

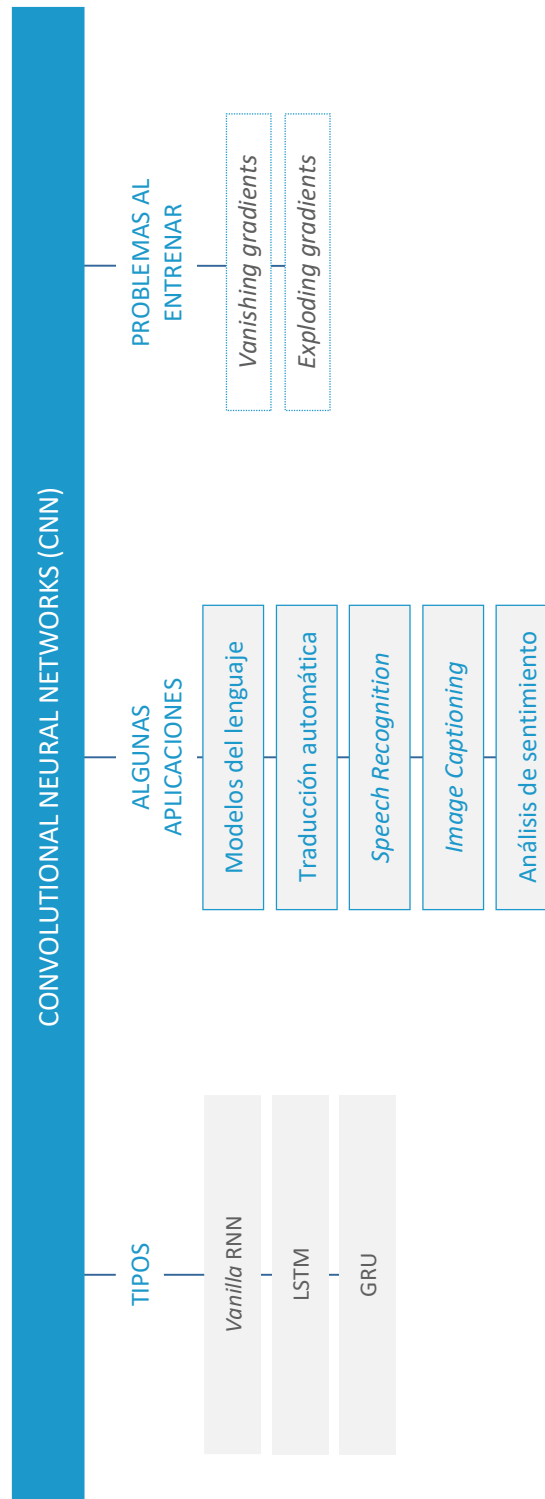
Sistemas Cognitivos Artificiales

Recurrent Neural Networks (RNN)

Índice

Esquema	3
Ideas clave	4
7.1. ¿Cómo estudiar este tema?	4
7.2. <i>Recurrent Neural Networks</i>	4
7.3. Modelos del lenguaje con RNN	15
7.4. Arquitecturas LSTM y GRU	22
Lo + recomendado	30
+ Información	34
Test	36

Esquema



7.1. ¿Cómo estudiar este tema?

Para estudiar este tema deberás leer las **Ideas clave** que se desarrollan a continuación.

En este tema veremos otro de los tipos de redes neuronales que, junto a las CNN, ha revolucionado las aplicaciones de *machine learning* a varios problemas reales. Las *recurrent neural networks* o RNN tratan problemas sobre **secuencias** y están siendo fundamentales en áreas como el procesamiento del lenguaje natural.

Es importante comprender el funcionamiento de las *recurrent neural networks*, cómo se retroalimentan a lo largo del tiempo, cómo se utilizan durante una secuencia y cómo pueden producir una secuencia de salida. Al acabar el tema, hemos de ser capaces de relacionar ciertas aplicaciones con el uso de modelos recurrentes. También es fundamental comprender el caso particular de los modelos del lenguaje y de qué forma una RNN puede tratar el problema de manera natural, además de poder convertirse en un modelo capaz de generar lenguaje. Finalmente, es necesario conocer cuáles son los problemas que tienen las RNN sencillas y las arquitecturas más avanzadas que los solucionan.

7.2. Recurrent Neural Networks

Hasta ahora hemos visto dos tipos de redes neuronales: *feed-forward neural networks* y *convolutional neural networks*. En ambas, tenemos un *input* de tamaño fijo (por ejemplo, una imagen) que produce una única salida (por ejemplo, una serie de probabilidades que permite determinar a qué clase

corresponde la imagen). Sin embargo, en muchos problemas de *machine learning* nos gustaría tener más flexibilidad, con arquitecturas con una longitud variable de *inputs* o de *outputs*.

Como vimos en el tema anterior, un texto puede ser representado por un conjunto de palabras sin orden en lo que se conoce como *bag-of-words*. En este tipo de representación, todas las palabras del texto se juntan en un solo *input* discreto en forma de vector cuya dimensión es del tamaño del vocabulario; donde por cada palabra en el vocabulario se guarda el número de veces que esta palabra aparece en el texto. No obstante, algo que parece que tendría más sentido desde un punto de vista lingüístico sería representar ese texto como una secuencia ordenada de palabras, donde una red neuronal tomaría como *input* palabra a palabra hasta ver todo el texto.

Las *recurrent neural networks*, redes neuronales recurrentes en castellano o simplemente RNN, son un tipo de red neuronal que es capaz de trabajar con información secuencial, manteniendo un estado interno o ***hidden state*** (que podemos considerar como una memoria) para procesar secuencias de entrada. Este tipo de arquitectura aplica una fórmula recurrente sobre una secuencia de entrada de manera que, en cada paso dado en la secuencia, se depende del nuevo valor de *input* x y del estado interno anterior h . Por cada avance de la red en la secuencia, se suele decir que hemos avanzado un ***time step***. Esto no quiere decir que todos los problemas que las RNN tratan sean temporales, si bien cualquier problema que puede desarrollarse en una serie temporal (por ejemplo, un vídeo, que es una secuencia de imágenes) es un gran candidato para ser tratado con una red recurrente.

La potencia de este tipo de redes radica en su capacidad de **modelar relaciones temporales entre elementos de la secuencia** a través del estado interno de la red, que actúa como una suerte de memoria sobre lo que la red ha visto hasta ahora. No es, por tanto, sorprendente que este tipo de arquitecturas haya sido de gran

importancia en problemas de procesamiento del lenguaje natural, donde el contexto y el orden son determinantes para la comprensión y tratamiento del texto.

Ejemplos de problemas secuenciales

Antes de ver más formalmente el funcionamiento de las RNN, veamos una serie de problemas con secuencias que pueden ser naturalmente tratados con este tipo de redes neuronales.

El caso más sencillo podría ser el de una secuencia de entrada y una única salida. En este tipo de problema, tenemos una secuencia de entrada sobre la que aplicamos una red neuronal recurrente de la que tomamos una única salida al final de la secuencia.

Ejemplo 1. Podría ser el **análisis de sentimiento** en una oración o texto. El texto sería una secuencia de palabras sobre las que aplicaríamos la RNN y la salida sería la probabilidad de que el texto tenga un sentimiento asociado positivo o negativo.

En la imagen inferior podemos ver cómo se desarrollaría esto: cada rectángulo rojo es el *input* por cada palabra (por ejemplo, el *word vector* asociado a esta), los rectángulos verdes representan el *hidden state* de la red una vez que se aplica cada entrada y, finalmente, el rectángulo azul es la salida de la red una vez se ha desarrollado toda la secuencia.

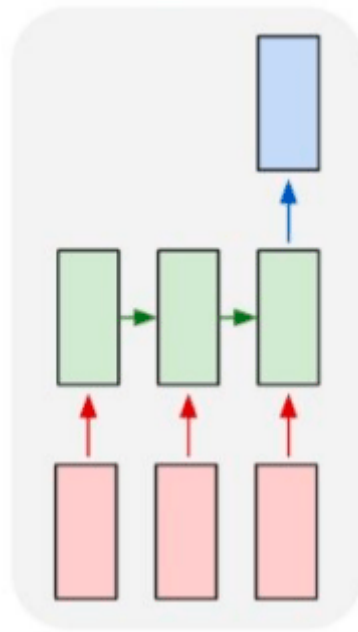


Figura 1. Secuencia de entrada de única salida.

Fuente: http://cs231n.stanford.edu/slides/2018/cs231n_2018_lecture10.pdf

Ejemplo 2. Otro caso que puede tratarse con las RNN son los problemas de secuencia a secuencia. Por ejemplo, la **traducción automática** o *machine translation*, donde tenemos una secuencia de entrada (una frase) y la salida es otra frase en el idioma objetivo.

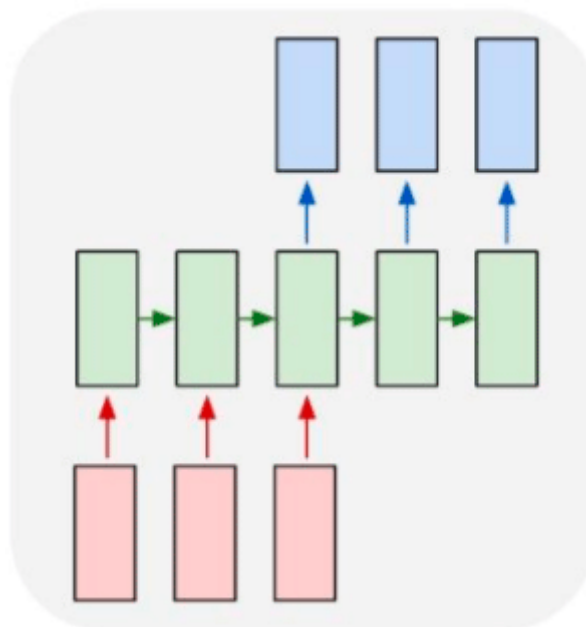


Figura 2. Secuencia de entrada con secuencia de salida.

Fuente: http://cs231n.stanford.edu/slides/2018/cs231n_2018_lecture10.pdf

En este contexto solo empezamos a generar la secuencia de salida una vez que la secuencia de entrada se ha leído completamente, como sucede en problemas donde necesitamos tener toda la información de la secuencia de entrada antes de proceder con la salida, lo cual tiene sentido para el problema de traducción, ya que hay que «leer» el texto a traducir antes de lanzarse con la traducción. No obstante, esto no tiene por qué ser el caso en general, podemos igualmente generar una secuencia de salida a la vez que leemos la de entrada. Por ejemplo, en un vídeo podemos clasificar qué tipo de imagen hay en cada *frame*, a la vez que vamos leyendo las imágenes de entrada, se van clasificando las imágenes.

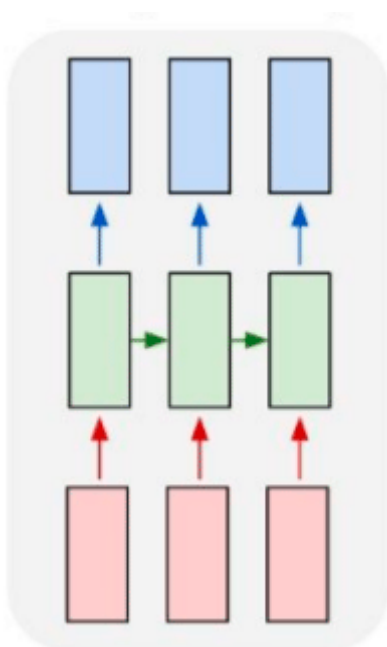


Figura 3. Otro tipo de secuencia de entrada con secuencia de salida.

Fuente: http://cs231n.stanford.edu/slides/2018/cs231n_2018_lecture10.pdf

Ejemplo 3. Incluso es posible generar una secuencia a partir de una sola entrada. Sería el caso del problema de *image captioning* que trata de extraer una pequeña frase que describa una imagen. Por ejemplo, «un niño jugando con juguete» a partir de la imagen del niño. Aquí obtenemos una secuencia de palabras a partir de la imagen de entrada.

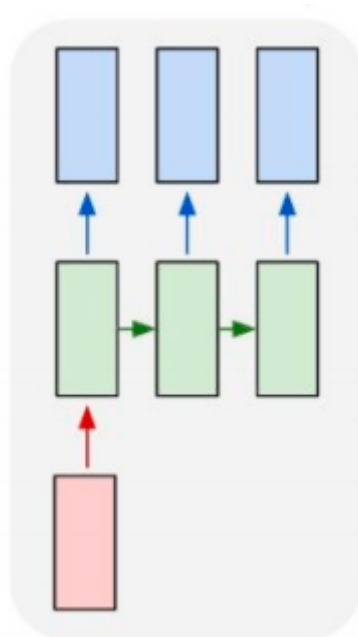


Figura 4. Secuencia a partir de una sola entrada.

Fuente: http://cs231n.stanford.edu/slides/2018/cs231n_2018_lecture10.pdf

Nótese que esto es un esquema muy simplificado y que la arquitectura real de este problema es mucho más compleja e implica también CNN.

Como vemos, las RNN son muy dinámicas y pueden tratar una gran variedad de problemas de manera natural.

Formulación de RNN

Como hemos dicho, una RNN es un tipo de red neuronal que toma una secuencia de valores de entrada y, de manera recurrente, aplica una transformación a partir de cada valor de entrada y del *hidden state* (estado interno) que posee la red en ese momento, obteniendo un nuevo estado interno y, de ser necesario, un nuevo valor de salida.

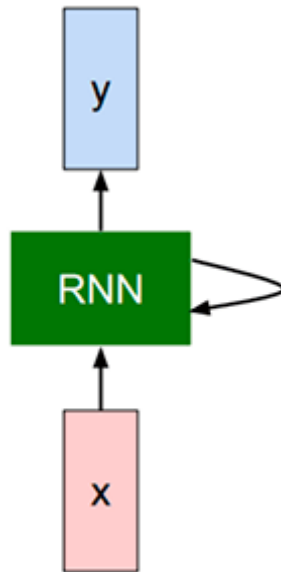


Figura 5. Representación esquemática de una RNN.

Fuente: http://cs231n.stanford.edu/slides/2018/cs231n_2018_lecture10.pdf

De manera formal, la función de recurrencia de una RNN viene dada por:

$$h_t = f_w(h_{t-1}, x_t)$$

Donde:

- ▶ h_t es el valor del *hidden state* (un vector) en el instante de tiempo t .
- ▶ f_w es una transformación no lineal del estilo de las que hemos visto en otras redes neuronales, con w como una matriz de parámetros o *weights* de la red neuronal.
- ▶ h_{t-1} es el valor del *hidden state* en el instante $t - 1$, esto es, el *hidden state* anterior de la RNN.
- ▶ x_t es el valor de *input* para el tiempo t (de nuevo, un vector).

Como vemos, tanto x como h son vectores. La dimensión del vector interno h_t es un hiperparámetro de la red, mientras que la de x depende del problema que estamos tratando.

Esta fórmula es una fórmula general que viene a decir que el nuevo estado interno de la RNN depende del estado anterior y del *input* actual.

El tipo de RNN más estándar, a veces conocido como *vanilla* RNN, viene dado por la siguiente fórmula:

$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$

Donde:

- ▶ W_{hh} y W_{xh} son las matrices de parámetros aplicadas a los vectores h_{t-1} y x_t respectivamente.

La fórmula es un producto matricial de una matriz por vector, de modo que el resultado final de hacer los productos y sumarlos es otro vector. Sobre este último se aplica la *nonlinearity* \tanh elemento a elemento, obteniendo el nuevo *hidden state*.

Finalmente, a partir del estado interno de la red recurrente podemos obtener la salida de la red y_t en el instante t :

$$y_t = W_{hy}h_t$$

Aquí estamos aplicando una transformación lineal sobre h_t mediante el producto de la matriz de pesos W_{hy} con el vector del *hidden state*. Nótese que durante esta clase no hemos visto la aplicación de transformaciones lineales en forma de productos matriciales. Este paso es equivalente al de aplicar una capa de una red neuronal estándar sin funciones de activación.

Con estas fórmulas tenemos el modo de avanzar una red recurrente sobre los valores de entrada y obtener una salida por cada *time step* t si el problema lo requiere. Es importante recalcar que los valores de los parámetros (en este caso, los valores de las matrices W_{hh} , W_{xh} y W_{hy}) no cambian en cada *time step*, es decir, un solo conjunto de parámetros es utilizado durante toda la secuencia.

«Desenrollando» una RNN

La formulación como recurrencia de las RNN puede resultar un poco liosa. Sin embargo, es relativamente sencillo de visualizar si «desenrollamos» el grafo de computación para una red, tal y como se ve en la siguiente imagen:

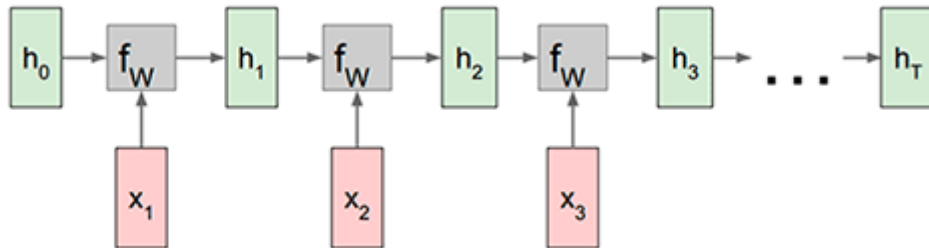


Figura 6. Grafo de computación de una RNN.

Fuente: http://cs231n.stanford.edu/slides/2018/cs231n_2018_lecture10.pdf

Vemos cómo se aplica la función f_W al estado anterior de la red y a las nuevas *inputs* que llegan en cada *time step*, dando lugar a un nuevo estado. Igualmente, es posible obtener una salida por cada estado al que avanzamos, como se ve en la siguiente imagen. En ella también se recalca cómo los *weights* o pesos W tienen el mismo valor en todo el desarrollo de la secuencia.

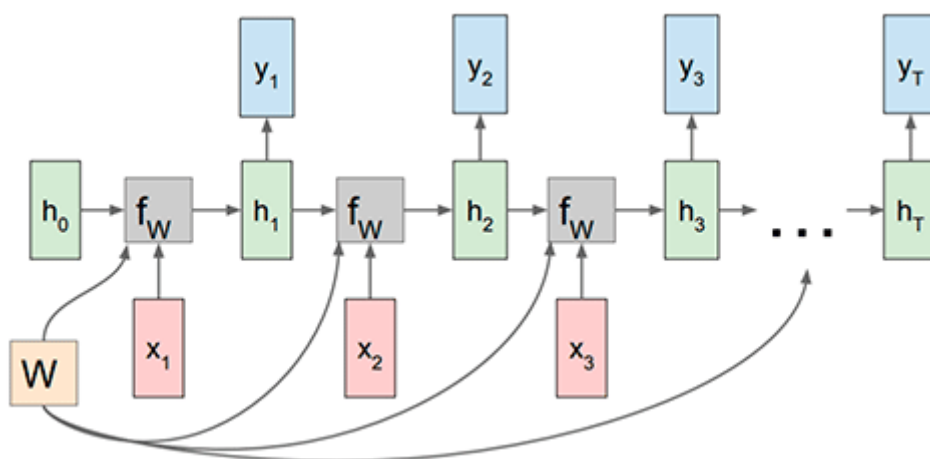


Figura 7. Grafo de computación de una RNN con *outputs*.

Fuente: http://cs231n.stanford.edu/slides/2018/cs231n_2018_lecture10.pdf

Como vemos, es necesario también tener un estado inicial h_0 de la red recurrente. Es muy común empezar con un vector de ceros.

De manera similar, podemos ver el caso de generar una secuencia con un solo *input*. En este caso, los vectores de *input* a partir de x_1 desaparecen (son 0) y simplemente desarrollamos el estado interno de la *hidden state* durante varios *time steps*.

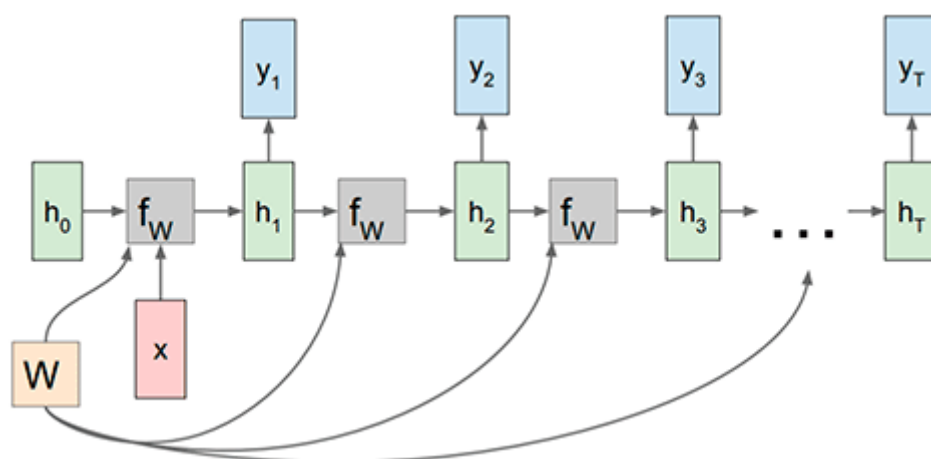


Figura 8. Secuencia a partir de un único *input*.

Fuente: http://cs231n.stanford.edu/slides/2018/cs231n_2018_lecture10.pdf

Backpropagation through time

Las RNN no son distintas al resto de redes neuronales en tanto en cuanto se utiliza el algoritmo de *backpropagation* para entrenarlas. En este caso, es necesario «desenrollar» la red a lo largo del tiempo y aplicar *backpropagation* a partir de todos los *outputs* que hemos obtenido (ver figura 9).

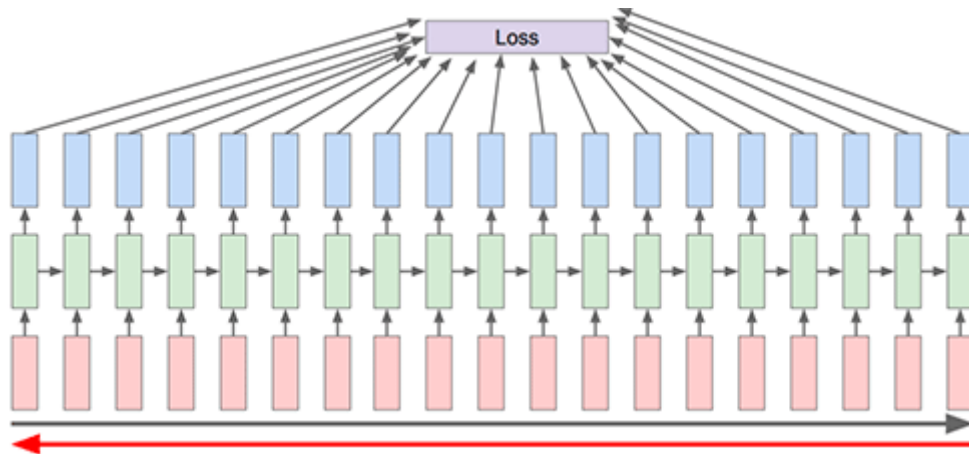


Figura 9. *Backpropagation Through Time* (BPTT).

Fuente: http://cs231n.stanford.edu/slides/2018/cs231n_2018_lecture10.pdf

Esto se conoce como ***backpropagation through time* (BPTT)**. La idea es hacer un *forward pass* a lo largo de toda la secuencia de entrada para calcular nuestra *loss* a partir de todos los *outputs*, y luego hacer el *backward pass* a lo largo de toda la secuencia. Si producimos una salida en cada *time step*, cada una de ellas tendrá cierta implicación en la *loss* calculada.

Este sistema puede hacerse computacionalmente muy costoso si nuestras secuencias de entrada son muy grandes. Por ello, en la práctica es normal utilizar ***truncated backpropagation through time* (TBPTT)**, donde la secuencia de entrada se «parte» en trozos y solo hacemos *backpropagation* en un pequeño número de *steps*. Por cada subsecuencia de entrada, el *hidden state* se mantiene desde la subsecuencia anterior, pero solo se hace *backpropagation* en la subsecuencia que estemos viendo en el momento (ver figura 10).

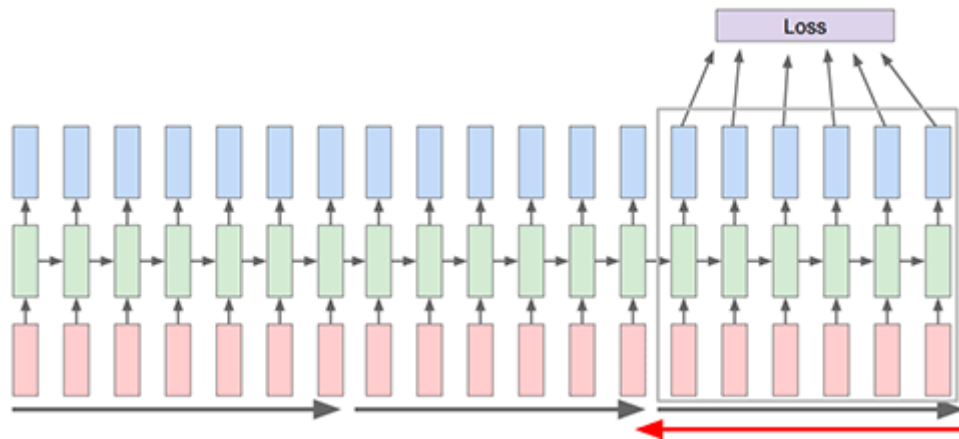


Figura 10. *Truncated Backpropagation Through Time (TBPTT)*.

Fuente: http://cs231n.stanford.edu/slides/2018/cs231n_2018_lecture10.pdf

7.3. Modelos del lenguaje con RNN

Veamos ahora una aplicación de las redes recurrentes donde estas han mejorado el estado de la técnica: los modelos del lenguaje, *language models* en inglés. Básicamente, un modelo del lenguaje es un modelo que asigna una probabilidad a una secuencia de palabras o incluso de caracteres:

$$p(w_1, w_2, \dots, w_T).$$

Modelos del lenguaje tradicionales

Estos modelos son de gran utilidad. Por ejemplo, en traducción automática se suele tener una serie de traducciones candidatas, por lo que podemos valernos de la probabilidad que asigna el modelo a cada una de ellas para elegir la mejor traducción. Así, en castellano deberíamos de tener:

$$p(\text{"la casa es pequeña"}) > p(\text{"pequeña la casa es"})$$

Otro sitio donde podemos ver la aplicación de modelos del lenguaje es en la búsqueda de Google, donde se nos da una serie de opciones probables que la gente busca comúnmente:

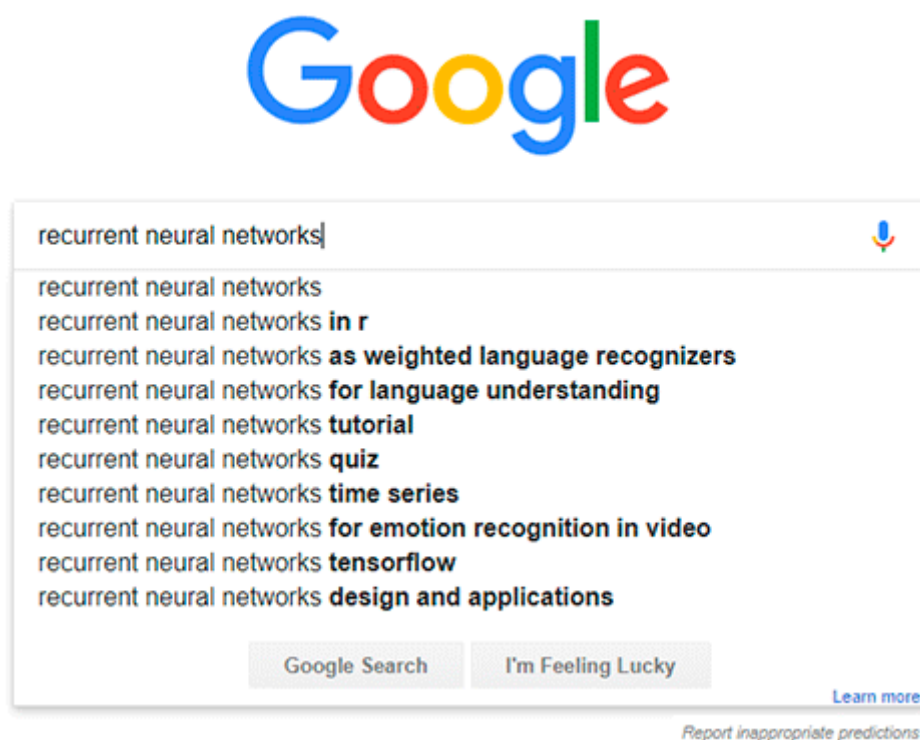


Figura 11. Ejemplo de modelo de lenguaje en el buscador de Google.

Fuente: www.google.com

Tradicionalmente, los modelos del lenguaje se han calculado de manera aproximada a través de probabilidades obtenidas mediante cuentas sencillas de palabras. La probabilidad de una serie de palabras para cada palabra viene dada por la probabilidad de que se dé esa palabra condicionada a las anteriores. Sin embargo, es normal aproximar esto con la suposición de que sería suficiente con conocer unas pocas palabras anteriores.

$$P(w_1, \dots, w_m) = \prod_{i=1}^m P(w_i | w_1, \dots, w_{i-1}) \approx \prod_{i=1}^m P(w_i | w_{i-(n-1)}, \dots, w_{i-1})$$

Estas probabilidades normalmente se estiman contando en los datos cuántas veces aparecen las palabras juntas. Por ejemplo, para calcular $p(w_2|w_1)$, es decir, la

probabilidad de que la palabra w_2 venga después de w_1 , se cuentan todas las veces que las dos palabras aparecen en ese orden y se divide por el número de veces que aparece w_1 en general.

$$p(w_2|w_1) = \frac{\text{count}(w_1, w_2)}{\text{ount}(w_1)}$$
$$p(w_3|w_1, w_2) = \frac{\text{count}(w_1, w_2, w_3)}{\text{ount}(w_1, w_2)}$$

Estos modelos sencillos, si bien funcionan de manera razonable, tienen un problema: si queremos condicionar una palabra a un número elevado de palabras anteriores, necesitaremos almacenar en memoria una cantidad enorme de cuentas con todas las posibles combinaciones de palabras. De este modo, estos modelos se vuelven computacionalmente intratables a mayor calidad de modelo deseada.

Modelos del lenguaje con RNN

Como podemos ya imaginar, este problema puede tratarse de manera natural con *recurrent neural networks*. Es posible suministrar la secuencia de palabras a nuestra red neuronal paso a paso, intentando hacer que la red sea capaz de predecir la palabra siguiente. De este modo, el estado interno de la RNN codifica en cierta medida el texto visto hasta ahora, siendo por tanto capaz de condicionar la probabilidad de la siguiente palabra a lo visto anteriormente.

El *input* por cada *time step* puede ser un *word vector* preentrenado con las técnicas vistas en el capítulo anterior, o simplemente una representación *one-hot*, un vector con todo 0 menos un 1 en la posición de la palabra en el vocabulario. Como salida de la red en cada *time step*, tenemos una capa softmax que nos da la probabilidad de todas las palabras en el vocabulario a partir de la palabra vista en ese *time step* y del *hidden state* de la red, que sintetiza el resto de palabras vistas anteriormente. De este modo, un modelo del lenguaje con RNN puede utilizarse como un **modelo generativo** capaz de generar textos a partir de un punto de inicio. Podemos pensar de nuevo en

el Autosuggest de Google, que intenta predecir nuestra búsqueda mediante la generación de una serie de candidatos probables dentro del conjunto de búsquedas en el portal.

Veamos un ejemplo de un modelo del lenguaje a nivel de caracteres o letras. Curiosamente, no es necesario contar con modelos basados en palabras para ser capaces de modelar el lenguaje; los modelos basados en caracteres han demostrado una gran versatilidad y un funcionamiento muy adecuado al ser entrenados con RNN.

Para nuestro ejemplo, basado en el contenido que se puede ver en los apuntes del curso de CS231N (ver «Lo más recomendado»), tenemos un vocabulario de cuatro caracteres:

[h, e, l, o]

Por cada carácter, tendremos una representación *one-hot* en la entrada:

- ▶ «h» sería [1, 0, 0, 0].
- ▶ «l» sería [0, 0, 1, 0].

En la siguiente imagen podemos ver cómo sería el entrenamiento mediante la secuencia de entrada «hello»:

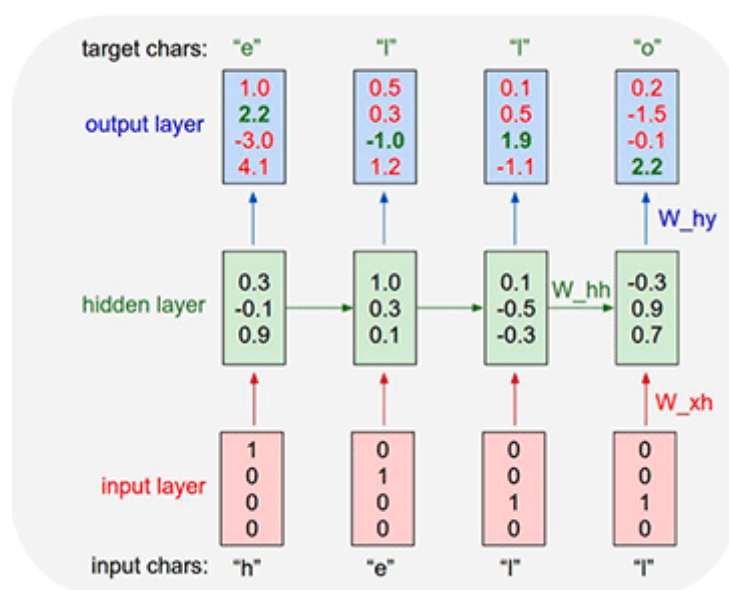


Figura 12. Entrenamiento de un modelo del lenguaje.

Fuente: http://cs231n.stanford.edu/slides/2018/cs231n_2018_lecture10.pdf

En la imagen podemos ver cómo vamos introduciendo carácter a carácter, siendo la salida a predecir el siguiente carácter en la secuencia. En la *output layer* podemos ver los *scores* que la red neuronal predice por cada carácter (nótese que estos valores no son probabilidades, ya que no se ha pasado la función softmax). También se puede apreciar cómo el modelo se ha equivocado, por ejemplo, en el primer carácter predicho, ya que ha dado mayor puntuación a la «o» en vez de a la «e». Como sabemos, durante el entrenamiento el modelo se corregirá asignando una *loss* a ese fallo.

Podemos ver también dónde operan las distintas matrices de pesos W_{hh} , W_{xh} y W_{hy} y cómo el *hidden state* se va actualizando con la nueva información que recibe.

Veamos ahora cómo podemos **utilizar nuestra RNN para generar texto**. Supongamos que la red ya ha sido entrenada y tenemos nuestras matrices W de pesos fijadas. Podemos hacer inferencia con la RNN para generar texto. Empezamos alimentando a la red con la letra «h»; al aplicar softmax a la última capa, la red nos da una serie de

probabilidades para cada letra del vocabulario que pueden venir después de la «h». Utilizando estas probabilidades, hacemos un muestreo y obtenemos la siguiente letra a partir de la distribución de probabilidad que nos da la red. En el ejemplo, aunque la letra más probable es la «o» con un 0.84 de probabilidad, al hacer el muestreo aleatorio hemos obtenido una «e». Esta letra obtenida se utiliza a su vez como entrada, con lo que obtendremos una nueva letra de salida, y así sucesivamente.

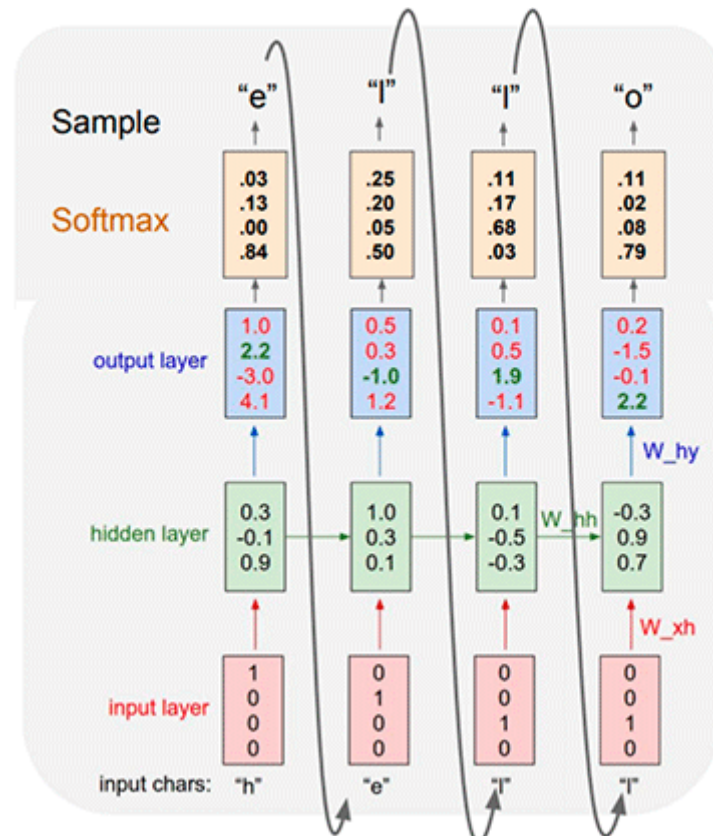


Figura 13. Uso de un modelo del lenguaje para generar texto.
Fuente: http://cs231n.stanford.edu/slides/2018/cs231n_2018_lecture10.pdf

Podríamos preguntarnos por qué aplicamos un muestreo sobre la distribución de probabilidad en vez de coger directamente la letra más probable. Si bien esto puede hacerse para obtener la combinación más probable, es normal utilizar estos modelos para generar distintas opciones y explorar el espacio de posibilidades, como se ve en el ejemplo del Autosuggest de Google.

Una demostración de lo que las redes recurrentes pueden ser capaces de hacer, veamos el resultado de entrenar un modelo del lenguaje basado en caracteres en las obras completas de Shakespeare:

PANDARUS:
Alas, I think he shall be come approached and the day
When little strain would be attain'd into being never fed,
And who is but a chain and subjects of his death,
I should not sleep.

Second Senator:
They are away this miseries, produced upon my soul,
Breaking and strongly should be buried, when I perish
The earth and thoughts of many states.

DUKE VINCENTIO:
Well, your wit is in the care of side and that.

Second Lord:
They would be ruled after this chamber, and
my fair nudes begun out of the fact, to be conveyed,
Whose noble souls I'll have the heart of the wars.

Clown:
Come, sir, I will make did behold your worship.

VIOLA:
I'll drink it.

VIOLA:
Why, Salisbury must find his flesh and thought
That which I am not apt, not a man and in fire,
To show the reining of the raven and the wars
To grace my hand reproach within, and not a fair are hand,
That Caesar and my goodly father's world;
When I was heaven of presence and our fleets,
We spare with hours, but cut thy council I am great,
Murdered and by thy master's ready there
My power to give thee but so much as hell:
Some service in the noble bondman here,
Would show him to her wine.

KING LEAR:
O, if you were a feeble sight, the courtesy of your law,
Your sight and several breath, will wear the gods
With his heads, and my hands are wonder'd at the deeds,
So drop upon your lordship's head, and your opinion
Shall be against your honour.

Figura 14. Ejemplo de modelo del lenguaje entrenado en Shakespeare.

Fuente: Adaptado de http://cs231n.stanford.edu/slides/2018/cs231n_2018_lecture10.pdf

O, como otro ejemplo, un modelo del lenguaje entrenado en el código fuente en C del kernel de Linux:

```
static void do_command(struct seq_file *m, void *v)
{
    int column = 32 << (cmd[2] & 0x80);
    if (state)
        cmd = (int)(int_state ^ (in_8(&ch->ch_flags) & Cmd) ? 2 : 1);
    else
        seq = 1;
    for (i = 0; i < 16; i++) {
        if (k & (1 << 1))
            pipe = (in_use & UMTHREAD_UNCCA) +
                ((count & 0x00000000ffffffff) & 0x0000000f) << 8;
        if (count == 0)
            sub(pid, ppc_md.kexec_handle, 0x20000000);
        pipe_set_bytes(i, 0);
    }
    /* Free our user pages pointer to place camera if all dash */
    subsystem_info = &of_changes[PAGE_SIZE];
    rek_controls(offset, idx, &soffset);
    /* Now we want to deliberately put it to device */
    control_check_polarity(&context, val, 0);
    for (i = 0; i < COUNTER; i++)
        seq_puts(s, "policy ");
}
```

Figura 15. Modelo del lenguaje entrenado en el kernel de Linux.

Fuente: http://cs231n.stanford.edu/slides/2018/cs231n_2018_lecture10.pdf

Como vemos, estos modelos tienen una gran capacidad de generar contenido con cierto sentido.

7.4. Arquitecturas LSTM y GRU

Hasta ahora hemos visto redes recurrentes en su variante más sencilla, también conocida como *vanilla* RNN. Este tipo de RNN tiene una serie de problemas que ha llevado a la creación de nuevas arquitecturas recurrentes más efectivas y fáciles de entrenar.

Vanishing gradients y exploding gradients

Dos problemas muy típicos en el entrenamiento de redes neuronales recurrentes son los *exploding gradients* y los *vanishing gradients* (en castellano, algo así como «gradientes que explotan» y «gradientes que se desvanecen»). Estos problemas vienen del hecho de que tenemos que hacer *backpropagation* a lo largo de una secuencia larga de elementos.

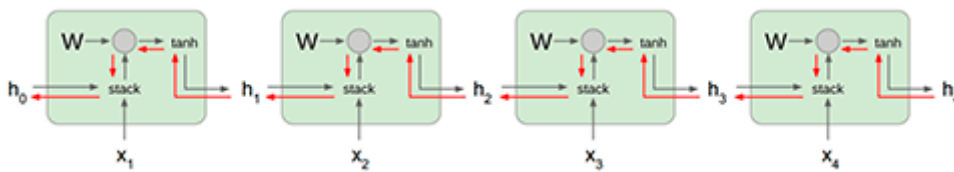


Figura 16. Grafo computacional de una secuencia con RNN.

Fuente: http://cs231n.stanford.edu/slides/2018/cs231n_2018_lecture10.pdf

No entraremos en la formalización matemática de estos problemas, pero podemos intuirlo de manera sencilla. Imaginemos el gradiente que fluye hacia atrás hasta h_0 . Este gradiente implica los mismos factores de W de manera continua, paso por paso (recordemos que los valores de W son los mismos a lo largo del tiempo). Pensemos en qué ocurre cuando multiplicamos un número por sí mismo en muchas ocasiones:

- ▶ Si este número es mayor que 1, obtenemos valores cada vez más grandes.
- ▶ Si este número está entre 0 y 1, vamos obteniendo valores más pequeños y cada vez más cercanos a 0.

Esto es lo que puede llegar a ocurrir en nuestras *vanilla* RNN. Según sean las propiedades de las matrices W y al hacer *backpropagation* en el tiempo, podemos encontrarnos con que:

- ▶ El gradiente pierda su valor de aproximación local y acabe divergiendo a un valor enorme (***exploding gradient***).
- ▶ O que se haga cada vez más pequeño hasta prácticamente desaparecer (***vanishing gradient***).

Como podemos imaginar, ambas situaciones afectan negativamente al entrenamiento de la red.

El problema de los *exploding gradients* puede solucionarse mediante un «truco» sencillo, conocido como **gradient clipping**. La idea es medir la norma vectorial del gradiente en cada *time step* y, si esta supera cierto valor, dividir cada número del gradiente por una constante. Sin embargo, esto no es una solución óptima y, además, no ayuda en el problema de los *vanishing gradients*, por lo que se hacen necesarias mejores arquitecturas recurrentes.

Arquitectura LSTM

La arquitectura LSTM (*Long Short-Term Memory*) data de 1997, por lo que podemos decir que sus creadores estaban muy adelantados a su tiempo, ya que la explosión en el uso de RNN no llegaría hasta más de quince años después. La fórmula de la arquitectura LSTM es un poco compleja y puede verse aquí comparada con una *vanilla* RNN:

Vanilla RNN	LSTM
$h_t = \tanh \left(W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix} \right)$	$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$ $c_t = f \odot c_{t-1} + i \odot g$ $h_t = o \odot \tanh(c_t)$

Figura 17. Comparación entre la arquitectura LSTM y una *vanilla* RNN.

Fuente: http://cs231n.stanford.edu/slides/2018/cs231n_2018_lecture10.pdf

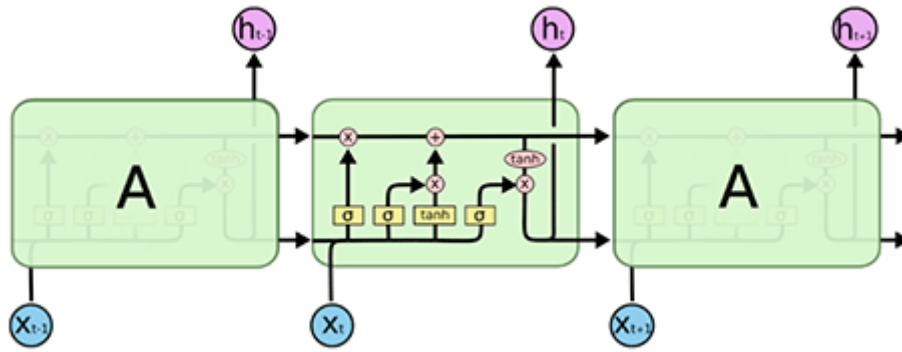


Figura 18. Arquitectura LSTM.

Fuente: <http://colah.github.io/posts/2015-08-Understanding-LSTMs/>

Las LSTM surgieron como una arquitectura encaminada a **solucionar los problemas de «memoria» de las *vanilla* RNN**. En la práctica, estas últimas presentan problemas para aprender relaciones con elementos de *time step* lejanos (es decir, que no están cerca del *time step* actual) debido a factores como los vistos en la sección anterior. Esto hace que gran parte del potencial teórico de las RNN se pierda. Por ejemplo, en el lenguaje es importante mantener información a lo largo de periodos relativamente largos, como podría ser mantener la información sobre el sujeto en la oración «Yo fui a casa después de ir al cine» cuando estamos analizando al final de la misma: «¿quién fue al cine?».

Las LSTM están diseñadas explícitamente para intentar solucionar este problema. Para ello, mantienen un estado interno *cell state* (c_t) además del tradicional *hidden state* (h_t), el cual representa una especie de «autopista de información» a lo largo del tiempo, como vemos en la siguiente imagen.

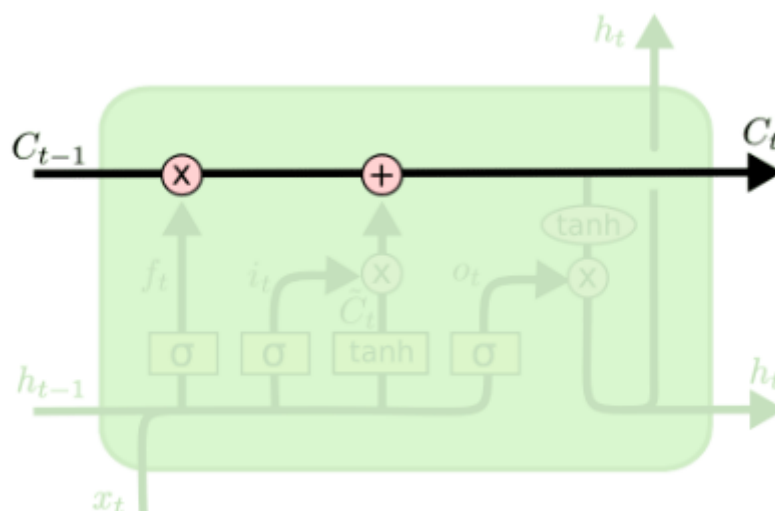


Figura 19. *Cell state* en la arquitectura LSTM.

Fuente: <http://colah.github.io/posts/2015-08-Understanding-LSTMs/>

Observando la fórmula de las LSTM, vemos que tenemos muchos elementos nuevos. En vez de calcular directamente «h», ahora obtenemos cuatro vectores distintos conocidos como **gates**: i , f , o y g , que después se combinan para obtener el *cell state* c y el *hidden state* h . Los productos que se ven entre vectores en la fórmula son productos elemento a elemento.

Como vemos, i , f y o resultan tras aplicar una función *sigmoid*, por lo que tienen valores entre 0 y 1. De este modo, actúan como *gates* que conservan (valores cercanos a 1) o eliminan (valores cercanos a 0) información. Más en particular:

- ▶ f se conoce como **forget gate** y, al ser multiplicada por el *cell state* anterior (c_{t-1}), representa cuánto tenemos que olvidar de los valores almacenados en este.
- ▶ i se conoce como **input gate** y representa cuánto hemos de escribir en c_t . Los valores a escribir en c vienen dados por g , que viene de aplicar una *tanh* como en las RNN tradicionales.
- ▶ c_t se obtiene de la mezcla de:
 - «Cuánto queremos recordar del pasado», el producto de f por c_{t-1} .
 - Y de «cuánta información nueva queremos añadir», el producto de i por g .

- Finalmente, el *hidden state* resultante se obtiene aplicando la transformación *tanh* sobre nuestro *cell state* y multiplicando por *o* (**output gate**), que nos dice cuánto de nuestro estado interno *c* hemos de revelar en el *hidden state*.

El funcionamiento de una LSTM es un poco lioso y puede parecer más magia que otra cosa. Sin embargo, su excelente rendimiento en la práctica ha sido fundamental para la popularidad de las RNN en el mundo del *machine learning*.

Finalmente, el hecho de que las LSTM no sufran de problemas de *vanishing* y *exploding gradients* viene de que los estados internos c_t crean una especie de «autopista de la información». Si bien no lo veremos formalmente aquí, el hecho de que *c* dependa de una interacción de sumas en vez de depender directamente de los parámetros *W* hace que se evite esa especie de multiplicación continua por el mismo valor, permitiendo que el flujo de gradientes hacia atrás sea más estable.

Arquitectura GRU

Otra arquitectura que se ha hecho popular en la actualidad es la **Gated Recurrent Unit** (GRU). Fue introducida en 2014 y utiliza un sistema similar de *gates* al visto en la LSTM. Las mayores diferencias con LSTM son que se combina el *cell state* y el *hidden state* en un solo elemento, así como la *forget gate* y la *input gate* en una sola puerta.

$$\begin{aligned} r_t &= \sigma(W_{xr}x_t + W_{hr}h_{t-1} + b_r) \\ z_t &= \sigma(W_{xz}x_t + W_{hz}h_{t-1} + b_z) \\ \tilde{h}_t &= \tanh(W_{xh}x_t + W_{hh}(r_t \odot h_{t-1}) + b_h) \\ h_t &= z_t \odot h_{t-1} + (1 - z_t) \odot \tilde{h}_t \end{aligned}$$

Figura 20. Fórmula de GRU.

Fuente: http://cs231n.stanford.edu/slides/2018/cs231n_2018_lecture10.pdf

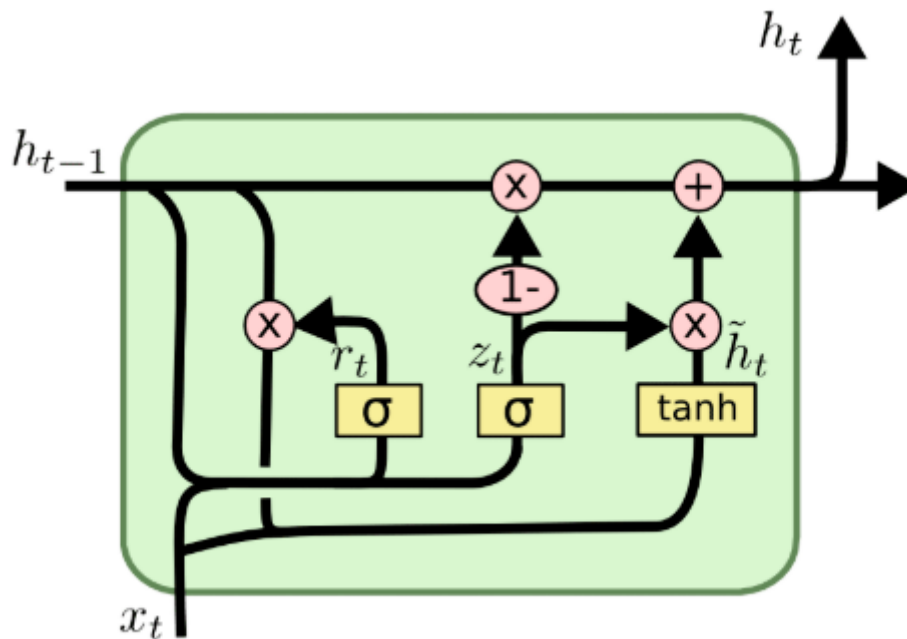


Figura 21. GRU.

Fuente: <http://colah.github.io/posts/2015-08-Understanding-LSTMs/>

Consideraciones prácticas

En la práctica, a la hora de abordar un problema con secuencias, lo más común es utilizar directamente LSTM o GRU. Estas se comportan mejor que las *vanilla* RNN en casi todas las situaciones, al no tener el problema de los *exploding* o *vanishing gradients* y ser capaces, por tanto, de modelar relaciones temporales más largas. La investigación en encontrar arquitecturas más efectivas o simples es un área muy activa, así como es el intentar comprender mejor cómo funcionan y aprenden estas redes desde un punto de vista más teórico.

Por otro lado, en la práctica suelen verse **arquitecturas profundas de RNN apiladas** (*stacked RNN*). En estas arquitecturas, la salida de una RNN se utiliza como la entrada secuencial de la RNN apilada encima de ella.

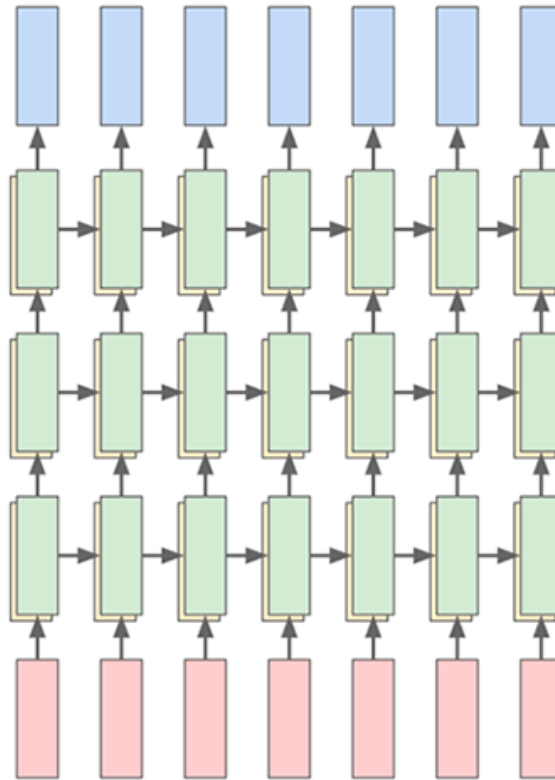


Figura 22. RNN apiladas.

Fuente: http://cs231n.stanford.edu/slides/2018/cs231n_2018_lecture10.pdf

No es habitual apilar un gran número de LSTM o GRU, las profundidades más frecuentes son entre 3 y 5 RNN. Nótese que las RNN son bastante complejas computacionalmente y su entrenamiento requiere de bastante tiempo.

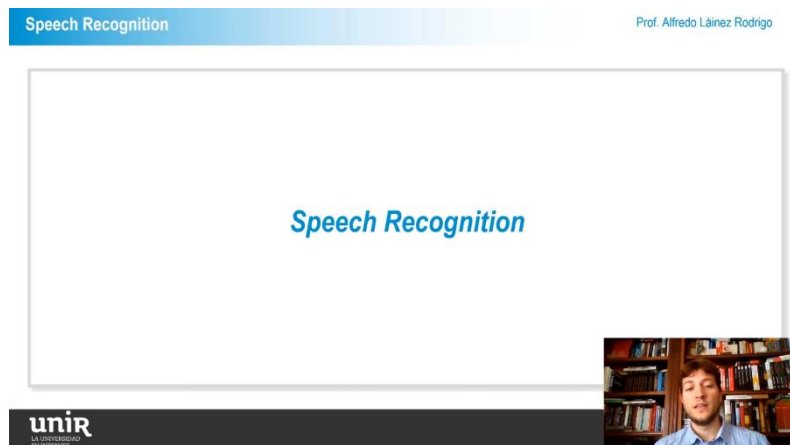
Finalmente, es también común ver **RNN bidireccionales** (*bidirectional RNN*), donde las secuencias de entrada se leen de atrás hacia delante y de delante hacia atrás.

Lo + recomendado

Lecciones magistrales

Speech Recognition

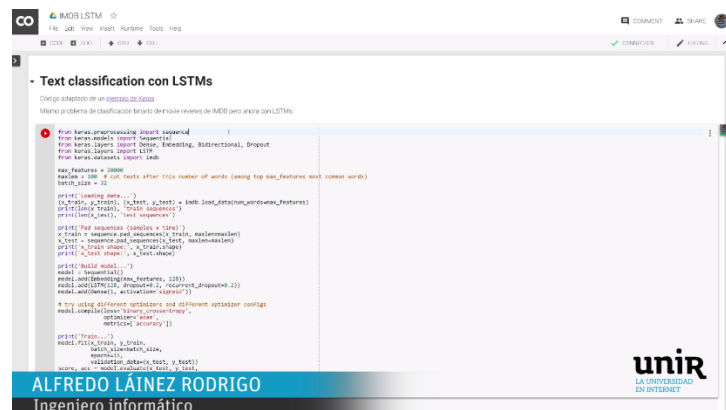
Introducción al *speech recognition* y a los sistemas de *deep learning* que han ayudado a mejorar las aplicaciones destinadas a transcribir audio a texto que se pueda entender si utilizamos otros sistemas.



Accede a la lección magistral a través del aula virtual

Clasificación de texto con RNN

Seguiremos trabajando con el problema de clasificación binario de *movie reviews* de IMDB, pero en esta ocasión con LSTMs.



```
from keras.preprocessing import sequence
from keras.datasets import imdb
from keras.layers import Dense, Embedding, LSTM, Dropout
from keras.models import Sequential
from keras.callbacks import EarlyStopping, ModelCheckpoint

max_features = 20000
max_len = 100 # cut texts after this number of words (among top max_features most common words)
batch_size = 32

print('loading data...')
(x_train, y_train), (x_test, y_test) = imdb.load_data(path='./data/IMDB dataset')
print('x_train shape:', x_train.shape)
print('y_train shape:', y_train.shape)
print('x_test shape:', x_test.shape)
print('y_test shape:', y_test.shape)

model = Sequential()
model.add(Embedding(max_features, 128))
model.add(LSTM(128, dropout=0.2, recurrent_dropout=0.2))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])

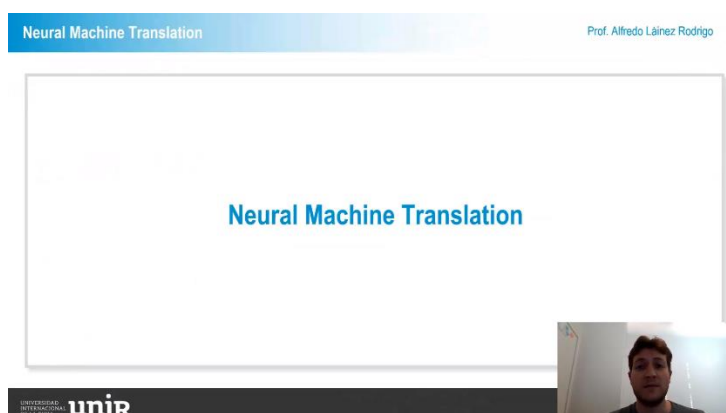
print('Train...')
model.fit(x_train, y_train, batch_size=batch_size, epochs=10, validation_data=(x_test, y_test))
```

ALFREDO LÁINEZ RODRIGO
Ingeniero informático

Accede a la lección magistral a través del aula virtual

Neuronal Machine Translation

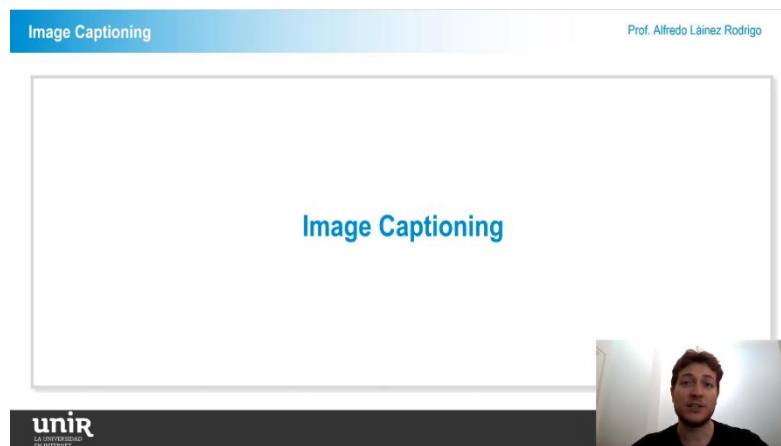
Veremos la traducción automática (*machine translation*) de textos, uno de los ámbitos donde el *deep learning* ha tenido más impacto, su funcionamiento actual y cómo las redes recurrentes han sido una mejora respecto a técnicas anteriores.



Accede a la lección magistral a través del aula virtual

Image Captioning

Combinar las redes convolucionales con redes recurrentes para describir con lenguaje qué contiene una imagen, útil para muchas tareas como la detección de objetos determinados.



Accede a la lección magistral a través del aula virtual

No dejes de leer

Understanding LSTM Networks

Olah, C. (27 de agosto de 2015). Understanding LSTM Networks [Blog post].

Magnífico post con visualizaciones para comprender mejor las redes recurrentes y las LSTM.

Accede al artículo a través del aula virtual o desde la siguiente dirección web:

<http://colah.github.io/posts/2015-08-Understanding-LSTMs/>

The Unreasonable Effectiveness of Recurrent Neural Networks

Karpathy, A. (21 de mayo de 2015). The Unreasonable Effectiveness of Recurrent Neural Networks [Blog post].

Blog post de Andrej Karpathy que trata sobre la efectividad de las RNN.

Accede al artículo a través del aula virtual o desde la siguiente dirección web:

<http://karpathy.github.io/2015/05/21/rnn-effectiveness/>

Recurrent Neural Networks Tutorial

Britz, D. (17 de septiembre de 2015). Recurrent Neural Networks Tutorial, Part 1 – Introduction to RNNs [Blog post].

Otro interesante post con un buen material gráfico para comprender las RNN.

Accede al artículo a través del aula virtual o desde la siguiente dirección web:

<http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/>

A fondo

RNN en *Deep Learning Book*

Goodfellow, I., Bengio, Y. y Courville, A. (2016). Sequence Modeling: Recurrent and Recursive Nets. En Autor, *The Deep Learning Book* (pp. 367-415). Cambridge (Estados Unidos): The MIT Press.

Capítulo sobre las RNN dentro de *Deep Learning Book*, material ya recomendado en temas anteriores y escrito por Ian Goodfellow, Yoshua Bengio y Aaron Courville.

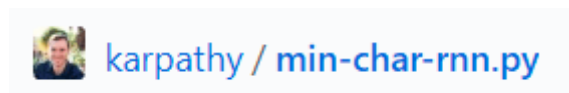
Accede al capítulo a través del aula virtual o desde la siguiente dirección web:

<http://www.deeplearningbook.org/contents/rnn.html>

Webgrafía

Char-RNN por Andrej Karpathy

RNN modelando un modelo del lenguaje a nivel de caracteres en unas 100 líneas de código.



Accede al artículo a través del aula virtual o desde la siguiente dirección:

<https://gist.github.com/karpathy/d4dee566867f8291f086>

Bibliografía

Hochreiter, S. y Schmidhuber, J. (1997). Long Short-Term Memory. *Neural Computation*, 9(8), 1735-1780. Recuperado de https://www.researchgate.net/publication/13853244_Long_Short-term_Memory

Cho, K., van Mierriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenck, H. y Bengio, Y. (2014): Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation. *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)* (pp. 1724-1734). Doha, Catar: Association for Computational Linguistics. Recuperado de <https://www.aclweb.org/anthology/D14-1179>

1. Las RNN se diferencian de las redes neuronales tradicionales en que (marca todas las respuestas correctas):
 - A. Pueden trabajar sobre una secuencia de *inputs* de manera natural.
 - B. Pueden producir una secuencia de *outputs* de manera natural.
 - C. Tienen un estado interno que permite guardar información sobre lo que la red ha visto hasta ahora.

2. En un problema de traducción automática (marca la respuesta correcta):
 - A. Tenemos un solo *input* y producimos una secuencia.
 - B. Tenemos una secuencia de entrada y producimos una sola salida.
 - C. Tenemos una secuencia de entrada y producimos una secuencia de salida.
 - D. Ninguna de las anteriores.

3. En una *vanilla* RNN, los parámetros a aprender en la red son (marca la respuesta correcta):
 - A. Las matrices W_{hh} , W_{xh} y W_{hy} .
 - B. Las matrices W_{hh} y W_{xh} .
 - C. Los estados internos h_t para cada t .
 - D. Los valores de las *input* x_t .
 - E. Ninguna de las anteriores.

4. El vector de *hidden state* h_t (marca la respuesta correcta):
 - A. Almacena todas las *inputs* que la red ha visto hasta el momento.
 - B. A menor tamaño, más capacidad de memorizar el pasado.
 - C. A mayor tamaño, la red neuronal entrenará o se ejecutará más rápido.
 - D. Es una representación interna en forma de vector que codifica los estados por los que la red ha pasado y las *inputs* que ha ido viendo en el tiempo.

5. Marca todas las respuestas correctas acerca de *backpropagation through time*:
- A. Es el equivalente al algoritmo de *backpropagation* aplicado a RNN.
 - B. En el *forward pass* calculamos todas las salidas a lo largo del tiempo.
 - C. A veces las secuencias pueden ser muy largas (por ejemplo, un texto completo para entrenar un modelo del lenguaje), por lo que se recurre a *truncated backpropagation through time*.
6. ¿Cuál es una diferencia entre los modelos del lenguaje tradicionales vistos en clase y un modelo del lenguaje con RNN? (Marca la respuesta correcta):
- A. En un modelo del lenguaje con RNN no modelamos probabilidades.
 - B. En un modelo del lenguaje con RNN el *hidden state* es la probabilidad de una palabra dadas todas las anteriores, no hace falta contar las ocurrencias de palabras.
 - C. Los modelos tradicionales utilizan redes neuronales no recurrentes.
 - D. Un modelo del lenguaje con RNN no necesita modelar probabilidades con cuentas, se utiliza un modelo de predicción en el que el estado interno de la red codifica la información necesaria para obtener las probabilidades buscadas.
7. ¿Qué ventajas puede tener utilizar *word vectors* en vez de representaciones *one-hot* como entrada en una red recurrente? (Marca todas las respuestas correctas):
- A. Ninguna, ambas son equivalentes.
 - B. Con una representación *one-hot* no podemos distinguir entre palabras distintas.
 - C. Los *word vectors* contienen información semántica de la palabra que una red neuronal puede utilizar para modelar mejor el problema a resolver.
 - D. Si el vocabulario es muy grande, una representación discreta como la *one-hot* da lugar a vectores de entrada muy grandes, mientras que una representación densa como los *word vectors* permite un vector más compacto.

8. *Gradient clipping* (marca la respuesta correcta):

- A. Controla la magnitud del gradiente durante *backpropagation*. Si este se hace muy grande, se reduce su magnitud para evitar el problema de *exploding gradients*.
- B. Controla la magnitud del gradiente durante *backpropagation*. Si este se hace muy pequeño, se incrementa su magnitud para evitar el problema de *vanishing gradients*.
- C. Controla la magnitud del gradiente durante el *forward pass*. Si este se hace muy grande, se reduce su magnitud para evitar el problema de *exploding gradients*.
- D. Controla la magnitud del gradiente durante el *forward pass*. Si este se hace muy pequeño, se reduce su magnitud para evitar el problema de *vanishing gradients*.

9. La arquitectura LSTM (marca las respuestas correctas):

- A. Surge como una arquitectura que trata de modelar con éxito relaciones temporales de larga distancia.
- B. La *input gate* (*i*) controla cuánto olvidamos del estado interno anterior.
- C. Utiliza tanto sigmoids como tanhs en sus representaciones internas.
- D. La *forget gate* (*f*) controla cuánta información fluye hacia el *hidden state*.
- E. Soluciona el problema de los *vanishing gradients*.

10. Al apilar RNN (marca la respuesta correcta):

- A. La salida del último *time step* de una RNN se pasa como primer *input* a la siguiente RNN.
- B. Solo la salida del primer *time step* de una RNN se pasa como *input* a la siguiente RNN.
- C. Las salidas de cada *time step* de la primera RNN son las entradas de la siguiente RNN.
- D. Ninguna de las anteriores.