

# Sequences, Time Series and Prediction



DeepLearning.AI

Víctor Díaz Bustos

21 de marzo de 2023

# Índice

<b>1. Introducción</b>	<b>3</b>
<b>2. Secuencias y predicciones</b>	<b>4</b>
2.1. Series temporales . . . . .	4
2.2. Machine learning aplicado a series temporales . . . . .	4
2.3. Patrones . . . . .	5
2.4. Conjuntos de entrenamiento, validación y prueba . . . . .	8
2.5. Métricas de evaluación . . . . .	9
2.6. Media móvil . . . . .	10
2.7. Ventana deslizante vs. ventana centrada . . . . .	12
<b>3. NN profundas para series temporales</b>	<b>13</b>
3.1. Preparación de características y etiquetas . . . . .	13
3.2. Red neuronal con una simple capa . . . . .	14
3.3. Predicciones . . . . .	15
3.4. Deep Neural Network . . . . .	17
3.5. Tasa de aprendizaje óptima . . . . .	18
<b>4. NN recurrentes para series temporales</b>	<b>22</b>
4.1. Forma de las entradas de una RNN . . . . .	23
4.2. Salida de la secuencia . . . . .	26
4.3. Capas Lambda . . . . .	27
4.4. Ajustar LR dinámicamente . . . . .	28
4.5. Función de pérdida de Huber . . . . .	30
4.6. LSTM . . . . .	30
4.7. Construir RNN . . . . .	31
4.7.1. LSTM bidireccional . . . . .	32
<b>5. Datos reales de series temporales</b>	<b>34</b>
5.1. Convoluciones . . . . .	34
5.2. LSTM Bidireccional . . . . .	35
5.3. Caida exponencial de LR . . . . .	37
<b>6. Datos importantes</b>	<b>38</b>

## Índice de figuras

1.	Tendencias ( <i>trend</i> ) . . . . .	5
2.	Estacionalidad ( <i>seasonality</i> ) . . . . .	5
3.	Tendencia+Estacionalidad . . . . .	5
4.	Ruido blanco . . . . .	6
5.	Autocorrelacion . . . . .	6
6.	Tendencia+Estacionalidad+Autocorrelacion+Ruido + Predicción	7
7.	Partición fija . . . . .	8

## **1. Introducción**

Este curso se enfoca en los modelos de secuencia, específicamente en la predicción de series temporales. Estas series son datos que cambian con el tiempo, como el precio de las acciones en la bolsa de valores, el clima o la actividad de las manchas solares. El curso comienza por crear una secuencia sintética de datos para entender los atributos comunes de las series de tiempo, como la estacionalidad, tendencias y el ruido. Luego, se exploran métodos estadísticos de aprendizaje automático para predecir los datos basados en estos atributos. El objetivo final del curso es aplicar estos conceptos a la predicción de la actividad de las manchas solares, que es importante para la NASA y otras agencias espaciales debido a su impacto en las operaciones de los satélites.

## 2. Secuencias y predicciones

En esta sección, nos centraremos en series temporales. Repasaremos algunos ejemplos de diferentes tipos de series temporales, así como también veremos previsiones básicas a su alrededor. Prepararemos datos de series temporales para algoritmos de aprendizaje automático. Así que empecemos con un vistazo a las series temporales, lo que son, y los diferentes tipos de ellas que puedes encontrar.

### 2.1. Series temporales

Las series temporales están en todas partes, en los precios de las acciones, pronósticos meteorológicos, tendencias históricas, etc.

Hay casos en los que hay un valor único en cada paso de tiempo, y como resultado, el término univariado se utiliza para describirlos. También podemos encontrar series temporales que tienen varios valores en cada paso de tiempo (*series temporal multivariante*). Los gráficos de series temporales multivariadas pueden ser útiles para comprender el impacto de los datos relacionados.

### 2.2. Machine learning aplicado a series temporales

¿Qué tipos de cosas podemos hacer con el aprendizaje automático en series temporales?

- La primera y más obvia es la **predicción de pronósticos basados en los datos**. Así que, por ejemplo, con un gráfico de tasas de natalidad y mortalidad sería muy útil predecir valores futuros para que las agencias gubernamentales puedan planificar la jubilación, la inmigración y otros impactos sociales de estas tendencias.
- En algunos casos, es posible que también desee **proyectar hacia el pasado** para ver cómo llegamos a donde estamos ahora. Este proceso se llama **imputación**.
- O es posible que simplemente desee **rellenar** agujeros en sus datos para datos que aún no existen. Con la imputación, podemos llenarlos.
- Además, la predicción de series temporales se puede utilizar para **detectar anomalías**.
- La otra opción es analizar las series temporales para **detectar patrones** en ellas que determinan lo que generó la serie misma. Un ejemplo clásico de esto es **analizar ondas sonoras** para **detectar palabras** en ellas que pueden ser utilizadas como una red neuronal para el **reconocimiento de voz**.

Las series temporales vienen en todas las formas y tamaños, pero hay una serie de patrones muy comunes. Así que es útil reconocerlos cuando los ves.

## 2.3. Patrones

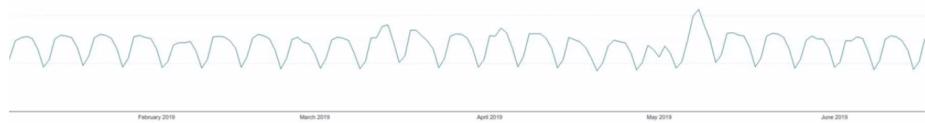
A continuación vemos algunos patrones comunes:

**Figura 1:** Tendencias (*trend*)



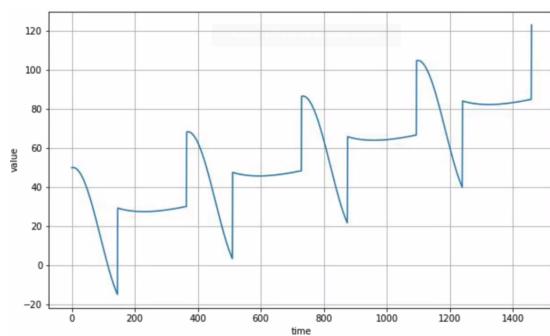
Tienen una dirección específica en la que se están moviendo. Esta es una tendencia hacia arriba.

**Figura 2:** Estacionalidad (*seasonality*)



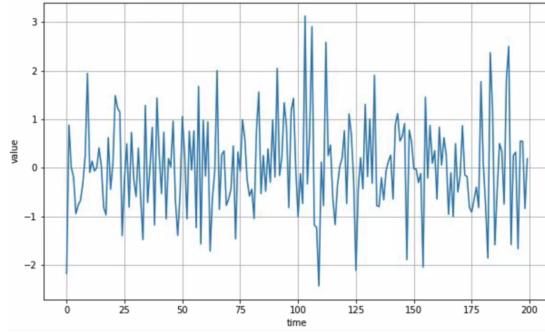
Los patrones se repiten a intervalos predecibles.

**Figura 3:** Tendencia+Estacionalidad



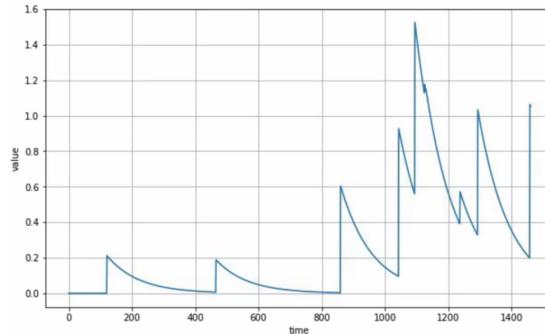
Algunas series temporales pueden tener una combinación de tendencia y estacionalidad. Hay una tendencia alcista global pero hay picos y valles locales.

**Figura 4:** Ruido blanco



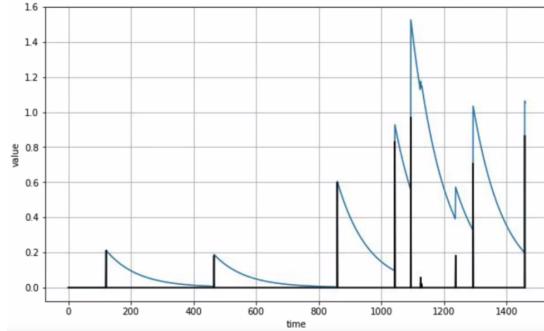
Por supuesto, también hay algunos que son probablemente no predecibles en absoluto y solo un conjunto completo de valores aleatorios produciendo lo que típicamente se llama ruido blanco. No hay mucho que podamos hacer con este tipo de datos.

**Figura 5:** Autocorrelacion



Consideremos esta serie temporal. No hay tendencia y no hay estacionalidad. Los picos aparecen en marcas de tiempo aleatorias. No se puede predecir cuándo va a pasar de nuevo o cuán fuertes serán. Pero claramente, toda la serie **no es aleatoria**. Entre los picos hay un tipo muy **determinista** de **descomposición**.

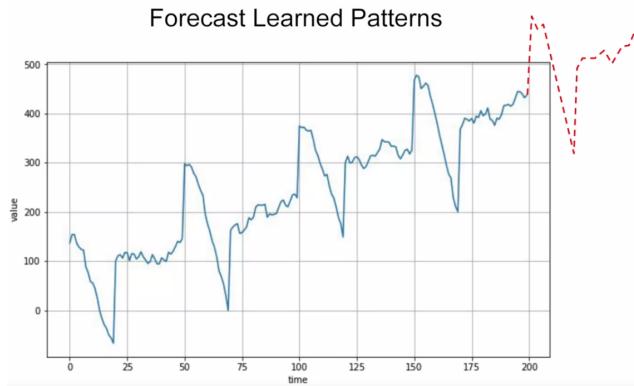
$$v(t) = 0.99 \times v(t-1) + \text{occasional spike}$$



Podemos ver aquí que el valor de cada paso de tiempo es *99 %* del valor de el paso de tiempo anterior más un pico ocasional.

Esta es una **serie cronológica automática correlacionada**. Es decir, se correlaciona con una copia retrasada de sí misma a menudo llamada un *retraso*.

**Figura 6:** Tendencia+Estacionalidad+Autocorrelacion+Ruido + Predicción



Las series temporales que encontrarás en la vida real probablemente tienen un poco de cada una de estas características: tendencia, estacionalidad, autocorrelación y ruido. Como hemos visto, un modelo de aprendizaje automático está diseñado para detectar patrones, y cuando detectamos patrones podemos hacer predicciones.

En su mayor parte, esto también puede funcionar con series temporales excepto por el ruido que es impredecible. Pero debemos reconocer que este asume que los patrones que existieron en el pasado continuarán por su supuesto en el futuro.

Para predecir sobre esto podríamos entrenar por un período limitado de tiempo. Por ejemplo, aquí donde doy sólo los últimos 100 pasos. Probablemente obtendrá un mejor rendimiento que si se hubiera entrenado en toda la serie temporal. Pero eso es romper el molde para la máquina típica, aprendiendo

donde siempre asumimos que más datos es mejor. Pero para la predicción de series de tiempo realmente depende de la serie de tiempo.

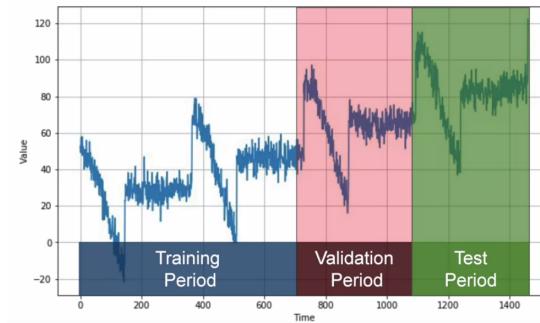
Si es **estacionario**, lo que significa que su comportamiento no cambia con el tiempo, entonces genial. Cuantos más datos tengas, mejor. Pero si **no es estacionario** entonces la ventana de tiempo óptima que debe usar para el entrenamiento variará.

## 2.4. Conjuntos de entrenamiento, validación y prueba

Podríamos, por ejemplo, tomar el último valor y asumir que el siguiente valor será el mismo, y esto se llama **predicción ingenua**.

Podemos hacer eso para obtener una línea de base por lo menos, y créanlo o no, esa línea de base puede ser bastante buena. Pero, ¿cómo se mide el rendimiento? Para medir el rendimiento de nuestro modelo de previsión, normalmente queremos dividir las series temporales en un *período de entrenamiento*, un *período de validación* y un *período de prueba*.

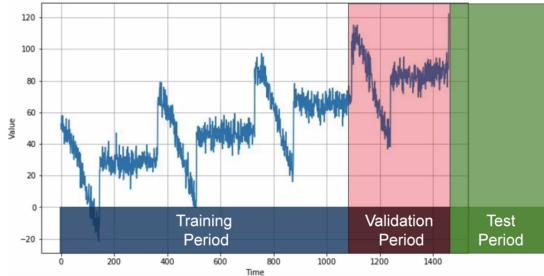
**Figura 7:** Partición fija



Si la serie temporal tiene alguna estacionalidad, generalmente desea asegurarse de que **cada período contenga un número entero de estaciones**.

Los datos de prueba son los datos más cercanos que tiene al punto actual en el tiempo. Y como tal, a menudo es la señal más fuerte para determinar valores futuros.

Debido a esto, en realidad es bastante común **renunciar al conjunto de pruebas**. Y sólo entrenar, usando un período de entrenamiento y un período de validación, y el conjunto de pruebas es en el futuro.



Vamos a seguir parte de esa metodología en este curso. La partición fija como esta es muy simple y muy intuitiva, pero también hay otra manera. Comenzamos con un corto período de entrenamiento, y poco a poco lo aumentamos. En cada iteración, entrenamos al modelo en un período de entrenamiento. Y lo usamos para pronosticar el día siguiente, o la semana siguiente, en el período de validación. esto se llama partición ***roll-forward***. Podría verlo haciendo particiones fijas varias veces, y luego refinando continuamente el modelo.

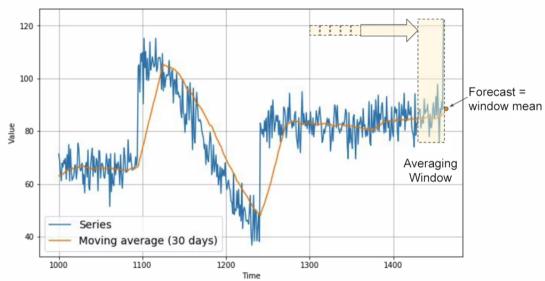
## 2.5. Métricas de evaluación

Una vez que tengamos un modelo y un período, entonces podemos evaluar el modelo en él, y necesitaremos una métrica para calcular su rendimiento.

- **errors** = `forecast - actual`. Es la diferencia entre los valores previstos de nuestro modelo y los valores reales durante el período de evaluación.
- **mse** = `np.square(errors).mean()`. La métrica más común para evaluar el rendimiento de predicción de un modelo es el error cuadrado medio (mse) donde elevamos al cuadrado los errores y luego calculamos su media. La razón de esto es deshacerse de los **valores negativos** y penalizar más los errores grandes.
- **rmse** = `np.sqrt(mse)`. Si queremos que la media del cálculo de nuestros errores sea de la **misma escala** que los errores originales, entonces solo obtenemos su raíz cuadrada.
- **mae** = `np.abs(errors).mean()`. En lugar de cuadrar para deshacerse de los negativos, solo usa su **valor absoluto**. Esto no penaliza los errores grandes tanto como lo hace el mse. Dependiendo de la tarea, convendrá más la mae o la mse.
- **mape** = `np.abs(errors / x.valid).mean()`. Mide el **porcentaje medio de error absoluto**, esta es la relación media entre el error absoluto y el valor absoluto, esto da una idea del tamaño de los errores en comparación con los valores.

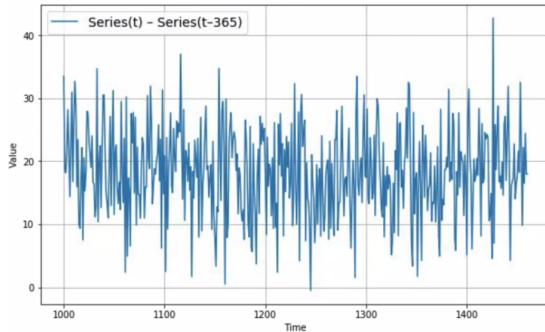
## 2.6. Media móvil

Un método común y muy simple de **predicción** es calcular una media móvil. La idea aquí es que la línea amarilla es una trama de la media de los valores azules sobre un período fijo llamado **ventana de promediación**.

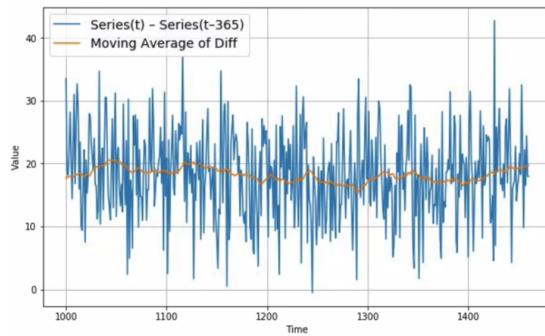


Esto elimina mucho ruido y nos da una curva que emula más o menos la serie original, pero no anticipa tendencia o estacionalidad.

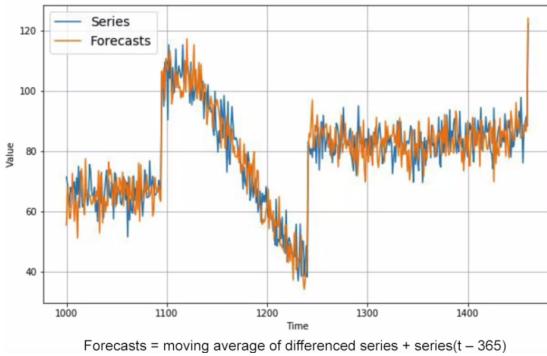
La **diferenciación** es un método para eliminar la tendencia y la estacionalidad.



Así, en lugar de estudiar la serie temporal en sí, estudiamos la diferencia entre el valor en el tiempo T y el valor en un período anterior.

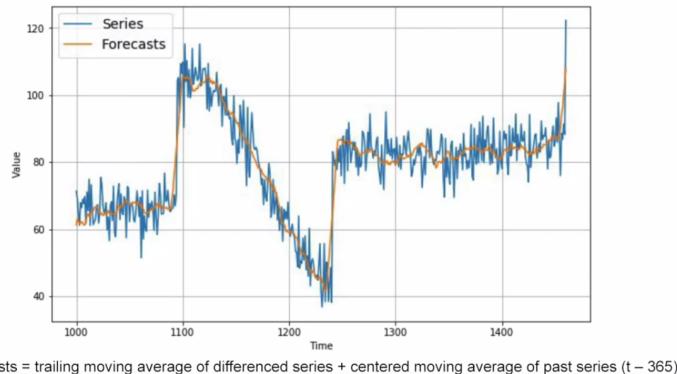


Entonces podemos utilizar una media móvil para pronosticar esta serie temporal que nos da estas previsiones. Pero estos son sólo pronósticos para la serie de tiempo de diferencia, no la serie de tiempo original. Para obtener las previsiones finales para la serie temporal original, sólo tenemos que añadir el valor en el tiempo T menos 365, y obtendremos estas previsiones.



Es un poco mejor que el pronóstico ingenuo pero no tremadamente mejor. La media móvil eliminó una gran cantidad de ruido, pero las previsiones finales siguen siendo bastante ruidosas.

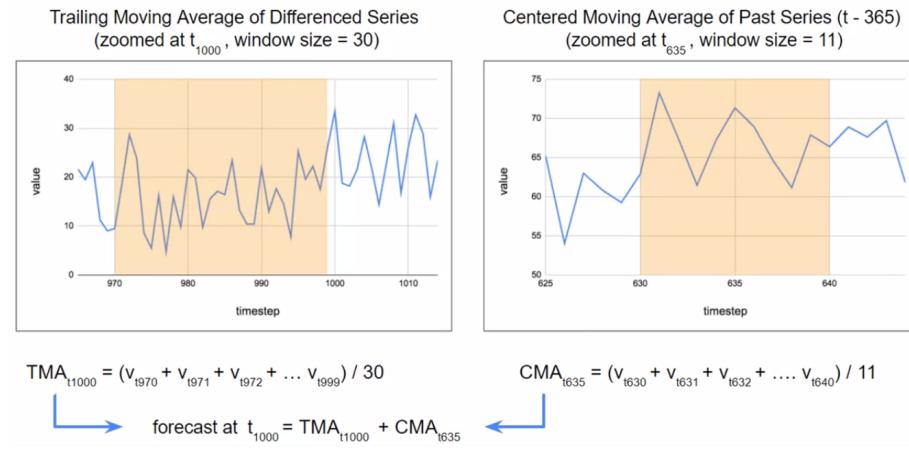
¿De dónde viene ese ruido? Bueno, eso viene de los valores pasados que agregamos de nuevo en nuestras previsiones. Así que podemos mejorar estos pronósticos eliminando también el ruido pasado usando una media móvil sobre eso. Si hacemos eso, obtenemos pronósticos mucho más suaves.



Ahora eso es mucho mejor que todos los métodos anteriores. De hecho, ya que se genera la serie, podemos calcular que un modelo perfecto dará un error absoluto medio de aproximadamente 4 debido al ruido. Aparentemente, con este enfoque, no estamos demasiado lejos de lo óptimo. Los enfoques simples a veces pueden funcionar bien.

## 2.7. Ventana deslizante vs. ventana centrada

Ten en cuenta que usamos la **ventana deslizante** al calcular el promedio móvil de los **valores presentes**. Pero utilizamos una **ventana centrada** para calcular el promedio móvil de los **valores pasados**. Los promedios móviles utilizando **ventanas centradas** pueden ser **más precisos** que los que utilizan ventanas deslizantes. Pero **no podemos utilizar ventanas centradas para suavizar los valores presentes** ya que no conocemos los valores futuros. Sin embargo, para **suavizar los valores pasados** podemos permitirnos usar **ventanas centradas**. Bien, ahora hemos visto algunos métodos estadísticos para predecir los próximos valores en una serie de tiempo.



### 3. NN profundas para series temporales

En esta sección empezaremos a mirar pronósticos usando un modelo denso, y cómo difiere de predicciones más ingenuas basadas en análisis numérico simple de los datos. Aprenderemos a aplicar una nueva red a estas secuencias.

#### 3.1. Preparación de características y etiquetas

En primer lugar, como con cualquier otro problema ML, tenemos que dividir nuestros datos en características y etiquetas. En este caso nuestra característica es efectivamente una serie de valores en la serie, con nuestra **etiqueta** siendo el **siguiente valor**. Llamaremos a ese número de valores que tratarán como nuestra característica, el **tamaño de la ventana**, donde estamos tomando una ventana de los datos y entrenando un modelo ML para predecir el siguiente valor.

```
1 dataset = tf.data.Dataset.range(10)
2 dataset = dataset.window(5, shift=1, drop_remainder=True)
3 dataset = dataset.flat_map(lambda window: window.batch(5))
4
5 for window in dataset:
6     print(window.numpy())
7
8 dataset = dataset.map(lambda window: (window[:-1], window[-1]))
9 dataset = dataset.shuffle(buffer_size=10)
10 dataset = dataset.batch(2).prefetch(1)
11
12 for x, y in dataset:
13     print("x = ", x.numpy())
14     print("y = ", y.numpy())
```

- [1] Crea datos en un rango de 10 valores.
- [2] Expande el conjunto de datos mediante el uso de ventanas (*windowing*). Sus parámetros son el tamaño de la ventana, cuánto queremos cambiar por cada vez y *drop\_remainder*, que edita nuestra ventana un poco para que tengamos datos de tamaño regular, truncando los datos y eliminando todos los restos.
- [3] Añade los datos a una lista de *numpy* para poder usarlos con el aprendizaje automático.
- [6] Llamamos al método *.numpy()* en cada elemento en el conjunto de datos, y cuando imprimimos ahora vemos que tenemos una lista numpy.
- [8] El siguiente paso es dividir los datos en entidades y etiquetas. Para cada elemento de la lista, tiene sentido tener todos los valores excepto el último en ser la característica, y luego el último puede ser la etiqueta. Esto se puede lograr con la función *.map()*

[9] A continuación, se baraja el conjunto de datos. El parámetro *buffer\_size* es la cantidad de elementos de datos que tenemos.

[10] Por último, podemos ver el procesamiento por lotes de los datos. Tomará un parámetro de tamaño, en este caso es 2. Así que lo que haremos es **agrupar los datos en conjuntos de 2**. Ahora tenemos 3 lotes de 2 elementos de datos cada uno. Y si nos fijamos en el primer set, verá las *x* e *y* correspondientes. Así que cuando *x* sea 4, 5, 6 y 7, nuestra *y* es 8, o cuando *x* es 0, 1, 2, 3, verá que nuestra *y* es 4.

```
x = [[4 5 6 7] [1 2 3 4]]  
y = [[8] [5]]  
x = [[3 4 5 6] [2 3 4 5]]  
y = [[7] [6]]  
x = [[5 6 7 8] [0 1 2 3]]  
y = [[9] [4]]
```

### 3.2. Red neuronal con una simple capa

Comencemos con una super simple que es efectivamente una regresión lineal. Mediremos su precisión, y luego trabajaremos desde allí para mejorarlo. Aquí está el código para hacer una regresión lineal simple.

```
1 split_time = 1000  
2 window_size = 20  
3 batch_size = 32  
4 shuffle_buffer_size = 1000  
5  
6 time_train = time[:split_time]  
7 x_train = series[:split_time]  
8 time_valid = time[split_time:]  
9 x_valid = series[split_time:]  
10  
11 dataset = windowed_dataset(series, window_size, batch_size,  
12     shuffle_buffer_size)  
12 10 = tf.keras.layers.Dense(1, input_shape=[window_size])  
13 model = tf.keras.models.Sequential([10])  
14  
15 model.compile(loss="mse", optimizer=tf.keras.optimizers.SGD(  
16     learning_rate=1e-6, momentum=0.9))  
16 model.fit (dataset, epochs=100, verbose=0)
```

[11] Esta función replica el código del apartado 3.1.

[12] Crea una sola capa densa con su forma de entrada es el tamaño de la ventana. Para la regresión lineal, eso es todo lo que necesita.

- [13] Define el modelo como una secuencia que contiene la capa única.
- [14] y [15] Compila y entrena el modelo. Se utiliza función de pérdida de *Error Cuadrático Medio* (**MSE**) y el optimizador de *Descenso de Gradiente Estocástico* (**SGD**), utilizamos esta metodología en lugar de la cadena cruda, para poder establecer parámetros para inicializarla como la tasa de aprendizaje (*learning\_rate*) y el impulso (*momentum*).
- [16] Finalmente, encajaremos el modelo en más de 100 épocas. El parámetro **verbose** indica la cantidad de información que se mostrará durante el entrenamiento del modelo. Este parámetro puede tomar valores enteros del 0 al 2 (0 - no se muestra ningún mensaje durante el entrenamiento, 1 - se muestra una barra de progreso durante el entrenamiento, indicando el número de épocas completadas y la métrica de entrenamiento y validación (si se especifica), 2 - se muestra una línea por cada época completada, indicando la métrica de entrenamiento y validación (si se especifica)).

### 3.3. Predicciones

Si miramos los valores y vemos que estos son los pesos para los valores en esa marca de tiempo particular y  $b$  es el sesgo o la pendiente, podemos hacer una regresión lineal estándar así para predecir el valor de  $y$  en cualquier paso multiplicando los valores  $x$  por los pesos y luego agregar el sesgo.

```
print("Layer weights: {}".format(l0.get_weights()))
Layer weights: [array([[ 0.01633573], ← W10
 [-0.02911791], ← W11
 [ 0.00845617], ← W12
 [-0.02175158], ← W13
 [ 0.04962169], ← W14
 [-0.03212642], ← W15
 [ 0.02596855], ← W16
 [-0.00689476], ← W17
 [ 0.0616533 ], ← W18
 [-0.00668752], ← W19
 [-0.02735964], ← W20
 [ 0.0377918 ], ← W21
 [-0.02855931], ← W22
 [ 0.05299238], ← W23
 [-0.0121608 ], ← W24
 [ 0.00138755], ← W25
 [ 0.0905595 ], ← W26
 [ 0.19994621], ← W27
 [ 0.2556632 ], ← W28
 [ 0.41660047]], dtype=float32), array([ 0.01430958], dtype=float32)]
```

$$Y = W_{10}X_0 + W_{11}X_1 + W_{12}X_2 + \dots + W_{19}X_{19} + b$$

Así que, por ejemplo, si tomo 20 elementos de mi serie e imprimo, puedo ver los valores 20x. Si quiero predecirlas, puedo pasar esa serie a mi modelo para obtener una predicción. El nuevo eje NumPy luego simplemente lo cambia a la dimensión de entrada que es utilizada por el modelo.

```
1 model.predict(series[1:21][np.newaxis])
```

La salida se verá así.

```
[ 49.35275  53.314735 57.711823 48.934444 48.931244 57.982895 53.897125
 47.67393  52.68371  47.591717 47.506374 50.959415 40.086178 40.919415
 46.612473 44.228207 50.720642 44.454983 41.76799  55.980938]
array([[ 49.08478]], dtype=float32)
```

La matriz superior es los 20 valores que proporcionan la entrada a nuestro modelo y la parte inferior es el valor previsto de vuelta desde el modelo.

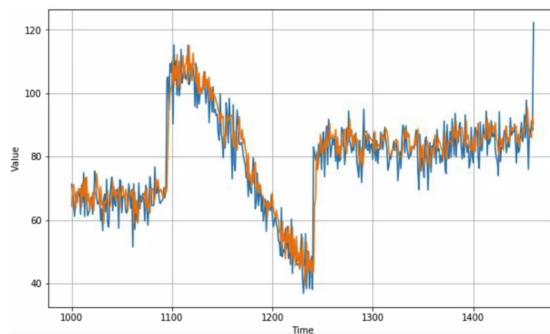
Así que si queremos trazar nuestras previsiones para cada punto en la serie temporal relativa a los 20 puntos anteriores donde nuestro tamaño de ventana era 20, podemos escribir código como este.

```
1 forecast = []
2 for time in range (len(series) - window_size):
3     forecast.append(model.predict(series[time : time+window_size][
4         np.newaxis]))
5 forecast = forecast[split_time-window_size : ]
6 results = np.array(forecast)[:, 0, 0]
```

[3] Iteramos sobre la serie tomando rebanadas y tamaño de ventana, prediciendo, y agregando los resultados a la lista de pronósticos.

[5] y [6] Habíamos dividido nuestra serie temporal en entrenamiento y pruebas tomando todo antes de un cierto tiempo es entrenamiento y el resto es validación. Así que tomaremos las previsiones después de el tiempo de división y las cargaremos en una matriz NumPy para gráficos.

El gráfico se ve así con los valores reales en azul y los predichos en naranja.



Ahora, vamos a medir el error absoluto medio (**MAE**) como hemos hecho antes, y podemos ver que estamos en un rango estimado similar al que estábamos con un análisis complejo que hicimos anteriormente.

```
1 tf.keras.metrics.mean_absolute_error(x_valid, results).numpy()
2 # 4.9526777
```

Ahora solo hay que usar una sola capa en una red neuronal para calcular una regresión lineal. Veamos si podríamos hacerlo mejor con un DNN completamente conectada.

### 3.4. Deep Neural Network

Vamos a ver si podemos mejorar la precisión de nuestro modelo utilizando una DNN. No es muy diferente del modelo de regresión lineal que vimos anteriormente. Y esta es una red neuronal profunda relativamente simple que tiene tres capas.

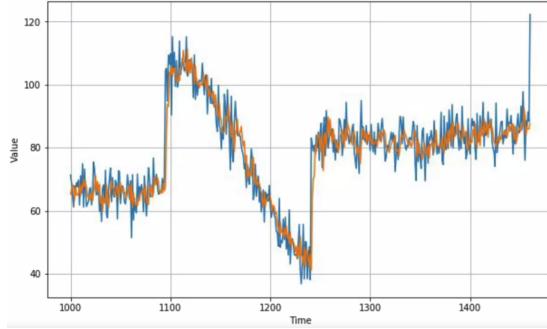
```
1 dataset = windowed_dataset(x_train, window_size, batch_size,
2                             shuffle_buffer_size)
3
4 model_tune = tf.keras.models.Sequential([
5     tf.keras.layers.Dense(10, activation="relu", input_shape=[window_size]),
6     tf.keras.layers.Dense(10, activation="relu"),
7     tf.keras.layers.Dense(1)
8 ])
9 model.compile(loss="mse", optimizer=tf.keras.optimizers.SGD(
10    learning_rate=1e-6, momentum=0.9))
model.fit(dataset, epochs=100, verbose=0)
```

[3] , [4], [5] y [6] Definiremos el modelo. Vamos a mantenerlo simple con tres capas de 10, 10 y 1 neuronas. *input\_shape* es del **tamaño de la ventana** y activaremos cada capa usando un *relu*.

[9] Compilamos el modelo como antes con una función de pérdida de error cuadrado medio y optimizador de gradiente estocástico.

[10] Finalmente, encajaremos el modelo en más de 100 épocas. El parámetro *verbose* indica la cantidad de información que se mostrará durante el entrenamiento del modelo. Este parámetro puede tomar valores enteros del 0 al 2 (0 - no se muestra ningún mensaje durante el entrenamiento, 1 - se muestra una barra de progreso durante el entrenamiento, indicando el número de épocas completadas y la métrica de entrenamiento y validación (si se especifica), 2 - se muestra una línea por cada época completada, indicando la métrica de entrenamiento y validación (si se especifica)).

El gráfico se ve así con los valores reales en azul y los predichos en naranja.



Es bastante bueno todavía. Y cuando calculamos el error absoluto medio, obtenemos un valor menor que antes, así que es un paso en la dirección correcta.

```
1 tf.keras.metrics.mean_absolute_error(x_valid, results).numpy()
2 # 4.9833784
```

### 3.5. Tasa de aprendizaje óptima

¿No sería bueno si pudiéramos elegir la tasa de aprendizaje óptima en lugar de la que elegimos? Podríamos aprender de manera más eficiente y construir un modelo mejor. Ahora veamos una técnica para que utiliza *callbacks*.

```
1 dataset = windowed_dataset(x_train, window_size, batch_size,
2                             shuffle_buffer_size)
3
4 model_tune = tf.keras.models.Sequential([
5     tf.keras.layers.Dense(10, activation="relu", input_shape=[window_size]),
6     tf.keras.layers.Dense(10, activation="relu"),
7     tf.keras.layers.Dense(1)
8 ])
9 lr_schedule = tf.keras.callbacks.LearningRateScheduler(lambda epoch
10 : 1e-8 * 10 ** (epoch/20))
11 model.compile(loss="mse", optimizer=tf.keras.optimizers.SGD(
12     learning_rate=1e-6, momentum=0.9))
13 history = model.fit(dataset, epochs=100, callbacks=[lr_schedule])
```

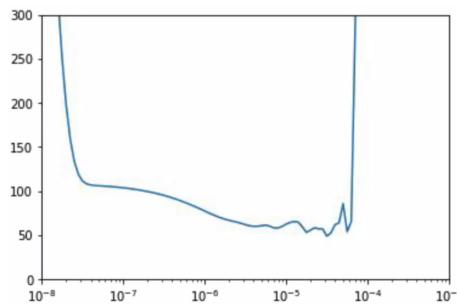
[9] Esto será llamado en el callback al final de cada época. Lo que hará es **cambiar las tasas de aprendizaje** a un valor basado en el número de la época. Así que en la época 1, es  $10^{-8} * 10^{1/20}$ . Y para cuando lleguemos a la época 100, será  $10^{-8} * 10^5$  ( $5=100/20$ ).

[12] Esto sucederá en cada callback porque lo configuramos en el parámetro callbacks del atuendo modelado.

Después de entrenar con esto, podemos trazar la última por época contra la tasa de aprendizaje por época usando este código.

```
1 lrs = 1e-8 * (10 ** (np.arange(100) / 20))
2 plt.semilogx(lrs, history.history["loss"])
3 plt.axis([1e-8, 1e-3, 0, 300])
```

Veremos un gráfico como este.

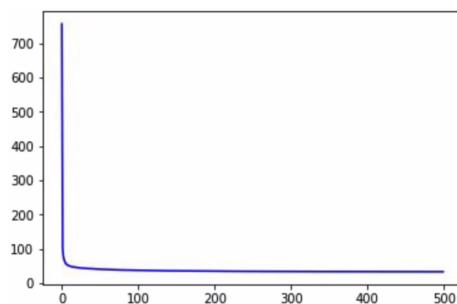


El eje  $y$  nos muestra la pérdida para esa época y el eje  $x$  nos muestra la tasa de aprendizaje. Entonces podemos tratar de elegir el punto más bajo de la curva donde todavía es relativamente estable así, y eso es justo alrededor de  $7 \times 10^{-6}$ .

Así que vamos a establecer que sea nuestra tasa de aprendizaje y luego volveremos a entrenar.

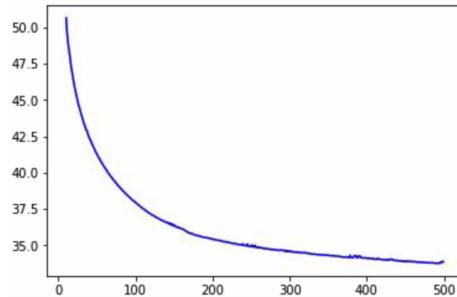
```
1 model.compile(loss="mse", optimizer=tf.keras.optimizers.SGD(
2     learning_rate=7e-6, momentum=0.9))
3 history = model.fit(dataset, epochs=100)
```

Vamos a comprobar los resultados después del entrenamiento para 500 épocas.

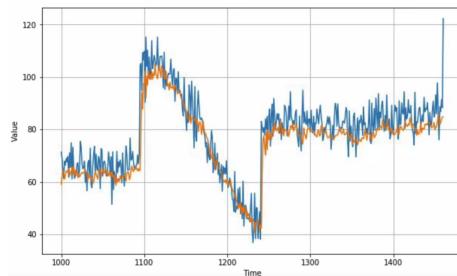


Parece que probablemente estamos perdiendo nuestro tiempo entrenando más allá de tal vez solo 10 épocas, pero está algo sesgado por el hecho de que **las pérdidas anteriores eran muy altas**. Si los recortamos y trazamos la pérdida para épocas después del número 10 con código como esto, entonces el gráfico nos contará una historia diferente.

```
1 loss = history.history['loss']
2 epochs = range(10, len(loss))
3 plot_loss = loss[10:]
4 print(plot_loss)
5
6 plt.plot(epochs, plot_loss, 'b', label='Training Loss')
7 plt.show()
```



Podemos ver que la pérdida continuaba disminuyendo incluso después de 500 épocas. Y eso demuestra que nuestra red está aprendiendo muy bien. Y los resultados de las predicciones superpuestas contra los originales se ven así.



Y el error absoluto medio en los resultados es significativamente menor que antes.

```
1 tf.keras.metrics.mean_absolute_error(x_valid, results).numpy()
2 # 4.4847784
```

```
1 def windowed_dataset(series, window_size, batch_size,
2     shuffle_buffer):
3     # Generate a TF Dataset from the series values
4     dataset = tf.data.Dataset.from_tensor_slices(series)
5
6     # Window the data but only take those with the specified batch
7     dataset = dataset.window(window_size + 1, shift=1,
8         drop_remainder=True)
9
10    # Flatten the windows by putting its elements in a single batch
11    dataset = dataset.flat_map(lambda window: window.batch(
12        window_size + 1))
13
14    # Create tuples with features and labels
15    dataset = dataset.map(lambda window: (window[:-1], window[-1]))
16
17    # Shuffle the windows
18    dataset = dataset.shuffle(shuffle_buffer)
19
20    # Create batches of windows
21    dataset = dataset.batch(batch_size).prefetch(1)
22
23    return dataset
```

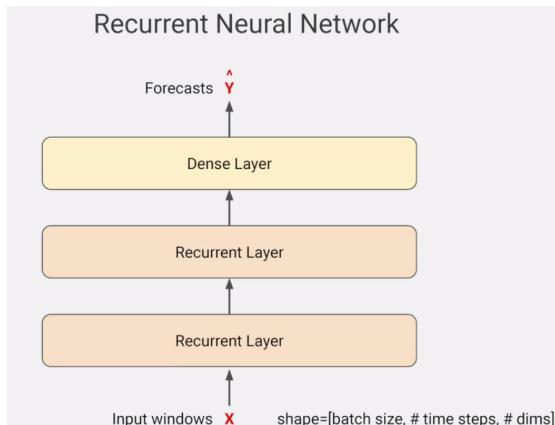
## 4. NN recurrentes para series temporales

En este apartado, vamos a usar redes neuronales recurrentes y LSTMs para pronosticar series temporales. Veremos los enfoques sin estado y con estado, capacitación en ventanas de datos.

Las series temporales son datos temporales, por lo que parece que deberíamos aplicar un **modelo de secuencia**, como un RNN o un LSTM.

Las **capas Lambda** nos permiten escribir una pieza arbitraria de código como una capa en la red neuronal. Así que en lugar de, por ejemplo, escalar sus datos con un paso explícito de pre-procesamiento y luego alimentar esos datos a la red neuronal, en su lugar puede tener una capa Lambda. Básicamente una función Lambda, una función sin nombre, pero implementada como una capa en la red neuronal que vuelve a enviar los datos, los escala. Ahora que el paso de pre-procesamiento ya no es un paso separado y distinto, es parte de la red neuronal.

Una **RNN** es una red neuronal que contiene capas recurrentes. Estos están diseñados para procesar secuencialmente secuencias de entradas. Son bastante flexibles, capaces de procesar todo tipo de secuencias y se pueden utilizar para predecir texto. Aquí los usaremos para procesar la serie temporal.

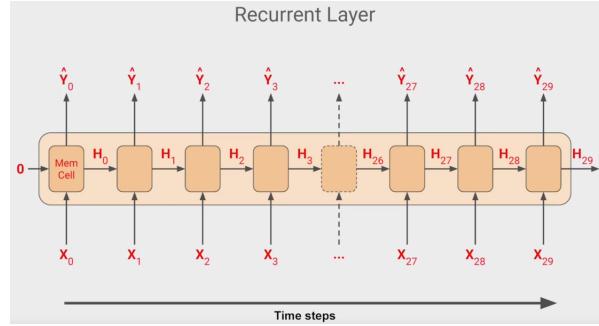


Este ejemplo, creará un RNN que contiene dos capas recurrentes y una capa densa final, que servirá como salida. Con un RNN, puedes alimentarlo en lotes de secuencias, y emitirá un lote de pronósticos.

Una diferencia respecto a las DNN será que la forma de entrada completa cuando usando RNN es tridimensional. La primera dimensión será el **tamaño del lote**, la segunda será las marcas de tiempo (**timestamps**), y la tercera es la **dimensionalidad de las entradas** en cada paso de tiempo.

Los modelos que hemos estado usando hasta ahora tenían entradas bidimensionales, la dimensión del lote y las características de entrada.

Antes de seguir avanzando, vamos a profundizar en las RNN.



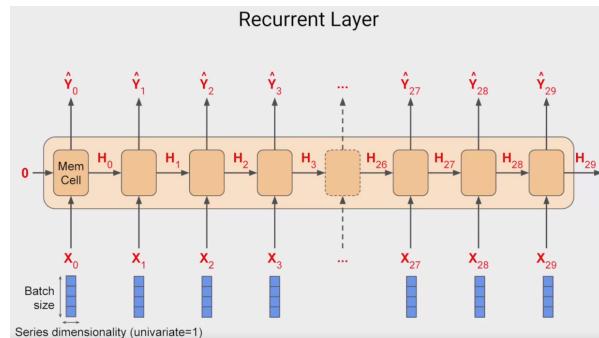
Lo que parece que hay muchas celdas, en realidad solo hay una, y se usa repetidamente para calcular las salidas. En cada paso de tiempo, la celda de memoria toma el valor de entrada para ese paso. Luego calcula la salida para ese paso, en este caso  $\hat{Y}_0$ , y un vector  $H_0$  de estado que se alimenta en el siguiente paso.  $H_0$  se introduce en la celda con  $X_1$  para producir  $\hat{Y}_1$  y  $H_1$ , que luego se introduce en la celda en el siguiente paso con  $X_2$  para producir  $\hat{Y}_2$  y  $H_2$ . Estos pasos continuarán hasta que lleguemos al final de nuestra dimensión de entrada

Esto es lo que le da a este tipo de arquitectura el nombre de una red neuronal recurrente, porque los valores se repiten debido a la salida de la celda, un paso que se devuelve en sí mismo en el siguiente paso de tiempo.

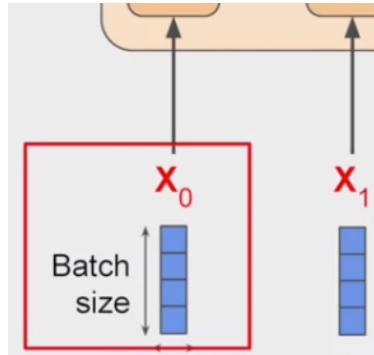
Esto es realmente útil para determinar estados. La ubicación de una palabra en una oración puede determinarla semántica. Del mismo modo, para series numéricas, cosas como números más cercanos en la serie podrían tener un impacto mayor que aquellos más lejos de nuestro valor objetivo.

#### 4.1. Forma de las entradas de una RNN

Hemos mencionado la forma de los datos y los lotes en los que se dividen los datos. Es importante echar un vistazo a eso, y vamos a profundizar en eso a continuación.

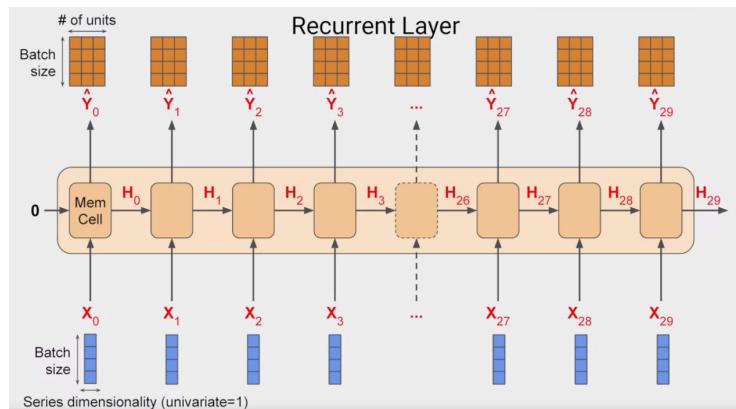


Las entradas son tridimensionales. Así que, por ejemplo, si tenemos un tamaño de ventana de 30 marcas de tiempo y las estamos agrupando en tamaños de 4, la forma será  $4 * 30$  veces 1, y cada marca de tiempo, la entrada de la celda de memoria será una matriz de  $4 \times 1$ , como esta.



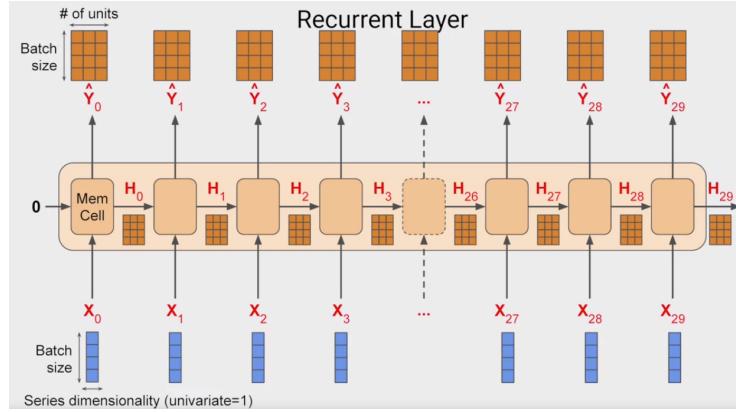
La celda también tomará la entrada de la matriz de estado del paso anterior. Pero, por supuesto, en este caso, en el primer paso, esto será cero.

Para los siguientes, será la salida de la celda de memoria. Pero aparte del vector de estado, la celda, por supuesto, producirá un valor  $Y$ , que podemos ver aquí.



Si la celda de memoria está compuesta por 3 neuronas, entonces la matriz de salida será de  $4 \times 3$  porque el tamaño del lote que llegó fue cuatro y el número de neuronas es tres. Así que la salida completa de la capa es tridimensional, en este caso,  $4 \times 30 \times 3$ . Con 4 siendo el **tamaño del lote** (Batch size), 3 siendo el **número de unidades** (# of units), y 30 siendo el número total de **pasos** ( $Y_0 - Y_{29}$ ).

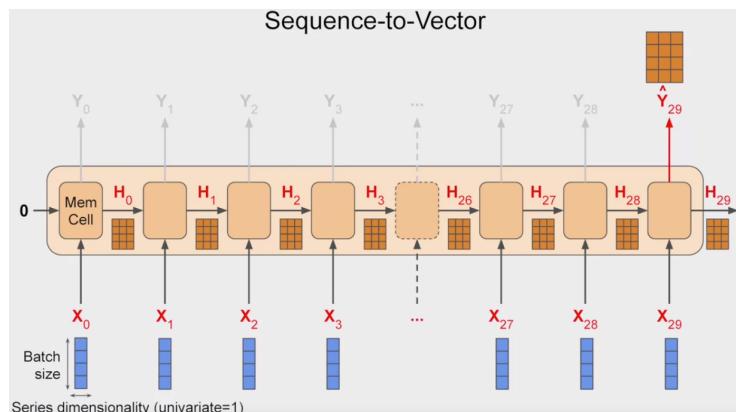
En una RNN simple, la salida de estado  $H$  es solo una copia de la matriz de salida  $Y$ .



Así que, por ejemplo,  $H_0$  es una copia de  $Y_0$ ,  $H_1$  es una copia de  $Y_1$ , y así sucesivamente.

Así que en cada marca de tiempo, la celda de memoria obtiene tanto la entrada actual como la salida anterior.

Ahora, en algunos casos, es posible que desee ingresar una secuencia, pero no desea generar y solo desea obtener un único vector para cada instancia en el lote.



Esto se suele llamar una **secuencia al vector RNN**. Pero en realidad, todo lo que haces es ignorar todas las salidas, excepto la última.

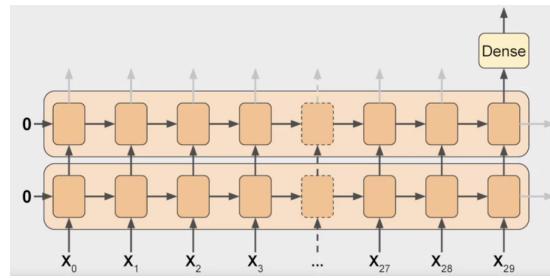
## 4.2. Salida de la secuencia

Considera esta RNN.

```

1 model = tf.keras.models.Sequential([
2     tf.keras.layers.SimpleRNN(20, return_sequences=True,
3         input_shape=[None,1]),
4     tf.keras.layers.SimpleRNN(20),
5     tf.keras.layers.Dense(1)
])

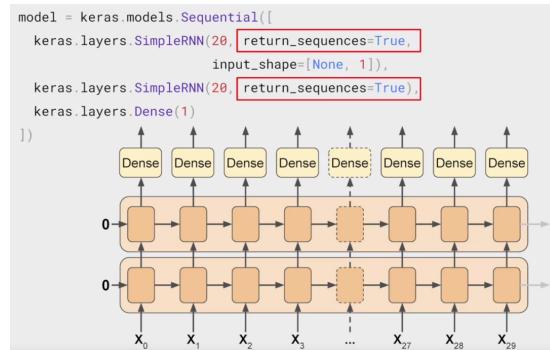
```



Estas tienen dos capas recuperadas, y la primera tiene una configuración `return_sequences=True`. Esto producirá una secuencia que alimenta a la siguiente capa. La siguiente capa no tiene `return_sequence` que se establece en `True`, por lo que solo dará salida al paso final.

Observe el `input_shape`, está establecida en `[None, 1]`. TensorFlow asume que la primera dimensión es el tamaño del lote, y que puede tener cualquier tamaño en absoluto, por lo que no es necesario definirla. Entonces la dimensión es el número de marcas de tiempo, que podemos establecer en `None`, lo que significa que el RNN puede manejar secuencias de cualquier longitud. La última dimensión es sólo `1` porque estamos utilizando **series temporales univariantes**.

Si establecemos `return_sequence` en `true` en todas las capas recurrentes, entonces todas las secuencias de salida y la capa densa obtendrá una secuencia como sus entradas.



Keras maneja esto usando la misma capa densa de forma independiente en cada sello de tiempo. Puede parecer varios aquí, pero es el mismo que se está reutilizando en cada paso de tiempo. Esto nos da lo que se llama una **secuencia a secuencia RNN**. Se alimenta un lote de secuencias y devuelve un lote de secuencias de la misma longitud. La dimensionalidad puede no coincidir siempre. Depende del número de unidades en la venta de memoria.

Así que volvamos ahora a un RNN de dos capas que tiene el segundo sin secuencias de retorno. Esto nos dará una salida a un solo denso.

### 4.3. Capas Lambda

Vamos a agregar un par de capas nuevas a esto, capas que usan el tipo *Lambda*.

Este tipo de capa es una que nos permite realizar operaciones arbitrarias para ampliar la funcionalidad de los kera de TensorFlow, y podemos hacerlo dentro de la propia definición del modelo.

```

1 model= tf.keras.models.Sequential([
2     tf.keras.layers.Lambda(lambda x: tf.expand_dims(x, axis=1),
3         input_shape=[None]),
4     tf.keras.layers.SimpleRNN(20, return_sequences=True),
5     tf.keras.layers.SimpleRNN(20),
6     tf.keras.layers.Dense(1),
7     tf.keras.layers.Lambda(lambda x: x * 100.0)
])

```

- [2] La primera capa de Lambda será usada para ayudarnos con nuestra dimensionalidad. Si recuerda cuando escribimos la función auxiliar del dataset de ventana, devolvió lotes bidimensionales de Windows en los datos, con la primera siendo el tamaño del lote y la segunda el número de marcas de tiempo. Pero una RNN espera tres dimensiones; *tamaño del lote*, el *número de marcas de tiempo*, y la *dimensionalidad de la serie*. Con la capa Lambda, podemos solucionar esto sin reescribir nuestra función auxiliar de dataset de ventana. Usando el Lambda, simplemente expandimos la matriz por una dimensión. Al establecer la forma de entrada en none, estamos diciendo que el modelo puede tomar secuencias de cualquier longitud.
- [3] Ampliamos las salidas en 100 para ayudar a entrenar. La función de activación predeterminada en las capas RNN es  $\tan(H)$  que es la **activación tangente hiperbólica**. Esto genera valores entre -1 y 1. Dado que los valores de las series de tiempo están en ese orden generalmente en los 10s (40s, 50s...), entonces escalando las salidas al mismo estadio pueden ayudarnos con el aprendizaje. Podemos hacer eso en una capa de Lambda también, simplemente lo multiplicamos por 100.

Así que ahora echemos un vistazo a lo que se necesita para construir el RNN completo para que podamos empezar a hacer algunas predicciones con él.

#### 4.4. Ajustar LR dinámicamente

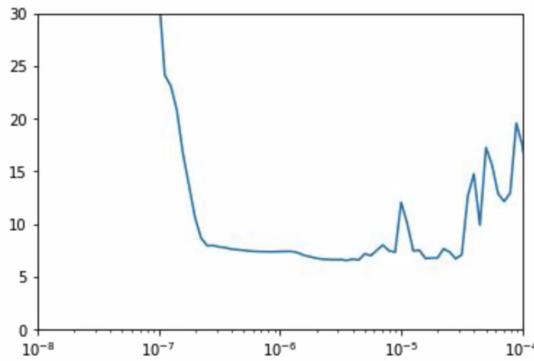
Optimizar la tasa de aprendizaje del optimizador puede ser bastante rápido para entrenar y podemos ahorrar mucho tiempo en nuestro ajuste de hiperparámetros.

```
1 train_set = windowed_dataset(x_train, window_size, batch_size,
2     shuffle_buffer_size)
3
4 model = tf.keras.models.Sequential([
5     tf.keras.layers.Lambda(lambda x: tf.expand_dims(x, axis=-1),
6         input_shape=[window_size]),
7     tf.keras.layers.SimpleRNN(40, return_sequences=True),
8     tf.keras.layers.SimpleRNN(40),
9     tf.keras.layers.Dense(1),
10    tf.keras.layers.Lambda(lambda x: x * 100.0)
11])
12 lr_schedule = tf.keras.callbacks.LearningRateScheduler(lambda epoch
13 : 1e8 * 10 **(epoch/20))
14
15 model.compile(loss=tf.keras.losses.Huber(), optimizer=tf.keras.
    optimizers.SGD(learning_rate=1e-8, momentum=0.9), metrics=["mae"])
16
17 history = model.fit(train_set, epochs=100, callbacks=[lr_schedule])
```

[11] Cada época cambia la tasa de aprendizaje un poco.

[13] *Huber()* es una función de pérdida que es menos sensible a los valores atípicos y como estos datos pueden ser un poco ruidosos, vale la pena darle una oportunidad.

Si ejecutamos esto por 100 épocas y medimos la pérdida en cada época, veremos que la tasa óptima de aprendizaje para el SGD desciende entre aproximadamente  $10^{-5}$  y  $10^{-6}$ .



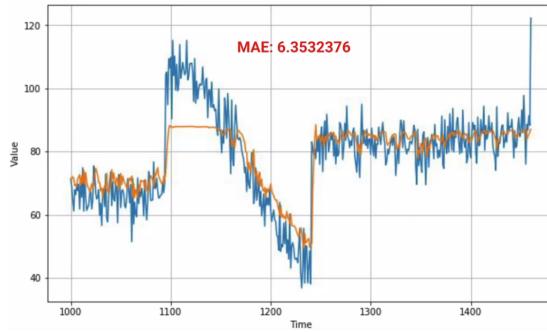
Así que ahora, establecemos los modelos compilados con una tasa de aprendizaje de  $5 * 10^{-6}$  y el optimizador de descenso de gradiente estocástico.

```

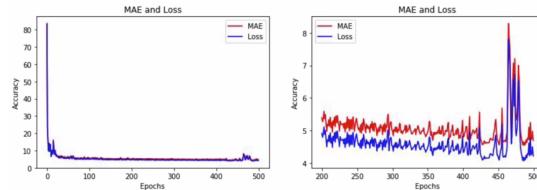
1 train_set = windowed_dataset(x_train, window_size, batch_size,
2     shuffle_buffer_size)
3
4 model = tf.keras.models.Sequential([
5     tf.keras.layers.Lambda(lambda x: tf.expand_dims(x, axis=-1),
6         input_shape=[window_size]),
7     tf.keras.layers.SimpleRNN(40, return_sequences=True),
8     tf.keras.layers.SimpleRNN(40),
9     tf.keras.layers.Dense(1),
10    tf.keras.layers.Lambda(lambda x: x * 100.0)
11])
12
13 lr_schedule = tf.keras.callbacks.LearningRateScheduler(lambda epoch
14     : 1e-8 * 10 **(epoch/20))
15
16 model.compile(loss=tf.keras.losses.Huber(), optimizer=tf.keras.
17     optimizers.SGD(learning_rate=5e-6, momentum=0.9), metrics=["mae"])
18
19 history = model.fit(train_set, epochs=500)

```

Tras entrenar por 500 épocas, obtendremos este gráfico, con un **MAE** en el conjunto de validación de aproximadamente 6.35.



Aquí está la pérdida y el MAE durante el entrenamiento con el gráfico a la derecha con zoom en las últimas épocas.



Teniendo en cuenta esto, probablemente valga la pena solo entrenamiento para alrededor de 400 épocas.

## 4.5. Función de pérdida de Huber

La función de pérdida de Huber es una función de pérdida utilizada en problemas de regresión que es menos sensible a valores atípicos (outliers) en los datos que la función de pérdida cuadrática tradicional.

La función de pérdida de Huber es una combinación de la función de pérdida cuadrática y la función de pérdida absoluta. Se define como:

$$L_\delta(y, f(x)) = \begin{cases} \frac{1}{2}(y - f(x))^2 & \text{for } |y - f(x)| \leq \delta, \\ \delta \cdot (|y - f(x)| - \frac{1}{2}\delta), & \text{otherwise.} \end{cases}$$

donde  $y$  es el valor real de la variable dependiente,  $f(x)$  es el valor predicho por el modelo para la variable dependiente, y  $\delta$  es un parámetro que controla la sensibilidad de la función de pérdida a los valores atípicos.

Cuando  $|y - f(x)|$  es menor o igual que delta, la función de pérdida de Huber se comporta como la función de **pérdida cuadrática**. Cuando  $|y - f(x)|$  es mayor que delta, la función de pérdida de Huber se comporta como la función de **pérdida absoluta**.

La función de pérdida de Huber es útil cuando se desea **minimizar el impacto de los valores atípicos** en el ajuste de un modelo de regresión. Al ajustar un modelo utilizando esta función de pérdida, **se penalizan menos los errores pequeños** y se penalizan **más los errores grandes**, pero solo **hasta cierto punto**. Esto significa que la función de pérdida de Huber proporciona un **equilibrio entre la robustez a los valores atípicos y la precisión en la predicción** de los valores de la variable dependiente.

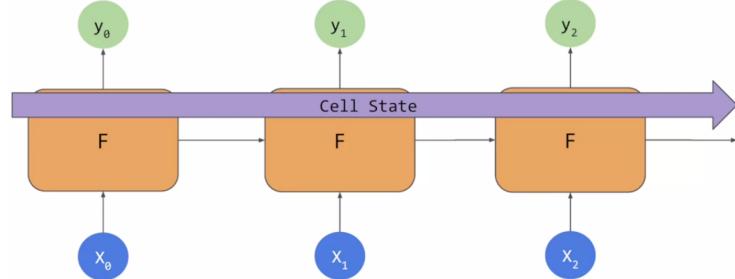
## 4.6. LSTM

Antes hemos usado una RNN para predecir valores en tu secuencia. Los resultados fueron buenos, pero necesitaron un poco de mejora. Un mejor enfoque podría ser usar **LSTM en lugar de RNN** para ver el impacto.

La RNN tenía celdas que tomaba parches como entradas o  $X$ , y calculaban una salida  $Y$ , así como el vector de estado, que alimentaba a la celda junto con la siguiente  $X$  que luego resultó en la  $Y$ , y el vector de estado y así sucesivamente.

El impacto de esto es que mientras que **el estado es un factor en los cálculos posteriores**, sus impactos **pueden disminuir considerablemente con las marcas de tiempo**.

LSTM añade la **celda estado**, que mantiene un estado a lo largo la vida del entrenamiento para que el estado pase de celda a celda, marca de tiempo a marca de tiempo, y se puede mantener mejor.



Esto significa que los datos de anteriores en la ventana pueden tener un mayor impacto en la proyección general que en el caso de RNN.

El estado también puede ser **bidireccional** para que el estado se mueva hacia adelante y hacia atrás. En el caso de los textos, esto fue realmente poderoso. Dentro de la predicción de secuencias numéricas, **puede o no ser**, y será interesante experimentar.

#### 4.7. Construir RNN

```

1 model_tune = tf.keras.models.Sequential([
2     tf.keras.layers.Lambda(lambda x: tf.expand_dims(x, axis=-1),
3                           input_shape=[window_size]),
4     tf.keras.layers.SimpleRNN(40, return_sequences=True),
5     tf.keras.layers.SimpleRNN(40),
6     tf.keras.layers.Dense(1),
7     tf.keras.layers.Lambda(lambda x: x * 100.0)
8 ])
9 model_tune.summary()

```

Model: "sequential"		
Layer (type)	Output Shape	Param #
lambda (Lambda)	(None, 20, 1)	0
simple_rnn (SimpleRNN)	(None, 20, 40)	1680
simple_rnn_1 (SimpleRNN)	(None, 40)	3240
dense (Dense)	(None, 1)	41
lambda_1 (Lambda)	(None, 1)	0

Total params: 4,961  
Trainable params: 4,961  
Non-trainable params: 0

```

1 lr_schedule = tf.keras.callbacks.LearningRateScheduler(lambda epoch
2   : 1e-8 * 10 **(epoch/20))
3 model_tune.compile(loss=tf.keras.losses.Huber(), optimizer=tf.keras
4   .optimizers.SGD(momentum=0.9))
5 history = model_tune.fit(dataset, epochs=100, callbacks=[lr_schedule])

```

```

1 model_tune.compile(loss=tf.keras.losses.Huber(), optimizer=tf.keras
2   .optimizers.SGD(learning_rate=1e-6, momentum=0.9), metrics=["mae"])
3 history = model_tune.fit(dataset, epochs=100, callbacks=[lr_schedule])

```

```

1 forecast = model_tune.predict(dataset)

```

#### 4.7.1. LSTM bidireccional

Reemplaza capas '*SimpleRNN*' por '*Bidirectional*'.

```

1 tf.keras.backend.clear_session()
2
3 model_tune = tf.keras.models.Sequential([
4   tf.keras.layers.Lambda(lambda x: tf.expand_dims(x, axis=-1),
5     input_shape=[window_size]),
6   tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(32,
7     return_sequences=True)),
8   tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(32)),
9   tf.keras.layers.Dense(1),
10  tf.keras.layers.Lambda(lambda x: x * 100.0)
11 ])
12 model_tune.summary()

```

[1] *clear\_session()* borra cualquier variable interna. Eso hace que sea fácil para nosotros experimentar sin modelos que impacten versiones posteriores de sí mismos.

[5] y [6] Después de la capa Lambda que expande las dimensiones para nosotros hemos agregado dos capas *LSTM bidireccional* con 32 celdas. Además, en la primera es necesario establecer *return\_sequences=True* para que funcione.

Si editamos nuestra fuente para agregar una tercera capa LSTM como esta, agregando la capa y teniendo las secuencias de retorno de la segunda capa es true podemos luego entrenarla y ejecutarla.

```
1 tf.keras.backend.clear_session()
2
3 model_tune = tf.keras.models.Sequential([
4     tf.keras.layers.Lambda(lambda x: tf.expand_dims(x, axis=-1),
5         input_shape=[window_size]),
6     tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(32,
7         return_sequences=True)),
8     tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(32,
9         return_sequences=True)),
10    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(32)),
11    tf.keras.layers.Dense(1),
12    tf.keras.layers.Lambda(lambda x: x * 100.0)
13])
14
15 model_tune.summary()
```

## 5. Datos reales de series temporales

En este último apartado, agregaremos convoluciones a la mezcla y pondremos todo lo que hemos trabajado en este documento para comenzar a pronosticar algunos datos del mundo real, y eso es mediciones de la actividad de manchas solares en los últimos 250 años.

### 5.1. Convoluciones

Vamos a combinar convoluciones con LSTM para obtener un modelo más apropiado. Luego aplicaremos eso a los datos del mundo real en lugar de este conjunto de datos sintéticos en el que hemos estado trabajando desde el comienzo de este curso.

```
1 model = tf.keras.models.Sequential([
2     tf.keras.layers.Conv1D(filters=32, kernel_size=5, strides=1,
3         padding="casual", activation="relu", input_shape=[None,1]),
4     tf.keras.layers.LSTM(32, return_sequences=True),
5     tf.keras.layers.LSTM(32),
6     tf.keras.layers.Dense(1),
7     tf.keras.layers.Lambda(lambda x: x * 200)
8 ])
9 model.compile(loss=tf.keras.losses.Huber(), optimizer=tf.keras.
10 optimizers.SGD(learning_rate=1e-5, momentum=0.9), metrics=["
    mae"])
model.fit(dataset, epochs=500)
```

[2] Esta es una capa de convolución unidimensional (*Conv1D*) de una red neuronal convolucional (CNN) y los parámetros se explican a continuación:

- ***filters***: número de filtros de convolución (o características) que se deben aprender en la capa. En este caso, se establece en 32, lo que significa que la capa aprenderá 32 características diferentes.
- ***kernel\_size***: tamaño del kernel de convolución.
- ***strides***: número de pasos que se mueve el kernel durante la convolución. En este caso, se establece en 1, lo que significa que el kernel se mueve 1 paso a la vez.
- ***padding***: tipo de relleno que se debe aplicar a la entrada antes de la convolución. En este caso, se establece en “casual”, lo que significa que se aplica un relleno causal a la entrada antes de la convolución. El relleno causal **garantiza que la salida en cada paso de tiempo solo dependa de las entradas anteriores y no de las entradas futuras**.
- ***activation***: función de activación que se debe aplicar a la salida de la convolución. En este caso, se establece en “relu”, lo que significa que

se utiliza la función de activación ReLU (unidad lineal rectificada) en la salida.

- ***input\_shape***: forma de la entrada a la capa. En este caso, se establece en [None, 1], lo que significa que la entrada puede tener cualquier longitud (None) pero debe tener una dimensión (1). La dimensión 1 se refiere a la señal unidimensional que se está procesando.

## 5.2. LSTM Bidireccional

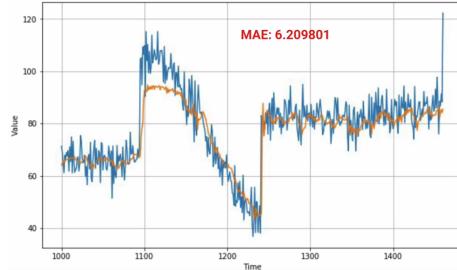
Una nota importante es que mientras nos deshacemos de la capa Lambda que reformó la entrada para que trabajemos con los LSTM, ahora, estamos especificando una forma de entrada en la curva 1D con *input\_shape=[None,1]*. Esto requiere que actualicemos la función helper *windowed\_dataset()* con la que hemos estado trabajando todo el tiempo. Simplemente usaremos *tf.expand\_dims()* en la función auxiliar para expandir las dimensiones de la serie antes de procesarla.

Un método para mejorar el desempeño del modelo podría ser hacer tus **LSTM Bidireccional**. Esto implicaría entrenar dos LSTM, una que procesa la secuencia de forma *ascendente* y otra que la procesa de forma *descendente*, y luego concatenar las salidas de ambas capas. Esto permitiría al modelo capturar mejor las dependencias a largo plazo en la secuencia y mejorar su capacidad para predecir valores futuros con mayor precisión. Sin embargo, esto también aumentaría significativamente la complejidad del modelo y el tiempo de entrenamiento requerido.

```

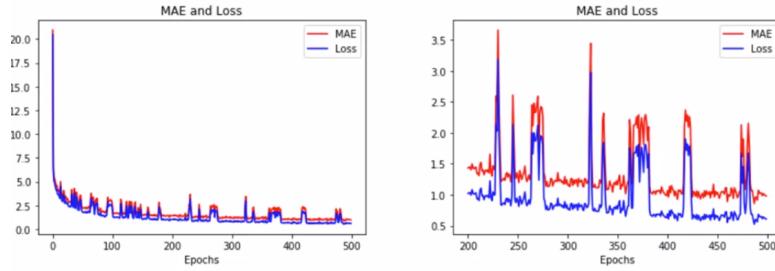
1 model = tf.keras.models.Sequential([
2     tf.keras.layers.Conv1D(filters=32, kernel_size=5, strides=1,
3                           padding="casual", activation="relu", input_shape=[None,1]),
4     tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(32),
5                                   return_sequences=True)),
6     tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(32)),
7     tf.keras.layers.Dense(1),
8     tf.keras.layers.Lambda(lambda x : x * 200)
9 ])

```



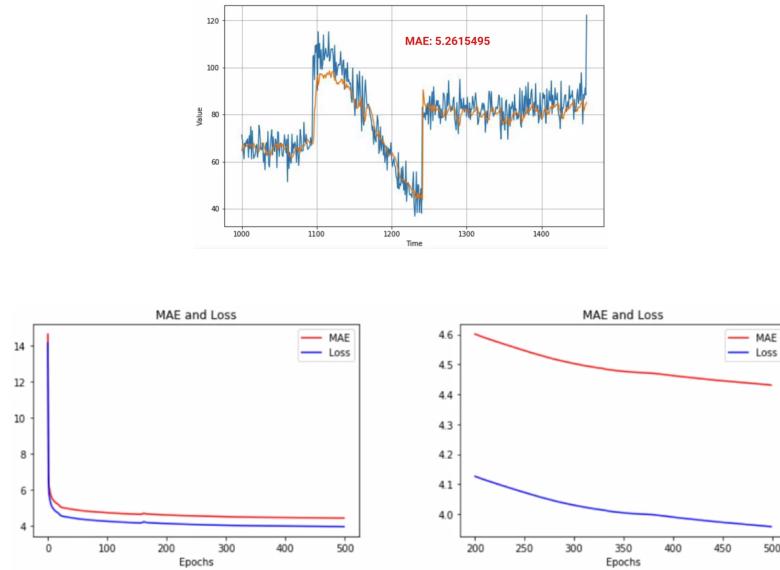
Al entrenar, esto se ve muy bien dando muy baja pérdida en los valores de MAE a veces incluso menos de uno. Pero desafortunadamente es demasiado

adecuado cuando trazamos las predicciones contra el conjunto de validación, no vemos mucha mejora y de hecho nuestro MAE ha caído. Así que sigue siendo un paso en la dirección correcta y considere una arquitectura como esta a medida que avanza, pero tal vez necesite modificar algunos de los parámetros para evitar el overfitting.



Uno de los problemas es visualizar cuando trazamos la pérdida contra el MAE. Hay mucho ruido e inestabilidad. Una causa común para picos pequeños como ese es un **tamaño de lote pequeño que introduce más ruido aleatorio**.

Así que, por ejemplo, experimentando con diferentes tamaños de lote tanto más grande y más pequeño que el original 32, y cuando probamos 16 se puede ver el impacto en el conjunto de validación y en la pérdida de entrenamiento y datos MAE.



Así que combinando CNN y LSTM hemos sido capaces de construir nuestro mejor modelo hasta ahora, a pesar de algunos bordes rugosos que podrían ser refinados.

### 5.3. Caida exponencial de LR

```
1 lr_schedule = tf.keras.optimizers.schedules.ExponentialDecay(  
2     initial_learning_rate=1e-7,  
3     decay_steps=400,  
4     decay_rate=0.96,  
5     staircase=True)
```

Este código está definiendo un plan de tasa de aprendizaje **exponencialmente decreciente** con una tasa de aprendizaje inicial  $1e - 7$ , una tasa de decaimiento de 0.96, y la tasa de aprendizaje se escalonará después de cada 400 pasos de entrenamiento (si staircase es True). Esta programación de tasas de aprendizaje se puede utilizar posteriormente en un optimizador como tf.keras.optimizers.SGD para entrenar un modelo de aprendizaje profundo.

## 6. Datos importantes

- Uno de los aspectos más importantes en el rendimiento del modelo es el *batch\_size* del *windowed\_dataset()*.