Sistema de predicción de la calidad del aire utilizando la red de sensores ciudadanos científicos mediante técnicas de inteligencia artificial

Juan Diego Gallego Villada

Tecnología en Electrónica

Directores de producto del laboratorio

Juan David Martínez Vargas

Andrés Felipe Giraldo Forero

INSTITUTO TECNOLÓGICO METROPOLITANO

Junio de 2020

RESUMEN

Este producto de laboratorio se enfoca en diseñar un sistema que, utilizando algoritmos de inteligencia artificial, permita realizar la predicción de mediciones futuras utilizando información de toda la red de sensores de bajo costo “ciudadanos científicos”, desplegada por el SIATA a lo largo del área metropolitana.

Debido a que las redes neuronales tradicionales son incapaces de relacionar información del pasado, relacionarla con una tarea y predecir un evento futuro, se opta por implementar una arquitectura especial de las redes neuronales recurrentes, la LSTM. Por lo tanto, se estudia qué es una LSTM y cómo implementarla en Python de la mejor manera.

Se selecciona un sensor objetivo de la red de sensores de bajo costo “ciudadanos científicos” que posea un mínimo de datos faltantes y que, además, tuviera sensores activos durante el 2008 como vecinos.

Se implementa la red LSTM en Python usando la plataforma Kaggle para hacer uso de su GPU en la nube. Además, se realizan 3 cálculos de error diferente (MAE, MAPE y RMSE) para comparar las predicciones hechas por la red con las mediciones reales del sensor objetivo.

Se logra la predicción de mediciones con un horizonte de predicción de una hora a partir de mediciones de 5 horas anteriores, para un pronóstico de casi 360 horas (una semana). Las predicciones se realizaron de tres maneras diferentes: usando mediciones anteriores del sensor objetivo para entrenar la red, usando datos de todos los sensores de la red y finalmente, usando datos de los sensores vecinos del sensor objetivo.

Con la metodología usada para el desarrollo de este producto de laboratorio, se obtienen pronósticos adecuados. Usando datos anteriores del sensor objetivo, se obtiene un pronóstico con un MAPE menor al 10%, posicionando a dichos datos de entrenamiento como la mejor opción siguiendo la metodología presentada en el presente documento.

Palabras clave: Sensores de calidad del aire, redes neuronales recurrentes, long short-term memory, Python, Inteligencia Artificial.

RECONOCIMIENTOS

Como estudiante de tecnología en electrónica próximo a culminar este ciclo educativo, agradezco:

A todos los docentes que durante mi proceso de formación han estado alegres y motivados a compartir su conocimiento y experiencia. Así como a los laboratoristas de la institución (Laboratorio de circuitos eléctricos, Laboratorio de microelectrónica y Laboratorio de control lógico programable PLC) por tener una excelente disposición para ayudar y resolver dudas.

A los profesores Juan David Martínez Vargas y Andrés Felipe Giraldo Forero, quienes por su experiencia, conocimiento y buena disposición han sido fundamentales en mi proceso de formación e investigación y, por consiguiente, en el satisfactorio desarrollo del presente producto de laboratorio.

Finalmente, pero no menos importante, agradezco a mí familia, quienes, con su apoyo incondicional, me han impulsado siempre a confiar en mis capacidades y a continuar aprendiendo.

ACRÓNIMOS

RNN Redes Neuronales Recurrentes

LSTM Long Short-Term Memory

SIATA Sistema de Alerta Temprana de Medellín y el Valle de Aburrá

GPU Graphics Processing Unit

MAE Error Medio Absoluto (Mean Absolute Error)

MAPE Error porcentual absoluto medio (Mean Absolute Percentage Error)

RMSE Error cuadrático medio (Root Mean Squared Error)

TABLA DE CONTENIDO

1. INTRODUCCIÓN

1.1 Generalidades:

La contaminación por material particulado en el aire de Medellín y su área metropolitana es alta, y tiene repercusiones negativas sobre la salud de los habitantes. Por esta razón, monitorear la calidad del aire puede ayudar a entes administrativos a tomar decisiones que permitan mitigar los impactos negativos en la población general. Para esta tarea, el SIATA ha desplegado a lo largo del área metropolitana una red de sensores de bajo costo conocida como “ciudadanos científicos”. Sin embargo, dada la naturaleza de los sensores, la medición generada por cada uno de ellos puede ser de baja precisión. Para mitigar dicho problema, se propone un sistema que, haciendo uso de algoritmos de inteligencia artificial, permita mejorar la medición de cada sensor usando la información de toda la red.

1.2 Objetivo general:

Diseñar una LSTM que realice la predicción de los valores medidos por un sensor de la red ciudadanos científicos a partir de mediciones de otros sensores de la red. Esto con el fin de obtener medidas de mayor precisión.

1.3 Objetivos específicos:

- Diseñar una LSTM capaz de predecir de forma acertada mediciones de un sensor de bajo costo a partir de sus mediciones pasadas.
- Adaptar dicha LSTM para que logre predecir acertadamente las mediciones de un sensor específico a partir de las mediciones de todos los sensores de la red de sensores.
- Predecir mediciones de un sensor específico a partir de las mediciones de los sensores vecinos.

1.4 Descripción de los capítulos:

Inicialmente se contextualiza al lector con teoría de las redes LSTMs. Se explica, se iluestra y se plantea mediante ecuaciones el funcionamiento de estas.

Se describe la metodología usada para el correcto desarrollo del presente producto de laboratorio.

Los resultados obtenidos se evidencian mediante el código en Python implementado en Kaggle, las gráficas de las predicciones vs los datos reales y los tres cálculos de error realizados para calificar las tres diferentes maneras de realizar las predicciones.

Para finalizar, se analizan los resultados obtenidos junto con sus limitaciones y casas de inexactitud. Además, se plantea la proyección del trabajo realizado, en el sentido de mayores desarrollos que se pueden alcanzar.

2. MARCO TEÓRICO

2.1 Redes Neuronales Recurrentes (RNNs)

A diferencia del cerebro humano, las redes neuronales tradicionales no poseen la capacidad de analizar eventos actuales a partir de datos pasados. Esto debido a que en ellas las salidas se consideran independientes de las entradas, pero, esto representa un gran inconveniente a la hora de realizar algunas tareas como lo son el reconocimiento del habla y el análisis de video. Ambas tareas son procesos secuenciales en los cuales la predicción de un evento futuro depende completamente del pasado. (Antonio, 2016).

Para resolver el inconveniente de retención de información se usan las redes neuronales recurrentes (RNNs). Estas redes neuronales cuentan con un lazo de retroalimentación que permite que la información sea retenida. Se le conoce como red neuronal recurrente porque ésta realiza el mismo cálculo para cada elemento de una secuencia y pasa cierta información, lo que significa que cada salida dependerá de los cálculos que se hayan realizado con anterioridad. Básicamente una RNN puede entenderse como múltiples capas de una misma red, en la que cada capa transmite información a su capa sucesora.

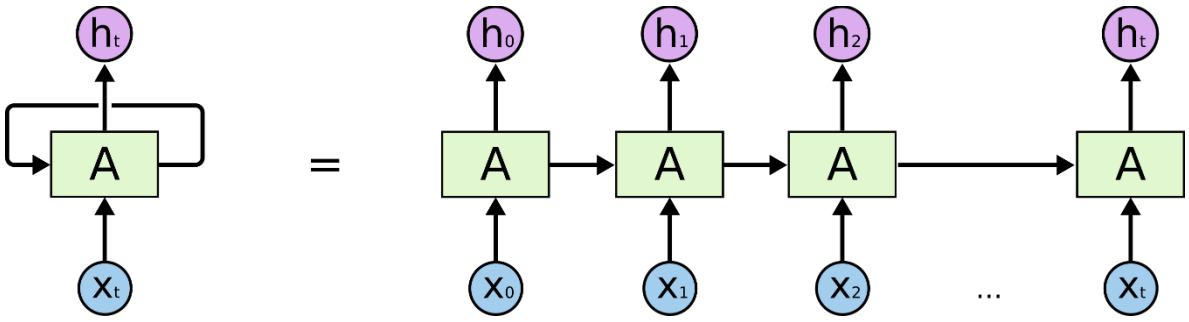


Figura 1. Red Neuronal Recurrente (RNN). (Olah, 2015)

Las RNNs son realmente útiles cuando se trata de realizar una tarea a partir de información previa pero reciente, es decir, estas redes en su arquitectura más básica pueden funcionar correctamente siempre cuando la brecha de tiempo entre la tarea que se realiza y la

información relevante para el contexto no sea grande. Así, la eficiencia de una red neuronal recurrente es inversamente proporcional al tamaño de la brecha anteriormente mencionada. (Olah, 2015).

2.2 Long Short-Term Memory (LSTM)

Para resolver el inconveniente expuesto anteriormente, se usan las Long Short Term Memory (LSTMs), estas no presentan inconvenientes para recordar información con brechas grandes de tiempo en una secuencia de datos ya que es su comportamiento por defecto.

Tal como se mencionaba anteriormente, una RNN puede entenderse como una cadena de módulos repetidos, en la que dicho modulo posee una única capa con una función de activación simple como lo es la tanh. Las LSTMs son similares, pero el módulo que se repite contiene cuatro capas que interactúan entre sí.

El concepto clave de las LSTMs es el cell state, que básicamente transporta la información a través de la cadena de módulos, evitando así sufrir del mismo inconveniente de memoria a corto plazo de las RNNs. La información del cell state es removida o añadida a medida que esta cruza a través de la cadena de módulos. (Phi, 2018).

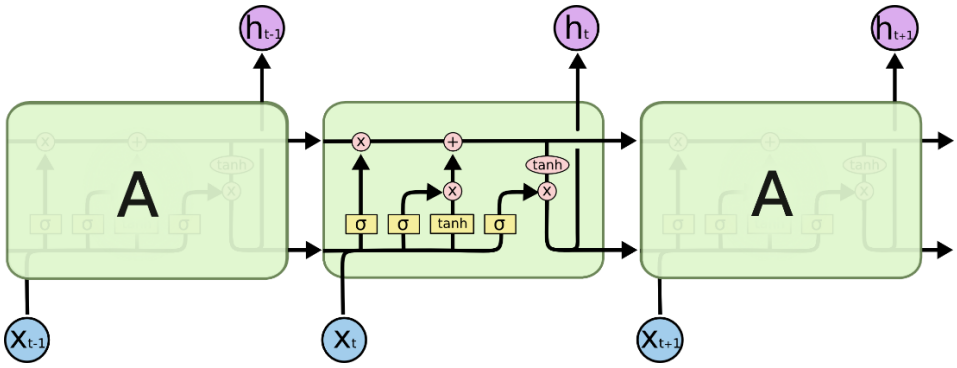
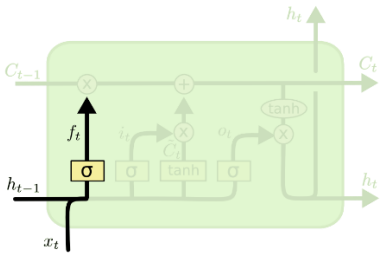


Figura 2. Esquema LSTM. (Olah, 2015)

Los mecanismos encargados de regular el flujo de datos del cell state son llamados compuertas. Estas compuertas añaden o remueven información. Dichas compuertas aprenden qué información modificar durante el entrenamiento.

Al igual que una RNN, las LSTMs poseen a grandes rasgos una capa con la función de activación tanh, pero adicional a esto, cuenta con tres compuertas explicadas a continuación.

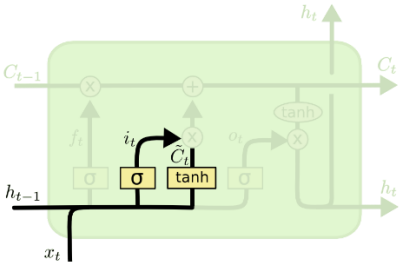
La primera compuerta es llamada “Forget Gate”, la cual decide qué información es retirada del cell state (C_t). Esta compuerta pasa el estado oculto anterior (h_{t-1}) y la entrada actual (x_t) a través de una función de activación sigmoide. La salida (f_t) de esta compuerta son valores entre 0 y 1, donde tender a 0 significa olvidar y tender a 1 significa conservar.



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

Figura 3. Forget Gate. (Olah, 2015)

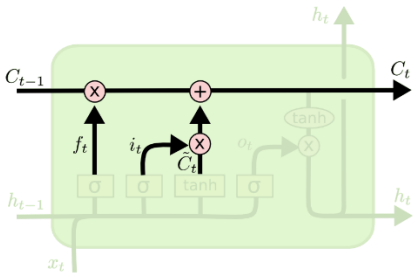
La compuerta que sigue es llamada “Input Gate”, la cual se encarga de actualizar el cell state. Para esto primero el estado oculto anterior (h_{t-1}) y la entrada actual (x_t) pasan por una función sigmoide que arroja valores entre 0 y 1, donde 0 significa sin importancia y 1 significa que es importante. Luego h_{t-1} y x_t pasan por una función tanh que crea un vector de valores candidatos entre -1 y 1 para regular la red. Finalmente, la salida de la función sigmoide (i_t) y de la función tanh ($\bar{C}_t$) son multiplicadas entre sí obteniendo $i_t * \bar{C}_t$ que son los datos a almacenar en el cell state.



$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

Figura 4. Input Gate. (Olah, 2015)

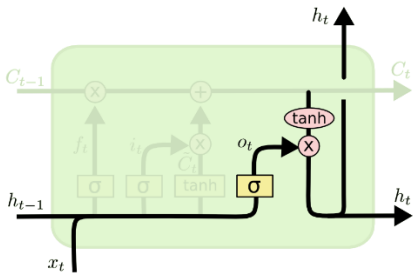
Con la información de las compuertas mencionadas anteriormente se conocen los cambios que se harán al cell state. Ahora, para actualizar el estado C_{t-1} a C_t se multiplica el estado anterior con la salida f_t, eliminando la información que se decidió olvidar. Luego se suma $i_t * \bar{C}_t$, lo que adiciona la nueva información al cell state.



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Figura 5. Cell State. (Olah, 2015)

La última compuerta es llamada "Output Gate", esta decidirá cuál será el estado oculto h_t. La salida de esta compuerta será una versión filtrada del cell state. Para esto se pasan una vez más h_{t-1} y x_t a través de una función sigmoide, su salida (O_t) decide qué partes del cell state saldrán. Luego el cell state actualizado anteriormente $C_t$es pasado por una función tanh (convierte a valores entre -1 y 1) para finalmente ser multiplicado con O_t. El estado oculto será información relevante para una tarea determinada para la que se esté usando la LSTM. (Olah, 2015).



$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

Figura 6. Output Gate. (Olah, 2015)

3. METODOLOGÍA

3.1 Base de datos

Para el desarrollo del presente producto de laboratorio se usaron datos de la red de bajo costo desarrollada por el SIATA llamada “ciudadanos científicos”, en la cual cada nodo de la red contiene un sensor de calidad del aire de bajo costo que permite obtener mediciones puntuales, cada minuto, de temperatura, humedad relativa y material particulado PM2.5.

Para construir la base de datos se toman solo las mediciones de material particulado PM2.5 durante el año 2008 y se agrupan los datos por cada hora en lugar de minuto a minuto.

Cabe resaltar que los datos no se encuentran completos, esto debido a problemas de suministro energético, comportamientos inadecuados del sensor, fallas en la conectividad a internet y otros problemas que surgen debido a la naturaleza de los sensores. Además, se eliminaron los nodos de sensores que no presentaron mediciones durante el año. Lo anterior debido a que no aportan información valiosa a la red y terminan generando ruido a la hora de aplicar el algoritmo de inteligencia artificial.

La base de datos consta de 130 sensores, en donde cada sensor cuenta con 8762 horas de medición.

3.2 Elección del sensor objetivo

A la hora de elegir el target sensor o sensor objetivo, se tuvieron en cuenta un par de consideraciones:

- El sensor objetivo debe tener la menor cantidad de datos faltantes.
- Los vecinos o sensores cercanos al sensor objetivo deben haber estado activos durante el año 2008.

Siguiendo dichas consideraciones se elige el sensor 14 (posición 22 en la base de datos), el cual cuenta con alrededor de 5% de datos faltantes (444). Dicho sensor se encuentra

ubicado en la comuna 13 de Medellín, concretamente en las coordenadas Latitud: 6.25449 Longitud: -75.62582.

Uno de los objetivos es predecir mediciones para el sensor objetivo a partir de sensores vecinos, esto quiere decir, sensores que se encuentren relativamente cerca a dicho sensor objetivo.

Los vecinos del sensor 14 son los sensores 204, 39, 37 y 108 tal y como se evidencia en la Figura 8.

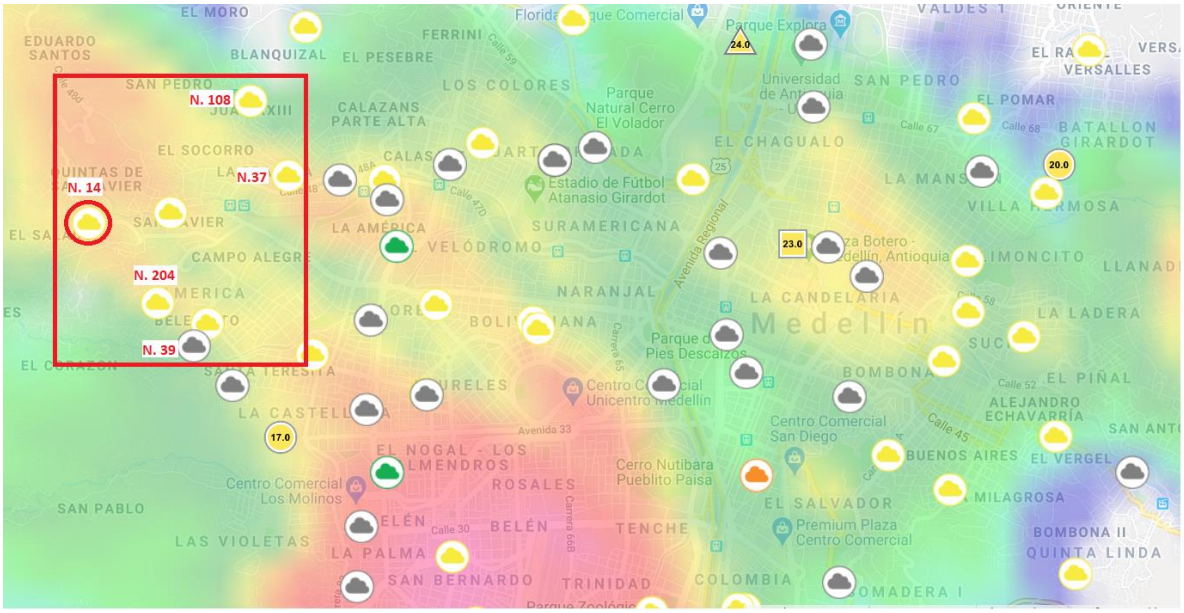


Figura 7. Geo portal SIATA, red Ciudadanos Científicos.

3.3 Implementación de red LSTM

Se realiza la implementación de una red LSTM en la plataforma Kaggle debido a que esta ofrece la opción de activar una GPU.

Se busca un horizonte de predicción de una hora a partir de una ventana deslizante de cinco pasos hacia el pasado (5 horas anteriores). El pronóstico es para una semana (360 horas).

Se busca predecir los datos de tres formas diferentes:

- Entrenando la LSTM con las mediciones anteriores del sensor objetivo.
- Entrenando la LSTM usando todos los sensores de la red.

- Entrenando la LSTM usando los sensores vecinos del sensor objetivo y el propio sensor objetivo.

Algunos parámetros importantes en la implementación de la red según los objetivos son:

- ***Test_size = 0.041:*** Se desea un pronóstico de alrededor de 360 horas, por lo tanto, se elige comparar las predicciones con 4.1% del total de las mediciones.
- ***Train_size = 0.959:*** El porcentaje restante de datos sería 95.5%, con estos datos se entrenará la LSTM.
- ***Sensor_idx:*** Este parámetro es el sensor objetivo, puede variar según cómo se esté realizando la predicción. Si se desea predecir a partir de las mediciones de la red completa se usa la posición del sensor en la base de datos (22). Si se desea predecir a partir de los datos del propio sensor, se usa el número cero, ya que este será el único sensor que reciba la red. Finalmente, si se desea predecir a partir de los datos de los sensores vecinos, entonces se usará la posición en la que se encuentre el sensor objetivo en el vector de sensores de entrada.
- ***Tw = 5:*** Serán el número de pasos de tiempo. Esto significa que se hará la predicción de la medición en una determinada hora, a partir de las mediciones obtenidas en 5 horas anteriores.
- ***n_features:*** Será el número de sensores que ingresan a la red para realizar la predicción.

Otros parámetros como ***batch_size***, ***train_episodes***, ***lr*** (tasa de aprendizaje), fueron elegidos de manera empírica, ya que, estos se fueron ajustando gradualmente según se obtenían mejores resultados.

3.4 Cálculos de error

Para comparar el desempeño de las tres formas de realizar las predicciones se usa el **MAPE**, el **MAE** y el **RMSE**.

Primero se define el error como el pronóstico o predicción (f_t) menos la demanda o resultado esperado (d_t), ver ecuación (1).

Según la definición anterior, si la predicción sobrepasa al resultado esperado entonces se obtendrá un error positivo y si la predicción es inferior al resultado esperado entonces se obtendrá un error negativo. (Vandeput, 2019)

$$e_t = f_t - d_t \quad (1)$$

3.4.1 MAPE (Error porcentual absoluto medio)

Es la suma de los errores absolutos individuales divididos por su respectivo resultado esperado, ver ecuación (2) donde n es el número de periodos. (Vandeput, 2019).

$$MAPE = \frac{1}{n}\sum \frac{|e_t|}{d_t} \quad (2)$$

3.4.2 MAE (Error medio absoluto)

Es la media de los errores absolutos, donde el valor ideal es 0.0, ver ecuación (3). (Vandeput, 2019).

$$MAE = \frac{1}{n}\sum |e_t| \quad (3)$$

3.4.3 RMSE (Error cuadrático medio)

Mide la cantidad de error que existe entre dos conjuntos de datos. Compara la predicción con lo esperado. El valor ideal es 0.0, ver ecuación (4). (Vandeput, 2019).

$$RMSE = \sqrt{\frac{1}{n}\sum e_t^2} \quad (4)$$

4. RESULTADOS Y DISCUSIÓN

4.1 Código en Python

Se importan los diferentes módulos y paquetes de librerías que se usarán posteriormente

```
import pandas as pd
import numpy as np
from matplotlib import pyplot as plt
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split
import torch.nn.functional as F
import tqdm
import torch
import torch.nn as nn
from torch.utils.data import TensorDataset, DataLoader

from sklearn import metrics
```

Se lee el DataSet Aire.csv usando pandas.

```
df = pd.read_csv('../input/pm25-2018-missingdata/Aire.csv')
df.info()
```

A continuación, hay tres diferentes selecciones de datos *X.* Cabe resaltar, que solo se usa una de las tres selecciones por vez según como se haya decidido hacer las predicciones:

Selección de datos para predicción con datos anteriores del sensor objetivo.

```
X = df.iloc[22,1:].values.T

X = np.expand_dims(X,1)
```

Selección de datos para predicción con datos de todos los sensores de la red.

```
X = df.iloc[:,1:].values.T
```

Selección de datos para predicción con datos de los sensores vecinos del sensor objetivo.

```
X = df.iloc[[22,58,93,92,4],1:].values.T
```

Conversión de datos a tipo float64.

```
X.astype("float64")
print("X Shape: ", X.shape, "\nX type:", type(X))
```

Selección del porcentaje de datos que se usarán para el entrenamiento (95.9%) y la validación (4.1%).

```
Xtrain, Xtest = train_test_split(X, train_size = 0.959, test_size = 0.041,
                                 shuffle = False)
Xtr, Xval = train_test_split(Xtrain,train_size=0.959, test_size=0.041,
                             shuffle=False)
print(Xtrain.shape, Xtest.shape)
print(Xtr.shape, Xval.shape)
```

Índice del sensor al que se le busca hacer la predicción. Como se explica en la metodología; este índice varía según la forma en cómo se esté haciendo la predicción:

- Si se desea predecir a partir de las mediciones de la red completa se usa la posición del sensor en la base de datos (22).
- Si se desea predecir a partir de los datos del propio sensor, se usa el número cero, ya que este será el único sensor que reciba la red.
- Si se desea predecir a partir de los datos de los sensores vecinos, entonces se usará la posición en la que se encuentre el sensor objetivo en el vector de sensores de entrada.

```
sensor_idx = 0
```

Normalización de los datos usando *MinMaxScaler*. Esto es reescalar los datos al rango de 0 a 1 ya que las LSTMs son sensibles a la escala del input.

```
scal_tr = MinMaxScaler()
scal_tr.fit(Xtr[:,sensor_idx].reshape(-1,1))

scal = MinMaxScaler()
Xtr = scal.fit_transform(Xtr)
Xval = scal.transform(Xval)
Xtest = scal.transform(Xtest)
```

Se define la función *create_sequences*, la cual tiene como parámetros de entrada la matriz de sensores (*matrix*), el índice del sensor objetivo (*targ_index*) y los puntos de la ventana deslizante hacia el pasado (*Time_Steps*). Dicha función retorna una secuencia de 5 (en este caso) datos anteriores y una etiqueta del valor real que el modelo debería predecir.

```
def create_sequences(matrix, targ_index, Time_Steps):

    dim_0 = matrix.shape[0] - Time_Steps
    dim_1 = matrix.shape[1]

    x = np.zeros((dim_0, Time_Steps,dim_1))
    y = np.zeros((dim_0,1))

    for i in tqdm.notebook.tqdm(range(dim_0)):

        aux = matrix[i : Time_Steps + i, :]
        x[i,:,:] = aux
        y[i,:] = matrix[Time_Steps+i, targ_index]

    print("length of time-series i/o",x.shape,y.shape)

    return x, y
```

Se establece el *Time_Steps* como y se crean las series de tiempo usando la función *create_sequences*.

```
tw = 5

Xtr, ytr = create_sequences(Xtr, sensor_idx,tw)
Xval, yval = create_sequences(Xval, sensor_idx,tw)
Xtest, ytest = create_sequences(Xtest, sensor_idx,tw)
```

Creación de paquetes de entrenamiento (*train_loader*) y de validación (*val_loader*).

```
batch_size = 24

train_data = TensorDataset(torch.from_numpy(Xtr), torch.from_numpy(ytr))
train_loader = DataLoader(train_data, shuffle = True, batch_size = batch_size,
                          drop_last = True)

val_data = TensorDataset(torch.from_numpy(Xval), torch.from_numpy(yval))
val_loader = DataLoader(val_data, shuffle = True, batch_size = batch_size,
                        drop_last = True)
```

Se hace uso de una GPU si se dispone de ella, de lo contrario se usará la CPU.

```
# torch.cuda.is_available() Comprueba si es posible usar GPU; Si es posible retorna True, de
lo contrario retorna False.
is_cuda = torch.cuda.is_available()


if is_cuda:
    device = torch.device("cuda")
else:
    device = torch.device("cpu")
```

Se define la arquitectura de la red LSTM como tal.

```
class LSTMNet(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim, n_layers, drop_prob=0.2):
        super(LSTMNet, self).__init__()
        self.hidden_dim = hidden_dim
        self.n_layers = n_layers

        self.lstm = nn.LSTM(input_dim, hidden_dim, n_layers, batch_first=True, dropout=dro
p_prob)
        self.fc = nn.Linear(hidden_dim, output_dim)
        self.relu = nn.ReLU()

    def forward(self, x, h):
        out, h = self.lstm(x, h)
        out = self.fc(self.relu(out[:,-1]))
        return out, h

    def init_hidden(self, batch_size):
        weight = next(self.parameters()).data
        hidden = (weight.new(self.n_layers, batch_size, self.hidden_dim).zero_().to(devic
e),
                  weight.new(self.n_layers, batch_size, self.hidden_dim).zero_().to(devic
e))
        return hidden
```

Se establecen algunas características de la red LSTM y se establece como modelo dicha red.

```
n_features = Xtr.shape[2] # Carácterísticas que se usarán para la predicción. En este caso e
s la cantidad de sensores que se usan en la predicción.
n_timesteps = tw # Es el número del "Time_Steps".
hidden_dim = 256
output_dim = 1 # Se desea predecir las mediciones para un solo sensor.
n_layers = 2


model = LSTMNet(n_features,hidden_dim,output_dim,n_layers)
model.to(device)

criterion = torch.nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

train_episodes = 60
```

Se entrena el modelo previamente creado.

```
model.train()
for t in range(train_episodes):
    htr = model.init_hidden(batch_size)
    for x,y in train_loader:
        htr = tuple([e.data for e in htr])
        model.zero_grad()
        output, htr = model(x.to(device).float(),htr)
        loss = criterion(output, y.to(device).float())
        #Calcular gradiente
        loss.backward()
        #Mover los datos
        optimizer.step()

    hval = model.init_hidden(batch_size)
    for xval,yval in val_loader:
        hval = tuple([e.data for e in hval])
        output_val, hval = model(xval.to(device).float(),htr)
        loss_val = criterion(output_val, yval.to(device).float())




    print('step : ' , t , ' loss_train: ' , loss.item(), ' loss_val: ', loss_val.item())
```

Se evalúan las predicciones (*out*) contra los datos reales (*ytest*).

```
#Evaluate the model
model.eval()
#Xtest = torch.from_numpy(Xtest[:,sensor_idx])
Xtest = torch.from_numpy(Xtest)
htest = model.init_hidden(Xtest.shape[0])
out, htest = model(Xtest.to(device).float(), htest)

out = out.cpu().detach().numpy()

out = scal_tr.inverse_transform(out)
ytest = scal_tr.inverse_transform(ytest)
```

Se grafican las predicciones contra los datos reales.

```
fig = plt.figure()

ax11 = fig.add_subplot(211)
ax11.plot(ytest, 'r', label='Real')

ax11.set_title('Mediciones reales vs Mediciones predichas ',fontweight='bold')
ax11.set_ylabel('PM2.5 [ug/m^3]')
ax11.set_xlabel('Tiempo [h]')
ax11.plot(out, label='Predicción')
ax11.legend(loc=(0,0))
plt.grid()
#plt.savefig('Prediccion_vecinos.pdf')

plt.show()
```

4.2 Predicción de mediciones entrenando la LSTM con mediciones anteriores del sensor objetivo

Tras entrenar la red LSTM usando mediciones anteriores del sensor objetivo, se obtiene la gráfica de la figura 8. En dicha figura se observa en azul la gráfica de las predicciones durante casi 360 horas (una semana) superpuesta a los datos reales del sensor en rojo.

Cabe resaltar que a pesar de que el sensor objetivo elegido tenía pocos datos faltantes, no fue suficiente para lograr que en ciertas mediciones (como en la hora 300 en la que hay un dato faltante), la red fuera capaz de atenuar de manera correcta la predicción, la cual no llega a cero, pero sí baja considerablemente.

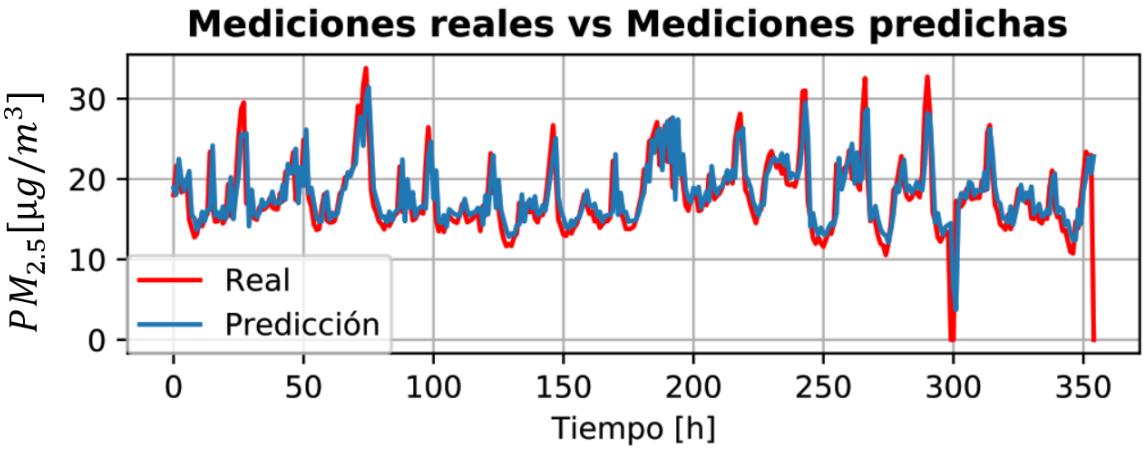


Figura 8. Gráfica de resultados entrenando red usando mediciones anteriores del sensor objetivo.

4.3 Predicción de mediciones entrenando la LSTM con todos los sensores de la red

Al entrenar la red LSTM con mediciones de todos los sensores de la red, se obtiene la gráfica de la figura 9. En ella se observa en azul la gráfica de las predicciones hechas nuevamente superpuesta a las mediciones reales del sensor objetivo durante casi 360 horas.

Se evidencia que la predicción usando toda la red de sensores permite una gran atenuación a la hora de predecir mediciones teniendo datos faltantes (cero). Pero, si bien lo anterior es un aspecto positivo, también se evidencia que los datos predichos se alejan considerablemente de la realidad. Esto, debido a que no todos los sensores de la red se encuentran ubicados en las mismas zonas y, por ende, no presentan mediciones congruentes a las del sensor objetivo.

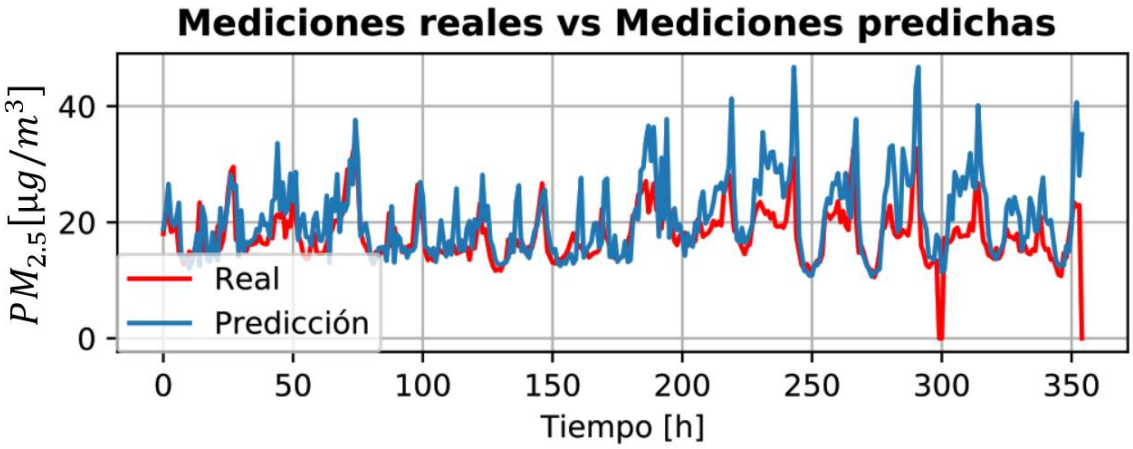


Figura 9. Gráfica de resultados entrenando la red usando mediciones de toda la red de sensores.

4.4 Predicción de mediciones entrenando la LSTM con sensores vecinos del sensor objetivo

Al entrenar la red LSTM usando sensores que, por su ubicación cercana a la del sensor objetivo, tendrán mediciones mucho más congruentes que los demás sensores de la red; se obtiene la gráfica de la figura 10. En la cual nuevamente se tiene en azul la gráfica de las mediciones predichas superpuestas a las mediciones reales durante 360 horas.

Se puede observar cómo a pesar de que no se logra atenuar en su totalidad la predicción de una medición faltante (0), se observa una mejor predicción para dicha medición a comparación de la obtenida en la figura 8.

Además, como las mediciones de los sensores vecinos son congruentes con las del sensor objetivo, se evidencia una predicción mucho más acertada que usando todos los sensores de la red.

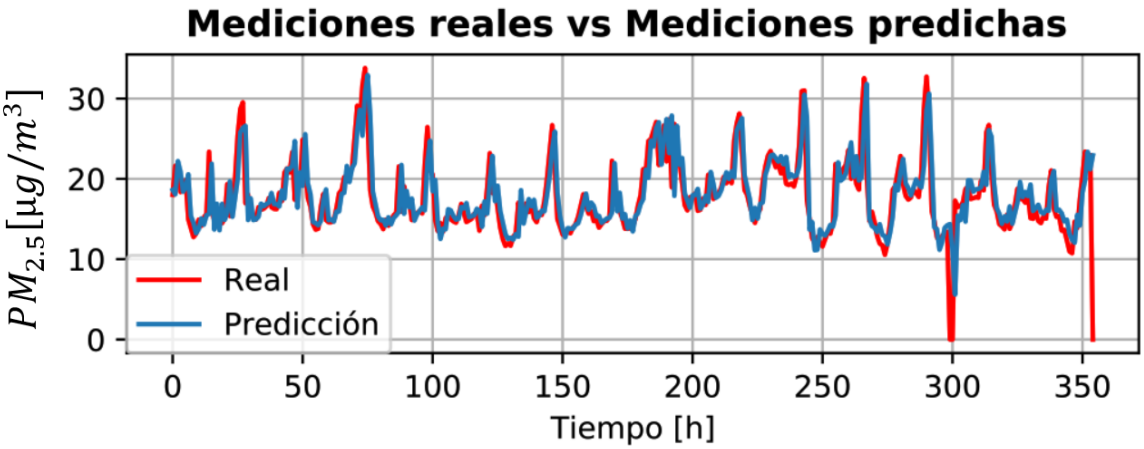


Figura 10. Gráfica de resultados entrenando la red usando mediciones de los sensores vecinos del sensor objetivo.

4.5 Cálculo y resultados de errores

En las mediciones reales del sensor objetivo durante las 360 horas que se muestran en las figuras 8, 9 y 10 se pueden observar algunos datos faltantes (mediciones que tienden a cero). Por lo tanto, se retiran los datos de las horas 299, 300, y 354 tanto de las mediciones reales como de las predicciones para que esto no afecte los cálculos de errores.

```
ytest_f = np.delete(ytest,[299,300,354])
out_f = np.delete(out, [299,300,354])
```

MAE:

Entrega un valor flotante, el cual, a medida que tiende a 0,0 es mejor.

```
from sklearn.metrics import mean_absolute_error
#Best value is 0.0

MAE = mean_absolute_error(ytest_f, out_f)
print("MAE: ",MAE)
```

MAPE:

Entrega un resultado en porcentaje.

```
def percentage_error(actual, predicted):
    res = np.empty(actual.shape)
    for j in range(actual.shape[0]):
        if actual[j] != 0:
            res[j] = (actual[j] - predicted[j]) / actual[j]
        else:
            res[j] = predicted[j] / np.mean(actual)
    return res

def mean_absolute_percentage_error(y_true, y_pred):
    return np.mean(np.abs(percentage_error(np.asarray(y_true), np.asarray(y_pred)))) * 100

MAPE = mean_absolute_percentage_error(ytest_f, out_f)
print("MAPE:", MAPE)
```

RMSE:

Entrega un valor flotante, el cual, a medida que tiende a 0,0 es mejor.

```
from sklearn.metrics import mean_squared_error

#squaredboolean value, optional (default = True), If True returns MSE value, if False return
s RMSE value.
#Best value is 0.0

RMSE = mean_squared_error(ytest_f, out_f, squared=False)
print("RMSE: ",RMSE)
```

En la tabla 1 se observan los resultados de tres diferentes cálculos de error para las diferentes formas en las que se realizaron las predicciones. Gracias a dichos errores, se evidencia que las predicciones con mayor similitud a las mediciones reales se obtienen entrenando la red usando mediciones pasadas del sensor objetivo.

Tabla 1. Cálculos de diferentes errores para cada forma de realizar la predicción.

Entrenamientos	MAE	MAPE (%)	RMSE
Mediciones anteriores del sensor	1.8769	9.9034	2.7404
Mediciones de toda la red de sensores	5.7929	32.4325	7.6658
Mediciones de vecinos del sensor objetivo	2.0088	11.0647	2.8303

5. CONCLUSIONES, RECOMENDACIONES Y TRABAJO FUTURO

Se logra la predicción de mediciones durante 360 horas para un sensor objetivo que hace parte de la red de sensores de bajo costo “ciudadanos científicos”. La predicción se realizó de tres formas: la primera entrenando la red con mediciones anteriores del mismo sensor objetivo, la segunda entrenando la red con mediciones de todos los sensores que hacen parte de la red de sensores, y la tercer entrenando la red con mediciones de sensores denominados “vecinos”, los cuales por su ubicación presentan medidas similares al sensor objetivo.

Las mejores predicciones se obtuvieron haciendo uso de los datos anteriores del sensor objetivo. Este resultado es esperado, ya que, la base de datos presenta numerosos datos faltantes y no permite aumentar la precisión de las predicciones mediante el uso de mediciones de otros sensores de la red. Por ende, como trabajo futuro se espera implementar un método que permita completar información faltante para disminuir el error en el pronóstico y además, ampliar el horizonte de predicción de una hora a algo mayor.

REFERENCIAS

Antonio, J. L. (19 de Octrube de 2016). *Linkedin.* Obtenido de La Inteligencia Artificial ha llegado: Las redes RNNs: https://www.linkedin.com/pulse/la-inteligencia-artificial-ha-llegado-las-redes-rnns-rivero-antonio

Olah, C. (27 de Agosto de 2015). *Understanding LSTM Networks.* Obtenido de Colah's Blog: https://colah.github.io/

Phi, M. (24 de Septiembre de 2018). *Towards Data Science.* Obtenido de Illustrated Guide to LSTM's and GRU's: A step by step explanation: https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21

Sepp Hochreiter, J. S. (15 de Noviembre de 1997). Long Short-Term Memory. *Neural Computation, 9*(8), 1735-1780.

Vandeput, N. (5 de Julio de 2019). *Forecast KPI: RMSE, MAE, MAPE & Bias.* Obtenido de Medium / Analytics Vidhya: https://medium.com/analytics-vidhya/forecast-kpi-rmse-mae-mape-bias-cdc5703d242d

APÉNDICE

Apéndice A

Código implementación de la LSTM realizando la predicción con datos anteriores del sensor objetivo

```
import pandas as pd
import numpy as np
from matplotlib import pyplot as plt
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split
import torch.nn.functional as F
import tqdm
import torch
import torch.nn as nn
from torch.utils.data import TensorDataset, DataLoader
from sklearn import metrics


df = pd.read_csv('../input/pm25-2018-missingdata/Aire.csv')
df.info()

#Selección de datos para predicción con datos anteriores del sensor
objetivo.
X = df.iloc[22,1:].values.T
X = np.expand_dims(X,1)


X.astype("float64")
print("X Shape: ", X.shape, "\nX type:", type(X))

Xtrain, Xtest = train_test_split(X, train_size = 0.959, test_size =
0.041,
                                 shuffle = False)
Xtr, Xval = train_test_split(Xtrain,train_size=0.959, test_size=0.041,
                             shuffle=False)
print(Xtrain.shape, Xtest.shape)
print(Xtr.shape, Xval.shape)


sensor_idx = 0


scal_tr = MinMaxScaler()
scal_tr.fit(Xtr[:,sensor_idx].reshape(-1,1))

scal = MinMaxScaler()
```

```
Xtr = scal.fit_transform(Xtr)
Xval = scal.transform(Xval)
Xtest = scal.transform(Xtest)


def create_sequences(matrix, targ_index, Time_Steps):

    dim_0 = matrix.shape[0] - Time_Steps
    dim_1 = matrix.shape[1]

    x = np.zeros((dim_0, Time_Steps,dim_1))
    y = np.zeros((dim_0,1))

    for i in tqdm.notebook.tqdm(range(dim_0)):

        aux = matrix[i : Time_Steps + i, :]
        x[i,:,:] = aux
        y[i,:] = matrix[Time_Steps+i, targ_index]

    print("length of time-series i/o",x.shape,y.shape)

    return x, y


tw = 5

Xtr, ytr = create_sequences(Xtr, sensor_idx,tw)
Xval, yval = create_sequences(Xval, sensor_idx,tw)
Xtest, ytest = create_sequences(Xtest, sensor_idx,tw)


batch_size = 24

train_data = TensorDataset(torch.from_numpy(Xtr), torch.from_numpy(ytr))
train_loader = DataLoader(train_data, shuffle = True, batch_size =
batch_size,
                          drop_last = True)

val_data = TensorDataset(torch.from_numpy(Xval), torch.from_numpy(yval))
val_loader = DataLoader(val_data, shuffle = True, batch_size =
batch_size,
                        drop_last = True)


is_cuda = torch.cuda.is_available()

if is_cuda:
    device = torch.device("cuda")
else:
    device = torch.device("cpu")


class LSTMNet(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim, n_layers,
drop_prob=0.2):
```

```
        super(LSTMNet, self).__init__()
        self.hidden_dim = hidden_dim
        self.n_layers = n_layers

        self.lstm = nn.LSTM(input_dim, hidden_dim, n_layers,
batch_first=True, dropout=drop_prob)
        self.fc = nn.Linear(hidden_dim, output_dim)
        self.relu = nn.ReLU()

    def forward(self, x, h):
        out, h = self.lstm(x, h)
        out = self.fc(self.relu(out[:,-1]))
        return out, h

    def init_hidden(self, batch_size):
        weight = next(self.parameters()).data
        hidden = (weight.new(self.n_layers, batch_size,
self.hidden_dim).zero_().to(device),
                  weight.new(self.n_layers, batch_size,
self.hidden_dim).zero_().to(device))
        return hidden



n_features = Xtr.shape[2]
n_timesteps = tw
hidden_dim = 256
output_dim = 1
n_layers = 2


model = LSTMNet(n_features,hidden_dim,output_dim,n_layers)
model.to(device)

criterion = torch.nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

train_episodes = 60



model.train()
for t in range(train_episodes):
    htr = model.init_hidden(batch_size)
    for x,y in train_loader:
        htr = tuple([e.data for e in htr])
        model.zero_grad()
        output, htr = model(x.to(device).float(),htr)
        loss = criterion(output, y.to(device).float())
        loss.backward()
        optimizer.step()

    hval = model.init_hidden(batch_size)
    for xval,yval in val_loader:
        hval = tuple([e.data for e in hval])
```

```
        output_val, hval = model(xval.to(device).float(),htr)
        loss_val = criterion(output_val, yval.to(device).float())

    print('step : ' , t , ' loss_train: ' , loss.item(), ' loss_val: ',
loss_val.item())


model.eval()
Xtest = torch.from_numpy(Xtest)
htest = model.init_hidden(Xtest.shape[0])
out, htest = model(Xtest.to(device).float(), htest)

out = out.cpu().detach().numpy()

out = scal_tr.inverse_transform(out)
ytest = scal_tr.inverse_transform(ytest)



fig = plt.figure()

ax11 = fig.add_subplot(211)
ax11.plot(ytest, 'r', label='Real')

ax11.set_title('Mediciones reales vs Mediciones predichas
',fontweight='bold')
ax11.set_ylabel('PM2.5 [ug/m^3]')
ax11.set_xlabel('Tiempo [h]')
ax11.plot(out, label='Predicción')
ax11.legend(loc=(0,0))
plt.grid()

plt.show()


ytest_f = np.delete(ytest,[299,300,354])
out_f = np.delete(out, [299,300,354])



from sklearn.metrics import mean_absolute_error
MAE = mean_absolute_error(ytest_f, out_f)
print("MAE: ",MAE)



def percentage_error(actual, predicted):
    res = np.empty(actual.shape)
    for j in range(actual.shape[0]):
        if actual[j] != 0:
            res[j] = (actual[j] - predicted[j]) / actual[j]
        else:
            res[j] = predicted[j] / np.mean(actual)
    return res
```

```
def mean_absolute_percentage_error(y_true, y_pred):
    return np.mean(np.abs(percentage_error(np.asarray(y_true),
np.asarray(y_pred)))) * 100

MAPE = mean_absolute_percentage_error(ytest_f, out_f)
print("MAPE:", MAPE)



from sklearn.metrics import mean_squared_error

RMSE = mean_squared_error(ytest_f, out_f, squared=False)
print("RMSE: ",RMSE)
```

Código implementación de la LSTM realizando la predicción con datos de toda la red de sensores

```
import pandas as pd
import numpy as np
from matplotlib import pyplot as plt
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split
import torch.nn.functional as F
import tqdm
import torch
import torch.nn as nn
from torch.utils.data import TensorDataset, DataLoader
from sklearn import metrics


df = pd.read_csv('../input/pm25-2018-missingdata/Aire.csv')
df.info()

#Selección de datos para predicción con datos de todos los sensores de la
red
X = df.iloc[:,1:].values.T


X.astype("float64")
print("X Shape: ", X.shape, "\nX type:", type(X))

Xtrain, Xtest = train_test_split(X, train_size = 0.959, test_size =
0.041,
                                 shuffle = False)
Xtr, Xval = train_test_split(Xtrain,train_size=0.959, test_size=0.041,
                             shuffle=False)
print(Xtrain.shape, Xtest.shape)
print(Xtr.shape, Xval.shape)


sensor_idx = 22
```

```
scal_tr = MinMaxScaler()
scal_tr.fit(Xtr[:,sensor_idx].reshape(-1,1))

scal = MinMaxScaler()
Xtr = scal.fit_transform(Xtr)
Xval = scal.transform(Xval)
Xtest = scal.transform(Xtest)


def create_sequences(matrix, targ_index, Time_Steps):

    dim_0 = matrix.shape[0] - Time_Steps
    dim_1 = matrix.shape[1]

    x = np.zeros((dim_0, Time_Steps,dim_1))
    y = np.zeros((dim_0,1))

    for i in tqdm.notebook.tqdm(range(dim_0)):

        aux = matrix[i : Time_Steps + i, :]
        x[i,:,:] = aux
        y[i,:] = matrix[Time_Steps+i, targ_index]

    print("length of time-series i/o",x.shape,y.shape)

    return x, y


tw = 5

Xtr, ytr = create_sequences(Xtr, sensor_idx,tw)
Xval, yval = create_sequences(Xval, sensor_idx,tw)
Xtest, ytest = create_sequences(Xtest, sensor_idx,tw)


batch_size = 24

train_data = TensorDataset(torch.from_numpy(Xtr), torch.from_numpy(ytr))
train_loader = DataLoader(train_data, shuffle = True, batch_size =
batch_size,
                          drop_last = True)

val_data = TensorDataset(torch.from_numpy(Xval), torch.from_numpy(yval))
val_loader = DataLoader(val_data, shuffle = True, batch_size =
batch_size,
                        drop_last = True)


is_cuda = torch.cuda.is_available()

if is_cuda:
    device = torch.device("cuda")
else:
```

```
    device = torch.device("cpu")


class LSTMNet(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim, n_layers,
drop_prob=0.2):
        super(LSTMNet, self).__init__()
        self.hidden_dim = hidden_dim
        self.n_layers = n_layers

        self.lstm = nn.LSTM(input_dim, hidden_dim, n_layers,
batch_first=True, dropout=drop_prob)
        self.fc = nn.Linear(hidden_dim, output_dim)
        self.relu = nn.ReLU()

    def forward(self, x, h):
        out, h = self.lstm(x, h)
        out = self.fc(self.relu(out[:,-1]))
        return out, h

    def init_hidden(self, batch_size):
        weight = next(self.parameters()).data
        hidden = (weight.new(self.n_layers, batch_size,
self.hidden_dim).zero_().to(device),
                  weight.new(self.n_layers, batch_size,
self.hidden_dim).zero_().to(device))
        return hidden



n_features = Xtr.shape[2]
n_timesteps = tw
hidden_dim = 256
output_dim = 1
n_layers = 2


model = LSTMNet(n_features,hidden_dim,output_dim,n_layers)
model.to(device)

criterion = torch.nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

train_episodes = 60



model.train()
for t in range(train_episodes):
    htr = model.init_hidden(batch_size)
    for x,y in train_loader:
        htr = tuple([e.data for e in htr])
        model.zero_grad()
        output, htr = model(x.to(device).float(),htr)
        loss = criterion(output, y.to(device).float())
```

```
            loss.backward()
            optimizer.step()

        hval = model.init_hidden(batch_size)
        for xval,yval in val_loader:
            hval = tuple([e.data for e in hval])
            output_val, hval = model(xval.to(device).float(),htr)
            loss_val = criterion(output_val, yval.to(device).float())

        print('step : ' , t , ' loss_train: ' , loss.item(), ' loss_val: ',
loss_val.item())


model.eval()
Xtest = torch.from_numpy(Xtest)
htest = model.init_hidden(Xtest.shape[0])
out, htest = model(Xtest.to(device).float(), htest)

out = out.cpu().detach().numpy()

out = scal_tr.inverse_transform(out)
ytest = scal_tr.inverse_transform(ytest)



fig = plt.figure()

ax11 = fig.add_subplot(211)
ax11.plot(ytest, 'r', label='Real')

ax11.set_title('Mediciones reales vs Mediciones predichas
',fontweight='bold')
ax11.set_ylabel('PM2.5 [ug/m^3]')
ax11.set_xlabel('Tiempo [h]')
ax11.plot(out, label='Predicción')
ax11.legend(loc=(0,0))
plt.grid()

plt.show()


ytest_f = np.delete(ytest,[299,300,354])
out_f = np.delete(out, [299,300,354])



from sklearn.metrics import mean_absolute_error
MAE = mean_absolute_error(ytest_f, out_f)
print("MAE: ",MAE)



def percentage_error(actual, predicted):
    res = np.empty(actual.shape)
    for j in range(actual.shape[0]):
```

```
        if actual[j] != 0:
            res[j] = (actual[j] - predicted[j]) / actual[j]
        else:
            res[j] = predicted[j] / np.mean(actual)
    return res

def mean_absolute_percentage_error(y_true, y_pred):
    return np.mean(np.abs(percentage_error(np.asarray(y_true),
np.asarray(y_pred)))) * 100

MAPE = mean_absolute_percentage_error(ytest_f, out_f)
print("MAPE:", MAPE)



from sklearn.metrics import mean_squared_error

RMSE = mean_squared_error(ytest_f, out_f, squared=False)
print("RMSE: ",RMSE)
```

Código implementación de la LSTM realizando la predicción con datos de los vecinos sel sensor objetivo

```
import pandas as pd
import numpy as np
from matplotlib import pyplot as plt
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split
import torch.nn.functional as F
import tqdm
import torch
import torch.nn as nn
from torch.utils.data import TensorDataset, DataLoader
from sklearn import metrics


df = pd.read_csv('../input/pm25-2018-missingdata/Aire.csv')
df.info()

#Selección de datos para predicción con datos de los vecinos
X = df.iloc[[22,58,93,92,4],1:].values.T


X.astype("float64")
print("X Shape: ", X.shape, "\nX type:", type(X))

Xtrain, Xtest = train_test_split(X, train_size = 0.959, test_size =
0.041,
                                 shuffle = False)
Xtr, Xval = train_test_split(Xtrain,train_size=0.959, test_size=0.041,
                             shuffle=False)
```

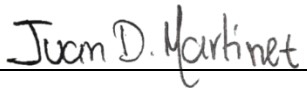
```
print(Xtrain.shape, Xtest.shape)
print(Xtr.shape, Xval.shape)


sensor_idx = 0


scal_tr = MinMaxScaler()
scal_tr.fit(Xtr[:,sensor_idx].reshape(-1,1))

scal = MinMaxScaler()
Xtr = scal.fit_transform(Xtr)
Xval = scal.transform(Xval)
Xtest = scal.transform(Xtest)


def create_sequences(matrix, targ_index, Time_Steps):

    dim_0 = matrix.shape[0] - Time_Steps
    dim_1 = matrix.shape[1]

    x = np.zeros((dim_0, Time_Steps,dim_1))
    y = np.zeros((dim_0,1))

    for i in tqdm.notebook.tqdm(range(dim_0)):

        aux = matrix[i : Time_Steps + i, :]
        x[i,:,:] = aux
        y[i,:] = matrix[Time_Steps+i, targ_index]

    print("length of time-series i/o",x.shape,y.shape)

    return x, y


tw = 5

Xtr, ytr = create_sequences(Xtr, sensor_idx,tw)
Xval, yval = create_sequences(Xval, sensor_idx,tw)
Xtest, ytest = create_sequences(Xtest, sensor_idx,tw)


batch_size = 24

train_data = TensorDataset(torch.from_numpy(Xtr), torch.from_numpy(ytr))
train_loader = DataLoader(train_data, shuffle = True, batch_size =
batch_size,
                          drop_last = True)

val_data = TensorDataset(torch.from_numpy(Xval), torch.from_numpy(yval))
val_loader = DataLoader(val_data, shuffle = True, batch_size =
batch_size,
                        drop_last = True)
```

```
is_cuda = torch.cuda.is_available()

if is_cuda:
    device = torch.device("cuda")
else:
    device = torch.device("cpu")


class LSTMNet(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim, n_layers,
drop_prob=0.2):
        super(LSTMNet, self).__init__()
        self.hidden_dim = hidden_dim
        self.n_layers = n_layers

        self.lstm = nn.LSTM(input_dim, hidden_dim, n_layers,
batch_first=True, dropout=drop_prob)
        self.fc = nn.Linear(hidden_dim, output_dim)
        self.relu = nn.ReLU()

    def forward(self, x, h):
        out, h = self.lstm(x, h)
        out = self.fc(self.relu(out[:,-1]))
        return out, h

    def init_hidden(self, batch_size):
        weight = next(self.parameters()).data
        hidden = (weight.new(self.n_layers, batch_size,
self.hidden_dim).zero_().to(device),
                  weight.new(self.n_layers, batch_size,
self.hidden_dim).zero_().to(device))
        return hidden



n_features = Xtr.shape[2]
n_timesteps = tw
hidden_dim = 256
output_dim = 1
n_layers = 2


model = LSTMNet(n_features,hidden_dim,output_dim,n_layers)
model.to(device)

criterion = torch.nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

train_episodes = 60



model.train()
for t in range(train_episodes):
    htr = model.init_hidden(batch_size)
```

```
    for x,y in train_loader:
        htr = tuple([e.data for e in htr])
        model.zero_grad()
        output, htr = model(x.to(device).float(),htr)
        loss = criterion(output, y.to(device).float())
        loss.backward()
        optimizer.step()

    hval = model.init_hidden(batch_size)
    for xval,yval in val_loader:
        hval = tuple([e.data for e in hval])
        output_val, hval = model(xval.to(device).float(),htr)
        loss_val = criterion(output_val, yval.to(device).float())

    print('step : ' , t , ' loss_train: ' , loss.item(), ' loss_val: ',
loss_val.item())


model.eval()
Xtest = torch.from_numpy(Xtest)
htest = model.init_hidden(Xtest.shape[0])
out, htest = model(Xtest.to(device).float(), htest)

out = out.cpu().detach().numpy()

out = scal_tr.inverse_transform(out)
ytest = scal_tr.inverse_transform(ytest)



fig = plt.figure()

ax11 = fig.add_subplot(211)
ax11.plot(ytest, 'r', label='Real')

ax11.set_title('Mediciones reales vs Mediciones predichas
',fontweight='bold')
ax11.set_ylabel('PM2.5 [ug/m^3]')
ax11.set_xlabel('Tiempo [h]')
ax11.plot(out, label='Predicción')
ax11.legend(loc=(0,0))
plt.grid()

plt.show()


ytest_f = np.delete(ytest,[299,300,354])
out_f = np.delete(out, [299,300,354])



from sklearn.metrics import mean_absolute_error

MAE = mean_absolute_error(ytest_f, out_f)
print("MAE: ",MAE)
```

```
def percentage_error(actual, predicted):
    res = np.empty(actual.shape)
    for j in range(actual.shape[0]):
        if actual[j] != 0:
            res[j] = (actual[j] - predicted[j]) / actual[j]
        else:
            res[j] = predicted[j] / np.mean(actual)
    return res

def mean_absolute_percentage_error(y_true, y_pred):
    return np.mean(np.abs(percentage_error(np.asarray(y_true),
np.asarray(y_pred)))) * 100

MAPE = mean_absolute_percentage_error(ytest_f, out_f)
print("MAPE:", MAPE)



from sklearn.metrics import mean_squared_error

RMSE = mean_squared_error(ytest_f, out_f, squared=False)
print("RMSE: ",RMSE)
```

FIRMA ESTUDIANTES 

FIRMA ASESORES 

FECHA ENTREGA: 26/06/2020

FIRMA COMITÉ TRABAJO DE GRADO DE LA FACULTAD _____

RECHAZADO___ ACEPTADO____ ACEPTADO CON MODIFICACIONES_____

ACTA NO._____

FECHA ENTREGA: _____

FIRMA CONSEJO DE FACULTAD_____

ACTA NO._____

FECHA ENTREGA: _____