además del texto deberemos obtener los datos de las variables de salida. En nuestro caso, las variables de salida serán el propio texto.

2.1.2. Pre-procesamiento del texto

Procesamiento previo, como identificación de contenido y extracción de características representativas. Se trata de una etapa en la que se simplifica el problema original, ya sea eliminando información no relevante o reduciendo la dimensionalidad del problema. Todos los elementos del texto no ofrecen la misma información. Por ejemplo, en la tokenización[1] por palabras, los tokens denominados *stop words*, no ofrecen nada mas que ruido y es por ello que suelen eliminarse. La consideración de una palabra como *stop word* dependerá de la tarea que estemos realizando. Por ejemplo, un número puede considerarse como tal para ciertas tareas pero no para otras. Para la reducción de dimensionalidad, entre otras técnicas, también se puede emplear el lematizado en caso de que estemos ante una tokenización por palabras. Con ello, se sustituyen todas las palabras por su lema, manteniendo la información que ofrecen pero pudiendo así reducir considerablemente la dimensionalidad. En nuestro problema, al tratarse de generación de textos, no se considerará ningún tipo de eliminación de *stop words* ni de sustitución por lema. Además, con el objetivo de poder ver las diferencias entre ambas, se tratará con dos tokenizaciones distintas: una por caracteres y otra por palabras.

2.1.3. Representación numérica

Una vez tokenizado el texto y habiéndolo ya pre-procesado, seguimos teniendo un conjunto de frases, palabras o letras de las que directamente no podemos aplicar ningún modelo de minería de datos. Para ello, es necesario obtener una representación numérica. Existen distintas técnicas para la vectorización del texto pero en este trabajo se hará uso de la codificación *one-hot* y del *Word2Vec*.

codificación one-hot

La codificación *one-hot* es una de las técnicas mas simples para la vectorización de los tokens. Este método representa cada token con un vector de forma $\{0,1\}^{|V|}$, donde V es el conjunto total de tokens diferentes del texto. Así pues, cada coordenada del vector corresponde con el token del conjunto V ordenado según su aparición. Dicho de otro modo, cada token del texto se representa con un vector de dimensión igual a número de tokens en el conjunto V, tomando su índice el valor 1 y 0 en el resto. Para entender mejor esto, veamos dos ejemplos distintos:

[1] Los tokens son las unidades individuales de significado con las que se está operando. Pueden ser palabras, caracteres o incluso oraciones completas. La tokenización es el proceso de dividir los documentos de texto en esas partes.

- Ejemplo en el caso de tokenización por palabras: Supongamos que tenemos un vocabulario ordenado con las siguientes palabras: {"mineria", "de", "texto"}. Entonces, los vectores que los representarían serían respectivamente $(1, 0, 0)$, $(0, 1, 0)$ y $(0, 0, 1)$.
- Ejemplo en el caso de tokenización por palabras: Ahora supongamos que tenemos la palabra "texto" y la quisiéramos tokenizar por caracteres para después codificarla con *one-hot*. Entonces nuestro conjunto de tokens ordenado sería el siguiente: {"t", "e", "x", "o"}; y su codificación la que viene respectivamente: $(1, 0, 0, 0)$, $(0, 1, 0, 0)$, $(0, 0, 1, 0)$ y $(0, 0, 0, 1)$.

Este método no es capaz de captar ningún tipo de significado entre sus tokens debido a que todos ellos son equidistantes entre sí, y cuando el número de tokens es elevado aparecen problemas de dimensionalidad. Por eso, en general no es recomendable para los casos de tokenización por palabras. No obstante, dado que los caracteres carecen de significado semántico propio y el número de diferentes caracteres en un texto no suele ser muy elevado, la vectorización por *one-hot* es una técnica bastante usada en los textos tokenizados por caracteres.

Word Embeddings

Los *word embeddings* son una clase de técnicas en las que las palabras individuales se representan como vectores de valor real en un espacio vectorial predefinido [3]. Cada palabra se asigna a un vector y los valores del vector se aprenden de una manera que se asemeja a una red neuronal y, por lo tanto, la técnica a menudo se agrupa en el campo del aprendizaje profundo. La representación distribuida se aprende en base al uso de palabras. Esto permite que las palabras que se usan de manera similar tengan representaciones similares, capturando su significado semántico natural. Existen varias técnicas que se pueden clasificar como dentro de los *word embeddings*; he aquí unos ejemplos [9]:

- *Word2Vec*[12] [13]: Este método revolucionó el campo de la PLN con su aparición en el año 2013, y se ha convertido en uno de los más populares. *Word2Vec* predice la palabra de destino a partir del contexto dado de palabras con un modelo de bolsa continua de palabras (CBOW) o predice el contexto de la palabra de destino mediante un *skip-gram*. El modelo CBOW determina la probabilidad condicional de la palabra objetivo al dar palabras de contexto, mientras que un modelo de *skip-gram* hace exactamente lo contrario, lo que determina las palabras de contexto circundantes al dar palabras centrales. Esto permite que su representación vectorial sea capaz de realizar operaciones como la de "Rey - hombre + mujer = Reina". Entre sus limitaciones destaca la incapacidad de representar grupos de palabras como por ejemplo "perrito caliente", que lo representaría como dos palabras ("perro" y "caliente") que nada tienen que ver con su verdadero significado. Esta técnica se explicará con más detenimiento en el capítulo 3.

- *GloVe* [17]: El algoritmo del *Global Vectors* o *GloVe* es una extensión del método *Word2Vec* para el aprendizaje eficiente de vectores de palabras. Las representaciones de palabras del modelo de espacio vectorial clásico se desarrollaron utilizando técnicas de factorización matricial que hacen un buen trabajo al usar estadísticas de texto global, pero no son tan buenos como los métodos aprendidos como *Word2Vec* para capturar el significado y demostrarlo en las tareas como calcular analogías (por ejemplo, el ejemplo de Rey y Reina anterior). En lugar de utilizar una ventana para definir el contexto local, GloVe construye una matriz explícita de palabra-contexto o ocurrencia de palabras utilizando estadísticas de todo el texto.

- *FastText* [2]: Otra extensión del primer método. Este representa cada palabra como un n-grama de caracteres en lugar de representar palabras directamente. Esta técnica es útil para capturar la semántica de palabras pequeñas. El *FastText* tiene la ventaja de representar palabras raras que pueden no haber sido vistas en el tiempo de entrenamiento.

2.1.4. Minería de datos

Obtenida la representación numérica de los datos textuales, seríamos capaces de aplicar modelos matemáticos que nos permitiesen alcanzar el objetivo de la tarea. Es decir, ahora aplicaríamos lo que se denomina la minería de datos. Nuestro objetivo es el de generar texto a partir de lo que se aprende en los conjuntos de datos obtenidos. Para ello, se hará uso de dos tipos de redes neuronales artificiales distintas, como lo son las redes neuronales recurrentes LSTM y las redes neuronales convolucionales, que se explicarán en el capítulo 3. Con esto, captaremos ciertos patrones en la representación numérica de los textos que a su vez nos permitirá generar texto nuevo.

2.1.5. Evaluación e interpretación de los resultados

En esta etapa nos encargaremos de analizar el funcionamiento de los modelos de generación de texto. En un problema de clasificación es tan fácil como por ejemplo ver las veces que acierta el modelo. Sin embargo, como veremos más adelante, en la generación de texto es algo mas complicado.

Capítulo 3

Redes neuronales

El *deep learning* o aprendizaje profundo es el aprendizaje automático con redes neuronales artificiales profundas, y el objetivo de este capítulo será explicar su funcionamiento.

3.1. Introducción

Las redes neuronales son modelos simples del funcionamiento del sistema nervioso. Las unidades básicas son las neuronas, que generalmente se organizan en capas. Una red neuronal artificial es un modelo simplificado que emula el modo en que el cerebro humano procesa la información: Funciona simultaneando un número elevado de unidades de procesamiento interconectadas que parecen versiones abstractas de neuronas.

Las unidades de procesamiento se organizan en capas. Hay tres partes normalmente en una red neuronal: una capa de entrada, con unidades que representan los campos de entrada; una o varias capas ocultas; y una capa de salida, con una unidad o unidades que representa el campo o los campos de destino. Las unidades se conectan con fuerzas de conexión variables (o ponderaciones). Los datos de entrada o *inputs* se presentan en la primera capa, y los valores se propagan desde cada neurona hasta cada neurona de la capa siguiente. Al final, se envía un resultado desde la capa de salida denominado *output* o simplemente valor de salida.

En la imagen 3.1 podemos ver un ejemplo de una red neuronal formada por tres capas. La capa de entrada consta de tres neuronas y cada una de ellas puede aceptar un valor de entrada, y están representadas por las variables x_1, x_2, x_3. Aceptar los datos de entrada es lo único que hace la primera capa. Cada neurona de la capa de entrada puede tener una única salida. Es posible tener menos valores de entrada

que las neuronas de entrada (entonces puede pasarles un valor de 0 a las neuronas no utilizadas), pero la red no puede tomar más valores de entrada que número de neuronas de entrada. Las entradas se pueden representar como una secuencia $x_1, x_2, ..., x_n$ (que en realidad es lo mismo que un vector de fila) o como un vector de columna $\boldsymbol{x} := (x_1, x_2, ..., x_n)^T$. Hay diferentes representaciones de la misma, y siempre elegiremos la representación que haga más fácil y rápido el cálculo de las operaciones que podamos necesitar. En nuestra elección de representación de datos, no estamos limitados por nada más que por la eficiencia computacional.

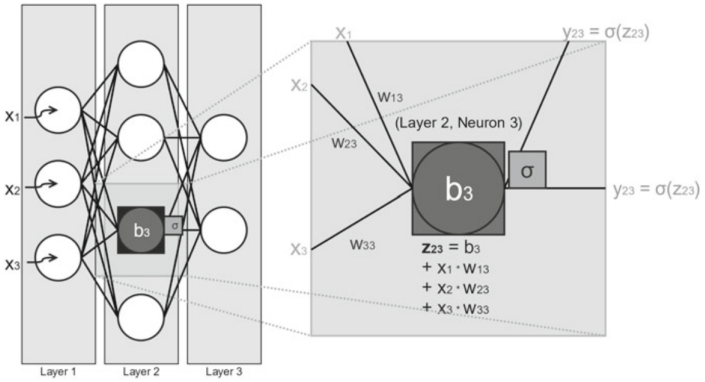


Figura 3.1: Una red neuronal superficial.

Como ya hemos señalado, todas las neuronas de la capa de entrada están conectadas a todas las neuronas de la capa oculta, pero las neuronas de la misma capa no están interconectadas.

Cada conexión entre la neurona j en la capa k y la neurona m en la capa n tiene un peso denotado por w_{jm}^{kn}, y, dado que por lo general queda claro en el contexto qué capas están involucradas, podemos omitir el superíndice y escribir simplemente w_{jm}. El peso regula cuánto del valor inicial se reenviará a una neurona determinada.

Volviendo a la figura 3.1 observamos que la neurona ampliada (neurona 3 de la capa 2) obtiene la entrada que es la suma de los productos de las entradas de la capa anterior y los pesos respectivos. En este caso, las entradas son x_1, x_2 y x_3, y los pesos son w_{13}, w_{23} y w_{33}. Cada neurona tiene un valor modificable, llamado sesgo o *bias* en inglés, que está representado aquí por b_3, y este sesgo se suma a la suma anterior. El resultado de esto se llama *logit* y tradicionalmente se denota por z (en nuestro caso, z_{23}).

Excepto en las neuronas lineales donde el valor del *logit* es proporcional al de salida, normalmente suele aplicarse una función no-lineal, $\sigma(z)$, denominada función de activación para obtener el *output* (tradicionalmente denotado por y) de la neurona. Matemáticamente, el *output* de la neurona j de la capa k se escribe de

la siguiente manera:

$$y_j^k = \sigma\left(\sum_{k=1}^{n} w_{kj}x_k + b_j\right) \tag{3.1}$$

donde n es el número de entradas, w_{kj} son los pesos asociados a los *input* x_k, y b_j el sesgo asociado a la neurona j. El valor de salida se utilizará como valor de entrada de otras neuronas, y repitiendo el proceso, finalmente obtendremos el valor de salida final.

Existen distintas funciones de activación que se pueden usar en distintas capas. Estas funciones, básicamente deciden si los valores introducidos a una neurona deben ser activadas o no, es decir, si su información es relevante o no. Esto es importante en la forma en que una red aprende porque no toda la información es igualmente útil. Parte de esa es solo ruido. Aquí es donde entran en escena las funciones de activación. Las funciones de activación ayudan a la red a utilizar la información importante y suprimir los puntos de datos irrelevantes. Las funciones de activación más destacadas en la literatura son las siguientes:

- Función *ReLU*: Pese a su simplicidad el ReLU o unidad lineal rectificada es la función de activación más utilizada en estos momentos. Matemáticamente viene dada de la siguiente manera:

$$\text{ReLU}(x) = \text{máx}\,(0, x) = \begin{cases} 0 & \text{si } x < 0 \\ x & \text{en el resto} \end{cases} \tag{3.2}$$

  Como puede verse, el ReLU está medio rectificado (desde abajo). $\sigma(z)$ es cero cuando z es menor que cero y $\sigma(z)$ es igual a z cuando z es superior o igual a cero. El problema de esta función es que todos los valores negativos se vuelven cero inmediatamente, lo que disminuye la capacidad del modelo para ajustarse o entrenarse a partir de los datos correctamente.

- Función *LeakyReLU*: En un intento de resolver el problema de la *ReLU* surge esta función con la siguiente expresión:

$$\text{LReLU}(x) = \begin{cases} \alpha x & \text{si } x < 0 \\ x & \text{en el resto} \end{cases} \tag{3.3}$$

  Tomando α valores cercanos a 0,01. Esta simple modificación ayuda a aumentar el rango de la función ReLU, permitiendo así aumentar la capacidad de ajuste y entrenamiento.

- Función sigmoide: La función sigmoide, también denominada función logística, está definida de esta manera:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{3.4}$$

  Al dar valores entre 0 y 1 resulta útil para los casos en los que se quiere predecir una probabilidad como salida. Su principal inconveniente es que puede hacer que una red neuronal se atasque en el momento del entrenamiento (por el denominado *vanishing grandient problem*).

- Función tangente hiperbólica: La tangente hiperbólica es similar a la función logística pero trasformando el valor de x en el rango de $[-1, 1]$. Matemáticamente se presenta tal que así:

$$\tanh x = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{3.5}$$

  Su principal ventaja respecto a la anterior es que las entradas negativas darán salidas fuertemente negativas y las entradas cercanas a cero darán salidas cercanas a cero. La función tanh se utiliza principalmente para la clasificación entre dos clases. Presenta el mismo problema que la función sigmoide: el problema del desvanicimiento del gradiente.

- Función Softmax: La función Softmax se describe a menudo como una combinación de múltiples sigmoides. Como hemos dicho, la función sigmoide devuelve valores entre 0 y 1, que pueden tratarse como probabilidades de que un punto de datos pertenezca a una clase o a otra. De una manera similar la función Softmax puede utilizarse para problemas de clasificación multiclase. Esta función devuelve valores en el rango de $[0, 1]$, cuya suma es igual a 1. Es por ello que se puede interpretar como una probabilidad y es utilizada en la capa de salida, como probabilidad de pertenecer a una cierta clase. La función Softmax se define de la siguiente manera:

$$\sigma(x_j) = \frac{e^{x_j}}{\sum_{k=1}^{K} e^{x_k}} \quad \text{donde } j = 1, ..., K \tag{3.6}$$

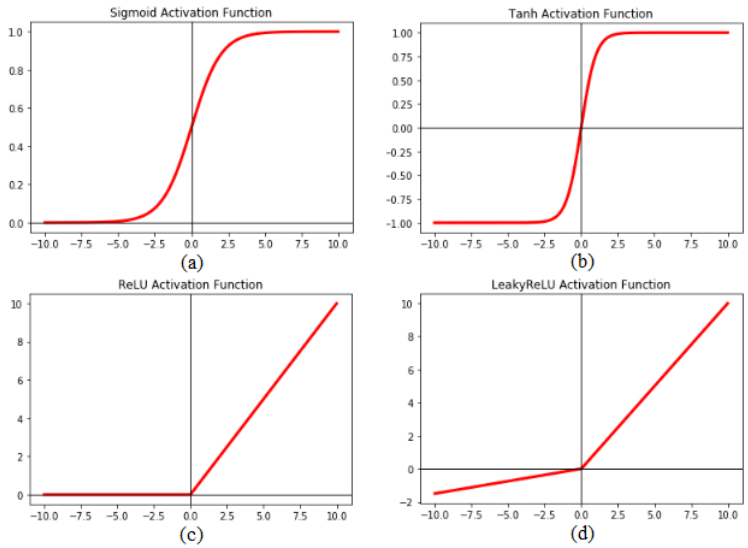


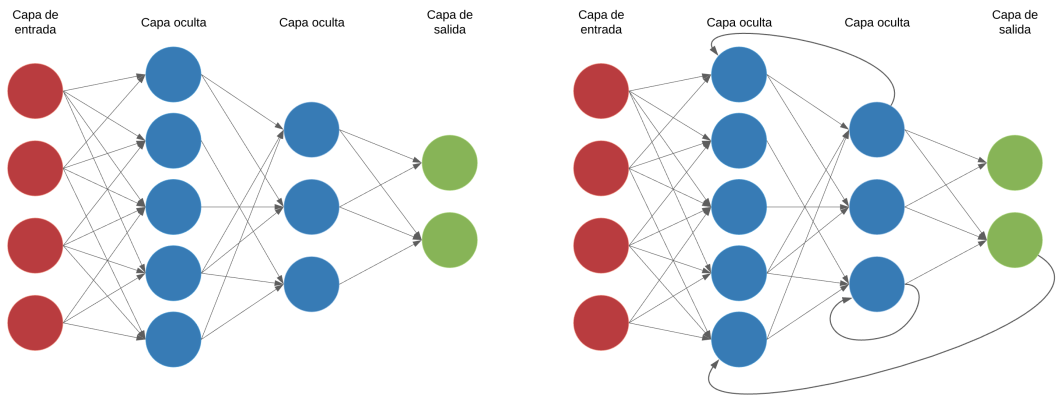
Figura 3.2: Algunas de las funciones de activación más populares.

Una red neuronal aprende examinando los registros individuales, generando una predicción para cada registro y realizando ajustes a las ponderaciones cuando realiza una predicción incorrecta. Este proceso se repite muchas veces y la red sigue mejorando sus predicciones hasta haber alcanzado uno o varios criterios de parada.

Al principio, todas las ponderaciones son aleatorias y las respuestas que resultan de la red son, posiblemente, disparatadas. La red aprende a través del entrenamiento. Continuamente se presentan a la red ejemplos para los que se conoce el resultado, y las respuestas que proporciona se comparan con los resultados conocidos. A medida que progresa el entrenamiento, la red se va haciendo cada vez más precisa en la replicación de resultados conocidos. Una vez entrenada, la red se puede aplicar a casos futuros en los que se desconoce el resultado.

Antes de continuar con la explicación es necesario comentar la clasificación de las redes neuronales según su configuración. Principalmente existen dos grupos de redes neuronales:

- Redes neuronales *Feedforward* (cuyas siglas en inglés son FNN): Como su nombre indica, en estas redes la información solamente se propaga hacia adelante. Es decir, la salida de una neurona solamente puede ser entrada de las neuronas de la siguiente capa.
- Redes neuronales recurrentes (cuyas siglas en inglés son RNN): En este tipo de redes la salida de una neurona puede ser entrada de neuronas de capas anteriores o de la misma capa. Se puece dar también, que la salida de una neurona, sea entrada de la misma.



(a) FNN (b) RNN

Figura 3.3: Ejemplo de las dos tipos de redes neuronales según su configuración.

3.2. Redes neuronales *feedforward* y conceptos básicos

En la literatura, uno de los primeros pasos a la hora de explicar los conceptos más básicos en las redes neuronales suele ser introduciendo ejemplos de redes neuronales *feedforward* o de propagación hacia adelante. De hecho, antes de haber hecho esta clasificación, en este trabajo ya habíamos visto un ejemplo de este tipo de redes (figura 3.1). Como suele ser habitual, comenzaremos explicando las redes más simples para ir escalando hacia casos más complejos.

3.2.1. Perceptrón simple

Formado por una única neurona de función escalón de Heaviside, también llamada función escalón unitario, el perceptrón simple es el primer paso para comprender el funcionamiento de las redes neuronales. Formalmente se define así:

$$z = b + \sum_i w_i x_i \tag{3.7}$$

$$y = \begin{cases} 1 & \text{si } z \leq 0 \\ 0 & \text{en el resto} \end{cases} \tag{3.8}$$

Donde x_i son las entradas, w_i los pesos, b es el sesgo y z es el logit. Es posible absorber el sesgo como uno de los pesos, por lo que solo necesitamos una regla de actualización del peso. Esto se muestra en la ecuación 3.9: para absorber el sesgo como un peso, es necesario agregar una entrada x_0 con el valor 1 y el sesgo es su peso. Es decir:

$$z = b + \sum_i w_i x_i = w_0 x_0 (= b) + w_1 x_1 + w_2 x_2 ... \tag{3.9}$$

De acuerdo con la ecuación anterior, b podría ser x_0 o w_0 (el otro debe ser 1). Dado que queremos cambiar el sesgo con el aprendizaje, y las entradas nunca cambian, debemos tratarlo como un peso. A este procedimiento lo llamamos absorción de sesgo o *bias absortion*.

El entrenamiento del perceptrón se hace mediante la llamada regla de aprendizaje del perceptrón, como sigue:

1. Se elige un caso de entrenamiento.
2. Si la salida predicha coincide con la etiqueta de salida, no se hace nada.
3. Si el perceptrón predice un 0 y debería haber predicho un 1, se agrega el vector de entrada al vector de peso.

4. Si el perceptrón predice un 1 y debería haber predicho un 0, se resta el vector de entrada al vector de peso

El perceptrón simple puede resultar útil en problemas simples de clasificación binaria. Sin embargo, esta estructura se ve muy limitada en problemas de clasificación no lineales, y le resulta imposible clasificar correctamente problemas no linealmente separables, como por ejemplo un problema XOR. Para solucionar estos problemas surge el perceptrón multicapa.

3.2.2. Perceptrón multicapa y la regla de *backpropagation*

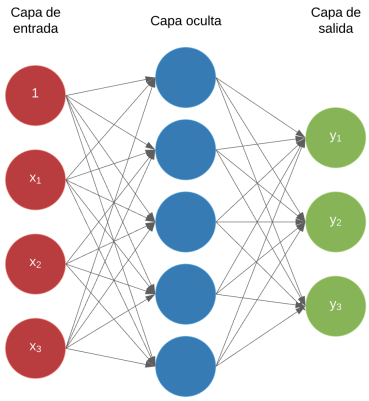


Figura 3.4: Perceptrón multicapa con una capa oculta.

El perceptrón multicapa es un modelo de redes neuronales formado por una capa de entrada, una o varias capas ocultas y una capa de salida con una conexión completa hacia adelante de las neuronas y con funciones de activación no lineales. En la imagen 3.4 se puede observar lo que sería un ejemplo de perceptrón multicapa con una única capa oculta. La salida de esta, es fácilmente deducible a partir de la expresión 3.1, y tendrá la siguiente forma:

$$y_i = \tilde{\sigma}\left(\sum_{j=0}^{m} w_{ji}^{(2)}\sigma\left(\sum_{k=0}^{n} w_{kj}^{(1)}x_k\right)\right) \tag{3.10}$$

donde $w_{ji}^{(2)}$ representa el peso asociado a la conexión de la neurona j de la segunda capa con la neurona i de la capa de salida, $w_{kj}^{(1)}$ representa el peso asociado a la conexión de la neurona k de la capa de entrada con la neurona j de la capa oculta, σ representa la función de activación de la capa oculta y $\tilde{\sigma}$ representa la función de activación de la capa de salida. Por otra parte, según esta notación, n representará el número de neuronas en la capa de entrada, mientras que m representará el número de neuronas de la capa oculta.

Si en vez de tener un modelo con una única capa oculta tuviésemos uno con dos capas ocultas como el que observamos en la figura 3.5 la expresión de 3.10 podría generalizarse de esta manera:

$$y_i = \tilde{\tilde{\sigma}} \left(\sum_{j=0}^{m} w_{ji}^{(3)} \tilde{\sigma} \left(\sum_{k=0}^{n} w_{kj}^{(2)} \sigma \left(\sum_{l=0}^{p} w_{lk}^{(1)} x_k \right) \right) \right) \tag{3.11}$$

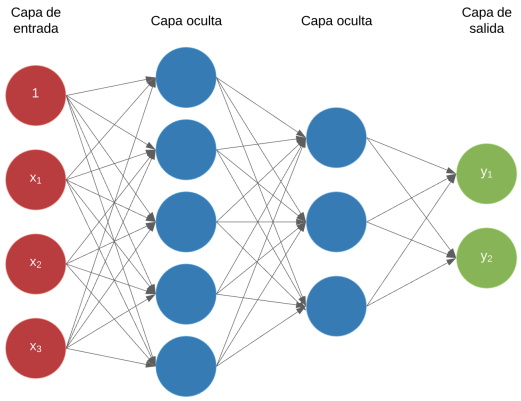


Figura 3.5: Perceptrón multicapa con dos capas ocultas.

No es difícil generalizar aún más la ecuación 3.11 para un caso totalmente general de $C \geq 3$ capas y obtener la siguiente expresión recursiva:

$$y_i = \sigma \left(\sum_{j=0}^{n_{C-1}} w_{ji}^{C-1} x_j^{C-1} \right) \ , \quad i = 1, ..., n_C \tag{3.12}$$

$$x_j^{C-1} = \sigma \left(\sum_{k=0}^{n_{C-2}} w_{kj}^{C-2} x_k^{C-2} \right) \ , \quad j = 1, ..., n_{C-1} \tag{3.13}$$

siendo donde n_C y n_{C-1} el número de neuronas en la capa de C y $C-1$ respectivamente, w_{ji}^{C-1} el peso asociado a la neurona j de la capa $C-1$ con la neurona i de la capa C y x_j^{C-1} la salida de la neurona j de la capa $C-1$. A pesar de que en esta expresión se denote de la misma manera, σ, a las funciones de activación de cada capa, es importante recalcar que no tienen porqué ser las mismas funciones.

Después de que en el año 1969 Minsky y Papert [14] demostraran que el perceptrón simple no podía resolver tareas de clasificación no lineales, se empezó a pensar que la combinación de varios perceptrones simples sería capaz de resolver ese problema. El principal obstáculo con la creación del perceptrón multicapa fue que se desconocía cómo extender la regla de aprendizaje del perceptrón para que funcionase con múltiples capas. Dado que se necesitan múltiples capas, la única opción parecía ser abandonar la regla del perceptrón y usar una regla diferente que

sea más robusta y capaz de aprender los pesos a través de la capa. Así de la mano de Paul Werbos[23] primero, y David Parker[16], Yann LeCun[10] y, Rumelhart, Hinton y Williams[18] más tarde, en los años 80 se desarrolló el algoritmo de *backpropagation* o retropropagación. Mediante este algoritmo, iban adaptando los pesos propagando los errores hacia atrás, es decir, propagando los errores hacia las capas ocultas. De esta forma se consiguieron trabajar con múltiples capas y con funciones de activación no lineales.

Algoritmo de retropropagación

La propagación hacia atrás de errores es básicamente un descenso de gradiente. Matemáticamente hablando, la retropropagación es:

$$w_{actualizado} = w_{anterior} - \eta \nabla E \tag{3.14}$$

donde w es el peso, η es la tasa de aprendizaje (para simplificar, se puede considerar 1 por ahora) y E es la función de costo que mide el rendimiento general. También podríamos escribirlo en notación de ciencias de la computación como una regla que asigna a w un nuevo valor:

$$w \leftarrow w - \eta \nabla E \tag{3.15}$$

Esto se lee como "el nuevo valor de w es w menos $\eta \nabla E$". Esto no es circular, ya que está formulado como una asignación ($\leftarrow$), no como una definición ($=$ o $:=$). Esto significa que primero calculamos el lado derecho y luego asignamos a w este nuevo valor. Observe que si escribiéramos esto matemáticamente, tendríamos una definición recursiva.

Quizás nos preguntemos si podríamos hacer el aprendizaje de pesos de una manera más simple, sin usar derivadas y descenso de gradientes. Podríamos probar el siguiente enfoque: seleccionar un peso w y modificarlo un poco y ver si eso ayuda. Si es así, nos quedaríamos con la modificación. Si empeora las cosas, cambiaríamos en la dirección opuesta (es decir, en lugar de sumar la pequeña cantidad del peso, restar). Si esto lo hace mejor, nos quedaríamos con este cambio. Si ninguno de los cambios mejora el resultado final, podemos concluir que w es perfecto tal como es y pasar al siguiente peso v.

Inmediatamente surgen tres problemas. Primero, el proceso lleva mucho tiempo. Después del cambio de peso, necesitamos procesar al menos un par de ejemplos de entrenamiento para cada peso para ver si es mejor o peor que antes. Simplemente hablando, esta es una pesadilla computacional. En segundo lugar, al cambiar los pesos individualmente, nunca descubriremos si una combinación de ellos funcionaría mejor. El primero de estos problemas se soluciona mediante el descenso de gradientes, mientras que el segundo se resuelve solo parcialmente. Este problema generalmente se denomina óptimos locales.

El tercer problema es que cerca del final del aprendizaje, los cambios tendrán que ser pequeños y es posible que el "pequeño cambio" de nuestra prueba de

algoritmo sea demasiado grande para aprender con éxito. La propagación hacia atrás también tiene este problema, y generalmente se resuelve utilizando una tasa de aprendizaje dinámica que se reduce a medida que avanza el aprendizaje.

Si formalizamos este enfoque obtendremos un método llamado aproximación en diferencias finitas:

1. Cada peso w_i, $1 \geq i \geq k$ se ajusta agregándole una pequeña constante ϵ y se evalúa el error general (con solo w_i cambiado), el cual se denotará por E_i^+.
2. Se vuelve a obtener el peso inicial w_i y se resta ϵ. Reevaluando el error se consigue E_i^-.
3. Se repite el proceso para todos los pesos $w_j, \geq j \geq l$
4. Finalmente, los nuevos pesos se establecerán en $w \leftarrow w - \frac{E_i^+ - E_i^-}{2\epsilon}$

La aproximación en diferencias finitas hace un buen trabajo al aproximar el gradiente, y no se usa más que aritmética elemental. Este método se puede utilizar para desarrollar la intuición de cómo se desarrolla el aprendizaje del peso en la propagación hacia atrás. Sin embargo, la mayoría de las bibliotecas actuales que tienen herramientas para la diferenciación automática realizan un descenso del gradiente en una fracción del tiempo que tomaría calcular la aproximación en diferencias finitas.

Ahora, pasemos a la retropropagación. Examinemos lo que sucede en la capa oculta de la red neuronal *feedforward*. Comenzamos con pesos y sesgos inicializados aleatoriamente, los multiplicamos con las entradas, los sumamos y los llevamos a través de la regresión logística que los "aplana" a un valor entre 0 y 1, y lo hacemos una vez más. Al final, obtenemos un valor entre 0 y 1 de la neurona logística en la capa de salida. Podemos decir que todo lo que esté por encima de 0.5 es 1 y por debajo de 0. Pero el problema es que si la red da 0.21 y la salida debería haber sido 1, solo conocemos el error que produjo la red (la función E), y deberíamos utilizar eso. Más precisamente, queremos medir cómo cambia E cuando cambia el w_i, lo que significa que queremos encontrar la derivada de E con respecto a las actividades de la capa oculta. Queremos encontrar todas las derivadas al mismo tiempo, y para ello usamos notaciones vectoriales y matriciales y, en consecuencia, el gradiente. Una vez que tengamos las derivadas de E con respecto a la actividad de la capa oculta, calcularemos fácilmente los cambios para los mismos pesos.

Entonces, este algoritmo consta de dos pasadas: una primera hacia adelante y otra hacia atrás después:

- En la pasada hacia adelante la información del vector *imput* $\boldsymbol{x} = (x_1, x_2, ..., x_n)$ se propaga hacia adelante mediante las ecuaciones 3.12 y 3.13 dando como resultado un vector de salida $\hat{\boldsymbol{y}}_{\boldsymbol{C}} = (\hat{y}_1, \hat{y}_2 ... \hat{y}_{n_C})$, representando C la capa de salida y, por lo tanto, n_C el número de neuronas en ella.

- En la pasada hacia atrás, el error se va propagando desde la capa de salida hacia la capa de entrada, ajustando los pesos de las conexiones con el objetivo de minimizar el error E.

Sin perdida de generalidad supongamos que la función de error esta definida como el error cuadrático medio. Así pues, el error en la observación n vendrá dado de la siguiente manera

$$E_n = \frac{1}{2}\sum_{i=1}^{n_C}\left(\hat{y}_i(n) - y_i(n)\right)^2 \tag{3.16}$$

Y por lo tanto el error global medio será:

$$E = \frac{1}{N}\sum_{n}^{N} E_n \tag{3.17}$$

Siendo N el número de observaciones.

Entonces, la actualización de los pesos tendrá la siguiente expresión general:

$$\boldsymbol{w}(n) = \boldsymbol{w}(n-1) - \eta\nabla E(\boldsymbol{w}(n-1)) \tag{3.18}$$

donde $\boldsymbol{w}(n)$ es el vector de pesos en la iteración n, siendo $\boldsymbol{w}(0)$ el vector de pesos inicial. Y de esta expresión se deduce el criterio de parada:

$$\Delta w(n) = w(n) - w(n-1) = -\eta\frac{\partial E}{\partial w}[n] \tag{3.19}$$

siendo w el peso a estimar y $\frac{\partial E}{\partial w}[n]$ la derivada parcial de E en la iteración n. Hasta cumplir este criterio, la actualización de pesos se hace de forma iterada, aunque como veremos más tarde este criterio se suele establecer en un número fijo de épocas realizadas.

3.2.3. Entrenamiento y sus posibilidades

Habiendo asentado ya una base de lo que son las redes neuronales y en qué se basa su entrenamiento, estamos preparados para profundizar un poco más en su funcionamiento. Hasta ahora, hemos visto que en el momento de definir una red neuronal y su entrenamiento existen una gran variedad de posibilidades y hiperparámetros que deberemos de fijar: función de pérdida, número de neuronas, número de capas, configuración de la red, tasa de aprendizaje, funciones de activación... En este apartado profundizaremos en algunos de estos aspectos además de comentar los problemas que pueden surgir y como evitarlos.

No obstante, antes de continuar explicaremos a que se le denomina ser un hiperparámetro. El nombre es bastante inusual, pero en realidad hay una razón simple detrás de él. Cada red neuronal es en realidad una función que asigna a

un vector de entrada dado (entrada) una etiqueta de clase (salida). La forma en que la red neuronal hace esto es a través de las operaciones que realiza y los parámetros que se le dan. Las operaciones incluyen funciones de activación, la multiplicación de matrices..., mientras que los parámetros son todos los números que no son de entrada que están por aprender: pesos y sesgos. Sabemos que los sesgos son simplemente pesos y que la red neuronal encuentra un buen conjunto de pesos propagando hacia atrás los errores que registra. Dado que las operaciones son siempre las mismas, esto significa que todo el aprendizaje realizado por una red neuronal es en realidad una búsqueda de un buen conjunto de pesos, o en otras palabras, es simplemente ajustar sus parámetros. Ahora que esto está claro, es fácil decir qué es un hiperparámetro. Un hiperparámetro es cualquier número utilizado en la red neuronal que no puede ser aprendido por la red. Esto es, el aprendizaje no puede ajustar los hiperparámetros y deben ajustarse manualmente; no existe una forma científica de hacerlo, es más una cuestión de intuición y experiencia. Un ejemplo sería la tasa de aprendizaje o la cantidad de neuronas en la capa oculta.

Tipos de entrenamiento

Según el momento en el que se actualicen los pesos existen tres estrategias de entrenamientos: Estrategia *off-line*, estrategia *on-line* y estrategia por *minibatch*. Sin embargo, antes de comentar las diferentes estrategias de entrenamiento por las que se puede optar, conviene explicar estos dos conceptos:

- Lote: También llamado *batch*, el lote es el conjunto de muestras a través de las cuales se trabaja antes de actualizar los parámetros internos del modelo. Se puede pensar en un lote como un bucle "for" que itera sobre una o más muestras y hace predicciones. Al final del lote, las predicciones se comparan con las variables de salida esperadas y se calcula un error. A partir de este error, se actualizan los valores de los pesos. Esto permite acelerar la convergencia en contraposición a si los pesos tuviesen que actualizarse tras pasar por todos los datos de entrenamiento.
- Época: Las épocas representan el número de veces que cada muestra del conjunto de datos de entrenamiento ha tenido la oportunidad de actualizar los parámetros internos del modelo. Normalmente se elige un número elevado de épocas pero esto, además de conllevar un gran coste computacional, a veces también conlleva un sobre-entrenamiento del modelo.

En consecuencia, se nos suman dos nuevos parámetros que deberemos fijar para realizar el entrenamiento de nuestra red. Ahora que se han explicado estos conceptos básicos, podemos continuar con la clasificación de tipos de entrenamiento:

- Estrategia *off-line*: También conocida como estrategia por lote o estrategia por *batch*, es un tipo de entrenamiento en el que los pesos se actualizan

después de haber pasado todo el conjunto de datos por la red. Es decir, solo se toma un único lote, el conjunto de datos de entrenamiento entero, y con él se minimiza directamente la función de pérdida E. A pesar de que teóricamente hablando esta estrategia sería la correcta, en grandes conjuntos de datos su convergencia es lenta y requiere mucho coste computacional. Es por ello que este método resulta útil únicamente en conjuntos de datos relativamente pequeños.

- Estrategia *on-line*: Al contrario que la anterior, esta estrategia actualiza los pesos con el procesado de cada muestra del conjunto de entrenamiento. Es decir, el tamaño de lote es igual a 1. Es por ello que esta estrategia pueda aplicarse en conjunto de datos grandes. No obstante, este método no es compatible con la programación en paralelo ya que es necesario el resultado obtenido después de procesar una observación antes de procesar la siguiente.

- Estrategia por *minibatch*: Es una estrategia híbrida que mantiene las ventajas de sus predecesoras. Este método divide el conjunto de datos de entrenamiento en pequeños lotes del mismo tamaño o *minibatches*, para ir actualizando los pesos con cada procesado de uno de ellos. Con ello, logramos que el entrenamiento sea factible para conjuntos de datos grandes sin perder la posibilidad de poder usar programación en paralelo.

Función de perdida

Como bien hemos indicado previamente todo el entrenamiento se realiza con el objetivo final de minimizar esta función. Su correcta elección nos permitirá describir manera más acertada el error entre los vectores de salida estimados, $\hat{\boldsymbol{y}}$, y los vectores de salida esperados, $\boldsymbol{y}$. Debido a que existen incontables maneras de medir este error, en este trabajo solamente se explicará la función de la que se va a hacer uso: Entropía cruzada categórica.

La entropía cruzada categórica es una función de pérdida que se utiliza en tareas de clasificación de clases múltiples. Estas son tareas en las que un ejemplo solo puede pertenecer a una de las muchas categorías posibles, y el modelo debe decidir cuál. La función de pérdida de entropía cruzada categórica calcula la pérdida de un ejemplo calculando la siguiente suma:

$$E(\hat{\boldsymbol{y}}, \boldsymbol{y}) = -\sum_{i=1}^{n_C} y_i \log(\hat{y}_i) \tag{3.20}$$

siendo n_C, el número de neuronas en la capa final o, lo que es lo mismo, el número de categorías de las que predecir la probabilidad de permanencia.

Esta función es una muy buena medida de cuán distinguibles son dos distribuciones de probabilidad discretas entre sí. En este contexto, $\hat{y}_i$ es la probabilidad estimada de que ocurra el evento i y la suma de todos $\hat{y}_i$ es 1, lo

que significa que puede ocurrir exactamente un evento. El signo menos asegura que la pérdida se reduce cuando las distribuciones se acercan entre sí.

Softmax es la única función de activación recomendada para usar con la función de pérdida de entropía cruzada categórica. Estrictamente hablando, la salida del modelo solo necesita ser positiva para que el logaritmo de cada valor de salida $\hat{y}_i$ exista. Sin embargo, el principal atractivo de esta función de pérdida es comparar dos distribuciones de probabilidad. La activación de softmax cambia la escala de la salida del modelo para que tenga las propiedades correctas.

Número de neuronas y número de capas

La elección del número de neuronas normalmente suele hacerse de manera experimental ya que no existen evidencias teóricas que respalden ciertamente una metodología para su elección. Lo mismo sucede con el número de capas. Configurar la arquitectura de la red sin duda es un paso importante a la hora de definirla. Un número demasiado bajo de estas cantidades generará modelos que no aprenden lo suficiente de los datos de entrenamiento (*underfitting*). Elevar el número de estos hiperparámetros siempre tendrá como consecuencia modelos más complejos y costosos computacionalmente, y además esta acción no siempre traerá mejores resultados. La elevación excesiva del número de neuronas de la capa oculta, resultará en modelos sobreentrenados (*overfitting*) que se ajustan bien al conjunto de datos de entrenamiento pero carecen de generalidad en su aplicación. Existen algunas reglas que en la práctica han resultado ser eficientes, como por ejemplo la regla de capa oculta-capa de entrada [1], o la regla de la pirámide [11]. Esta última, se basa en la suposición de que el número de neuronas de la capa oculta, debe ser menor que el número de neuronas de la capa de entrada pero mayor que el número de neuronas de la capa de salida.

En la práctica, como bien he comentado antes, esta elección suele hacerse a prueba y error, intentando conseguir un equilibrio entre el coste computacional del modelo y su eficacia.

Tasa de aprendizaje

La idea de incluir una tasa de aprendizaje se propuso explícitamente por primera vez en el año 2000 [20]. Como hemos visto anteriormente, la tasa de aprendizaje controla la cantidad de actualización que queremos, ya que la tasa de aprendizaje es parte de la regla general de actualización de peso, es decir, entra en juego al final de la propagación hacia atrás. Antes de pasar a los tipos de tasa de aprendizaje, exploremos por qué la tasa de aprendizaje es importante en un entorno abstracto.

Construyamos un ejemplo abstracto de aprendizaje imaginando que tenemos una canica que representa nuestro modelo y una curva en forma de cuenco redondo y profundo, cuya altura representa el error del modelo. El gradiente es como la gravedad e intenta minimizar la función de pérdida o el error pero queremos que sea diferente a la gravedad física para que la cantidad de movimiento en esta dirección no esté determinada por la posición exacta del mínimo para la altura, es decir, no se asiente en el fondo sino que puede moverse al otro lado del cuenco. Dejamos la cantidad de movimiento sin especificar en este momento, pero asumimos que rara vez es la cantidad exacta necesaria para alcanzar el mínimo real: a veces es un poco más y se sobrepasa, y otras es un poco menos y no lo alcanza. Aquí hay que señalar un punto muy importante: la curvatura "apunta" al mínimo, pero estamos siguiendo la curvatura en el punto en el que nos encontramos actualmente, y no en el mínimo. En cierto sentido, la canica es extremadamente "miope": solo ve la curvatura actual y se mueve a lo largo de ella. Sabremos que hemos encontrado el mínimo cuando tengamos la curvatura de 0. Tenga en cuenta que en nuestro ejemplo tenemos un çuenco idealizado", que tiene solo un punto donde la curvatura es 0, y ese es el mínimo global para y. Imagínese cuántas superficies más complejas podría haber donde no podemos decir que el punto de curvatura 0 es el mínimo global, pero también tengamos en cuenta que si pudiéramos tener una transformación que transforme cualquiera de estas superficies complejas en nuestro cuenco, tendríamos un aprendizaje algoritmo perfecto.

Cada época de aprendizaje es un movimiento (de alguna cantidad) en la "dirección general" de la curvatura del cuenco, y una vez hecho, se queda donde está. La segunda época "descongela" la situación, y nuevamente se sigue la dirección general hacia la curvatura. Este segundo movimiento podría ser la continuación del primero, o un movimiento en una dirección casi opuesta si la canica sobrepasa el mínimo (la base del cuenco). El proceso puede continuar indefinidamente, pero después de varias épocas, los movimientos serán realmente pequeños e insignificantes, por lo que podemos detenernos después de un número predeterminado de épocas o cuando la mejora no sea significativa.

La tasa de aprendizaje controla la cantidad de movimiento que vamos a realizar. Una tasa de aprendizaje de 1 significa hacer todo el movimiento, y una tasa de aprendizaje de 0,1 significa hacer solo el 10 % del movimiento. Como se mencionó anteriormente, podemos tener una tasa de aprendizaje global o una tasa de aprendizaje dinámica para que cambie según ciertas condiciones que especifiquemos.

Volviendo al ejemplo de la canica, imaginemos que ahora tenemos un cuenco poco profundo con forma de elipse alargada. Si dejamos caer la canica sobre la parte del borde más lejana al centro, se moverá a lo largo de una curvatura muy poco profunda y tomará una gran cantidad de épocas encontrar su camino hacia el fondo del cuenco. La tasa de aprendizaje puede ayudar aquí. Si tomamos solo una fracción del movimiento, la dirección de la curvatura para el siguiente movimiento será considerablemente mejor que si nos movemos de un borde de un cuenco alargado y poco profundo al borde opuesto. Dará pasos más pequeños pero encontrará una

buena dirección mucho más rápidamente (ver figura 3.6).

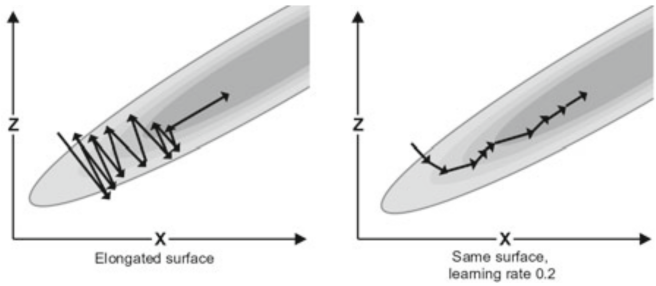


Figura 3.6: Ejemplo ilustrativo de la canica.

Esto nos abre un debate sobre el valor de la tasa de aprendizaje η. Una tasa de aprendizaje pequeña puede conllevar a una trayectoria más "suave", pero también a un aprendizaje más lento. Por otro lado, una tasa de aprendizaje alta lleva a un entrenamiento más rápido pero puede ocasionar oscilaciones en superficies del error de alta curvatura o incluso puede provocar el estancamiento en un mínimo local y no global. Los valores que se utilizan con más frecuencia suelen estar en el intervalo $[0, 1]$ como $0{,}1, 0{,}01, 0{,}001...$ Valores como 0,03 simplemente se perderán y se comportarán de manera muy similar al logaritmo más cercano, que en este caso es 0,01.

Como sucede con otros hiperparametros, para esta cantidad tampoco existen reglas que ayuden fijar su valor, y por eso suele fijarse mediante pruebas de ensayo-error. No obstante para nuestro caso no será necesario realizar dichas pruebas, ya que se hará uso de una tasa de aprendizaje adaptativa con el método de *Rmsprop*, el cual mas tarde explicaremos.

Momento

Dirigimos nuestra atención ahora a una idea similar a la tasa de aprendizaje, pero diferente llamada momento o inercia. Hablando informalmente, la tasa de aprendizaje controla qué parte del movimiento se debe mantener en el paso actual, mientras que el momento controla qué parte del movimiento del paso anterior se debe mantener en el paso actual. El problema que la inercia intenta resolver es el problema de los mínimos locales. Volvamos a nuestra idea con el cuenco pero ahora modifiquemos el cuenco para que tenga mínimos locales. La canica cae como de costumbre hacia el mínimo y continúa a lo largo de la curvatura, y se detiene cuando la curvatura es 0. Pero el problema es que la curvatura 0 no es necesariamente el mínimo global, es solo local. Si fuera un sistema físico, la canica tendría impulso y caería por encima del mínimo local a un mínimo global, allí iría y oscilaría un poco para luego asentarse. El impulso en las redes neuronales es solo la formalización de esta idea. Entre los distintos momentos que se le puede agregar a la ecuación 3.18, el más intuitivo es el momento estandar, que deja la expresión

original tal que así:

$$\boldsymbol{w}(n) = \boldsymbol{w}(n-1) - \eta \nabla E(\boldsymbol{w}(n-1)) + \mu \boldsymbol{\Delta w}(n-1) \tag{3.21}$$

donde $\mu > 0$ sería el correspondiente término del momento. Con esto se consigue acelerar el descenso en direcciones similares en iteraciones consecutivas (gradientes consistentes) y, estabilizar si se tienen oscilaciones de signos en varias iteraciones consecutivas. Por tanto, se evitan oscilaciones (inestabilidad) en "valles" de la superficie de error a la par que acelera la convergencia en regiones con poca pendiente. Es por eso que se dice que esta idea de actualización de los pesos se basa en el gradiente para modificar la "velocidad" del vector de pesos en vez de su "posición", como hace el algoritmo original.

Regularización

Uno de los aspectos más importantes a la hora de entrenar redes neuronales es evitar el sobreentrenamiento. La regularización se refiere a un conjunto de diferentes técnicas que reducen la complejidad de un modelo de red neuronal durante el entrenamiento y, por lo tanto, evitan el este sobreajuste. Entre las técnicas más populares de regularización se encuentran las siguientes:

- Regularización L_2: La regularización L_2 es el tipo más común de todas las técnicas de regularización y también se conoce comúnmente en inglés como *weights decay*. La idea de este método es utilizar la norma euclidiana para el término de regularización. Con esto, la función de pérdida L_2 regularizada, $\tilde{E}$, simplemente será, la función de pérdida original definida para la tarea, E, más el término de regularización:

$$\tilde{E} = E + \lambda \left\| \boldsymbol{w} \right\|_2^2 = E + \lambda \sum_i w_i^2 \tag{3.22}$$

  siendo λ es conocido como el parámetro de regularización y es el hiperparámetro que ajusta cuanto de regularización queremos. La intuición detrás de esto es que durante el procedimiento de aprendizaje, se preferirán pesos más pequeños, pero se considerarán pesos más grandes si la disminución general del error es significativa. La elección de λ determina cuánto se preferirán los pesos pequeños (cuando λ es mayor, la preferencia por pesos pequeños es mayor).

- Regularización L_1: La regularización L_1, también conocida en inglés como *basis pursuit denoising* usa el valor absoluto en lugar de los cuadrados:

$$\tilde{E} = E + \lambda \left\| \boldsymbol{w} \right\|_1 = E + \lambda \sum_i |w_i| \tag{3.23}$$

  Para la mayoría de los problemas de clasificación y predicción, la regularización L_2 resulta mejor. Sin embargo, hay ciertas problemas en las que

la L_1 sobresale: aquellos que contienen una gran cantidad de datos irrelevantes. Estos pueden ser datos muy ruidosos o características que no son informativas, pero también pueden ser datos escasos (donde la mayoría de las características son irrelevantes porque faltan).

- *Dropout*: Sin duda otra técnica muy popular y eficaz. El *Dropout* funciona de manera distinta que las anteriores ya que no modifica la función de pérdida sino la estructura de la red. Su funcionamiento es tan simple como que consiste en la eliminación temporal de algunas de las neuronas de la red, forzando a la red a trabajar con parte de sus neuronas. Concretamente, durante el entrenamiento en cada época, cada neurona tiene una probabilidad p de ser eliminada. De esta manera las neuronas se convierten menos dependientes entre ellas.

Problema de desvanecimiento y explosión

El problema de desvanecimiento describe la situación en la que una red no puede propagar información de gradiente útil desde el extremo de salida del modelo a las capas cercanas al extremo de entrada del modelo. Cuando se hace uso de la técnica del gradiente descendente cada uno de los pesos de la red neuronal recibe una actualización proporcional a la derivada parcial de la función de error con respecto al peso actual en cada iteración del entrenamiento. El problema es que, en algunos casos, el gradiente será muy pequeño, lo que evitará que el peso cambie su valor. En el peor de los casos, esto puede impedir por completo que la red neuronal siga entrenando. Como un ejemplo de la causa del problema, las funciones de activación tradicionales, como la función sigmoide, tienen gradientes en el rango $[0, 1]$, y la retropropagación calcula los gradientes mediante la regla de la cadena. Dado que su derivada es $\sigma'(x) = \sigma(x)(1 - \sigma(x))$, para casos de x cercanos a los extremos del rango, los pesos no variarán prácticamente nada, provocando un aprendizaje lento de la red. Además, el problema empeora a medida que aumentamos el número de capas ocultas ya que el gradiente decrece exponencialmente. Esto es por causa de que en la actualización de los pesos se va requiriendo de un número mayor de derivadas de funciones de activación a medida que aumenta el número de capas.

Por otro lado puede ocurrir lo contrario. Cuando el gradiente toma valores muy altos, los pesos se actualizan muy rápidamente provocando que el aprendizaje que experimenta el modelo sea prácticamente nulo. A este problema se le denomina problema de explosión.

Inicialización de los pesos

Como hemos dicho previamente, el primer paso en el entrenamiento de una red es dar unos valores de inicio a sus pesos. Valores de inicio que suelen ser aleatorios, pero que deberemos de determinar su distribución. Una distribución correcta puede

ser clave para conseguir la convergencia del algoritmo en una cantidad de tiempo razonable. Si los pesos iniciales son demasiado pequeños entonces la varianza de los datos de entrada empieza a disminuir a medida que pasan por cada capa de la red, conduciendo al problema de desvanecimiento. Por otro lado, si los pesos iniciales son demasiado grandes, entonces la varianza crece rápidamente a lo largo de la red, dando lugar al problema de explosión.

Glorot y Bengio [6], propusieron en el año 2010, una regla de inicialización que lleva el nombre del primero, *Xavier*, cuyo obejetivo era que la varianza de los datos de entrada y salida coincidieran. Para ello, propusieron inicializar la matriz de pesos W siguiendo la siguiente distribución uniforme:

$$W \sim U\left[-\frac{\sqrt{6}}{\sqrt{n_{C-1}+n_C}}, +\frac{\sqrt{6}}{\sqrt{n_{C-1}+n_C}}\right] \tag{3.24}$$

donde n_{C-1} y n_C son el número de neuronas en la capa anterior y en la actual respectivamente. Esta inicialización de los pesos consigue evitar tanto el problema de desvanecimiento como el de explosión, y aunque hay una gran variedad de técnicas de inicialización, hoy en día esta es una de las más populares.

Algoritmo de *RMSProp*

En pocas palabras, *RMSProp* utiliza una tasa de aprendizaje adaptativa en lugar de tratar la tasa de aprendizaje como un hiperparámetro. Esto significa que la tasa de aprendizaje cambia con el tiempo. Los gradientes de funciones muy complejas, como las redes neuronales, tienen una tendencia a dar problemas de desvanecimiento y explosión a medida que los datos se propagan a través de la función. *RMSProp* se desarrolló como una técnica estocástica para el aprendizaje por *minibatches*. Esta técnica se ocupa de evitar los problemas de desvanecimiento y explosión mediante el uso de un promedio exponencial. La idea básica es usar un factor de desintegración, $\rho \in (0, 1)$, y ponderar las derivadas parciales cuadradas que ocurren en anteriores actualizaciones por ρ^n. Tengamos en cuenta que esto se puede lograr fácilmente multiplicando el agregado al cuadrado actual (es decir, la estimación en ejecución) por ρ y luego sumando $(1 - \rho)$ por la derivada parcial actual (al cuadrado). La estimación en ejecución se inicializa en 0. Esto provoca algún sesgo (indeseable) en las primeras iteraciones, que desaparece a largo plazo. Por lo tanto, si A_i es el valor promediado exponencialmente del i-ésimo parámetro w_i, tenemos la siguiente forma de actualizar A_i:

$$A_i(n) = \rho A_i(n-1) + (1-\rho)\left(\frac{\partial E}{\partial w_i}[n]\right)^2 \tag{3.25}$$

donde n representa el número de iteración. La raíz cuadrada de este valor para cada parámetro se usa para normalizar su gradiente:

$$w_i(n) = w_i(n-1) + \frac{\alpha}{\sqrt{A_i(n)}}\frac{\partial E}{\partial w_i}[n] \tag{3.26}$$

3.3. Redes neuronales convolucionales

Las redes neuronales convolucionales están diseñadas para trabajar con entradas estructuradas en cuadrícula, que tienen fuertes dependencias espaciales en las regiones locales de la cuadrícula. El ejemplo más obvio de datos estructurados en cuadrícula es una imagen bidimensional. Este tipo de datos también exhibe dependencias espaciales, porque las ubicaciones espaciales adyacentes en una imagen a menudo tienen valores de color similares de los píxeles individuales. Una dimensión adicional captura los diferentes colores, lo que crea un volumen de entrada tridimensional. Por lo tanto, las características de una red neuronal convolucional tienen dependencias entre sí basadas en distancias espaciales. Otras formas de datos secuenciales como texto, series de tiempo y secuencias también pueden considerarse casos especiales de datos estructurados en cuadrícula con varios tipos de relaciones entre elementos adyacentes. La gran mayoría de las aplicaciones de redes neuronales convolucionales se centran en datos de imágenes, pero su uso cada vez esta más extendido en campos como la minería de texto.

Una característica definitoria importante de las redes neuronales convolucionales es una operación, que se conoce como convolución. Una operación de convolución es una operación de producto escalar entre un conjunto de pesos estructurado en cuadrícula y entradas estructuradas en cuadrícula similares extraídas de diferentes localidades espaciales en el volumen de entrada. Este tipo de operación es útil para datos con un alto nivel de ubicación espacial o de otro tipo, como datos de imágenes. Por lo tanto, las redes neuronales convolucionales se definen como redes que usan la operación convolucional en al menos una capa, aunque la mayoría de las redes neuronales convolucionales usan esta operación en múltiples capas.

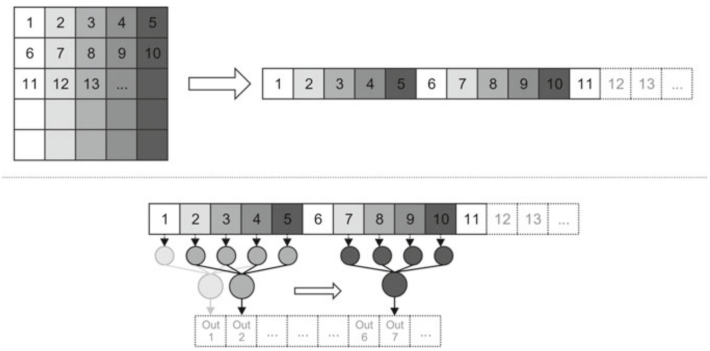


Figura 3.7: Capa convolucional 1D.

Una capa convolucional toma una matriz y una pequeña función de activación con por ejemplo tamaño de entrada 4 (estos tamaños suelen ser 4 o 9, a veces 16) y pasa la función a toda la matriz. Esto significa que la primera entrada consta de los componentes 1–4 del vector aplanado, la segunda entrada son los componentes 2–5, la tercera son los componentes 3–6, y así sucesivamente. Puede ver una descripción general del proceso en la parte inferior de la figura 3.7. Este proceso crea un vector

de salida que es más pequeño que el vector de entrada general, ya que comenzamos en el componente 1, pero tomamos cuatro componentes y producimos una sola salida. El resultado final es que si nos moviéramos a lo largo de un vector de 10 dimensiones con la función de activación (esta función se llama campo receptivo local en las redes neuronales convolucionales), produciríamos un vector de salida de 7 dimensiones (ver la parte inferior de la figura 3.7). Este tipo de capa convolucional se denomina capa convolucional 1D.

Padding

También podemos adoptar un enfoque diferente y decir que queremos que la dimensión de salida sea la misma que la de entrada, pero luego nuestro *campo receptivo local*[1] de 4 dimensiones tendría que comenzar en la entrada en "celdas" -1, 0, 1, 2 y luego continuar a 0, 1, 2, 3, y así sucesivamente, terminando en 9, 10, 11 . Poner componentes -1, 0 y 11 para que el vector de salida tenga el mismo tamaño que el vector de entrada se llama *padding* o relleno. Los componentes adicionales generalmente obtienen valores 0, pero a veces tiene sentido tomar los valores del primer y último componente de la matriz o el promedio de todos los valores. Lo importante a la hora de rellenar es pensar cómo no "engañar" a la capa convolucional para aprender las regularidades del relleno.

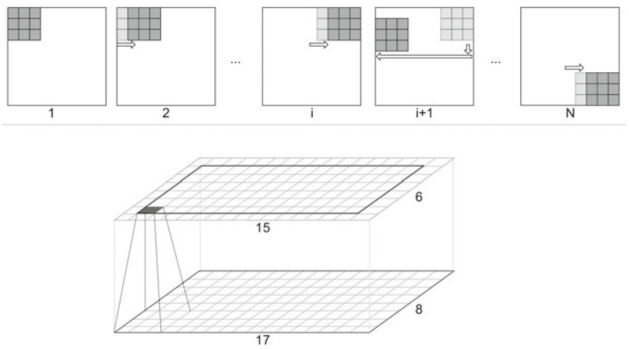


Figura 3.8: Capa convolucional 2D.

Repasemos la situación en 2D, como si no aplanamos la imagen en un vector. Esta es la configuración clásica para capas convolucionales, y dichas capas se denominan capas convolucionales 2D o capas convolucionales planas. Si usáramos 3D, lo llamaríamos espaciales, y para 4D o más hiperespaciales.

Las entradas de las funciones de activación ahora también deberían ser de dos dimensiones, y esta es la razón por la que usamos con mayor frecuencia 4,

[1]Cada neurona dentro de una CNN es responsable de una región definida de los datos de entrada, y esto permite que las neuronas aprendan patrones como líneas, bordes y pequeños detalles que componen la imagen. Esta región definida del espacio a la que está expuesta una neurona o unidad en los datos de entrada se denomina Campo Receptivo Local.

9 y 16, ya que son cuadrados de 2 por 2, 3 por 3 y 4 por 4 respectivamente. El paso ahora representa un movimiento de este cuadrado en la imagen, mirando desde la izquierda, yendo a la derecha y una vez terminado, una fila hacia abajo, moviendose completamente hacia la izquierda sin escanear y comenzando a escanear de izquierda a derecha (ver figura 3.8). Una cosa que se vuelve obvia es que ahora obtendremos menos resultados. Si usamos un campo receptivo local de 3 por 3 para escanear una imagen de 10 por 10, como resultado del campo receptivo local obtendremos una matriz de 8 por 8 (ver parte inferior de la figura 3.8.

Una red neuronal convolucional tiene múltiples capas. Imaginemos una red neuronal convolucional que consta de tres capas convolucionales y una capa completamente conectada. Supongamos que procesará una matriz cuadrada de dimensión 10 por 10 y que las tres capas tienen un campo receptivo local de 3 por 3.

La primera capa toma una matriz de 10 por 10, produce una salida (tiene pesos y sesgos inicializados aleatoriamente) de tamaño 8 por 8, que luego se le da a la segunda capa convolucional (que tiene su propio campo receptivo local con inicialización aleatoria de pesos y sesgos pero hemos decidido que sea también 3 por 3), lo que produce una salida de tamaño 6 por 6, y esto se le da a la tercera capa (que tiene un tercer campo receptivo local). Esta tercera capa convolucional produce una imagen de 4 por 4. Luego lo aplanamos a un vector de 16 dimensiones y lo pasamos a una capa estándar completamente conectada que tiene una neurona de salida con su correspondiente función no lineal. Comparando la salida de la función no lineal con la etiqueta que le corresponde, se calcula el error y se retropropaga, y esto se repite para cada *imput* en el conjunto de datos que completa el entrenamiento de la red.

En el caso de las redes convulucionales 2D el *padding* es igualmente realizable. No obstante, en este caso tendríamos un “marco” de relleno.

Mapas de características

Ahora que sabemos cómo funciona una red neuronal convolucional, podemos usar un truco. Recordemos que una capa convolucional escanea por ejemplo una imagen de 10 por 10 con una con un campo receptivo local de 3 por 3 (9 pesos, 1 sesgo) y crea una nueva matriz de 8 por 8 como salida. Imaginemos también que la imagen tiene tres canales para colores. ¿Cómo procesarías una imagen con tres canales? Una respuesta natural es correr sobre el mismo campo receptivo (que tiene pesos y sesgos entrenables pero inicializados aleatoriamente). Esta es una buena estrategia. Pero, ¿qué pasa si lo invertimos y en lugar de usar un campo receptivo local en tres canales, queremos usar cinco campos receptivos locales en un canal? Recordemos que un campo receptivo local se define por su tamaño y por sus pesos y sesgos. La idea aquí es mantener el mismo tamaño pero inicializar los otros campos receptivos con diferentes pesos y sesgos. Esto significa que cuando

escanean una imagen de 3 canales de 10 por 10, construirán 15 imágenes de salida de 8 por 8. Estas imágenes se denominan mapas de características. Es como tener una imagen de 8 por 8 con 15 canales. Esto es muy útil ya que solo un mapa de características que aprende una buena representación (por ejemplo, ojos y narices en imágenes de perros) aumentará considerablemente la precisión general de la red (suponga que la tarea de toda la red es clasificar imágenes de perros y varias objetos que no son perros (es decir, detectar un perro en una imagen)).

Una de las ideas principales aquí es que una imagen (o matriz) de 3 canales de 10 x 10 se convierte en una matriz de 15 canales de 8 x 8. La imagen de entrada se transformó en un objeto más pequeño pero más profundo, y esto sucederá en cada capa convolucional. Reducir el tamaño de la matriz significa empaquetar la información en una representación más compacta (pero más profunda).

Max Pooling

En nuestra búsqueda de la compacidad, podemos agregar una nueva capa antes o después de una capa convolucional. Esta nueva capa se denomina capa de agrupación máxima, o *max pooling* en inglés. La capa de agrupación máxima toma un tamaño de grupo como hiperparámetro, generalmente 2 por 2. Luego procesa su matriz de entrada de la siguiente manera: divide la matriz en áreas de 2 por 2, como una cuadrícula, y toma de cada agrupación de cuatro elementos el elemento con el valor máximo. Finalmente se compone estos elementos en una nueva matriz, con el mismo orden que la matriz original. Una capa de agrupación máxima de 2 por 2 produce una matriz que tiene una cuarta parte del tamaño de la matriz original. Por supuesto, en lugar del máximo, se puede idear una selección o creación de píxeles diferente, como el promedio de los cuatro píxeles o el mínimo, pero generalmente se utiliza el máximo.

La idea detrás de la agrupación máxima es que la información importante en una matriz rara vez está contenida en píxeles adyacentes (esto explica la parte 'elegir uno de cuatro'. Nótese que esta es una suposición muy fuerte que puede no ser válida en general.

Por lo general, una red neuronal convolucional se compone de una capa convolucional seguida de una capa de agrupación máxima, seguida de una capa convolucional, y así sucesivamente. A medida que la matriz pasa por la red, después de varias capas, obtenemos una matriz pequeña con muchos canales. Luego, podemos aplanar esto a un vector y usar una una función de activación al final para extraer qué partes son relevantes para nuestra tarea. La función de activación seleccionará qué partes de la representación se utilizarán para la clasificación y creará un resultado que se comparará con el objetivo para luego propagar el error hacia atrás. Esto forma una red neuronal convolucional completa.

3.4. Redes neuronales recurrentes

Todas las redes neuronales que hemos visto hasta ahora tienen conexiones que propagan la información hacia adelante, y es por eso que las hemos llamado "redes neuronales de *feedforward*". Resulta que al tener conexiones que realimentan la salida de nuevo en una capa como entradas, podemos procesar secuencias de longitud desigual. Esto hace que la red sea profunda, pero comparte pesos, por lo que evita en parte el problema del *vanishing gradient*. Las redes que tienen tales bucles de retroalimentación se denominan redes neuronales recurrentes. En la historia de las redes neuronales recurrentes, hay un giro interesante. Tan pronto como la idea del perceptrón no pareció buena, la idea de hacer un "perceptrón de múltiples capas" pareció natural. Recordemos que esta idea era teórica y anterior a la retropropagación (que fue ampliamente aceptada después de 1986), lo que significa que nadie pudo hacerla funcionar en ese entonces. Entre las ideas teóricas exploradas se encontraba agregar una sola capa, agregar varias capas y agregar ciclos de retroalimentación, que son todas ideas naturales y simples. Esto fue antes de 1986. Dado que la retropropagación aún no estaba disponible, J. J. Hopfield introdujo la idea de las redes Hopfield [8], que pueden considerarse las primeras redes neuronales recurrentes exitosas. Estas redes eran diferentes de lo que hoy consideramos redes neuronales recurrentes. Hoy en día las redes neuronales recurrentes más importantes son las redes de memoria a corto-largo plazo o LSTM, que fueron inventadas en 1997 por Hochreiter y Schmidhuber [7]. Son las redes neuronales recurrentes más utilizadas y responsables de muchos resultados de vanguardia en varios campos, desde el reconocimiento de voz hasta la traducción automática. En esta sección, nos centraremos en desarrollar los conceptos necesarios para explicar las LSTM en detalle.

Como se comentado previamente las RNN, o redes neuronales recurrentes, permiten que las salidas de las neuronas sean también entradas de neuronas de capas anteriores o incluso de neuronas de su propia capa. Estos bucles son los que permiten que la información persista, consiguiendo en la red esa capacidad de "memoria temporal". A continuación, en la figura 3.9, se observa un ejemplo de una red neuronal recurrente, donde la salida en el instante (o observación) t de la capa oculta sirve como entrada del siguiente instante (o observación) $t + 1$:

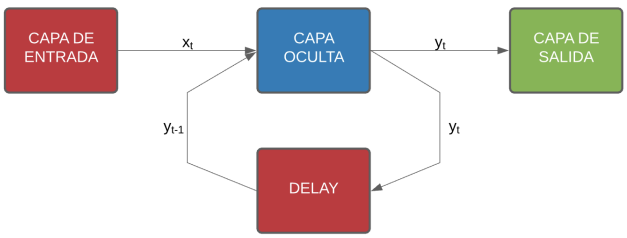


Figura 3.9: Ilustración de una red neuronal recurrente simple.

Con el objetivo de hacer ver que una red neuronal recurrente no se aleja

tanto como parece de una *feedforward*, es habitual mostrarla como un perceptrón multicapa en paralelo:

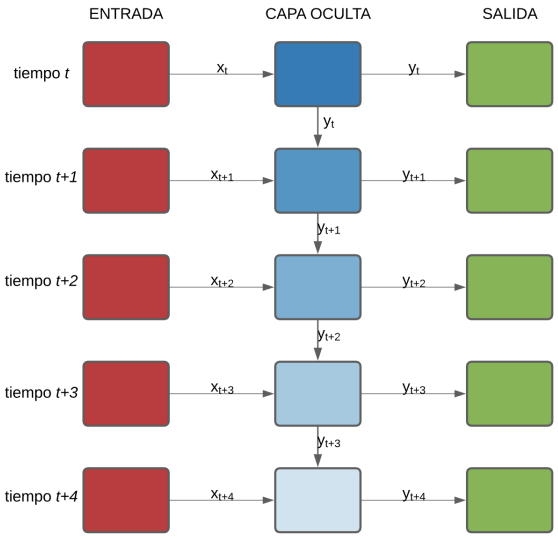


Figura 3.10: Red neuronal recurrente en formato similar al perceptrón multicapa.

Como obserbamos en la figura 3.9, la única diferencia que existe entre esta red y el perceptrón multicapa es la necesidad de agregar una nueva variable, t, que representa el "tiempo". Con esto, la salida de capa oculta en la observación t, y_t, queda determinada de la siguiente manera:

$$y_t = \sigma(Ux_t + Wy_{t-1}), \quad t = 1, ...n \tag{3.27}$$

siendo n el número de observaciones y, U y W las matrices de pesos de la entrada x_t y la salida de la capa oculta del instante anterior, y_{t-1} (donde $y_0 = 0$).

Por otra parte, el descenso en la tonalidad de la capa oculta de la figura 3.10, nos ilustra la perdida de información con el paso del tiempo. Es decir, la capacidad de memoria de las redes recurrentes convencionales esta limitada y no son capaces de aprender dependencias a largo plazo. La sensibilidad disminuye a medida que se introducen nuevos *output*, sobreescribiendo las activaciones de la capa oculta, y por consiguiente "olvidando" así las primeras entradas. Con el objetivo de evitar esta pérdida de información y permitir las dependencias a largo plazo surgieron las redes recurrentes de memoria a largo y corto plazo, cuyas siglas en ingles son LSTM (*long short-term memory*)

LSTM

El remplazo de los nodos de la capa oculta por lo que se denominan celdas de memoria es la novedad que presentaron estas redes recurrentes. Estas unidades

son las que dotan a la red de esa memoria a largo plazo de la que carecen las redes recurrentes convencionales. Estas celdas de memoria cuentan con una especie de "sistema de compuertas" que lo que hacen es decidir que información debe guardarse en la memoria actual, cual debe olvidarse y que información se debe transmitir al resto de capas. Así pues se consideran trés puertas: puerta de entrada, puerta de olvido y puerta de salida. Y para $t \geq 1$ estas puertas tendrán las siguientes expresiones respectivamente:

$$\begin{aligned} i_t &= \sigma(U^i x_t + W^i y_{t-1}) \\ f_t &= \sigma(U^f x_t + W^f y_{t-1}) \\ o_t &= \sigma(U^o x_t + W^o y_{t-1}) \end{aligned} \tag{3.28}$$

donde U, W son las matrices de pesos que deberemos estimar, x_t es el vector de entrada de la capa en el tiempo t, y_{t-1} es el vector de salida de la capa en el instante $t-1$ y σ representa a la función de activación, que normalmente suele ser la función sigmoide. La puerta de entrada es la que decide qué información al estado oculto deja pasar, la puerta de olvido es la que decide qué parte de la "memoria" guarda y la de salida es la que decide qué información deja salir de la capa oculta. Este mecanismo de compuertas permite que la información perdure en el tiempo, dotando así a la red de "memoria". El comportamiento general de una capa oculta LSTM quedará determinado además de por las expresiones 3.28, por las expresiones que vienen:

$$\begin{aligned} g_t &= \text{ReLU}(U^g x_t + W^g y_{t-1}) \\ c_t &= c_{t-1} * f_t + g_t * i_t \\ y_t &= \text{ReLU}(c_t) * o_t \end{aligned} \tag{3.29}$$

siendo c_t la representación de la celda de memoria interna en el momento t (donde $c_0 = 0$), g_t la entrada al estado oculto, y_t la salida del estado oculto en el tiempo t (donde $y_0 = 0$), y U^g, W^g las matrices de pesos a estimar. Además observamos el signo "$*$" que indica la multiplicación componente a componente de vectores y la función de activación ReLU que a pesar de que podríamos haber elegido cualquier otra, es la que se utilizará.

En la imagen 3.11 podemos ver ilustrado el funcionamiento de la capa oculta de una red recurrente LSTM. En ella observamos que el sistema de compuertas 3.28 viene representado por los círculos de color rosa. Además, tenemos "g", o la posible entrada a la capa, que se determina con la primera expresión de 3.29. En el tiempo t, la entrada g_t se multiplicará elemento a elemento por la puerta de entrada i. Como la función sigmoide presenta valores en el rango entre 0 y 1, esta operación puede interpretarse como la elección de la cantidad de "información" de g_t que se deja entrar a la celda. Por otra parte, la celda de memoria interna en el instante anterior, C_{t-1}, se multiplicará elemento a elemento por la puerta de olvido f_t , generando la memoria que perdura. De igual modo que antes, esto se puede interpretar como la elección de la cantidad de "información" que se mantiene de la celda de memoria interna del instante anterior. La suma de estas dos cantidades conforma lo que será la nueva celda de memoria interna C_t, a la que se le aplica la función ReLU y se multiplica elemento a elemento por la puerta de salida o_t,

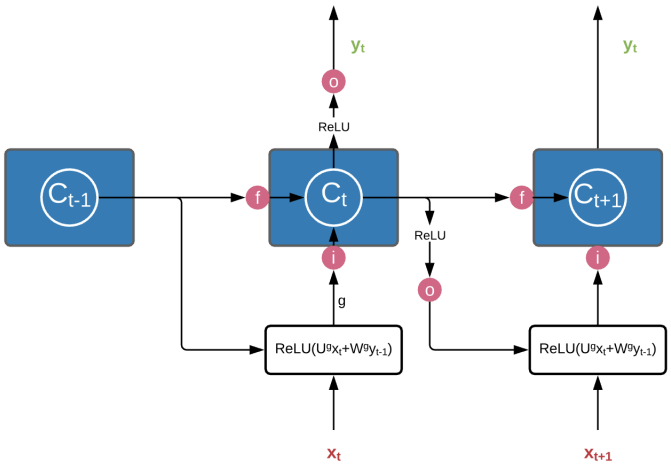


Figura 3.11: Funcionamiento de la celda de memoria interna.

obteniendo así la salida de la capa oculta en el momento t, y_t. A su vez, esta salida servirá como entrada al estado oculto del siguiente instante de tiempo.

Entrenamiento de redes recurrentes

El algoritmo que permite entrenar las redes recurrentes es conocido como algoritmo de retropropagación a través del tiempo, (BPTT, sus siglas en inglés). Este algoritmo se basa en la retropropagación original pero adaptandose a la estructura de las redes recurrentes [22].

3.5. Word2Vec

Word2Vec es una de las técnicas más populares para aprender incrustaciones de palabras utilizando una red neuronal de dos capas. Su entrada es un texto y su salida es un conjunto de vectores. Los *word embedings* a través de *Word2Vec* pueden hacer que el lenguaje natural sea legible por computadora, luego se puede usar una mayor implementación de operaciones matemáticas en palabras para detectar sus similitudes. Un conjunto de vectores de palabras bien entrenados colocará palabras similares cerca unas de otras en ese espacio.

Hay dos algoritmos de entrenamiento principales para *Word2Vec*, uno es la bolsa continua de palabras (CBOW, con sus siglas en inglés) y el otro se llama *skip-gram* (ver figura 3.12). Ambos son redes neuronales de una sola capa oculta. La principal diferencia entre estos dos métodos es que CBOW utiliza el contexto para predecir una palabra objetivo, mientras que *skip-gram* utiliza una palabra para predecir un contexto objetivo.

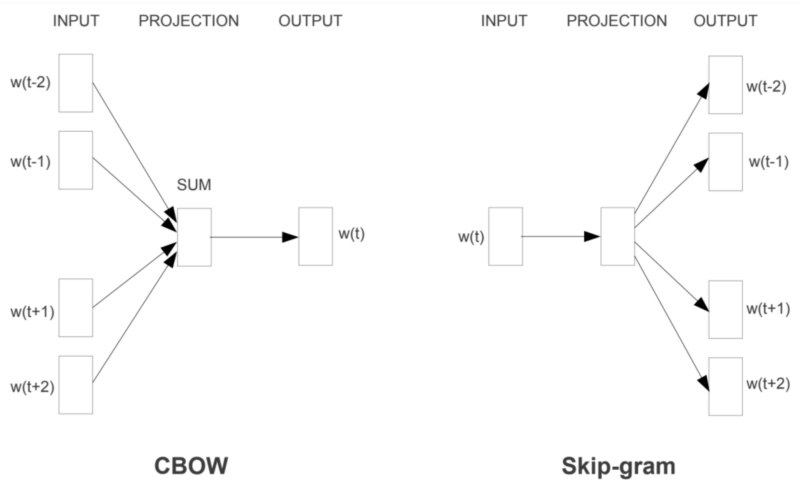


Figura 3.12: Ilustración de los modelos de *Word2Vec*

El planteamiento en ambos modelos es similar: definir una ventana simétrica (contexto) en torno a una palabra central y plantear un problema de optimización basado en predecir la palabra central dado el contexto o viceversa. Sin pérdida de generalidad, a continuación se explicará el modelo de CVOW que nos permitirá la representación de los tokens de nuestros conjuntos de datos.

CBOW

El modelo CBOW predice y genera vectores representativos estudiando el contexto de la palabra inicial. Esta técnica utiliza una codificación *one-hot* para representar numéricamente el vocabulario de longitud V. Por lo tanto, en principio, la palabra w_i vendrá representada tal que así:

$$w_i = (x_1, x_2, ..., x_i, ..., x_V) \tag{3.30}$$

donde todas los elementos cuyo índice sean diferente a i tendrán un valor nulo y $x_i = 1$.

En el caso más común de CBOW, el contexto utilizado se expande más allá de una palabra y utiliza un contexto de longitud C. El objetivo de este modelo será conseguir la probabilidad condicional de la palabra de salida w_O, basándose en las palabras de entrada $W_c = \{w'_{I1}, w'_{I2}, ..., w'_{Ic}\}$, es decir, calcular $P(w_O|w'_{I1}, w'_{I2}, ..., w'_{Ic})$. Esto se consigue mediante iteraciones dentro del modelo, basándose en las dos matrices de pesos $\boldsymbol{W}$ y $\boldsymbol{W'}$. $\boldsymbol{W}$ es la matriz que conecta la capa de entrada con la capa oculta, y $\boldsymbol{W'}$ conecta la capa de salida con la oculta. Las dos matrices tienen dimensiones similares, $\boldsymbol{W}$ tiene forma de $V \times N$ mientras que $\boldsymbol{W'}$ está definida como $N \times V$ donde N es la dimensión de la capa oculta y V es el tamaño de los *Embedding* que resultarán del modelo.

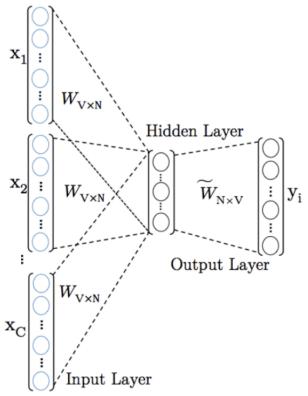


Figura 3.13: Modelo CBOW con un contexto de C palabras utilizadas como entrada en la red

Definamos $\boldsymbol{h}$ como viene:

$$\boldsymbol{h} = \boldsymbol{W}^T x = \boldsymbol{v}_{w_I}^T \tag{3.31}$$

donde se representa la proyección de la palabra de entrada sobre el nuevo vector $\boldsymbol{h}$. Si entendemos el modelo complejo de CBOW como una simple combinación de palabras de entrada, la $\boldsymbol{h}$ tomará la siguiente expresión:

$$\boldsymbol{h} = \frac{1}{C}\boldsymbol{W}^T(\boldsymbol{x}_1 + \boldsymbol{x}_2 + ... + \boldsymbol{x}_C) = \frac{1}{C}(\boldsymbol{v}_{w_1} + \boldsymbol{v}_{w_2} + ... + \boldsymbol{v}_{w_C})^T \tag{3.32}$$

siendo el vector resultante una simple combinación lineal de las palabras de entrada pertenecientes al contexto C. El contexto C se define con una ventana deslizante sobre la palabra elegida. La palabra número n, en una ventana de tamaño S tendrá como contexto las palabras definidas en el conjunto $C_{w_n} = \left\{w_{n-S/2}, ..., w_n, ..., w_{n+S/2}\right\}$.

El objetivo de $\boldsymbol{h}$ es ponderar un criterio de evaluación utilizando la información de la segunda mitad del modelo contenida en $\boldsymbol{W}'$.

$$u_j = \boldsymbol{v}_{w_j}'^T \boldsymbol{h} \tag{3.33}$$

Utilizando la función softmax, podemos aproximar nuestro objetivo inicial, la probabilidad condicional, de la siguiente manera:

$$y_j = P(w_j|w_1', w_2', ..., w_c') = \frac{e^{u_j}}{\sum_{k=1}^{V} e^{u_k}} \tag{3.34}$$

El objetivo del modelo será maximizar la probabilidad condicional combinada de todo el vocabulario. Por ello, habrá que maximizar la función objetivo o la función *log-likelyhood*, que se define de la siguiente manera:

$$\sum_{j=1}^{V} \log(P(w_j|w_{j-S/2}', ..., w_j', ..., w_{j+S/2}')) \tag{3.35}$$

La codificación continua de palabras *Word2Vec* es capaz de capturar la similitud de palabras de manera que dos palabras similares estarán cercanas en el espacio n-dimensional de palabras que se crea. Destaca por su habilidad al capturar información semántica y sintáctica de las palabras, pero siempre hay que tener en cuenta que la información útil que proporciona dependerá del contexto que tengamos.

Capítulo 4

Entrenamiento para la generación de *keywords*

4.1. Obtención del texto

Como previamente se ha comentado los datos (que en este caso vienen en forma de textos) para realizar el estudio los ha proporcionado la empresa Quarizmi. Estos datos vienen en forma de frases, que en realidad son las búsquedas que han realizado distintos usuarios de Google que han hecho por lo menos una vez "click" en anuncios de publicidad de tres clientes de la empresa Quarizmi. Dado que los usuarios que han realizado dichas búsquedas se han interesado por estos anuncios lo que se pretende es crear nuevas *keywords* o palabras clave basados en estas búsquedas, para dirigir los futuros anuncios de estos tres clientes a usuarios que estén interesados en sus productos. Para ello, intentaremos aplicar todo lo visto hasta ahora, pero antes, veamos como son los textos que tenemos.

Del primer cliente, al que nos referiremos como "Cliente A", tenemos una lista formada por 883 búsquedas y podemos encontrar búsquedas como estas:

- franquicia co aliment
- abrir franquicia supermercado
- franquicias de alimentacion sin royaliti y sin canon de entrada
- supermercados dia franquicias
- franquicias tiendas de ropa infantil
- mejores franquicias hosteleria

- franquicias de ropa moda joven
- franquyicia de administracion de fincas gratis

Observamos que son frases no muy largas, y muy específicas y concisas. Por otro lado al tratarse de búsquedas realizadas por Google, será común encontrar varias frases que contengan errores ortográficos o que simplemente no estén bien redactadas.

Del segundo cliente, denominado como "Cliente B", tenemos una lista formada por 5899 frases, en las que veremos frases del siguiente estilo:

- 805 la roche posay anthelios spray fresco viso invisibile spf50 75ml
- xls medical precio y opiniones
- gel hidroalcoholico aloe vera 5 litros
- pasta lacer 200 ml
- pilexil forte anticaida
- fotoprotector isdin hydrolotion spf 50 200 ml
- comprar vitamina c sesderma
- alcohol 96 1l madrid

Finalmente, del que llamaremos el "Cliente C" tenemos una lista de 10727 búsquedas entre las cuales están estas:

- como ser bombero en españa
- guardia civil número
- tecnico en anatomia patolo estudiar a distancia
- fp policia
- grado superior de cocina la pobla de farnals
- grado superior enfermería online
- policía nacional gijon
- precio curso doblaje barcelona

La elección de estos tres conjuntos de datos no ha sido casualidad. Esta se ha hecho con el objetivo de probar los modelos con conjuntos de datos con características distintas, ya que como podemos apreciar, el número de muestras de cada cliente y su variabilidad puede resultar muy distinto.

Cada una de estas frases podrá ser utilizada para entrenar los modelos de cada cliente, pero para ello, será necesario transformarlas previamente, representándolas numéricamente. Dado que nuestra tarea consiste en generar texto, hemos considerado que no es necesario ninguna eliminación de *stop words* ni de sustitución por lema. Es por eso que después de tokenizar ya podríamos pasar a la fase de representación numérica de los textos. Como se ha mencionado previamente, se han realizado dos tipos de tokenización: por caracteres y por palabras. De esta manera, trabajamos con dos tipos de tareas según la generación de texto: la que consiste en ir generando texto carácter a carácter y la que consiste en ir generando texto por palabras. Una vez tokenizado deberemos de representar cada token numéricamente, pero como veremos a continuación dependiendo del tokenizado que se haga, se utilizarán distintas técnicas de representación numérica del texto.

4.2. Generación de texto por caracteres

Dentro de los modelos de generación de texto por caracteres que hemos considerado, se encuentran el modelo de red LSTM y el modelo de red convolucional.

4.2.1. Modelo de red LSTM

El modelo trabajado en este apartado está basado en un modelo de generación de haikus de Jeremu Neyman [15]. Este modelo utiliza una red neuronal LSTM que se entrena con haikus (poemas tradicionales japoneses de estructura silábica de 5-7-5) para posteriormente generar nuevos poemas caracter a caracter. Sin embargo tiene un pequeño giro: El número de sílabas de cada línea se proporciona a la red, se pasa a través de una capa densa (con función de activación ReLU) y luego se agrega al estado interno de LSTM, como se ve en la figura 4.1. La idea de nuestro modelo es hacer lo mismo pero con el número de caracteres por *keywords* creadas. De este modo, seríamos capaces de mas o menos controlar el tamaño de *Keywords* creadas; y digo mas o menos por que esto no será una condición restrictiva sino una tendencia a que se cumpla dicha condición. Nuestro modelo se completa con una capa densa con función de activación de softmax que lo que nos dará es un vector de probabilidades que indique la probabilidad de cada carácter de ser el siguiente.

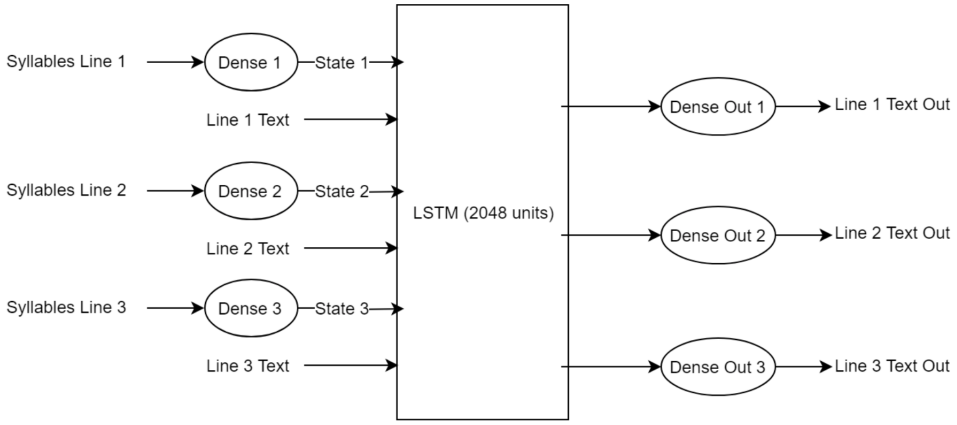


Figura 4.1: Diagrama de alto nivel del modelo generador de haikus[15]

Antes de proceder con el entrenamiento del modelo, sin embargo, recordemos que deberemos representar numéricamente los conjuntos de datos que ya hemos tokenizado. Como anticipábamos en el capítulo 2, la representación numérica más adecuada para la tokenización por caracteres es la codificación *one-hot*. Una vez transformado cada carácter en un vector de *one-hot*, se obtiene un conjunto de conjuntos de vectores (o una matriz) en el que cada conjunto de vectores (o fila) representa a una frase. Un inconveniente que surge llegado a este punto es que cada frase contiene un número de caracteres diferentes y a la hora de introducirlos como datos de entrada de la red no se puede tener entradas de distintas longitudes. Para solucionar esto, simplemente se rellenan las frases con espacios vacíos para llegar a la longitud máxima de caracteres de las frases del conjunto de datos del correspondiente cliente. Una vez preparado por completo el conjunto de datos, se dividen los datos que se usarán como datos de entrenamiento de los que se usarán como datos de validación. En nuestro caso se ha optado por el 90 % de todo el conjunto de datos para entrenamiento y 10 % para validación. Es necesario comentar también, que durante el entrenamiento los datos de salida de la red con los que se retropropagará el error son los propios datos de entrada, es decir, las búsquedas codificadas en *one-hot*.

El siguiente paso es determinar los hiperparámetros. Aunque la mayor parte de la configuración de la red que se ha utilizado, se haya justificado y determinado en el capítulo 3, volvamos a recordarlo:

- La estretegia por la que se ha optado es la de estrategia por minibatch por sus ventajas respecto a las demás.
- La función de pérdida que mejor se adapta al problema es la de la entropía categórica cruzada.
- La inicialización de pesos no puede ser otra que la de Xavier uniforme.
- El algoritmo de optimización que se ha escogido es el Rmsprop, con su valor por defecto $\rho = 0{,}95$.

Por otra parte queda por especificar el valor de la tasa de aprendizaje, η, el tamaño de los *minibatches*, el número de celdas de memoria y el número de épocas. Ya sabemos que la elección de los hiperparámetros a veces no es más que cuestión de prueba-error. Por eso, en el caso de la primera, en principio se ha tomado el valor por defecto que ofrece Keras, $\eta = 0{,}1$, y si en el entrenamiento surgiese algún inconveniente que pudiese estar relacionado con esto se probaría a cambiarlo. Con el tamaño de los minilotes se seguirá un planteamiento similar: a priori, se ha fijado en 64 (basándose en [15]) pero posteriormente se podría cambiar. El número de épocas y celdas de memoria, en cambio, merecen un estudio más profundo, ya que estos influyen más en la calidad del modelo. Para ello, se ha realizado un barrido con ambos hiperparámetros, en el que se estudia la influencia de cada hiperparámetro.

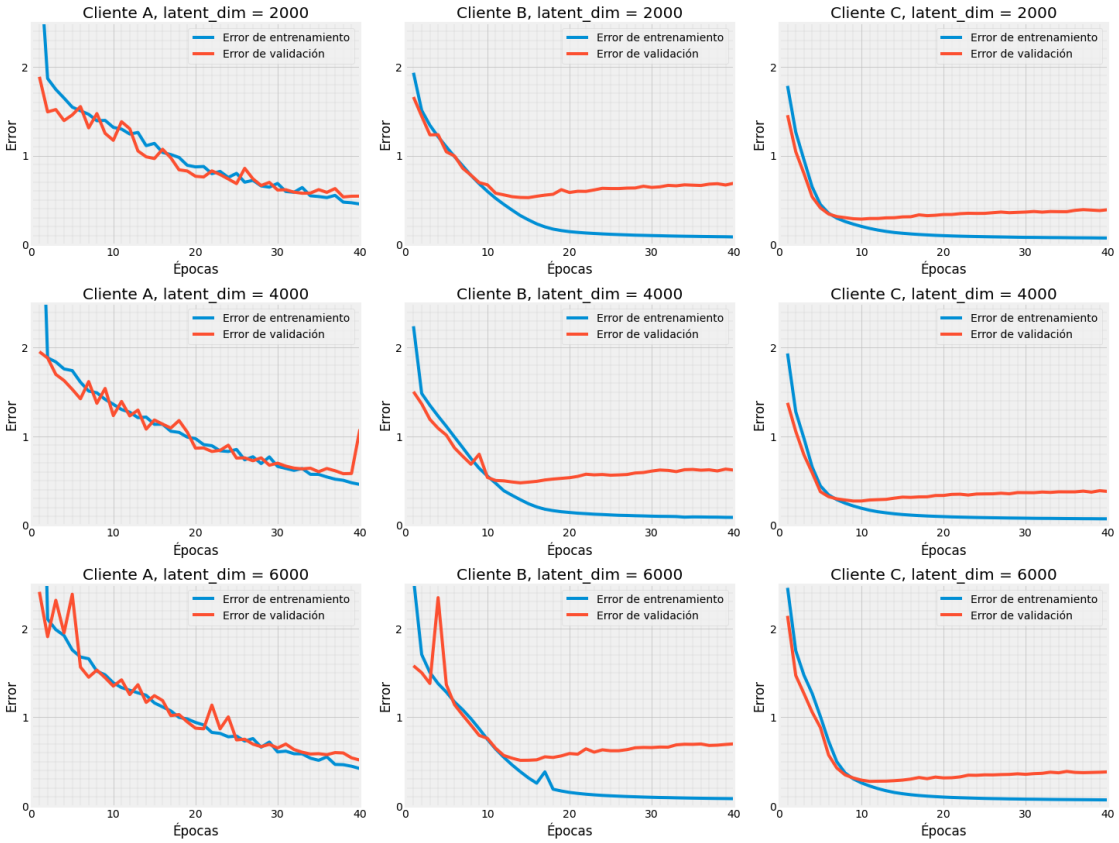


Figura 4.2: Error de distintos modelos con LSTM por caracteres según cliente, número de celdas de memoria (denominado latent-dim) y épocas.

En la figura 4.2 observamos los errores (calculados con la entropía categórica cruzada) de cada modelo. El número de épocas ideal para cada caso será aquel que minimice el error de validación, ya que cuando el error de validación empieza a incrementarse mientras el de entrenamiento sigue decreciendo se está dando un sobre-ajuste del modelo. Con un vistazo general puede llamar la atención la lenta convergencia de los ejemplos del Cliente A en comparación con el resto. Sin embargo, recordemos que este cliente era con mucha diferencia el que menos datos proporcionaba. Por lo que es lógico pensar que necesitará más épocas para ir

reduciendo el error. Si se observa bien, este fenómeno sucede también en los otros dos clientes: Cuantos más datos se tenga, más rápida será la convergencia. Por otra parte, resulta curioso que el número de celdas de memoria (denominado por latent-dim) no tenga apenas influencia en el error. Cuanto mayor sea esta cantidad mayor es la complejidad del modelo, por lo que si la mejora no es apreciable, el modelo más simple será mejor.

Para validar el entrenamiento podemos hacer uso de la precisión o *accuracy* que se muestra en la figura 4.3. En estos gráficos observaremos con que porcentaje dará el modelo el máximo de probabilidad de salida al siguiente token; o dicho de otra forma, el token con más probabilidad que se predice, con que porcentaje coincidirá con el que de verdad sea el siguiente token. A primera vista se puede observar la fuerte relación de los gráficos de la precisión con los de el error. Si nos fijamos bien, veremos que cuando uno sube el otro tiende a bajar y viceversa. Por ello, las conclusiones que se pueden obtener de estas figuras son similares. En primer lugar, podríamos pensar que a los modelos del cliente A, no les basta con 40 épocas para llegar a la convergencia y podríamos intuir que tanto la precisión del entrenamiento, como la de validación, tenderán a incrementarse elevando el número de épocas. Además, nos damos cuenta que efectivamente el subir la cantidad de celdas de memoria mas allá de las 2000, únicamente nos producirá modelos más complejos que no darán mejores resultados. En cuanto a la precisión de cada modelo, podemos estar satisfechos con los resultados obtenidos, ya que para los modelos óptimos de los clientes B y C en cuanto a error, se obtienen *accuracys* de validación superiores al 0,85.

Finalmente analicemos los tiempos de computación de cada ejecución. Como los tiempos de ejecución entre de cada época prácticamente no varían (es decir, el tiempo de ejecución es linealmente proporcional a la cantidad de épocas), solamente se analizarán los tiempos de entrenamiento de todas las épocas por cliente y cantidad de celdas de memoria. En la tabla 4.1 se observa que cuanto más se eleve la complejidad del modelo (subiendo la cantidad de unidades de memoria) mas evidente es la ralentización del entrenamiento. Por otro lado, parece que el número de datos utilizado afecta aproximadamente de manera lineal a la duración del entrenamiento.

número de celdas	Cliente A	Cliente B	Cliente C
2000	14 min	95 min	212 min
4000	46 min	288 min	655 min
6000	99 min	621 min	1379 min

Tabla 4.1: Tiempos de computación de distintos modelos con LSTM por caracteres según cliente y número de celdas de memoria.

Teniendo todo lo que hemos comentado en cuenta, como mejor opción para un modelo general que se adapte lo mejor posible a cada uno de los clientes analizados,

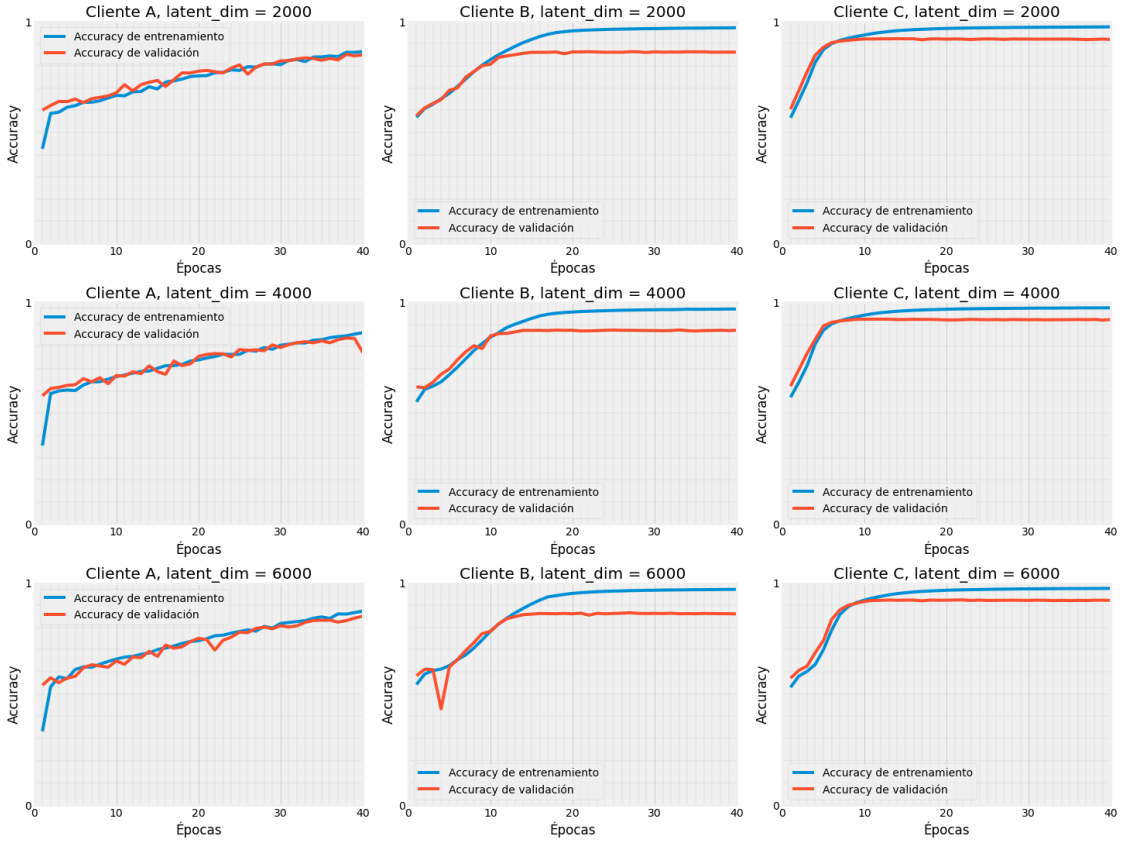


Figura 4.3: Precisión de distintos modelos con LSTM por caracteres según cliente, número de celdas de memoria (denominado latent-dim) y épocas.

parece un modelo con 2000 celdas de memoria entrenado durante 10 épocas con los hiperparámetros que hemos definido. Esta definición es la que mejor resultados propone por lo menos para los clientes B y C, y en cuanto a error, *accuracy* y tiempo de entrenamiento se refiere. Sin embargo, si quisiéramos ser más específicos para cada cliente, para el A tendríamos que seguir entrenando durante más épocas. Por eso, puede verse que un segundo barrido de entrenamientos durante 60 épocas se ha realizado en las figuras 4.4 y 4.5. En estas figuras podemos concluir que el modelo más óptimo para este cliente es el que tiene 2000 unidades de celdas de memoria y además es suficiente con entrenarlo durante 40 épocas.

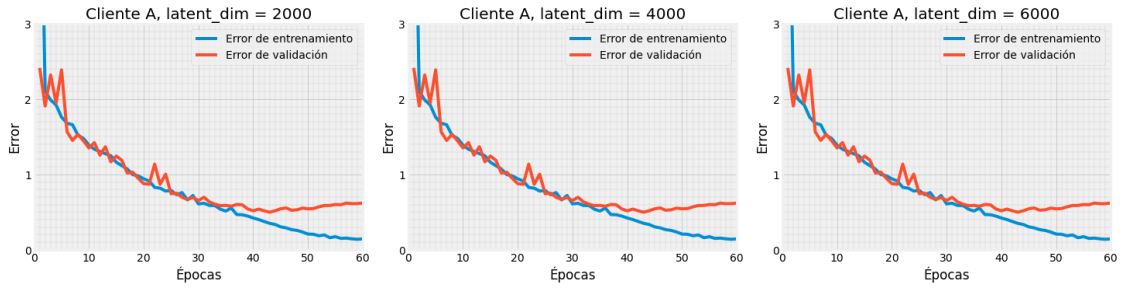


Figura 4.4: Error de distintos modelos con LSTM por caracteres para el cliente A según número de celdas de memoria y épocas.

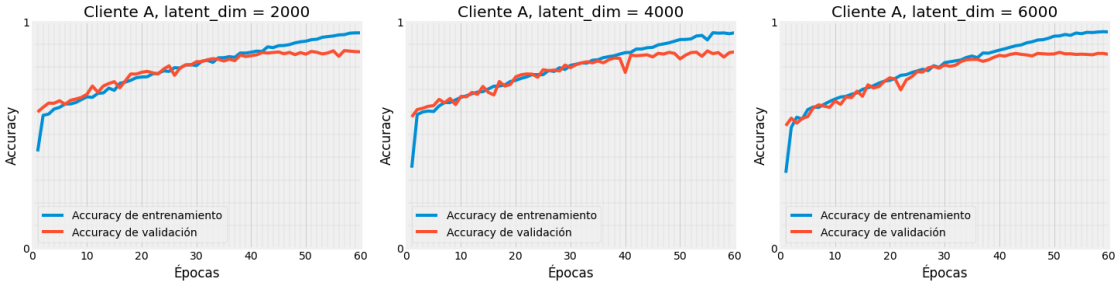


Figura 4.5: Precisión de distintos modelos con LSTM por caracteres para el cliente A según número de celdas de memoria y épocas.

Aunque no se haya comentado previamente, hasta ahora y durante todo el trabajo se elegirán modelos entrenados durante un número de épocas que sea múltiplo de diez. La razón para esta elección es simple: A pesar de que podamos entrenar durante un número bastante elevado de épocas, guardar los modelos entrenados con sus correspondientes parámetros e hiperparámetros ocupa espacio en la memoria, más de 500 Mb por cada modelo en este caso. Como nos podemos imaginar es inviable guardar todos los modelos y por ello se eligió guardar los modelos cada diez épocas de entrenamiento. Esto nos limitará a la hora de elegir la mejor opción pero también veremos que no habrá mucha diferencia entre el modelo elegido y el que pueda considerarse más óptimo.

Generación de *keywords*

Habiendo elegido ya los modelos que se creen más apropiados es hora de empezar con la generación de nuevas posibles búsquedas que los usuarios que quieran los productos de estos clientes puedan efectuar. Para la generación de nuevas *keywords* nos basaremos en el generador del artículo [15]. Para empezar con la generación de texto aleatoriamente propondremos un carácter inicial, que codificandolo en *one-hot* servirá como entrada para la red. De esta obtendremos un vector de probabilidades que indique la probabilidad de cada carácter de ser el siguiente, y en función de esa distribución se seleccionará aleatoriamente un carácter, que será el siguiente. En esta selección influirá el parámetro de temperatura, T, que se utilizará para cambiar las probabilidades y controlar la posibilidad de que caracteres con baja probabilidad tiendan a escogerse más veces. Así, la probabilidad de que el token (en este caso carácter) número i, $\tilde{p}_i$, sea elegido como el siguiente tendrá esta expresión:

$$\tilde{p}_i = \frac{e^{\frac{\ln p_i}{T}}}{\sum_j^n e^{\frac{\ln p_j}{T}}} \tag{4.1}$$

donde p_j es el probabilidad del token número j según la red neuronal y n es el número de tokens. De esta manera, al aumentar T, hacemos que la tendencia de

selección para carácteres con probabilidad baja aumente.

Después de elegir un carácter el proceso se repetirá tomando como entrada, los caracteres de entrada de la anterior selección y el propio carácter, creando token a token una nueva oración. Esta secuencia se dará por terminada cuando por el prefijado de la longitud del nuevo texto, la generación de nuevos caracteres se detenga.

Explicado ya el funcionamiento de la generación, comencemos generando palabras clave para el cliente A, que recordemos que el modelo por el que hemos optado para este caso es el de 2000 unidades de memoria entrenado durante 40 épocas. Con $T = 1$ (equivalente a tomar las probabilidades de salida de la red) se pueden obtener los resultados de la tabla 4.2. Vemos que estos no son del todo malos ya que en general, aunque tienda a haber fallos en algunas letras, se entiende lo que se quiere decir. No obstante, las frases resultan bastante repetitivas sobre todo porque palabras como "varieadades", "supermercado" o "franquicias" aparecen constantemente. Este es el conjunto de datos con menos clientes y por lo tanto esto tendrá bastante influencia a la hora ver lo que se ha generado. Además, las frases del conjunto de datos también resultaban repetitivas hasta cierto punto.

veriadades franquicia de supermercados
franquicias de taererica de supermercado dia
tienda de franquicia de supermercado dia
franquicia mida
vicies de franquicias de erecicion de coches
franquicia de panaderia
abrie franquicia de paquetería
veriadades de cafeterara franquicia
franquicia de alimentacion

Tabla 4.2: Generación de *keywords* por caracteres para el cliente A

Usando la misma temperatura que en la anterior pero ahora con un modelo de latent-dim= 2000 entrenado durante 10 épocas generaremos frases del estilo de los de la tabla 4.3. Estas frases en general parecen bastante coherentes, para encontrar fallos de escritura hay que fijarse bien y no son tan repetitivas como las generadas para el cliente A. El conjunto de este cliente, ofrece frases que son en sí mismas muy específicas, con nombres de productos y términos muy concretos y por ello los textos generados tenderán a serlo también. Por eso analizándolas una a una veremos que algunas de ellas tienden a ser demasiado específicas hasta el punto de que pueden perder algo de coherencia.

Para acabar con este apartado analizaremos los resultados con $T = 1$, para el modelo del cliente C con 2000 celdas de memoria y 10 épocas. Estos textos generados se pueden leer en la tabla 4.4. Se trata de oraciones con pocos fallos

y total coherencia, que además son muy específicos pero variados. Por lo tanto son resultados bastante buenos. Parece ser que al aumentar el número de datos de entrenamiento la calidad de los textos generados mejora, o por lo menos con los conjuntos de textos que hemos analizado.

xls medical forte 5 precio más
sesderma sesretinal mature skin serum pack
bimanan plus q quemagrasas opiniones
opiniones de decol forte
bimanan komplett chocolate crujiente precio
gel hidroalcoholico de manos de manos dicora
carrofart cellage firming cream reafirmante opiniones
la roche posay rosaliac parche de magnesio comprar onal
comprar alcohol 70 grados en san sebastián de los reyes

Tabla 4.3: Generación de *keywords* por caracteres para el cliente B

curso para niños de cocina en la semana blanca en madrid para estudiar
que hay que estudiar para ser auxiliar de enfermeria en calle colombia
hay grado medio de nutricion y dietetica
universidades en tarragona dónde cursar un grado de enfermeria
academia oposiciones susta de correos
cursos de cocina española en madrid españa
institutos grado medio auxiliar de enfermeria
oposiciones auxiliar administrativo bizkaia
academia oposiciones secundaria asturias
grado medio de estética
sueldo academia guardia civil 2020

Tabla 4.4: Generación de *keywords* por caracteres para el cliente B

4.2.2. Modelo de red convolucional

El siguiente modelo esta basado en un modelo de generación de memes de Dylan Wenzlau [21]. En este artículo vemos un ejemplo de uso de redes neuronales convolucionales para la generación de texto. Para nuestro caso sin embargo, la arquitectura de la red variará un poco: Primero se repetirá una sucesión de Capa convolucional, *Max Pooling* de 2×2 y un *Dropout* de 0.25 (como en el artículo [21]) un total de tres veces. La salida de esta secuencia será entrada de una nueva

capa convolucional donde su *output*, después de aplanarlo, servirá de entrada para una capa densa con función de activación Softmax que dará la salida de la red.

En este caso la entrada a la red no puede ser de la misma forma que la entrada al modelo con LSTM. Para el modelo convolucional la idea es que este intente predecir cual será el carácter que viene a continuación, recibiendo como información de entrada una parte de una frase. Esto se ve mejor con el siguiente ejemplo:

Frase	Entrada	Salida
fp bombero	'f'	'p'
fp bombero	'fp'	' '
fp bombero	'fp '	'b'
fp bombero	'fp b'	'o'
fp bombero	'fp bo'	'm'
fp bombero	'fp bom'	'b'
fp bombero	'fp bomb'	'e'
fp bombero	'fp bombe'	'r'
fp bombero	'fp bomber'	'o'

Tabla 4.5: Ejemplo de estrategia de entrenamiento de nuestro modelo

Aunque la estrategia *input-output* de la red sea distinta, no lo es la forma de representar numéricamente los textos. Al igual que antes, los caracteres se transformarán con la codificiación *one-hot*, dando como resultado una representación matricial para cada frase de entrada y un vector para cada carácter de salida. Al igual que antes, es necesario que todas las entradas y salidas de la red tengan las mismas dimensiones, por lo que se tienen que rellenar las frases con espacios vacíos hasta alcanzar la longitud máxima de caracteres de las frases del conjunto de datos. Con todo esto, como es lógico, el número de frases con el que contábamos para entrenar el modelo aumenta significativamente, y resulta en 26484, 147590, 369642 número de datos de entrada y salida para los clientes A, B y C respectivamente. Una vez obtenidos estos datos ya transformados, ya podemos dividirlos en los conjuntos de datos de entrenamiento y validación, con un 90 % y 10 % respectivamente, siendo esta la proporción usada en el primer modelo.

La configuración del entrenamiento, sus hiperparámetros y la justificación de estos sigue siendo similar al modelo de LSTM:

- La estrategia por la que se ha optado es la de estrategia por minibatch por sus ventajas respecto a las demás.
- La función de pérdida que mejor se adapta al problema es la de la entropía categórica cruzada.

- La inicialización de pesos no puede ser otra que la de Xavier uniforme.
- El algoritmo de optimización que se ha escogido es el Rmsprop, con su valor por defecto $\rho = 0{,}95$.
- En principio la tasa de aprendizaje que ofrece Keras, $\eta = 0{,}1$ debería ir bien.
- El tamaño de los lotes elegido es de 64.
- La mejor opción para las funciones de activación de la capa oculta sin duda es la de la ReLU.

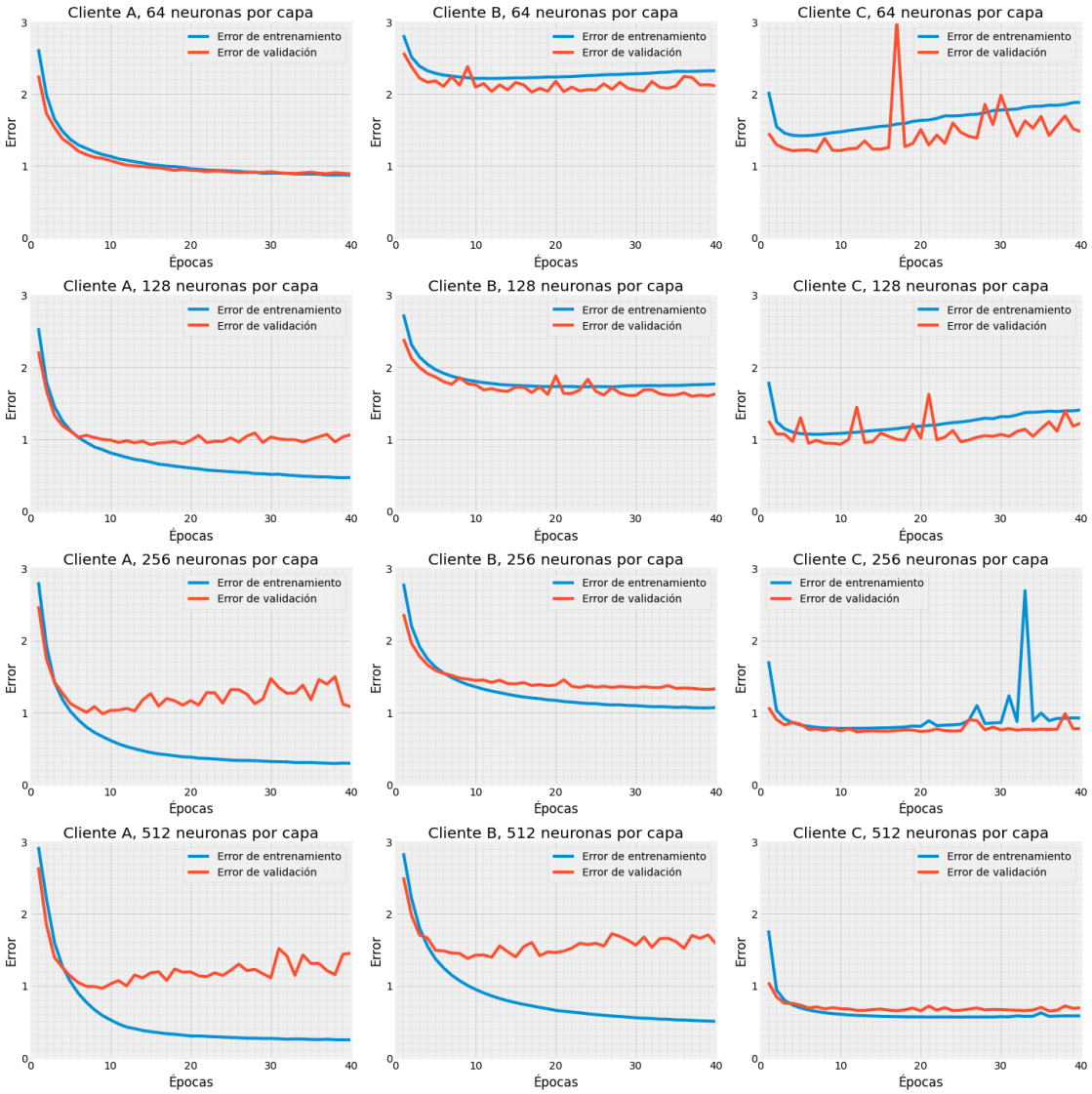


Figura 4.6: Error de distintos modelos convolucionales por caracteres según cliente, número de neuronas por capa y épocas.

En la figura 4.6 podemos ver el comportamiento del error durante los distintos entrenamientos. Del mismo modo que se hace en el modelo [21], aquí también se ha escogido un número de neuronas por capa igual para todas las capas. Para analizar

distintos resultados y encontrar el modelo óptimo se ha hecho un barrido con distintos números de neuronas por capa y épocas por cada cliente. Debido a que se trata con modelos formulados de distinto modo no se puede comparar directamente el error del modelo LSTM con el error del modelo convolucional. El análisis de error es útil para determinar si el entrenamiento se ha realizado correctamente y para saber cuando se empieza a sobre-entrenar el modelo. Relacionado con esto, un fenómeno curioso que se aprecia en el cliente C es el incremento del valor de la función de pérdida y algunos picos que surgen. Podríamos pensar que el algoritmo de retropropagación este funcionando mal pero en este caso el incremento del error sucede por la tasa de aprendizaje. Para evitar este fenómeno deberíamos probar con un valor inferior de η. De esta manera evitaríamos el incremento del error de entrenamiento aunque el modelo más óptimo no cambiaría demasiado.

Por lo demás parece parece que el entrenamiento se ha realizado correctamente. A diferencia de antes, al añadir mas complejidad a la red (aumentando el número de neuronas) se observa que el error de entrenamiento decrece más. Sin embargo, parece que esto tampoco hace mejorar mucho la validación, que es lo importante ya que al final nos quedaremos con el modelo donde la validación sea mejor.

Por otro lado en la figura 4.7 tenemos la precisión de los distintos modelos. La precisión es la forma de medir con que porcentaje acertará el modelo cual será el siguiente carácter. Estos gráficos parece que no hacen mas que reafirmar lo que comentábamos con los gráficos del error. Sin embargo, si nos fijamos bien, apreciaremos que la precisión si que mejora aumentando el número de neuronas en los clientes B y C, aunque no en el A. El mayor impacto de la cantidad de neuronas por capa se observa en el cliente B pero en los dos últimos podemos concluir que es suficiente con 256 neuronas por capa, ya que con 512 el *accuracy* de validación no aumenta significativamente. Para finalizar, en relación con los gráficos del valor de la función de pérdida, se puede apreciar que en el momento que un modelo empieza a sobre-ajustarse su curva de validación de precisión deja de aumentar, por lo que en cierto modo el aumento de precisión y el descenso del error van de la mano.

neuronas por capa	Cliente A	Cliente B	Cliente C
64	11 min	61 min	224 min
128	33 min	220 min	710 min
256	99 min	652 min	2196 min
512	253 min	1661 min	4939 min

Tabla 4.6: Tiempos de computación de distintos modelos según cliente y cantidad de neuronas por capa.

Los tiempos de computación de cada ejecución se pueden ver en la tabla 4.6. Obsérvese que al doblar el numero de neuronas por capa al principio, aproximadamente hace ralentizar la ejecución en un tercio. No obstante, se intuye que esta relación va perdiendo fuerza cada vez que se dobla el número de neuronas.

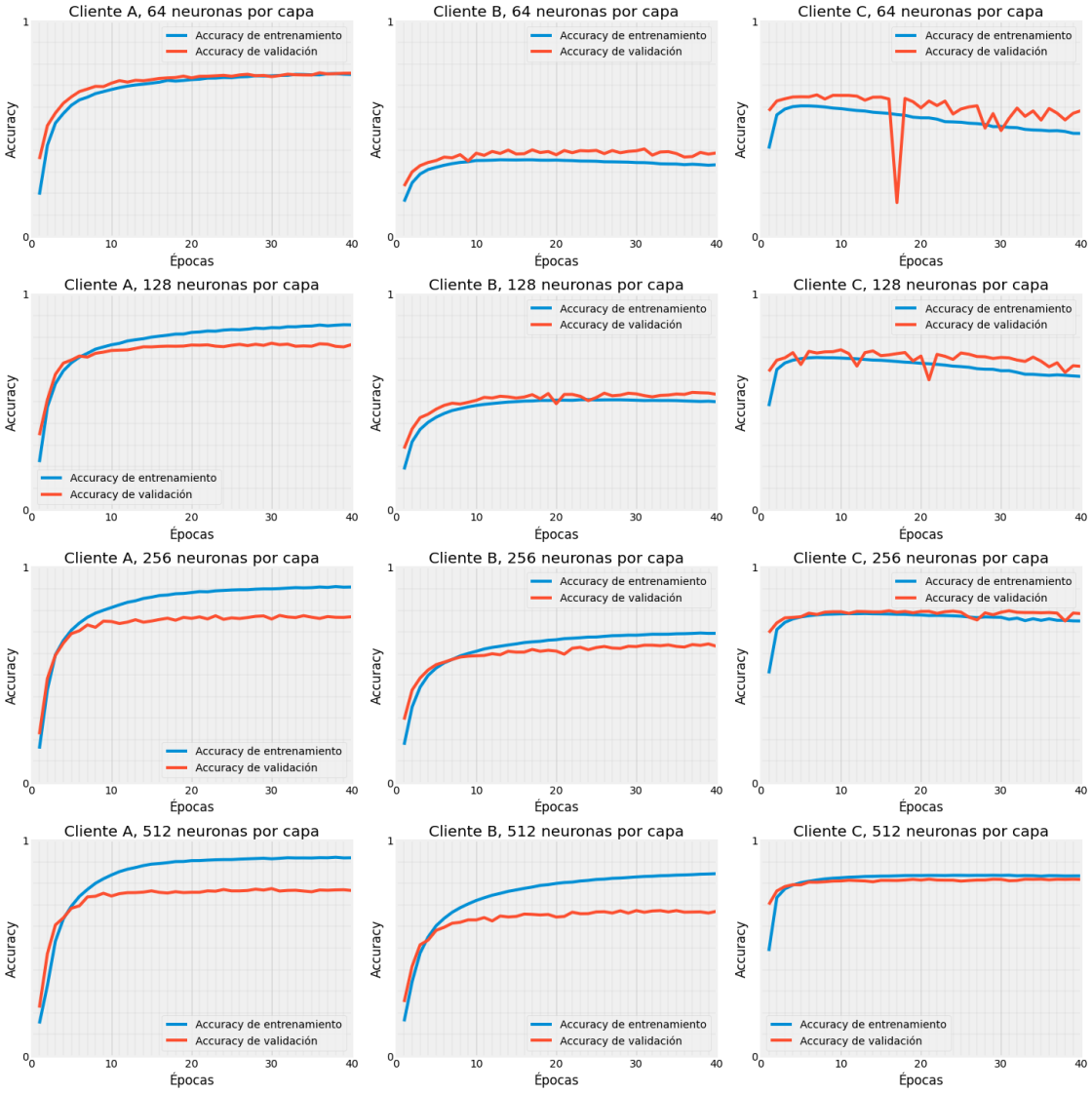


Figura 4.7: Precisión de distintos modelos convolucionales por caracteres según cliente, número de neuronas por capa y épocas.

Por contrario, el aumento de neuronas parece que más efecto tiene en el aumento del tiempo de ejecución cuantos más datos se tengan. En cuanto a la comparación del tiempo con el anterior modelo, es fácil ver que las filas de la tabla 4.1 son comparables con las tres primeras filas de la tabla 4.6.

Teniendo todo esto en cuenta, si se tuviese que elegir un modelo de red convolucional general que se adaptase lo mejor posible a los tres clientes, uno con 256 neuronas por capa entrenado durante 10 épocas con los hiperparámetros que hemos definido podría ser buena opción. Dicho modelo, es el que mejor resultados ofrece para los clientes B y C por lo menos si evaluamos en cuanto a error, precisión y tiempo de entrenamiento. Si pretendiéramos ser más específicos con cada cliente, quizá al primer cliente le convendría más un modelo con menos neuronas por capa como por ejemplo uno de 128 o incluso de 64, ya que complicar más el modelo solo

aumentaría la complejidad y el tiempo de ejecución.

Generación de *keywords*

Después de elegir los mejores modelos, es hora de ponerlos a prueba en la generación de *keywords*. Para generar nuevas búsquedas clave se seguirá el mismo principio que el seguido para el caso LSTM: Aleatoriamente se propondrá un carácter inicial que servirá como primera entrada para la red. De esta obtendremos un vector de probabilidades que indique la probabilidad de cada carácter de ser el siguiente, y en función de esa distribución se seleccionará aleatoriamente un carácter. Al igual que en el caso anterior, tendremos el parámetro de temperatura, T, con el que podremos controlar la tendencia a escoger los caracteres con probabilidades mas bajas. De este modo obtenemos una lista de dos caracteres que servirá como entrada para predecir el siguiente carácter y repetir este proceso. A diferencia de antes, la longitud de la frase creada se controlará de manera más artificial cortando el proceso de generación una vez acabada la palabra donde se ha alcanzado el número de caracteres que se había prefijado.

Habiendo ya explicado como funciona la generación de texto veamos algunos ejemplos. En primer lugar, generaremos algunos *keywords* para el cliente A, con $T = 1$ y haciendo uso del modelo de 128 neuronas por capa entrenado durante 20 épocas, veremos textos del siguiente estilo:

quieres de paquet
supermercado franquicias de supermercados
ntar franquicias de panaderia cafeteria
hay franquicias de supermercados de
l as mejor franquicias
supermercado franquicias
erenquicias de
berianas de supermercados pequeños
x franquicias de supermerca

Tabla 4.7: Generación de *keywords* por caracteres para el cliente A

El hecho de contar con un *accuracy* de validación de 0,76 hace que a priori el modelo no parezca malo del todo. Sin embargo, viendo los resultados que ofrece en la tabla 4.7 nos damos cuenta de que no son tan buenos. Las palabras que se forman en su mayoría se forman sin errores de escritura, aunque de vez en cuando vemos oraciones sin ningún sentido, pero el conjunto de palabras tiende a no mostrar información relevante. Es más, se aprecia un exceso de repetición de las palabras "supermercado" y "franquicias". Esto puede deberse a la cantidad de frases de

entrenamiento y a su calidad, ya que como hemos visto en los ejemplos de este cliente ambas palabras se repiten más de una vez. Además es interesante observar como algunas oraciones parecen inacabadas (las que acaban en "de") debido a que la forma de terminarlas es más artificial que en el modelo de LSTM.

Con un $T = 1$, el modelo de 256 número de neuronas por capa entrenado durante 10 épocas, los resultados para el cliente B serán similares a los que tenemos en la tabla 4.8. A pesar de tener una precisión validada claramente inferior de 0,59, se aprecian resultados mucho mejores que en el caso de A. No obstante, algunas frases parecen carecer de sentido por ejemplo al concretar dos volúmenes distintos. Por otra parte, es necesario comentar que para esta temperatura se ha observado que uno de cada diez frases generadas, se encuentran en el conjunto de datos de entrenamiento. Para evitar esto e intentar generar frases que no se hayan usado para entrenar se puede elevar el parámetro de temperatura.

alcohol 96 1 litro comprar 200 ml
redoxon vitamina
omega vitamina crema productor gel 30 precion
mejores cremas protectoras
heliocare 360 plus flude 50 ml
comprar alcohol
redoxon vitamina c 360 capsulas preci
dercos de pelo frtaalecturico
gel hidroalcoholico 1 litro de 120ml
tetinas dr brown nivel 25 550 ml

Tabla 4.8: Generación de *keywords* por caracteres para el cliente B

Finalmente, hagamos lo propio con el cliente C. El modelo de 256 neuronas por capa entrenado durante 10 épocas generará textos del tipo de la tabla 4.9. Para este caso se repite que una décima parte de las frases generadas son frases que han sido utilizadas para el entrenamiento. Recordemos que este modelo tiene una precisión de validación del 80 %, lo cual se refleja en los resultados. Se puede afirmar que no son malos resultados, aunque en algunos casos veamos frases incompletas.

temario oposiciones bombero
oposiciones bombero madrid
buscar de maquillaje en
policia nacional
cursos de cocina
vericion de peluqueria madrid
hay que hacer para ser policía nacional
yonductor de ambulancia y dietetica y biomee
servicio de polici
fp grado medio auxiliar

Tabla 4.9: Generación de *keywords* por caracteres para el cliente C

4.3. Generación de texto por palabras

En este apartado también se desarrollará tanto modelos de red con LSTM como modelos de red convolucional.

4.3.1. Modelo de red LSTM

El modelo que se usará en esta sección será el mismo modelo que se ha utilizado en la sección 4.2.1. La única diferencia con respecto al anterior modelo será que esta vez, al tratarse de generación por palabras, el modelo tendrá que ser entrenado por palabras. Es decir, en la representación numérica, el texto deberá ser tokenizado por palabras. Podríamos usar una codificación *one-hot*, al igual que antes, para representar numéricamente el texto. No obstante, dado que estamos hablando de que cada conjunto de datos cuenta con más de 4000 palabras distintas, cada palabra vendría representada con un vector de más de 4000 elementos. Trabajar con vectores de tal magnitud encarece mucho los gastos computacionales y no es para nada eficiente. Por ello, para reducir la dimensionalidad habrá que buscar alguna alternativa. El *Word2Vec* además de solucionarnos el problema de la dimensionalidad, nos proporciona una representación basada en el contexto semántico y sintáctico. Entonces, a priori parece una buena opción aplicar esta técnica a nuestros textos para transformarlos en vectores.

La librería *gensim* será la que nos ayude a representar nuestras palabras con *Word2Vec*. Este paquete nos permite aplicar directamente esta técnica sin tener que programar nosotros toda la red. Para ello, es necesario prefijar el tamaño de ventana de contexto y el tamaño del vector de salida que se quiere. En nuestro caso, se han fijado en 5 y 100 respectivamente, ya que nuestras frases no son tan largas

y no tenemos tantas palabras como para necesitar vectores de salida mayores.

Una vez tengamos los textos transformados, efectuamos el mismo entrenamiento que en la sección 4.2.1, es decir:

- La estrategia por la que se ha optado es la de estrategia por minibatch por sus ventajas respecto a las demás.
- La función de pérdida que mejor se adapta al problema es la de la entropía categórica cruzada.
- La inicialización de pesos no puede ser otra que la de Xavier uniforme.
- El algoritmo de optimización que se ha escogido es el Rmsprop, con su valor por defecto $\rho = 0{,}95$.
- En principio la tasa de aprendizaje que ofrece Keras, $\eta = 0{,}1$ debería ir bien.
- El tamaño de los lotes elegido es de 64.

El barrido de épocas, y número de unidades de memoria en cada cliente puede verse en la figura 4.10. De primeras, llama la atención el comportamiento del error de validación sobre todo en los clientes A y B, que prácticamente no baja nada. El entrenamiento actúa de forma correcta porque su error va decreciendo en cada época, pero en estos clientes no se consigue un modelo lo suficientemente general para poder reducir el error de validación también. En el cliente C en cambio, que recordemos que es el cliente con más datos, parece que el error de validación consigue reducirse pero tampoco mucho.

Analizando la precisión de estos modelos en la figura 4.9, observaremos que el *accuracy* de los modelos de los clientes A y B tampoco sube con el entrenamiento. Si que lo hace en cambio para el cliente C, aunque tampoco mucho. La precisión de los modelos puede ser considerablemente alta, pero el hecho de que suba poco o directamente no se incremente nada puede indicarnos que algo no va bien, y más si el valor de la función de pérdida tiene un comportamiento similar.

En cuanto los tiempos de ejecución, hemos conseguido modelos cuyo entrenamiento es considerablemente inferior a los modelos de tokenización por caracteres (ver tabla 4.10). Este resultado es totalmente lógico si pensamos que ahora por cada palabra se obtiene un vector mientras que antes cada palabra venía representado con un número de vectores igual al número de caracteres que formaban la palabra. Además, parece que el número de datos influye aproximadamente de la misma manera que en los anteriores casos, es decir, linealmente, y esto se hace notar más para los clientes B y C. Por otra parte, el hecho de aumentar el número de celdas de memoria cada vez tiene menos peso en el tiempo, dado que el tiempo aumenta proporcionalmente mucho más en el salto de 2000 a 4000 que de 4000 a 6000.

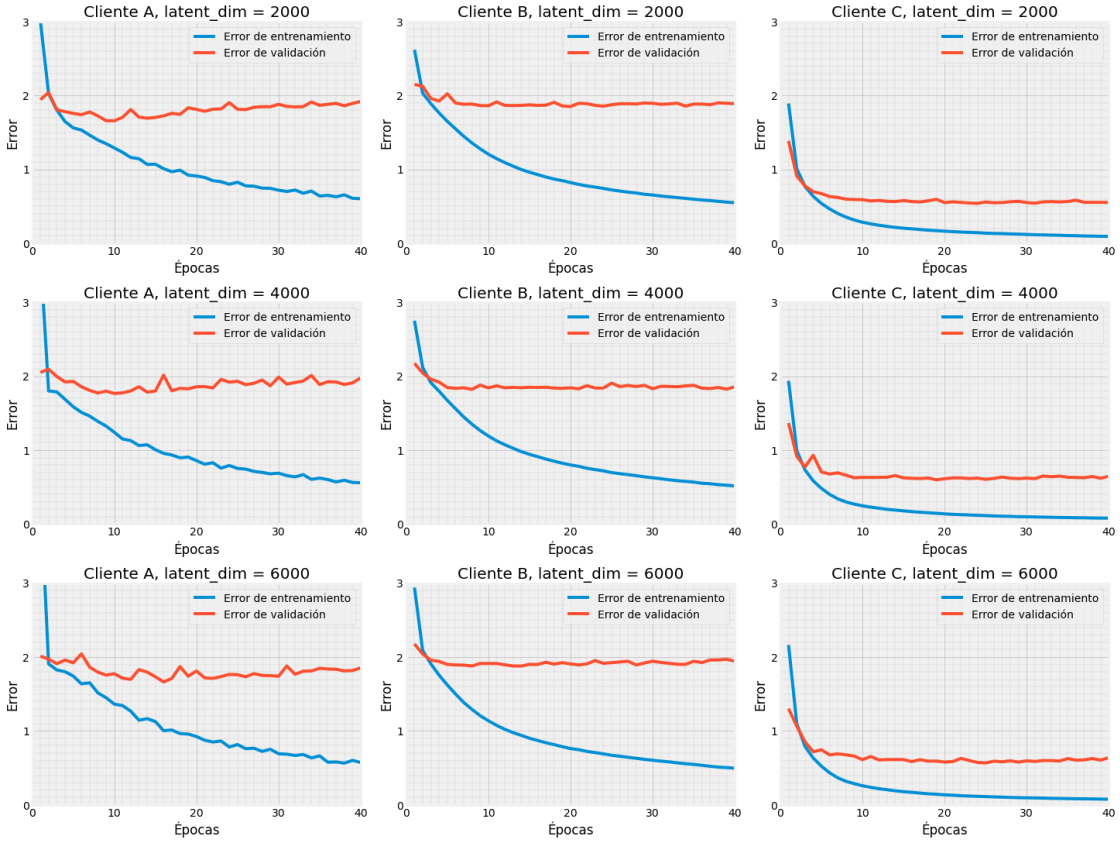


Figura 4.8: Error de distintos modelos con LSTM por palabras según cliente, número de celdas de memoria (latent-dim) y épocas.

número de celdas	Cliente A	Cliente B	Cliente C
2000	4 min	10 min	23 min
4000	24 min	71 min	148 min
6000	49 min	143 min	303 min

Tabla 4.10: Tiempos de computación de distintos modelos con LSTM por palabras según cliente y número de celdas de memoria.

Como mejor modelo general se puede escoger al que tiene 2000 celdas de memoria entrenado durante 10 épocas. Con esto, no se puede esperar mucho para la generación de *keywords* del cliente A y del cliente B, pero con cualquier otra elección tampoco se podría esperar nada mejor. Esta elección es la que mejor se adapta al cliente C, ya que el aumentar la cantidad de celdas de memoria lo único que se produce es más complejidad en el modelo y por lo tanto más coste computacional, y el aumentar las épocas tampoco produciría nada más que mayor tiempo de ejecución de entrenamiento.

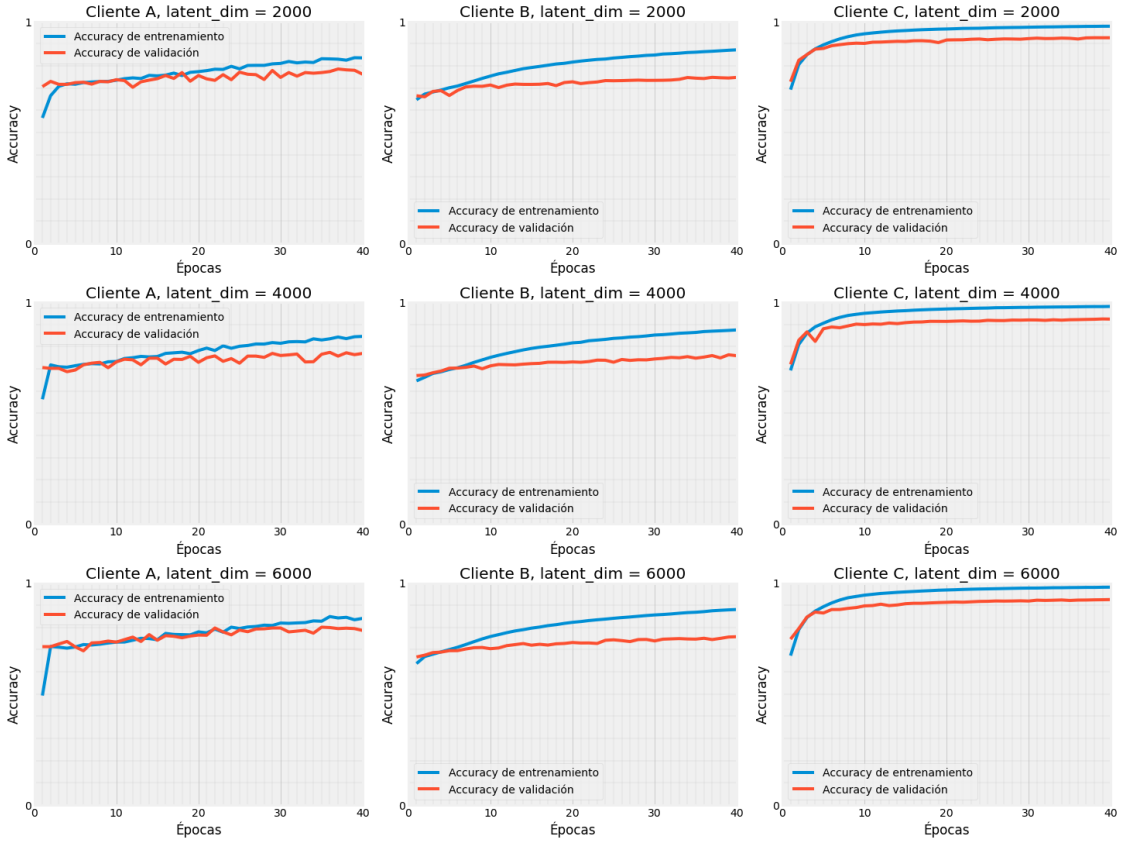


Figura 4.9: Precisión de distintos modelos con LSTM por palabras según cliente, número de celdas de memoria (latent-dim) y épocas.

Generación de *keywords*

Siguiendo el procedimiento de la generación de *keywords* con el primer modelo, pero en este caso por palabras en vez de caracteres, seremos capaces de generar las nuevas palabras calve. Con los modelos de 2000 celdas, entrenados durante 10 épocas podremos obtener los resultados de las tablas 4.11, 4.12 y 4.13. Viendo estas tablas enseguida nos damos cuenta que los resultados no son aprovechables como búsquedas claves. Además de la poca coherencia de las oraciones se observa cierta tendencia a la repetición de algunos tokens. Quizá se podría salvar alguna frase de la tabla 4.12 porque los datos de este cliente tienden a ser muy específicos en cuanto a nombres de los productos y por ello alguna frase que se crea podría tener un mínimo de sentido, pero en general estamos hablando de unos resultados muy malos.

granos france abrir franquicia
ranking franquicias franquicia ropa en infantil
neumaticos qué necesito al pizza españa
super inmoboliarias rentables pizzeria de franquicia
condis que vale a o folder ropa
café la mejores coches euros amazon
santa passion tiendas auto kfc ecologico dia
365 cual restauracion y ropa
danone france mejores ropa franquicia

Tabla 4.11: Generación de *keywords* por palabras para el cliente A

doctor sensilis s medical forte opiniones
kaidax desmaquillante de de gel
tratar dónde micro plata 180 caseros spanje 75gr 75gr 75gr sesderma sesderma
gx ergy 700 shake wash m solar
iappharma recambio gotas b5 capsulas opiniones
ceema desmaquillante de de gel
mayores satisfyer micro shake calmantes caseros repuestos sp gr essence gr serum
nilo sterillium 700 shake xt m solar
12 redoxon digital hydra 40 brown

Tabla 4.12: Generación de *keywords* por palabras para el cliente B

cuánto examenes cobra fer virtual virtual
policiq pedir nuevo curs convocatoria 2021
creativa bases ciudadano planner título 2021 2021 nuevo
plasencia puedo es es es es
marcelo examenes nuevo test 2018 2018 justicia
cafe logos mi blog justicia 2019 2019
ad maxima nuevo teleoperadora electricos universitarios previa
denuncias medical minima educacio justicia
gallego telefonos nuevo oposicions

Tabla 4.13: Generación de *keywords* por palabras para el cliente C

4.3.2. Modelo de red convolucional

Continuando con la dinámica del trabajo en esta sección se repetirá el mismo modelo que el de la sección 4.2.2 pero con el uso de palabras en vez de caracteres. Como en el caso de la LSTM, aquí también hemos elegido el *Word2Vec* como forma de representación vectorial de los datos textuales. Este cambio, nos limita a la hora de intentar replicar el modelo original por caracteres, ya que por temas de las dimensiones de los vectores que sustituyen a las frases, las matrices se van reduciendo hasta que llegan a un punto que no pueden reducirse mas. Para evitar este problema, introduciremos la técnica del *padding* en las capas de convolución. Por lo demás se seguirán las mismas condiciones de entrenamiento:

- La estrategia por la que se ha optado es la de estrategia por minibatch por sus ventajas respecto a las demás.
- La función de pérdida que mejor se adapta al problema es la de la entropía categórica cruzada.
- La inicialización de pesos no puede ser otra que la de Xavier uniforme.
- El algoritmo de optimización que se ha escogido es el Rmsprop, con su valor por defecto $\rho = 0{,}95$.
- Después de observar con unas pequeñas pruebas que el error de entrenamiento tendía a subir se ha bajado la tasa da aprendizaje a $\eta = 0{,}01$.
- El tamaño de los lotes elegido es de 64.
- La mejor opción para las funciones de activación de la capa oculta sin duda es la de la ReLU.

El error del barrido por cliente, cantidad de neuronas por capa y número de épocas con estos hiperparámetros nos da la siguiente figura 4.10. A primera vista todos los entrenamientos tienen bastante mala pinta porque la pérdida de entrenamiento apenas decrece y la de la validación directamente no lo hace. El único entrenamiento que se ve un poco mejor es el del cliente C, aunque en este tampoco veamos mucho descenso del error. Los gráficos de precisión que se ven en 4.11 no nos dan tampoco muchas esperanzas de que el entrenamiento haya sido correcto. En estos observamos como el *accuracy* prácticamente se mantiene constante durante todo el entrenamiento, indicándonos que el modelo no se mejora. Estos resultados no son problema de la configuración de la red neuronal o de los hiperpárametros usados, mas bien nos vienen a decir que probablemente este modelo no sea apropiado para la generación de textos por palabras.

Los tiempos de ejecución de estos entrenamientos se muestran en la tabla 4.14. Al igual que antes, es lógico concluir que como ahora por cada palabra obtenemos un vector y antes cada palabra venía representado con un número de vectores igual

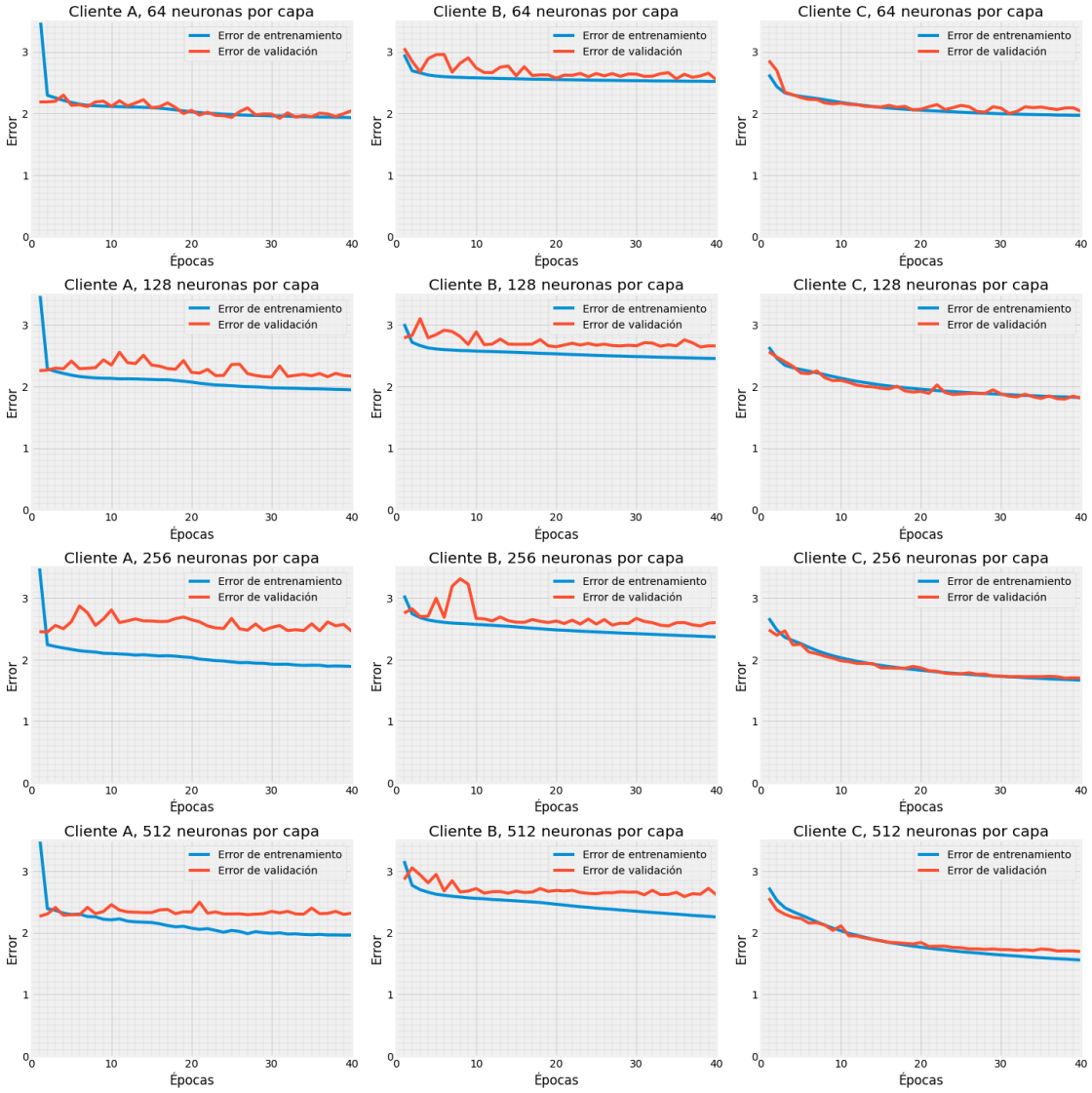


Figura 4.10: Error de distintos modelos convolucionales por palabras según cliente, número de neuronas por capa y épocas.

al número de caracteres que formaban la palabra, el tiempo de computación de estos entrenamientos será inferior. Parece que ene este caso no se cumple tan bien la relación de linealidad del tiempo de ejecución y el número de datos. Por otra parte, se ve que el aumento de neuronas por capa cada vez tiene mas relevancia en el peso. No obstante, este análisis no tiene mucha importancia debido a que por el transcurso del entrenamiento y sus gráficos de error y precisión no somos capaces de afirmar que algún modelo sea mejor que otro, aunque si que podemos afirmar que todos son bastante malos.

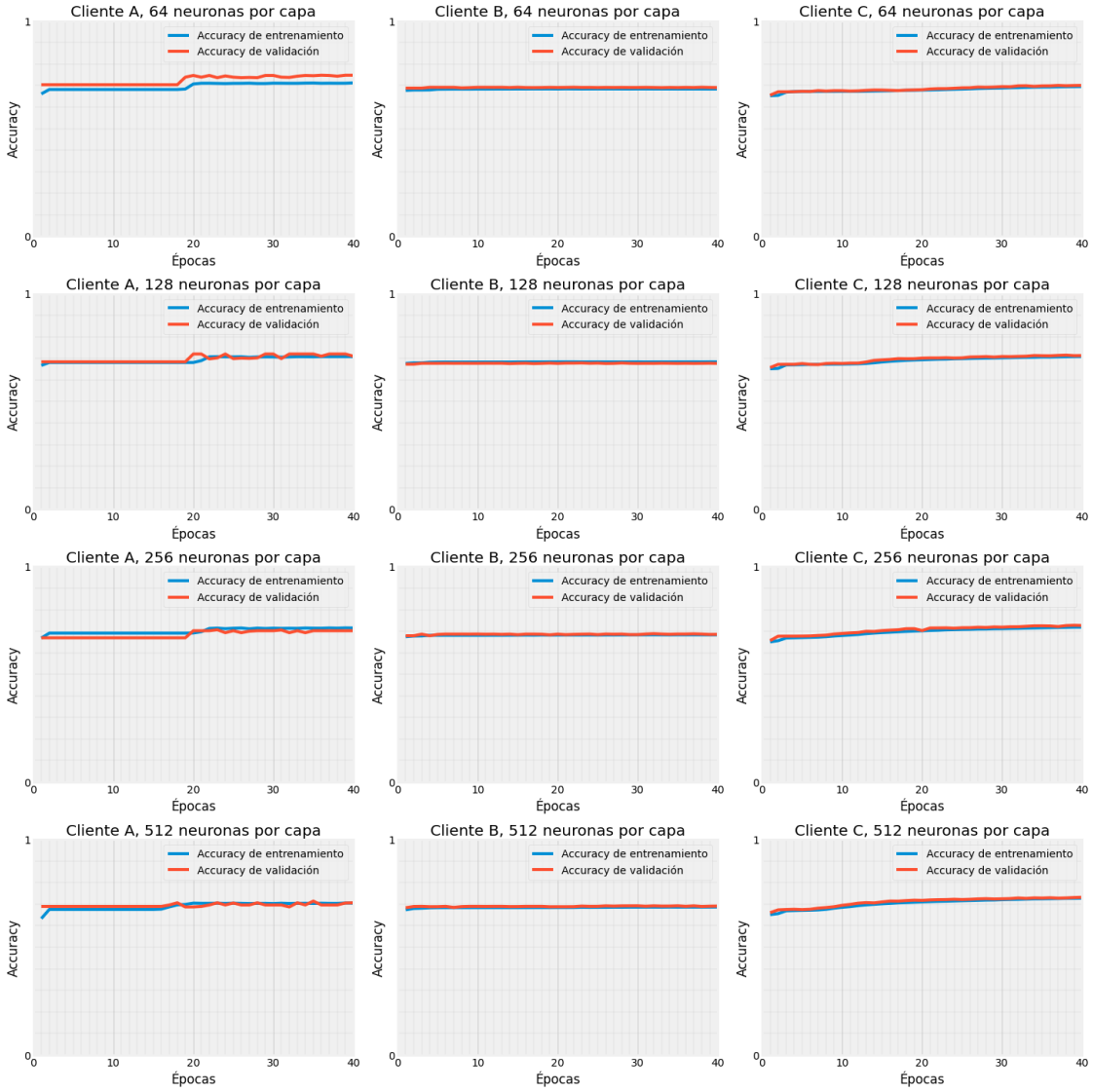


Figura 4.11: Precisión de distintos modelos convolucionales por palabras según cliente, número de neuronas por capa y épocas.

neuronas por capa	Cliente A	Cliente B	Cliente C
64	3 min	6 min	59 min
128	6 min	31 min	76 min
256	18 min	75 min	346 min
512	101 min	284 min	1951 min

Tabla 4.14: Tiempos de computación de distintos modelos convolucionales por palabras según cliente y cantidad de neuronas por capa.

Generación de *keywords*

A pesar de no haber obtenido buenos resultados y de intuir que la generación de palabras clave será bastante mala se procederá con la generación de *keywords*,

en las tablas 4.15, 4.16 y 4.17. Para generar estas palabras clave se ha utilizado una temperatura de una unidad, $T = 1$, en los modelos entrenados en 10 épocas de cada cliente de 64 neuronas por capa con la misma estrategia de generación utilizada hasta ahora. Esta elección se justifica entendiendo que el aumentar el número de épocas de entrenamiento o la cantidad de neuronas no producirá modelos con mayor *accuracy*. Como era de esperar, en estas tablas no observamos ni siquiera una única generación de texto con coherencia. Además, las oraciones creadas destacan por la repetición de ciertos tokens como el espacio vacío en las tablas 4.15 y 4.17, y el token "iese" en la tabla 4.16.

complete para
ar de de sin
reafirmante crema
lancetas comprar de
bariéderm de de rebotica
optimum de para pasta
di y facial
aldem de crema manos

Tabla 4.15: Generación de *keywords* por palabras para el cliente A

on guardia para academia iese iese
este de iese iese iese iese
na de iese iese iese iese iese iese
ejercicio para iese iese
fabra oposiciones iese iese
bazan en para iese en iese iese
libro oposiciones oposiciones iese iese
duracion para auxiliar iese iese

Tabla 4.16: Generación de *keywords* por palabras para el cliente B

24h en
camion de
plastico ropa
starbuck
día de
franquicia paqueteria
t4 trasporte
supermercats ropa de

Tabla 4.17: Generación de *keywords* por palabras para el cliente C

Capítulo 5

Conclusiones y trabajo futuro

5.1. Conclusiones

En este Trabajo de Fin de Máster, se ha abordado el estudio de la generación de *keywords* para campañas publicitarias en internet. Para realizar esta tarea se han implementado distintos modelos y se han ido analizando uno a uno para tratar de encontrar el mejor.

Como conclusión general, se puede afirmar que ni el modelo con LSTM, ni el modelo de red convolucional han resultado muy útiles a la hora de generar *keywords* palabra por palabra. En ambos casos los resultados han sido cuanto menos decepcionantes ya que no se ha logrado generar frases con una coherencia mínima. Esto puede deberse a principalmente dos factores: Las frases de entrenamiento y/o los modelos utilizados. Las frases de entrenamiento están tomadas de búsquedas realizadas por *Google*, que como se sabe y puede observarse, tienden a ser frases mas simplificadas de lo habitual, que además, presentan estructuras sintácticas no muy completas. Dado que el *Word2Vec* es una técnica que intenta representar las palabras haciendo uso del contexto sintáctico y semántico de los textos es posible que no haya logrado captar del todo bien estos contextos en las frases utilizadas, y esto haya derivado en un mal funcionamiento de nuestros modelos. Además, el hecho de que las propias frases presenten fallos de escritura y diferentes declinaciones para las mismas palabras, hace aumentar el número de palabras y por tanto su variabilidad lo que no ayuda en nada a dicha técnica. Por otro lado, es posible también que los modelos utilizados no sean suficientemente complejos como para aplicarlos en la generación por palabras, y que por ello, aunque se lograse que la representación numérica de las palabras fuese mejor, estos no obtuviesen mejores resultados. Con modelos más complejos, se hace referencia a combinaciones de redes neuronales que por ejemplo permitiesen codificar los textos de manera que se asegure que únicamente la información relevante se mantenga y decodificar esta

información para predecir el siguiente token.

En cuanto a la generación por caracteres, se puede decir que los resultados han sido más positivos que los de generación por palabras. Los modelos con LSTM han ofrecido mejores resultados en cuanto precisión que los convolucionales, y esto se ha notado en los *keywords* que se han generado con cada modelo. Otra ventaja que los primeros tienen con los segundos es su forma mas coherente y natural de finalizar las frases. Mientras que con LSTM los textos creados mantenían coherencia y sentido hasta el final, en la generación con redes convolucionales muchas veces se observaba que algunas frases parecían incompletas. Por lo tanto según el estudio realizado, la generación carácter por carácter mediante redes LSTM es la mejor opción (por lo menos entre las opciones que se ha puesto a prueba) a la hora de crear *keywords*, y de hecho podría resultar útil para automatizar las campañas de publicidad. La generación de textos con redes convolucionales en cambio, no ha resultado tan exitosa pero es una línea de trabajo muy reciente y por eso hay motivos suficientes para estar satisfechos con los resultados obtenidos.

Actualmente, la aplicación de técnicas de aprendizaje profundo en el campo de generación de textos es bastante novedosa. El objetivo en este trabajo era analizar distintos modelos y evaluar su funcionamiento para la generación automática de palabras clave para campañas publicitarias. Aunque no se hayan logrado los mejores resultados, se ha podido aportar un granito de arena y es por ello por lo que se puede estar satisfecho.

5.2. Trabajo futuro

En cuanto al trabajo futuro parece bastante evidente cuales podrían ser las dos líneas que podrían tomarse después de este trabajo, y las dos son la continuación del estudio de la generación de *keywords* por palabras :

- La primera tiene que ver con el primer factor que comentábamos antes por el que podrían no funcionar correctamente estos modelos: El *Word2Vec*. Relacionado con esto, podrían haber distintas nuevas líneas de trabajo futuro como por ejemplo el probar con otras técnicas como las ya mencionadas *GloVe* o *FastText*. No obstante, puede resultar más efectivo trabajar un poco los textos de entrenamiento. Es decir, por ejemplo, como se mencionaba en las conclusiones muchas palabras se presentan con fallos de escritura y/o varias declinaciones lo que hace que el número de palabras se multiplique. Por lo tanto, otra opción sería centrarse en la fase de pre-procesamiento del texto, ya sea corrigiendo los fallos de escritura, eliminando las *stop words* y/o lematizando. De esta manera, además de conseguir simplificar el texto original, permitiría que el *Word2Vec* funcionase mejor y en consecuencia que las redes neuronales pudieran ser entrenadas de una manera más correcta.

- Otra línea de trabajo futuro por la que se podría optar es la de analizar modelos más complejos. Los resultados de los modelos estudiados indicaban que el entrenamiento no se producía correctamente y comentábamos que una de las razones podría ser porque los modelos no eran suficientemente complejos como para capturar la información de los textos. Con modelos mas complejos no se refiere a por ejemplo aumentar el número de neuronas o de capas, sino a combinar distintas redes. En los últimos años, ha habido muchos avances en la generación de textos con aprendizaje profundo, y dos modelos parecen haber resultado bastante buenos con esta tarea [9]: Los denominados *Variational Auto-Encoders (VAE)* y los *Generative Adversarial Networks (GAN)*. Ambos, son modelos que utilizan redes neuronales, pero son bastante más complejos que los que hemos usado en este trabajo. Viendo que la rama de la generación de textos con *deep-learning* ha tomado ese camino, probar con cualquiera de los dos modelos puede parecer una buena opción de trabajo futuro.

Bibliografía

[1] M. Berry and G. Linoff. *Data Mining Techniques*. John Wiley and Sons, 1997.

[2] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics*, June 2017.

[3] Jason Brownlee. What are word embeddings for text?, 2017.

[4] Jiakang Chang. Text mining 101. *openminted*, 2018.

[5] Domo. Data never sleeps 8.0, 2021.

[6] X. Glorot and Y.Bengio. Understanding the difficulty of training deep feedforward neural networks. *Society for Artificial Intelligence and Statistics*, 2010.

[7] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural comput*, 9:1735–1780, 1997.

[8] J. J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Procedings of the National Academy of Sciences*, 79:2554–2558, 1982.

[9] T. Iqbal and S. Qureshi. The survey: Text generation models in deep learning. *Journal of King Saud University – Computer and Information Sciences*, page 135–146, April 2020.

[10] Y. LeCun. Une procédure d'apprentissage pour réseau a seuil asymmetrique. *Proc. Cogn*, 85:599–604, 1985.

[11] T. Masters. Practical neural network recipes in c++. *Academic Press Professional*, 1993.

[12] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. *Cornell University*, 09 2013.

[13] T. Mikolov, K. Chen, G. Corrado, J. Dean, and I. Sutskeber. Distributed representations of words and phrases and their compositionality. *Cornell University*, 10 2013.

[14] M. Minsky and S. Papert. Perceptrons: An introduction to computational geometry. *MIT Press*, 1969.

[15] Jeremy Neiman. Generating haiku with deep learning. *towards data science*, 2018.

[16] D.B. Parker. Learning-logic. *MIT Center for Computational Research in Economics and Management Science*, 1985.

[17] Jeffrey Pennington, Richard Socher, and Christopher Manning. GloVe: Global vectors for word representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, Doha, Qatar, October 2014. Association for Computational Linguistics.

[18] D.E. Rumelhart, R.J G.E. Hinton, and Williams. Learning internal representations by error propagation. *Parallel Distrib. Process*, 1:318–362, 1986.

[19] M. Shacklett. Unstructured data: A cheat sheet. *TechRepublic*, 2017.

[20] J. Wen, J.L. Zhao, S.W. Luo, and Z. Han. The improvements of bp neural network learning algorithm in proceedings of 5th international conference on signal processing. *IEEE Press*, pages 1647–1649, 2000.

[21] Dylan Wenzlau. Meme text generation with a deep convolutional network in keras and tensorflow. *towards data science*, 2019.

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

[23] P.J. Werbos. Beyond regression: New tools for prediction and analysis in the behavioral sci- ences. *Harvard University*, 1975.

Anexos

Entrenamiento de modelo LSTM por caracteres

```
1
2  """
3  Created on Mon May 17 09:54:01 2021
4
5  @author: ina
6  """
7
8  import os
9  from pathlib import Path
10
11 import tensorflow.compat.v1 as tf1
12 tf1.disable_v2_behavior()
13 tf_session = tf1.Session()
14
15 from tensorflow.compat.v1.keras import backend as K
16 K.set_session(tf_session)
17
18 from tensorflow.keras.callbacks import ModelCheckpoint,
       CSVLogger
19 from tensorflow.keras.layers import Add, Dense, Input, LSTM
20 from tensorflow.keras.models import Model
21 from tensorflow.keras.preprocessing.text import Tokenizer
22
23 import numpy as np
24 import pandas as pd
25 import joblib
26
27 import datetime
28 import time
29 for o in range(10):
```

```
30      try:
31          from keras.utils import np_utils
32          print('Works!')
33          break
34      except Exception as e:
35          print(e,o)
36          time.sleep(0.5)
37
38
39  # -------------------FUNCIONES Y CLASES
        ----------------------------------------
40
41  class TrainingLine:
42      def __init__(self, name, previous_line, lstm, n_tokens):
43          self.char_input = Input(shape=(None, n_tokens),
44                                  name='char_input_%s' % name)
45
46          self.numberchar_input = Input(
47              shape=(1,),name='numberchar_input_%s' % name)
48          self.numberchar_dense = Dense(
49              lstm.units, activation='relu', name='
                    numberchar_dense_%s' % name)
50          self.numberchar_dense_output = self.numberchar_dense(
                self.numberchar_input)
51
52          self.lstm = LSTM(
53          latent_dim, return_state=True, return_sequences=True,
                name='lstm_%s' % name)
54          initial_state = [self.numberchar_dense_output, self.
                numberchar_dense_output]
55
56          self.lstm_out, self.lstm_h, self.lstm_c = lstm(
57              self.char_input, initial_state=initial_state)
58
59          self.output_dense = Dense(
60              n_tokens, activation='softmax', name='output_%s' %
                    name)
61          self.output = self.output_dense(self.lstm_out)
62
63  def create_training_model(latent_dim, n_tokens):
64      lstm = LSTM(latent_dim, return_state=True, return_sequences
            =True, name='lstm')
65      lines = []
66      inputs = []
```

```
67      outputs = []
68
69      previous_line = lines[-1] if lines else None
70      lines.append(TrainingLine('line_0', previous_line, lstm,
            n_tokens))
71      inputs += [lines[-1].char_input, lines[-1].numberchar_input
            ]
72      outputs.append(lines[-1].output)
73
74      training_model = Model(inputs, outputs)
75      training_model.compile(optimizer='rmsprop', loss='
            categorical_crossentropy')
76
77      return training_model, lstm, lines, inputs, outputs
78
79  class GeneratorLine:
80      def __init__(self, name, training_line, lstm, n_tokens):
81          self.char_input = Input(
82              shape=(None, n_tokens), name='char_input_%s' % name
                    )
83
84          self.numberchar_input = Input(
85              shape=(1,), name='numberchar_input_%s' % name)
86          self.numberchar_dense = Dense(
87              lstm.units, activation='relu', name='
                    numberchar_dense_%s' % name)
88          self.numberchar_dense_output = self.numberchar_dense(
                self.numberchar_input)
89
90          self.h_input = Input(shape=(lstm.units,), name='
                h_input_%s' % name)
91          self.c_input = Input(shape=(lstm.units,), name='
                c_input_%s' % name)
92          initial_state = [self.h_input, self.c_input]
93
94          self.lstm = lstm
95
96          self.lstm_out, self.lstm_h, self.lstm_c = self.lstm(
97              self.char_input, initial_state=initial_state)
98
99          self.output_dense = Dense(
100             n_tokens, activation='softmax', name='output_%s' %
                    name)
101         self.output = self.output_dense(self.lstm_out)
```

```
102
103             self.numberchar_dense.set_weights(
104                 training_line.numberchar_dense.get_weights())
105             self.lstm.set_weights(lstm.get_weights())
106             self.output_dense.set_weights(training_line.
                    output_dense.get_weights())
107
108
109     def entrenar(epochs=30, latent_dim=500,root_path="",name="259
            _424_9086"):
110         sample_size = 1
111         data_path = os.path.join(root_path,"KW_ST_files", name,"
                st_report2020.csv")
112
113         output_dir = Path('%d_%d_output_test_%s' % (latent_dim,
                epochs,name))
114         try:
115             output_dir.mkdir()
116         except:
117             pass
118         sample_size = 1
119         data_path = os.path.join(root_path,"KW_ST_files", name,"
                st_report2020.csv")
120
121
122         time_file = open(output_dir / ("inicial_time_%s-%s.txt" % (
                latent_dim,epochs)),"a")
123         time_file.write(str(datetime.datetime.now().time()))
124         time_file.close()
125
126
127         df_raw = pd.read_csv(data_path)
128         df_raw = df_raw.sample(frac=sample_size)
129         conversions = np.array(df_raw["Conversions"])
130
131
132
133         df = df_raw['Search term'].drop([np.where(conversions<1)
                ][0][0],axis=0)
134         max_length = int(max([df_raw["Search term"].str.len().
                quantile(.99)]))
135         df = pd.DataFrame(df[(df_raw['Search term'].str.len() <=
                max_length)].copy())
136         ncharacters = []
```

```
137     ncharacters = [len(df['Search term'].values[i])  for i in
            range(len(df))]
138     df['characters'] = ncharacters
139 
140     df['in'] = (df['Search term'].str[0] + df['Search term']).
            str.pad(max_length+2, 'right', '\n')
141     df['out'] = df['Search term'].str.pad(max_length+2, 'right'
            , '\n')
142 
143 
144     inputs = df["in"]
145 
146     tokenizer = Tokenizer(filters='', char_level=True)
147     tokenizer.fit_on_texts(inputs)
148     n_tokens = len(tokenizer.word_counts) + 1
149 
150     # X is the input for each line in sequences of one-hot-
            encoded values
151     X = np_utils.to_categorical([
152         tokenizer.texts_to_sequences(inputs)
153     ], num_classes=n_tokens)
154 
155     outputs = df["out"]
156 
157     # Y is the output for each line in sequences of one-hot-
            encoded values
158     Y = np_utils.to_categorical([
159         tokenizer.texts_to_sequences(outputs)
160     ], num_classes=n_tokens)
161 
162     # X_characters is the count of characters for each line
163 
164     X_characters = df[['characters']].values
165 
166 
167 
168 
169     ##      TRAINING MODEL
170     training_model, lstm, lines, inputs, outputs =
            create_training_model(latent_dim, n_tokens)
171 
172     if epochs==60:
173 
174         joblib.dump([latent_dim, n_tokens, max_length,
```

```
            tokenizer], str(output_dir / ('metadata_%s-%s.pkl' %
             (latent_dim,epochs)) ))
175
176         filepath = str(output_dir / ( "%s-{epoch:02d}-{loss:.2f
               }-{val_loss:.2f}.hdf5" % latent_dim))
177
178         checkpoint = ModelCheckpoint(
179             filepath, monitor='loss', verbose=1, save_best_only
                =True, mode='min',period=10)
180
181         csv_logger = CSVLogger(
182             str(output_dir / ('training_log_%s-%s.csv' % (
                latent_dim,epochs))), append=True, separator=','
                )
183
184
185
186         callbacks_list = [checkpoint, csv_logger]
187     else:
188         callbacks_list = None
189
190
191     training_model.fit([
192         X[0],X_characters[:,0]], Y[0], batch_size=64,epochs=
            epochs,
193         validation_split=.1 ,callbacks=callbacks_list)
194
195     return lstm, lines, tokenizer, n_tokens, max_length
196
197
198
199
200
201 #-------------------------ENTRENAMIENTO
       ---------------------------------
202 root_path = os.path.abspath(os.path.dirname(__file__))
203 os.chdir(root_path)
204 dataframes = ["437_910_0412","991_035_4076","259_424_9086"]
205 latent_dimension = [2000,4000,6000]
206 number_epoch = [60]
207 for dataframe_name in dataframes:
208     for n in latent_dimension:
209         for e in number_epoch:
210             latent_dim = n
```

```
211                 lstm, lines, tokenizer, n_tokens, max_length =
                        entrenar(
212                 epochs=e,latent_dim=latent_dim,root_path=root_path,
                        name=dataframe_name)
```

Módulo para generación por caracteres con modelo LSTM

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Mon May 17 12:09:58 2021

@author: ina
"""


import tensorflow.compat.v1 as tf1 #new
tf1.disable_v2_behavior() #new
tf_session = tf1.Session()

from tensorflow.compat.v1.keras import backend as K #new
# from keras import backend as K
K.set_session(tf_session)

from keras.layers import Add, Dense, Input, LSTM
from keras.models import Model
from keras.utils import np_utils

import numpy as np

from keras.layers import Add, Dense, Input, LSTM


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)

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.numberchar_input = Input(shape=(1,), name='
            numberchar_input_%s' % name)
```

```
40          self.numberchar_dense = Dense(lstm.units, activation='
                relu', name='numberchar_dense_%s' % name)
41          self.numberchar_dense_output = self.numberchar_dense(
                self.numberchar_input)
42
43          #self.lstm = LSTM(latent_dim, return_state=True,
                return_sequences=True, name='lstm_%s' % name)
44          initial_state = [self.numberchar_dense_output, self.
                numberchar_dense_output]
45
46          self.lstm_out, self.lstm_h, self.lstm_c = lstm(self.
                char_input, initial_state=initial_state)
47
48          self.output_dense = Dense(n_tokens, activation='softmax
                ', name='output_%s' % name)
49          self.output = self.output_dense(self.lstm_out)
50
51  def create_training_model(latent_dim, n_tokens):
52      lstm = LSTM(latent_dim, return_state=True, return_sequences
            =True, name='lstm')
53      lines = []
54      inputs = []
55      outputs = []
56
57      previous_line = lines[-1] if lines else None
58      lines.append(TrainingLine('line_0', previous_line, lstm,
            n_tokens))
59      inputs += [lines[-1].char_input, lines[-1].numberchar_input
            ]
60      outputs.append(lines[-1].output)
61
62      training_model = Model(inputs, outputs)
63      training_model.compile(optimizer='rmsprop', loss='
            categorical_crossentropy')
64
65      return training_model, lstm, lines, inputs, outputs
66
67  class GeneratorLine:
68      def __init__(self, name, training_line, lstm, n_tokens):
69          self.char_input = Input(shape=(None, n_tokens), name='
                char_input_%s' % name)
70
71          self.numberchar_input = Input(shape=(1,), name='
                numberchar_input_%s' % name)
```

```
72          self.numberchar_dense = Dense(lstm.units, activation='
                relu', name='numberchar_dense_%s' % name)
73          self.numberchar_dense_output = self.numberchar_dense(
                self.numberchar_input)
74
75          self.h_input = Input(shape=(lstm.units,), name='
                h_input_%s' % name)
76          self.c_input = Input(shape=(lstm.units,), name='
                c_input_%s' % name)
77          initial_state = [self.h_input, self.c_input]
78
79          self.lstm = lstm
80
81          self.lstm_out, self.lstm_h, self.lstm_c = self.lstm(
                self.char_input, initial_state=initial_state)
82
83          self.output_dense = Dense(n_tokens, activation='softmax
                ', name='output_%s' % name)
84          self.output = self.output_dense(self.lstm_out)
85
86          self.numberchar_dense.set_weights(training_line.
                numberchar_dense.get_weights())
87          #self.lstm.set_weights(lstm.get_weights())
88          self.output_dense.set_weights(training_line.
                output_dense.get_weights())
89
90  class Generator:
91      def __init__(self, lstm, lines, tf_session, tokenizer,
            n_tokens, max_line_length):
92          self.tf_session = tf_session
93          self.tokenizer = tokenizer
94          self.n_tokens = n_tokens
95          self.max_line_length = max_line_length
96
97          self.lstm = LSTM(
98              lstm.units, return_state=True, return_sequences=
                    True,
99              name='generator_lstm'
100         )
101         self.lines = [
102             GeneratorLine(
103                 'generator_line',
104                 lines[0], self.lstm, self.n_tokens
105             )
```

```
106             ]
107             self.lstm.set_weights(lstm.get_weights())
108
109         def generate_haiku(self, numberchar=25, temperature=.1,
                first_char=None):
110             output = []
111             h = None
112             c = None
113
114             if first_char is None:
115                 first_char = chr(int(np.random.randint(ord('a'),
                        ord('z')+1)))
116
117             next_char = self.tokenizer.texts_to_sequences(
                    first_char)[0][0]
118
119
120             line = self.lines[0]
121             s = self.tf_session.run(
122                 line.numberchar_dense_output,
123                 feed_dict={
124                     line.numberchar_input: [[numberchar]]
125                 }
126             )
127
128             if h is None:
129                 h = s
130                 c = s
131             else:
132                 h = h + s
133                 c = c + s
134
135             line_output = [next_char]
136             end = False
137             next_char = None
138             for i in range(self.max_line_length):
139
140                 char, h, c = self.tf_session.run(
141                     [line.output, line.lstm_h, line.lstm_c],
142                     feed_dict={
143                         line.char_input: [[
144                             np_utils.to_categorical(
145                                 line_output[-1],
146                                 num_classes=self.n_tokens
```

```
                            )
                        ]],
                        line.h_input: h,
                        line.c_input: c
                    }
                )
                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(cleaned_text)
            output.append(cleaned_text)

            return output
```

Entrenamiento de modelo convolucional por caracteres

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Thu Jun 17 17:13:23 2021

@author: ina
"""


import os

from tensorflow.keras.callbacks import ModelCheckpoint,
    CSVLogger
from tensorflow.keras.layers import Add, Dense
from tensorflow.keras.models import Model
from tensorflow.keras.preprocessing.text import Tokenizer

from pathlib import Path
import numpy as np
import pandas as pd
import joblib


from tensorflow.keras.layers import  Dropout
from tensorflow.keras.layers import Conv2D, MaxPooling2D,
    Flatten
from tensorflow.keras import Sequential

import datetime
import time
for o in range(10):
    try:
        from keras.utils import np_utils
        print('Works!')
        break
    except Exception as e:
        print(e,o)
        time.sleep(0.5)


def entrenar(epochs=30, neuronas_capa=512, kernel_size = 3,
```

```
    root_path="",name="259_424_9086"):
40      data_path = os.path.join(root_path,"KW_ST_files", name,"
            st_report2020.csv")
41
42      output_dir = Path('%d_%d_cnn_%s' % (neuronas_capa,epochs,
            name))
43      try:
44          output_dir.mkdir()
45      except:
46          pass    # Percent of samples to use for training, might
                be necessary if you're running out of memory
47      data_path = os.path.join(root_path,"KW_ST_files", name,"
            st_report2020.csv")
48
49
50      time_file = open(output_dir / ("inicial_time_%s-%s.txt" % (
            neuronas_capa, name)),"a")
51      time_file.write(str(datetime.datetime.now().time()))
52      time_file.close()
53      df_raw = pd.read_csv(data_path)
54      # df_raw = df_raw.sample(frac=sample_size)
55      conversions = np.array(df_raw["Conversions"])
56
57
58
59      df = df_raw['Search term'].drop([np.where(conversions<1)
            ][0][0],axis=0)
60      max_length = int(max([df_raw["Search term"].str.len().
            quantile(.99)]))
61      df = pd.DataFrame(df[(df_raw['Search term'].str.len() <=
            max_length)].copy())
62      ncharacters = []
63      ncharacters = [len(df['Search term'].values[i])  for i in
            range(len(df))]
64      df['characters'] = ncharacters
65
66      search = np.array(df["Search term"])
67      inp_list = []
68      out_list = []
69      for i in range(len(search)):
70          for j in range(1,len(search[i])):
71              inp_list.append(search[i][0:j])
72              out_list.append(list(search[i][j]))
73      inputs = pd.DataFrame(inp_list)
```

```
74      outputs = pd.DataFrame(out_list)
75      inputs[0] = inputs[0].str.pad(max_length+2,"right","\n")
76
77      tokenizer = Tokenizer(filters='', char_level=True)
78      tokenizer.fit_on_texts(inputs[0])
79      n_tokens = len(tokenizer.word_counts) + 1
80      X = np_utils.to_categorical(np.array(tokenizer.
            texts_to_sequences(inputs[0])), num_classes=n_tokens)
81      # Y = np_utils.to_categorical([tokenizer.texts_to_sequences
            (outputs[0])])
82      Y = np_utils.to_categorical(tokenizer.texts_to_sequences(
            outputs[0]), num_classes=n_tokens)
83      X = X.reshape(len(X),max_length+2,n_tokens,1)
84
85      model = Sequential()
86
87      #add model layers
88      model.add(Conv2D(neuronas_capa, kernel_size=kernel_size,
            activation='relu', input_shape=(max_length+2,n_tokens,1)
            ))
89      model.add(MaxPooling2D(pool_size = (2,2)))
90      model.add(Dropout(0.25))
91
92      model.add(Conv2D(neuronas_capa, kernel_size=kernel_size,
            activation='relu'))
93      model.add(MaxPooling2D(pool_size = (2,2)))
94      model.add(Dropout(0.25))
95
96      model.add(Conv2D(neuronas_capa, kernel_size=kernel_size,
            activation='relu'))
97      model.add(MaxPooling2D(pool_size = (2,2)))
98      model.add(Dropout(0.25))
99
100
101     model.add(Conv2D(neuronas_capa, kernel_size=kernel_size,
            activation='relu'))
102     model.add(Flatten())
103     model.add(Dense(n_tokens, activation='softmax'))
104
105
106     model.compile(loss='categorical_crossentropy', optimizer='
            rmsprop', metrics=['acc'])
107
108
```

```
109     joblib.dump([neuronas_capa, n_tokens, max_length, tokenizer
            ], str(output_dir / ('metadata_%s-%s.pkl' % (
            neuronas_capa, name)) ))
110
111     filepath = str(output_dir / ( "%s-{epoch:02d}-{loss:.2f}-{
            val_loss:.2f}.hdf5" % neuronas_capa))
112
113     checkpoint = ModelCheckpoint(
114                 filepath, monitor='loss', verbose=1,
                        save_best_only=True, mode='min',period=10)
115
116     csv_logger = CSVLogger(
117         str(output_dir / ('training_log_%s-%s.csv' % (
                neuronas_capa, name))), append=True, separator=',')
118
119     callbacks_list = [checkpoint, csv_logger]
120
121
122     history = model.fit(
123             X, Y, batch_size=neuronas_capa,epochs=epochs,
124             validation_split=.1, callbacks=callbacks_list)
125     return model, history
126 #--------------------------ENTRENAMIENTO
        -----------------------------------------------
127
128 root_path = os.path.abspath(os.path.dirname(__file__))
129 os.chdir(root_path)
130 dataframes = ["259_424_9086","437_910_0412","991_035_4076"]
131 neuronas_capa = [64,128,256,512]
132 number_epoch = [40]
133 for dataframe_name in dataframes:
134     for n in neuronas_capa:
135         for e in number_epoch:
136             model, history = entrenar(
137                 epochs=e, neuronas_capa=n, kernel_size = 3,
                        root_path=root_path,name=dataframe_name)
```

Módulo para generación por caracteres con modelo convolucional

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Mon May 17 12:09:58 2021

@author: ina
"""


import tensorflow.compat.v1 as tf1
tf1.disable_v2_behavior()
tf_session = tf1.Session()

from tensorflow.compat.v1.keras import backend as K
K.set_session(tf_session)

import numpy as np


import time
for o in range(10):
    try:
        from keras.utils import np_utils
        print('Works!')
        break
    except Exception as e:
        print(e,o)
        time.sleep(0.5)

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)

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

```
41          self.tokenizer = tokenizer
42          self.n_tokens = n_tokens
43          self.max_line_length = max_line_length
44          self.model = model
45      def generate_kword(self, temperature=.1, first_char=None,
            kw_length=None):
46          encoded_output = []
47          if first_char is None:
48              first_char = chr(int(np.random.randint(ord('a'),
                    ord('z')+1)))
49          if kw_length is None:
50              kw_length = self.max_line_length-5
51
52          first_char_encoded = np_utils.to_categorical(
53              self.tokenizer.texts_to_sequences(first_char)
                    [0][0],num_classes=self.n_tokens)
54          encoded_output.append(self.tokenizer.texts_to_sequences
                (first_char)[0][0])
55          space = np_utils.to_categorical(
56              self.tokenizer.texts_to_sequences('\n'),num_classes
                    =self.n_tokens).reshape(self.n_tokens,1)
57          space2 = np_utils.to_categorical(
58              self.tokenizer.texts_to_sequences(' '),num_classes=
                    self.n_tokens).reshape(self.n_tokens,1)
59
60          out = np.repeat(
61              space.reshape(1,self.n_tokens),self.max_line_length
                    +2,axis=0).reshape(
62                  1,self.max_line_length+2,self.n_tokens,1)
63          out[0][0] = first_char_encoded.reshape(self.n_tokens,1)
64          encoded_output.append(self.tokenizer.texts_to_sequences
                (first_char)[0][0])
65
66          for i in range(1,kw_length):
67              new_out = sample(self.model.predict(out)[0],
                    temperature)
68              encoded_output.append(new_out)
69              out[0][i] = np_utils.to_categorical(new_out,
                    num_classes=self.n_tokens).reshape(self.n_tokens
                    ,1)
70              if np.array_equal(space,out[0][i]):
71                  break
72          if not np.array_equal(space,out[0][i]) :
73              for j in range(1,10):
```

```
74                    new_out = sample(self.model.predict(out)[0],
                          temperature)
75                    encoded_output.append(new_out)
76                    out[0][i+j] = np_utils.to_categorical(new_out,
                          num_classes=self.n_tokens).reshape(self.
                          n_tokens,1)
77                    if np.array_equal(space2,out[0][i+j]):
78                        break
79
80
81
82            output_raw = self.tokenizer.sequences_to_texts([
                  encoded_output])
83            output = output_raw[0].strip()[1:].replace('    ', '\n')
                  .replace(' ', '').replace('\n', ' ')
84            return output
```

Entrenamiento de modelo LSTM por palabras

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Mon May 17 09:54:01 2021

@author: ina
"""
import os
import gensim
from pathlib import Path

import tensorflow.compat.v1 as tf1
tf1.disable_v2_behavior()
tf_session = tf1.Session()

from tensorflow.compat.v1.keras import backend as K
K.set_session(tf_session)

from tensorflow.keras.callbacks import ModelCheckpoint,
    CSVLogger
from tensorflow.keras.layers import Add, Dense, Input, LSTM
from tensorflow.keras.models import Model
from tensorflow.keras.preprocessing.text import Tokenizer

import numpy as np
import pandas as pd
import joblib

import datetime
import time
for o in range(10):
    try:
        #from keras.utils import to_categorica
        from keras.utils.np_utils import to_categorical
        print('Works!')
        break
    except Exception as e:
        print(e,o)
        time.sleep(0.5)


```

```
41 # -------------------FUNCIONES Y CLASES
       ----------------------------------------
42
43 class TrainingLine:
44     def __init__(self, name, previous_line, lstm, n_tokens):
45         self.char_input = Input(shape=(None, 100),
46                                 name='char_input_%s' % name)
47
48         self.numberchar_input = Input(
49             shape=(1,),name='numberchar_input_%s' % name)
50         self.numberchar_dense = Dense(
51             lstm.units, activation='relu', name='
                numberchar_dense_%s' % name)
52         self.numberchar_dense_output = self.numberchar_dense(
               self.numberchar_input)
53
54         self.lstm = LSTM(
55         latent_dim, return_state=True, return_sequences=True,
               name='lstm_%s' % name)
56         initial_state = [self.numberchar_dense_output, self.
               numberchar_dense_output]
57
58         self.lstm_out, self.lstm_h, self.lstm_c = lstm(
59             self.char_input, initial_state=initial_state)
60
61         self.output_dense = Dense(
62             n_tokens, activation='softmax', name='output_%s' %
                name)
63         self.output = self.output_dense(self.lstm_out)
64
65 def create_training_model(latent_dim, n_tokens):
66     lstm = LSTM(latent_dim, return_state=True, return_sequences
           =True, name='lstm')
67     lines = []
68     inputs = []
69     outputs = []
70
71     previous_line = lines[-1] if lines else None
72     lines.append(TrainingLine('line_0', previous_line, lstm,
           n_tokens))
73     inputs += [lines[-1].char_input, lines[-1].numberchar_input
           ]
74     outputs.append(lines[-1].output)
75
```

```
76      training_model = Model(inputs, outputs)
77      training_model.compile(optimizer='rmsprop', loss='
            categorical_crossentropy',metrics=['categorical_accuracy
            ','accuracy'])
78
79      return training_model, lstm, lines, inputs, outputs
80
81  class GeneratorLine:
82      def __init__(self, name, training_line, lstm, n_tokens):
83          self.char_input = Input(
84              shape=(None, 100), name='char_input_%s' % name)
85
86          self.numberchar_input = Input(
87              shape=(1,), name='numberchar_input_%s' % name)
88          self.numberchar_dense = Dense(
89              lstm.units, activation='relu', name='
                  numberchar_dense_%s' % name)
90          self.numberchar_dense_output = self.numberchar_dense(
                self.numberchar_input)
91
92          self.h_input = Input(shape=(lstm.units,), name='
                h_input_%s' % name)
93          self.c_input = Input(shape=(lstm.units,), name='
                c_input_%s' % name)
94          initial_state = [self.h_input, self.c_input]
95
96          self.lstm = lstm
97
98          self.lstm_out, self.lstm_h, self.lstm_c = self.lstm(
99              self.char_input, initial_state=initial_state)
100
101         self.output_dense = Dense(
102             n_tokens, activation='softmax', name='output_%s' %
                    name)
103         self.output = self.output_dense(self.lstm_out)
104
105         self.numberchar_dense.set_weights(
106             training_line.numberchar_dense.get_weights())
107         self.lstm.set_weights(lstm.get_weights())
108         self.output_dense.set_weights(training_line.
                output_dense.get_weights())
109
110
111
```

```
112 def entrenar(epochs=30, latent_dim=500,root_path="",name="259_424_9086"):
113
114     sample_size = 1
115     data_path = os.path.join(root_path,"KW_ST_files", name,"st_report2020.csv")
116
117     output_dir = Path('%d_%d_words_output_test_%s' % (latent_dim,epochs,name))
118     try:
119         output_dir.mkdir()
120     except:
121         pass     # Percent of samples to use for training, might be necessary if you're running out of memory
122     data_path = os.path.join(root_path,"KW_ST_files", name,"st_report2020.csv")
123
124
125
126
127     time_file = open(output_dir / ("inicial_time_%s-%s.txt" % (latent_dim,epochs)),"a")
128     time_file.write(str(datetime.datetime.now().time()))
129     time_file.close()
130
131     df_raw = pd.read_csv(data_path)
132     df_raw = df_raw.sample(frac=sample_size)
133     conversions = np.array(df_raw["Conversions"])
134
135     df = pd.DataFrame()
136
137     df['Search term'] = df_raw['Search term'].drop([np.where(conversions<1)][0][0],axis=0)
138     max_length = int(max([pd.Series([df_raw['Search term'].values[n].count(' ')+1 for n in range(len(df_raw))]).quantile(.99)]))
139     df = pd.DataFrame(df[(df['Search term'].str.count(' ') <= max_length)].copy())
140     nwords = []
141     # ncharacters = [len(df['Search term'].values[i])  for i in range(len(df))]
142     nwords = [df['Search term'].values[n].count(' ')+1 for n in range(len(df))]
143     df['characters'] = nwords
```

```
144
145     df['in']= [df['Search term'].values[n]+ (1)*' \n' for n in
            range(len(df['Search term']))]
146     df['out'] = [df['Search term'].values[n]+ (max_length-df['
            Search term'].values[n].count(' ')+1)*' \n' for n in
            range(len(df['Search term']))]
147
148
149     outputs = df["out"]
150     tokenizer = Tokenizer(filters='',lower=True, split=' ',
            char_level=False)
151     tokenizer.fit_on_texts(outputs)
152     n_tokens = len(tokenizer.word_counts) + 1
153     tokenized_list = tokenizer.texts_to_sequences(outputs)
154     max_length = max([len(tokenized_list[n]) for n in range(0,
            len(tokenized_list))])
155
156     inputs = np.array([np.array(df['in'])[n].split(' ') for n
            in range(len(np.array(df['in'])))])
157     model_w2v = gensim.models.Word2Vec(inputs,min_count=1,
            window=5)
158     model_w2v.save(str(output_dir / 'model_w2v'))
159     model_w2v = gensim.models.Word2Vec.load(str(output_dir / '
            model_w2v'))
160     w2v = model_w2v.wv
161     X0 = []
162     X0 = np.array([[w2v[np.array(outputs)[n].split(' ')[k]] for
             k in range(len(np.array(outputs)[0].split(' ')))] for n
             in range(len(outputs))])
163     words_in_sentence = len(np.array(outputs)[0].split(' '))
164     X = X0.reshape(1,len(X0),words_in_sentence,100)
165
166
167     Y = to_categorical([
168             tokenizer.texts_to_sequences(outputs)
169         ], num_classes=n_tokens)
170
171     # X_characters is the count of characters for each line
172
173     X_characters = df[['characters']].values
174
175
176
177
```

```
178             ##      TRAINING MODEL
179         training_model, lstm, lines, inputs, outputs =
                create_training_model(latent_dim, n_tokens)
180
181
182         joblib.dump([latent_dim, n_tokens, max_length, tokenizer],
                str(output_dir / ('metadata_%s-%s.pkl' % (latent_dim,
                epochs)) ))
183
184         filepath = str(output_dir / ( "%s-{epoch:02d}-{loss:.2f}-{
                val_loss:.2f}.hdf5" % latent_dim))
185
186         checkpoint = ModelCheckpoint(
187                     filepath, monitor='loss', verbose=1,
                        save_best_only=True, mode='min',period=10)
188
189         csv_logger = CSVLogger(
190                     str(output_dir / ('training_log_%s-%s.csv' % (
                        latent_dim,epochs))), append=True, separator
                        =',')
191
192
193         callbacks_list = [checkpoint, csv_logger]
194
195         training_model.fit([
196                 X[0],X_characters[:,0]], Y[0], batch_size=64,epochs
                    =epochs,
197                 validation_split=.1 ,callbacks=callbacks_list)
198
199         return lstm, lines, tokenizer, n_tokens, max_length
200
201
202     #-------------------------ENTRENAMIENTO
            ---------------------------------
203     root_path = os.path.abspath(os.path.dirname(__file__))
204     os.chdir(root_path)
205     dataframes = ["259_424_9086","437_910_0412","991_035_4076"]
206     latent_dimension = [2000,4000,6000]
207     number_epoch = [40]
208     for dataframe_name in dataframes:
209         for n in latent_dimension:
210             for e in number_epoch:
211                 latent_dim = n
212                 lstm, lines, tokenizer, n_tokens, max_length =
```

```
                    entrenar(
213                 epochs=e,latent_dim=latent_dim,root_path=root_path,
                    name=dataframe_name)
```

Módulo para generación por palabras con modelo LSTM

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Mon May 17 12:09:58 2021

@author: ina
"""


import tensorflow.compat.v1 as tf1
tf1.disable_v2_behavior()
tf_session = tf1.Session()

from tensorflow.compat.v1.keras import backend as K
K.set_session(tf_session)

from tensorflow.keras.callbacks import ModelCheckpoint,
    CSVLogger
from tensorflow.keras.layers import Add, Dense, Input, LSTM
from tensorflow.keras.models import Model

import numpy as np
import time

for o in range(10):
    try:
        from keras.utils import np_utils
        print('Works!')
        break
    except Exception as e:
        print(e,o)
        time.sleep(0.5)



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)
```

```
41      return np.argmax(probas)
42
43  class TrainingLine:
44      def __init__(self, name, previous_line, lstm, n_tokens):
45          self.char_input = Input(shape=(None, n_tokens), name='
                char_input_%s' % name)
46
47          self.numberchar_input = Input(shape=(1,), name='
                numberchar_input_%s' % name)
48          self.numberchar_dense = Dense(lstm.units, activation='
                relu', name='numberchar_dense_%s' % name)
49          self.numberchar_dense_output = self.numberchar_dense(
                self.numberchar_input)
50
51          #self.lstm = LSTM(latent_dim, return_state=True,
                return_sequences=True, name='lstm_%s' % name)
52          initial_state = [self.numberchar_dense_output, self.
                numberchar_dense_output]
53
54          self.lstm_out, self.lstm_h, self.lstm_c = lstm(self.
                char_input, initial_state=initial_state)
55
56          self.output_dense = Dense(n_tokens, activation='softmax
                ', name='output_%s' % name)
57          self.output = self.output_dense(self.lstm_out)
58
59  def create_training_model(latent_dim, n_tokens):
60      lstm = LSTM(latent_dim, return_state=True, return_sequences
            =True, name='lstm')
61      lines = []
62      inputs = []
63      outputs = []
64
65      previous_line = lines[-1] if lines else None
66      lines.append(TrainingLine('line_0', previous_line, lstm,
            n_tokens))
67      inputs += [lines[-1].char_input, lines[-1].numberchar_input
            ]
68      outputs.append(lines[-1].output)
69
70      training_model = Model(inputs, outputs)
71      training_model.compile(optimizer='rmsprop', loss='
            categorical_crossentropy',metrics=['accuracy'])
72
```

```
73      return training_model, lstm, lines, inputs, outputs
74
75  class GeneratorLine:
76      def __init__(self, name, training_line, lstm, n_tokens):
77          self.char_input = Input(
78              shape=(None, 100), name='char_input_%s' % name)
79
80          self.numberchar_input = Input(
81              shape=(1,), name='numberchar_input_%s' % name)
82          self.numberchar_dense = Dense(
83              lstm.units, activation='relu', name='
                 numberchar_dense_%s' % name)
84          self.numberchar_dense_output = self.numberchar_dense(
                self.numberchar_input)
85
86          self.h_input = Input(shape=(lstm.units,), name='
                h_input_%s' % name)
87          self.c_input = Input(shape=(lstm.units,), name='
                c_input_%s' % name)
88          initial_state = [self.h_input, self.c_input]
89
90          self.lstm = lstm
91
92          self.lstm_out, self.lstm_h, self.lstm_c = self.lstm(
93              self.char_input, initial_state=initial_state)
94
95          self.output_dense = Dense(
96              n_tokens, activation='softmax', name='output_%s' %
                 name)
97          self.output = self.output_dense(self.lstm_out)
98
99          self.numberchar_dense.set_weights(
100             training_line.numberchar_dense.get_weights())
101         self.lstm.set_weights(lstm.get_weights())
102         self.output_dense.set_weights(training_line.
                output_dense.get_weights())
103
104 class Generator:
105     def __init__(self, lstm, lines, tf_session, tokenizer,
            n_tokens, max_line_length, wv):
106         self.tf_session = tf_session
107         self.tokenizer = tokenizer
108         self.n_tokens = n_tokens
109         self.max_line_length = max_line_length
```

```
110
111         self.lstm = LSTM(
112             lstm.units, return_state=True, return_sequences=
                True,
113             name='generator_lstm'
114         )
115         self.lines = [
116             GeneratorLine(
117                 'generator_line',
118                 lines[0], self.lstm, self.n_tokens
119             )
120         ]
121         self.lstm.set_weights(lstm.get_weights())
122         self.wv = wv
123
124     def generate_kwords(self, nwords=5, temperature=.1):
125         output = []
126         h = None
127         c = None
128
129
130         # first_word = chr(int(np.random.randint(ord('a'), ord
                ('z')+1)))
131
132         # next_char = self.tokenizer.texts_to_sequences(
                first_char)[0][0]
133         next_char = np.random.randint(self.n_tokens)
134         line = self.lines[0]
135         s = self.tf_session.run(
136             line.numberchar_dense_output,
137             feed_dict={
138                 line.numberchar_input: [[nwords]]
139             }
140         )
141
142         if h is None:
143             h = s
144             c = s
145         else:
146             h = h + s
147             c = c + s
148
149         line_output = [next_char]
150         end = False
```

```
151             next_char = None
152             for i in range(self.max_line_length):
153                 char, h, c = self.tf_session.run(
154                     [line.output, line.lstm_h, line.lstm_c],
155                     feed_dict={
156                         line.char_input:
157 
158                             [self.wv[self.tokenizer.
                                sequences_to_texts([[line_output
                                [-1]]])]]
159 
160                         ,
161                         line.h_input: h,
162                         line.c_input: c
163                     }
164                 )
165 
166                 char = sample(char[0,0], temperature)
167                 if char == 1 and not end:
168                     end = True
169                 if char != 1 and end:
170                     next_char = char
171                     char = 1
172 
173                 line_output.append(char)
174 
175             cleaned_text = self.tokenizer.sequences_to_texts([
176                 line_output
177             ])[0].strip()[0:].replace(
178                 '    ', '\n'
179             ).replace('\n', ' ')
180 
181             print(cleaned_text)
182             output.append(cleaned_text)
183 
184             return output
```

Entrenamiento de modelo convolucional por palabras

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Thu Jun 17 17:13:23 2021

@author: ina
"""


import os
import gensim
from tensorflow import keras
from pathlib import Path


from tensorflow.keras.callbacks import ModelCheckpoint, CSVLogger
from tensorflow.keras.layers import Add, Dense
from tensorflow.keras.models import Model
from tensorflow.keras.preprocessing.text import Tokenizer

import numpy as np
import pandas as pd
import joblib


from tensorflow.keras.layers import  Dropout
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten
from tensorflow.keras import Sequential



import datetime
import time
for o in range(10):
    try:
        #from keras.utils import to_categorica
        from keras.utils.np_utils import to_categorical
        print('Works!')
        break
```

```
40      except Exception as e:
41          print(e,o)
42          time.sleep(0.5)
43
44
45  #------------------------------FUNCION
        ---------------------------
46  def entrenar(epochs=30, neuronas_capa=512, kernel_size = 3,
        root_path="",name="259_424_9086"):
47
48      data_path = os.path.join(root_path,"KW_ST_files", name,"
            st_report2020.csv")
49
50      output_dir = Path('%d_%d_cnn_%s' % (neuronas_capa,epochs,
            name))
51      try:
52          output_dir.mkdir()
53      except:
54          pass     # Percent of samples to use for training, might
                be necessary if you're running out of memory
55      data_path = os.path.join(root_path,"KW_ST_files", name,"
            st_report2020.csv")
56      sample_size = 1
57
58      time_file = open(output_dir / ("inicial_time_%s-%s.txt" % (
            neuronas_capa, name)),"a")
59      time_file.write(str(datetime.datetime.now().time()))
60      time_file.close()
61
62      df_raw = pd.read_csv(data_path)
63      df_raw = df_raw.sample(frac=sample_size)
64      conversions = np.array(df_raw["Conversions"])
65
66      df = pd.DataFrame()
67
68      df['Search term'] = df_raw['Search term'].drop([np.where(
            conversions<1)][0][0],axis=0)
69      max_length = int(max([pd.Series([df_raw['Search term'].
            values[n].count(' ')+1 for n in range(len(df_raw))]).
            quantile(.99)]))
70      df = pd.DataFrame(df[(df['Search term'].str.count(' ') <=
            max_length)].copy())
71      nwords = []
72      # ncharacters = [len(df['Search term'].values[i])  for i in
```

```
        range(len(df))]
    nwords = [df['Search term'].values[n].count(' ')+1 for n in
        range(len(df))]
    df['characters'] = nwords



    df['in']= [df['Search term'].values[n]+ ' \n' for n in
       range(len(df['Search term']))]


    w2v_inputs = np.array([np.array(df['in'])[n].split(' ') for
        n in range(len(np.array(df['in'])))])
    model_w2v = gensim.models.Word2Vec(w2v_inputs,min_count=1,
       window=5)
    model_w2v.save(str(output_dir / 'model_w2v'))
    model_w2v = gensim.models.Word2Vec.load(str(output_dir / '
       model_w2v'))
    w2v = model_w2v.wv
    #-------------------------

    tokenizer = Tokenizer(filters='',lower=True, split=' ',
       char_level=False)
    tokenizer.fit_on_texts(df['Search term'])
    tokenizer.fit_on_texts('\n')
    search = tokenizer.texts_to_sequences(df['Search term'])
    n_tokens = len(tokenizer.word_counts)+1



    inp_list = []
    out_list = []
    for i in range(len(search)):
        for j in range(1,max_length+1):
            try:
                search[i][j]
            except:
                search[i].append(tokenizer.texts_to_sequences('
                   \n')[0][0])
            inp_list.append(search[i][0:j])
            out_list.append(search[i][j])
    inputs = pd.DataFrame(inp_list)
    outputs = pd.DataFrame(out_list)
```

```
109         inputs = inputs.fillna(int(tokenizer.texts_to_sequences('\n
                ')[0][0]))
110 
111 
112         X = np.array([[w2v[tokenizer.sequences_to_texts([[inputs[k
                ][n]]])]for k in range(max_length)]for n in range(len(
                inputs))])
113         Y = to_categorical(outputs, num_classes=n_tokens)
114         X = X.reshape(len(X),max_length,100,1)
115 
116         model = Sequential()
117             #add model layers
118         model.add(Conv2D(neuronas_capa, kernel_size=kernel_size,
                activation='relu', input_shape=(max_length,100,1)))
119         model.add(MaxPooling2D(pool_size = (2,2)))
120         model.add(Dropout(0.25))
121 
122         model.add(Conv2D(neuronas_capa, kernel_size=kernel_size,
                activation='relu',padding='same'))
123         model.add(MaxPooling2D(pool_size = (2,2)))
124         model.add(Dropout(0.25))
125 
126         model.add(Conv2D(neuronas_capa, kernel_size=kernel_size,
                activation='relu',padding='same'))
127         model.add(MaxPooling2D(pool_size = (2,2)))
128         model.add(Dropout(0.25))
129 
130         model.add(Conv2D(neuronas_capa, kernel_size=kernel_size,
                activation='relu',padding='same'))
131         model.add(Flatten())
132         model.add(Dense(n_tokens, activation='softmax'))
133 
134 
135         opt = keras.optimizers.RMSprop(learning_rate=0.0001)
136         model.compile(loss='categorical_crossentropy', optimizer=
                opt, metrics=['categorical_accuracy','accuracy'])
137 
138 
139 
140         joblib.dump([neuronas_capa, n_tokens, max_length, tokenizer
                ], str(output_dir / ('metadata_%s-%s.pkl' % (
                neuronas_capa, name)) ))
141 
142         filepath = str(output_dir / ( "%s-{epoch:02d}-{loss:.2f}-{
```

```
        val_loss:.2f}.hdf5" % neuronas_capa))

    checkpoint = ModelCheckpoint(
                filepath, monitor='loss', verbose=1,
                    save_best_only=True, mode='min',period=10)

    csv_logger = CSVLogger(
        str(output_dir / ('training_log_%s-%s.csv' % (
            neuronas_capa, name))), append=True, separator=',')

    callbacks_list = [checkpoint, csv_logger]


    history = model.fit(
            X, Y, batch_size=64,epochs=epochs,
            validation_split=.1, callbacks=callbacks_list)
    return model, history
#---------------ENTRENAMIENTO
    ---------------------------------------
root_path = os.path.abspath(os.path.dirname(__file__))
os.chdir(root_path)
dataframes = ["259_424_9086","437_910_0412","991_035_4076"]
neuronas_capa = [64,128,256,512]
number_epoch = [40]
for dataframe_name in dataframes:
    for n in neuronas_capa:
        for e in number_epoch:
            model, history = entrenar(
                epochs=e, neuronas_capa=n, kernel_size = 3,
                    root_path=root_path,name=dataframe_name)
```

Módulo para generación por palabras con modelo convolucional

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Mon May 17 12:09:58 2021

@author: ina
"""


import tensorflow.compat.v1 as tf1 #new
tf1.disable_v2_behavior() #new
tf_session = tf1.Session()

from tensorflow.compat.v1.keras import backend as K #new
# from keras import backend as K
K.set_session(tf_session)


import numpy as np
import time

for o in range(10):
    try:
        from keras.utils import np_utils
        print('Works!')
        break
    except Exception as e:
        print(e,o)
        time.sleep(0.5)

def sample(preds, temperature=1.0):
    # helper function to sample an index from a probability
       array
    # From https://github.com/llSourcell/keras_explained/blob/
       master/gentext.py
    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)
```

```
40
41  class Generator:
42      def __init__(self, tokenizer, n_tokens, max_line_length,
            model, wv):
43          self.tokenizer = tokenizer
44          self.n_tokens = n_tokens
45          self.max_line_length = max_line_length
46          self.model = model
47          self.wv = wv
48      def generate_kword(self, temperature=.1, first_word=None,
            kw_length=None):
49          encoded_output = []
50          if first_word is None:
51              first_word = np.random.randint(self.n_tokens)
52          if kw_length is None:
53              kw_length = self.max_line_length-2
54
55
56          encoded_output.append(first_word)
57          end = False
58          space = np_utils.to_categorical(
59              self.tokenizer.texts_to_sequences('\n'),num_classes
                    =self.n_tokens).reshape(self.n_tokens,1)
60
61          space_w2v = self.wv['\n']
62
63          out = np.repeat(
64              space_w2v.reshape(1,100),self.max_line_length,axis
                    =0).reshape(
65                  1,self.max_line_length,100,1)
66          out[0][0] = np.array(self.wv[first_word]).reshape
                (100,1)
67
68          for i in range(1,kw_length):
69
70              new_out = sample(self.model.predict(out)[0],
                    temperature)
71              encoded_output.append(new_out)
72              out[0][i] = np.array(self.wv[self.tokenizer.
                    sequences_to_texts([[new_out]])[0]]).reshape
                    (100,1)
73              if np.array_equal(space,out[0][i]) and np.
                    array_equal(space,out[0][i-1]) and not end:
74                  end = True
```

```
75                if np.array_equal(space,out[0][i]) and end:
76                    break
77            if not np.array_equal(space,out[0][i]) :
78                for j in range(1,2):
79                    new_out = sample(self.model.predict(out)[0],
                          temperature)
80                    encoded_output.append(new_out)
81                    out[0][i+j] = np.array(self.wv[self.tokenizer.
                          sequences_to_texts([[new_out]])[0]]).reshape
                          (100,1)
82                    if np.array_equal(space,out[0][i+j]):
83                        break
84
85
86            output_raw = self.tokenizer.sequences_to_texts([
                  encoded_output])
87            output = output_raw[0].strip()[0:].replace('    ', '\n')
                  .replace('\n', ' ')
88            return output
```