Redes Neuronales

Juan Isaula

by: Juan Isaula
Febrero 2022

Contents

Chapter 1

Redes Neuronales

Alan Turing fue el primero en estudiar el cerebro como una forma de ver el mundo de la computación. Sin embargo, los primeros teóricos que concibieron los fundamentos de la computación neuronal fueron Warren McCulloch, un neurofisiólogo y Walter Pitts, un matemático, quienes en 1943, lanzaron una teória acerca de la forma de trabajar de las neuronas (Un cálculo lógico de la inminente idea de la actividad nerviosa - Boletin de Matemáticas Biofísica 5: 155 - 133). Ellos modelaron una red neuronal simple mediante circuitos eléctricos.

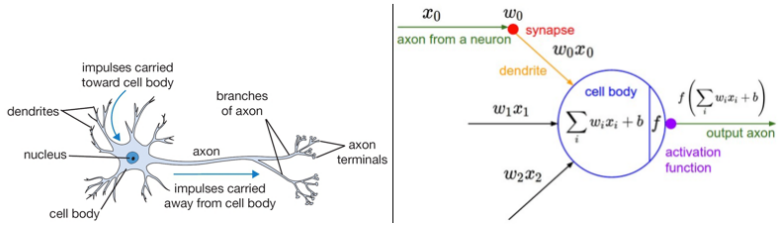
Las redes neuronales son modelos matemáticos / computacionales inspirados en como las neuronas se conectan entre sí en el cerebro. Los inicios (1940-1960). Las redes neuronales alcanzarón la fama a fines de la decada de los 80. El área de las Redes Neuronales originalmente se inspiró principalmente en el objetivo de modelar sistemas neuronales biológicos, pero desde entonces ha divergido y se ha convertido en una cuestión de ingeniería y logrando buenos resultados en tareas de Aprendizaje Automático.

1.1 Motivación biológica y conexiones

La unidad computacional básica del cerebro es una **neurona**. Se pueden encontrar aproximadamente 86 mil millones de neuronas en el sistema nervioso humano y están conectadas con aproximadamente $10^{14} - 10^{15}$ **sinapsis**. El siguiente diagrama muestra una caricatura de una neurona biológica (izquierda) y un modelo matemático común (derecha). Cada neurona recibe señales de entrada de sus **dendritas** y produce señales de salida a lo de su (único) **axón**. El axón finalmente se ramifica y se conecta a través de las sinapsis con las dendritas de otras neuronas. En el modelo computacional de una, las señales que viajan a lo largo de los axones (por ejemplo x_0) interactúan multiplicativamente (por ejemplo $w_0 x_0$) con las dendritas de la otra neurona en función de la fuerza sináptica en esa sinapsis (por ejemplo w_0). La idea es que las fuerzas sinápticas (los pesos w) son aprendibles y contralan la fuerza de la influencia (y su dirección: exitatoria (peso positivo) o inhibitoria (peso negativo)) de una neurona sobre otra. En el modelo básico las dendritas llevan la señal al cuerpo celular donde se suman todas. Si la suma final esta por encima de cierto umbral, la neurona puede dispararse , enviando un pico a lo largo de su axón. En el modelo computacional, asumimos que los tiempos precisos de los picos no importan y que solo la frecuencia de los disparos comunica la información. Con base en esta interpretación del código de tasa, modelamos la tasa de activación de la neurona con una función de activación f, que representa la frecuencia de los picos a lo largo del axón. Historicamente una opción común de función de activación es la función sigmoidea σ, ya que toma una entrada de valor real (la intensidad de la señal después de la suma) y la aplasta para que oscile entre 0 y 1. Veremos los detalles de estas funciones de activación más adelante en esta sección.

Figura 1. Dibujo de caricatura de una neurona biológica (izquierda) y su modelo matemámatico

(derecha)

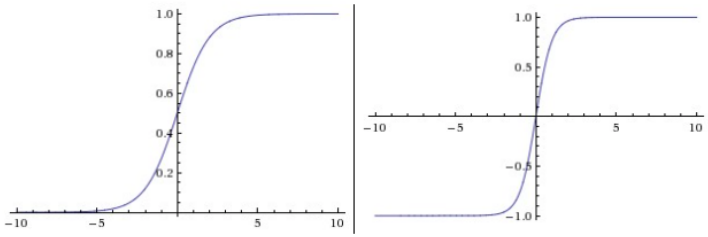


Cada neurona realiza un producto escalar con la entrada y sus pesos, agrega el sesgo y aplica la no linealidad (o función de activación), en este caso la sigmoide $\sigma(x) = \frac{1}{1+e^{-x}}$.

1.2 Funciones de activación de uso común

Cada función de activación (o no linealidad) toma un solo número y realiza cierta operación matemática fija sobre el. Hay varias funciones de activación que puede encontrar en la practica:

Figura 2: Izquierda: la no linealidad sigmoidea aplasta los números reales para que oscilen entre [0,1]. Derecha: La no linealidad *tanh* aplasta los números reales para que oscilen entre [-1,1].



En la práctica, la no linealidad sigmoidea ha caído recientemente en desgracia y rara vez se usa. Tiene dos grandes inconvenientes:

- **Las sigmoidea saturan y eliminan los gradientes:** Una propiedad muy indespensable de la neurona sigmoidea es que cuando la activación de la neurona se satura en cualquiera de los extremos de 0 y 1, el gradiente en estas regiones es casi cero. Recuerde que durante la retropropagación, este gradiente (local) se multiplicará por el gradiente de la salida de esta puerta para todo el objetivo. Por tanto, si el gradiente local es muy pequeño, efectivamente matara el gradiente y casi ninguna señal fluirá a través de la neurona a sus pesos y recursivamente a sus datos. Además, se debe prestar especial atención al inicializar los pesos de las neuronas sigmoideas para evitar la saturación. Por ejemplo, si los pesos iniciales son demasiado grandes, la mayoría de las neuronas se saturarán y la red apenas aprenderá.

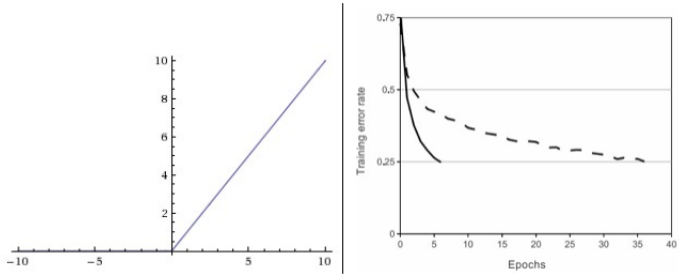
- **Las salidas sigmoidea no están centradas en cero:** Esto no es deseable ya que las neuronas en capas posteriores de procesamiento en una red neuronal estarían recibiendo datos que no están centrados en cero. Esto tiene implicaciones en la dinámica durante el descenso de gradiente, porque sli los datos que llegan a una neurona siempre son positivos (por ejemplo $x > 0$ por elemento en $f = w^T x+b$), entonces el gradiente en los pesos w durante la retropropagación, todo será positivo o todo negativo. Esto podría introducir dinámicas zigzag no deseadas en las actualizaciones de gradiente para los pesos. Sin embargo, tenga en cuenta que una vez que estos gradientes se suman en un lote de datos, la actualización final de los pesos puede tener signos variables, lo que mitiga un poco este problema.

tanh. La no linealidad de *tanh* se muestra en la figura 2 a la derecha. Aplasta un número de valor real al rango [-1,1]. Al igual que la neurona sigmoidea, sus activaciones se saturan, pero a diferencia de la neurona sigmoidea, su salida esta centrada en

cero. Por lo tanto, en la practica siempre se prefiere la no linealidad *tanh* a la no linealidad sigmoidea. Tenga en cuenta que la neurona *tanh* es simplemente una neurona sigmoidea escalada, en particular se cumple lo siguiente:

$$tanh(x) = 2\sigma(2x) - 1$$

Figura 3. Izquierda: Función de activación de la Unidad Lineal Rectificada (ReLU) que es cero cuando $x < 0$ y luego lineal con pendiente 1 cuando $x > 0$



ReLU: La Unidad Lineal Rectificada se ha vuelto muy popular en los últimos años. Calcula la función $f(x) = \max(0, x)$. En otras palabras, la activación simple tiene un umbral de cero (ver figura 3).

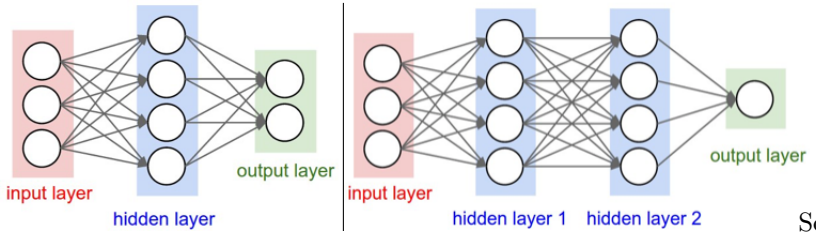
Leaky ReLU: Los ReLU con fugas son intentos de solucionar el problema de ReLU moribundo. En lugar de que la función sea cero cuando $x < 0$, una ReLU con fugas tendrá una pequeña pendiente positiva (de 0.01, más o menos).

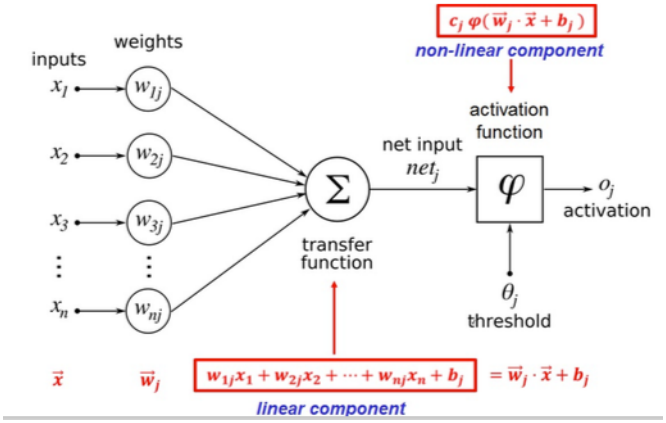
Lo discutido previamente, nos lleva a la siguiente pregunta ¿Qué tipo de neurona debo usar ? Use la no linealidad ReLU, tenga cuidado con sus tasas de aprendizaje y posiblemente controle la fracción de unidades muertas en una red. Si esto le preocupa, pruebe Leaky ReLU. Nunca use sigmoidea. Pruebe *tanh*, pero espere que funcione peor que ReLU.

1.3 Arquitecturas de Redes Neuronales

Las redes neuronales se modelan como colecciones de neuronas que están conectadas en un gráfico acíclico. En otras palabras, las salidas de algunas neuronas pueden convertise en entradas para otras neuronas. Los ciclos no están permitidos ya que eso implicaría un bucle infinito en el paso hacia adelante de una red. Para las redes neuronales regulares, el tipo de capas más común es la capa completamente conectada en la que las neuronas entre dos capas adyacentes estan completamente conectadas por pares, pero las neuronas dentro de una sola capa no comparten conexiones. A continuación se muestran dos ejemplos de topologías de redes neuronales que utilizan una pila de capas totalmente conectadas.

Figura 4. Izquierda: Red neuronal de 2 capas (una capa oculta de 4 neuronas y una capa de salida con 2 neuronas) y tres entradas. Derecha: una red neuronal de 3 capas con 3 entradas, dos capas ocultas de 4 neuronas cada una y una capa de salida. Note que en ambos casos hay conexiones (sinapsis) entre neuronas a través de capas, pero no dentro de una capa.





1.3.1 Dimensionamiento de redes neuronales

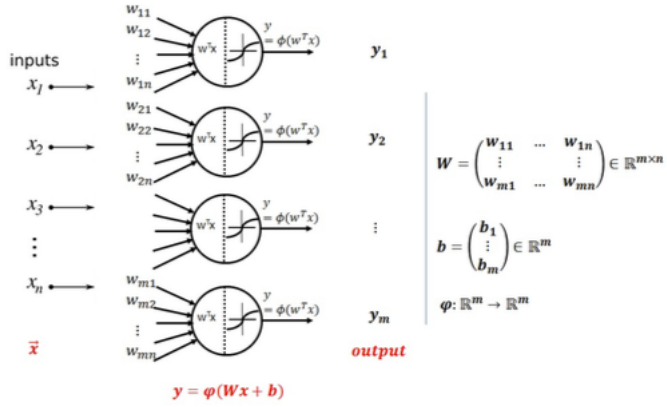
Las dos métricas que la gente suele utilizar para medir el tamaño de las redes neuronales son el número de neuronas o, más comúnmente, el número de parámetros. Trabajando con las dos redes de ejemplo en la imagen de arriba:

- La primera red (izquierda) tiene $4 + 2 = 6$ neuronas (sin contar las entradas), $[3 \times 4] + [4 \times 2] = 20$ pesos y $4 + 2 = 6$ sesgos, para un total de 26 parámetros aprendibles.

- La segunda red (derecha) tiene $4 + 4 + 1 = 9$ neuronas, $[3 \times 4] + [4 \times 4] + [4 \times 1] = 12 + 16 + 4 = 32$ pesos y $4 + 4 + 1 = 9$sesgos para un total de 41 parámetros.

Podemos modelar una neurona biológica de la siguiente manera:

Solemos combinar varias neuronas en una capa:



1.4 Función de Perdida

La función de perdida es la función objetivo que describe la diferencia entre las aproximaciones $\hat{y}_i$ y los valores reales y_i, de la muestra dada. Medira el error de la discrepancia entre cada para $(\hat{y}_i, y_i)$.

- **Regresión:** En un problema de regresión, las formas comunes de medir la diferencia entre la estimación y_i y la verdad básica deseada y_i son el MSE:

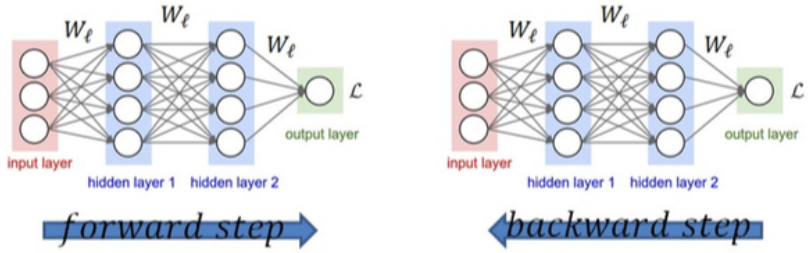
$$L(x_i, y_i) = MSE = \frac{1}{n}\sum_{i=1}^{n} ||y_i - \hat{y}_i||_2^2 \tag{1.1}$$

y

$$L(x_i, y_i) = MAE = \frac{1}{n}\sum_{i=1}^{n} ||y_i - \hat{y}_i||_1 \tag{1.2}$$

A veces, la función de pérdida también incorpora alguna regularización términos, dependiendo del problema y las preferencias del usuario. Por ejemplo

$$L(x_i, y_i) = \frac{1}{n}\sum_{i=1}^{n} ||y_i - \hat{y}_i||_1 + \lambda_1 \sum_{ij} ||w_{ij}|| + \lambda ||\nabla_w \hat{y}_i||_2^2 \tag{1.3}$$



1.5 Retropropagación

La retropropagación es el mecanismo que se utiliza para calcular los parámetros de la red neuronal (pesos $w_{ij\ell}$ y sesgos $b_{i\ell}$) a partir de la función de perdida.

Desde un punto de vista matemático, esto es simplemente calcular las derivadas

$$\nabla_{w_{ij\ell}} L(x, y) \qquad y \qquad \nabla_{b_{j\ell}} L(x, y) \tag{1.4}$$

para cada parámetro $w_{ij\ell}$ y sesgo $b_{i\ell}$.

Chapter 2

Redes Neuronales para Pronóstico de Series Temporales

En este capítulo, comenzaremos explorando algunas de las arquitecturas de redes neuronales más eficientes para el pronóstico de series de tiempo. Nos centraremos en la teória e implementación de las redes neuronales recurrentes (RNN), las unidades recurrentes cerradas (GRU) y las redes de memoria a corto plazo (LSTM). Comprender los principios básicos de las RNN será una buena base para su aplicación directa y el dominio de otras arquitecturas similares.

El término RNN se usa en dos sentidos:

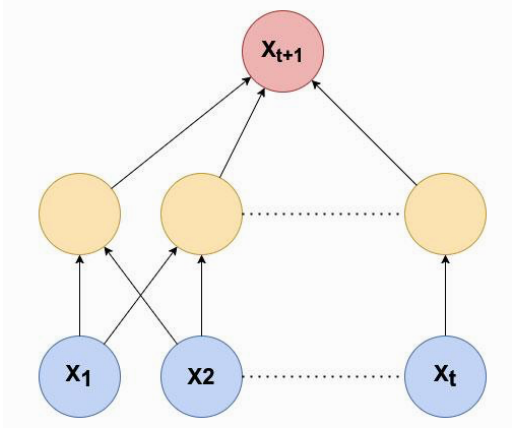
- **General:** Significa que cualquier red neuronal cuya arquitectura se construye utilizando un gráfico computacional recurrente.
- **Especial:** Significa una arquitectura concreta de una red neuronal, que se trata en esta sección.

La RNN tiene un concepto de estado oculto. Un estado oculto puede ser tratado como memoria interna. El estado oculto no intenta recordar todos los valores pasados de la secuencia sino solo su efecto. Debido a la memoria interna, las RNN pueden recordar cosas importantes sobre su entrada, lo que les permite ser muy precisos al predecir valores futuros.

Veamos la RNN en acción con un ejemplo. Digamos que tenemos el siguiente problema de pronóstico:

predecir el monto del bono de trabajo de John Smith al final del mes

Tenemos algunos eventos como secuencia de entrada. RNN toma la memoria interna (estado oculto), la combina con un evento y devuelve la nueva memoria interna actualizada (estado oculto). La memoria interna (estado oculto) contiene un resumen de eventos anteriores.



La figura anterior es el caso de una red neuronal conectada como modelo de predección de un paso.

Digamos que tenemos los siguientes eventos en el mes actual:

- John Smith demuestra una excelente presentación
- La capitalización de la empresa sube un 2%.
- John Smith llega tarde a una conversación importante.

Cada uno de estos eventos, de alguna manera, afecta el tamaño del bono mensual. Digamos que tenemos algún sistema de puntuación que registra puntuaciones. La predicción de la bonificación futura se realiza en función de los puntos acumulados

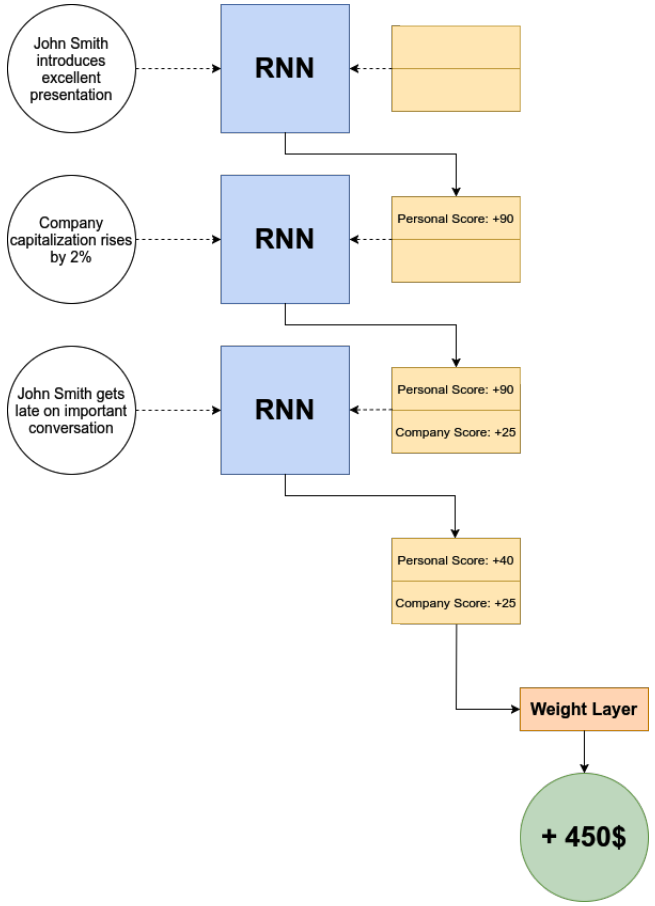
Eventos	**Efectos**	**Sistema puntos**
Comienzo		Puntaje personal: 0 Puntaje compañía: 0
Excelente presentación	+90 puntos	Puntaje personal: 90 puntaje compañía:0
Capitalización empresa +2%	+25 puntos	Puntaje personal:90 Puntaje compañía:25
John llega tarde conversación	-50 puntos	Puntaje personal: 40 Puntos compañía: 25

El esquema previo representa el sistema de puntuación de bonificación mensual. En el último paso de predicción, los puntos se suman según un cierto sistema de pesos:

$$\text{puntuación personal} \times 10\$ + \text{puntuación compañía} \times 2\$ = 450\$$$

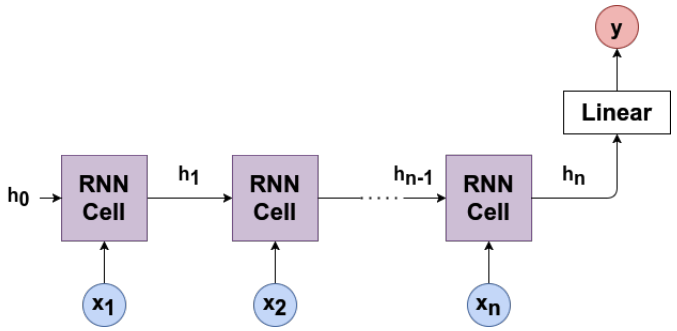
RNN tiene un sistema de puntuación (estado oculto). Aprende a actualizar este sistema de puntuación de acuerdo con la secuencia de entrada de valores. Y asigne los puntajes recopilados a alguna predicción.

De forma simplificada se puede demostrar la actividad de RNN como se muestra en la siguiente figura. La figura que verán a continuación presenta una RNN en acción con 3 eventos. Es por eso que un RNN se llama así. En procesamiento de cada nuevo valor en la secuencia vuelve a calcular el estado oculto de forma recurrente. Este principio subyace en todas las arquitecturas recurrentes: RNN, GRU y LSTM.



Ya tenemos cierta comprensión de cómo actúa una RNN, así que estudiemos la teoría de RNN de una manera más formal. En las RNN, la secuencia de entrada recorre un ciclo. Cuando toma una decisión, considera la entrada actual y también lo que ha

aprendido de las entradas que recibió anteriormente. el gráfico computacional de RNN se puede presentar de la siguiente manera:

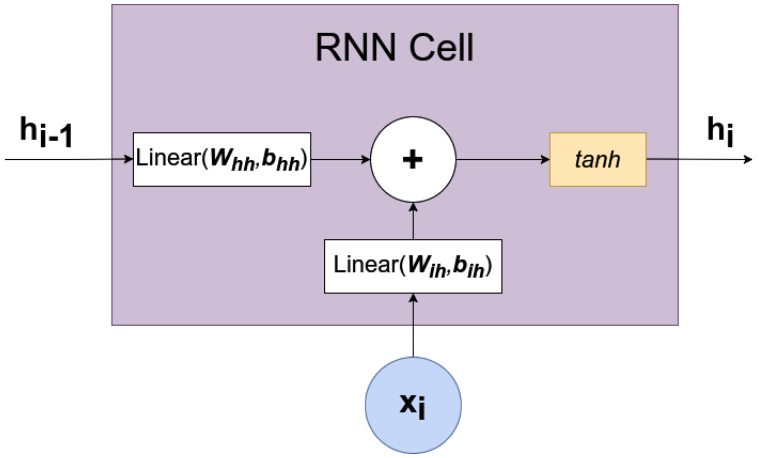


donde,

- $x_1, x_2, ..., x_n$ es la secuencia de entrada.
- h_i capas ocultas. h_i es un vector de longitud h.
- $RNNCell$: representa la capa de red neuronal que calcula la siguiente función

$$h_t = tanh(w_{ih}x_t + b_{ih} + w_{hh}h_{(t-1)} + b_{hh})$$

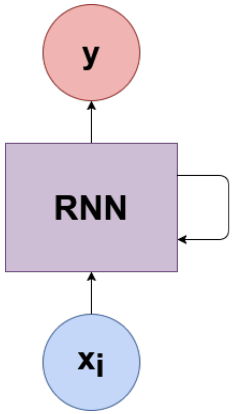
A continuación se presenta una demostración de como construir el gráfico de RNN Cell:



La RNN Cell combina información sobre el valor actual de la secuencia x_i y el estado oculto previamente oculto h_{i-1}. RNN Cell devuelve un estado oculto actualizado después de aplicar la función de activación. Una RNN tiene los siguiente parámetros, que se ajustan durante el entrenamiento:

- w_{ih} Pesos ocultos de entrada.
- b_{ih} Sesgo oculto de entrada.
- w_{hh} Pesos ocultos ocultos.
- b_{hh} Sesgos ocultos ocultos.

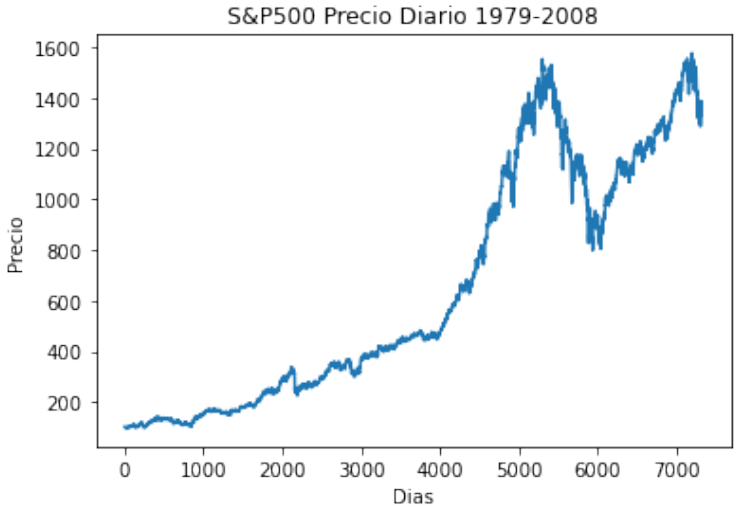
A veces, los científicos de datos usan la siguiente representación de RNN:

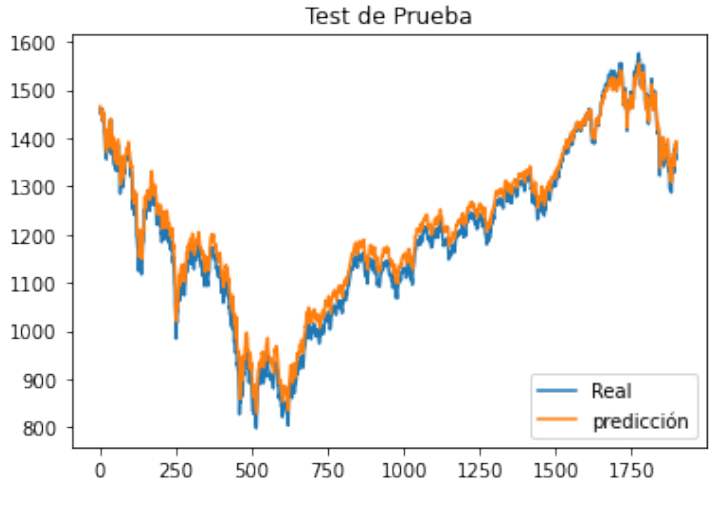



```python
import torch.nn as nn

class RNN(nn.Module):

    def __init__(self,
                 hidden_size,
                 in_size = 1,
                 out_size = 1):
        super(RNN, self).__init__()
        self.rnn = nn.RNN(
            input_size = in_size,
            hidden_size = hidden_size,
            batch_first = True)
        self.fc = nn.Linear(hidden_size, out_size)

    def forward(self, x, h = None):
        out, _ = self.rnn(x, h)
        last_hidden_states = out[:, -1]
        out = self.fc(last_hidden_states)
        return out, last_hidden_states
```

Vemos que la capa **torch.nn.RNN** devuelve estados ocultos para cada iteración. El último estado oculto se pasa a la lineal. Nuestro modelo devuelve dos salidas: predicción y estado oculto. Es crucial reutilizar estados ocultos durante la evaluación de RNN. Consideraremos esto más a fondo. En este capítulo utilizaremos un conjunto de datos del precio diario del S&P500 (https://github.com/JJ-team-2021/Modelos-de-Caja-Negra/blob/main/RNN/SP500_Datos.csv)

Ahora estamos listos para examinar la implementación **PyTorch** de RNN:



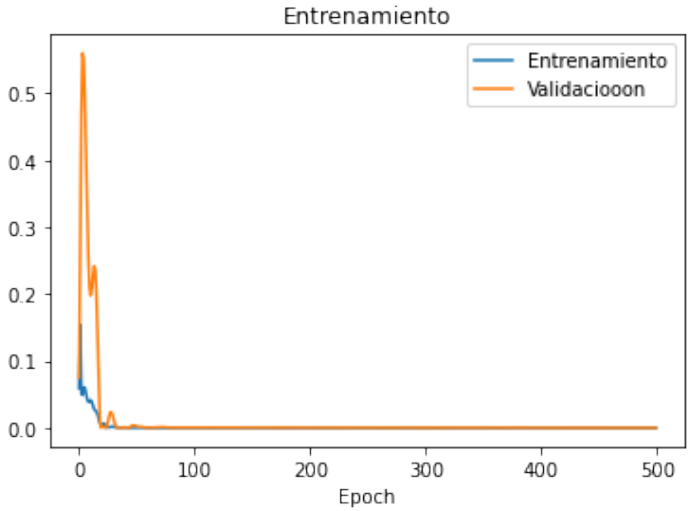


Podemos ver que esta es una serie de tiempo realmente complicada. Tiene varios factores de estacionalidad con picos difícilmente predicibles.

Vamos a mostrar cómo se desempeña la RNN en la serie de tiempo por día del S&P500.

La RNN muestra un gran rendimiento en el conjunto de datos de prueba. El modelo que hemos entrenado predice picos estacionales.

Finalmente, examinemos el proceso de entrenamiento en sí mismo



El progreso del entrenamiento es fluido, sin picos bruscos e impredecibles. Ahora podemos afirmar con la promesa y la eficacia de la aplicación de la RNN a los problemas de pronósticos de series de tiempo. Las RNN pueden encontrar dependencias y patrones dificiles en secuencias y producir prediciones precisas.

A pesar de todas las ventajas de RNN, tiene importantes desventajas:

- Debido a la complejidad computacional, sufren problemas de gradiente de desaparición. El proceso de entrenamiento se vuelve demasiado lento. El problema del gradiente de fuga es un problema común a todas las RNN.

- El estado oculto se actualiza en cada iteración, lo que dificultad el almacenamiento de información a largo plazo en RNN. Las arquitecturas GRU y LSTM resuelven este problema. Tienen enfoques similares sobre cómo almacenar información a largo plazo.