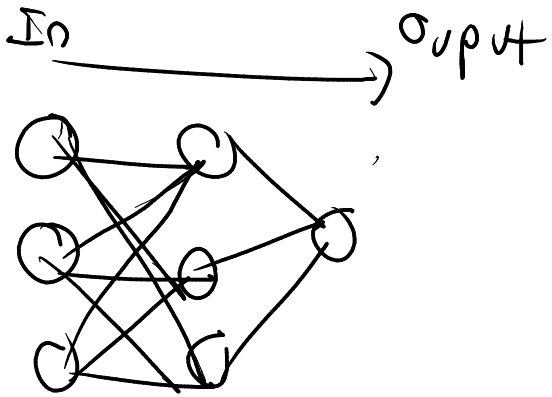
Bootcamp Inteligencia Artificial
Nivel Innovador

Semana 10:

Deep Learning

Agenda

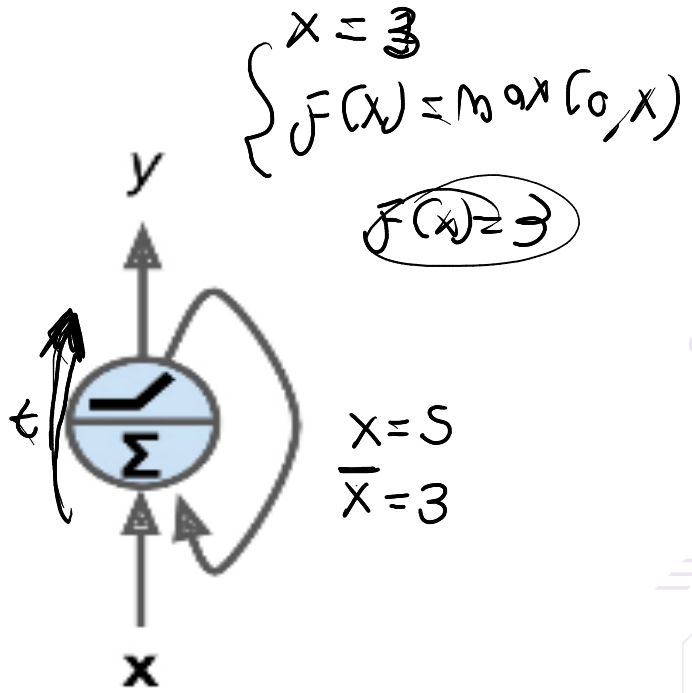


1. Redes Neuronales Recurrentes

1.1 Neuronas y Capas Recurrentes

Hasta ahora, todas las redes neuronales que hemos visto fluyen en una sola dirección: hacia adelante. La señal entra por un lado y es procesada por capas densas de neuronas, capas de convolución, capas de activación, capas de normalización, lo que gustes y mandes, y al final obtienes tus resultados del otro lado de la red.

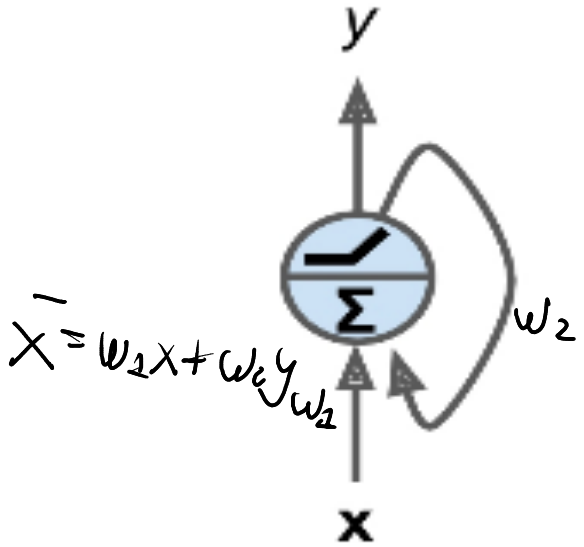
Pero ahora vamos a ver que en las Redes Neuronales Recurrentes (RNN), la señal fluye… repetidamente a través de la misma capa y las mismas neuronas. Veamos al RNN más sencillo de lo que nos podemos imaginar.



1.2 Neuronas y Capas Recurrentes

Primero, tenemos una sola neurona, con sus pesos lineales de un lado, su función de activación del otro. A esta Neurona le vamos a meter nuestra señal X – ahora en una red neuronal normal, esperaríamos que la procese su función lineal, luego su función de activación, y nos escupa los resultados por acá arriba.

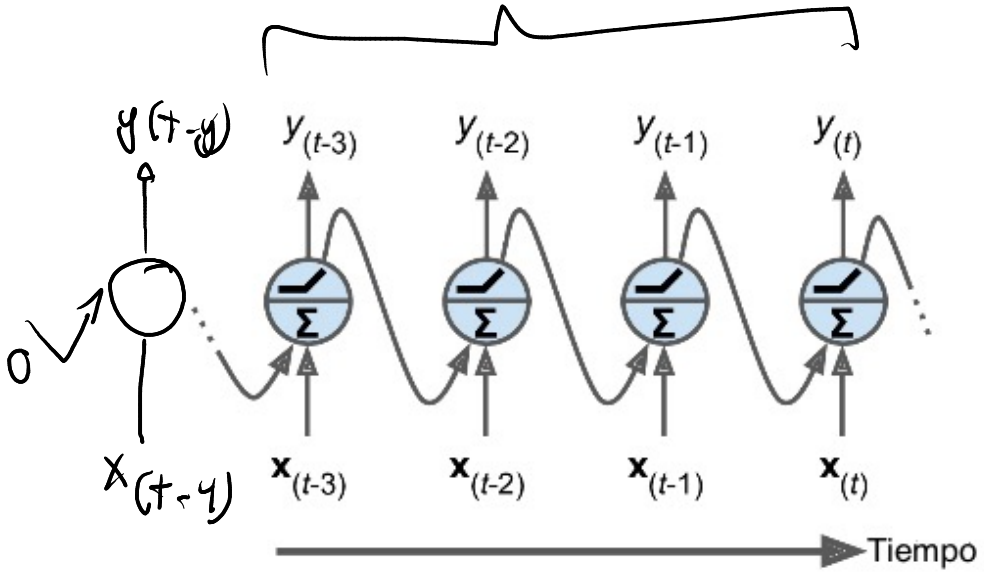
No en esta RNN – va a pasar por los procesos que mencionamos antes, y la salida que escupe esta neurona vuelve a introducirse a la neurona, una y otra y otra y otra y otra vez.



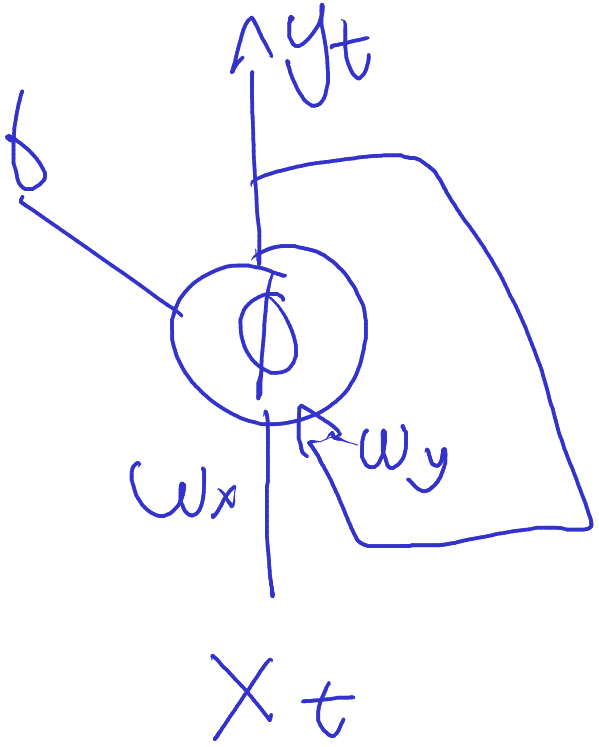
1.3 Neuronas y Capas Recurrentes

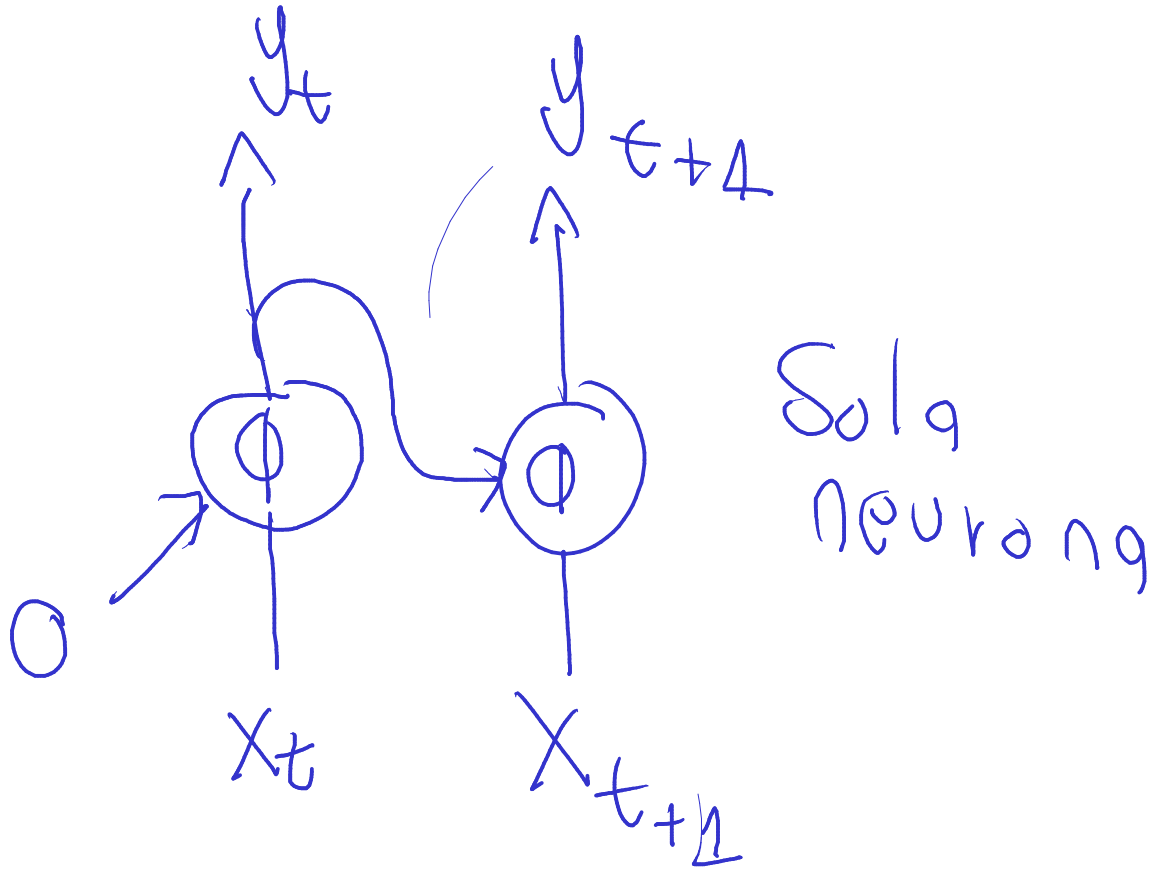
Se ve sencillo, pero representar las RNNs de esta manera no es práctico. Recuerda que al final de cuentas vas a estar leyendo papers sobre esta clase de temas, y las publicaciones científicas no van a traer animaciones profesionalmente hechas para ilustrar los más nuevos avances en RNNs.

Así que en realidad vamos a visualizar nuestros RNNs desenrollados a lo largo del tiempo: Imaginemos que haremos esta repetición 4 veces.



$$y_{t+1} = \phi(w_x x_{t+1} + w_y y_t + b)$$





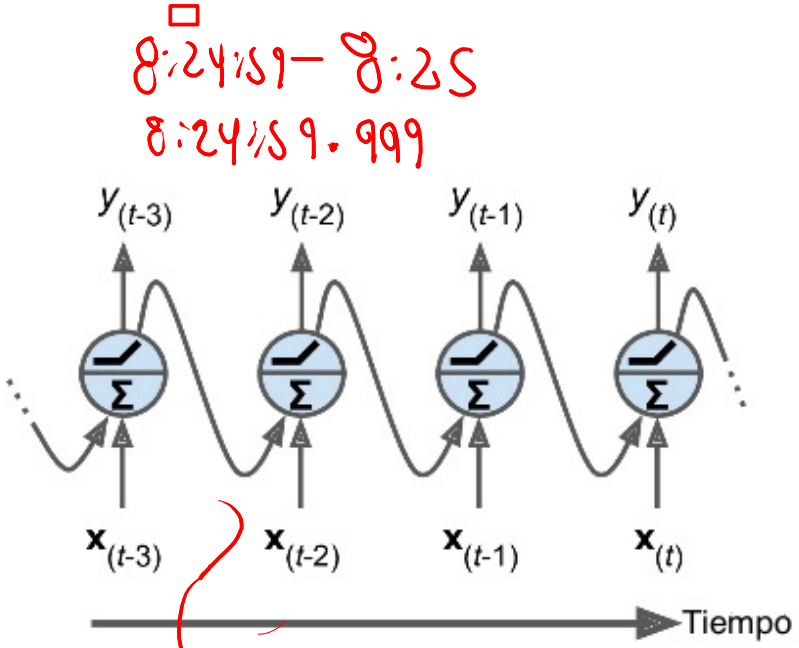
Sola neurong

1.4 Neuronas y Capas Recurrentes

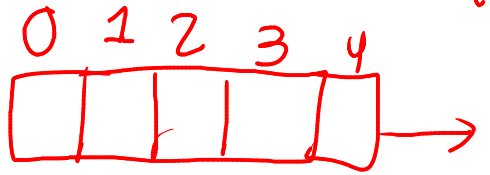
Entonces lo que haríamos sería dibujar nuestra neurona 1 vez y definiremos este dibujo como el cuadro o “frame” (en inglés, acostúmbrese que le diré frame) número t-3.

Esto por que como vamos a ciclar la señal, o repetir la señal, como quieran verlo, 4 veces, vamos a tener el cuatro t-3, t-2, t-1 y t.

Tenemos nuestro frame t-3, le metemos la señal X(t-3) y nos va a escupir el resultado Y(t-3). Ahora dibujamos la misma neurona – mismos pesos (W), misma función de activación, aquí a la derecha. Vamos a meterle de nuevo nuestra señal original X (aunque ahora será t-2) y también recibirá la salida del frame anterior. Esto es lo mismo que, en el diagrama original, ciclar nuestra señal a través de la misma neurona.



$\bar{X}_{t-2} = y_{(t-3)} \times w_i + x_{(t-2)} w_j$



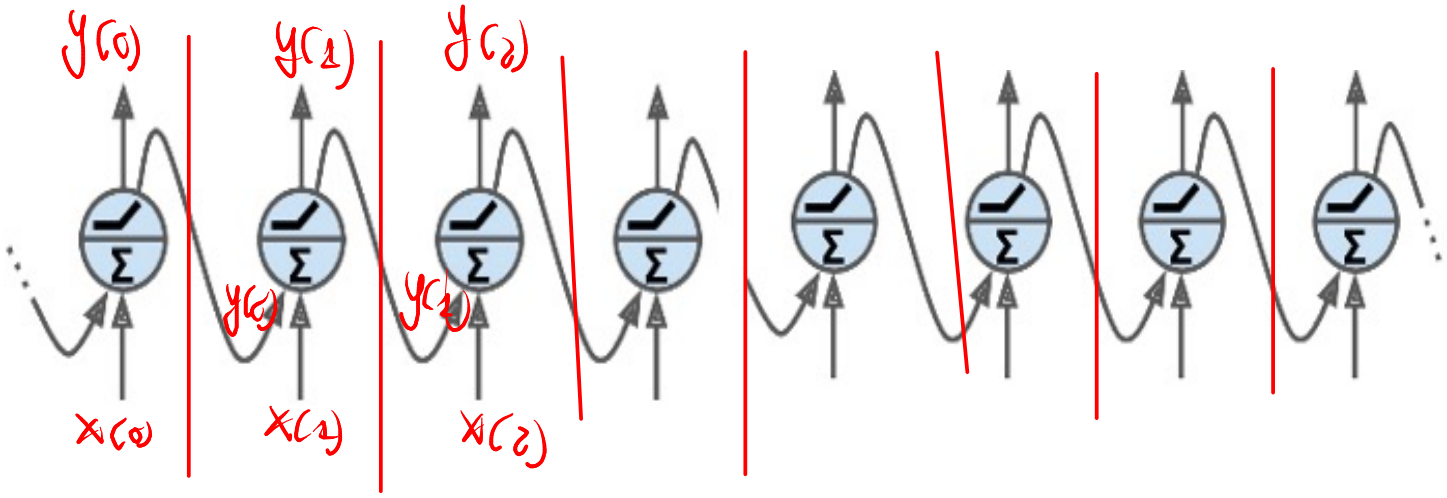
For i in range(0,10)

$i = 0, 1, 2, \ldots, 9$

1.5 Neuronas y Capas Recurrentes

Volviendo al frame t-2, la neurona nos va a escupir la señal. Pasamos ahora al frame t-2, donde tenemos nuestra misma neurona, y vuelve a recibir la exacta misma entrada X y aparte, el resultado del frame anterior t-3 … y por último repetimos el proceso en el momento t.

La exacta misma neurona, recibe la exacta misma entrá X y aparte el resultado de t-1 y finalmente nos escupe una salida. Podríamos, si quisiéramos, repetir este proceso más de 4 veces, en cuyo caso solo tendríamos que seguir agregando neuronas a nuestro diagrama.

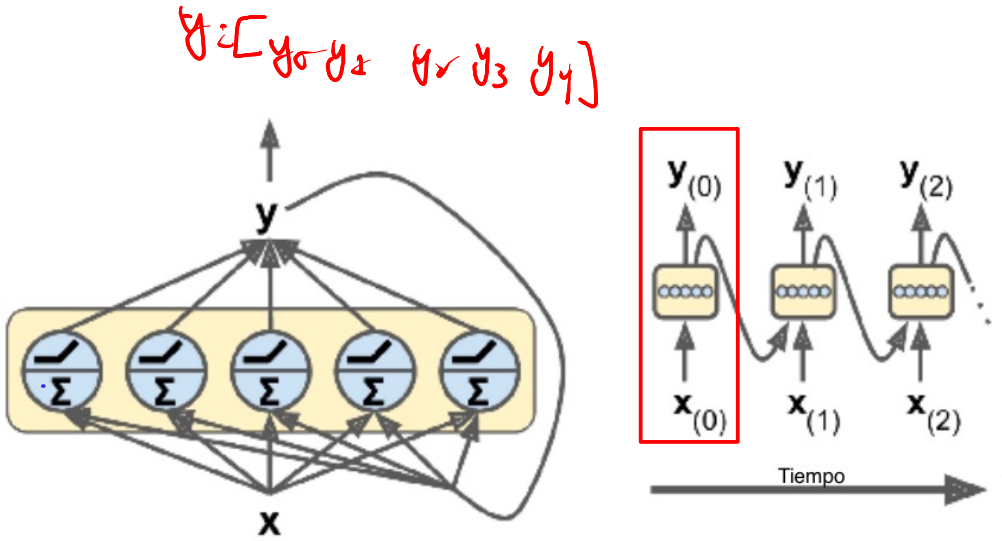


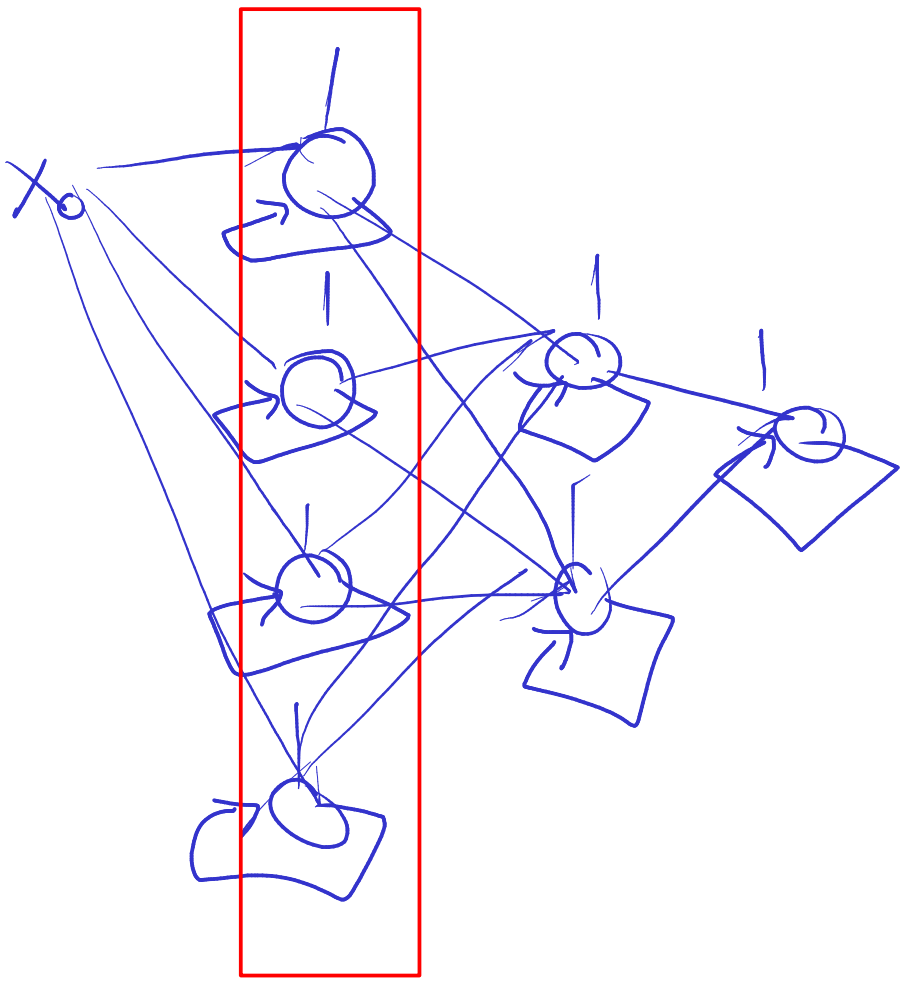
1.6 Neuronas y Capas Recurrentes

Aunque en esencia, es solo estar ciclando la señal a través de la misma neurona tantas veces como lo creamos necesario. Este modelo es sencillo – solo es una neurona. Vamos a echarle un poquito de peligro.

¿Qué pasa si yo quiero una capa entera con 5 neuronas? Fácil, voy a plantear mis 5 neuronas aquí.

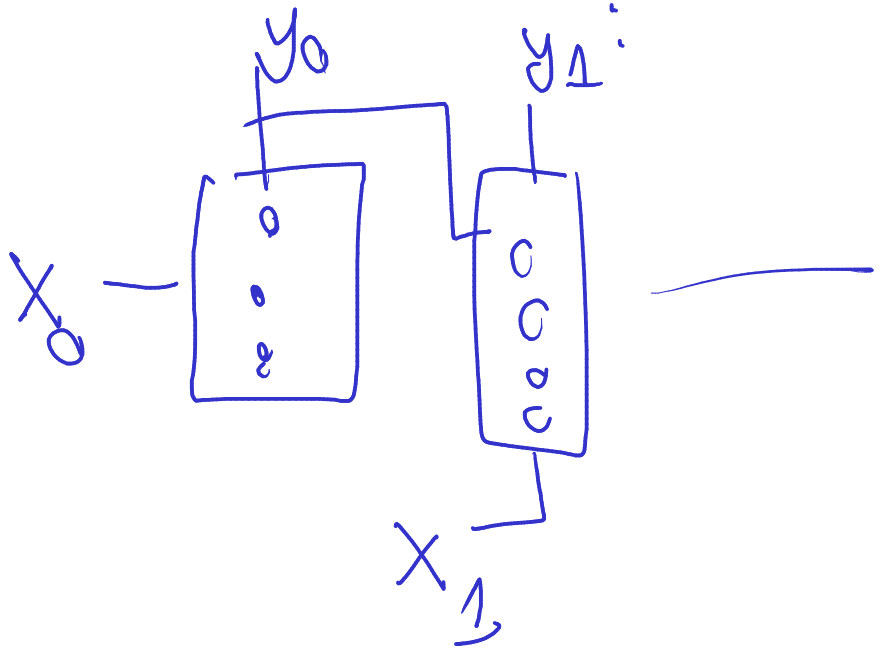
Ingresando toda a la misma entrada X – las neuronas las procesaran con sus respectivos pesos W y su función de activación, y obtendremos nuestra salida Y. Esta salida es un vector.





$$y_0 = \phi(w_x x_0 + b + 0\, w_y)$$

$$y_1 = \phi(w \times x_1 + b + y_0 w_y)$$

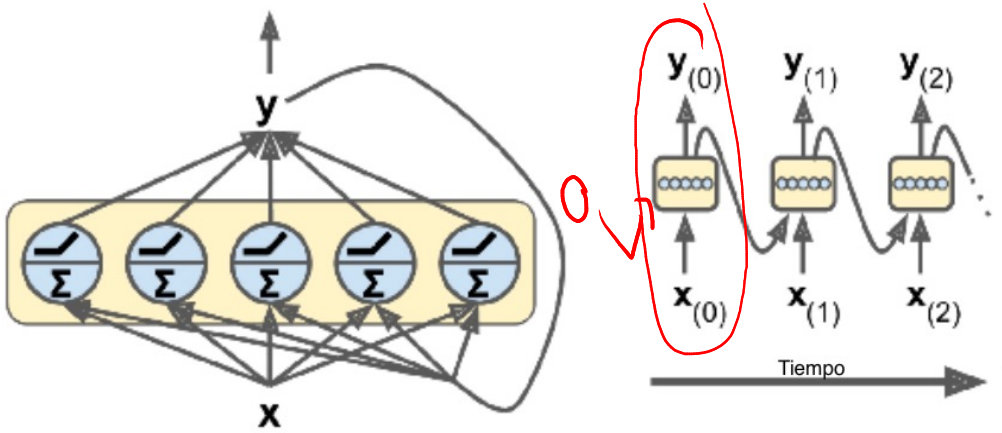


1.7 Neuronas y Capas Recurrentes

Para este primer ciclo obtendremos nuestro vector de salida Y, y lo usaremos para volver a alimentar a nuestras mismas neuronas. Y entonces comienza el ciclo de repetición, donde en cada fase, estaremos alimentando las entradas con el mismo vector de salida Y. Así repetimos hasta que terminemos todos los ciclos de repetición.

¿Cómo dibujamos esta clase de redes neuronales recurrentes en su diagrama por frames? Fácil, en vez de poner la neurona, simplemente diagramamos la capa. El proceso es el mismo, solo que ahora, por decisión mía totalmente, en vez de usar la nomenclatura t-1, vamos a comenzar en el paso 0.

Así que esta es nuestra capa en el paso 0, entra el vector X y sale el Vector Y0 – luego pasamos a la misma capa en el frame 1, donde entra el vector X1, y la salida anterior que habíamos obtenido (Y0).



$$f(wx_{(0)}^T) = y_{(0)}$$

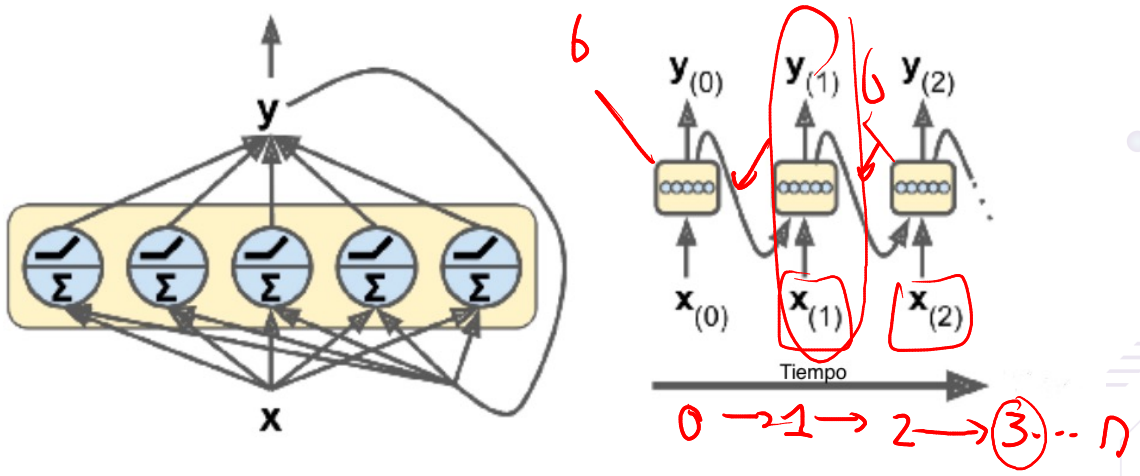
$$f(wx_{(1)}^T + w_1 y_{(0)}) = y_{(1)}$$

1.8 Neuronas y Capas Recurrentes

Y luego pasamos a Y2 donde hacemos el mismo procedimiento, y así consecutivamente hasta completar el número de repeticiones necesarias.

Cada neurona recurrente tiene 2 sets de pesos: una para las entradas x_yotra para las salidas que recibió del paso de tiempo anterior y_((t-1)). Los vectores se van a llamar los pesos w_x y w_y

Si consideramos toda la capa recurrente entonces podemos colocar todos los pesos de cada una de las neuronas en 2 matrices de pesos W_x y W_ypero con mayúsculas.



$$y(2) = f(W_x x(2) + W_y y(1))$$

$$y(3) = f(W_x x(3) + W_y y(2))$$

1.9 Neuronas y Capas Recurrentes

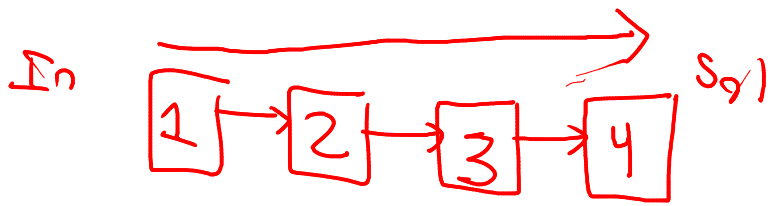
El vector de salida de toda la capa recurrente entonces puede ser computado como lo vemos en la siguiente ecuación:

$$y_t = \phi(W_x^\intercal x_t + W_y^\intercal y_{(t-1)} + b)$$

Agarramos los valores de W_x y obtenemos su producto punto con respecto a las entradas de x. Luego agarramos los valores de W_y y obtenemos su producto punto con y(t-1). Sumamos el resultado de esas 2 operaciones y aparte le sumamos el término de bias que nunca puede faltar.

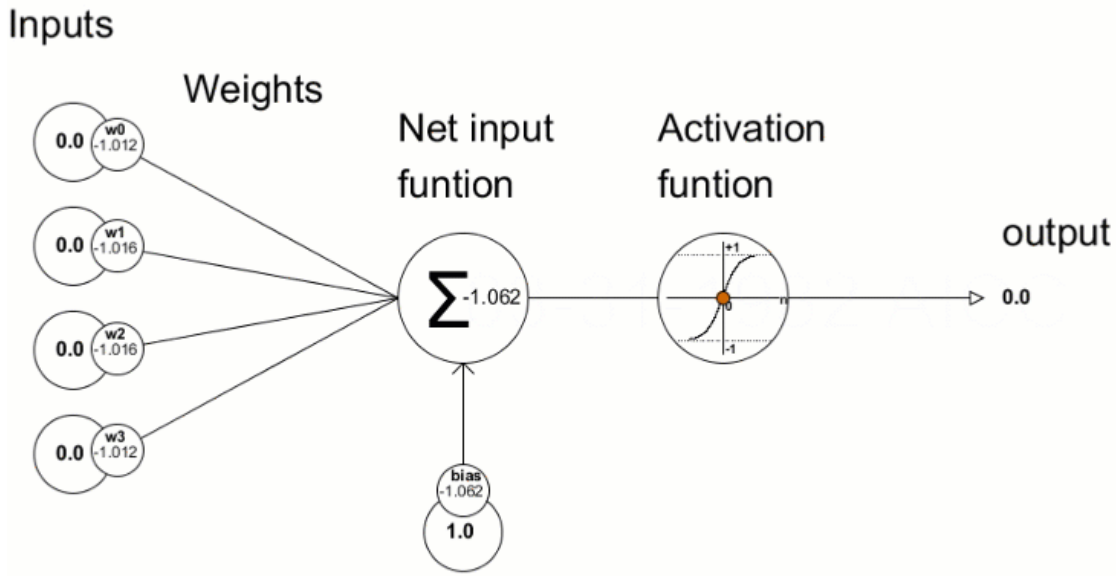
Al final, obtenemos y_t de resultado, que se utilizará en la siguiente iteración del ciclo para alimentar a la misma capa.

1.10 Celdas de memoria



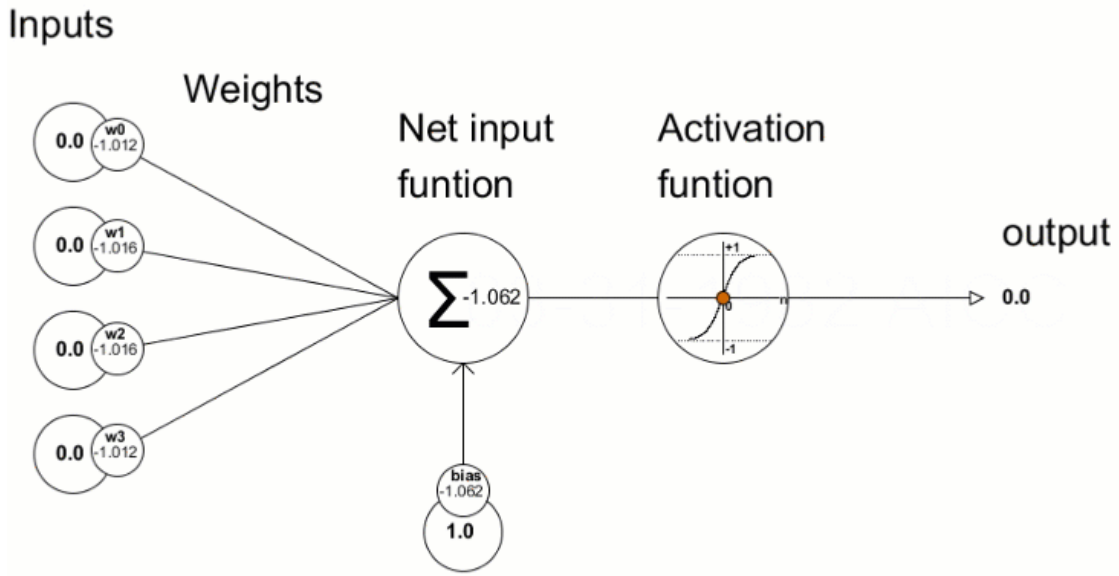
Vamos pensando en la última neurona de una red neuronal convolucional. Su resultado va a depender de la entrada X que le metamos, y al mismo tiempo de la entrada Y que viene de la neurona anterior.

Asimismo, el resultado de esa neurona anterior va a depender de la entrada que le meta la neurona anterior, o sea esta neurona que está a 1 paso removida también está influyendo nuestro resultado de la neurona final. Y, al mismo tiempo, el resultado de la neurona anterior anterior va a depender de lo que sea que le pase la neurona que va en el paso antes de este, y así consecutivamente.



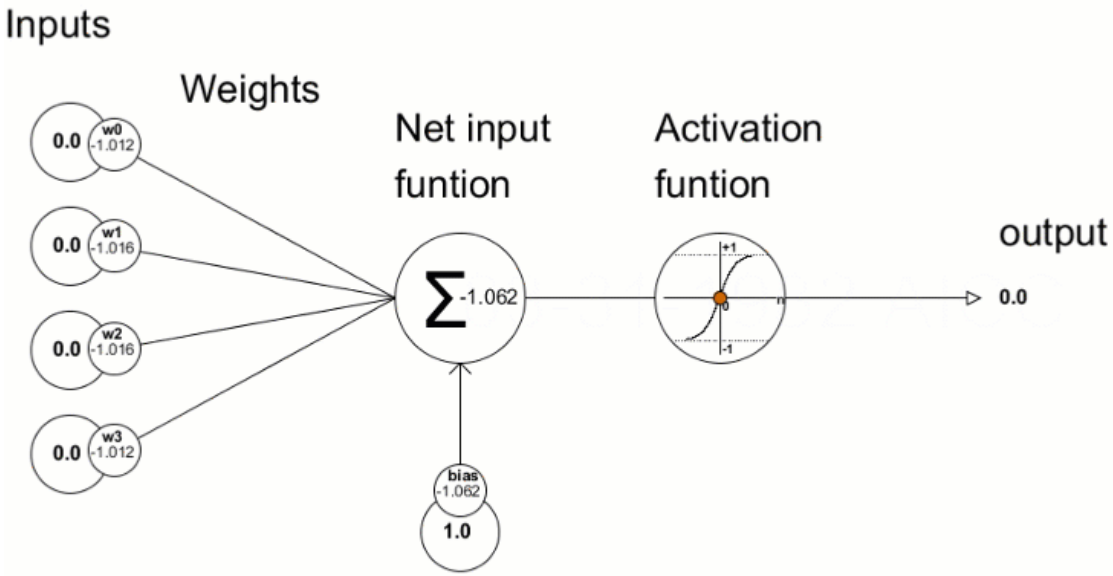
1.11 Celdas de memoria

El resultado de la neurona final va a ser influenciado por todas las neuronas, o todos los pasos anteriores, cada vez en menor medida claro, pero la influencia está ahí. Incluso si tenemos una red neuronal recurrente de 20 ciclos, la neurona que representa el ciclo uno le manda una señal a nuestra última neurona que, muy débilmente, está representada en nuestra salida.



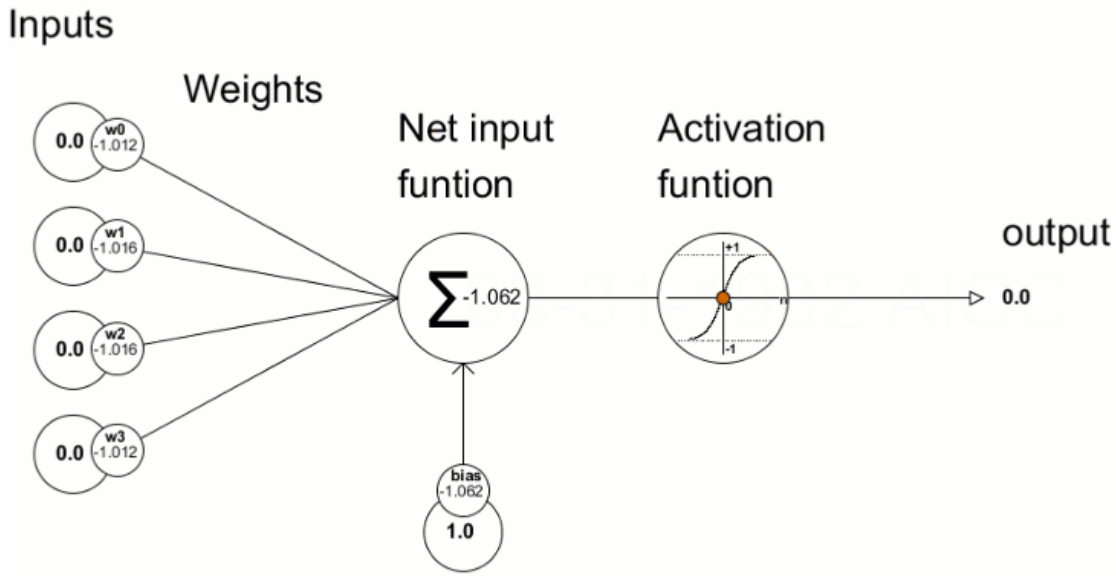
1.12 Celdas de memoria

Tomemos nuestra neurona – ya sabemos que la entrada es x y la salida es y. Pero el estado oculto, ese ciclo de volverle a meter su salida en sí misma, la conoceremos como h_t que es el estado de h en algún momento t (recuerda que la t denota el ciclo de recurrencia en el que vamos) es una función de algunas entradas en ese momento (como x_t) y su estado en el momento previo h_(t-1) entonces podríamos decir que h_t=f(h_(t-1) ,x_t).



1.13 Celdas de memoria

Su salida y_t, es una función del estado previo y sus entradas actuales. ¿Por qué todo este lío con el álgebra? Porque en las celdas básicas que vimos en la introducción, la salida y simplemente es igual al estado h, ambas son salidas de la neurona aplicando su función lineal de pesos y su función de activación. Pero en unos momentos veremos que NO en todos los casos aplica que y es igual a h.

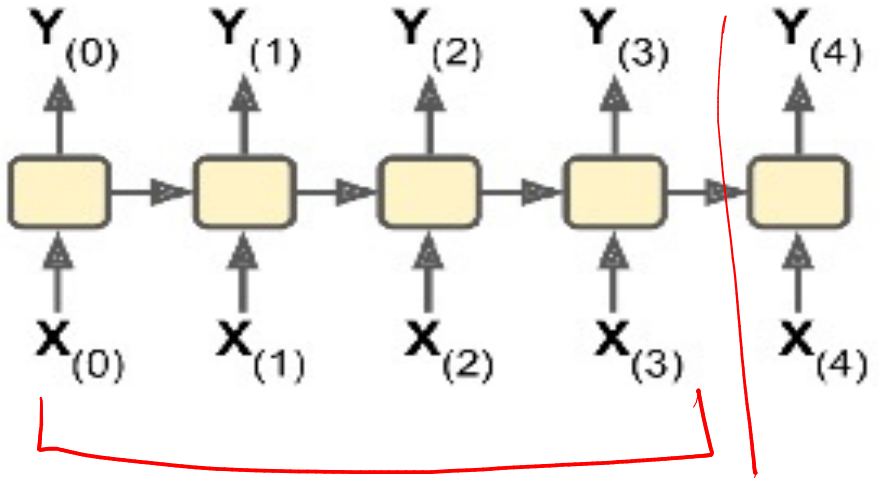


1.14 Secuencias de Entrada y de Salida

Imaginemos que estamos intentando utilizar nuestras redes neuronales para predecir el mercado. Tenemos la lista de precios de Amazon (AMZN) a lo largo de los últimos 5 días de trading y queremos que la red neuronal recurrente nos diga cómo va a amanecer al día siguiente.

Para entrenar nuestra red neuronal le meteremos una secuencia de precios a lo largo de los últimos 5 días, y esperaríamos de regreso los precios, pero adelantados un día hacia el futuro, o sea desde hace 4 días hasta mañana.

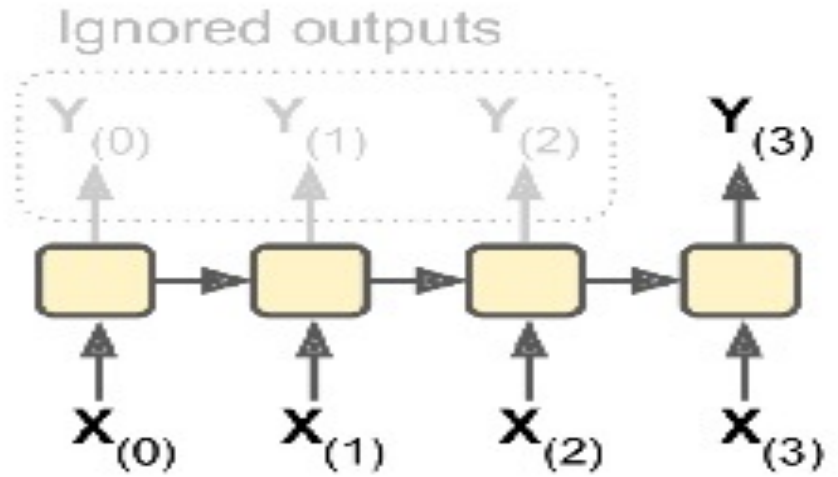
Esta clase de RNN básicas son las redes neuronales que se les conoce como secuencia a secuencia. Le metemos una secuencia, le sacamos una secuencia.



1.15 Secuencias de Entrada y de Salida

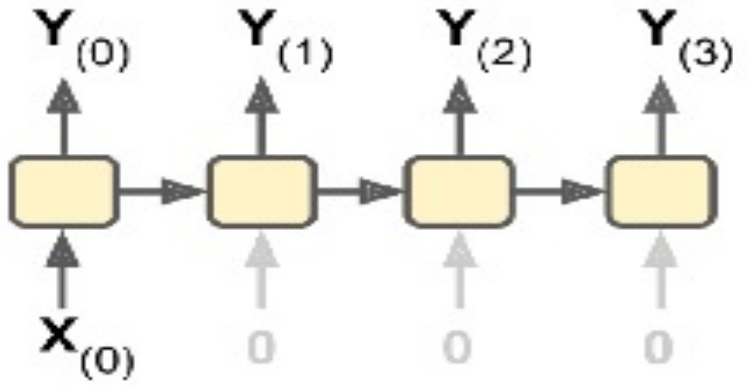
Por otro lado, tal vez queremos hacer un medidor de sentimiento – alimentarle, por ejemplo, las noticias del día y que nos diga si son noticias positivas (+1) o negativas (-1). En este caso usaríamos una arquitectura que se llama secuencia a vector.

Lo que hacemos es alimentar la RNN con una secuencia de entradas e ignoramos todas las salidas excepto la última. Al final de cuentas solo necesitamos una conclusión – si la noticia es buena o mala.



1.16 Secuencias de Entrada y de Salida

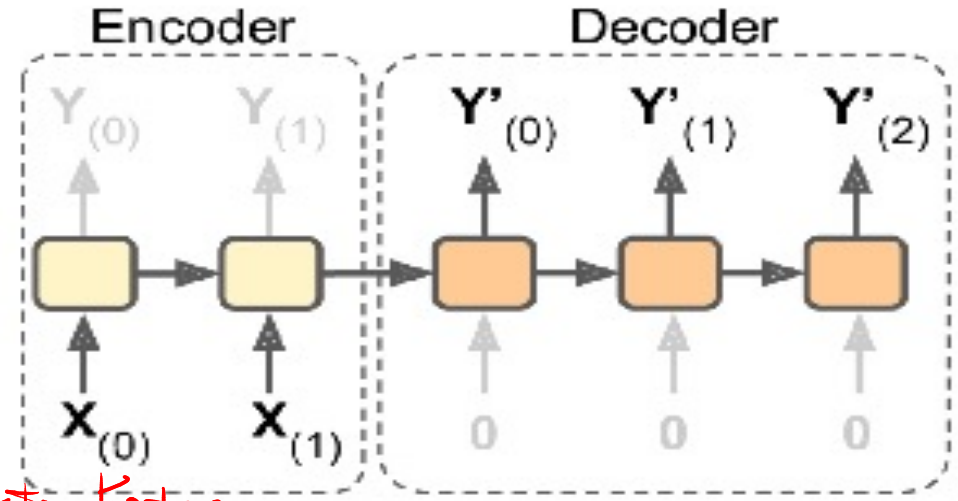
Ahora, vamos a suponer que queremos hacer lo contrario – vamos a alimentarle un vector una sola vez y dejar que saque una secuencia. Este modelo de vector a secuencia sería útil si le damos una imagen y queremos que nos escupa una descripción de la imagen.

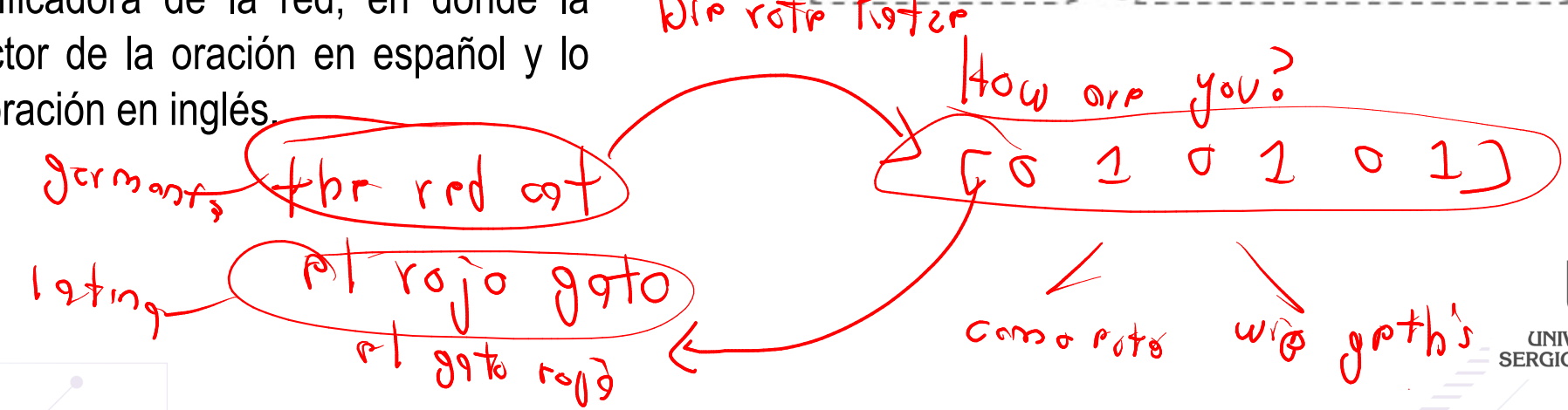


1.17 Secuencias de Entrada y de Salida

Y, por último, viene la arquitectura más interesante de todas: los codificadores – decodificadores.

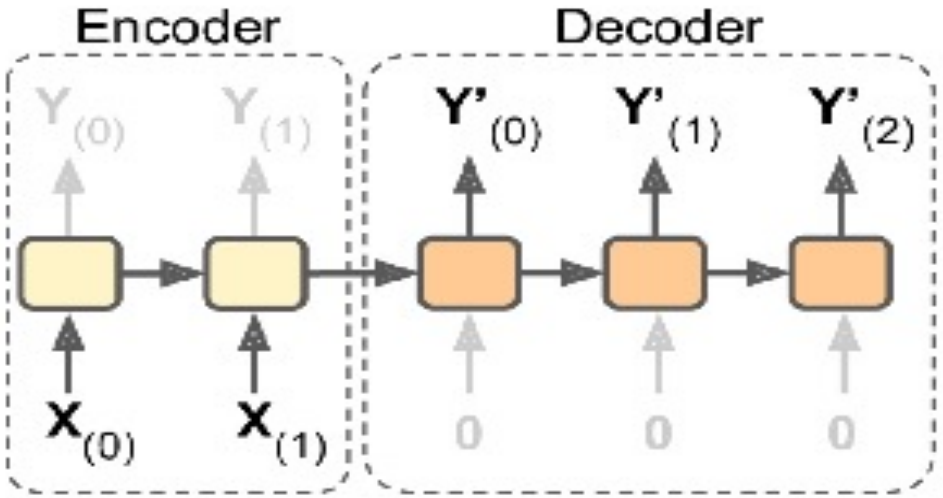
Esta estructura se usa para hacer traducciones. En la parte de codificación de la RNN le daríamos una oración en español, por ejemplo, y el RNN la convertiría en un simple vector. Después la pasaría a la sección descodificadora de la red, en donde la RNN agarra el vector de la oración en español y lo decodifica en una oración en inglés.



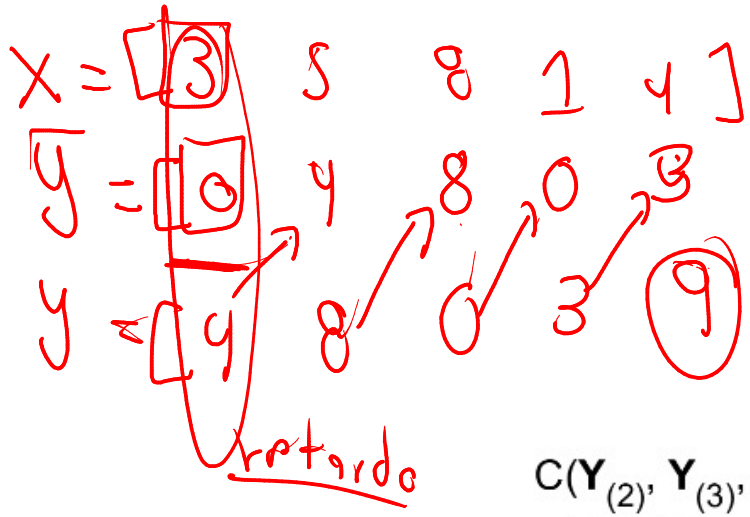


1.18 Secuencias de Entrada y de Salida

Esto funciona mucho mejor que intentar traducir en tiempo real palabra por palabra porque la gramática y estructura de los idiomas son diferentes. Por ejemplo, en el alemán la palabra que dices al final de una oración puede cambiar completamente el significado del mensaje – así que tienes que esperarte al final de la oración para asegurarte que captaste todo.

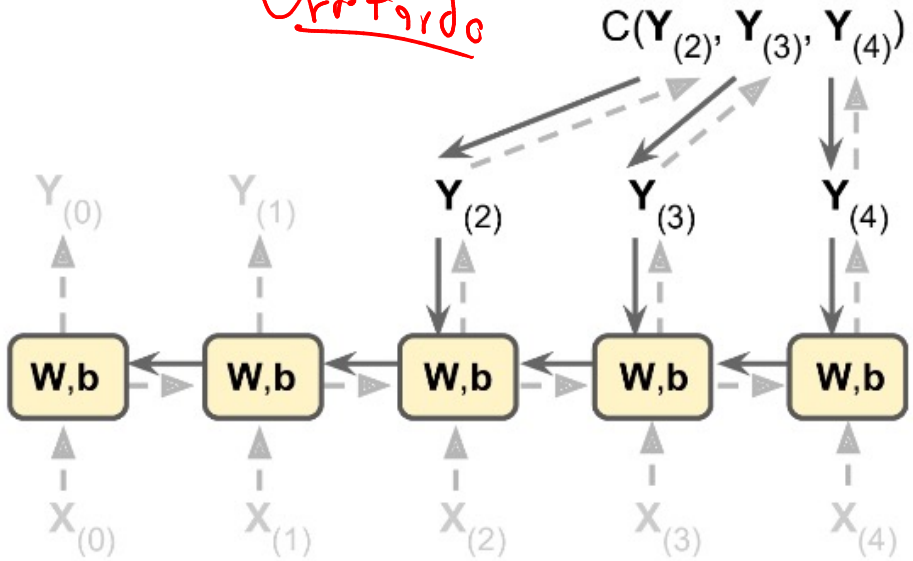


1.19 Entrenando RNNs



Entrenar un RNN se hace igual que para entrenar cualquier otra red neuronal – se usa backpropagation.

Dado que las RNNs son únicas en el sentido de que la señal se está reciclando constantemente, lo que en realidad haremos es desenrollar el RNN a través del tiempo e imaginar que su figura real es esta:

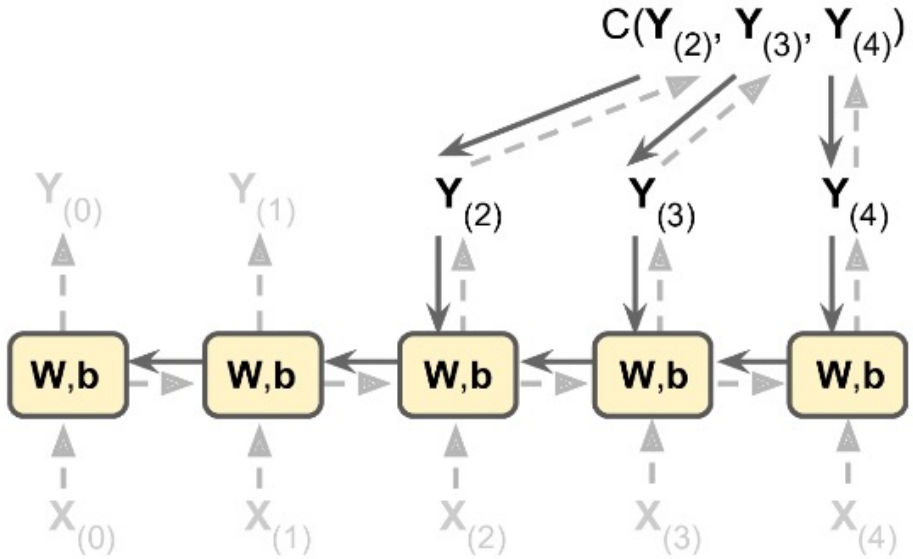


Una vez que lo estamos viendo de esta manera, se vuelve muy similar a una red neuronal común y podemos aplicar la técnica de Backpropagation a través del tiempo.

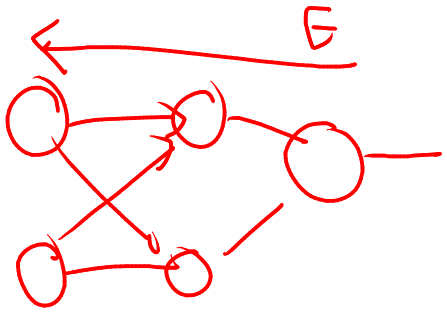
1.20 Entrenando RNNs

Al igual que en la backpropagation regular, primero hay un pase hacia adelante a través de la red neuronal desenrollada. Luego la secuencia de salida es evaluada usando alguna función de costo (como MSE) para cada una de las salidas que obtuvimos.

En un modelo de secuencia a secuencia, la función de costo va a tomar en cuenta todas las salidas, mientras que en un modelo se secuencia a vector, la función de costo puede ignorar muchas de las salidas Y.



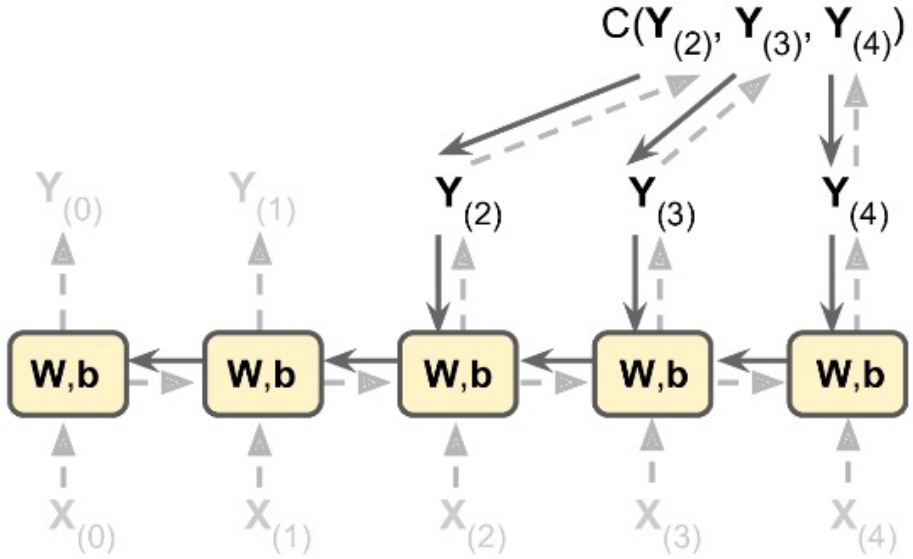
1.21 Entrenando RNNs



Una vez que tenemos nuestro error, los gradientes de esa función de costo se propagan de regreso (se backpropagan en un pochismo asqueroso) a través de la red desenrollada.

Finalmente, actualizamos los parámetros de nuestra red (las W y las B) usando los gradientes que computamos usando Backpropagation a través del tiempo.

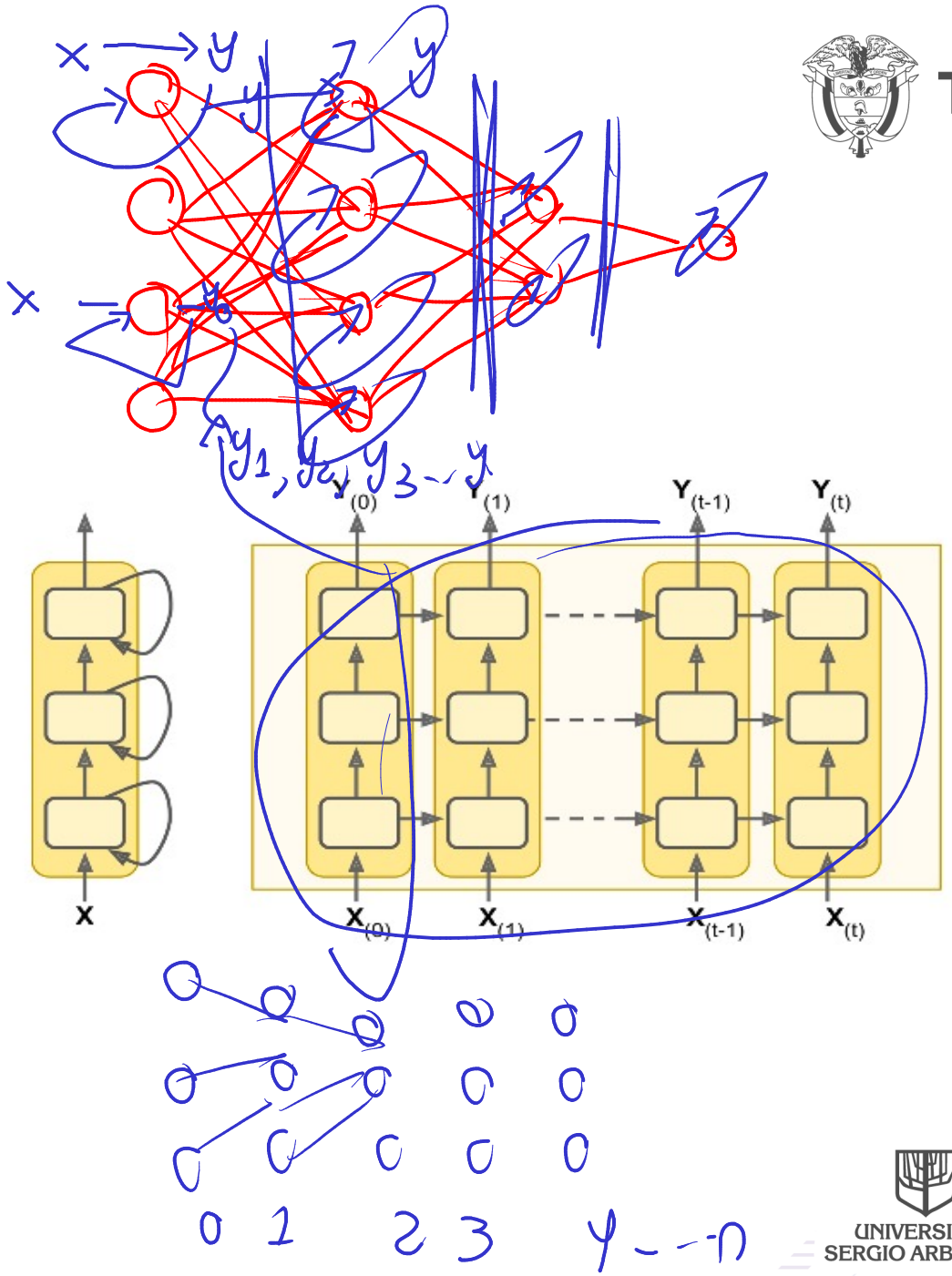
Nota que los gradientes fluyen de regreso a través de todas las salidas usadas por la función de costo, no solo a través de la salida final.



1.22 RNN profundo

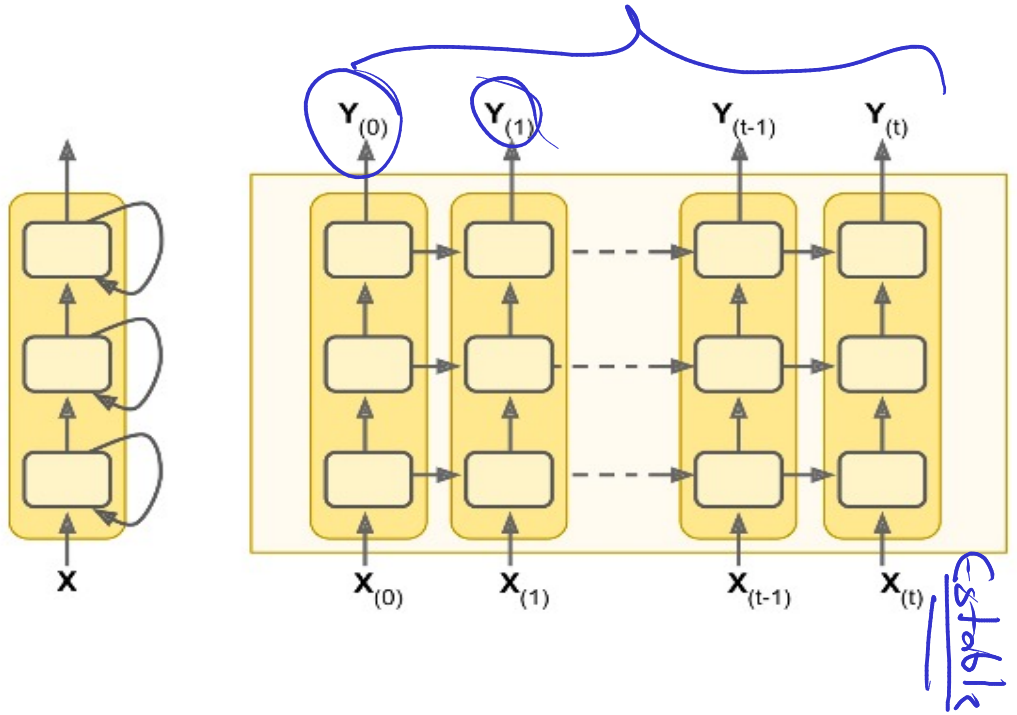
Ya vimos que un RNN puede ser desde una simple y sencilla neurona que vive reciclando sus salidas tantas veces como se lo pidamos. Vimos que también podemos acumular varias de estas neuronas al mismo tiempo para hacer una capa de neuronas recurrentes, donde, igual que antes, todas van a reciclar sus salidas.

Si apilamos varias capas de estas una encima de otra, vamos a tener una Red Neuronal Recurrente profunda. Su funcionamiento es igual que lo que hemos visto anteriormente. Las salidas se reciclan antes de crear una salida definitiva que servirá de entrada para la siguiente capa.



1.23 RNN profundo

Si desenrollamos nuestro modelo para verlo a lo largo del tiempo, se verá de la siguiente manera, donde podemos ver como cada capa alimenta nuestra siguiente capa en la pila, y cada capa se alimenta a su misma versión, pero un paso en el futuro.

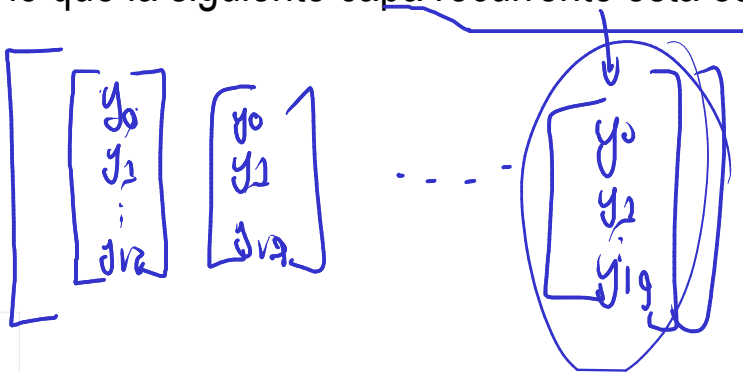


1.24 Manejar Secuencias Largas

En este ejemplo usamos tres capas de Simple RNN:

```
model = keras.models.Sequential([
    keras.layers.SimpleRNN(20,return_sequences=True, input_shape=[None,1]),
    keras.layers.SimpleRNN(20,return_sequences=True, input_shape=[None,1]),
    keras.layers.SimpleRNN(1)
])
```

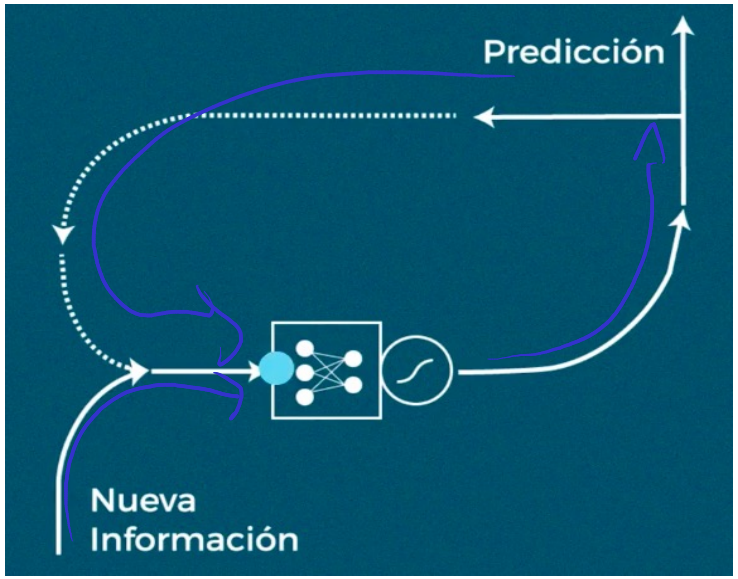
Una nota importante es este parámetro de Return Sequences – debe de ser True para todas las capas recurrentes. Cuando es verdad, la capa nos va a escupir un arreglo 3D que contiene las salidas de todos los pasos de tiempos – esto es lo que la siguiente capa recurrente está esperando como entrada.



1.25 El Problema de Memoria de Corto Plazo

Debido a las transformaciones que los datos pasan cuando atraviesan un RNN, se va a perder información en cada paso del tiempo. Después de un rato, el estado actual del RNN no contiene ni una pizca de las primeras entradas.

Supongamos que tienes un RNN que vas a usar para traducción. Ahora, habíamos dicho que para traducir en redes neuronales no se hace palabra por palabra, primero se lee toda la oración y luego qué se ingirió toda, se traduce completa. Esto para asegurarse que se captó todo el significado y contexto de lo que se quiere decir.



1.26 El Problema de Memoria de Corto Plazo - LSTM

LSTM

Propuesta en 1997 por Sepp Hochreiter y Jürgen Schmidhuber, LSTM significa Long Short-Term Memory o Memoria de Largo y Corto Plazo en español, (pero le diremos LSTM) a partir de ahora. Vamos a comenzar fingiendo que LSTM es una caja negra que nos ayuda a resolver el problema de la memoria de largo plazo.

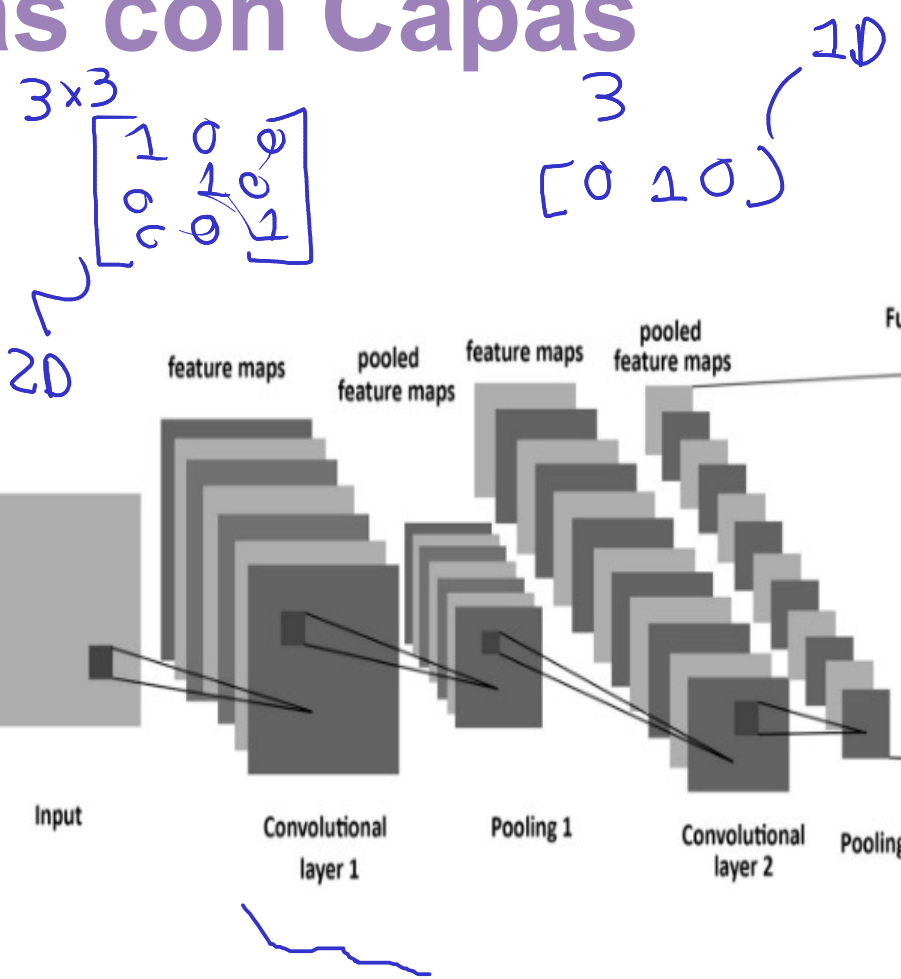
Keras ya trae su propia integración de LSTM y es muy sencilla de aplicar:

```
model = keras.models.Sequential([
    keras.layers.LSTM(20, return_sequences=True, input_shape=[None, 1]),
    keras.layers.LSTM(20, return_sequences=True),
    keras.layers.TimeDistributed(keras.layers.Dense(10))
])
```

1.27 Procesando secuencias con Capas convolucionales de 1D

Una capa convolucional de 2D desliza varios filtros a través de una imagen, produciendo varios feature maps en 2 dimensiones (uno por filtro). Asimismo, una capa convolucional de 1D desliza varios kernels a través de una secuencia, produciendo un feature map de 1D por filtro.

Cada filtro va a aprender a detectar un patrón secuencial muy corto (del tamaño del filtro) – si usas 10 filtros, entonces la salida de la capa va a estar compuesta de 10 secuencias unidimensionales – o puedes verlo como una sola secuencia de 10 dimensiones.

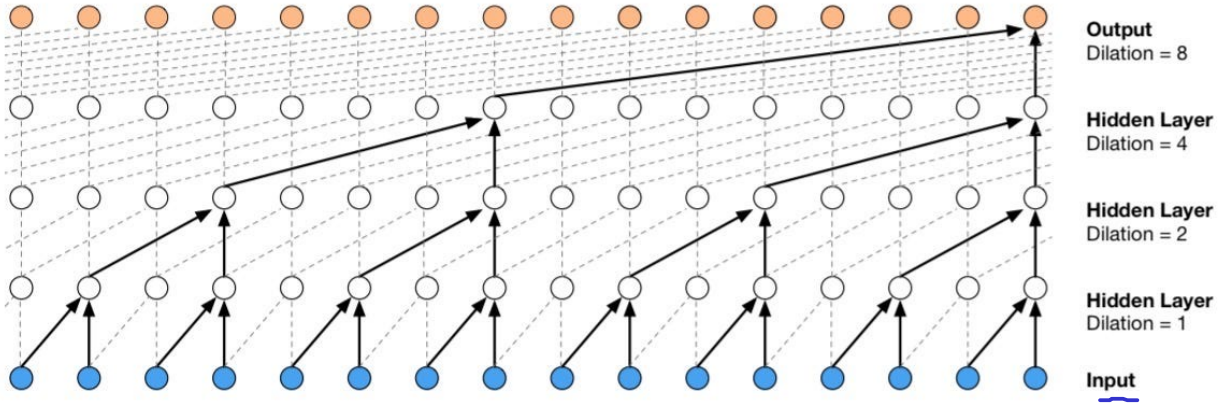


1.28 Procesando secuencias con Capas convolucionales de 1D - Wavenet

Los genios de Deep Mind, Aaron van den Oord y compañía introdujeron en el 2016 una arquitectura llamada Wavenet.

Apilaron capas convolucionales de 1D, duplicando su tasa de dilación en cada capa, la primera capa convolucional recibe una muestra de solo dos pasos de tiempo a la vez, mientras que la siguiente ve cuatro pasos de tiempo, la siguiente ocho, y así.

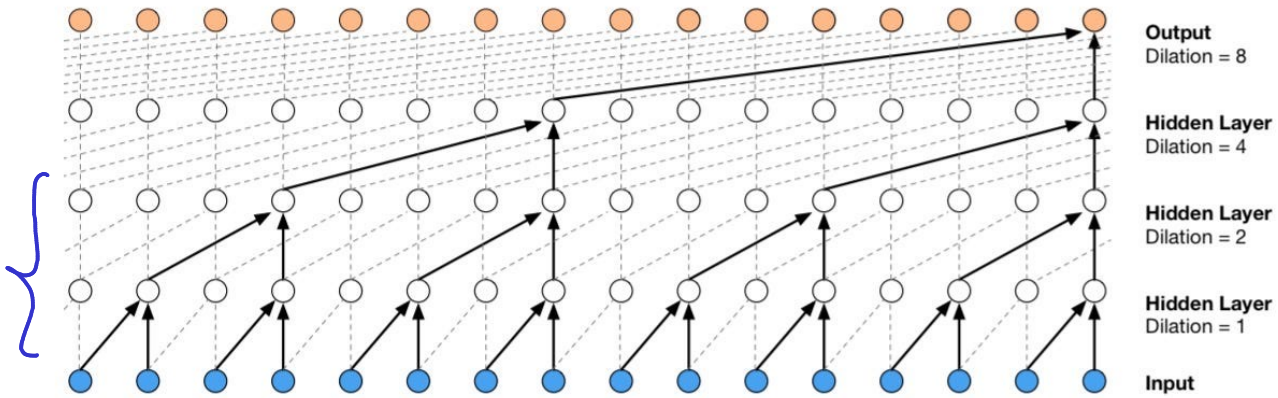
De esta manera, las capas bajas aprenden patrones de corto plazo y las capas altas aprenden patrones de largo plazo.

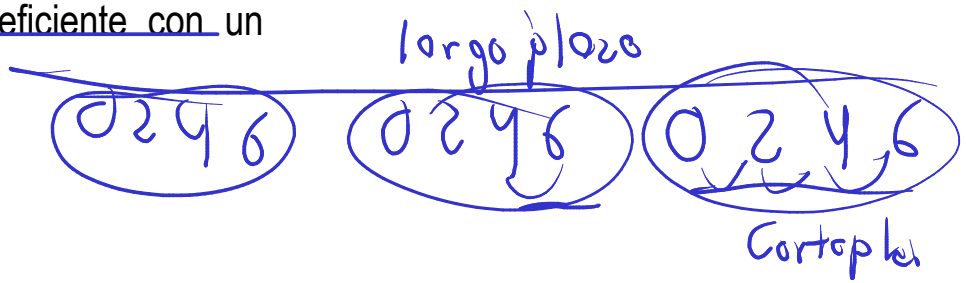


1.29 Procesando secuencias con Capas convolucionales de 1D - Wavenet

Gracias a la duplicación de la tasa de dilación, la red puede procesar secuencias extremadamente grandes muy eficientemente.

Al apilar 10 capas convolucionales con tasa de dilación de 1,24,8, 256, 512, luego otra piola de 10 capas idénticas, y luego otra pila con otro grupo idéntico de 10 capas. Justificaron esta arquitectura señalando que cada stack de 10 capas convolucionales de 10 capas, estas tasas de dilación van a actuar como una capa convolucional súper eficiente con un Kernel de 1024, por lo que apilaron 3 bloques.





Preguntas