LINEAR REGRESSION WITH TENSORFLOW 2

This notebook introduced a few techniques to handle a regression problem. Credit: François Chollet

Mean Squared Error (MSE) is a common loss function used for regression problems (different loss functions are used for classification problems). Similarly, evaluation metrics used for regression differ from classification. A common regression metric is Mean Absolute Error (MAE). When numeric input data features have values with different ranges, each feature should be scaled independently to the same range. If there is not much training data, one technique is to prefer a small network with few hidden layers to avoid overfitting. Early stopping is a useful technique to prevent overfitting.

In a regression problem, we aim to predict the output of a continuous value, like a price or a probability. Contrast this with a classification problem, where we aim to select a class from a list of classes (for example, where a picture contains an apple or an orange, recognizing which fruit is in the picture).

```
import pathlib

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
```

```
/usr/local/lib/python3.6/dist-packages/statsmodels/tools/_testing.py:19: FutureWarning: pandas.util.testing is depreca
  import pandas.util.testing as tm
```

```
import tensorflow as tf

from tensorflow import keras
from tensorflow.keras import layers

print(tf.__version__)
```

```
2.2.0
```

```
!pip install -q git+https://github.com/tensorflow/docs
import tensorflow_docs as tfdocs
import tensorflow_docs.plots
import tensorflow_docs.modeling
```

```
  Building wheel for tensorflow-docs (setup.py) ... done
```

```
dataset_path = keras.utils.get_file("auto-mpg.data", "http://archive.ics.uci.edu/ml/machine-learning-databases/auto-mpg
dataset_path
```

```
Downloading data from http://archive.ics.uci.edu/ml/machine-learning-databases/auto-mpg/auto-mpg.data
32768/30286 [================================] - 0s 2us/step
'/root/.keras/datasets/auto-mpg.data'
```

```
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()
```

	MPG	Cylinders	Displacement	Horsepower	Weight	Acceleration	Model Year	Origin
393	27.0	4	140.0	86.0	2790.0	15.6	82	1
394	44.0	4	97.0	52.0	2130.0	24.6	82	2
395	32.0	4	135.0	84.0	2295.0	11.6	82	1
396	28.0	4	120.0	79.0	2625.0	18.6	82	1
397	31.0	4	119.0	82.0	2720.0	19.4	82	1

Make sure to check and clean the data

```
dataset.isna().sum()
```

```
MPG             0
Cylinders       0
Displacement    0
```

Drop the rows in this case. Otherwise, you can also replace the cell values with average values, for instance

```
dataset = dataset.dropna()
```

Origin is a categorical value and has to be converted into a one-hot encoding.

```
dataset['Origin'] = dataset['Origin'].map({1: 'USA', 2: 'Europe', 3: 'Japan'})
```

```
dataset = pd.get_dummies(dataset, prefix='', prefix_sep='')
dataset.tail()
```

	MPG	Cylinders	Displacement	Horsepower	Weight	Acceleration	Model Year	Europe	Japan	USA
393	27.0	4	140.0	86.0	2790.0	15.6	82	0	0	1
394	44.0	4	97.0	52.0	2130.0	24.6	82	1	0	0
395	32.0	4	135.0	84.0	2295.0	11.6	82	0	0	1
396	28.0	4	120.0	79.0	2625.0	18.6	82	0	0	1
397	31.0	4	119.0	82.0	2720.0	19.4	82	0	0	1

Now split the dataset into a training set and a test set.

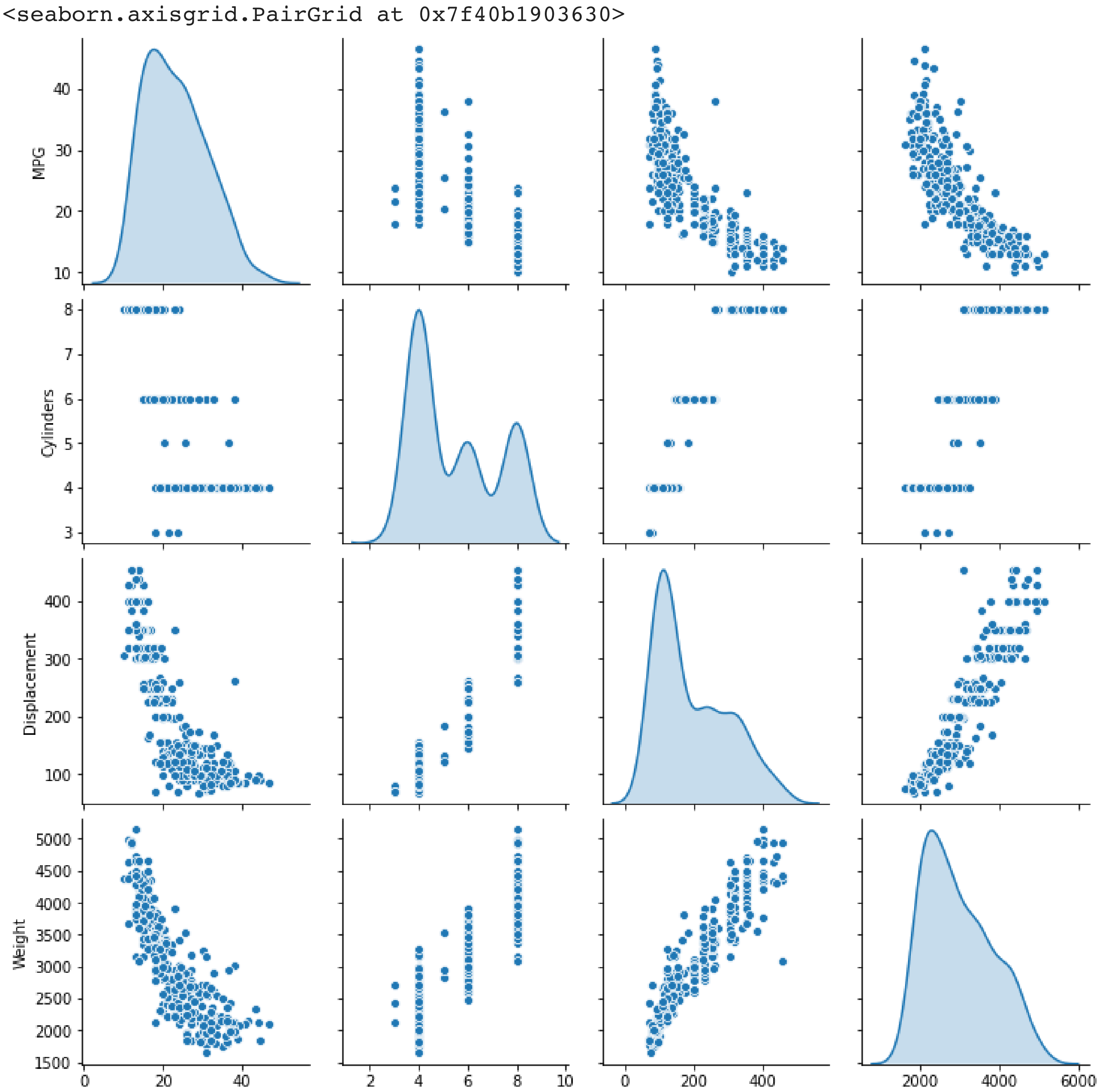
We will use the test set in the final evaluation of our model.

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

We take a quick look at the joint distribution of a few pairs of columns from the training set.

```
sns.pairplot(train_dataset[["MPG", "Cylinders", "Displacement", "Weight"]], diag_kind="kde")
```

```
<seaborn.axisgrid.PairGrid at 0x7f40b1903630>
```



We describe the overall statistics

```
train_stats = train_dataset.describe()
train_stats.pop("MPG")
train_stats = train_stats.transpose()
train_stats
```

	count	mean	std	min	25%	50%	75%	max
Cylinders	314.0	5.477707	1.699788	3.0	4.00	4.0	8.00	8.0
Displacement	314.0	195.318471	104.331589	68.0	105.50	151.0	265.75	455.0
Horsepower	314.0	104.869427	38.096214	46.0	76.25	94.5	128.00	225.0
Weight	314.0	2990.251592	843.898596	1649.0	2256.50	2822.5	3608.00	5140.0
Acceleration	314.0	15.559236	2.789230	8.0	13.80	15.5	17.20	24.8
Model Year	314.0	75.898089	3.675642	70.0	73.00	76.0	79.00	82.0
Europe	314.0	0.178344	0.383413	0.0	0.00	0.0	0.00	1.0
Japan	314.0	0.197452	0.398712	0.0	0.00	0.0	0.00	1.0
USA	314.0	0.624204	0.485101	0.0	0.00	1.0	1.00	1.0

We separate the target value, or "label", from the features. This label is the value that you will train the model to predict.

```
train_labels = train_dataset.pop('MPG')
test_labels = test_dataset.pop('MPG')
```

It is good practice to normalize features that use different scales and ranges. Although the model might converge without feature normalization, it makes training more difficult, and it makes the resulting model dependent on the choice of units used in the input.

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

We use a Sequential model with two densely connected hidden layers, and an output layer that returns a single, continuous value. The model building steps are wrapped in a function, build_model, since we create a second model, later on.

```
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()
```

Use the model.summary method to print a simple description of the model.

```
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
_________________________________________________________________
```

```
example_batch = normed_train_data[:10]
example_result = model.predict(example_batch)
example_result
```

```
array([[ 0.2597918 ],
       [ 0.29629475],
       [ 0.02762973],
       [ 0.59187376],
       [ 0.40167347],
       [ 0.02249654],
       [ 0.5023507 ],
       [ 0.6483175 ],
       [-0.02960253],
```

```
EPOCHS = 1000

history = model.fit(
  normed_train_data, train_labels,
  epochs=EPOCHS, validation_split = 0.2, verbose=0,
  callbacks=[tfdocs.modeling.EpochDots()])
```

```
Epoch: 0, loss:551.1287,  mae:22.2415,  mse:551.1287,  val_loss:530.1335,  val_mae:21.7736,  val_mse:530.1335,
....................................................................................................
Epoch: 100, loss:6.1849,  mae:1.7943,  mse:6.1849,  val_loss:8.8206,  val_mae:2.2294,  val_mse:8.8206,
....................................................................................................
Epoch: 200, loss:5.5172,  mae:1.6226,  mse:5.5172,  val_loss:8.1999,  val_mae:2.1261,  val_mse:8.1999,
....................................................................................................
Epoch: 300, loss:4.8669,  mae:1.5122,  mse:4.8669,  val_loss:8.2752,  val_mae:2.1900,  val_mse:8.2752,
....................................................................................................
Epoch: 400, loss:4.3033,  mae:1.4319,  mse:4.3033,  val_loss:8.7515,  val_mae:2.2820,  val_mse:8.7515,
....................................................................................................
Epoch: 500, loss:3.8291,  mae:1.3197,  mse:3.8291,  val_loss:8.3260,  val_mae:2.2633,  val_mse:8.3260,
....................................................................................................
Epoch: 600, loss:3.6612,  mae:1.2571,  mse:3.6612,  val_loss:8.2493,  val_mae:2.1647,  val_mse:8.2493,
....................................................................................................
Epoch: 700, loss:3.1073,  mae:1.2217,  mse:3.1073,  val_loss:8.6671,  val_mae:2.2725,  val_mse:8.6671,
....................................................................................................
Epoch: 800, loss:2.8253,  mae:1.1201,  mse:2.8253,  val_loss:7.7339,  val_mae:2.1613,  val_mse:7.7339,
....................................................................................................
Epoch: 900, loss:2.8763,  mae:1.1216,  mse:2.8763,  val_loss:8.3411,  val_mae:2.2210,  val_mse:8.3411,
....................................................................................................
```

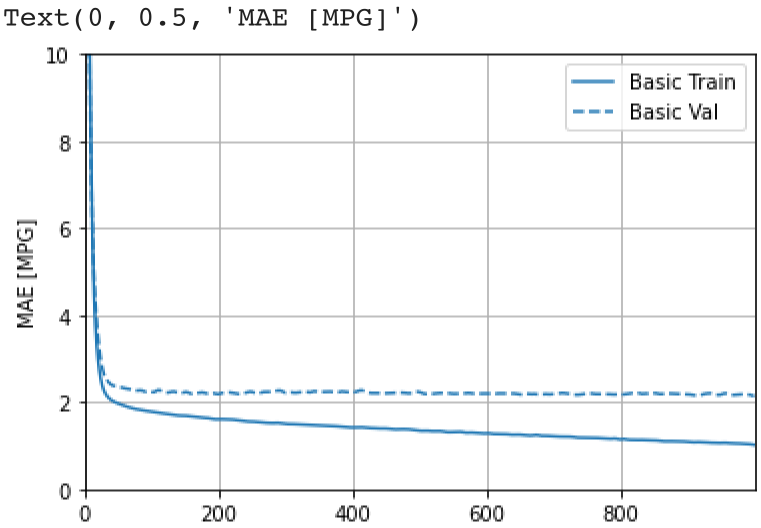
```
hist = pd.DataFrame(history.history)
hist['epoch'] = history.epoch
hist.tail()
```

	loss	mae	mse	val_loss	val_mae	val_mse	epoch
995	2.613018	1.049560	2.613018	7.883402	2.173120	7.883402	995
996	2.482682	0.987949	2.482682	7.483046	2.123491	7.483046	996
997	2.789662	1.046976	2.789662	7.846499	2.180307	7.846499	997
998	2.616198	1.016463	2.616198	7.439682	2.114568	7.439682	998
999	2.565062	0.993662	2.565062	7.348742	2.068708	7.348742	999

```
plotter = tfdocs.plots.HistoryPlotter(smoothing_std=2)
```

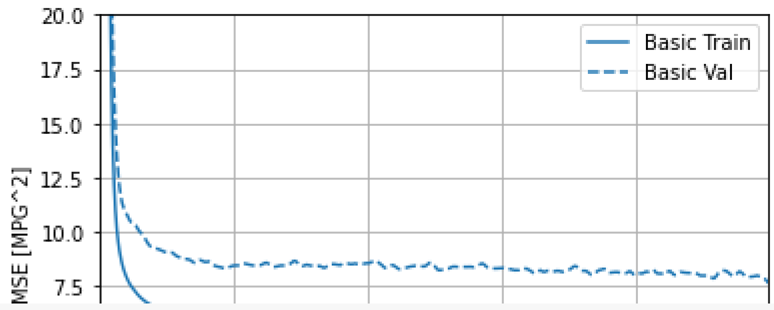
```
plotter.plot({'Basic': history}, metric = "mae")
plt.ylim([0, 10])
plt.ylabel('MAE [MPG]')
```

```
Text(0, 0.5, 'MAE [MPG]')
```




```
plotter.plot({'Basic': history}, metric = "mse")
plt.ylim([0, 20])
plt.ylabel('MSE [MPG^2]')
```

```
Text(0, 0.5, 'MSE [MPG^2]')
```

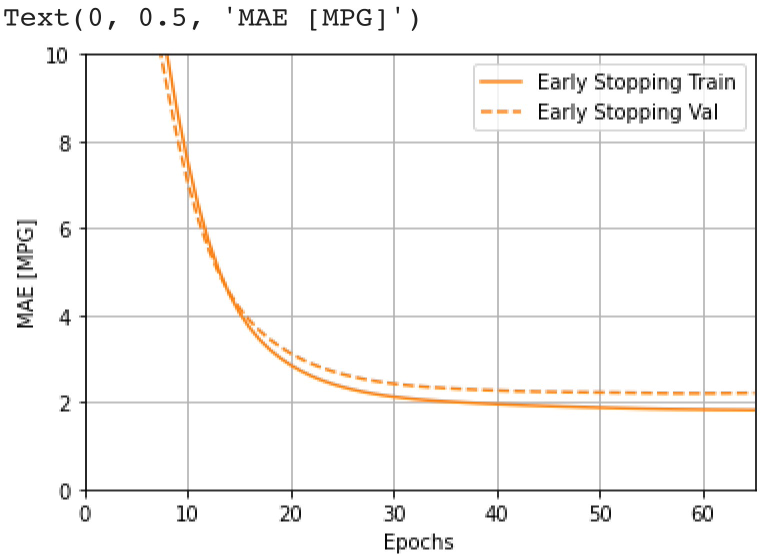



```
model = build_model()

# The patience parameter is the amount of epochs to check for improvement
early_stop = keras.callbacks.EarlyStopping(monitor='val_loss', patience=10)

early_history = model.fit(normed_train_data, train_labels,
                    epochs=EPOCHS, validation_split = 0.2, verbose=0,
                    callbacks=[early_stop, tfdocs.modeling.EpochDots()])
```

```
Epoch: 0, loss:550.4537,  mae:22.3010,  mse:550.4537,  val_loss:532.5619,  val_mae:21.8994,  val_mse:532.5619,
...................................................................
```

```
plotter.plot({'Early Stopping': early_history}, metric = "mae")
plt.ylim([0, 10])
plt.ylabel('MAE [MPG]')
```

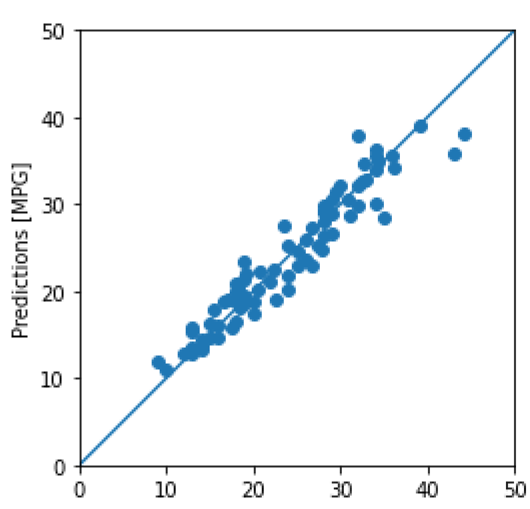
```
Text(0, 0.5, 'MAE [MPG]')
```



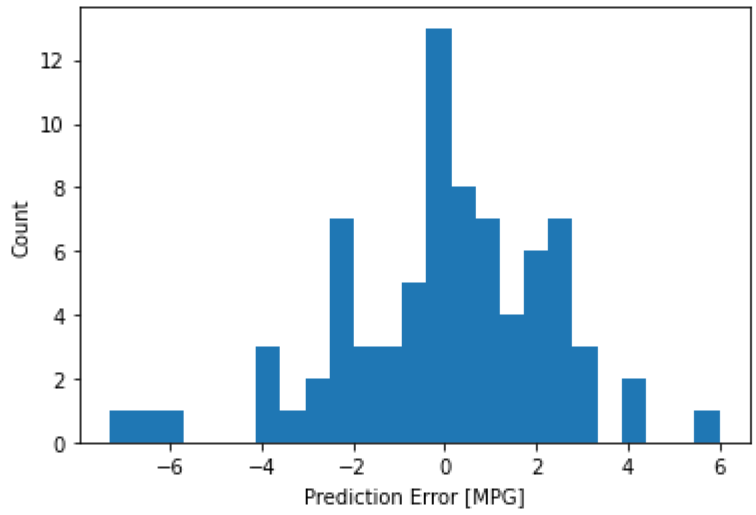

```
loss, mae, mse = model.evaluate(normed_test_data, test_labels, verbose=2)

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

```
3/3 - 0s - loss: 5.5541 - mae: 1.7632 - mse: 5.5541
Testing set Mean Abs Error:  1.76 MPG
```

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

a = plt.axes(aspect='equal')
plt.scatter(test_labels, test_predictions)
plt.xlabel('True Values [MPG]')
plt.ylabel('Predictions [MPG]')
lims = [0, 50]
plt.xlim(lims)
plt.ylim(lims)
_ = plt.plot(lims, lims)
```




```
error = test_predictions - test_labels
plt.hist(error, bins = 25)
plt.xlabel("Prediction Error [MPG]")
_ = plt.ylabel("Count")
```



Count
Prediction Error [MPG]