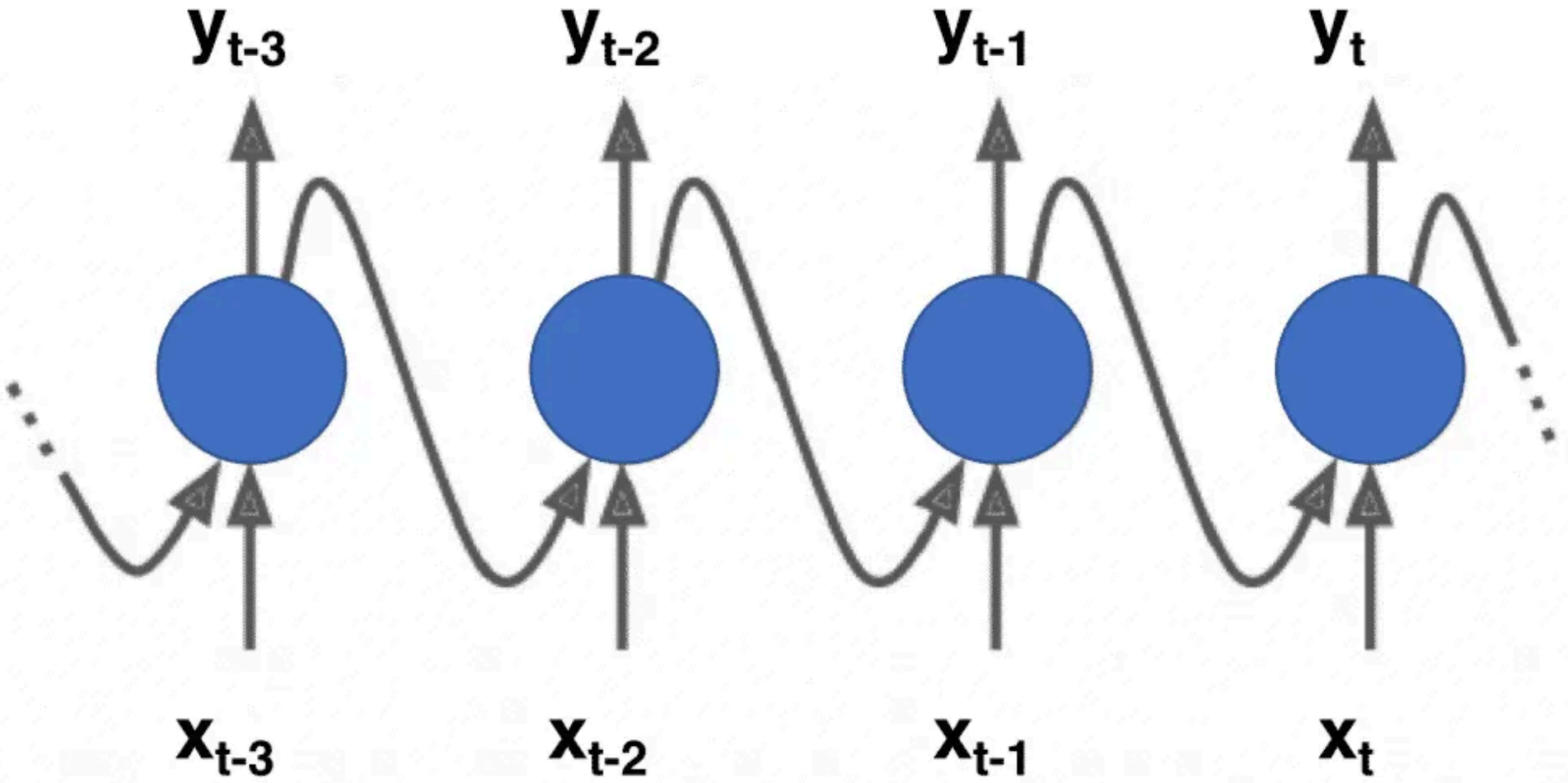


Redes Neuronales Recurrentes

22/09/2019

> ***Contenido abierto del libro DEEP LEARNING Introducción práctica con Keras***
>
> *Este post contiene el capítulo 7 del libro "Deep Learning – Introducción práctica con Keras (SEGUNDA PARTE)" de Kindle Direct Publishing con ISBN 978-1-687-47399-8 en la colección WATCH THIS SPACE – Barcelona (Book 6).*

Nota: En el proceso semiautomático de generación de esta versión HTML a partir de la versión del libro en papel, se han perdido algunos formatos de fuente de texto (por ej. diferenciar cuando nos referimos a código) o se han eliminado espacios en blanco. A pesar de esta «falta de elegancia» en el texto, el resultado es correcto y permite aprender sin problemas de esta versión HTML.

En los capítulos anteriores hemos presentado cómo podemos usar *Deep Learning*sobre datos de tipo imagen para aplicarlos a problemas de visión por computador, uno de los ámbitos más activos en inteligencia artificial. Pero hay otros ámbitos, como el de procesado de lenguaje natural[1] (*Natural Language Processing*, NLP), donde hay también grandes avances para poder solucionar problemas de compresión de texto. En este capítulo trataremos un ejemplo con datos de tipo texto para explicar las redes neuronales recurrentes.

Las redes neuronales recurrentes, o *Recurrent Neural Networks*(RNN) en inglés, son una clase de redes para analizar datos de

Siguiendo el carácter práctico de este libro, después de presentar los mínimos conceptos básicos de RNN requeridos para entender su potencial, en este capítulo nos centraremos en seguir un caso práctico que crea un sencillo modelo que lee y aprende de textos de un autor y luego intenta generar un texto nuevo que pretende parecer que hubiera sido escrito por el mismo autor.

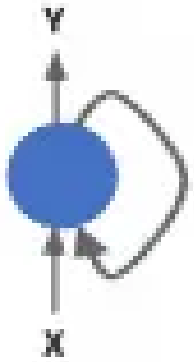
Conceptos básicos de redes neuronal recurrente

Las redes neuronales recurrentes (RNN) fueron ya concebidas en la década de 1980. Pero estas redes han sido muy difíciles de entrenar por sus requerimientos en computación y hasta la llegada de los avances de estos últimos años, que presentábamos al principio del libro, no se han vuelto más accesibles y popularizado su uso por la industria.

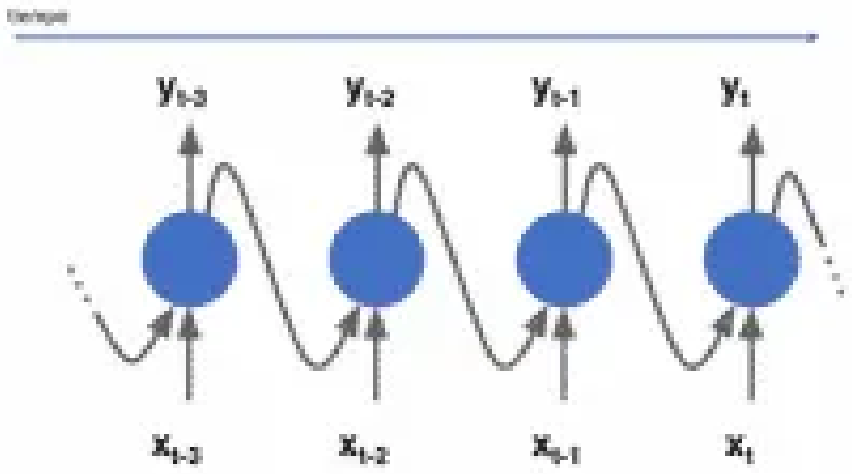
Neurona recurrente

Hasta ahora hemos visto redes cuya función de activación solo actúa en una dirección, hacia delante, desde la capa de entrada hacia la capa de salida, es decir, que no recuerdan valores previos. Una red RNN es parecida, pero incluye conexiones que apuntan “hacia atrás”, una especie de retroalimentaciones entre las neuronas dentro de las capas.

Imaginemos la RNN más simple posible, compuesta por una sola neurona que recibe una entrada, produciendo una salida, y enviando esa salida a sí misma, como se muestra en la siguiente figura:



En cada instante de tiempo (también llamado *timestep* en este contexto), esta neurona recurrente recibe la entrada xde la capa anterior, así como su propia salida del instante de tiempo anterior para generar su salida y. Podemos representar visualmente esta pequeña red desplegada en el eje del tiempo como se muestra en la figura:



Siguiendo esta misma idea, una capa de neuronas recurrentes se puede implementar de tal manera que, en cada instante de

Ahora cada neurona recurrente tienen dos conjuntos de parámetros, uno que lo aplica a la entrada de datos que recibe de la capa anterior y otro conjunto que lo aplica a la entrada de datos correspondiente al vector salida del instante anterior. Sin entrar demasiado en formulación, y siguiendo la notación explicada en la primera parte del libro, podríamos expresarlo de la siguiente manera:

$$y_t = f\,(Wx_t + Uy_{t-1} + b)$$

Donde $x=(x_1,\ldots,x_T)$ representa la secuencia de entrada proveniente de la capa anterior, W los pesos de la matriz y $b$el bias vistos ya en las anteriores capas. Las RNN extienden esta función con una conexión recurrente en el tiempo donde Ues la matriz de pesos que opera sobre el estado de la red en el instante de tiempo anterior (y_{t-1}) anterior. Ahora, en la fase de entrenamiento a través del *Backpropagation*también se actualizan los pesos de esta matriz.

Memory cell

Dado que la salida de una neurona recurrente en un instante de tiempo determinado es una función de entradas de los instantes de tiempo anteriores, se podría decir que una neurona recurrente tiene en cierta forma memoria. La parte de una red neuronal que preserva un estado a través del tiempo se suele llamar *memory cell*(o simplemente *cell*).

Y precisamente esta “memoria interna” es lo que hace de este tipo de redes muy adecuadas para problemas de aprendizaje automático que involucran datos secuenciales. Gracias a su memoria interna, las RNN pueden recordar información relevante sobre la entrada que recibieron, lo que les permite ser más precisas en la predicción de lo que vendrá después manteniendo información de contexto a diferencia de los otros tipos de redes que hemos visto, que no pueden recordar acerca de lo que ha sucedido en el pasado, excepto lo reflejado en su entrenamiento a través de sus pesos.

Proporcionar modelos con memoria y permitirles modelar la evolución temporal de las señales es un factor clave en muchas

temporal que conecta los datos a menudo es más importante que el contenido espacial (de los píxeles) de cada dato (imagen) individual.

Para ilustrar el concepto de “memoria” de una RNN, imaginemos que tenemos una red neuronal como las vistas en capítulos anteriores, le pasamos la palabra «neurona» como entrada y esta red procesa la palabra carácter a carácter. En el momento en que alcanza el carácter «r», ya se ha olvidado de «n», «e» y «u», lo que hace que sea casi imposible para la red neuronal predecir qué letra vendrá después. Pero en cambio, una RNN permite recordar precisamente esto. Conceptualmente, la RNN tiene como entradas el presente y el pasado reciente. Esto es importante porque la secuencia de datos contiene información crucial para saber lo que viene a continuación.

Backpropagation a través del tiempo

Recordemos que en las redes neuronales presentadas anteriormente, básicamente se hace *Forward-Propagation*para obtener el resultado de aplicar el modelo y verificar si este resultado es correcto o incorrecto para obtener la *Loss*. Después se hace *Backward-Propagation*(o *Backpropagation*) que recordemos que no es otra cosa que ir hacia atrás a través de la red neuronal para encontrar las derivadas parciales del error con respecto a los pesos de las neuronas. Esas derivadas son utilizadas por el algoritmo *Gradient Descent*para minimizar iterativamente una función dada, ajustando los pesos hacia arriba o hacia abajo, dependiendo de como se disminuye la *Loss*.

Entonces, con *Backpropagation*básicamente se intenta ajustar los pesos de nuestro modelo mientras se entrena. Dado el carácter introductorio del libro no entraremos en formalizaciones, pero nos gustaría que el lector pudiera intuir cómo se realiza el *Backpropagation*en una RNN, lo que se llama *Backpropagation Through Time (*BPTT*)*. El desenrollar es una herramienta conceptual y de visualización que nos puede ayudar a comprender cómo puede conseguirse realizar el *Backpropagation*pero incluyendo la dimensión “tiempo”.

desenrollar se observa por qué se puede considerar una RNN como una secuencia de redes neuronales en la que se puede realizar un *Backpropagation*relativamente equivalente al que conocíamos.

Al realizar el proceso de BPTT, se requiere a nivel matemático incluir la conceptualización de desenrollar, ya que la *Loss*de un determinado instante de tiempo depende del instante (*timestep*) anterior. Dentro de BPTT, el error es propagado hacia atrás desde el último hasta el primer instante de tiempo, mientras se desenrollan todos los instantes de tiempo. Esto permite calcular la *Loss*para cada instante de tiempo, lo que permite actualizar los pesos. Pero el lector ya intuye que el grafo no cíclico que resulta del desplegado en el tiempo es enorme y poder realizar el BPTT es computacionalmente costoso.

Exploding Gradients y *Vanishing Gradients*

Dos cuestiones importantes que afectan a las RNN (aunque afecta en general a cualquier tipo de red muy grande en números de parámetros sea o no sea recurrente) son *Exploding Gradients*y *Vanishing Gradients*. No pretendemos entrar en detalle dado el carácter introductorio del libro, pero consideramos adecuado mencionarlos para que el lector entienda la problemática dado el impacto que han tenido ambos en el desarrollo de extensiones actuales de RNN.

Recordemos que un gradiente es una derivada parcial con respecto a sus entradas que mide cuánto cambia la salida de una función al cambiar las entradas un poco, por decirlo en un lenguaje lo más general posible. También decíamos que se puede ver como la pendiente de una función en un punto, que cuanto más alto es el gradiente, más pronunciada es la pendiente y más rápido puede aprender un modelo. Pero si la pendiente es cero, el modelo se detiene en el proceso de aprender.

En resumen, el gradiente indica el cambio a realizar en todos los pesos con respecto al cambio en el error. Hablamos de «gradientes explosivos» o *Exploding Gradients*en inglés cuando el algoritmo asigna una importancia exageradamente alta a los pesos, sin mucha razón y esto genera un problema en el entrenamiento. En este caso el problema se puede resolver fácilmente si se truncan o reducen los gradientes.

Hablamos de «gradientes desaparecidos» o *Vanishing Gradients*cuando los valores de un gradiente son demasiado pequeños y el modelo deja de aprender o requiere demasiado tiempo debido a ello. Este fue un problema importante en la década de 1990 y mucho más difícil de resolver que los *Exploding Gradients*. Afortunadamente, se resolvió mediante el concepto de *gate units*(puertas) que introducimos a continuación.

Long-Short Term Memory

Long-Short Term Memory(LSTM) son una extensión de las redes neuronales recurrentes, que básicamente amplían su memoria para aprender de experiencias importantes que han pasado hace mucho tiempo. Las LSTM permiten a las RNN recordar sus entradas durante un largo período de tiempo. Esto se debe a que LSTM contiene su información en la memoria, que puede considerarse similar a la memoria de un ordenador , en el sentido que una neurona de una LSTM puede leer, escribir y borrar información de su memoria.

Esta memoria se puede ver como una «celda» bloqueada, donde «bloqueada» significa que la célula decide si almacenar o eliminar información dentro (abriendo la puerta o no para almacenar), en función de la importancia que asigna a la información que está recibiendo. La asignación de importancia se decide a través de los pesos, que también se aprenden mediante el algoritmo. Esto lo podemos ver como que aprende con el tiempo qué información es importante y cuál no.

En una neurona LSTM hay tres puertas a estas «celdas» de información: puerta de entrada (*input gate*), puerta de olvidar (*forget gate*) y puerta de salida (*output gate*). Estas puertas determinan si se permite o no una nueva entrada, se elimina la información porque no es importante o se deja que afecte a la salida en el paso de tiempo actual.

Las puertas en una LSTM son análogas a una forma sigmoide, lo que significa que van de 0 a 1 en la forma que hemos visto en capítulos anteriores. El hecho de que sean análogas a una función de activación *sigmoid*como las vistas anteriormente, permite incorporarlas (matemáticamente hablando) al proceso de *Backpropagation*. Como ya hemos comentado, los problemas de los *Vanishing Gradients*se resuelven a través de LSTM porque mantiene los gradientes lo suficientemente empinados y, por lo tanto, el entrenamiento es relativamente corto y la precisión alta.

Keras ofrece también otras implementaciones de RNN como es la *Gated Recurrent Unit*(GRU)[5]. Las capas GRU aparecieron en el 2014, y usan el mismo principio que LSTM, pero están simplificadas de manera que su rendimiento está a la par con LSTM pero computacionalmente son más eficiente.

Sin duda el tema es muy extenso y profundo, pero creemos que con esta breve introducción se puede seguir el caso práctico que presentamos a continuación y con el que el lector aprenderá a usar redes RNN.

Caso de estudio: generación de texto

Antes de empezar a usar el teclado de nuestro ordenador para aprender con un caso práctico, vamos a presentar un ejemplo que creemos que es muy adecuado para adentrarse por primera vez en este apasionante (y extenso) mundo de las RNN.

Este caso de estudio trata de generar texto usando una RNN basada en caracteres y de esta manera también podemos usar el caso al mismo tiempo para mostrar el uso de datos de texto. En este ejemplo se entrena un modelo de red neuronal para predecir el siguiente carácter a partir de una secuencia de carácteres. Con este modelo intencionadamente simple, para mantener el carácter pedagógico del ejemplo, se consigue generar secuencias de texto más largas llamando al modelo repetidamente.

Datos de tipo texto y redes neuronales

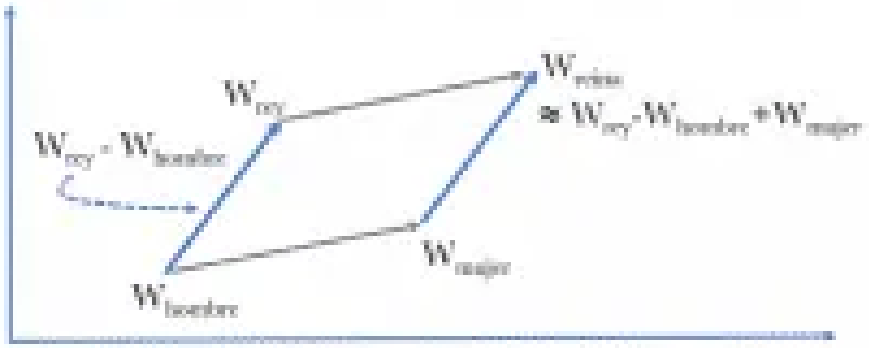
Los modelos para NLP se entrenan a partir de un corpus lingüístico, un conjunto amplio y estructurado de ejemplos reales de uso de la lengua. En cuanto a su estructura, variedad y complejidad, un corpus debe reflejar la modalidad de la lengua de la forma más exacta posible. La idea es que representen al lenguaje de la mejor forma posible para que los modelos de NLP puedan aprender los patrones necesarios para entender el lenguaje.

Pero previamente recordemos que todas las entradas en una red neuronal deben ser tensores de datos numéricos. Cualquier dato que se necesite procesar (sonido, imágenes, texto) primero debe ser convertido en un tensor numérico, un paso llamado

en el ejemplo de esta sección). En redes neuronales se usan dos tipos principales de vectorización: *One-hot Encoding***[6]**y *WordEmbedding***[7]**.

De manera breve podríamos describir la técnica *One-hot Encoding*como el proceso de asociar un índice único para cada palabra y después transformar este índice en un vector binario de tamaño igual a la del vocabulario, que en el caso de datos tipo texto, todo son ceros excepto en la posición correspondiente al índice de la palabra. Aunque *One-hot Encoding*es simple, hay puntos débiles, por ejemplo, el tamaño que pueden llegar a tener los vectores si el vocabulario del corpus usado es muy grande.

La solución a este problema es utilizar la otra técnica *Word Embedding*, que en vez de crear vectores dispersos de gran tamaño, crea vectores en un espacio de menor dimensión pero que preserva las relaciones semánticas[8], un detalle muy importante. Un ejemplo famoso que se usa para mostrar de manera simple el potencial de *Word Embedding*para explotar las relaciones semánticas es Word2Vec[9](un algoritmo concreto de *Word Embedding*), donde moviéndonos en ciertas dimensiones podemos descubrir relaciones entre ciertas palabras, por ejemplo el género. Fijémonos en la siguiente figura:



En ella se muestra que la relación algebraica de las palabras tiene un sentido semántico, una “álgebra de palabras”, por decirlo de alguna manera.Es decir, resulta que el *Word Embedding*correspondiente a *reina*es el más cercano al resultado de calcular

Mientras que los vectores obtenidos a través de la codificación *One-hot Encoding*son binarios, dispersos (en su mayoría sus elementos de ceros) y de gran tamaño (el mismo que la cantidad de palabras en el vocabulario), los vectores obtenidos con la codificación con la técnica *Word Embeddings*son vectores de menor tamaño y más densos (es decir que no tienen mayoritariamente zeros).

Estos vectores que representan a las palabras codificadas con *Word Embeddings*pueden ser obtenidos a la vez que se entrena la red neuronal (empiezan con vectores aleatorios y luego se aprenden de la misma manera que se aprenden los pesos de una red neuronal). Pero también se pueden incorporar en el modelo estos vectores con valores ya preentrenados y no ser entrenado. En Keras la vectorización se puede incorporar al modelo como una capa inicial de la red neuronal usando tf.keras.layers.Embedding[10].

Debemos dejar aquí este tema dado el carácter introductorio del libro, pero quisiera hacer notar al lector su importancia en estos momentos en el área de NLP. Recordemos que con el *Transfer Learning*aprovechamos conocimiento adquirido anteriormente y en el área de la visión por computador tenemos excelentes conjuntos de modelos pre-entrenados (como ya vimos en anteriores capítulos). En el caso del procesamiento del lenguaje natural, últimamente se han hecho grandes avances en *Transfer Learning*gracias precisamente a las técnicas de vectorización aquí presentadas.

Y antes de acabar con el tema, otro detalle importante. Como hemos dicho, podemos aplicar el *Deep Learning*a la NLP mediante la representación de las palabras como vectores en un espacio continuo, de baja dimensión gracias a la técnica *Word Embeddings*. En este caso, cada palabra tenía un solo vector, independientemente del contexto en el que aparecía la palabra en el texto. Pero esto plantea problemas con palabras polisémicas, por ejemplo, en las cuales todos los significados de una palabra tienen que compartir la misma representación de vector. Trabajos recientes han creado con éxito representaciones de palabras contextualizadas, es decir, vectores de palabras que son sensibles al contexto en el que aparecen.

En resumen, el desarrollo de modelos preentrenados ha surgido recientemente como un paradigma estándar en la práctica del *Deep Learning*para el procesamiento del lenguaje natural con ejemplos de modelos entrenados como BERT[11], GPT-2[12],

Character–Level Language Models

Para intentar buscar un ejemplo lo más simple posible en el que podamos aplicar una red neuronal recurrente, hemos considerado usar el ejemplo de "*Character level language model*" propuesto por Andrej Karpathy[15]en su artículo *The Unreasonable Effectiveness of Recurrent Neural Networks*[16](y parcialmente basado en su implementado en el tutorial *Generate text with an RNN*de la web de TensorFlow[17]).

En realidad, se trata de uno de los modelos pioneros en procesado de texto a nivel de carácter llamado char-rnn[18]. Consiste en darle a la RNN una palabra y se le pide que modele la distribución de probabilidad del siguiente carácter que le correspondería a la secuencia de caracteres anteriores. Con este modelo, si lo llamamos repetitivamente, podremos generar texto carácter a carácter.

Como ejemplo, supongamos que solo tenemos un vocabulario de cuatro letras posibles [«a»,»h»,»l»,»o»], y queremos entrenar a una RNN en la secuencia de entrenamiento «hola». Esta secuencia de entrenamiento es, de hecho, una fuente de 3 ejemplos de entrenamiento por separado: La probabilidad de «o» debería ser verosímil dada el contexto de «h», «l» debería ser verosímil en el contexto de «ho», y finalmente «a» debería ser también verosímil dado el contexto de «hol".

Para usar el modelo, introducimos un carácter en la RNN y obtenemos una distribución sobre qué carácter probablemente será el siguiente. Tomamos una muestra de esta distribución y la retroalimentamos para obtener el siguiente carácter. ¡Repetimos este proceso y estamos generando texto!

Caso de estudio: Primera parte de este libro

Para poder ser manejable el caso de estudio a nivel explicativo en este texto, consideraremos un ejemplo muy sencillo y limitado, tanto en datos como de modelo, con el único proposito pedagógico de entender los conceptos básicos, sin poner el foco en la calidad de los resultados del modelo. Para este propósito como *dataset*usaremos la primera parte de este libro (en texto plano que el lector puede encontrar en el GitHub de este libro[19]).

Incluso siendo un *dataset*extremadamente limitado para poder ser considerado un corpus real nos sirve para generar como salida unas oraciones donde, aunque no tienen gramaticalmente demasiado sentido, se pueden apreciar que en algunos casos la estructura del texto de salida se asemeja a una frase real. Y esto teniendo en cuenta que cuando comenzó el entrenamiento, el modelo no sabía ni deletrear una palabra, y se ha entrenado en este *dataset* realmente minúsculo que además es un texto confuso al pasarlo a texto plano (al crear un corpus con la mezclar datos y código como es el caso de este libro).

Por tanto, el lector puede extrapolar la potencia de esta tecnología cuando se ponen a trabajar modelos muy complejos con ingentes cantidades de datos, eso sí, requiriendo una capacidad de computación solo al alcance de unos pocos.

Implementación en Keras

En este apartado vamos a introducir el código en Keras que implementa el caso de estudio que nos ocupa y que el lector podrá probar por su cuenta. Como hemos hecho en anteriores capítulos, iremos explicando el código línea a línea y proponemos al lector que vaya ejecutando al mismo tiempo el código que puede encontrar en el GitHub del libro.

Descarga y preprocesado de los datos

El primer paso en este ejemplo será el descargar y preparar el conjunto de datos con el que entrenaremos nuestra red neuronal:

```
path_to_file2 = tf.keras.utils.get_file('2-Introduccion-practica-con-Keras-1a.txt',
'https://raw.githubusercontent.com/jorditorresBCN/Deep-Learning-Introduccion-practica-
con-Keras/master/Deeplearning-Introduccion-practica-con-Keras-PRIMERA-PARTE.txt')

text = open(path_to_file2, 'rb').read().decode(encoding='utf-8')
print('Longitud del texto:        {} carácteres'.format(len(text)))

vocab = sorted(set(text))

print ('El texto está compuesto de estos {} carácteres:'.format(len(vocab)))
print (vocab)
```

```
Longitud del texto:        207119 carácteres
El texto está compuesto de estos 93 carácteres:
['\t', '\n', '\x', ' ', '!', '"', '$', '%', '&', '(', ')', '*', '+', ',', '-', '.
', '/', '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', ':', ';', '<', '=', '>'
, '?', '@', 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N',
'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', '[', ']', '_', 'a', 'b', '
c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's
', 't', 'u', 'v', 'w', 'x', 'y', 'z', '\xad', 'á', 'é', 'í', 'ó']
```

Como estamos tratando el caso de estudio a nivel de carácter, podríamos considerar que aquí el corpus son los caracteres, por tanto un corpus muy pequeño.

Recordemos que las redes neuronales solo procesan valores numéricos, no letras, por tanto tenemos que traducir los caracteres a representación numérica. Para ello crearemos dos “tablas de traducción”: una de caracteres a números y otra de números a caracteres:

```
char2idx = {u:i for i, u in enumerate(vocab)}
idx2char = np.array(vocab)
```

Ahora tenemos una representación de entero (*integer*) para cada carácter que podemos ver ejecutando el siguiente código:

```
for char,_ in sipchar2ida, range(len(vocab)):
    print(' [%s]: [%d],'.format(repr(char), char2ida[char]))
```

```
' ': 32,
'!': 33,
'"': 34,
'#': 35,
'$': 36,
'%': 37,
'&': 38,
"'": 39,
'(': 40,
')': 41,
'*': 42,
'+': 43,
',': 44,
'-': 45,
'.': 46,
'/': 47,
'0': 48,
'1': 49,
'2': 50,
'3': 51,
'4': 52,
'5': 53,
'6': 54,
'7': 55,
'8': 56,
'9': 57,
':': 58,
';': 59,
'?': 60,
'_': 61,
'a': 62,
'b': 63,
'c': 64,
'd': 65,
'e': 66,
'f': 67,
'g': 68,
'h': 69,
'i': 70,
'j': 71,
'k': 72,
'l': 73,
'm': 74,
'n': 75,
'o': 76,
'p': 77,
'q': 78,
'r': 79,
's': 80,
't': 81,
'u': 82,
'v': 83,
'w': 84,
'x': 85,
'y': 86,
'z': 87,
'\xa0': 88,
'¡': 89,
'¿': 90,
'á': 91,
'é': 92,
```

```
text_as_int = np.array([char2idx[c] for c in text])
```

Para comprobarlo podemos mostrar los 50 primeros caracteres del texto contenido en el tensor text_as_int:

```
print ('texto: {}'.format(repr(text[:50])))
print ('{}'.format(repr(text_as_int[:50])))
```

```
texto: '\r\nDeep Learning\r\nIntroduccion practica con Keras\r\n'
array([ 3,  1, 37, 66, 66, 77,  3, 45, 66, 62, 70, 75, 78, 75, 68,  2,  1,
       42, 75, 81, 79, 76, 65, 82, 64, 64, 70, 76, 75,  3, 77, 79, 62, 64,
       81, 75, 64, 62,  3, 64, 76, 75,  3, 64, 66, 78, 63, 88,  2,  1])
```

Preparación de los datos para ser usados por la RNN

Para entrenar el modelo prepararemos unas secuencias de caracteres como entradas y salida de un tamaño determinado. En nuestro ejemplo hemos definido el tamaño de 100 caracteres con la variable seq_length (que el lector puede probar de modificar por su cuenta).

Empezamos dividiendo el texto que tenemos en secuencias deseq_length+1de caracteres con las cuales luego contruiremos los datos de entrenamiento compuestos por las entradas de seq_lengthcaracteres y las salidas correspondientes que contienen la misma longitud de texto, excepto que se desplaza un carácter a la derecha. Volviendo al ejemplo de “Hola” anterior, y suponiendo un seq_length=3, la secuencia de entrada será «Hol», y la de salida será «ola».

secuencias de seq_length+1 de índice de caracteres:

```
char_dataset = tf.data.Dataset.from_tensor_slices(text_as_int)

seq_length = 100

sequences = char_dataset.batch(seq_length+1, drop_remainder=True)
```

Podemos comprobar que sequencescontiene el texto dividido en paquetes de 101 caracteres como esperamos (por ejemplo mostremos las 10 primeras secuencias):

```
for item in sequences.take(10):
  print(repr(''.join(idx2char[item.numpy()])))
```

```
'Prólogo\r\nEn 1013, Isaac Asimov publicó Segunda Fundación, el tercer libro de la saga de la Fundación '
"(a el desconocemos aquí otras fuentes, este es un tema de debate). En Segunda Fundación aparece por"
' primera vez Arkady Darell, una de los principales personajes de la parte final de la saga. En su pri'
'mera aventa, Arkady, que tiene 14 años, está haciendo sus tareas escolares. En concreto, una redacció'
'n que lleva por título El futuro del Plan Shalmont. Para hacer la redacción, Arkady está utilizando '
'un Transcriptor\, un dispositivo que convierte su voz en palabras escritas. Este tipo de dispositivo,'
' que para Isaac Asimov era ciencia ficción en 1953, lo tenemos al alcance de la mano en la mayoría de'
' nuestros smartphones, y el Deep Learning es uno de los responsables de que ya tengamos este tipo de '
'aplicaciones, siendo la tecnología esto de ellos.En la actualidad disponemos de GPUs (Graphics Proces'
'sor Units), que solo cuestan alrededor de 100 euros, que estarían en la línea del Teplón hace unos po'
```

De esta secuencia se obtiene el conjunto de datos de training que contenga tanto los datos de entrada (desde la posición 0 a la 99) como los datos de salida (desde la posición 1 a la 100). Para ello se crea una función que realiza esta tarea y se aplica a todas las secuencias usando el método map( )de la siguiente forma:

```
def split_input_target(chunk):
    input_text = chunk[:-1]
    target_text = chunk[1:]
    return input_text, target_text

dataset = sequences.map(split_input_target)
```

En este punto, datasetcontiene un conjunto de parejas de secuencias de texto (con la representación numérica de los caracteres), donde el primer componente de la pareja contiene un paquete con una secuencia de 100 caracteres del texto original y la segunda su correspondiente salida, también de 100 caracteres. Podemos comprobarlo visualizándolo por pantalla (por ejemplo mostrando la primera pareja):

```
for input_example, target_example in  dataset.take(1):
  print ('Input data: ', repr(''.join(idx2char[input_example.numpy()])))
  print ('Target data:', repr(''.join(idx2char[target_example.numpy()])))
```

```
Input data:  'Prologo\r\nEn 1953, Isaac Asimov publico Segunda Fundacion, el terc
er libro de la saga de la Fundacion'
Target data: 'rologo\r\nEn 1953, Isaac Asimov publico Segunda Fundacion, el terce
r libro de la saga de la Fundacion '
```

En este punto del código disponemos de los datos de entrenamiento en el tensor dataseten forma de parejas de secuencias de 100 *integers*de 64 bits que representan un carácter del vocabulario:

```
print(dataset)
```

En realidad los datos ya están preprocesados en el formato que se requiere para ser usados en el entreno de la red neuronal, pero recordemos que en redes neuronales los datos se agrupan en *batches* antes de pasarlos al modelo. En nuestro caso hemos decidido un tamaño de *batch* de 64, que nos facilita la explicación, pero como recordará el lector del capítulo 3, este es un hiperparámetro importante de ajustar correctamente teniendo en cuenta diferentes factores, como el tamaño de la memoria disponible, por poner un ejemplo. En este código, para crear los *batches* de parejas de secuencias hemos considerado usar tf.data que además nos permite barajar[20] las secuencias previamente:

Recapitulando, ahora en el tensor dataset disponemos de los datos de entrenamiento ya listos para ser usados para entrenar el modelo: *batches* compuestos de 64 parejas de secuencias de 100 *integers* de 64 bits que representan el carácter correspondiente en el vocabulario.

Construcción del modelo RNN

Para construir el modelo usaremos tf.keras.Sequential que ya conocemos. Usaremos una versión mínima de RNN para facilitar la explicación, que contenga solo una capa LSTM. En concreto definimos una red de solo 3 capas:

```
def build_model(vocab_size, embedding_dim, rnn_units, batch_size):
  model = tf.keras.Sequential([
    tf.keras.layers.Embedding(vocab_size, embedding_dim,
                              batch_input_shape=[batch_size, None]),
    tf.keras.layers.LSTM(rnn_units,
                         return_sequences=True,
                         stateful=True,
                         recurrent_initializer='glorot_uniform'),
    tf.keras.layers.Dense(vocab_size)
  ])
  return model


vocab_size = len(vocab)
embedding_dim = 256
rnn_units = 1024



model = build_model(
  vocab_size = vocab_size,
  embedding_dim=embedding_dim,
  rnn_units=rnn_units,
  batch_size=BATCH_SIZE)
```

La primera capa es de tipo *Word Embedding*como las que antes hemos presentado muy brevemente que mapea cada carácter de entrada en un vector *Embedding*. Esta capa tf.keras.layers.Embeddingpermite especificar varios argumentos que se pueden consultar en todo detalle en el manual de TensorFlow[21]

En nuestro caso el primero que especificamos es el tamaño del vocabulario, indicado con el argumento vocab_size, que indica cuantos vectores *Embedding*tendrá la capa. A continuación indicamos las dimensiones de estos vectores *Embedding*mediante el argumento embedding_dim, que en nuestro caso hemos decidido que sea 256. Finalmente se indica el tamaño del *batch*que usaremos para entrenar, en nuestro caso 64.

La segunda capa es de tipo LSTM introducida anteriormente en este capítulo. Esta capa tf.keras.layers.LSTMtiene varios argumentos posibles que se pueden consultar en el manual de TensorFlow[22], aquí solo usaremos algunos y dejamos los valores por defecto del resto. Quizás el más importante es el número de neuronas recurrentes que se indica con el argumento unitsy que en nuestro caso hemos decidido que sea 1024 neuronas.

Con return_sequencese indica que queremos predecir el carácter siguiente a todos los caracteres de entrada, no solo el siguiente al último carácter.

El argumento statefulindica, explicado de manera simple, el uso de las capacidades de memoria de la red entre *batches*. Si este argumento está instanciado a falsese indica que a cada nuevo *batch*se inicializan las *memory cells*comentadas anteriormente, mientras que si está a truese está indicando para cada *batch*se mantendrán las actualizaciones hechas durante la ejecución del *bach*anterior.

El último argumento que usamos es recurrent_kernel, donde indicamos cómo se deben inicializar los pesos de las matrices internas de la red. En este caso usamos la distribución uniforme glorot_uniform, habitual en estos casos.

Finalmente la última capa es de tipo Dense, ya explicada previamente en este libro. Aquí es importante el argumento units que nos dice cuantas neuronas tendrá la capa y que nos marcará la dimensión de la salida. En nuestro caso será igual al tamaño de nuestro vocabulario (vocab_size).

Como siempre, es interesante usar el método summary()para visualizar la estructura del modelo:

Podemos comprobar que la capa LSTM consta de muchos parámetros (más de 5 millones) como era de esperar. Intentemos analizar un poco más esta red neuronal. Para cada carácter de entrada (transformado a su equivalente numérico), el modelo busca su vector de *Embedding* correspondiente y luego ejecuta la capa LSTM con este vector *Embedding*como entrada. A la salida de la LSTM aplica la capa Densepara decidir cual es el siguiente carácter.

Inspeccionemos las dimensiones de los tensores para poder comprender más a fondo el modelo. Fijemonos en el primer *batch*del conjunto de datos de entrenamiento y observemos su forma:

Vemos que en esta red la secuencia de entrada son *batch*de 100 caracteres, pero el modelo una vez entrenado puede ser ejecutado con cualquier tamaño de cadena de entrada. Este es un detalle al que luego volveremos.

Como salida el modelo nos devuelve un tensor con una dimension adicional con la verosimilitud para cada carácter del vocabulario:

Sin entrar en detalle, pedimos al lector que se fije en que la capa densa de esta red neuronal no tiene una función de activación *softmax*como la capa densa que se presentó en el capítulo 2. De aquí que retorne el vector con un indicador de "evidencia" para cada carácter.

El siguiente paso es elegir uno de los caracteres. Sin entrar en detalle, no se eligirá el carácter más "probable" (mediante argmax)como se hizo en el capítulo 2 puesto que el modelo pueda entrar en un bucle. Lo que se hará es obtener una muestra de la distribución de salida. Pruébelo para el primer ejemplo en el *batch*:

Con tf.random.categoricalse obtiene una muestra de una distribución categórica y con squeezese elimina la dimensiones del tensor de tamaño 1. De esta manera en cada instante de tiempo se obtiene una predicción del índice del siguiente carácter.

Entrenamiento del modelo RNN

En este punto, el problema puede tratarse como un problema de clasificación estándar para el que debemos definir la función de *Loss*y el optimizador.

Para la función de *Loss*usaremos la función estándar tf.keras.losses.sparse_categorical_crossentropydado que estamos considerando datos categóricos. Dado que el retorno hemos visto que se trata de unos valores de verisimilitud (no de probabilidades como si hubiéramos ya aplicado *softmax*) se instanciará el argumentofrom_logitsa True.

En cuanto al optimizador usaremos tf.keras.optimizers.Adamcon los argumentos por defecto del optimizador Adam.

Con esta función de loss definida y usando el optimizador Adam con sus argumentos por defecto, ya podemos llamar al método compile () de la siguiente manera:

En este ejemplo aprovecharemos para usar los *Checkpoints***[23]**, una técnica de tolerancia de fallos para procesos cuyo tiempo de ejecución es muy largo. La idea es guardar una instantánea del estado del sistema periódicamente para recuperar desde ese punto la ejecución en caso de fallo del sistema. En nuestro caso, cuando entrenamos modelos *Deep Learning*, el *Checkpoint* lo forman básicamente los pesos del modelo. Estos *Checkpoint* se pueden usar también para hacer predicciones tal cual como haremos en este ejemplo.

La librería de Keras proporciona *Checkpoints a través de* la API *Callbacks.* Concretamente usaremos tf.keras.callbacks.ModelCheckpoint[24]para especificar cómo salvar los *Checkpoints*a cada *epoch*durante el entrenamiento, a través de un argumento en el método fit()del modelo.

En el código debemos especificar el directorio en el que se guardarán los *Checkpoints* que salvaremos y el nombre del fichero (que le añadiremos el número de *epoch*para nuestra comodidad):

Ahora ya está todo preparado para empezar a entrenar la red con el método fit():

Generación de texto usando el modelo RNN

Ahora que tenemos ya entrenado el modelo pasemos a usarlo para generar texto. Para mantener este paso de predicción simple, vamos a usar un tamaño de *batch*de 1. Debido a la forma en que se pasa el estado de la RNN de un instante de tiempo al siguiente, el modelo solo acepta un tamaño de *batch*fijo una vez construido. Por ello, para poder ejecutar el modelo con un tamaño de *batch*diferente, necesitamos reconstruir manualmente el modelo con el método build( )del modelo y restaurar sus pesos desde el *Checkpoints*(cogemos el ultimo con tf.train.latest_checkpoint ()):

Ahora que tenemos el modelo entrenado y preparado para usar, generaremos texto a partir de una palabra de partida con el siguiente código:

El código empieza con inicializaciones como: definir el número de caracteres a predecir con la variable num_generate,convertir la palabra inicial (start_string) a su correspondiente representación numérica y preparan lo tensores necesarios:

Usando la misma idea del código original char-rnn[25]de Andrey Karpathy, se usa una variable temperaturepara decidir cómo de conservador en sus predicciones queremos que se comporte nuestro modelo. En nuestro ejemplo la hemos inicializado a 0.5:

Con “temperaturas altas” (hasta 1) se permitirá más creatividad al modelo para generar texto pero a costa de más errores (por ejemplo, errores ortográficos, etc.). Mientras que con “temperaturas bajas” habrá menos errores pero el modelo mostrará poca creatividad. Propongo que el lector pruebe con diferentes valores y vea su efecto. Incluso le propongo que use el *dataset* Shakespeare.txt [26] (de tamaño mucho mayor que el presentado en este libro) que Andrey Karpathy usa en su ejemplo original.

A partir de este momento empieza el bucle para generar los caracteres que le hemos indicado (que usa el carácter de entrada la primera vez) y luego sus própias predicciones como entrada a cada iteración al modelo RNN:

Recordemos que estamos en un *batch* de 1 pero el modelo retorna el tensor del *batch* con las dimensiones que lo habíamos entrenado y por tanto debemos reducir la dimensión *batch*:

Este carácter acabado de predecir se usa como nuestra próxima entrada al modelo, retroalimentando el modelo para que ahora tenga más contexto (en lugar de una sola letra). Después de predecir la siguiente letra, se retroalimenta nuevamente, y así sucesivamente de manera que es cómo aprende a medida que se obtiene más contexto de los carácteres predichos previamente:

Empecemos con una palabra que no conoce el corpus, por ejemplo “Alcohol”, que nada tiene que ver con *Deep Learning*:

Como vemos el modelo no es capaz de generar ningún texto que tenga ningún parecido a un posible texto relacionado con el tema.

Probemos ahora con una palabra como «modelo» o “activación” a ver que pasa:

conjuntos de datos de tipo texto. Por ejemplo en el artículo "*The Unreasonable Effectiveness of Recurrent Neural Network*" del blog de Andrey Karpathy[27]el lector puede encontrar varios ejemplos de datos de tipo texto que el lector puede usar directamente simplemente cambiando la URL del fichero de texto de entrada al código propuesto en este capítulo.

Hasta aquí un ejemplo muy simple pero que espero que haya sido útil al lector para comprender la idea que hay detrás de las redes neuronales recurrentes. Un tipo de arquitectura que solo hace pocos años que han iniciado su andadura con resultados impresionante en un amplio abanico de tareas como *machine translation*[28](2015), *language modeling*[29](2015) o *speech recognition*[30](2013) por poner algunos ejemplos.

Es sin duda una de las áreas de investigación más activas en *Deep Learning*en estos momentos, en la que incluso nuestro grupo de investigación está realizando aportaciones[31]. Pero también es un área que genera mucho debate, al poderse crear sistemas que escriben prosa de manera convincente como el presentado recientemente por OpenAI[32]. Un modelo entrenado con miles de millones de palabras para poder crear artículos "creibles", que muestra cómo estos algoritmos podrían usarse para engañar a las personas a gran escala, automatizando por ejemplo la generación de noticias falsas en redes sociales. Pero para el propósito de este libro creo que hemos llegado al punto adecuado para dejar estas redes y pasar a otras también muy interesantes como son las *Generative Adversarial Networs*que presentamos en el siguiente capítulo.

Referencias del capítulo 7

[1]Véasehttps://es.wikipedia.org/wiki/Procesamiento_de_lenguajes_naturales [Accedido: 18/08/2019]

[2]Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In ICLR, 2015. https://arxiv.org/pdf/1409.0473.pdf [Accedido: 20/09/2019]

[3]Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. Recurrent neural network regularization. In ICLR, 2015

[4]Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In ICASSP, 2013. https://arxiv.org/pdf/1303.5778.pdf [Accedido: 20/09/2019]

[5]Véasehttps://arxiv.org/pdf/1412.3555v1.pdf [Accedido: 18/08/2019]

[6]Véasehttps://es.wikipedia.org/wiki/One-hot [Accedido: 18/08/2019]

[7]Véasehttps://es.wikipedia.org/wiki/Word_embedding [Accedido: 18/08/2019]

[8]Véase https://developers.google.com/machine-learning/crash-course/embeddings/translating-to-a-lower-dimensional-space [Accedido: 18/08/2019]

[9]Véase https://en.wikipedia.org/wiki/Word2vec [Accedido: 18/08/2019]

[10]Véase https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/keras/layers/Embedding [Accedido: 08/09/2019]

[11]Véase Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics (NAACL). 4171–4186. https://arxiv.org/pdf/1810.04805.pdf [Accedido: 08/09/2019]

[12]Véase Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners https://d4mucfpksywv.cloudfront.net/better-language-models/language-models.pdf[Accedido: 08/09/2019]

[13]Véase Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. 2018. Deep contextualized word representations. In Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics (NAACL). 2227–2237 https://arxiv.org/pdf/1802.05365.pdf[Accedido: 08/09/2019]

[14]Véase Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Ruslan Salakhutdinov, and Quoc V. Le. 2019. XLNet: Generalized Autoregressive Pretraining for Language Understanding. https://arxiv.org/pdf/1906.08237.pdf [Accedido: 08/09/2019]

[15]Véasehttps://en.wikipedia.org/wiki/Andrej_Karpathy [Accedido: 18/08/2019]

[16]Véasehttp://karpathy.github.io/2015/05/21/rnn-effectiveness/ [Accedido: 18/08/2019]

[17]Véasehttps://www.tensorflow.org/beta/tutorials/text/text_generation [Accedido: 18/08/2019]

[18]Véasehttps://github.com/karpathy/char-rnn [Accedido: 18/08/2019]

[19]Véase https://github.com/jorditorresBCN/Deep-Learning-Introduccion-practica-con-Keras/blob/master/DeepLearning-Introduccion-practica-con-Keras-PRIMERA-PARTE.txt

[20]El argumento del método shuffle indica el tamaño del búfer para barajar. Recordemos que tf.data puede generar secuencias infinitas, por lo que no intenta barajar toda la secuencia de datos de entrada en la memoria, solo el número de datos que se le indica en el argumento.

[21]Véase https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/keras/layers/Embedding [Accedido: 18/08/2019]

[22]Véase https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/keras/layers/LSTM [Accedido: 18/08/2019]

[23]Véase https://en.wikipedia.org/wiki/Application_checkpointing [Accedido: 18/08/2019]

[24]Véase https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/keras/callbacks

[25]Véase https://github.com/karpathy/char-rnn [Accedido: 18/08/2019]

[26]El lector solo debe cambiar la dirección la URL del fichero de donde se cargan los datos a https://cs.stanford.edu/people/karpathy/char-rnn/shakespear.txt [Accedido: 18/08/2019]

[27]Véase http://karpathy.github.io/2015/05/21/rnn-effectiveness/ [Accedido: 18/08/2019]

[28]Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In ICLR , 2015.[Accedido: 18/08/2019]

[29]Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. Recurrent neural network regularization. In ICLR , 2015. [Accedido: 18/08/2019]

[30]Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In ICASSP , 2013. [Accedido: 18/08/2019]

[31]SKIP RNN: Learning to skip state updates in Recurrent Neural Networks Víctor Campos, Brendan Jouz , Xavier Giró-i-Nieto , Jordi Torres , Shih-Fu Chang. in ICLR 2018. [Accedido: 18/08/2019]

[32]Véase https://arxiv.org/pdf/1908.09203.pdf [Accedido: 18/08/2019]

← Entrada anterior

Entrada siguiente →

JORDI TORRES
UPC Campus Nord , C6 217
C/ Jordi Girona 1-3,
08034 Barcelona

Contact: jordi @ torres.ai

Descargo de responsabilidad: Las opiniones e informaciones expresadas en este sitio web son exclusivamente mías y no reflejan la política o posición oficial de ninguna de las instituciones con las que estoy afiliado. Este sitio web también está dedicado a apoyar mis actividades docentes en la UPC, compartiendo información y artículos sobre los avances en las áreas de Supercomputación e Inteligencia Artificial. No asumo ninguna responsabilidad por errores involuntarios u omisiones en el contenido proporcionado.

Disclaimer: The opinions and information expressed on this website are solely my own and do not reflect the official policy or position of any institutions with which I am affiliated. This website also supports my UPC teaching activities by sharing information and articles on advances in Supercomputing and Artificial Intelligence. I assume no responsibility for inadvertent errors or omissions in the content provided.