Redes Neuronales Recurrentes

BOOTCAMP INTELIGENCIA ARTIFICIAL INNOVADOR

AÑO 2024

BOOTCAMP INTELIGENCIA ARTIFICIAL EXPLORADOR

Redes Neuronales Recurrentes

- ### 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.

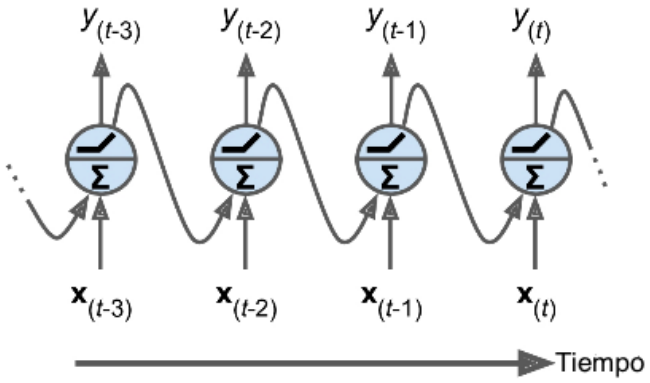
Pero ahora vamos a ver que en las Redes Neuronales Recurrentes (RNN), la señal fluye… repetidamente a través de la misma capa y las mismas neuronas. Veamos al RNN más sencillo de lo que nos podemos imaginar.

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.

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.



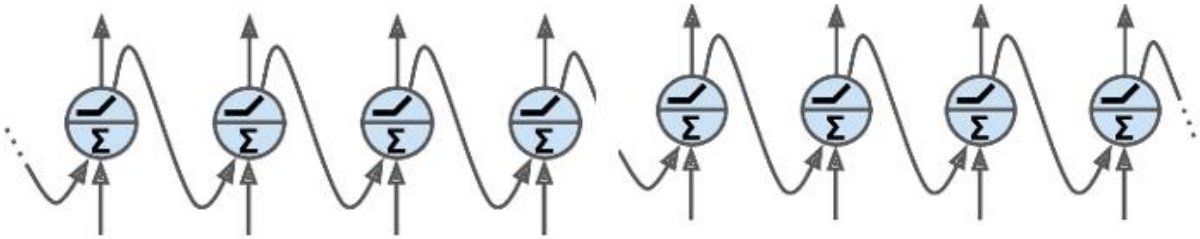
Entonces lo que haríamos sería dibujar nuestra neurona 1 vez y definiremos este dibujo como el cuadro o "frame" (en inglés, acostúmbrese que le diré frame) número t-3.

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.

Pero me estoy adelantando, 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.

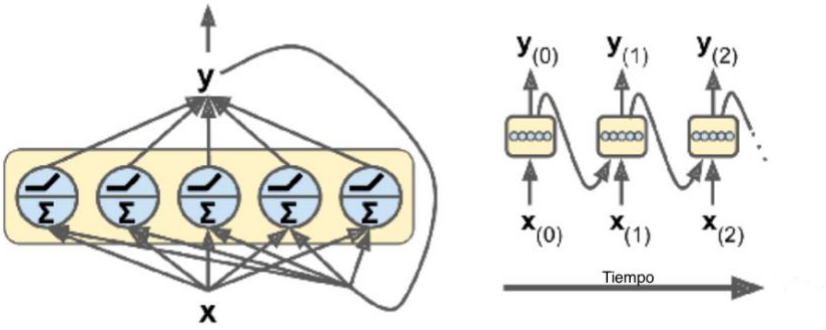
Ok, volviendo al frame t-2, la neurona nos va a escupir la señal. Pasamos ahora al frame t-2, donde tenemos nuestra misma neurona, y vuelve a recibir la exacta misma entrada X y aparte, el resultado del frame anterior t-3 … y por último repetimos el proceso en el momento t.

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.



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.



Voy a meterles a toda 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.

Todo normal hasta ahorita, y sigue el momento de la repetición. 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).

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 $x_y$otra 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 $W_y$pero con mayúsculas.

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.

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.

Si te pones a analizarlo, 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.

Entonces podríamos decir que nuestras neuronas tienen cierto tipo de memoria, porque almacenan algo de información que venía desde el principio (aunque mientras más lejano esté en el tiempo, más débil es la memoria).

Una parte de una red neuronal que preserva algún estado a lo largo del tiempo se le conoce como "Celda de Memoria" o Memory Cell en inglés.

Una neurona recurrente solita, o una capa de neuronas recurrentes, es una celda muy básica, capaz de aprender solo patrones cortos – generalmente solo 10 pasos en general. Nota que esto de los 10 pasos es una generalización, no una regla.

Lo siento chicos, pero vamos a tener que formalizar esto de neuronas y memoria con Álgebra.

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

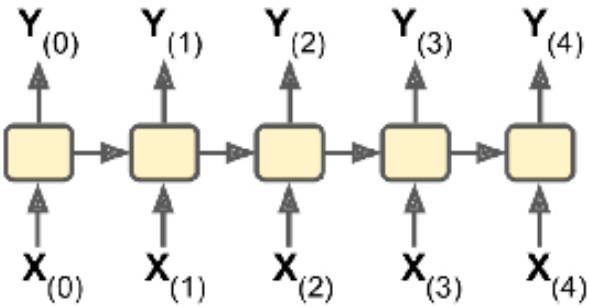
Su salida y_t, es una función del estado previo y sus entradas actuales. ¿Por qué todo este lío con el álgebra? Porque en las celdas básicas que vimos en la introducción, la salida y simplemente es igual al estado h, ambas son salidas de la neurona aplicando su función lineal de pesos y su función de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

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.

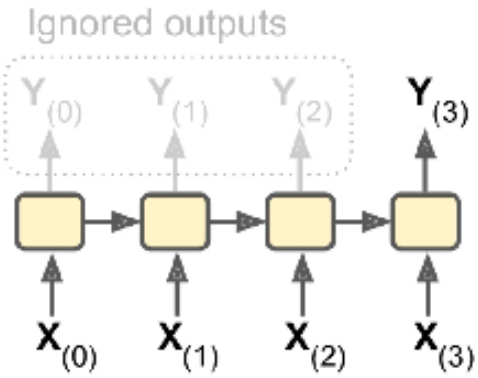
Para entrenar nuestra red neuronal le meteremos una secuencia de precios a lo largo de los últimos 5 días, y esperaríamos de regreso los precios, pero adelantados un día hacia el futuro, o sea desde hace 4 días hasta mañana.

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.

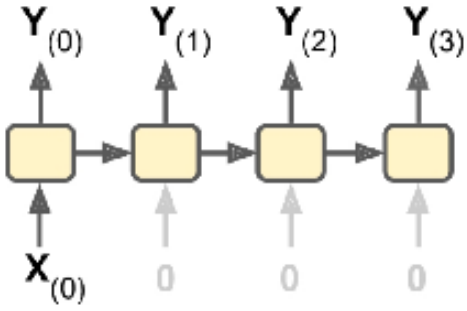


Por otro lado, tal vez queremos hacer un medidor de sentimiento – alimentarle, por ejemplo, las noticias del día y que nos diga si son noticias positivas (+1) o negativas (-1). En este caso usaríamos una arquitectura que se llama *secuencia a vector.*

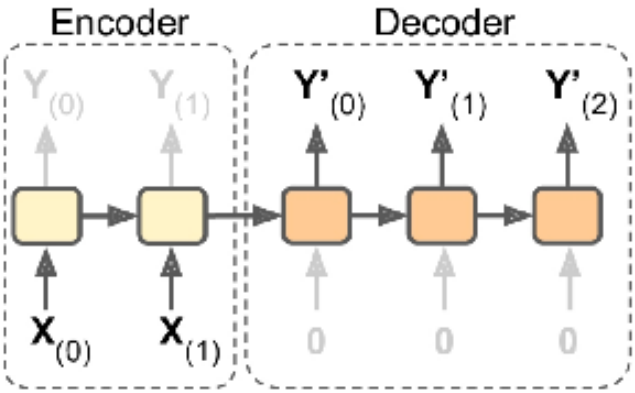
Lo que hacemos es alimentar la RNN con una secuencia de entradas e ignoramos todas las salidas excepto la última. Al final de cuentas solo necesitamos una conclusión – si la noticia es buena o mala.



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.



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.

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.

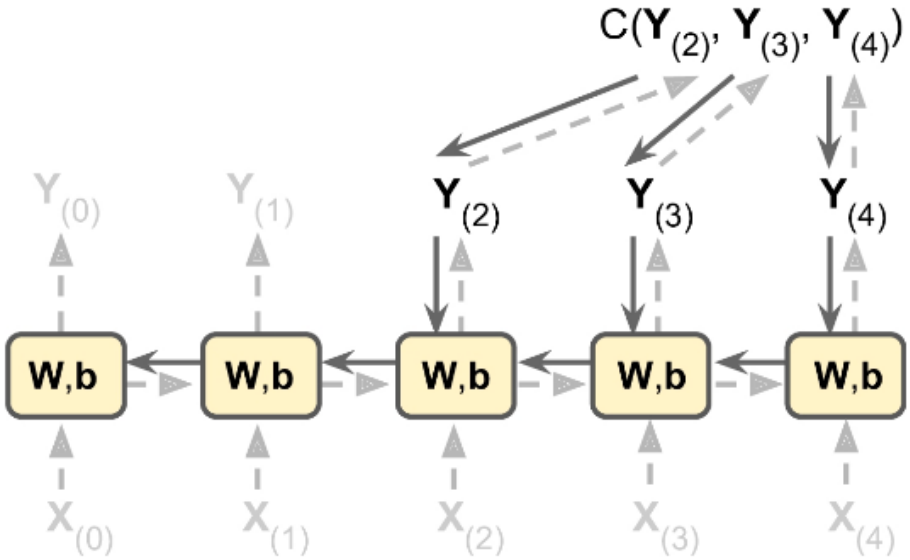
El esquema de Codificador-Decodificador es genial, pero es complejo, y por lo mismo, lo veremos en un capítulo posterior.

- ## 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.



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.*

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.

Ok, por suerte, igual que con toda la teoría, no tienes que preocuparte por esto, Keras lo hará por ti.

¡OK, hora de comenzar a echar código!

- ## Series de Tiempo

Vamos a comenzar con la aplicación más básica de los RNNs: series de tiempo. Puedes estar intentando darle seguimiento al precio de alguna acción en el mercado o tal vez quieres saber cuánta gente ha visto tu anuncio en Facebook cada día o qué tal seguir la producción exacta de kilos de margarina en azerbaiyán cada mes. Para todo esto vas a necesitar una serie de tiempo – una secuencia de uno o más valores por cada paso de tiempo, donde estos pasos pueden ser año, mes, día.

Cuando estás intentando seguir una sola variable, por ejemplo, la cantidad de gente que muere asfixiada por sus sábanas en USA por año (claro que sí sucede) entonces se llaman series univariantes porque solo es una sola variable. No se necesita ser Einstein para entender esto.

Ahora si quieres seguir diferentes variables a lo largo del mismo tiempo, por ejemplo, la cantidad de helados de chocolate vendidos en una nevería al día, junto con la cantidad de paletas, sándwiches de nieve, palitos de chocolate y helado para perros (también, claro que existe) vendidos por esa misma nevería, entonces tenemos una serie de tiempo multivariante.

Cuando estamos trabajando con series de tiempo generalmente estamos intentando realizar una de 2 tareas: predecir el futuro (miren, si supiera como hacerlo a ciencia cierta tendría mi propio avión y estaría pérdido en el alcohol en una isla, no hablando de redes neuronales), o intentar rellenar espacios en blanco.

Volviendo al ejemplo de la nevería, supongamos que la venta de la paleta de limón es lo que mantiene a toda la nevería a flote, 80% de nuestros ingresos vienen de las paletas de limón.

Supongamos que por un incendio en el mercado San Juan de Dios perdimos nuestros registros meticulosamente guardados sobre las ventas de paletas de limón en marzo y abril del año pasado. ¿Qué podemos hacer?

Podemos utilizar una RNN para estimar las ventas reales utilizando datos de otros años y otros meses del mismo año. Hacer esto de rellenar espacios en blanco en una serie de tiempo se llama Imputación.

Esta tarea es mucho más… acertada que intentar usar RNNs para predecir el futuro. Miren, yo sé que están pensando que, de aquí a la bolsa a predecir el movimiento del dólar, y las acciones y el bitcoin y boom, tiburón instantáneo.

Nel, como veremos en los siguientes ejemplos, las RNNs no son la panacea para predecir el futuro financiero. Difícilmente nos darán mejor desempeño que una regresión lineal.

¿Saben para qué si son una pistola las RNNs? Predecir los siguientes compases en una canción. Predecir las siguientes palabras en una oración. La clase de cosas que no te va a generar dinero instantáneo, pero puede resultar en aplicaciones increíbles que proveen un servicio a la sociedad y te generan dinero.

Vamos a comenzar con una serie de tiempo generada por nuestra función de generar series de tiempo:

```
def generate_time_series(batch_size,n_steps):
    freq1, freq2, offsets1,offsets2 = np.random.rand(4,batch_size,1)
    time = np.linspace(0,1,n_steps)
    series = 0.5 * np.sin((time - offsets1) * (freq1 *10 + 10))
    series += 0.2 * np.sin((time - offsets2)* (freq1*10+10))
    series += 0.1 * (np.random.rand(batch_size, n_steps)-0.5)
    return series[...,np.newaxis].astype(np.float32)
```

Esta función nos va a armar series de tiempo senoidales – tantas como nosotros queramos. Ahora primero nos va a dar una función senoidal equis, vainilla, y después va a echarle otra función senoidal encima… y la va a mover aleatoriamente para que sea más interesante.

Cuando trabajamos con series de tiempo, generalmente estamos tratando con arreglos de 3 dimensiones – tamaño del bache (o sea cantidad de series), la cantidad de pasos de tiempo y la dimensionalidad (sea 1 si es una serie univariante o más para una serie multivariante).

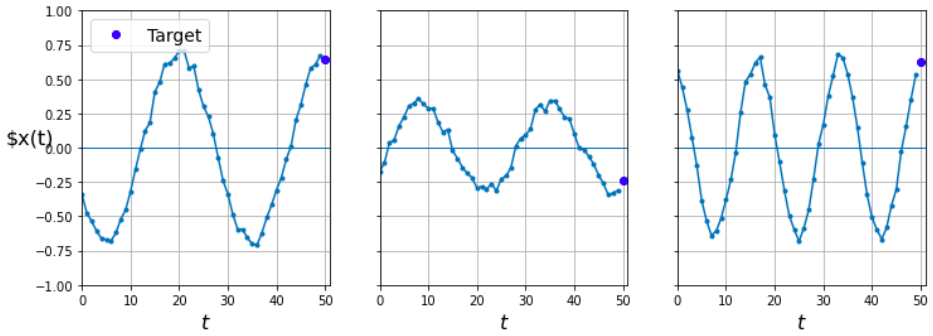
Ahora crearemos un set de entrenamiento (7000 datos), de validación (2000 datos) y de prueba (1000 datos) usando nuestra serie de tiempo.

```
n_steps = 50
series = generate_time_series(10000,n_steps+1)


X_train, y_train = series[:7000,:n_steps], series[:7000,-1]
X_valid,y_valid = series[7000:9000, :n_steps], series[7000:9000,-1]
X_test,y_test = series[9000:,:n_steps],series[9000:,-1]
```

X_train contiene 7000 series temporales, mientras **que** *X_Valid* contiene 2000 y *X_test* contiene 1000. Dado que queremos pronosticar un solo valor para cada serie, los objetivos son vectores de columna.

Por último, estaremos analizando las series de tiempo con las gráficas que se muestran:



Que necesitan el siguiente código:

```
def plot_series(series, y=None, y_pred=None, x_label="$t$", y_label="$x(t)$",
legend=True):
  plt.plot(series, ".-")
  if y is not None:
    plt.plot(n_steps,y,"bo", label = "Target")
  if y_pred is not None:
    plt.plot(n_steps, y_pred, "rx", markersize=10, label ="Prediction")
  plt.grid(True)
```

```
    if x_label:
        plt.xlabel(x_label,fontsize=16)
    if y_label:
        plt.ylabel(y_label,fontsize=16, rotation=0)
    plt.hlines(0,0,100,linewidth=1)
    plt.axis([0, n_steps + 1, -1,1])
    if legend and (y or y_pred):
        plt.legend(fontsize=14, loc = "upper left")


fig, axes = plt.subplots(nrows = 1, ncols=3, sharey=True, figsize = (12,4))
for col in range(3):
    plt.sca(axes[col])
    plot_series(X_valid[col,:,0],y_valid[col,0],y_label=("$x(t)" if col == 0 else None),
legend=(col == 0))
```

- **Métricas Base**

Miren, lo que va a pasar ahorita es que vamos a correr el RNN, nos va a dar un error de como 3%, y vamos a decir "éxito total", palmadas en la espalda para todos los participantes, foto conmemorativa del evento y ponemos en nuestro currículo que somos expertos en Redes Neuronales Recurrentes. La verdad es que no. Una regresión lineal probablemente nos gane.

Siempre que vamos a hacer este tipo de ejercicios de predicción, o de imputar, es necesario tener una medida base contra la cual nos estemos peleando. De otra manera vamos a salir felices de que nuestro modelo es genial cuando en realidad está fracasando frente a métricas más sencillas.

En este caso vamos a predecir el último valor de nuestras series – esto se llama predicción ingenua y generalmente es muy difícil de ganarle.

Otra manera de hacer nuestra predicción base es usando una red completamente conectada. Esto nos da un MSE de 0.004. Va a ser difícil ganarle a nuestra red neuronal sencilla.

- **Ejercicio de Cuaderno de trabajo:**

```
#Vamos a hacer predicciones ingenuas
y_pred = X_valid[:,-1]
Vamos a ver el error de eso
np.mean(keras.losses.mean_squared_error(y_valid,y_pred))

#Muestra la gráfica con tu predicción ingenua
plot_series(X_valid[0,:,0],y_valid[0,0],y_pred[0,0])
```

```
plt.show()

#Vamos ahora a hacer predicciones lineales. Arma tu modelo secuencial con una capa de flatten y dense:
model = keras.models.Sequential([
  keras.layers.Flatten(input_shape=[50,1]),
  keras.layers.Dense(1)
])

#Compila con pérdida MSE y optimizer ADAM
model.compile(loss="mse", optimizer ="adam")
#Entrena con 20 épocas
history = model.fit(X_train,y_train,epochs=20, validation_data = (X_valid,y_valid))

#Evalúa tu modelo sobre valid
model.evaluate(X_valid,y_valid)

#Vamos a armar una gráfica (definir una función) para ver las curvas de aprendizaje
def plot_learning_curves(loss, val_loss):
  plt.plot(np.arange(len(loss)) + 0.5, loss, "b.-", label="Training loss")
  plt.plot(np.arange(len(val_loss)) + 1, val_loss, "r.-", label="Validation loss")
  plt.gca().xaxis.set_major_locator(mpl.ticker.MaxNLocator(integer=True))
  plt.axis([1, 20, 0, 0.05])
  plt.legend(fontsize=14)
  plt.xlabel("Epochs")
  plt.ylabel("Loss")
  plt.grid(True)

#Grafica tus curvas de aprendizaje
plot_learning_curves(history.history["loss"], history.history["val_loss"])
plt.show()

# Haz una predicción sobre X_valid (con model. predict) y gráficala con tu función de plot series
y_pred = model.predict(X_valid)
plot_series(X_valid[0,:,0],y_valid[0,0],y_pred[0,0])
plt.show()
```

- **Implementar un RNN Simple**

Este es la RNN más simple que podemos construir:

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

Es de una sola capa, con una sola neurona, como el que vimos en el ejemplo de al principio.

En un RNN no necesitamos especificar el largo de la secuencia de entrada en la figura, ya que una red neuronal recurrente puede procesar cualquier número de pasos. Por default, la capa de SimpleRNN va a usar la función de activación *Tanh*. Esto por lo que vimos anteriormente – ReLU no es tan útil dentro de RNN.

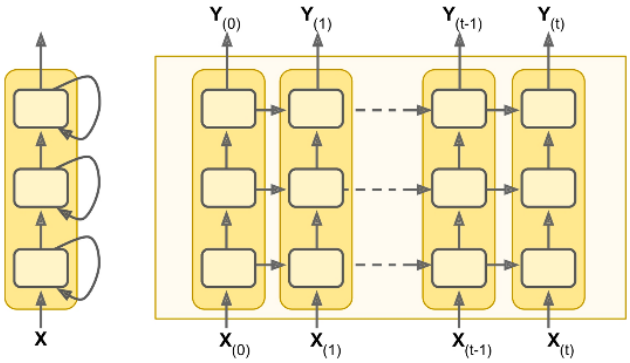
En términos de desempeño, un modelo lineal tiene un parámetro por entrada y por paso de tiempo, más un término de bias (o sea un total de 51 parámetros en nuestro ejemplo). Por otro lado, una neurona recurrente en un RNN simple solo hay un parámetro por entrada y por dimensión oculta, más un término de entrada. En este ejemplo, esos son solo 3 parámetros. Ok, así que un modelo simple tal vez no es la mejor opción – tuvo problemas para ganarle a nuestros modelos básicos. Vamos agregando más capas recurrentes y metiéndonos en territorio de RNNs profundos.

- **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.

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.



Implementar un RNN con Keras es pan comido- solo échale capas recurrentes una encima de otra.

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.

Si lo dejamos en falso, la capa nos va a dejar caer un arreglo 2D que solo contiene la salida del último paso de tiempo. Esto solo es útil en la última capa de una red recurrente – cuando al final solo queremos la salida mega procesada.

Como estamos prediciendo una serie de tiempo de una sola variable, nuestra última capa es de 1 sola neurona. O sea, por cada paso de tiempo tenemos que llevar solo un valor de salida.

Pero como tenemos una sola unidad, eso significa que el estado oculto es solo un número. Eso no es muy útil para nosotros: probablemente el RNN va a usar los estados ocultos de las otras capas para traer toda la información que necesita de paso a paso, y no usará el estado oculto de la última capa para casi nada.

También, una vez que una capa de SimpleRNN usa la función de activación *Tanh*, te va a obligarte a tragarte un valor entre -1 o 1, te guste o no. ¿Qué pasa si no quieres que tu resultado acabe en este rango o usar una función de activación alternativa?

Por eso para la última capa de una red RNN es mejor usar una capa Densa – es poquito más rápido, casi igual de exacta, y te deja usar la función de activación que tu corazón te pida.

Nota que, si vamos a hacer este cambio, el *return_sequences = True* de la penúltima capa de RNN desaparece, por que ahora esta es nuestra nueva última capa de RNN, y pues, ya no necesitamos sacar nuestros datos por ahí.

- ## Predecir Varios Pasos en el Futuro

OK, hasta ahorita solo hemos hecho una sola predicción, un solo paso en el futuro. Si estuviéramos prediciendo acciones día por día, este modelo nos diría que va a suceder mañana con el precio de la acción ¿Qué pasa si quisiéramos saber más a futuro que va a pasar? Por ejemplo, predecir cómo va a amanecer el precio de la acción en cada uno de los próximos 7 días a futuro.

La primera opción que tenemos es usar el modelo que ya tenemos entrenado. Es muy fácil. Lo obligamos a predecir el siguiente valor (el precio de mañana, por ejemplo) y luego sumamos ese valor a la lista de valores reales de nuestros precios de acciones.

Luego usamos esa nueva lista actualizada para predecir el siguiente valor (el día después de mañana) y lo volvemos a meter a nuestra lista, usaremos para predecir el día después de después de mañana, que meteremos a nuestra lista… y así sucesivamente.

Chequense como lo haremos en código:

```
series = generate_time_series(1,n_steps+10)
X_new, Y_new = series[:,:n_steps], series[:,n_steps:]
X = X_new
for step_ahead in range(10):
  y_pred_one = model.predict(X[:,step_ahead:])[:,np.newaxis,:]
  X=np.concatenate([X,y_pred_one], axis = 1)

Y_pred = X[:, n_steps:]
```

Ahora, si saben cualquier cosa de predicciones, sabrán que la adivinanza para el precio de mañana será mucho mejor, que la de pasado mañana, que a su vez es muy probable que sea mejor que la de la próxima semana. Ok, pero lo importante… ¿Este método sirve? no, en realidad no.

Una comparación rápida contra un modelo lineal sencillo que cualquier alumno de prepa puede hacer nos dice que nuestro RNN todo Fancy y sofisticado es una reverenda basura.

```
n_steps=50
series = generate_time_series(10000,n_steps+10)
```

¿Así que la gente que se dedica a redes neuronales son unos charlatanes? Si, pero ese es tema de otro curso. Broma, no hay otro curso.

Hay una segunda opción que tal vez rescate el nombre de las RNNs – usar la red para predecir los siguientes 10 valores todos al mismo tiempo. Usaremos un modelo de secuencia a vector, pero nos va a escupir 10 valores de jalón en vez de uno – dicho eso, tenemos que cambiar los targets para que sean vectores que contienen los siguientes 10 valores. Y ahora sí, sólo necesitamos la capa de salida que tenga 10 unidades en vez de 1.

```
np.random.seed(42)
tf.random.set_seed(42)

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

model.compile(loss="mse", optimizer="adam")
history = model.fit(X_train, Y_train, epochs=20,
                    validation_data=(X_valid, Y_valid))
```

Ok, hacemos nuestras predicciones.

Y resulta que ahora sí tenemos un mejor modelo que el modelo lineal chafa que estamos usando de base. ¿Así que ya empacamos y nos vamos no? ¡Ya tenemos el santo grial de los predictores y podemos ir comprando nuestra casa de vacaciones en Bali! no, hay otras opciones.

En este caso reaccionamos a nuestro modelo a que nos entregue a nosotros, sus amos, los siguientes 10 valores solo al final de la serie. Lo cual tiene sentido, solo queremos la predicción de los próximos 10 días en el futuro.

Pero lo que les voy a proponer, y tengan la mente abierta, es que vamos a pedirle al modelo que nos prediga los siguientes 10 valores en cada uno de los pasos de tiempo.

O sea, podemos convertir esta Red RNN de tipo Secuencia a vector, a una RNN de tipo secuencia a secuencia.

Esto significa que en el paso 0, vamos a sacar un vector con las predicciones para los pasos de tiempo 1 a 10, luego en el paso 1 las predicciones para los pasos 2 a 11, y así consecutivamente.

La ventaja de este método es que el RNN va a estar siendo evaluado a cada paso del tiempo y la pérdida va a contener un término para cada salida del RNN en cada momento. Como le subimos la cantidad de gradientes de error, el modelo va a ser más estable y rápido.

Para su implementación, vamos a tener que armar una secuencia de este largo que la secuencia que contiene un vector de 10 dimensiones en cada paso:

```
np.random.seed(42)

n_steps = 50
series = generate_time_series(10000, n_steps + 10)
X_train = series[:7000, :n_steps]
X_valid = series[7000:9000, :n_steps]
X_test = series[9000:, :n_steps]
Y = np.empty((10000, n_steps, 10))

for step_ahead in range(1, 10 + 1):
  Y[..., step_ahead - 1] = series[..., step_ahead:step_ahead + n_steps, 0]

Y_train = Y[:7000]
Y_valid = Y[7000:9000]
Y_test = Y[9000:]
```

Para que el modelo de verdad sea de secuencia a secuencia, vamos a prenderle el *TRUE* a todas las capas de RNN en su propiedad de *return-sequences*, y vamos a usar una cosa que se llama *TImeDistributed Layer* para la capa del final.

No es una capa como tal, es una envoltura para una capa (en nuestro caso una capa densa) que hace que aplique a cada paso de su secuencia de entrada.

En este ejemplo, la capa densa va a tener 20 unidades de entrada porque la RNN anterior tenía 20 neuronas. Luego va a ajustar la salida a 10 dimensiones porque, la capa densa 10 unidades.

Todas las salidas se necesitan durante el entrenamiento. Pero solo la salida en el último paso se usa para predicciones y para evaluación. Así que vamos a confiar en el MSE para todas las salidas para el entrenamiento, vamos a usar una medida personalizada para la evaluación, para solo computar el MSE sobre la salida del último paso:

El MSE nos va a dar mejor que el modelo anterior.

Los RNNs simples como los que acabamos de hacer son buenísimos cuando se trata de predecir series de tiempo o manejar otra clase de secuencias, pero no les va bien en series de tiempo largas o secuencias. Vamos viendo alternativas.

• Manejar Secuencias Largas

Ok, les dije en el video anterior que los RNNs no son buenos con secuencias largas, pero lo dejé en nivel “wey, créeme”.

Supongamos que tenemos los últimos 50 días de precios de TESLA y queremos predecir los próximos 5. Esto significa que nuestro RNN se va a desenrollar a lo largo del tiempo en un equivalente de 50 "capas" si así lo quieren ver. Perfectamente manejable.

Pero si quieres usar los datos de TSLA desde que comenzaron a cotizar en bolsa, marzo 11 del 2011, entonces vas a tener 2,887 días de información hasta el día de hoy (4 de abril del 2022, y recuerda que la bolsa solo cotiza en días hábiles, no es un OXXO para andar trabajando 24/7).

¿Una RNN que desenrollada tiene 2887 capas? Ajá, buena suerte no enfrentándose al problema del gradiente que se desvanece.

Lo que, es más, no solo tienes los problemas clásicos que vienen con una red neuronal profunda – un RNN trae problemas propios, únicos e inigualables. Lo principal entre ellos es que se le va a olvidar lo que pasó al principio de la secuencia.

O sea, le vas a dar 2887 días de cotizaciones de TESLA comenzando desde el 2011 hasta el 2022, pero siendo realistas, lo que pasó en el 2020 para acá va a cargar 99% del peso al momento de entrenar.

¿Técnicamente esto es bueno, pero entonces para que se pone a calcular todo lo demás? Si le diste todos los días es para que use todos los días. Si no te queda claro por que la memoria de Dory es un problema en los RNNs, ahorita en unos minutos vamos a ver un ejemplo concreto donde nos puede llevar a problemas.

- ## Gradientes Inestables

Las buenas noticias es que varios trucos que vimos en el capítulo 2 para aliviar el problema de los gradientes inestables funcionan con las *RNNs*: inicialización apropiada de parámetros, optimizadoras más rápidas, dropout, entre otros. Sin embargo, nuestra solución ideal fue *ReLU*. Malas noticias, no funciona en RNN y aparte es activamente dañina para la red.

El problema viene con el funcionamiento del descenso del gradiente. Si se actualizan los pesos de tal manera que aumentan ligeramente en el primer paso de tiempo, como se usan los mismos pesos en el siguiente paso de tiempo, entonces los valores pueden incrementar ligeramente, y luego incrementar más en el siguiente paso y luego más, y así consecutivamente, hasta que las salidas exploten en tamaño.

Las funciones de activación que no saturan (como RELU y sus familiares) no pueden prevenir eso. Por eso usamos *TanH* como el default- nos limita a solo obtener valores entre -1 y 1.

También, los gradientes pueden llegar a volverse masivos si usamos funciones de activación que no saturan, así que, si tienes problemas con gradientes inestables, no los vas a resolver con RELU en un RNN. Otra herramienta de nuestro kit no nos va a servir para mucho es Batch Normalization.

¿Por qué? En una red profunda de tipo feedforward (alimentación hacia adelante, o la típica red normal y vainilla que usamos anteriormente) metemos capas de Batch Normalization entre cada una de las capas Densas o Convolucionales que estábamos trabajando.

Técnicamente aquí también puedes meter capas de Batch Normalization entre cada capa de *SimpleRNN.* pero si haces eso no estás entendiendo de dónde viene el problema.

Recuerda que cada capa de SImpleRNN es básicamente su propia red por que se desenrolla a lo largo del tiempo para cada uno de los pasos que tengas en tus entradas, así que, no vas a poder meter Batch Normalization entre cada una de esas “capas” que representan el tiempo.

Ok, mentira, técnicamente si se puede, pero es inútil porque la capa de Batch Normalization que logres meter ahí va a tener los mismos pesos y parámetros entre cada uno de los pasos de tiempo y pues en el mejor de los casos, esto no va a empeorar tu modelo, si tienes suerte.

La alternativa que sí funciona es una que se llama Layer Normalization, introducida por Jimmy Lei Ba en el 2016 – muy parecida a Batch Normalization. Pero Layer Normalization normaliza a través de la dimensión de las variables. En un RNN, la capa de Layer Normalization se usa justo después de la combinación lineal de las entradas y las capas ocultas.

Vamos a usar *tf.keras* para implementar Normalización de Variables con una simple celda de memoria.

Para esto necesitamos definir una celda de memoria personalizada. Es como una capa normal, excepto que su método de *Call* agarra el argumento de las entradas y los estados ocultos del paso previo.

El argumento de estados es una lista que contiene uno o más tensores. El caso de una celda simple de RNN que contiene un solo tensor igual a las salidas de el time step previo, pero otras celdas pueden tener varios tensores de estado múltiple.

Una celda también debe tener un atributo de *state_size* y un atributo de *output_size.* En un RNN simple, ambos son simplemente iguales al número de unidades.

El siguiente código va a implementar una celda de memoria personalizada que se porta como una capa de SimpleRNN, pero aplica Layer Normalization a cada paso del tiempo.

```
#Crea una clase que se llama LNSimpleRNNCELL
class LNSimpleRNNCell(keras.layers.Layer):
    def __init__(self,units, activation = "tanh", **kwargs):
        super().__init__(**kwargs)
        self.state_size=units
        self.output_size = units
        self.simple_rnn_cell=keras.layers.SimpleRNNCell(units,
activation=None)
        self.layer_norm=LayerNormalization()
        self.activation = keras.activations.get(activation)
    def get_initial_state(self, inputs=None, batch_size = None,
dtype=None):
        if inputs is not None:
            batch_size = tf.shape(inputs)[0]
            dtyape = inputs.dtype
        return [tf.zeros([batch_size, self.state_size], dtype=dtype)]
    def call(self, inputs, states):
        outputs, new_states = self.simple_rnn_cell(inputs, states)
        norm_outputs = self.activation(self.layer_norm(outputs))
        return norm_outputs, [norm_outputs]
```

El código es bastante sencillo. Nuestra celda hereda de *keras.layers.Layer*. El constructor toma el número de unidades y la función de activación deseada, y pone los atributos de state_size y output_size. Luego crea una *SimpleRNNCell s*in función de activación. *SimpleRNNCell* es diferente a *simpleRNN p*or que procesa 1 solo paso de toda la secuencia, no la secuencia entera como SimpleRNN.

Luego creamos la capa de Layer Normalization, y finalmente agarramos la función de activación.

La magia sucede con en *call*, donde aplicamos nuestra celda simple de RNN, que computa la combinación lineal de las entradas y los estados ocultos, y luego devuelve el resultado 2 veces. Luego se aplica la normalización de capa, seguido por la función de activación. Al final regresa la salida 2 veces, una como las salidas formales, otra como los nuevos estados ocultos. Para usar esta celda personalizada, todo lo que necesitamos es crear una capa de *keras.layers.RNN*, y le pasamos una instancia de esta nueva celda.

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

Con esta técnica puedes alivianar el problema de los gradientes inestables y entrenar un RNN mucho más eficiente. Ahora veamos cómo lidiar con el problema de memoria a corto plazo.

- ## 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. ¿Se acuerdan de Dory de Nemo? Es el mismo problema.

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.

¿Si pones a Dory a traducir una oración medio larguita que va a pasar? Para cuando termine de leer la oración va a haber olvidado las primeras palabras.

Lo mismo sucede con los *RNNs* que hemos visto hasta ahora.

Para lidiar con este problema, se han inventado varias celdas bastante complicadas de RNNs que si tienen capacidad de memoria a largo plazo. Así que no son como Dory. La más exitosa, y popular, de estas se llama 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))
])
```

O también puedes usar una capa de RNN así sencilla, pero de argumento le pasas una celda de *LSTM* y también funciona igual. Dicho eso, la razón por la cual usamos la capa dedicada de *LSTM* (en vez de la celda) es que la capa está optimizada para correr en tarjetas de video, así que puede resultar ser más rápida. Bueno, ya vimos cómo echarla a andar en *Keras*, pero pues no sabemos cómo funciona.

Vamos viendo un ejemplo muy sencillo:

> Imaginemos que vamos a escribir una canción de reggaetón. Como ustedes sabrán, estas canciones se distinguen por su amplitud intelectual y la profundidad de sus letras. Así que vamos a quebrarnos la cabeza armando un repertorio de posibles frases que podemos incluir en nuestra canción:
>
> - *Ella perrea sola.*
> - *Sola perrea la Bichota.*
> - *La Bichota perrea con ella.*

Nuestro diccionario de palabras es pequeño: Ella, perrea, sola, Bichota, con, la.

Vamos a intentar armar una red neuronal para organizar estas frases en el orden correcto – nota que el orden correcto es el que les acabo de dar. Si la última letra fue Ella perrea sola, sigue “Sola perrea la bichota”.

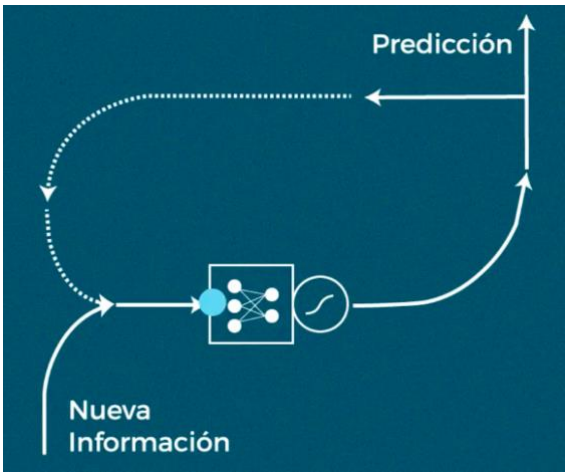
Ahora al final de cuentas esto se va a alimentar como un vector a nuestro diccionario. Si la última palabra que vi fue Bichota, entonces se va a alimentar un vector de esta manera, donde todas las demás palabras son ceros y bichota es 1.

Si metemos los datos a entrenar vamos a esperar ciertos patrones – siempre que nos referimos a una señorita en la canción, ella, Bichota o sola, entonces las únicas 2 palabras que pueden seguir es “Perrea” o un “.” Que significa cortón y frase nueva.

Si nuestra red neuronal ve Perrea o “.” Entonces sigue una referencia a una dama: Ella, Bichota, Sola, La.

Vamos imaginando que metemos esto a una Red Neuronal Recurrente normal.

Así se vería nuestro diagrama:



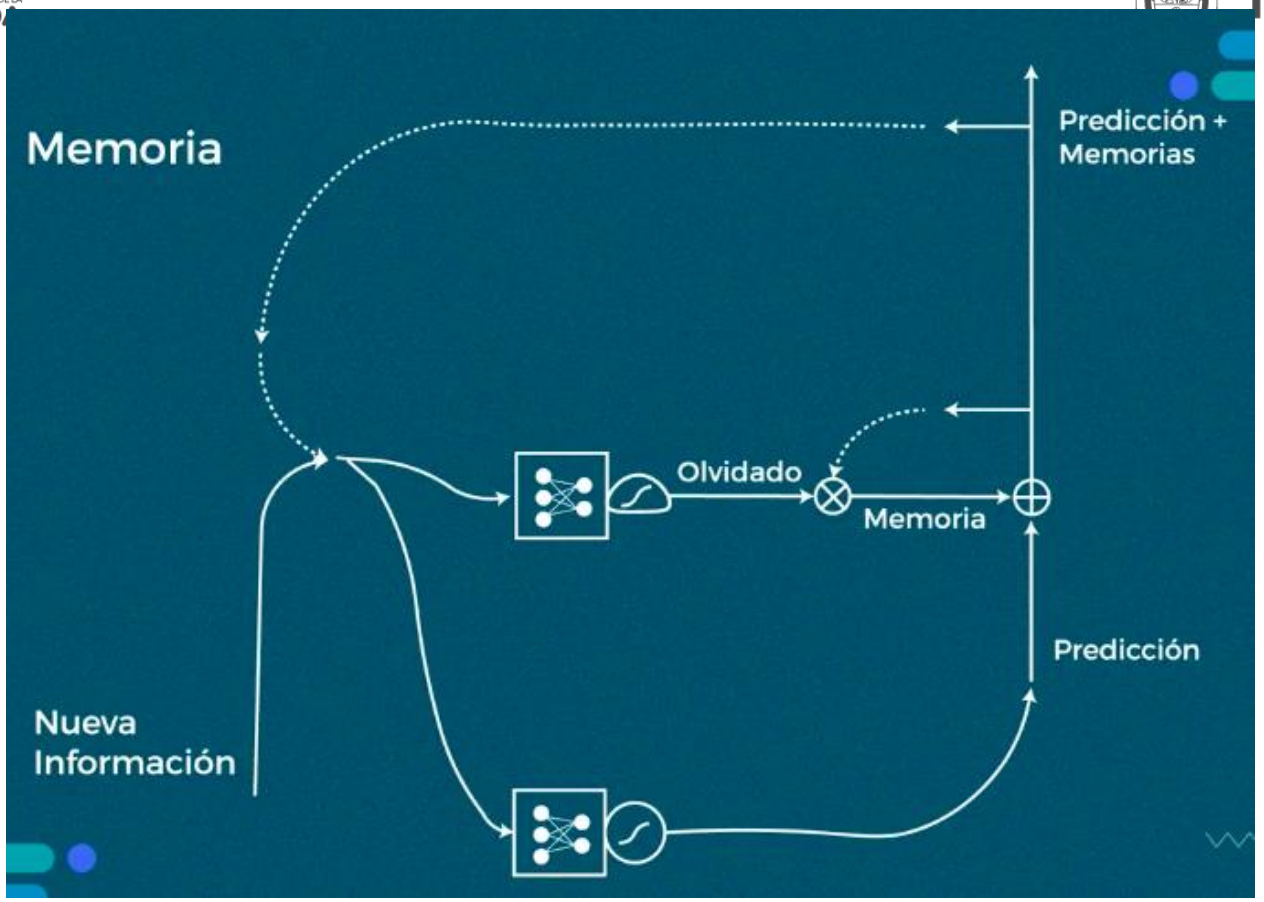
Entra la información por aquí, sube a nuestra función de aplicación (la combinación lineal de entradas y pesos) y luego pasa por nuestra función de activación *TanH* – posteriormente sale la predicción como salida, como la nueva entrada (o estado oculto) que se vuelve a alimentar a la red neuronal porque es una red neuronal RECURRENTE.

Ahora la Red Neuronal Recurrente simple va a generar errores, si o sí. Podríamos obtener una oración como: *La Bichota perrea La Bichota*. Desafortunadamente, nuestras reglas lo permiten ya que después de un perrea viene una referencia a alguien.

O también podríamos tener una frase interminable que diga: *Ella perrea sola perrea la Bichota perrea con ella perrea sola*. Por último, podríamos obtener una frase que diga *Bichota. Ella. Sola.*

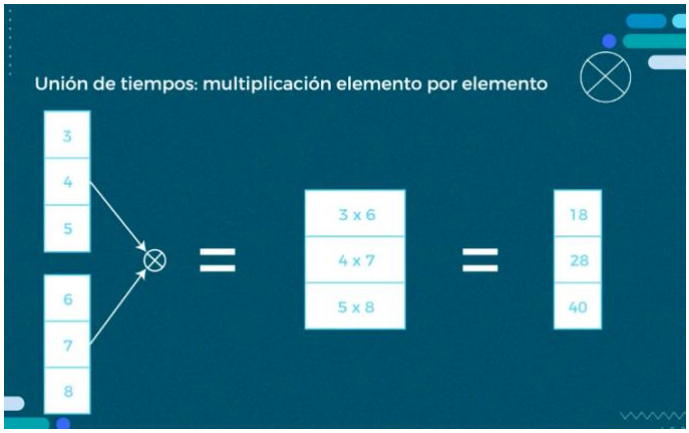
Esto es por que cada una de nuestras predicciones solo observa un paso atrás en el tiempo, no tiene memoria sobre lo que paso antes, así que sin saber cual era el contenido completo de las frases anteriores, va a estar aplicando las reglas de una manera que no se espera.

Para resolver esto, vamos a expandir nuestra red neuronal recurrente. Vamos a tener que agregarle una parte intermedia que le llamaremos “memoria”.



Déjenme les presento estos símbolos nuevos, que la verdad son muy universales en los diagramas de LSTM – primero, nuestra función de activación TanH si se fijan ahora trae un fondo plano en su círculo.

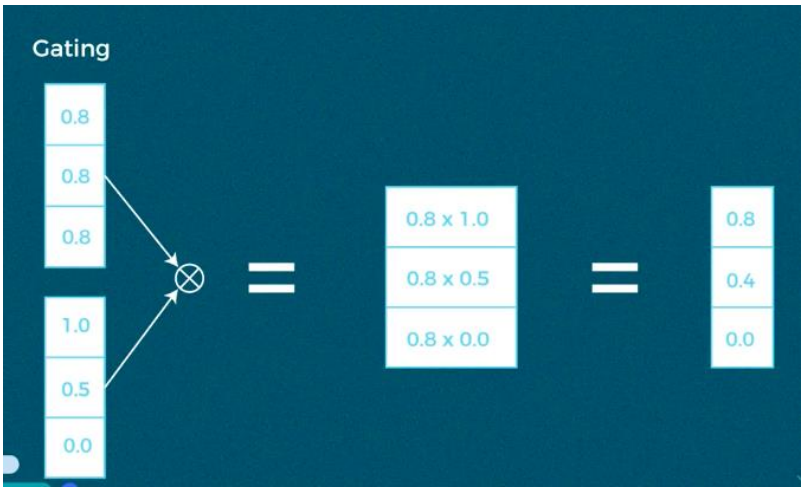
Luego, este círculo de aquí trae una X como de cruce de ferrocarril, significa multiplicación de los elementos. Y este círculo que tiene un símbolo de cruz también significa suma.



Por ejemplo, si empezamos con 2 vectores iguales, simplemente vamos sumando los datos de cada vector, elemento por elemento, así que el vector de salida es del mismo tamaño que los vectores d entrada.

El símbolo de multiplicación es lo mismo – multiplicación elemento por elemento. Igual, traemos 2 vectores, multiplicamos cada uno de los elementos y nuestro resultado es un vector del mismo tamaño que las entradas con los productos de los vectores de entrada.

Esto de la multiplicación nos va a servir para activar el proceso de memoria ¿qué se olvida y qué se recuerda? Podríamos decir que tenemos una señal – un vector con datos de verdad. Queremos que solo pasen ciertos datos de esta señal, para eso vamos a agarrar otro vector que va a decidir que entra y que se va.



Si queremos que el primer elemento pase, entonces vamos a usar un uno, si queremos que el segundo elemento pase a medias, usamos un .5 y si no queremos que pase en lo absoluto, usamos un 0.Entonces al momento de hacer nuestras multiplicaciones elemento por elemento, nuestro nuevo vector es el resultado de haber decidido que pasar y que bloquear.

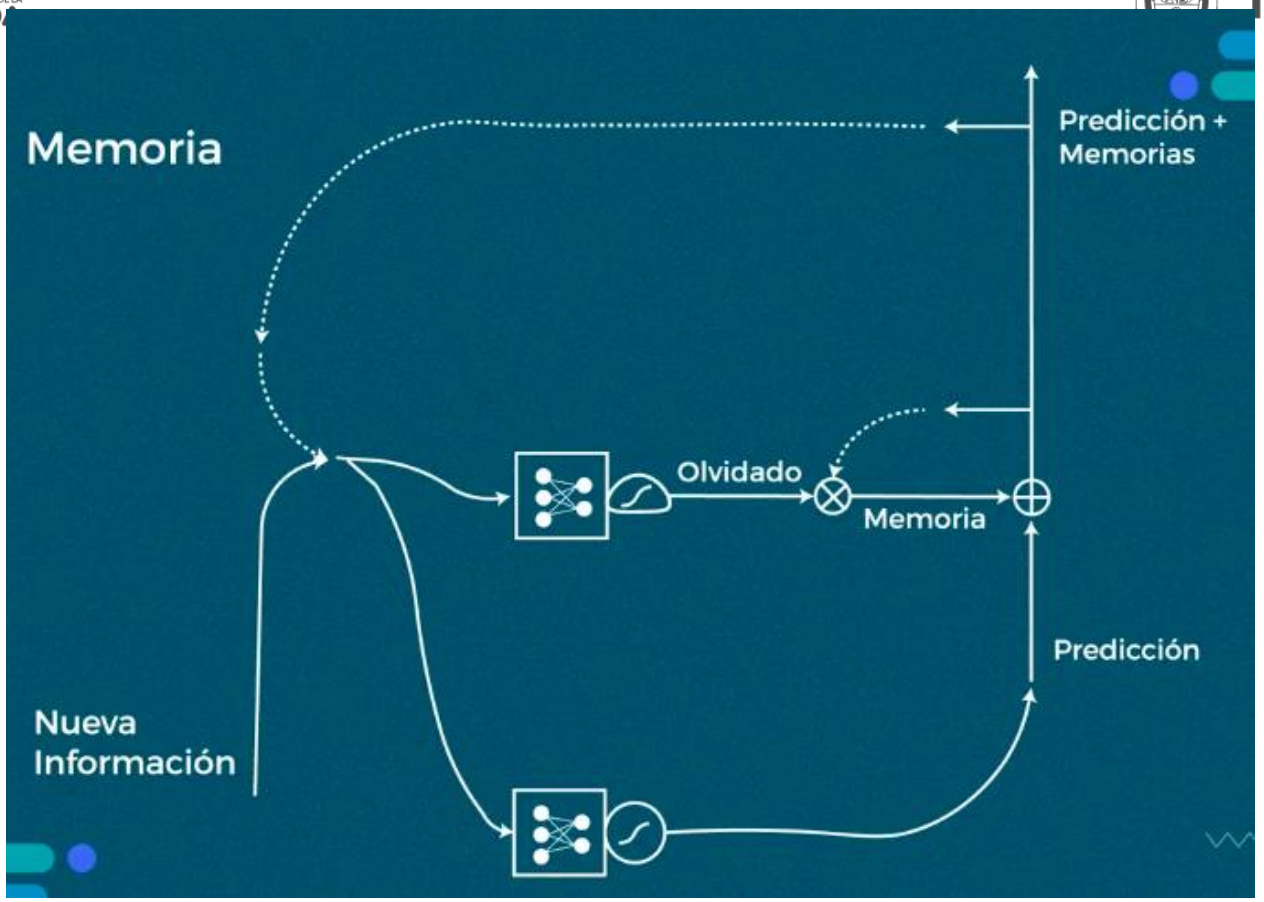
Este proceso se llama “Gating” en inglés – que es como cercar en español. No hay una traducción directa. Simplemente es como tener una puerta en un antro y estamos dejando pasar a quien queremos y bateando a los que no.

Para terminarlo vamos a rematar con una función de activación logística – la queremos usar por que nos va a limitar nuestros valores a que caigan entre 0 y 1. Y la función logística se representa con este simbolito que vimos anteriormente.

OK, entonces va a entrar nuestra nueva información, que todavía entra, se le aplica el mismo proceso que vimos para nuestra Red Recurrente, y el resultado que se escupe se usa como salida y como entrada para el siguiente ciclo.

Pero ahora que tenemos los datos que van a entrar al siguiente ciclo, van a pasar por nuestra puerta del olvido. Algunos datos van a olvidarse a través de esta cerca, y los que se recuerden van a sumarse a nuestra predicción original. Y muchachos, aquí hay una red neuronal completamente diferente que decide que va a olvidar y que va a recordar.

Cuando estamos combinando nuestras predicciones con nuestras memorias, no siempre queremos soltar las memorias en el mismo momento que nuestras predicciones – tal vez tenemos que guardar nuestras memorias para el siguiente paso, o los siguientes 2 pasos, o los siguientes 10.



En ese caso queremos guardar nuestras memorias usando un tercer paso de selección. Va a tener su propia red neuronal, que va a decidir qué memorias se mantienen internas, y cuales se liberan.

Va a tener su propia función logística de activación, y su propia cerca o puerta para decidir qué se mata y que se publica. Y por último tenemos una función de activación TanH nueva para asegurarnos que la suma de nuestra predicción con lo que sacamos en la puerta del olvido se queda entre -1 y 1.

Ahora, que traigamos nuevas predicciones, vamos a coleccionar varias de ellas mediante nuestra celda de memoria, sumarlas predicciones originales con las memorias relevantes, y luego pasarlas por la puerta de selección, que va a escoger solo algunas cuantas para liberar como la predicción del momento.

OK, por último, vamos agregando otra celda – una celda de “ignorar” que nos permite ignorar y hacerle el fuchi guacala a varias de las predicciones que está sacando nuestra red neuronal original.

Tiene su propia red neuronal, su propia función de activación logística, y su propia puerta aquí en este lugar. Esto es básicamente un esquema de LSTM, con sus 4 componentes y sus 4 redes neuronales internas.

Vamos retomando el ejemplo de nuestra canción de reggaetón. Supongamos que ya le pasamos a nuestra red LSTM montón de ejemplos de letras de reggaetón que queremos imitar. Así que ya decidió que ignorar, que olvidar y recordar, y que seleccionar.

Así que la última frase fue “Ella Perrea Sola.”, y la siguiente palabra fue “Sola”. Así que nuestra red neuronal sencilla va a haber predicho Sola, Ella y La Bichota por que la última parte fue un punto, pero escogió “Sola”.

Asi que tenemos nuestra nueva información, que es “Sola”, tenemos las 4 posibilidades (que eran Sola Ella, Bichota, La) y le pasamos esos 2 vectores a nuestro LSTM.

La red neuronal básica va a hacer predicciones. Sabe que habiendo visto “Sola” va a decir que lo que sigue es “perrea”, pero como acaba de ver “Sola” también va a mandar una señal diciendo “Esto no es”.

Las celdas de ignorar y olvidar van a decir “sí que pase así el vector” y nuestra celda de selección lo va a detener para la inspección final.

Las 2 posibilidades después un nombre son “.” y “perrea”. Como la predicción ya va en “perrea”, el vector de selección lo deja pasar, pero si recuerdan también venía un voto negativo diciendo “Sola no es”. La parte de Selección va a decir, gracias por la información y va a bloquear este segundo vector, de esta manera, lo último que salió fue “perrea”. Y listo, ya tenemos una primera iteración de LSTM.

Así que pasamos al siguiente paso. En este momento la palabra “perrea” acaba de ocurrir, así que ahora la predicción es que va a venir sola, Ella, Bichota, La. Las 4 posibilidades pasan por la celda de ignorar sin problema, pero llegando a la celda de olvidar/recordar, aquí tenemos un problema.

Habíamos dicho en la vez pasada que le habían llegado las palabras perrea y “No es sola”. La celda de memoria va a decir 😕 “Acabo de ver perrea, así que la voy a descartar, y acabo de ver un mensaje que dice que la palabra “SOLA” no es, así que también tiene que ser descartada”.

Así que tenemos un voto positivo para Sola, porque la regla lo permite, tenemos un voto negativo para Sola, porque la memoria lo impide, así que se cancelan. Ahora solo llegan predicciones para Ella y Bichota.

Ambas son gramaticalmente correctas. Básicamente LSTM puede ver para atrás varios pasos y recordar, con base a predicciones anteriores, cuál es el resultado que sigue.

- ## Celdas GRU

Una variante de las celdas *LSTM* es la celda *GRU* – como el protagonista de mi villano favorito (pero tristemente, sin relación alguna).

Se llama *Gated Recurrent Unit* en inglés y sus siglas le dan su nombre. Fue propuesta en el 2014 por Kyunghyun Cho.

Una celda GRU es una versión simplificada de la celda LSTM – y es igual de poderosa en términos de resultados, así que eso explica su popularidad.

Vamos viendo que es lo que le quitan a LSTM en la Celda GRU – aquí tenemos una imagen de LSTM.

- En primera, nuestra celda de selección se va, bye. El vector completo es escupido en cada paso de tiempo. Dicho eso, hay una nueva puerta de control que dictamina que parte del paso anterior se le va a mostrar a la capa principal.

- En segunda, un solo control de puerta está encargado de ambas la puerta de olvido y la de entrada. Si el control de puerta saca un 1, la puerta de olvidar está abierta, y la de entrada está cerrada. Si la salida es 0, entonces sucede lo puesto. En otras palabras, cuando necesitamos almacenar una memoria, la ubicación donde sea guardada es lo primero en ser borrada.
- Por último, ambos vectores de estado se unen en uno solo.

Keras nos da una capa de GRU, así que usarla en nuestra RNN es cosa super sencilla de solo reemplazar nuestro SimpleRNN o LSTM con un GRU.

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

A pesar de que los LSTMs y GRUs son buenos con la memoria, aun así, tienen problemas serios. No pueden hacer memoria en patrones de largo plazo en secuencias de 100 pasos o más, como muestras de audio, series de tiempo largas o oraciones muy extensas.

Una manera de resolver este problema es acortar las secuencias de entrada usando capas convolucionales.

- **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.

El chiste aquí es utilizar un *stride* o zancada que reduzca el tamaño de la cadena inicial, mientras se preservan los datos importantes originales. Acortando las secuencias, la capa convolucional puede ayudar a las capas de GRU a detectar patrones más largos.

Vamos armando un modelo con una capa convolucional para realizar la misma tarea que llevamos haciendo desde antes, utilizando celdas de GRU.

- **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.

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.

WaveNet todavía no ha sido implementado en keras, pero es una de las opciones más poderosas para la aplicación de RNNs en tareas de audio. Muchas veces LSTMs y GRUs no pueden procesar audios de complejidad.

Paper: https://arxiv.org/pdf/1609.03499.pdf

- ## Conclusión

¡Con esto terminamos RNNs! (por fin). En el siguiente capítulo veremos más sobre RNNs – especialmente cómo se pueden usar para taclear varias tareas de procesamiento de lenguaje natural.