Introducing the Keras Functional API

Learning Objectives

1. Understand embeddings and how to create them with the feature column API
2. Understand Deep and Wide models and when to use them
3. Understand the Keras functional API and how to build a deep and wide model with it

Introduction

In the last notebook, we learned about the Keras Sequential API. The Keras Functional API provides an alternate way of building models which is more flexible. With the Functional API, we can build models with more complex topologies, multiple input or output layers, shared layers or non-sequential data flows (e.g. residual layers).

In this notebook we'll use what we learned about feature columns to build a Wide & Deep model. Recall, that the idea behind Wide & Deep models is to join the two methods of learning through memorization and generalization by making a wide linear model and a deep learning model to accommodate both. You can have a look at the original research paper here: Wide & Deep Learning for Recommender Systems.

(image: https://ai.googleblog.com/2016/06/wide-deep-learning-better-together-with.html)

The Wide part of the model is associated with the memory element. In this case, we train a linear model with a wide set of crossed features and learn the correlation of this related data with the assigned label. The Deep part of the model is associated with the generalization element where we use embedding vectors for features. The best embeddings are then learned through the training process. While both of these methods can work well alone, Wide & Deep models excel by combining these techniques together.

Each learning objective will correspond to a **#TODO** in the student lab notebook -- try to complete that notebook first before reviewing this solution notebook.

In [ ]:

```
# Use the chown command to change the ownership of the repository.
!sudo chown -R jupyter:jupyter /home/jupyter/training-data-analyst
```

In [ ]:

```
# Ensure the right version of Tensorflow is installed.
!pip freeze | grep tensorflow==2.3 || pip install tensorflow==2.3
```

```
Collecting tensorflow==2.3
Downloading tensorflow-2.3.0-cp37-cp37m-manylinux2010_x86_64.whl (320.4 MB)
|████████████████████████████████| 320.4 MB 29 kB/s s eta 0:00:011
Requirement already satisfied: opt-einsum>=2.3.2 in /opt/conda/lib/python3.7/site-packag
es (from tensorflow==2.3) (3.3.0)
Requirement already satisfied: absl-py>=0.7.0 in /opt/conda/lib/python3.7/site-packages
(from tensorflow==2.3) (0.8.1)
Collecting gast==0.3.3
```

```
.
.
.
Successfully installed gast-0.3.3 tensorboard-2.3.0 tensorflow-2.3.0 tensorflow-estimato
r-2.3.0
```

Kindly ignore the deprecation warnings and incompatibility errors.

Start by importing the necessary libraries for this lab.

In [ ]:

```
# Importing necessary modules/libraries such as numpy, pandas and datetime.
import datetime
import os
import shutil

import numpy as np
import pandas as pd
import tensorflow as tf

from matplotlib import pyplot as plt
from tensorflow import keras

from tensorflow import feature_column as fc

from tensorflow.keras import Model
from tensorflow.keras.layers import (
    Input, Dense, DenseFeatures, concatenate)
from tensorflow.keras.callbacks import TensorBoard

print(tf.__version__)
```

```
2.3.0
```

In [ ]:

```
# It sets the backend of matplotlib to the 'inline' backend. The output of plotting com
# below the code cell that produced it. The resulting plots will then also be stored in
%matplotlib inline
```

Load raw data

We will use the taxifare dataset, using the CSV files that we created in the first notebook of this sequence. Those files have been saved into `../data` .

In [ ]:

```
!ls -l ../data/*.csv
```

```
-rw-r--r-- 1 jupyter jupyter 123590 Sep 22 10:12 ../data/taxi-test.csv
-rw-r--r-- 1 jupyter jupyter 579055 Sep 22 10:12 ../data/taxi-train.csv
-rw-r--r-- 1 jupyter jupyter 123114 Sep 22 10:12 ../data/taxi-valid.csv
```

Use tf.data to read the CSV files

We wrote these functions for reading data from the csv files above in the previous notebook. For this lab we will also include some additional engineered features in our model. In particular, we will compute the difference in latitude and longitude, as well as the Euclidean distance between the pick-up and drop-off locations. We can accomplish this by adding these new features to the features dictionary with the function `add_engineered_features` below.

Note that we include a call to this function when collecting our features dict and labels in the `features_and_labels` function below as well.

In [ ]:
```
# Selecting specific CSV_COLUMNS, LABEL_COLUMN, DEFAULTS, UNWANTED_COLS.
CSV_COLUMNS = [
    'fare_amount',
    'pickup_datetime',
    'pickup_longitude',
    'pickup_latitude',
    'dropoff_longitude',
    'dropoff_latitude',
    'passenger_count',
    'key'
]
LABEL_COLUMN = 'fare_amount'
DEFAULTS = [[0.0], ['na'], [0.0], [0.0], [0.0], [0.0], [0.0], ['na']]
UNWANTED_COLS = ['pickup_datetime', 'key']


# Create an input function reading a file using the Dataset API
def features_and_labels(row_data):
    label = row_data.pop(LABEL_COLUMN)
    features = row_data

    for unwanted_col in UNWANTED_COLS:
        features.pop(unwanted_col)

    return features, label


# Reading CSV files into a dataset.
def create_dataset(pattern, batch_size=1, mode='eval'):
    dataset = tf.data.experimental.make_csv_dataset(
        pattern, batch_size, CSV_COLUMNS, DEFAULTS)

    dataset = dataset.map(features_and_labels)

    if mode == 'train':
        dataset = dataset.shuffle(buffer_size=1000).repeat()

    # take advantage of multi-threading; 1=AUTOTUNE
    dataset = dataset.prefetch(1)
    return dataset
```

Feature columns for Wide and Deep model

For the Wide columns, we will create feature columns of crossed features. To do this, we'll create a collection of Tensorflow feature columns to pass to the `tf.feature_column.crossed_column` constructor. The Deep columns will consist of numeric columns and the embedding columns we want to create.

In [ ]:
```
# TODO 1
# 1. Bucketize latitudes and longitudes
NBUCKETS = 16
latbuckets = np.linspace(start=38.0, stop=42.0, num=NBUCKETS).tolist()
lonbuckets = np.linspace(start=-76.0, stop=-72.0, num=NBUCKETS).tolist()

fc_bucketized_plat = fc.bucketized_column(
    source_column=fc.numeric_column("pickup_longitude"), boundaries=lonbuckets)
fc_bucketized_plon = fc.bucketized_column(
    source_column=fc.numeric_column("pickup_latitude"), boundaries=latbuckets)
fc_bucketized_dlat = fc.bucketized_column(
    source_column=fc.numeric_column("dropoff_longitude"), boundaries=lonbuckets)
fc_bucketized_dlon = fc.bucketized_column(
    source_column=fc.numeric_column("dropoff_latitude"), boundaries=latbuckets)

# 2. Cross features for locations
fc_crossed_dloc = fc.crossed_column(
    [fc_bucketized_dlat, fc_bucketized_dlon],
    hash_bucket_size=NBUCKETS * NBUCKETS)
fc_crossed_ploc = fc.crossed_column(
    [fc_bucketized_plat, fc_bucketized_plon],
    hash_bucket_size=NBUCKETS * NBUCKETS)
fc_crossed_pd_pair = fc.crossed_column(
    [fc_crossed_dloc, fc_crossed_ploc],
    hash_bucket_size=NBUCKETS**4)

# 3. Create embedding columns for the crossed columns
fc_pd_pair = fc.embedding_column(categorical_column=fc_crossed_pd_pair, dimension=3)
fc_dloc = fc.embedding_column(categorical_column=fc_crossed_dloc, dimension=3)
fc_ploc = fc.embedding_column(categorical_column=fc_crossed_ploc, dimension=3)
```

Gather list of feature columns

Next we gather the list of wide and deep feature columns we'll pass to our Wide & Deep model in Tensorflow. Recall, wide columns are sparse, have linear relationship with the output while continuous columns are deep, have a complex relationship with the output. We will use our previously bucketized columns to collect crossed feature columns and sparse feature columns for our wide columns, and embedding feature columns and numeric features columns for the deep columns.

In [ ]:

```
# TODO 2
wide_columns = [
    # One-hot encoded feature crosses
    fc.indicator_column(fc_crossed_dloc),
    fc.indicator_column(fc_crossed_ploc),
    fc.indicator_column(fc_crossed_pd_pair)
]

deep_columns = [
    # Embedding_column to "group" together ...
    fc.embedding_column(fc_crossed_pd_pair, dimension=10),

    # Numeric columns
    fc.numeric_column("pickup_latitude"),
    fc.numeric_column("pickup_longitude"),
    fc.numeric_column("dropoff_longitude"),
    fc.numeric_column("dropoff_latitude")
]
```

Build a Wide and Deep model in Keras

To build a wide-and-deep network, we connect the sparse (i.e. wide) features directly to the output node, but pass the dense (i.e. deep) features through a set of fully connected layers. Here’s that model architecture looks using the Functional API.

First, we'll create our input columns using tf.keras.layers.Input.

In [ ]:

```
INPUT_COLS = [
    'pickup_longitude',
    'pickup_latitude',
    'dropoff_longitude',
    'dropoff_latitude',
    'passenger_count'
]

inputs = {colname : Input(name=colname, shape=(), dtype='float32')
          for colname in INPUT_COLS
}
```

Then, we'll define our custom RMSE evaluation metric and build our wide and deep model.

In [ ]:

```
# Here, tf.reduce_mean computes the mean of elements across dimensions of a tensor.
# tf.sqrt Computes element-wise square root of the input tensor.
# tf.square computes square of x element-wise.
def rmse(y_true, y_pred):
    return tf.sqrt(tf.reduce_mean(tf.square(y_pred - y_true)))

# TODO 3
def build_model(dnn_hidden_units):
    # Create the deep part of model
    deep = DenseFeatures(deep_columns, name='deep_inputs')(inputs)
    for num_nodes in dnn_hidden_units:
        deep = Dense(num_nodes, activation='relu')(deep)

    # Create the wide part of model
    wide = DenseFeatures(wide_columns, name='wide_inputs')(inputs)

    # Combine deep and wide parts of the model
    combined = concatenate(inputs=[deep, wide], name='combined')

    # Map the combined outputs into a single prediction value
    output = Dense(units=1, activation=None, name='prediction')(combined)

    # Finalize the model
    model = Model(inputs=list(inputs.values()), outputs=output)

    # Compile the keras model
    model.compile(optimizer="adam", loss="mse", metrics=[rmse, "mse"])

    return model
```

Next, we can call the `build_model` to create the model. Here we'll have two hidden layers, each with 10 neurons, for the deep part of our model. We can also use `plot_model` to see a diagram of the model we've created.

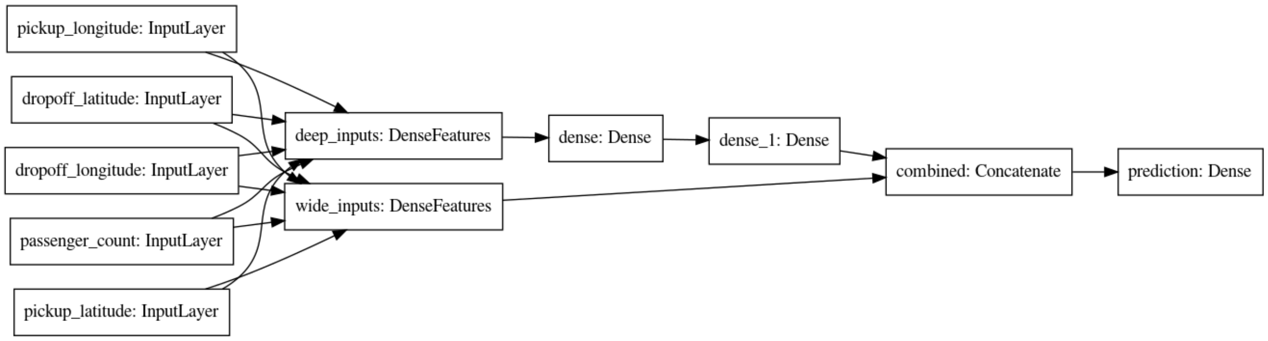
In [ ]:

```
HIDDEN_UNITS = [10,10]

# Calling the build model
model = build_model(dnn_hidden_units=HIDDEN_UNITS)

# Converts a Keras plot_model to see a diagram of the model that we have created.
tf.keras.utils.plot_model(model, show_shapes=False, rankdir='LR')
```

Out[ ]:


pickup_longitude: InputLayer
dropoff_latitude: InputLayer
dropoff_longitude: InputLayer
passenger_count: InputLayer
pickup_latitude: InputLayer
deep_inputs: DenseFeatures
wide_inputs: DenseFeatures
dense: Dense
dense_1: Dense
combined: Concatenate
prediction: Dense

Next, we'll set up our training variables, create our datasets for training and validation, and train our model.

(We refer you the the blog post ML Design Pattern #3: Virtual Epochs for further details on why express the training in terms of `NUM_TRAIN_EXAMPLES` and `NUM_EVALS` and why, in this training code, the number of epochs is really equal to the number of evaluations we perform.)

In [ ]:

```
BATCH_SIZE = 1000
NUM_TRAIN_EXAMPLES = 10000 * 5  # training dataset will repeat, wrap around
NUM_EVALS = 50  # how many times to evaluate
NUM_EVAL_EXAMPLES = 10000  # enough to get a reasonable sample

trainds = create_dataset(
    pattern='../data/taxi-train*',
    batch_size=BATCH_SIZE,
    mode='train')

evalds = create_dataset(
    pattern='../data/taxi-valid*',
    batch_size=BATCH_SIZE,
    mode='eval').take(NUM_EVAL_EXAMPLES//1000)
```

In [ ]:

```
%%time
# Here, %%time prints the wall time for the entire cell
steps_per_epoch = NUM_TRAIN_EXAMPLES // (BATCH_SIZE * NUM_EVALS)

OUTDIR = "./taxi_trained"
shutil.rmtree(path=OUTDIR, ignore_errors=True) # start fresh each time

history = model.fit(x=trainds,
                    steps_per_epoch=steps_per_epoch,
                    epochs=NUM_EVALS,
                    validation_data=evalds,
                    callbacks=[TensorBoard(OUTDIR)])
```

```
Epoch 1/50
1/1 [==============================] - 3s 3s/step - loss: 220.7379 - rmse: 14.8573 - ms
e: 220.7379 - val_loss: 248.7275 - val_rmse: 15.7663 - val_mse: 248.7275
Epoch 2/50
WARNING:tensorflow:From /opt/conda/lib/python3.7/site-packages/tensorflow/python/ops/sum
mary_ops_v2.py:1277: stop (from tensorflow.python.eager.profiler) is deprecated and will
be removed after 2020-07-01.
.
.
.
Epoch 49/50
1/1 [==============================] - 3s 3s/step - loss: 91.6058 - rmse: 9.5711 - mse:
91.6058 - val_loss: 117.5902 - val_rmse: 10.8209 - val_mse: 117.5902
Epoch 50/50
1/1 [==============================] - 3s 3s/step - loss: 113.9101 - rmse: 10.6729 - ms
e: 113.9101 - val_loss: 115.4815 - val_rmse: 10.7409 - val_mse: 115.4815
CPU times: user 3min 34s, sys: 4min 35s, total: 8min 9s
Wall time: 2min 51s
```

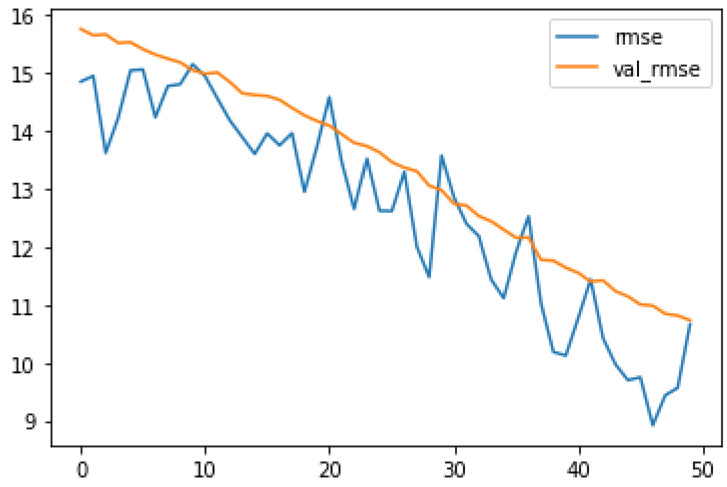
Just as before, we can examine the history to see how the RMSE changes through training on the train set and validation set.

In [ ]:

```
RMSE_COLS = ['rmse', 'val_rmse']

# Pandas DataFrame is two-dimensional size-mutable, potentially heterogeneous tabular d
pd.DataFrame(history.history)[RMSE_COLS].plot()
```

Out[ ]:


rmse
val_rmse