import pathlib

import matplotlib.pyplot as plt

import pandas as pd

import seaborn as sns

import tensorflow as tf

from tensorflow import keras

from tensorflow.keras import layers

dataset\_path = keras.utils.get\_file("auto-mpg.data", "http://archive.ics.uci.edu/ml/machine-learning-databases/auto-mpg/auto-mpg.data")

dataset\_path

column\_names = ['MPG','Cylinders','Displacement','Horsepower','Weight',  
                'Acceleration', 'Model Year', 'Origin']

raw\_dataset = pd.read\_csv(dataset\_path, names=column\_names,  
                      na\_values = "?", comment='\t',  
                      sep=" ", skipinitialspace=True)  
  
dataset = raw\_dataset.copy()  
dataset.tail()

**Limpie la data**

dataset.isna().sum()

MPG 0

Cylinders 0

Displacement 0

Horsepower 6

Weight 0

Acceleration 0

Model Year 0

Origin 0

dtype: int64

dataset = dataset.dropna()

La columna de "Origin" realmente es categorica, no numerica. Entonces conviertala a un "one-hot":

origin = dataset.pop('Origin')

dataset['USA'] = (origin == 1)\*1.0  
dataset['Europe'] = (origin == 2)\*1.0  
dataset['Japan'] = (origin == 3)\*1.0  
dataset.tail()

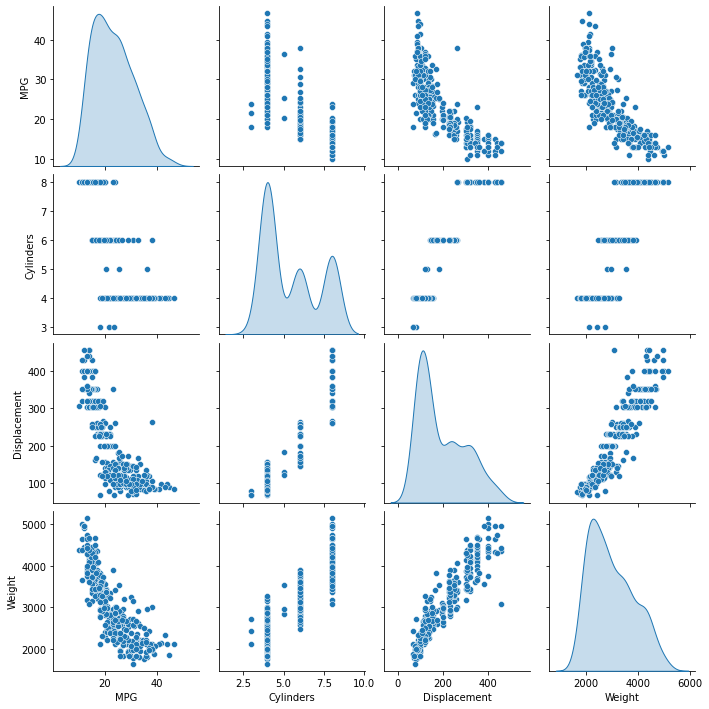
**Dividamos la data en entrenamiento y prueba**

train\_dataset = dataset.sample(frac=0.8,random\_state=0)  
test\_dataset = dataset.drop(train\_dataset.index)

**Inspeccione la data**

Revise rapidamente la distribucion conjunta de un par de columnas de el set de entrenamiento.

sns.pairplot(train\_dataset[["MPG", "Cylinders", "Displacement", "Weight"]], diag\_kind="kde")



Tambien revise las estadisticas generales:

train\_stats = train\_dataset.describe()  
train\_stats.pop("MPG")  
train\_stats = train\_stats.transpose()  
train\_stats

**Separe las caracteristicas de las etiquetas.**

Separe el valor objetivo, o la "etiqueta" de las caracteristicas. Esta etiqueta es el valor que entrenara el modelo para predecir.

train\_labels = train\_dataset.pop('MPG')  
test\_labels = test\_dataset.pop('MPG')

**Normalice la data**

Revise otra vez el bloque de train\_stats que se presento antes y note la diferencia de rangos de cada caracteristica.

Es una buena práctica normalizar funciones que utilizan diferentes escalas y rangos. Aunque el modelo \* podría \* converger sin normalización de características, dificulta el entrenamiento y hace que el modelo resultante dependa de la elección de las unidades utilizadas en la entrada.

Nota: Aunque generamos intencionalmente estas estadísticas solo del conjunto de datos de entrenamiento, estas estadísticas también se utilizarán para normalizar el conjunto de datos de prueba. Necesitamos hacer eso para proyectar el conjunto de datos de prueba en la misma distribución en la que el modelo ha sido entrenado.

def norm(x):  
  return (x - train\_stats['mean']) / train\_stats['std']  
normed\_train\_data = norm(train\_dataset)  
normed\_test\_data = norm(test\_dataset)

Estos datos normalizados es lo que usaremos para entrenar el modelo.

Precaución: las estadísticas utilizadas para normalizar las entradas aquí (media y desviación estándar) deben aplicarse a cualquier otro dato que se alimente al modelo, junto con la codificación de un punto que hicimos anteriormente. Eso incluye el conjunto de pruebas, así como los datos en vivo cuando el modelo se usa en producción.

**El modelo**

Construyamos nuestro modelo. Aquí, utilizaremos un modelo secuencial con dos capas ocultas densamente conectadas y una capa de salida que devuelve un único valor continuo. Los pasos de construcción del modelo se envuelven en una función, build\_model, ya que crearemos un segundo modelo, más adelante.

def build\_model():  
  model = keras.Sequential([  
    layers.Dense(64, activation='relu', input\_shape=[len(train\_dataset.keys())]),  
    layers.Dense(64, activation='relu'),  
    layers.Dense(1)  
  ])  
  
  optimizer = tf.keras.optimizers.RMSprop(0.001)  
  
  model.compile(loss='mse',  
                optimizer=optimizer,  
                metrics=['mae', 'mse'])  
  return model

model = build\_model()

**Inspeccione el modelo**

Use el método .summary para imprimir una descripción simple del modelo

model.summary()

Model: "sequential"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

dense (Dense) (None, 64) 640

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_1 (Dense) (None, 64) 4160

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_2 (Dense) (None, 1) 65

=================================================================

Total params: 4,865

Trainable params: 4,865

Non-trainable params: 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Ahora pruebe el modelo. Tome un lote de ejemplos 10 de los datos de entrenamiento y llame amodel.predict en él.

example\_batch = normed\_train\_data[:10]  
example\_result = model.predict(example\_batch)  
example\_result

array([[-0.01398571],

[ 0.09287428],

[-0.06822745],

[ 0.01392132],

[ 0.2776244 ],

[-0.02446011],

[ 0.24629724],

[ 0.585566 ],

[-0.01556234],

[ 0.31573048]], dtype=float32)

Parece estar funcionando, y produce un resultado de la forma y tipo esperados.

**Entrenar a la modelo**

Entrene el modelo durante 1000 épocas y registre la precisión de entrenamiento y validación en el objeto history.

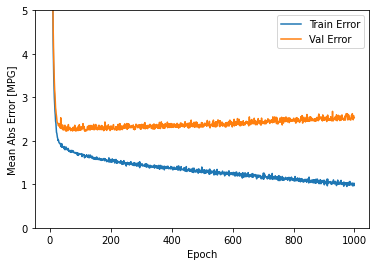
# Display training progress by printing a single dot for each completed epoch  
class PrintDot(keras.callbacks.Callback):  
  def on\_epoch\_end(self, epoch, logs):  
    if epoch % 100 == 0: print('')  
    print('.', end='')  
  
EPOCHS = 1000  
  
history = model.fit(  
  normed\_train\_data, train\_labels,  
  epochs=EPOCHS, validation\_split = 0.2, verbose=0,  
  callbacks=[PrintDot()])

Visualice el progreso de entrenamiento del modelo usando las estadísticas almacenadas en el objeto history.

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

def plot\_history(history):  
  hist = pd.DataFrame(history.history)  
  hist['epoch'] = history.epoch  
  
  plt.figure()  
  plt.xlabel('Epoch')  
  plt.ylabel('Mean Abs Error [MPG]')  
  plt.plot(hist['epoch'], hist['mae'],  
           label='Train Error')  
  plt.plot(hist['epoch'], hist['val\_mae'],  
           label = 'Val Error')  
  plt.ylim([0,5])  
  plt.legend()

  plt.figure()  
  plt.xlabel('Epoch')  
  plt.ylabel('Mean Square Error [$MPG^2$]')  
  plt.plot(hist['epoch'], hist['mse'],  
           label='Train Error')  
  plt.plot(hist['epoch'], hist['val\_mse'],  
           label = 'Val Error')  
  plt.ylim([0,20])  
  plt.legend()  
  plt.show()  
  
  
plot\_history(history)

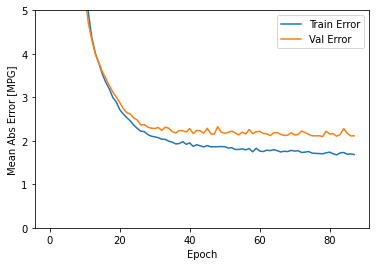


Este gráfico muestra poca mejora, o incluso degradación en el error de validación después de aproximadamente 100 épocas. Actualicemos la llamada model.fit para detener automáticamente el entrenamiento cuando el puntaje de validación no mejore. Utilizaremos una \* devolución de llamada de EarlyStopping \* que pruebe una condición de entrenamiento para cada época. Si transcurre una cantidad determinada de épocas sin mostrar mejoría, entonces detiene automáticamente el entrenamiento.

Puedes obtener más información sobre esta devolución de llamada [Aca](https://www.tensorflow.org/versions/master/api_docs/python/tf/keras/callbacks/EarlyStopping?hl=es-419).

model = build\_model()  
  
# The patience parameter is the amount of epochs to check for improvement  
early\_stop = keras.callbacks.EarlyStopping(monitor='val\_loss', patience=10)  
  
history = model.fit(normed\_train\_data, train\_labels, epochs=EPOCHS,  
                    validation\_split = 0.2, verbose=0, callbacks=[early\_stop, PrintDot()])  
  
plot\_history(history)

........................................................................................



El gráfico muestra que en el conjunto de validación, el error promedio generalmente es de alrededor de +/- 2 MPG. ¿Es esto bueno? Le dejaremos esa decisión a usted.

Veamos qué tan bien generaliza el modelo al usar el conjunto \*\* test \*\*, que no usamos al entrenar el modelo. Esto nos dice qué tan bien podemos esperar que el modelo prediga cuándo lo usamos en el mundo real.

loss, mae, mse = model.evaluate(normed\_test\_data, test\_labels, verbose=2)  
  
print("Testing set Mean Abs Error: {:5.2f} MPG".format(mae))

3/3 - 0s - loss: 5.9382 - mae: 1.9334 - mse: 5.9382

Testing set Mean Abs Error: 1.93 MPG

**Haga Predicciones**

Finalmente, prediga los valores de MPG utilizando datos en el conjunto de pruebas:

test\_predictions = model.predict(normed\_test\_data).flatten()

|  |  |
| --- | --- |
| plt.scatter(test\_labels, test\_predictions) plt.xlabel('True Values [MPG]') plt.ylabel('Predictions [MPG]') plt.axis('equal') plt.axis('square') plt.xlim([0,plt.xlim()[1]]) plt.ylim([0,plt.ylim()[1]]) \_ = plt.plot([-100, 100], [-100, 100]) | png |

Parece que nuestro modelo predice razonablemente bien. Echemos un vistazo a la distribución de errores.

|  |  |
| --- | --- |
| error = test\_predictions - test\_labels plt.hist(error, bins = 25) plt.xlabel("Prediction Error [MPG]") \_ = plt.ylabel("Count") | png |

No es del todo gaussiano, pero podríamos esperar eso porque el número de muestras es muy pequeño.

**Conclusion**

Este cuaderno introdujo algunas técnicas para manejar un problema de regresión.

* El error cuadrático medio (MSE) es una función de pérdida común utilizada para problemas de regresión (se utilizan diferentes funciones de pérdida para problemas de clasificación).
* Del mismo modo, las métricas de evaluación utilizadas para la regresión difieren de la clasificación. Una métrica de regresión común es el error absoluto medio (MAE).
* Cuando las características de datos de entrada numéricos tienen valores con diferentes rangos, cada característica debe escalarse independientemente al mismo rango.
* Si no hay muchos datos de entrenamiento, una técnica es preferir una red pequeña con pocas capas ocultas para evitar el sobreajuste.
* La detención temprana es una técnica útil para evitar el sobreajuste.