Rodrigo Sánchez Molina

RED NEURONAL REGRESIVA PARA PREDECIR EL CONSUMO ENERGÉTICO DE LA ETSII

1. Introducción

En este documento se ilustra el procedimiento de entrenamiento de una red neuronal de regresión para predecir el consumo energético de la ETSII [Escuela Técnica Superior de Ingenieros Industriales de Madrid]. La implementación de la red se hará en Python por la extensión y facilidad de dicho lenguaje para el uso de estas técnicas en la literatura. Para el uso y entrenamiento de la red, se hace uso de la herramienta *Google Collaborate* puesto que se dispone de recursos muy potentes que proporciona Google de forma gratuita. De esta manera, podremos acceder desde cualquier dispositivo, sea cual sea la potencia de cada uno.

Solamente se indicarán fragmentos de código que se consideren relevantes. El resto de código se encuentra al final del documento o en GitHub. Esto se debe a que el cuaderno de trabajo empleado es autoexplicativo y posee de comentarios y cabeceras de texto.

2. Análisis y preparación de los datos

Se cargan los datos de consumo de energía de la ETSII desde un archivo "*.csv*". Se plotean los datos de consumo energético de los datos que se dispone para intuir si se aprecia estacionalidad y detectar datos erróneos para poder eliminarlos y que el aprendizaje sea mejor. Se dispone de datos de consumo desde enero de 2014 hasta diciembre de 2020.

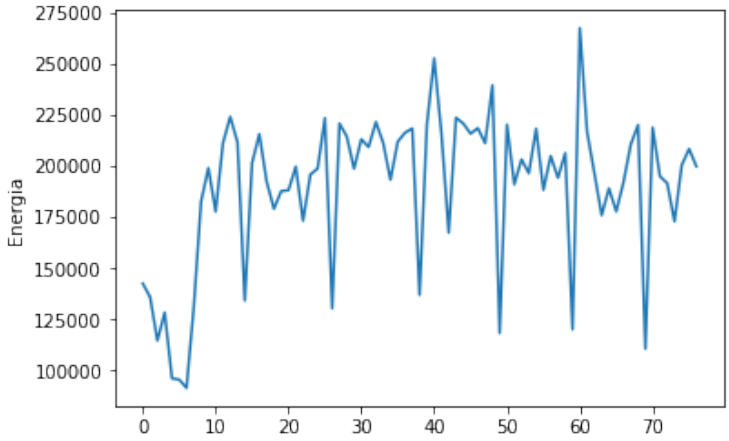


Figura 1. Evolución del consumo de energético de la ETSII de los datos de entrada a la red

3. Generación del dataset de entrenamiento y de validación

Una vez importadas las librerías oportunas [tensorflow y keras principalmente], se debe separar la tabla de datos con el objetivo de entrenar la red y poder comprobar su efectividad sobre datos no entrenados, para ver su eficacia. No tendría sentido entrenar una red y predecir perfectamente los datos de salida de entrenamiento, puesto que cuando se use en la realidad tendrá como input otros datos [años y meses] de los que no se tiene constancia previa en la red. En particular se separa en un 80% de los datos para el entrenamiento y un 20% para la evaluación.

Una vez separados los datasets en *train y test sets*, se deben normalizar los datos para facilitar el entrenamiento. Si los datos son muy grandes y sin escalar, el entrenamiento de la red se complica y es posible que no podamos converger a una solución decente. Hay muchas formas de normalizar, en este caso se ha escogido una estandarización de los datos tal y como se muestra en la siguiente

figura. De esta manera, nos quitamos de problemas de estandarización. Bien es cierto, que no impide que la red no fuese entrenable si no se normalizase, pero suele ser una buena práctica.

Normalizamos tanto entradas como salidas

```
[48] def norm(x):
       return (x - train_stats['mean']) / train_stats['std']
     normed_train_data = norm(train_dataset)
     normed_test_data = norm(test_dataset)

     def norm_(x):
       return (x - train_labels_stats['mean']) / train_labels_stats['std']
     normed_train_data_labels = norm_(train_labels)
     normed_test_data_labels = norm_(test_labels)
```

Figura 2. Estandarización de los datos

4. Definición de la red neuronal

La red se podría definir de muchas maneras, desde niveles más bajos a niveles más altos de abstracción. Para este modelo sencillo, se hace uso de keras puesto que es muy fácil su implementación.

Se define una red secuencial [es decir, no presenta saltos entre capas dentro de la red]. En este caso, se han definido dos *hidden layers* [capas internas u ocultas] con 64 unidades y una capa de salida de una única unidad [se trata de una regresión]. Si se tratase de una red neuronal de clasificación, entonces la salida sería múltiple en base a las clases identificables.

Por otro lado, se define la función de pérdidas como el error cuadrático medio y se escoge el método de optimización RMSprop, el cual es adecuado según la literatura para entrenar redes de regresión.

Definimos el modelo de la red a entrenar

```
def build_model():
  model = keras.Sequential([
    tf.keras.layers.Dense(64, activation='relu', input_shape=[len(train_dataset.keys())]),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(1)
  ])

  optimizer = tf.keras.optimizers.RMSprop(0.005)

  model.compile(loss='mse',
                optimizer=optimizer,
                metrics=['mae', 'mse'])
  return model
```

Figura 3. Definición de la red neuronal

5. Entrenamiento de la red

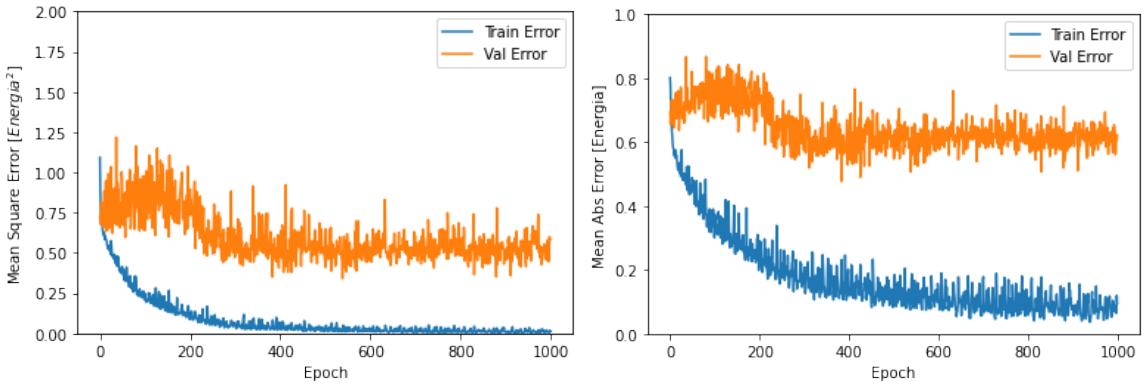
Para el entrenamiento de la red, se hace uso de la función model.fit(), donde se le pasan los valores de entrada y sus salidas teóricas. Se le indica el número de épocas de entrenamiento. En particular, se subdivide el set de entrenamiento para validar durante las iteraciones, de manera que se tiene un *loss* y un *validation_loss*. Esto es muy común en el entrenamiento de las redes neuronales. Es preciso notar, que no se ha hecho uso de la métrica accuracy, porque en regresión no tiene sentido acertar el valor exacto de entrada [es prácticamente imposible]. El accuracy se usa en clasificación.

6. Resultados

A continuación, se presentan de forma gráfica los resultados obtenidos para el entrenamiento. Posteriormente en el apartado de Predicciones se aportará el resultado sobre el test set.

	loss	mae	mse	val_loss	val_mae	val_mse	epoch
995	0.013254	0.086761	0.013254	0.445994	0.560279	0.445994	995
996	0.010389	0.084077	0.010389	0.477920	0.573509	0.477920	996
997	0.006270	0.064186	0.006270	0.600458	0.617195	0.600458	997
998	0.018842	0.119702	0.018842	0.586668	0.604884	0.586668	998
999	0.014034	0.094684	0.014034	0.584526	0.621721	0.584526	999

Figura 4. Métricas al fin del entrenamiento



7. Predicciones

Para elaborar las predicciones sobre los datos de validación se hace uso de la función model.evaluate(). Donde se le aporta entrada y salida teóricas. Se obtiene un *loss* de 0.2648.

```
loss, mae, mse = model.evaluate(normed_test_data, normed_test_data_labels, verbose=2)

print("Testing set Mean Abs Error: {:5.2f} Energy".format(mae))
```

```
1/1 - 0s - loss: 0.2684 - mae: 0.3966 - mse: 0.2684
Testing set Mean Abs Error:  0.40 Energy
```

Figura 5. Comportamiento sobre el dataset de validación

Como se aprecia en la figura 6, los datos se encuentran cercanos a la recta de pendiente unidad.

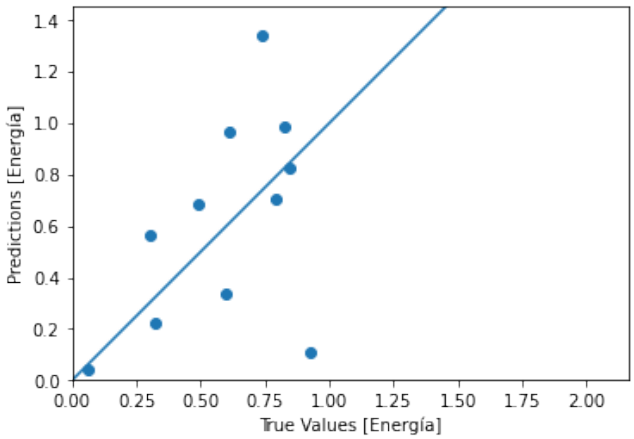


Figura 6. Predicciones frente a valores reales de consumo energético

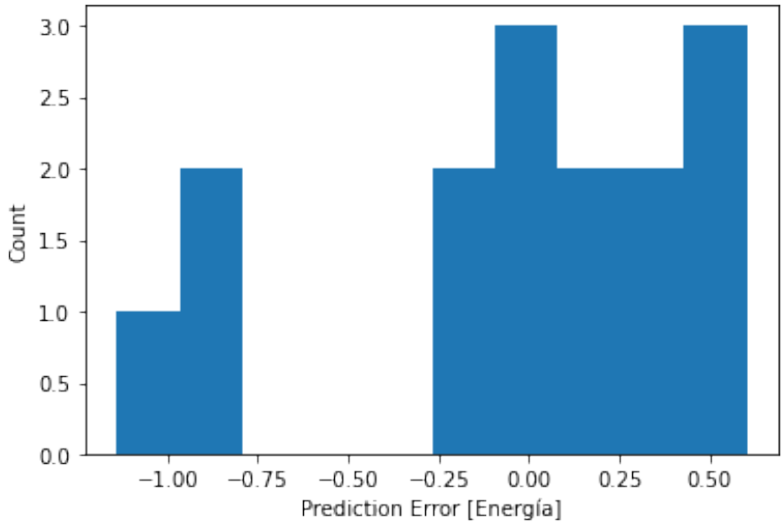


Figura 7. Distribución del error obtenido

8. Conclusiones

Como se puede apreciar, la red no es tan precisa como cabría esperar. Los datos son demasiado escasos, por lo que elaborar una predicción sobre los mismos no tiene por qué ser adecuada.

Se debe notar que, no tendría mucho sentido entrenar la red con muchas más épocas, puesto que, en vista a la gráfica obtenida durante el entrenamiento, aparentemente se vuelve horizontal. Se podrían añadir capas, variar el tamaño de las mismas y encontrar una mejor solución.

Como alternativa a este tipo de red, en la literatura se mencionan las redes LSTM [son también secuenciales], las cuales son ampliamente utilizadas para aprender secuencias. Personalmente, considero que este enfoque podría dar un mejor resultado que el obtenido puesto que se aprecia cierta estacionalidad en los datos de consumo energético de la figura 1.

Importamos liberías

```
import pandas as pd
from google.colab import files
import tensorflow as tf
from tensorflow import keras
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import io
```

Leemos el archivo de datos

```
uploaded = files.upload()
```

Guardamos en una variable la tabla leída

```
df2 = pd.read_csv(io.BytesIO(uploaded['ETSII-Data-01_organized.csv']), sep=';')
dataset = df2.copy()
dataset.tail()
```

Ploteamos los datos de energía

```
sns.lineplot(x=dataset.index, y="Energia", data=dataset);
```

Separamos el dataset de entrenamiento y de validación

```
train_dataset = dataset.sample(frac=0.8,random_state=0)
test_dataset = dataset.drop(train_dataset.index)

train_labels = train_dataset.pop('Energia')
test_labels = test_dataset.pop('Energia')

train_labels = train_labels.astype(dtype=float)
test_labels = test_labels.astype(dtype=float)
```

Calculamos algunas estadísticas

```
train_labels_stats = train_labels.describe()

train_stats = train_dataset.describe()
train_stats = train_stats.transpose()

print(train_stats)
```

Normalizamos tanto entradas como salidas

```
def norm(x):
  return (x - train_stats['mean']) / train_stats['std']
normed_train_data = norm(train_dataset)
normed_test_data = norm(test_dataset)

def norm_(x):
  return (x - train_labels_stats['mean']) / train_labels_stats['std']
normed_train_data_labels = norm_(train_labels)
normed_test_data_labels = norm_(test_labels)
```

Definimos el modelo de la red a entrenar

```
def build_model():
  model = keras.Sequential([
    tf.keras.layers.Dense(64, activation='relu', input_shape=[len(train_dataset.key
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(1)
  ])

  optimizer = tf.keras.optimizers.RMSprop(0.005)

  model.compile(loss='mse',
                optimizer=optimizer,
                metrics=['mae', 'mse'])
  return model
```

Instanciamos y mostramos resumen

```
model = build_model()
model.summary()
```

Entrenamos a la red

```
EPOCHS = 1000

history = model.fit(
  normed_train_data, normed_train_data_labels,
  epochs=EPOCHS, validation_split = 0.2)
```

Guardamos la información del entrenamiento

```
hist = pd.DataFrame(history.history)
hist['epoch'] = history.epoch
hist.tail()
```

Ploteamos la información del entrenamiento

```
def plot_history(history):
  hist = pd.DataFrame(history.history)
  hist['epoch'] = history.epoch

  plt.figure()
  plt.xlabel('Epoch')
  plt.ylabel('Mean Abs Error [Energia]')
  plt.plot(hist['epoch'], hist['mae'],
           label='Train Error')
  plt.plot(hist['epoch'], hist['val_mae'],
           label = 'Val Error')
  plt.ylim([0,1])
  plt.legend()

  plt.figure()
  plt.xlabel('Epoch')
  plt.ylabel('Mean Square Error [$Energia^2$]')
  plt.plot(hist['epoch'], hist['mse'],
           label='Train Error')
  plt.plot(hist['epoch'], hist['val_mse'],
           label = 'Val Error')
  plt.ylim([0,2])
  plt.legend()
  plt.show()


plot_history(history)
```

Evaluamos el test set

```
loss, mae, mse = model.evaluate(normed_test_data, normed_test_data_labels, verbose=

print("Testing set Mean Abs Error: {:5.2f} Energy".format(mae))
```

Ploteamos los resultados del test set

```
test_predictions = model.predict(normed_test_data).flatten()

plt.scatter(normed_test_data_labels, test_predictions)
plt.xlabel('True Values [Energía]')
plt.ylabel('Predictions [Energía]')
plt.axis('equal')
plt.axis('square')
plt.xlim([0,plt.xlim()[1]])
plt.ylim([0,plt.ylim()[1]])
_ = plt.plot([-150, 150], [-150, 150])
```

Ploteamos distribución del error en forma de histograma

```
error = test_predictions - normed_test_data_labels
plt.hist(error, bins = 10)
plt.xlabel("Prediction Error [Energía]")
_ = plt.ylabel("Count")
```