ML Design Pattern #1: Transform

Moving an ML model to production is much easier if you keep inputs, features, and transforms separate

Lak Lakshmanan
Sep 19, 2019 · 4 min read

An occasional series of design patterns for ML Engineers. Full list here.

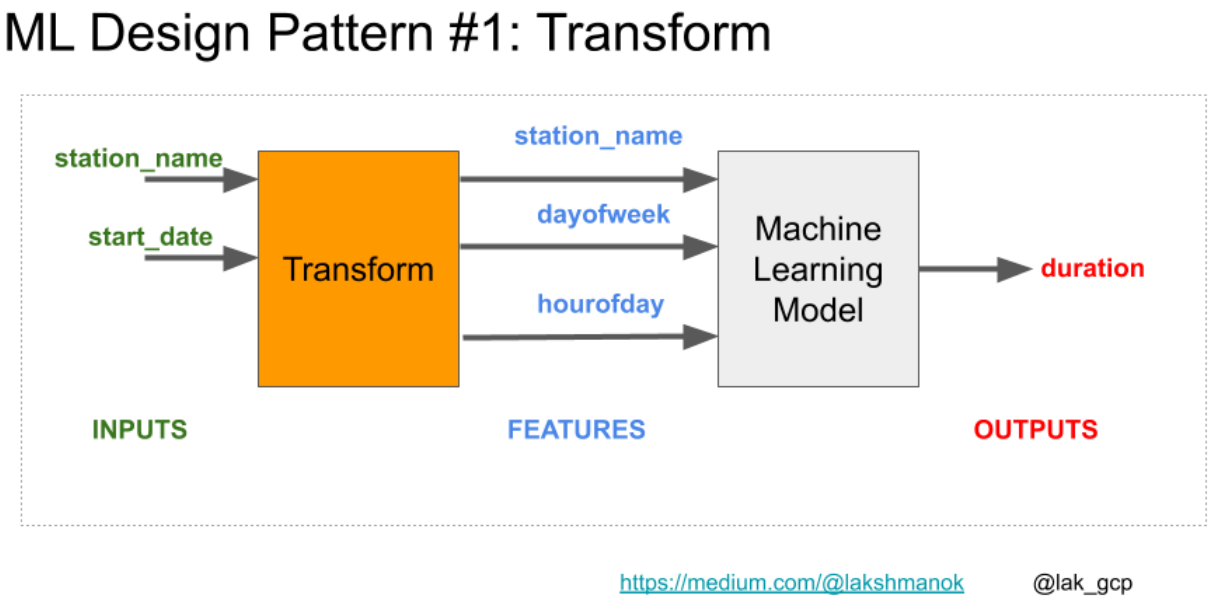
I will illustrate the Transform design pattern using BigQuery ML because SQL makes the concept easy to understand without getting bogged down in syntax. Then, I'll explain how to implement the pattern in TensorFlow 2.0 and Keras.

The problem: Inputs != Features

When you train a machine learning model, you train it with features that are extracted from the raw inputs. Take this simple model that is trained to predict the duration of bicycle rides in London in BigQuery ML:

```
CREATE OR REPLACE MODEL ch09eu.bicycle_model
OPTIONS(input_label_cols=['duration'],
 model_type='linear_reg')
AS
SELECT
 duration
 , start_station_name
 , CAST(EXTRACT(dayofweek from start_date) AS STRING)
 as dayofweek
 , CAST(EXTRACT(hour from start_date) AS STRING)
 as hourofday
FROM
 `bigquery-public-data.london_bicycles.cycle_hire`
```

This model has three features (start_station_name, dayofweek, and hourofday) computed from two inputs: start_station_name and start_date:



But the SQL code above mixes up the inputs and features and doesn't keep track of the transformations that were carried out. This comes back to bite when we try to predict with this model. Because the model was trained on three features, this is what the prediction signature has to look like:

```
SELECT * FROM ML.PREDICT(MODEL ch09eu.bicycle_model,(
   'Kings Cross' AS start_station_name
 , '3' as dayofweek
 , '18' as hourofday
))
```

Note that, at inference time, we have to know the transformations that were applied, and remember to send in '3' for dayofweek. It's not just that we have to know what features the model was trained on. That '3' … is that Tuesday or Wednesday? Depends on which library was used by the model! This is one of the key reasons why productionization of ML models is so hard.

Transform to the rescue

BigQuery ML on Google Cloud gives you a way out — use the TRANSFORM clause. Using TRANSFORM ensures transformations are automatically applied during ML.PREDICT. The model above should be rewritten as:

```
CREATE OR REPLACE MODEL ch09eu.bicycle_model
OPTIONS(input_label_cols=['duration'],
 model_type='linear_reg')
TRANSFORM(
 SELECT * EXCEPT(start_date)
 , CAST(EXTRACT(dayofweek from start_date) AS STRING)
 as dayofweek
 , CAST(EXTRACT(hour from start_date) AS STRING)
 as hourofday
)
AS
SELECT
 duration, start_station_name, start_date
FROM
 `bigquery-public-data.london_bicycles.cycle_hire`
```

Notice how we have clearly separated out the inputs (in the SELECT clause) from the features (in the TRANSFORM clause). Now, the prediction is a whole lot easier. We can simply send to the model a timestamp:

```
SELECT * FROM ML.PREDICT(MODEL ch09eu.bicycle_model,(
   'Kings Cross' AS start_station_name
 , CURRENT_TIMESTAMP() as start_date
))
```

BigQuery ML keeps track of the transformations for you, saves in the model graph, and automatically applies the transformations during prediction. Neat, eh?

Note: at the time of writing, "transform" is in alpha.

Transformations in Keras / TensorFlow 2.0

Yeah, BigQuery ML is the bee's knees. But you are hardcore and want to do the transformation trick in TensorFlow. Can you? Sure! Keras in TensorFlow 2.0 supports feature columns, and these are saved in the model graph. But you have to be very, very clear to keep all the concepts separate in your mind. Ready?

Let's say that we want to take in four inputs (pickup latitude, pickup longitude, dropoff latitude, dropoff longitude) and create a transformed feature which is the Euclidean distance. Let's say that we also want to scale the inputs (BigQuery ML automatically scales the inputs).

1. Make every input to the Keras model an Input Layer, and make every transformation a Lambda Layer. You will have four Input Layers:

```
inputs = {
        colname : tf.keras.layers.Input(name=colname, shape=(),
dtype='float32')
           for colname in ['pickup_longitude', 'pickup_latitude',
'dropoff_longitude', 'dropoff_latitude']
}
```

2. Maintain a dictionary of transformed features, and scale these inputs using Lambda layers:

```
transformed = {}
for lon_col in ['pickup_longitude', 'dropoff_longitude']:
            transformed[lon_col] = tf.keras.layers.Lambda(
                lambda x: (x+78)/8.0,
                name='scale_{}'.format(lon_col)
            )(inputs[lon_col])
for lat_col in ['pickup_latitude', 'dropoff_latitude']:
            transformed[lat_col] = tf.keras.layers.Lambda(
                lambda x: (x-37)/8.0,
                name='scale_{}'.format(lat_col)
            )(inputs[lat_col])
```

You will also have one Lambda Layer for the euclidean distance, which is computed from four of the Input Layers:

```
def euclidean(params):
    lon1, lat1, lon2, lat2 = params
```

```
    londiff = lon2 - lon1
    latdiff = lat2 - lat1
    return tf.sqrt(londiff*londiff + latdiff*latdiff)

transformed['euclidean'] = tf.keras.layers.Lambda(euclidean,
name='euclidean')([
            inputs['pickup_longitude'],
            inputs['pickup_latitude'],
            inputs['dropoff_longitude'],
            inputs['dropoff_latitude']
        ])
```

3. All five of these transformed layers will be concatenated into a DenseFeatures Layer:

```
dnn_inputs = tf.keras.layers.DenseFeatures(feature_columns.values())
(transformed)
```

4. But wait! The constructor for DenseFeatures requires a set of feature columns — you will have to specify how to take each of the transformed values and convert them into an input to the neural network. You might use them as-is, or you might one-hot encode them or you might choose to bucketize the numbers. For simplicity, let's just use them all as-is:

```
feature_columns = {
        colname: tf.feature_column.numeric_column(colname)
           for colname in ['pickup_longitude', 'pickup_latitude',
'dropoff_longitude', 'dropoff_latitude']
}
feature_columns['euclidean'] =
tf.feature_column.numeric_column('euclidean')
```

5. Once you have a DenseFeatures, you can build the rest of your Keras model as usual:

```
h1 = tf.keras.layers.Dense(32, activation='relu', name='h1')
(dnn_inputs)
h2 = tf.keras.layers.Dense(8, activation='relu', name='h2')(h1)
output = tf.keras.layers.Dense(1, name='fare')(h2)
model = tf.keras.models.Model(inputs, output)
model.compile(optimizer='adam', loss='mse', metrics=['mse'])
```

A complete example is here on GitHub.

Efficient transformations with tf.transform

One drawback to the above approach is that the transformations will be carried out during each iteration of training. This is not such a big deal if all that you are doing is scaling. But what if your transformations are more robust? What if you want to scale using the mean and variance, and you need a pass through all the data first to compute these variables?

Use tf.transform for an efficient way of carrying out transformations and saving them so that the transformations can be applied by tf-serving during prediction time. See this canonical TFX example. Obviously, that takes a lot more engineering.

Machine Learning ML Engineering Keras Bigquery TensorFlow