Refactorización de Redes Neuronales

Alcance:

- Conocer la diferencia entre Weight Decay y regularización L2
- Entender los problemas fundamentales que presentan las redes feed forward clasicas en el tratamiento de problemas temporales.
- Entender los fundamentos de las redes recurrentes
- Implementar una red recurrente.

Precis: Weight Decay vs Regularización L2

En el tema anterior mencionamos distintos métodos de regularización, dentro de los cuales vimos que para redes neuronales también podemos utilizar la conocida regularización **L2**. Sin embargo, en la literatura muchas veces se hace referencia a un tipo de regularización conocido como *'Weight Decay'*, el cual se menciona como un sinónimo de regularización **L2** (Goodfellow et al).
Weight decay se refiere a la práctica de incorporar un parámetro de escala en la regla de actualización de los pesos **[4]**. Para gradiente decendente la expresión de actualización de pesos con weight decay queda de la siguiente forma:

$$W_{i+1} := \beta \cdot w_i - \alpha \cdot \frac{\partial J}{\partial w}$$

A esto se le conoce como *weight decay*, dado que

$$\beta$$

es fijado en un valor cercano

$$\beta \in [0, 1)$$

De esta forma, en las sucesivas iteraciones del modelo,

$$\beta$$

se ve elevado en consideración al número de iteraciones realizadas. Por cada actualización del término, éste se ve escalado por la siguiente forma funcional

$$\beta^n$$

Dado los límites definidos en

$$\beta$$

si ignoramos el gradiente, los pesos decaerán de forma exponencial en la medida que aumentamos la cantidad de épocas de entrenamiento.

Por otro lado, la regularización **L2** se refiere a incorporar un factor de penalización en la función de costo **J**:

$$\begin{aligned} J_{l2} &= J + \lambda \cdot w^2 \\ \frac{\partial J_{l2}}{\partial w} &= \frac{\partial J}{\partial w} + 2\lambda w \end{aligned}$$

Para gradiente descendente la regla de pesos con regularización **L2** es:

$$\begin{aligned} W_{i+1} &:= w - \alpha \cdot \frac{\partial J_{l2}}{\partial w} \\ &:= w - \alpha \cdot (\frac{\partial J}{\partial w} + 2\lambda w) \\ &:= w(1 - 2\alpha\lambda) - \alpha \cdot \frac{\partial J}{\partial w} \end{aligned}$$

Por tanto, weight decay y regularización **L2** son entonces matemáticamente equivalentes bajo la elección de un

$$\alpha$$

y

$$\lambda$$

convenientes.

Resumiendo:

- Weight decay hace que los pesos sean cada vez más pequeños durante el entrenamiento.
- Regularización L2 previene que los pesos sean demasiado grandes.

Ambas aproximaciones buscan lo mismo: Evitar pesos demasiado grandes.

En este punto, daríamos por terminada la clase y nos iriamos a nuestras casas con la agradable sensación de haber demostrado que weight decay y regularizacion L2 son lo mismo, sin embargo, estamos olvidando un punto importante: **Nuestros calculos son válidos solo para gradiente descendente o similares**.

En efecto, si utilizamos otro optimizador, como por ejemplo Adam, este comportamiento deja de ser trivial. Esto fue propuesto hace poco, al parecer los optimizadores con un learning rate adaptativo se benefician más de weight decay que de regularización L2 (aún cuando matemáticamente son idénticos)[5]. De hecho, el paper citado en **[5]** al parecer generó bastante debate entre los académicos que lo evaluaron (aquellos que quieran profundizar más sobre el debate, pueden consultar el siguiente link).

Esto nos muestra que el subtópico de las redes neuronales aún está sujeto a debate y quedan muchas cosas por estudiar y cuestionarse, podemos estar seguros que en algunos años más habrán modelos y heurísticas muchisimo más efectivas que las actuales.

Sobre la decisión de si implementar regularización L2 o weight decay o ambas, lamentablemente esa tarea será de quién implemente el modelo de acuerdo a su criterio e intuición.

Contexto y Memoria: Limitantes de las redes neuronales feed forward

En el conocido programa de televisión *'Pasapalabra'*, uno de los desafíos consiste en memorizar palabras (usualmente relacionadas en un cierto contexto). En la medida que los jugadores son capaces de recitar la secuencia correctamente, se va adicionando nuevas palabras a la lista a memorizar. Este proceso es intuitivo para los seres humanos, dado que somos capaces de **contextualizar** la información y **memorizar** la secuencia.

Resulta que una red neuronal clásica como la *feed forward* no es capaz de contextualizar ni memorizar la información. Todos aquellos inputs procesados sólo se mantienen en la capa oculta, olvidando el impulso ejecutado cuando éste pasa a una siguiente capa. Para solucionar este problema, una **Red Neuronal Recurrente** adopta el principio de procesar secuencias de manera iterativa y preserva aquella información relevante en un estado interno.

La limitante en el flujo feedforward se puede esquematizar de la siguiente manera:

1. Una capa oculta dentro de la arquitectura recibirá un input en un momento

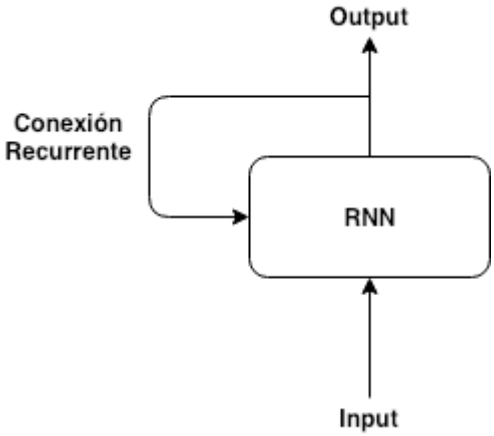
   $$t \in T$$

   donde **T** representa la cantidad de capas existentes en la arquitectura.
2. Este input de datos será procesado mediante la función de activación, entregando un peso que posteriormente será captado por la capa oculta **t+1**.
3. En cada capa, la información dentro de ésta se olvida posteriormente al envío del impulso a la siguiente capa.
4. Esto genera que la arquitectura neuronal siempre se encuentre en el mismo estado.

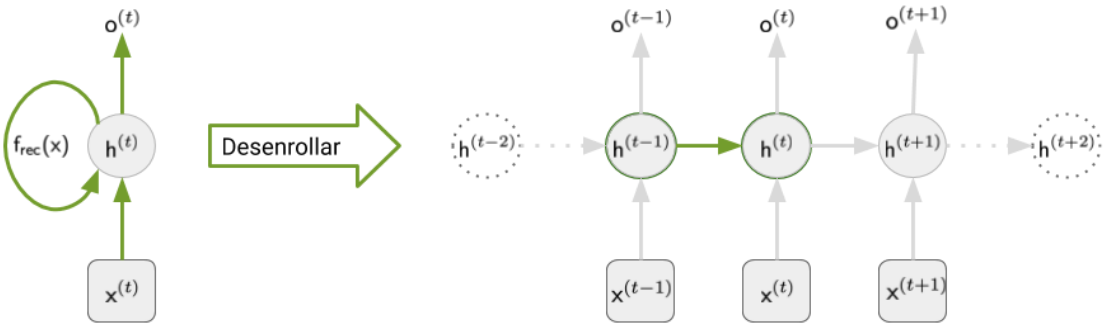
Existen dos grandes aplicaciones donde las redes neuronales feed forward son incapaces de solucionar esto: *problemas de serie de tiempo*, tales como la predicción de la bolsa o temperaturas, y *la contextualización de palabras dentro de oraciones*. En ambos problemas existe una dependencia temporal/sintáctica de los pasos previos para inferir un estado correcto.

Redes Neuronales Recurrentes

Una **red neuronal recurrente** es parte de la familia de las arquitecturas feed forward, con la peculiaridad que son capaces de preservar y enviar información temporal en cada paso. Éstas toman cada vector dentro de una secuencia de entrada y los modelan uno a la vez. Esto permite considerar contexto y memoria dentro de una secuencia temporal. La estructura básica de una red recurrente se ve como sigue:



Por lo general, una red neuronal recurrente suele presentarse de forma "enrrollada". Usualmente, para tener un mejor entendimiento de sus componentes, estas se pueden "desenrrollar" a la siguiente estructura:



En la versión "enrrollada" que encontramos a la izquierda de la figura, encontramos un input temporal en

$$x^{(t)}$$

que corresponde al dato ingresado en el tiempo actual, el cual estará asociado a un estado de celda específico que corresponde al procesamiento de la señal en el periodo actual

$$h^{(t)}$$

Ignorando la conexión recurrente

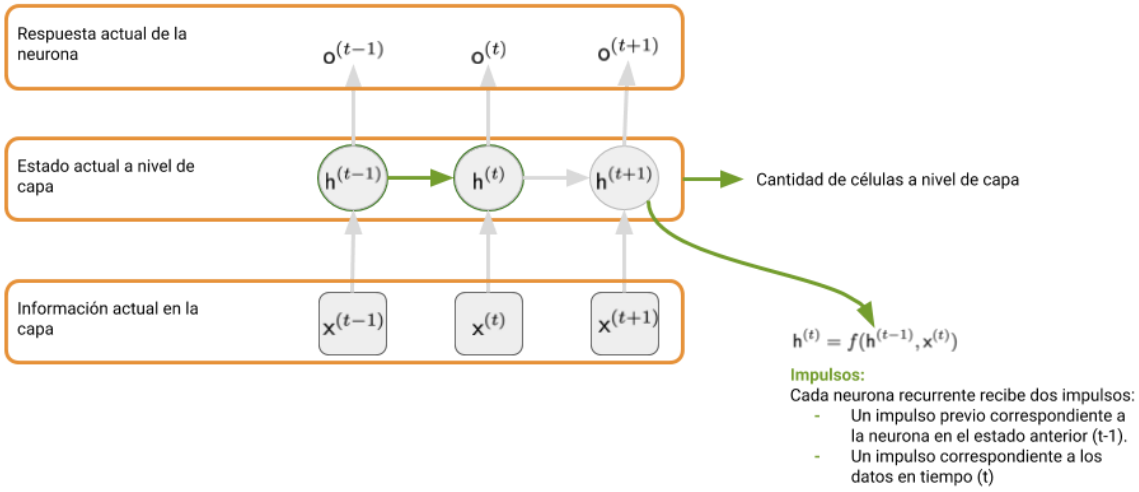
$$f_{rec}(x)$$

por el momento, observamos que este impulso procesado

$$x^{(t)} \rightarrow h^{(t)}$$

dará como resultado en un output específico.

Para tener un mejor entendimiento de los componentes de la red neuronal recurrente, tomemos el siguiente diagrama:



Resulta que cada componente detallado hace referencia a la información actual, el estado actual y la respuesta actual a nivel de neurona, ignorando secuencialidad. El rol de la conexión recurrente es aumentar la **cantidad de células a nivel de capa**, de manera tal de aumentar la capacidad de contextualizar y memorizar información. De manera adicional, las conexiones recurrentes permitirán incluír de manera adicional a la información actual, el estado de la neurona en estados previos. Así, cada neurona recurrente recibirá un impulso previo correspondiente a la neurona en el estado anterior (t-1) y un impulso correspondiente a los datos en tiempo (t) mediante la siguiente función:

$$h^{(t)} = f(h^{(t-1)}, x^{(t)})$$

El proceso de ingreso de datos es idéntico a una estructura *feed forward*, con la salvedad que genera dos outputs. Uno que representa la **respuesta actual** de la red a un estímulo específico, y otro que representa el **estado actual** de la red. Cuando vamos a una entrevista de trabajo y nos preguntan sobre algún tema en específico, nosotros somos capaces de **contextualizar** la pregunta y **memorizar** información previa. Una aplicación de red neuronal con arquitectura recurrente se ejemplifica con *Alexa* de Amazon. Cuando nosotros le preguntamos *"Alexa, ¿cómo estará el tiempo mañana?"*, la máquina procesa de forma contextual que en primer lugar le estamos solicitamos que responda una pregunta, posteriormente entenderá que deseamos saber sobre el clima, y finalmente le preguntamos sobre el estado para mañana. Cada palabra se entiende en un momento determinado **t**. Cada palabra dentro de la oración representa el input en periodo

$$x^{(t)}$$

que ingresa a la función **H** y preserva información actual en **f(x)** que representa la conexión recurrente.

La forma 'Desenrrollada' también nos permite dilucidar cómo funciona el proceso secuencia del entrenamiento. Basta ver la forma expandida para notar que es una **red neuronal feed forward con tantas capas ocultas como pasos temporales existan declarados en la red recurrente**. Un tema no menor es la complejidad algorítmica del paso **backpropagation** es

$$\mathcal{O}(n)$$

donde **n** es la cantidad de secuencias declaradas en la estructura recurrente, generando una complejidad algorítmica lineal.

Ejemplifiquemos esta limitante con el siguiente ejercicio: Estamos interesados en generar una red neuronal con **una sola capa oculta** que prediga las acciones de la bolsa, considerando que vamos a evaluar cada día del último año. Para que ésta arquitectura decida sobre el comportamiento futuro de las acciones, necesaritaríamos entrenar una red densa con 365 (o 366) capas ocultas. El problema radica en el tiempo de entrenamiento determinado por la complejidad algorítmica lineal. Para cuando esté entrenada nuestra red, es probable que el escenario haya cambiado radicalmente y hayamos perdido nuestra inversión

Capas recurrentes en Keras

En `Keras` existen varios tipos de capas asociados a la recurrencia del modelo planteado. El modelo de capa raíz utilizado se llama `Recurrent`. Cabe destacar que esta **no es una capa** como las que nosotros conocemos. Más bien, es un modelo en el que se basan las demás, por lo que no debe ser usado en ningún modelo de arquitectura.

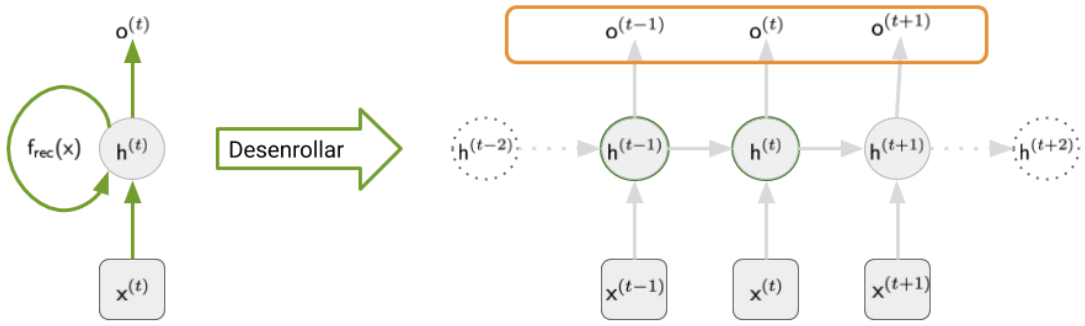
Dentro de ésta, existen tres grandes variantes que podemos implementar

- `SimpleRNN`: Red Neuronal Recurrente
- `LSTM`: Long Short Term Memory
- `GRU`: Gated Recurrent Unit.

Antes de definir nuestro problema secuencial, debemos considerar qué problema buscamos solucionar. Resulta que podemos extraer un resultado intermedio y un resultado final. Cada una de las opciones ofrece al investigador analizar el output evaluado en un paso **t** o en el paso final.

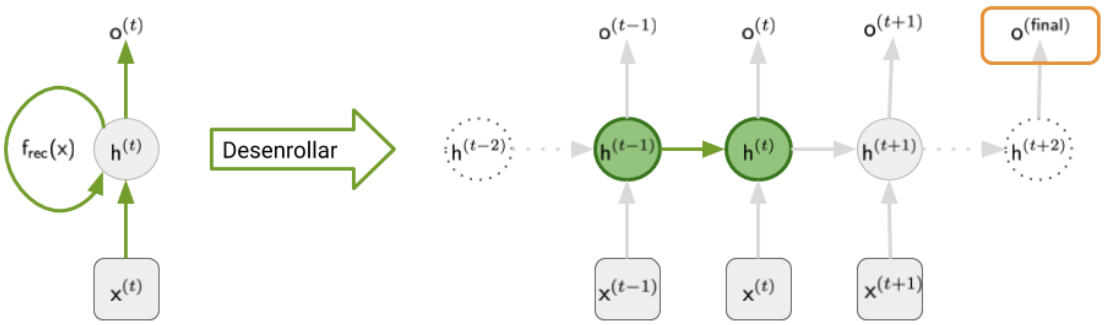
Dentro de `Keras`, podemos extraer esta información mediante el parámetro `return_sequences` en la capa.

- Cuando declaramos `return_sequences=True`, el modelo a implementar se conoce como **Many-to-Many** que permite que la red construída con `Keras` entregue la predicción en cada paso secuencial:



Modelo Many To Many: permite rescatar la predicción de cada paso secuencial.

- Cuando declaramos `return_sequences=False`, el modelo a implementar se conoce como **Many-to-One** que entrega sólo la predicción final en la red recurrente:



Modelo **Many To One:** Entrega sólo la predicción final en la red recurrente.

Finalmente, mencionar que en el contexto de redes recurrentes, se hace referencia al término **'lag'** (o rezago) como la cantidad de pasos temporales que observa la red.

Implementación de distintas arquitecturas

A continuación se presentará el código de implementación para cada una de las variantes de las redes neuronales recurrentes.

Digresión: Parámetros relevantes en las capas recurrentes:

Las capas recurrentes aceptan inputs con la forma `(samples, time_steps, features)`.

- `samples`: Cuantos registros existen.
- `time_steps`: Cuantos rezagos se van a generar en la red.
- `features`: Cantidad de atributos existentes en cada ejemplo. Por ejemplo, si queremos analizar **una** oración, eso corresponde a un número de muestras (samples) de 1, como queremos analizar una palabra a la vez, `time_steps` corresponde al **número de palabras en la frase** mientras que `features` corresponde al vector de atributos (descriptor) formado a partir de cada palabra, es decir, la **dimensionalidad de la codificación de la palabra**.

Red neuronal recurrente simple: `keras.layers.SimpleRNN`

La primera implementación responde a la implementación que hemos visto hasta ahora. Como toda implementación basada en `Keras`, necesitamos implementar un tipo de modelo (`keras.models.Sequential`), y un tipo de capa (`keras.layers.SimpleRNN`). Hasta el momento la implementación es idéntica a los ejemplos de feed forward vistos anteriormente:

```
import warnings
warnings.filterwarnings('ignore')
from keras.models import Sequential
from keras.layers import SimpleRNN
from keras.optimizers import Adam
```

Con nuestro modelo instanciado, procedemos a agregar una capa con el argumento `.add`. En este paso debemos declarar la dimensionalidad de los datos de entrada con `units`, así como la función de activación y elementos a nivel de cada neurona dentro de la capa. Resulta que debemos declarar un rezago con el argumento `input_shape` que considerará cúantas secuencias previas debe considerar el modelo recurrente. Finalmente, solicitaremos que preserve cada paso intermedio con `return_sequences=True`.

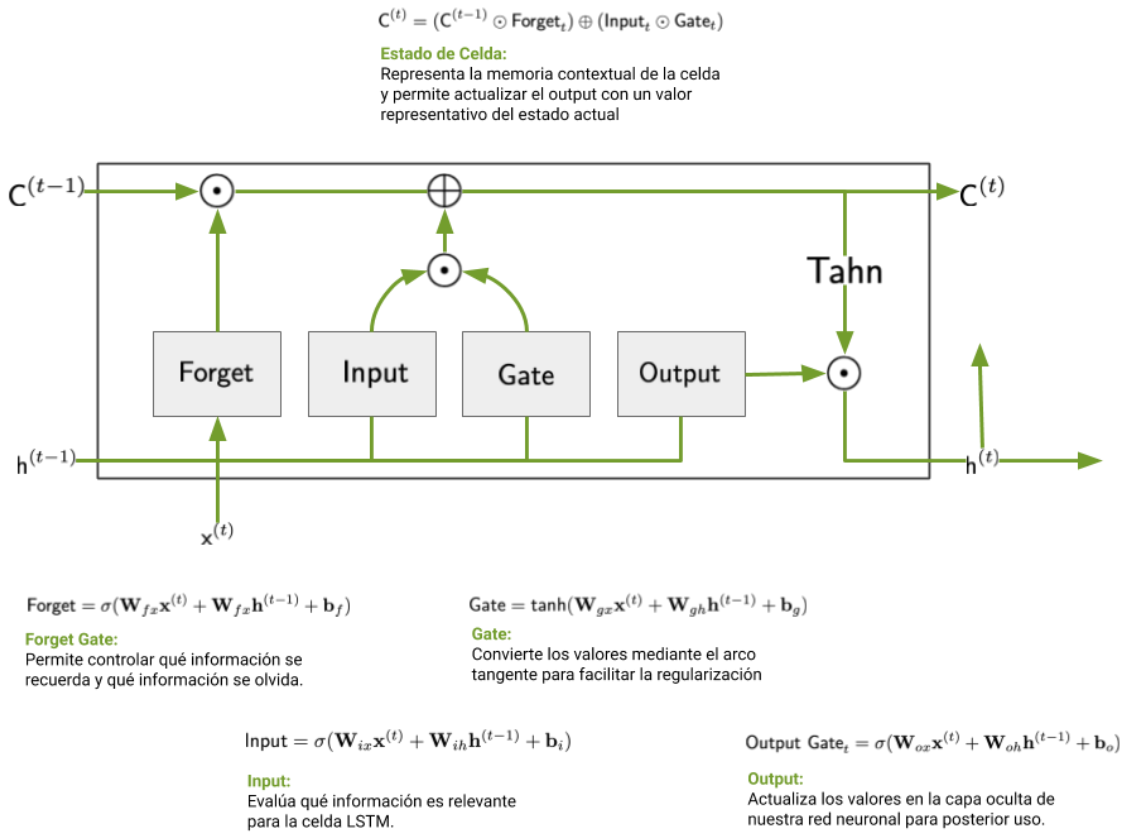
```
lag = 3 # definimos la cantidad de rezagos

# instanciamos un canvas secuencial
model_simplernn = Sequential()
# añadimos capa
model_simplernn.add(
    # recurrente simple
    SimpleRNN(units = 1, # con una neurona por paso temporal
              input_shape = (1,lag, ), # con un rezago de 3
              return_sequences = True) # solicitando resultados intermedios
    )

# Compilamos el modelo con funcion de perdida MSE y SGD como optimizador
model_simplernn.compile(
    loss='mean_squared_error', # definimos la métrica de pérdida
     optimizer=Adam()) # y un optimizador de gradiente Adam
```

Digresión: Sobre la inefectividad de la estructura recurrente simple.

Resulta que la estructura recurrente simple presenta un problema grave, asociado con los *gradientes desvanecientes*. Si bien se espera que una red neuronal recurrente pueda preservar información de muchas secuencias anteriores en un estado actual **t**, en la medida que agregamos secuencias a considerar en el poder representacional, la inestabilidad del modelo aumenta de forma substancial (Chollet, 202).

Superando el problema de recurrencia: `keras.layers.LSTM`

Para superar el contratiempo de los gradientes desvanecientes de la estructura recurrente simple, se propusieron las arquitecturas *Long Short Term Memory* y *Gated Recurrent Units*. La intuición detrás de esto es separar el comportamiento conexión recurrente en varios pasos. Antes que guardar **toda la información** dentro de la función recurrente, es mejor separar el comportamiento sobre qué se guarda y qué se elimina mediante una serie de filtros conocidos como "*compuertas*". La forma de operación de cada compuerta se detalla en el siguiente diagrama:



Mediante la desagregación de la función recurrente, estamos modificando constantemente la memoria a preservar. Al mantener solo una parte de la memoria en la secuencia **t**, nuestra red neuronal es capaz de asimimar un contexto secuencial mayor. A grandes rasgos, cada compuerta presenta una función de activación (como una función

$$\text{Sigmoide}(\mathbf{x}) \in [0, 1]$$

que determina cuándo la información se puede propagar o preservar.

Comencemos por importar nuestra capa con `keras.layers.LSTM`

```
from keras.layers import LSTM
```

Para aumentar la velocidad de entrenamiento de nuestra red LSTM, implementaremos una variante de datos de ingreso conocida como *"Online Learning"*, la cual permite asimilar datos de manera individual. De manera adicional a la ventaja en los tiempos de entrenamiento, un entrenamiento "Online" facilita la adaptación de la red a cada ejemplo de forma dinámica. Esta es una situación deseable cuando trabajamos con datos secuenciales.

```
lag = 3
batch_size = 1
model = Sequential()
```

De manera adicional, declararemos la opción `stateful=True` para que la red preserve la información del estado en distintas sesiones de entrenamiento.

```
# El parametros stateful significa que la red va a recordar el estado entre
diferentes sesiones del entrenamiento
model.add(LSTM(4,batch_input_shape = (batch_size,lag,1), stateful = True))

model.compile(loss = 'mean_squared_error', optimizer = 'adam')
```

`keras.layers.GRU`: Gated Recurrent Units

Resulta que la cantidad de pasos existentes dentro de LSTM aumenta la complejidad de entrenamiento, dado que existen tres compuertas a evaluar. Las Gated Recurrent Units buscan reexpresar los pasos en dos: Una compuerta de *reset* que determina cómo mezclar la información previa y el input actual, y una compuerta *update* que determina la cantidad de datos en **t-1** a preservar. De esta manera, mientras que las células LSTM exponen sólo el estado de salida, las GRU exponen completamente su contenido en memoria.

```
from keras.layers import GRU

lag = 3
model_gru = Sequential()

model_gru.add(GRU(units = 1,input_shape = (1,lag), return_sequences=True))

model_gru.compile(loss='mean_squared_error', optimizer='Adam')
```

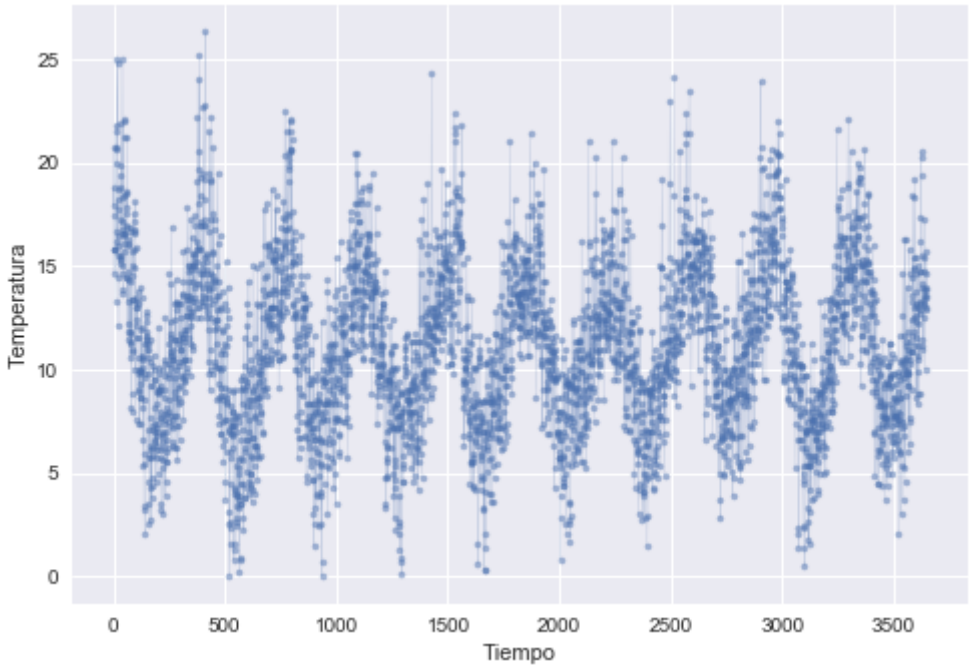
Temperaturas en Melbourne: Implementación de redes neuronales recurrentes

Ahora que hemos visto diferentes tipos de capas recurentes, probémoslas para tratar de predecir la temperatura de la ciudad de Melbourne, Australia.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
plt.rcParams['figure.figsize'] = (10, 6)
plt.style.use('seaborn')

df = pd.read_csv("data/time_series_data.csv",sep=',',usecols=
[1],engine='python',skipfooter = 3)
# renombramos la columna
df.rename(columns={'Daily minimum temperatures in Melbourne, Australia, 1981-
1990': 'Temp'}, inplace = True)
# forzamos los datos a números flotantes
df.loc[:] = df[:].astype('float32')
```

Comencemos graficando la temperatura a lo largo del tiempo para ver si hay algún patrón interesante:

```
plt.plot(df['Temp'].index,
        df['Temp'], '.-', alpha=.5,
        linewidth=.2)
plt.xlabel('Tiempo')
plt.ylabel('Temperatura');
```



A partir del gráfico observamos un patrón temporal en los datos. Los datos tienden a seguir una función trigonométrica *seno*. La temperatura tiende a fluctuar entre cierto rango de valores, condicional a la época del año. Esto representa el comportamiento climático en promedio por cada estación. Las curvas bajas representan una disminución en la temperatura, hecho más probable cuando sea invierno.

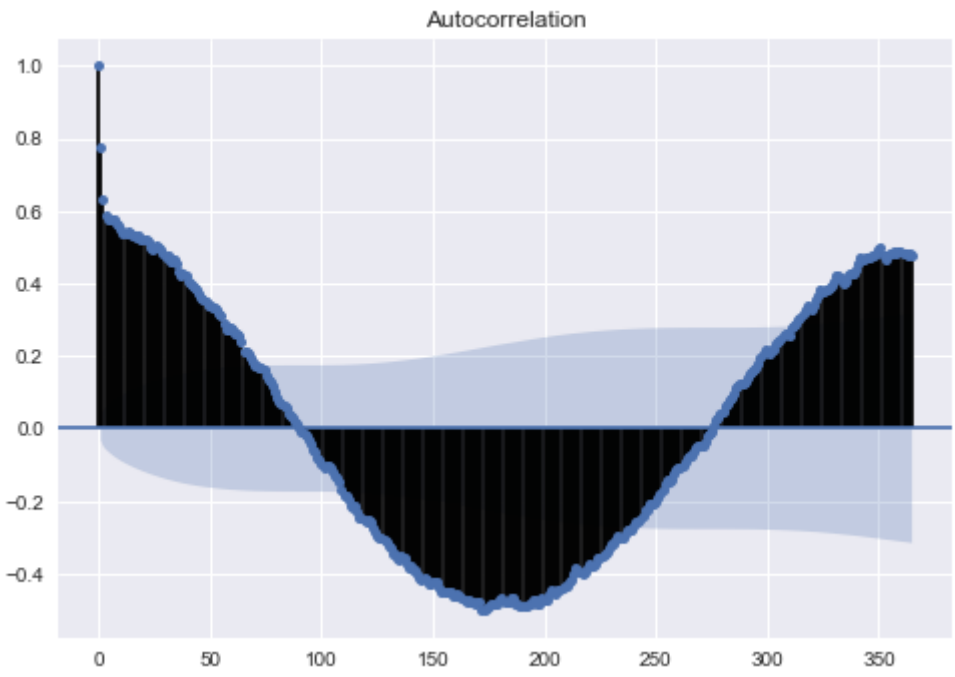
Otra buena práctica ante este tipo de datos secuenciales es implementar correlogramas. Éstos permiten analizar cuál es el grado de correlación existente entre rezagos de una misma observación. Como en este ejemplo tenemos un registro diario entre 1981 y 1990, el vector objetivo es un buen candidato a analizar desde las series de tiempo.

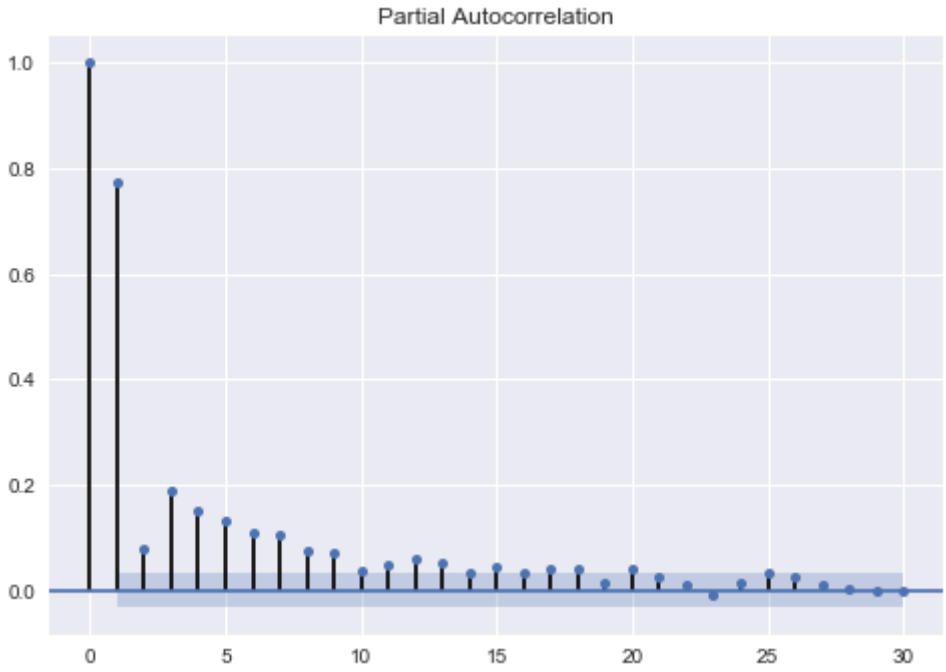
Para implementar un análisis de correlación, haremos uso de la función `statsmodels.api.graphics.tsa.plot_acf` y `statsmodels.api.graphics.tsa.plot_pacf` para analizar la autocorrelación entre unidades y autocorrelación parcial.

Lo que buscamos es evidencia de autocorrelación, lo cual confirma la naturaleza secuencial de los datos en el registro. Para ello, el gráfico de autocorrelación debe reportar correlaciones fuera del "tubo" demarcado con celeste claro. Se aprecia que las correlaciones entre unidades siguen en mismo patrón de seno, y en el periodo analizado (365 rezagos), las correlaciones son substanciales.

El gráfico de autocorrelación parcial señala cuál es la relación estricta entre una observación y su rezago previo. Éste reporta que para un registro específico, aproximadamente existe evidencia de correlación hasta los 15 días antes.

```
import statsmodels.api as sm
sm.graphics.tsa.plot_acf(df['Temp'], lags=365);
sm.graphics.tsa.plot_pacf(df['Temp'], lags=30);
```





Otro aspecto a considerar es que siempre debemos verificar la cantidad de datos faltantes en el dataset y en caso de estos ser muchos, estudiar algún patrón de sesgo en el muestreo:

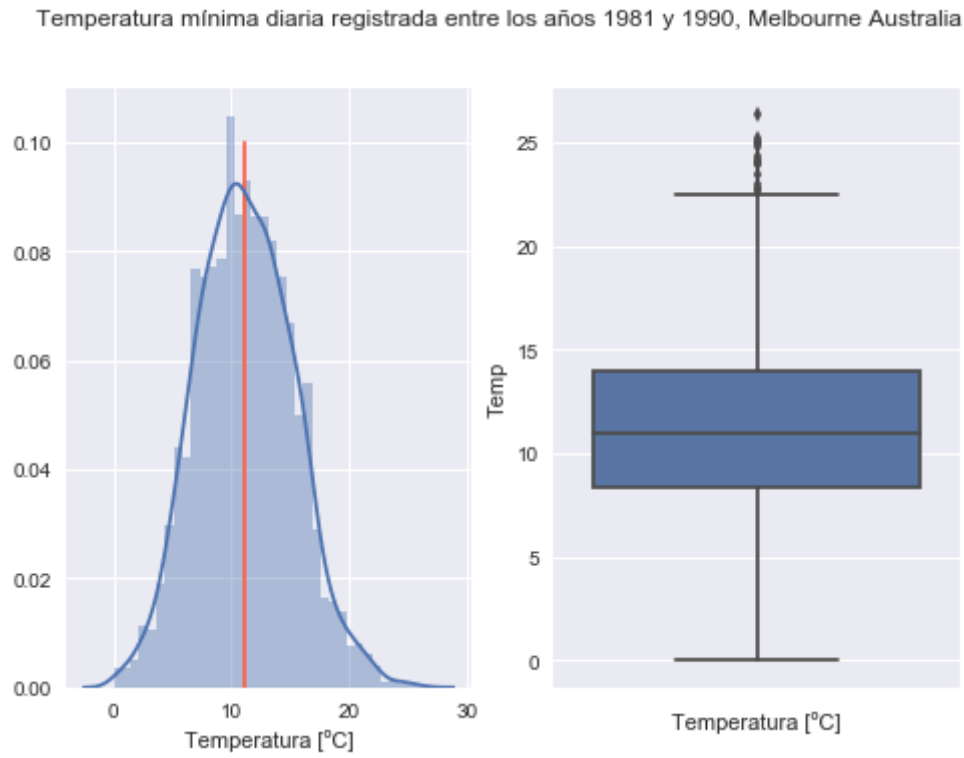
```
print('Cantidad de NaN en el dataset: %d' %df['Temp'].isnull().sum())
df.describe()
```

```
Cantidad de NaN en el dataset: 0
```

	Temp
count	3650.000000
mean	11.177745
std	4.071843
min	0.000000
25%	8.300000
50%	11.000000
75%	14.000000
max	26.299999

Finalmente, otra buena práctica es analizar la distribución empírica de la variable a analizar.

```
import numpy as np

fig, (ax1,ax2) = plt.subplots(1,2, sharey = False);

y = np.linspace(0, 0.1)
x = np.repeat(round(df['Temp'].mean(), 2), len(y))

plt.suptitle('Temperatura mínima diaria registrada entre los años 1981 y 1990, Melbourne Australia')
ax1.plot(x,y, color="tomato", ls="-")
sns.distplot(df, ax = ax1).set_xlabel('Temperatura [ºC]')
sns.boxplot(y = df['Temp'],ax = ax2).set_xlabel('Temperatura [ºC]');
```



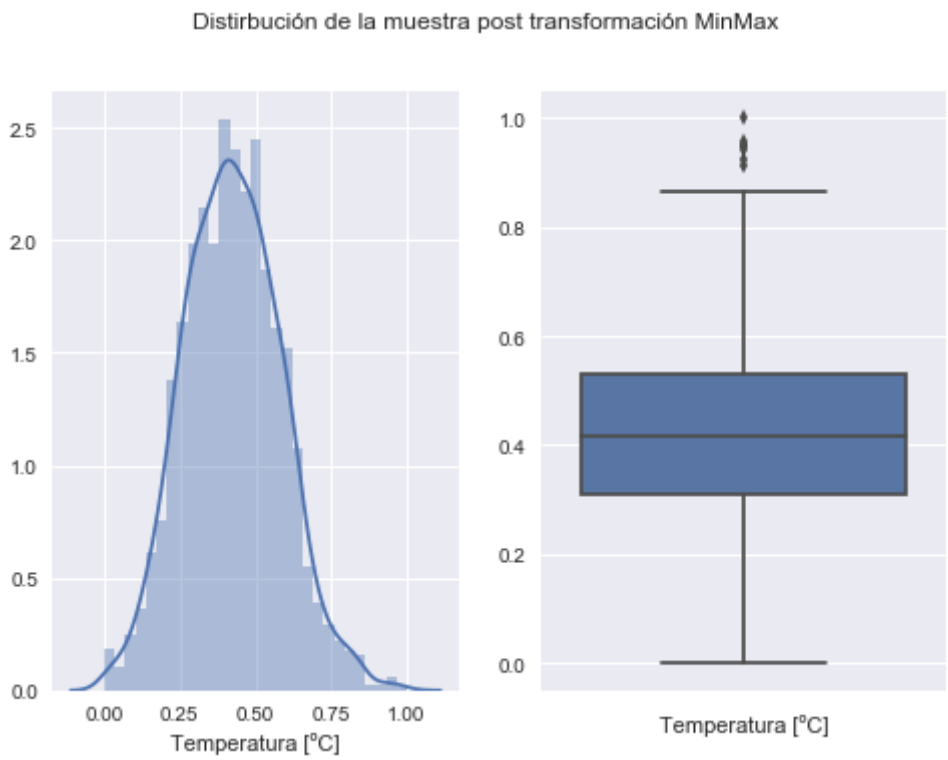
- Distribución semejante a normal, el boxplot muestra la simetría de la distribución y revela pocos outliers los cuales son de baja magnitud. Se marca con una línea naranja la media de la muestra (11.15 [ºC]).
- Dada la forma de la distribución y la cantidad de datos es seguro suponer pseudo-normalidad y utilizar estandarización en lugar de normalización.

```
from sklearn.preprocessing import MinMaxScaler

# OJO: estamos implementando hold-out validation ya que los conjunto de entrenamiento y validacion son disjuntos
df_train, df_val, df_test = df[:2000].values, df[1500:2000].values, df[2000:].values
df_train = pd.DataFrame(df_train)
df_test = pd.DataFrame(df_test)
df_val = pd.DataFrame(df_val)

# Escalamos los datos
scaler = MinMaxScaler(feature_range=(0,1)).fit(df_train)
```

```
stream_train_scaled = scaler.transform(df_train)
stream_test_scaled = scaler.transform(df_test)
stream_val_scaled = scaler.transform(df_val)
```

Ahora veamos como quedó la variable luego de escalar:

```
fig, (ax1,ax2) = plt.subplots(1,2, sharey = False);

plt.suptitle('Distirbución de la muestra post transformación MinMax')
#ax1.plot(x,y, color="tomato", ls="-")
sns.distplot(stream_train_scaled, ax = ax1).set_xlabel('Temperatura [ºC]')
sns.boxplot(y = stream_train_scaled,ax = ax2).set_xlabel('Temperatura [ºC]');
```



Diseño de la base de datos

Puesto que las redes recurrentes tienen la capacidad de recibir batches de secuencias en `Keras`, si queremos entregarle a la red la temperatura de los tres días anteriores en un determinado paso temporal, debemos debemos crear un dataset que tenga la siguiente estructura:

Tº(t-3)	Tº(t-2)	Tº(t-1)	Tº(t)	
36	40	37	43	
...	....	....	.....	

Donde **Tº(t)** es la temperatura objetivo que queremos que prediga. Para esto utilizaremos la función `create_dataset`. Esta función nos permitirá crear `n` de rezagos en nuestro vector objetivo.

```
def create_dataset(data, lag = 1):
    """
    create dataset
    """
    # generamos listas vacías de entrenamiento y validación
    train, target = [], []
    # generamos una copia del dataset
    dataset = pd.DataFrame(data)
    # para cada tupla de posición y elemento
    for index, row in dataset.itertuples():
        # concatenamos los valores rezagados para train
        train.append(dataset.iloc[index:index+lag].values)
        # concatenamos el valor actual para el vector objetivo
        target.append(dataset.iloc[index+lag,].values)
        if(index+lag+1 == dataset.shape[0]):
            break;
    return np.array(train), np.expand_dims(np.array(target), axis = 2)
```

Antes de entrenar nuestras redes neuronales recurrentes, vamos a definir los típicos conjuntos de entrenamiento, validación y holdout.

```
# Definición y dimensionado del conjunto de entrenamiento
trainX, trainY = create_dataset(stream_train_scaled, 3)
trainX = np.reshape(trainX, (trainX.shape[0], 1, trainX.shape[1]))
trainY = np.reshape(trainY, (trainY.shape[0], 1, trainY.shape[1]))

# Definición y dimensionado del conjunto de validación
testX, testY = create_dataset(stream_test_scaled, 3)
testX = np.reshape(testX, (testX.shape[0], 1, testX.shape[1]))
testY = np.reshape(testY, (testY.shape[0], 1, testY.shape[1]))

# Definición y dimensionado del conjunto holdout
valX,valY = create_dataset(stream_val_scaled, 3)
```

```
valX = np.reshape(valX, (valX.shape[0], 1, valX.shape[1]))
valY = np.reshape(valY, (valY.shape[0], 1, valY.shape[1]))
```

Esquemas de entrenamiento

Para ver el comportamiento de las redes neuronales recurrentes, estableceremos dos esquemas de entrenamiento. El esquema 1 representa una implementación online, mientras que el segundo genera batches de a 10. El detalle se encuentra detallado en las siguientes tablas.

Setup de Entrenamiento

Parámetros	Esquema 1	Esquema 2
Rezagos (lag)	3	3
Batch Size	1 (Online)	10
Timesteps	1	1
Epochs	25	25
Activación Capa	Tangente Hiperbólica	Tangente Hiperbólica
Activación LSTM	Sigmoide	Sigmoide
Optimizador	Adam	Adadelta
Función de Pérdida	MSE	MSE

Entrenamiento del Primer Esquema

```
from keras.layers import Dense, LSTM
from keras.models import Sequential
from keras.callbacks import History
history = History()
# definición de parámetros de entrenamiento
lag = 3; batch_1 = 1; inputs = 4 # La observación original más tres rezagos

model_1 = Sequential()

model_1.add(LSTM(inputs,
                 input_shape = (1, lag),
                 # activación a nivel de capas
                 activation='tanh',
                 # activación a nivel de recurrencia
                 recurrent_activation='sigmoid',
                 # resultados intermedios
                 return_sequences = True,
                 name = 'Capa_LSTM'))
# Capa de egreso
model_1.add(Dense(1,activation='linear', name = 'Output_'))

model_1.compile(loss='mean_squared_error', optimizer='adam')
model_1.summary()
```

```
%%time
model_1.fit(trainX, trainY,
            epochs=25, batch_size=batch_1,
            verbose=0, validation_data = (valX,valY),
            callbacks = [history]);
# Preservaremos el historial de entrenamiento
hist_1 = history.history
```

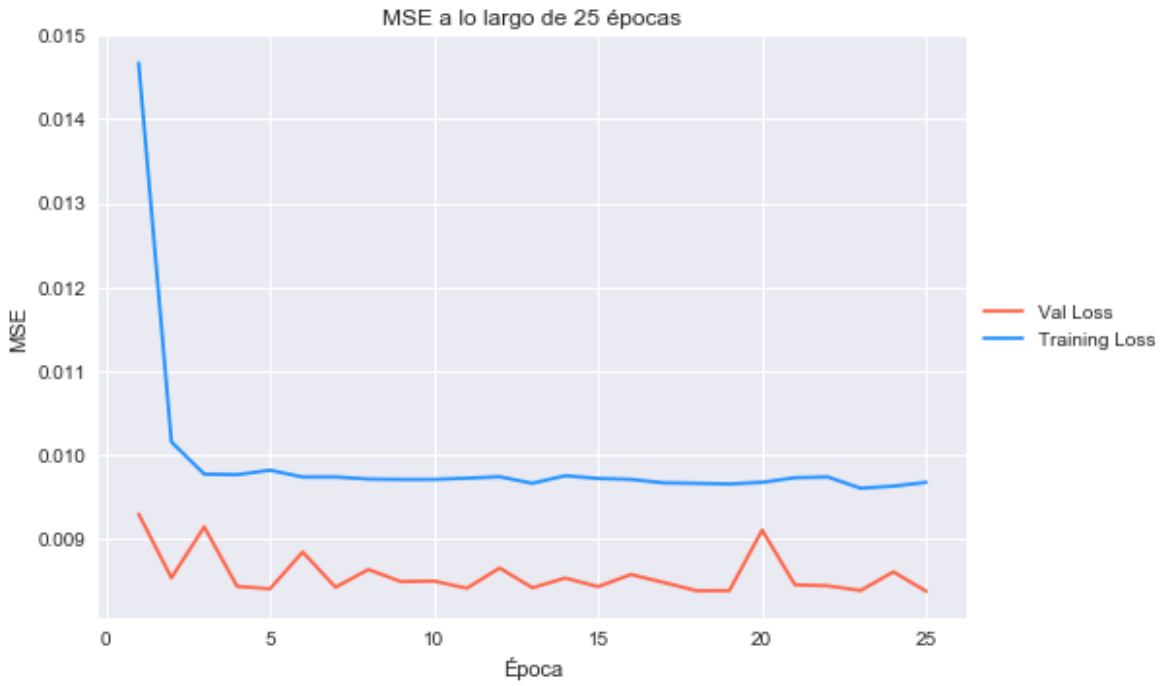
```
CPU times: user 3min 26s, sys: 1min 15s, total: 4min 42s
Wall time: 1min 24s
```

Si evaluamos la función de pérdida entre el inicio y el fin de las épocas de entrenamiento, observamos una disminución de aproximadamente 0.0009 puntos.

```
diferencial_loss = round(hist_1['val_loss'][0]-hist_1['val_loss'][-1], 4)
print('Pérdida final en validación: {0}'.format(hist_1['val_loss']
[-1].round(4)))
print('Diferencial de pérdida entre inicio y final :
{0}'.format(diferencial_loss))
```

```
Pérdida final en validación: 0.0084
Diferencial de pérdida entre inicio y final : 0.0009
```

```
plt.title('MSE a lo largo de 25 épocas')
plt.xlabel('Época')
plt.ylabel('MSE')
sns.lineplot(range(1,len(hist_1['val_loss'])+1),
             hist_1['val_loss'],
             label = 'Val Loss', color = 'tomato');
sns.lineplot(range(1,len(hist_1['loss'])+1),
             hist_1['loss'], label = 'Training Loss', color = 'dodgerblue');
plt.legend(loc='center left', bbox_to_anchor=(1, .5))
```



Lo primero que podemos observar es que el rendimiento del modelo se mantiene casi constante a lo largo de todo el entrenamiento: La red no está aprendiendo!. El desempeño que observamos se debe casi netamente a la potencia de la arquitectura. Tenemos varias opciones cuando esto pasa:

- Podemos modificar la agenda de entrenamiento con la esperanza de observar convergencia para un número de épocas más grande. Sin embargo, con la cantidad de datos que tenemos y dada la simpleza de la arquitectura probablemente aumentar el número de epoch no vaya a mejorar mucho la solución.
- Podemos modificar algunos hiperparámetros que puedan estar restringiendo demasiado (o muy poco) el espacio de búsqueda del modelo. Por ejemplo, si fijamos regularizaciones demasiado estrictas el modelo se va a ver impedido de poder explorar correctamente el espacio de búsqueda.
- Podemos cambiar la arquitectura, en el caso de las redes neuronales cambiando la arquitectura cambiamos el espacio de búsqueda sin cambiar el modelo, esto es una característica interesante del modelo neuronal (Más sobre esto en la digresión al final de este ejemplo).
- Podemos simplemente cambiar de modelo y probar con otro que permita realizar predicciones sobre series temporales.

Ahora veamos de forma gráfica que tan bien se ajustan las predicciones al comportamiento real del fenómeno.

```
# Conjunto de entrenamiento
trainPredict_1 = model_1.predict(trainX, batch_size = batch_1)
trainPredict_1 = scaler.inverse_transform(trainPredict_1[:,:,0])
trainY_1 = scaler.inverse_transform(trainY[:,:,0])

# Conjunto de validación
testPredict_1 = model_1.predict(testX, batch_size = batch_1)
testPredict_1 = scaler.inverse_transform(testPredict_1[:,:,0])
testY_1 = scaler.inverse_transform(testY[:,:,0])
```

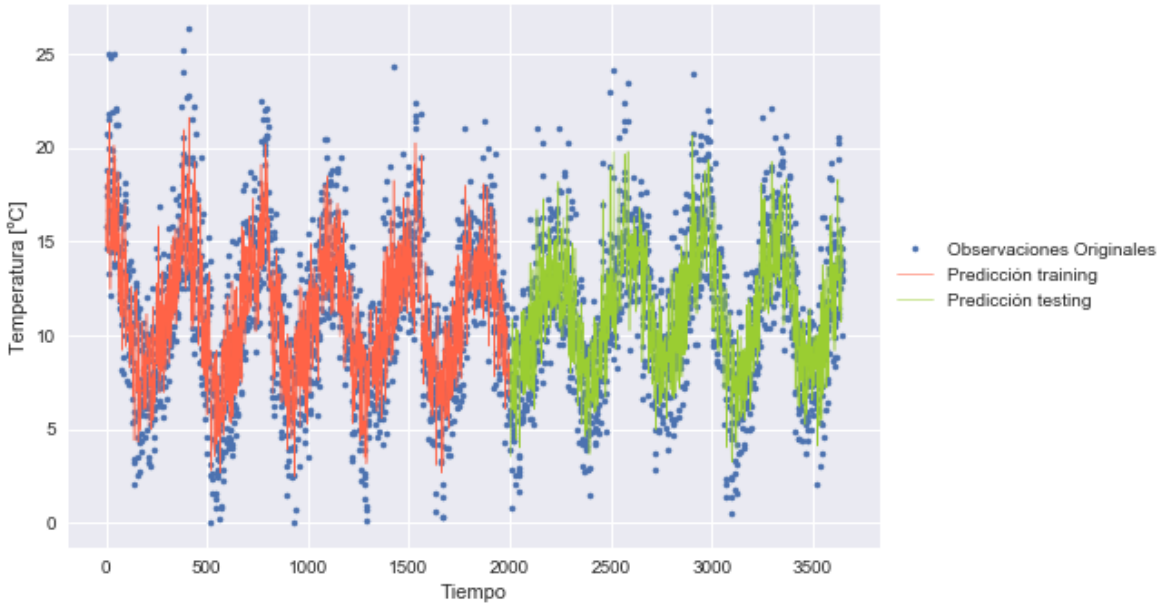
```
import math
from sklearn.metrics import mean_squared_error
report_rmse = lambda x, y: round(math.sqrt(mean_squared_error(x[:, 0], y[:,
0])), 3)

print("RMSE Train: {}".format(report_rmse(trainY_1, trainPredict_1)))
print("RMSE Test: {}".format(report_rmse(testY_1, testPredict_1)))
```

```
RMSE Train: 2.558
RMSE Test: 2.451
```

Necesitamos hacer un cambio de la serie por el rezago que introdujimos, de lo contrario vamos a ver las predicciones corridas con respecto al valor predicho.

```
trainPredictPlot = np.empty_like(df.values)
trainPredictPlot[:, :] = np.nan
trainPredictPlot[lag-1:len(trainPredict_1)+lag-1, :] = trainPredict_1
trainPredictPlot = pd.DataFrame(trainPredictPlot, columns = ['Temp'])

#shift test predictions for plotting
testPredictPlot = np.empty_like(df.values)
testPredictPlot[:,:] = np.nan
train_end_index = 2*lag + len(trainPredict_1)
testPredictPlot[train_end_index:, :] = testPredict_1
testPredictPlot = pd.DataFrame(testPredictPlot, columns=['Temp'])
```

```
data_range = range(0, len(df['Temp']))
plt.plot(data_range, df['Temp'], '.', label='Observaciones Originales')
plt.plot(data_range, trainPredictPlot['Temp'],
         color='tomato', lw=.5, label='Predicción training')
plt.plot(data_range, testPredictPlot['Temp'],
         color='yellowgreen', lw=.5, label='Predicción testing')
plt.xlabel('Tiempo')
plt.ylabel('Temperatura [ºC]');
plt.legend(loc='center left', bbox_to_anchor=(1, .5));
```

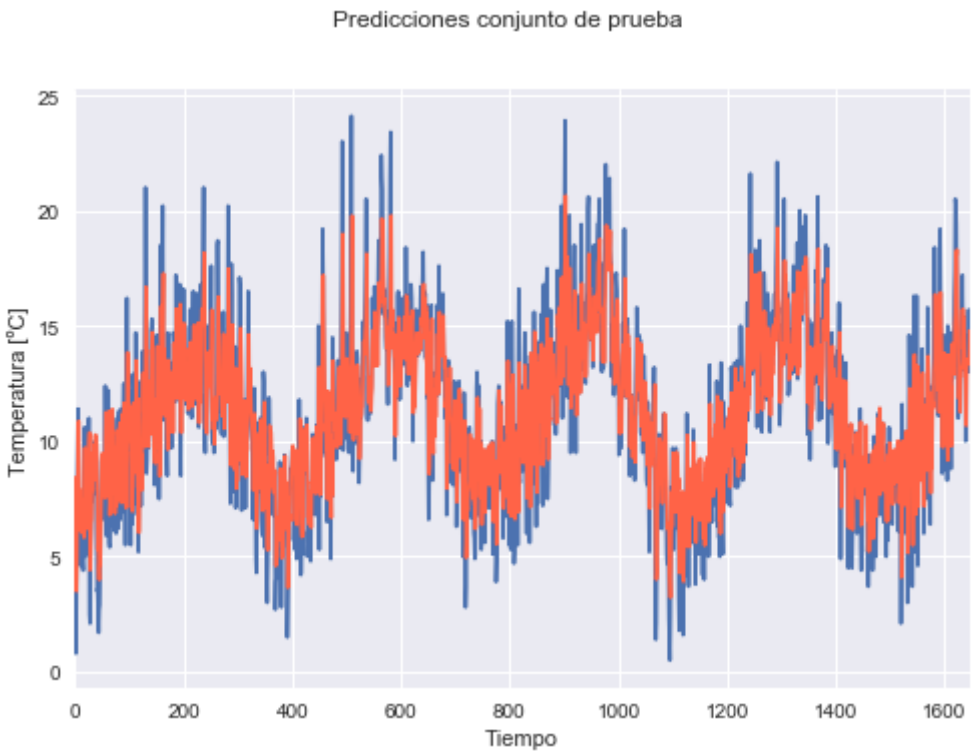


Podemos ver que en general nuestro modelo se comporta bastante bien para predecir el comportamiento del fenómeno, sin embargo, aún puede mejorar algunas cosas:

- Subestima por mucho los puntos de máximos/mínimos extremos. Si nuestro modelo está pensado para predecir y alertar cuando ocurran olas de calor o frio extremo necesitamos mejorar eso.
- En general, aún sin contar puntos extremos, el modelo subestima la predicción, en otras palabras *'se queda corto'* casi siempre.

Veamos de cerca el conjunto de pruebas:

```
testPredictPlot = testPredictPlot.dropna()
testPredictPlot.columns = ['Temp']
testPredictPlot.set_index('Temp')
fig, ax = plt.subplots(1,1)

sns.tsplot(df['Temp'][-len(testPredictPlot):], ax = ax)
sns.tsplot(testPredictPlot['Temp'], color='tomato', ax= ax);
fig.suptitle('Predicciones conjunto de prueba')
plt.xlabel('Tiempo')
plt.ylabel('Temperatura [ºC]');
```


Predicciones conjunto de prueba
Temperatura [°C]
Tiempo
0
5
10
15
20
25
0
200
400
600
800
1000
1200
1400
1600

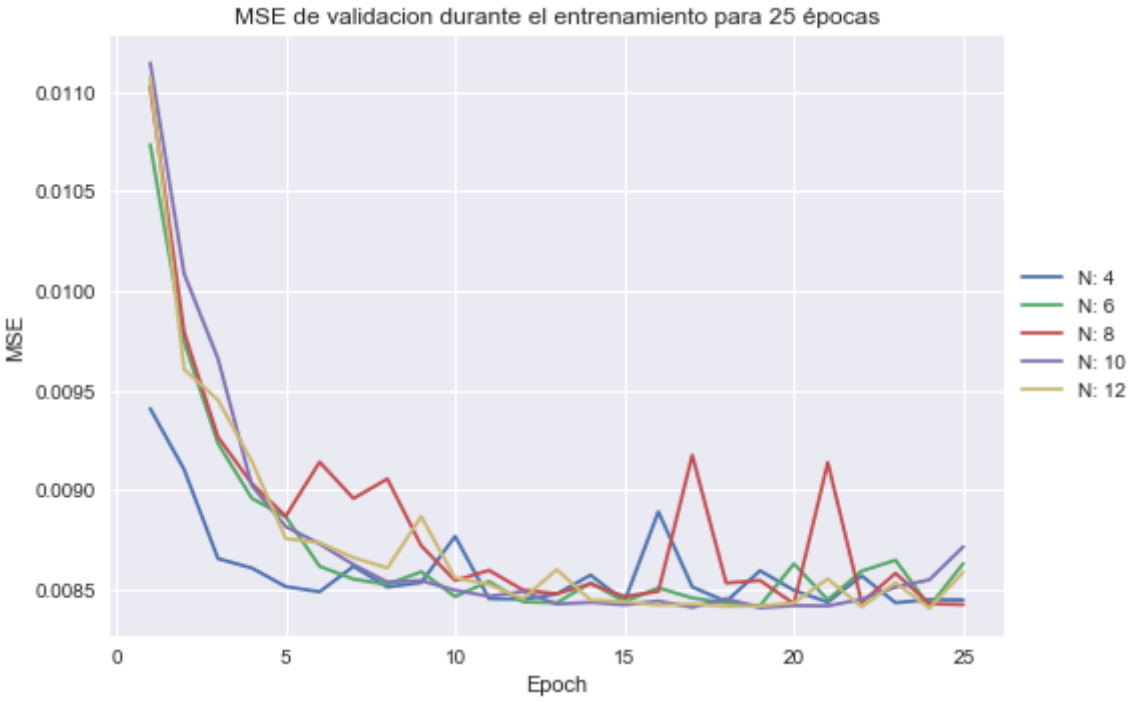
Determinando el número de bloques

Hasta el momento hemos trabajado con un número fijo de bloques a implementar en la capa. Para este experimiento evaluaremos el efecto de la cantidad de bloques LSTM dentro de ésta. Por motivos prácticos, volveremos a generar los conjuntos de entrenamiento, validación y holdout.

```
lag = 3
trainX, trainY = create_dataset(stream_train_scaled, lag)
testX, testY = create_dataset(stream_test_scaled, lag)
valX,valY = create_dataset(stream_val_scaled, lag)

trainX = np.reshape(trainX, (trainX.shape[0], 1, trainX.shape[1]))
trainY = np.reshape(trainY, (trainY.shape[0], 1, trainY.shape[1]))
testX = np.reshape(testX, (testX.shape[0], 1, testX.shape[1]))
testY = np.reshape(testY, (testY.shape[0], 1, testY.shape[1]))
valX = np.reshape(valX, (valX.shape[0], 1, valX.shape[1]))
valY = np.reshape(valY, (valY.shape[0], 1, valY.shape[1]))
```

```
# Definimos el tamaño de los batches.
batch_size = 10
# generamos una lista vacía que contenga la historia de cada modelo
implementado
loss_nb = []

# definimos el rango de bloques a implementar en la capa
nb = range(4,13,2)
# para cada bloque definido
for blocks in nb:
    # implementamos una estructura conectada completa
    model_blocks = Sequential()
    # incorporamos una capa LSTM con un número definido de bloques
    model_blocks.add(LSTM(units = blocks,
                          # con un rezago de 3
                          input_shape = (1,lag),
                          # una función de activación tanh a nivel de capa
                          activation='tanh',
                          # una función de activacieon sigmoide a nivel de
bloque
                          recurrent_activation='sigmoid',
                          return_sequences = True))

    # definimos la capa de salida
    model_blocks.add(Dense(1,activation='linear'))
    # compilamos con adadelta y medimos mse
    model_blocks.compile(loss='mean_squared_error', optimizer='adadelta')
    # generamos el fit del modelo
    model_blocks.fit(trainX, trainY, epochs=25, batch_size=10, verbose = 0,
              callbacks = [history],
              validation_data = (valX,valY))
    loss_nb.append(history.history)
```

```
# MSE training
for i, loss in enumerate(loss_nb):
    plt.plot(range(1,26),
             loss['val_loss'],
             label = 'N: {0}'.format(nb.start+i*nb.step));

plt.title('MSE de validacion durante el entrenamiento para 25 épocas')
plt.xlabel('Epoch')
plt.ylabel('MSE')
plt.legend(loc='center left', bbox_to_anchor=(1, .5));
```




```
# MSE test
print('Nº Bloques | Error de validación')
print('-----------|----------------------')
# definimos un nivel de
min_loss = 1e3
model_min_loss = 0
for i, loss in enumerate(loss_nb):
    if min(loss['val_loss']) < min_loss:
        min_loss = min(loss['val_loss']).round(6)
        model_min_loss = nb.start+i*nb.step
    print('{0} \t   | {1}'.format(nb.start+i*nb.step,
min(loss['val_loss']).round(6)))
print('\nEl modelo con el número de bloques {0} obtuvo el menor MSE, con un
error de {1}'.format(model_min_loss,min_loss ))
```

```
Nº Bloques | Error de validación
-----------|----------------------
4      | 0.008435
6      | 0.008417
8      | 0.008426
10     | 0.008409
12     | 0.008408

El modelo con el número de bloques 12 obtuvo el menor MSE, con un error de
0.008408
```

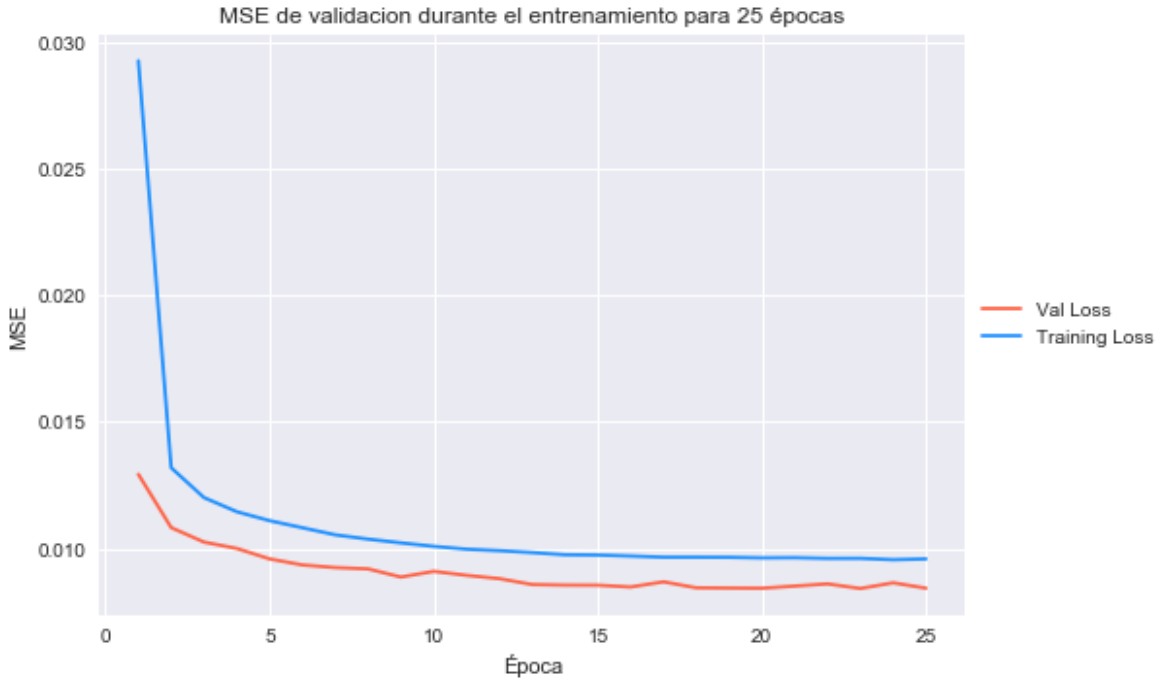
Para finalizar, probemos el rendimiento de las GRU

```
lag = 3
model_gru = Sequential()

model_gru.add(GRU(units = 1,input_shape = (1,lag),
                  return_sequences=True,
                  recurrent_initializer = 'orthogonal',
                  activation = 'tanh'))

model_gru.compile(loss='mean_squared_error', optimizer='Adam')

model_gru.fit(trainX, trainY, epochs=25,
              batch_size=1,
              verbose = 0,
              callbacks = [history],
              validation_data=(valX,valY));
loss_gru = history.history
```

```
# Conjunto de pruebas
testPredict = model_gru.predict(testX, batch_size = 1)
testPredict = scaler.inverse_transform(testPredict[:,0])
test_score_gru = mean_squared_error(scaler.inverse_transform(testY[:,0])[:,0],
testPredict[:,0])
```

```
print('Pérdida final en validación: {0}'.format(loss_gru['val_loss']
[-1].round(4)))
print('Diferencial de pérdida entre inicio y final:
{0}'.format(round(loss_gru['val_loss'][0]-loss_gru['val_loss'][-1], 4)))

plt.title('MSE de validacion durante el entrenamiento para 25 épocas')
plt.xlabel('Época')
plt.ylabel('MSE')
sns.lineplot(range(1,len(loss_gru['val_loss'])+1), loss_gru['val_loss'], label
= 'Val Loss', color = 'tomato');
sns.lineplot(range(1,len(loss_gru['loss'])+1), loss_gru['loss'], label =
'Training Loss', color = 'dodgerblue');
plt.legend(loc='center left', bbox_to_anchor=(1, .5));
```

```
Pérdida final en validación: 0.0084
Diferencial de pérdida entre inicio y final: 0.0045
```


MSE de validacion durante el entrenamiento para 25 épocas
0.030
0.025
0.020
0.015
0.010
MSE
0
5
10
15
20
25
Época
Val Loss
Training Loss

Conclusión: Sobre los espacios de búsqueda, parámetros y límites de un modelo

Quizás se hayan dado cuenta que en esta lectura los modelos LSTM y GRU presentan unas curvas de entrenamiento curiosas: En primer lugar, son casi planas. En segundo lugar, la curva de validación muestra que el modelo se comporta mejor en la validación que en el entrenamiento, ¿Cómo es esto posible?

Hay mucho de lo que hablar, partiremos por un tema transversal en nuestro estudio y que corresponde a la base de la disciplina de la Inteligencia Artificial. Cuando modelamos un fenómeno de forma matemática, siempre nos enfrentamos al problema de encontrar los parámetros del modelo que hacen que el modelo se apegue lo mejor posible a la realidad. Este es un problema complejo que hemos mencionado antes y toma una forma particular en el caso de las redes neuronales. El espacio de búsqueda de un modelo para un problema depende exclusivamente de los datos, del modelo mismo y de sus parámetros. La geografía y amplitud del espacio de búsqueda no dependen en ningún caso de los hiperparámetros que escojamos. Estos últimos solo ayudan a hacer más ameno el entrenamiento y a recorrer de mejor forma el espacio.

Teniendo en cuenta lo anterior, un modelo con más parámetros por lo general tendrá un espacio de búsqueda más grande. Para ver esto basta hacer el siguiente ejercicio mental: piensen en un modelo que solamente acepta un parámetro, luego, nuestra búsqueda se reduce a recorrer una curva; cuando aumentamos la cantidad de parámetros a dos tenemos un plano y así sucesivamente. Como se podrán imaginar, lo ideal es tener un modelo con poca cantidad de parámetros e hiperparámetros pero que a a la vez entregue un buen rendimiento, a esto nos referimos como 'parsimonía'. Lamentablemente, en la vida real pocas veces podemos modelar satisfactoriamente un fenómeno con modelos simples (lo que no significa que mayor cantidad de parámetros sea necesariamente un indicador de un buen modelo), por lo que nos vemos obligados a recurrir a modelos más complejos.

Muchas veces es imposible alcanzar los resultados esperados con un modelo en particular debido al comportamiento del modelo frente a los datos, es decir, al espacio de búsqueda que genera para esos datos en específico. Por lo tanto, **no existe un solo modelo que se desempeñe bien para toda tarea que se le de**; a esto se le conoce como **'No Free Lunch Theorem'**. Cuando parece imposible alcanzar los resultados deseados con un modelo, podemos sospechar que nos acercamos al límite del modelo, en otras palabras, el modelo es muy simple para la complejidad del fenómeno.

En el caso de las redes neuronales, los parámetros son los pesos, por lo que al cambiar la arquitectura de la red (agregar más parámetros mediante la adición de más células o capas o modificar el orden de cualquier forma) cambia el espacio de búsqueda.

En las curvas que observamos pueden estar pasando varias cosas, pero dado que los datos se ven bastante ordenados, siguen un patrón claro y estamos utilizando excelentes optimizadores. El hecho de que nuestro modelo no sea capaz de disminuir más el error puede ser porque ese es el límite del mismo. Ppara solucionar esto podemos utilizar otro modelo, o cambiar la arquitectura de la red agregando más capas, capas distintas o incluso más neuronas.

Referencias

- [1] *"Deep Learning"*.- I. Goodfellow, Y. Bengio y A. Courville.
- [2] *"A simple weight decay can improve generalization"*.- Krogh y Hertz, 1991.
- [3] *"ImageNet classification with Deep Convolutional Neural Networks"*.- Hinton, Sutskever y Krizhevsky, 2012.
- [4] *"Comparing Biases for Minimal Network Construction with Backpropagation"*.- Hanson y Pratt, 1989.
- [5] *"Fixing Weight Decay Regularization in Adam"*.- Loshchilov y Hutter, 2017.
- [6] *"Long Short-term Memory"*.- S. Hochreiter y J. Schmidhuber, 1997.
- *"Deep Learning: A practitioners approach"*.- J. Patterson y A. Gibson, capítulo 4: "Major Architectures for Deep Learning".
- *"Deep Learning"*.- I. Goodfellow, Y. Bengio y A. Courville, capítulo 10: "Sequence Modeling: Recurrent and Recursive Nets".
- *"Deep Learning with Python"*.- F. Chollet, capítulo 6, parte 2: "Deep Learning for text and sequences".