Basic regression: Predict fuel efficiency

In a *regression* problem, the aim is to predict the output of a continuous value, like a price or a probability. Contrast this with a *classification* problem, where the aim is 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).

This tutorial uses the classic Auto MPG (https://archive.ics.uci.edu/ml/datasets/auto+mpg) dataset and demonstrates how to build models to predict the fuel efficiency of the late-1970s and early 1980s automobiles. To do this, you will provide the models with a description of many automobiles from that time period. This description includes attributes like cylinders, displacement, horsepower, and weight.

This example uses the Keras API. (Visit the Keras tutorials (https://www.tensorflow.org/tutorials/keras) and guides (https://www.tensorflow.org/guide/keras) to learn more.)

```
$ # Use seaborn for pairplot.
$ pip install -q seaborn
```

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

# Make NumPy printouts easier to read.
np.set_printoptions(precision=3, suppress=True)
```

```
import tensorflow as tf

from tensorflow import keras
from tensorflow.keras import layers

print(tf.__version__)
```

The Auto MPG dataset

The dataset is available from the UCI Machine Learning Repository (https://archive.ics.uci.edu/ml/).

Get the data

First download and import the dataset using pandas:

```
url = 'http://archive.ics.uci.edu/ml/machine-learning-databases/auto-mpg/auto·
column_names = ['MPG', 'Cylinders', 'Displacement', 'Horsepower', 'Weight',
                'Acceleration', 'Model Year', 'Origin']

raw_dataset = pd.read_csv(url, names=column_names,
                          na_values='?', comment='\t',
                          sep=' ', skipinitialspace=True)
```

```
dataset = raw_dataset.copy()
dataset.tail()
```

Clean the data

The dataset contains a few unknown values:

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

Drop those rows to keep this initial tutorial simple:

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

The "`Origin`" column is categorical, not numeric. So the next step is to one-hot encode the values in the column with pd.get_dummies (https://pandas.pydata.org/docs/reference/api/pandas.get_dummies.html).

Note: You can set up the **`tf.keras.Model`** (https://www.tensorflow.org/api_docs/python/tf/keras/Model) to do this kind of transformation for you but that's beyond the scope of this tutorial. Check out the Classify structured data using Keras preprocessing layers (https://www.tensorflow.org/tutorials/structured_data/preprocessing_layers) or Load CSV data (https://www.tensorflow.org/tutorials/load_data/csv) tutorials for examples.

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

```
dataset = pd.get_dummies(dataset, columns=['Origin'], prefix='', prefix_sep='')
dataset.tail()
```

Split the data into training and test sets

Now, split the dataset into a training set and a test set. You will use the test set in the final evaluation of your models.

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

Inspect the data

Review the joint distribution of a few pairs of columns from the training set.

The top row suggests that the fuel efficiency (MPG) is a function of all the other parameters. The other rows indicate they are functions of each other.

```
sns.pairplot(train_dataset[['MPG', 'Cylinders', 'Displacement', 'Weight']], d:
```

Let's also check the overall statistics. Note how each feature covers a very different range:

```
train_dataset.describe().transpose()
```

Split features from labels

Separate the target value—the "label"—from the features. This label is the value that you will train the model to predict.

```
train_features = train_dataset.copy()
test_features = test_dataset.copy()

train_labels = train_features.pop('MPG')
test_labels = test_features.pop('MPG')
```

Normalization

In the table of statistics it's easy to see how different the ranges of each feature are:

```
train_dataset.describe().transpose()[['mean', 'std']]
```

It is good practice to normalize features that use different scales and ranges.

One reason this is important is because the features are multiplied by the model weights. So, the scale of the outputs and the scale of the gradients are affected by the scale of the inputs.

Although a model *might* converge without feature normalization, normalization makes training much more stable.

Note: There is no advantage to normalizing the one-hot features—it is done here for simplicity. For more details on how to use the preprocessing layers, refer to the Working with preprocessing layers (https://www.tensorflow.org/guide/keras/preprocessing_layers) guide and the Classify structured data using Keras preprocessing layers (https://www.tensorflow.org/tutorials/structured_data/preprocessing_layers) tutorial.

The Normalization layer

The `tf.keras.layers.Normalization` (https://www.tensorflow.org/api_docs/python/tf/keras/layers/Normalization) is a clean and simple way to add feature normalization into your model.

The first step is to create the layer:

```
normalizer = tf.keras.layers.Normalization(axis=-1)
```

Then, fit the state of the preprocessing layer to the data by calling `Normalization.adapt` (https://www.tensorflow.org/api_docs/python/tf/keras/layers/Normalization#adapt):

```
normalizer.adapt(np.array(train_features))
```

Calculate the mean and variance, and store them in the layer:

```
print(normalizer.mean.numpy())
```

When the layer is called, it returns the input data, with each feature independently normalized:

```
first = np.array(train_features[:1])

with np.printoptions(precision=2, suppress=True):
  print('First example:', first)
```

```
print()
print('Normalized:', normalizer(first).numpy())
```

Linear regression

Before building a deep neural network model, start with linear regression using one and several variables.

Linear regression with one variable

Begin with a single-variable linear regression to predict `'MPG'` from `'Horsepower'`.

Training a model with `tf.keras` (https://www.tensorflow.org/api_docs/python/tf/keras) typically starts by defining the model architecture. Use a `tf.keras.Sequential` (https://www.tensorflow.org/api_docs/python/tf/keras/Sequential) model, which represents a sequence of steps (https://www.tensorflow.org/guide/keras/sequential_model).

There are two steps in your single-variable linear regression model:

- Normalize the `'Horsepower'` input features using the `tf.keras.layers.Normalization` preprocessing layer.
- Apply a linear transformation ($y = mx + b$) to produce 1 output using a linear layer (`tf.keras.layers.Dense` (https://www.tensorflow.org/api_docs/python/tf/keras/layers/Dense)).

The number of *inputs* can either be set by the `input_shape` argument, or automatically when the model is run for the first time.

First, create a NumPy array made of the `'Horsepower'` features. Then, instantiate the `tf.keras.layers.Normalization` and fit its state to the `horsepower` data:

```
horsepower = np.array(train_features['Horsepower'])

horsepower_normalizer = layers.Normalization(input_shape=[1,], axis=None)
horsepower_normalizer.adapt(horsepower)
```

Build the Keras Sequential model:

```
horsepower_model = tf.keras.Sequential([
    horsepower_normalizer,
    layers.Dense(units=1)
])

horsepower_model.summary()
```

This model will predict `'MPG'` from `'Horsepower'`.

Run the untrained model on the first 10 'Horsepower' values. The output won't be good, but notice that it has the expected shape of `(10, 1)`:

```
horsepower_model.predict(horsepower[:10])
```

Once the model is built, configure the training procedure using the Keras **`Model.compile`** (https://www.tensorflow.org/api_docs/python/tf/keras/Model#compile) method. The most important arguments to compile are the **`loss`** and the **`optimizer`**, since these define what will be optimized (**`mean_absolute_error`**) and how (using the **`tf.keras.optimizers.Adam`** (https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/Adam)).

```
horsepower_model.compile(
    optimizer=tf.keras.optimizers.Adam(learning_rate=0.1),
    loss='mean_absolute_error')
```

Use Keras **`Model.fit`** (https://www.tensorflow.org/api_docs/python/tf/keras/Model#fit) to execute the training for 100 epochs:

```
%%time
history = horsepower_model.fit(
    train_features['Horsepower'],
    train_labels,
    epochs=100,
    # Suppress logging.
    verbose=0,
    # Calculate validation results on 20% of the training data.
    validation_split = 0.2)
```

Visualize the model's training progress using the stats stored in the `history` object:

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

```
def plot_loss(history):
  plt.plot(history.history['loss'], label='loss')
  plt.plot(history.history['val_loss'], label='val_loss')
  plt.ylim([0, 10])
  plt.xlabel('Epoch')
  plt.ylabel('Error [MPG]')
  plt.legend()
  plt.grid(True)
```

```
plot_loss(history)
```

Collect the results on the test set for later:

```
test_results = {}

test_results['horsepower_model'] = horsepower_model.evaluate(
    test_features['Horsepower'],
    test_labels, verbose=0)
```

Since this is a single variable regression, it's easy to view the model's predictions as a function of the input:

```
x = tf.linspace(0.0, 250, 251)
y = horsepower_model.predict(x)
```

```
def plot_horsepower(x, y):
  plt.scatter(train_features['Horsepower'], train_labels, label='Data')
  plt.plot(x, y, color='k', label='Predictions')
```

```
  plt.xlabel('Horsepower')
  plt.ylabel('MPG')
  plt.legend()
```

```
plot_horsepower(x, y)
```

Linear regression with multiple inputs

You can use an almost identical setup to make predictions based on multiple inputs. This model still does the same $y = mx + b$ except that m is a matrix and x is a vector.

Create a two-step Keras Sequential model again with the first layer being `normalizer` (`tf.keras.layers.Normalization(axis=-1)`) you defined earlier and adapted to the whole dataset:

```
linear_model = tf.keras.Sequential([
    normalizer,
    layers.Dense(units=1)
])
```

When you call [`Model.predict`](https://www.tensorflow.org/api_docs/python/tf/keras/Model#predict) on a batch of inputs, it produces `units=1` outputs for each example:

```
linear_model.predict(train_features[:10])
```

When you call the model, its weight matrices will be built—check that the `kernel` weights (the m in $y = mx + b$) have a shape of `(9, 1)`:

```
linear_model.layers[1].kernel
```

Configure the model with Keras [`Model.compile`](https://www.tensorflow.org/api_docs/python/tf/keras/Model#compile) and train with [`Model.fit`](https://www.tensorflow.org/api_docs/python/tf/keras/Model#fit) for 100 epochs:

```
linear_model.compile(
    optimizer=tf.keras.optimizers.Adam(learning_rate=0.1),
    loss='mean_absolute_error')
```

```
%%time
history = linear_model.fit(
    train_features,
    train_labels,
    epochs=100,
    # Suppress logging.
    verbose=0,
    # Calculate validation results on 20% of the training data.
    validation_split = 0.2)
```

Using all the inputs in this regression model achieves a much lower training and validation error than the **`horsepower_model`**, which had one input:

```
plot_loss(history)
```

Collect the results on the test set for later:

```
test_results['linear_model'] = linear_model.evaluate(
    test_features, test_labels, verbose=0)
```

Regression with a deep neural network (DNN)

In the previous section, you implemented two linear models for single and multiple inputs.

Here, you will implement single-input and multiple-input DNN models.

The code is basically the same except the model is expanded to include some "hidden" non-linear layers. The name "hidden" here just means not directly connected to the inputs or outputs.

These models will contain a few more layers than the linear model:

- The normalization layer, as before (with `horsepower_normalizer` for a single-input model and `normalizer` for a multiple-input model).
- Two hidden, non-linear, `Dense` layers with the ReLU (`relu`) activation function nonlinearity.
- A linear `Dense` single-output layer.

Both models will use the same training procedure, so the `compile` method is included in the `build_and_compile_model` function below.

```
def build_and_compile_model(norm):
  model = keras.Sequential([
      norm,
      layers.Dense(64, activation='relu'),
      layers.Dense(64, activation='relu'),
      layers.Dense(1)
  ])

  model.compile(loss='mean_absolute_error',
                optimizer=tf.keras.optimizers.Adam(0.001))
  return model
```

Regression using a DNN and a single input

Create a DNN model with only `'Horsepower'` as input and `horsepower_normalizer` (defined earlier) as the normalization layer:

```
dnn_horsepower_model = build_and_compile_model(horsepower_normalizer)
```

This model has quite a few more trainable parameters than the linear models:

```
dnn_horsepower_model.summary()
```

Train the model with Keras `Model.fit` (https://www.tensorflow.org/api_docs/python/tf/keras/Model#fit):

```
%%time
history = dnn_horsepower_model.fit(
    train_features['Horsepower'],
    train_labels,
    validation_split=0.2,
    verbose=0, epochs=100)
```

This model does slightly better than the linear single-input `horsepower_model`:

```
plot_loss(history)
```

If you plot the predictions as a function of `'Horsepower'`, you should notice how this model takes advantage of the nonlinearity provided by the hidden layers:

```
x = tf.linspace(0.0, 250, 251)
y = dnn_horsepower_model.predict(x)
```

```
plot_horsepower(x, y)
```

Collect the results on the test set for later:

```
test_results['dnn_horsepower_model'] = dnn_horsepower_model.evaluate(
    test_features['Horsepower'], test_labels,
    verbose=0)
```

Regression using a DNN and multiple inputs

Repeat the previous process using all the inputs. The model's performance slightly improves on the validation dataset.

```
dnn_model = build_and_compile_model(normalizer)
dnn_model.summary()
```

```
%%time
history = dnn_model.fit(
    train_features,
    train_labels,
    validation_split=0.2,
    verbose=0, epochs=100)
```

```
plot_loss(history)
```

Collect the results on the test set:

```
test_results['dnn_model'] = dnn_model.evaluate(test_features, test_labels, ver
```

Performance

Since all models have been trained, you can review their test set performance:

```
pd.DataFrame(test_results, index=['Mean absolute error [MPG]']).T
```

These results match the validation error observed during training.

Make predictions

You can now make predictions with the **`dnn_model`** on the test set using Keras **`Model.predict`** (https://www.tensorflow.org/api_docs/python/tf/keras/Model#predict) and review the loss:

```
test_predictions = dnn_model.predict(test_features).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)
```

It appears that the model predicts reasonably well.

Now, check the error distribution:

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

If you're happy with the model, save it for later use with **`Model.save`** (https://www.tensorflow.org/api_docs/python/tf/keras/Model#save):

```
dnn_model.save('dnn_model.keras')
```

If you reload the model, it gives identical output:

```
reloaded = tf.keras.models.load_model('dnn_model.keras')

test_results['reloaded'] = reloaded.evaluate(
    test_features, test_labels, verbose=0)
```

```
pd.DataFrame(test_results, index=['Mean absolute error [MPG]']).T
```

Conclusion

This notebook introduced a few techniques to handle a regression problem. Here are a few more tips that may help:

- Mean squared error (MSE) (`tf.keras.losses.MeanSquaredError` (https://www.tensorflow.org/api_docs/python/tf/keras/losses/MeanSquaredError)) and mean absolute error (MAE) (`tf.keras.losses.MeanAbsoluteError` (https://www.tensorflow.org/api_docs/python/tf/keras/losses/MeanAbsoluteError)) are common loss functions used for regression problems. MAE is less sensitive to outliers. Different loss functions are used for classification problems.
- Similarly, evaluation metrics used for regression differ from classification.
- When numeric input data features have values with different ranges, each feature should be scaled independently to the same range.
- Overfitting is a common problem for DNN models, though it wasn't a problem for this tutorial. Visit the Overfit and underfit (/tutorials/keras/overfit_and_underfit) tutorial for more help with this.

Last updated 2024-07-12 UTC.