Introducing the Keras Sequential API

Learning Objectives

1. Build a DNN model using the Keras Sequential API
2. Learn how to use feature columns in a Keras model
3. Learn how to train a model with Keras
4. Learn how to save/load, and deploy a Keras model on GCP
5. Learn how to deploy and make predictions with at Keras model

Introduction

The Keras sequential API allows you to create Tensorflow models layer-by-layer. This is useful for building most kinds of machine learning models but it does not allow you to create models that share layers, re-use layers or have multiple inputs or outputs.

In this lab, we'll see how to build a simple deep neural network model using the Keras sequential api and feature columns. Once we have trained our model, we will deploy it using AI Platform and see how to call our model for online prediciton.

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 repository to user
!sudo chown -R jupyter:jupyter /home/jupyter/training-data-analyst
```

In [ ]:
```
# The datetime module used to work with dates as date objects.
import datetime
# The OS module in python provides functions for interacting with the operating system.
import os
# The shutil module in Python provides many functions of high-level operations on files
# This module helps in automating process of copying and removal of files and directori
import shutil

# Here we'll import data processing libraries like Numpy, Pandas and Tensorflow
import numpy as np
import pandas as pd
import tensorflow as tf

# Import pyplot package from matplotlib library
from matplotlib import pyplot as plt
# Import keras package from tensorflow library
from tensorflow import keras

# Import Sequential function from tensorflow.keras.models
from tensorflow.keras.models import Sequential
# Import Dense, DenseFeatures function from tensorflow.keras.layers
from tensorflow.keras.layers import Dense, DenseFeatures
# Import TensorBoard function from tensorflow.keras.callbacks
```

```
from tensorflow.keras.callbacks import TensorBoard

# Here we'll show the currently installed version of TensorFlow
print(tf.__version__)
%matplotlib inline
```

```
2.3.2
```

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 shows the working directory's contents.
# Using -l parameter will lists the files with assigned permissions
!ls -l ../data/*.csv
```

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

In [ ]:

```
# Output the first ten rows from the file where name having prefix `taxi`.
!head ../data/taxi*.csv
```

```
==> ../data/taxi-test.csv <==
6.0,2013-03-27 03:35:00 UTC,-73.977672,40.784052,-73.965332,40.801025,2,0
19.3,2012-05-10 18:43:16 UTC,-73.954366,40.778924,-74.004094,40.723104,1,1
7.5,2014-05-20 23:09:00 UTC,-73.999165,40.738377,-74.003473,40.723862,2,2
12.5,2015-02-23 19:51:31 UTC,-73.9652099609375,40.76948165893555,-73.98949432373047,40.7
39742279052734,1,3
10.9,2011-03-19 03:32:00 UTC,-73.99259,40.742957,-73.989908,40.711053,1,4
7.0,2012-09-18 12:51:11 UTC,-73.971195,40.751566,-73.975922,40.756361,1,5
19.0,2014-05-20 23:09:00 UTC,-73.998392,40.74517,-73.939845,40.74908,1,6
8.9,2012-07-18 08:46:08 UTC,-73.997638,40.756541,-73.973303,40.762019,1,7
4.5,2010-07-11 20:39:08 UTC,-73.976738,40.751321,-73.986671,40.74883,1,8
7.0,2013-12-12 02:16:40 UTC,-73.985024,40.767537,-73.981273,40.779302,1,9

==> ../data/taxi-train.csv <==
11.3,2011-01-28 20:42:59 UTC,-73.999022,40.739146,-73.990369,40.717866,1,0
7.7,2011-06-27 04:28:06 UTC,-73.987443,40.729221,-73.979013,40.758641,1,1
10.5,2011-04-03 00:54:53 UTC,-73.982539,40.735725,-73.954797,40.778388,1,2
16.2,2009-04-10 04:11:56 UTC,-74.001945,40.740505,-73.91385,40.758559,1,3
33.5,2014-02-24 18:22:00 UTC,-73.993372,40.753382,-73.8609,40.732897,2,4
6.9,2011-12-10 00:25:23 UTC,-73.996237,40.721848,-73.989416,40.718052,1,5
6.1,2012-09-01 14:30:19 UTC,-73.977048,40.758461,-73.984899,40.744693,2,6
9.5,2012-11-08 13:28:07 UTC,-73.969402,40.757545,-73.950049,40.776079,1,7
9.0,2014-07-15 11:37:25 UTC,-73.979318,40.760949,-73.95767,40.773724,1,8
3.3,2009-11-09 18:06:58 UTC,-73.955675,40.779154,-73.961172,40.772368,1,9

==> ../data/taxi-valid.csv <==
5.3,2012-01-03 19:21:35 UTC,-73.962627,40.763214,-73.973485,40.753353,1,0
25.3,2010-09-27 07:30:15 UTC,-73.965799,40.794243,-73.927134,40.852261,3,1
27.5,2015-05-19 00:40:02 UTC,-73.86344146728516,40.76899719238281,-73.96058654785156,40.
76129913330078,1,2
5.7,2010-04-29 12:28:00 UTC,-73.989255,40.738912,-73.97558,40.749172,1,3
11.5,2013-06-23 06:08:09 UTC,-73.99731,40.763735,-73.955657,40.768141,1,4
18.0,2014-10-14 18:52:03 UTC,-73.997995,40.761638,-74.008985,40.712442,1,5
4.9,2010-04-29 12:28:00 UTC,-73.977315,40.766182,-73.970845,40.761462,5,6
32.33,2014-02-24 18:22:00 UTC,-73.985358,40.761352,-73.92427,40.699145,1,7
17.0,2015-03-26 02:48:58 UTC,-73.93981170654297,40.846473693847656,-73.97361755371094,4
```

```
0.786983489990234,1,8
12.5,2013-04-09 09:39:13 UTC,-73.977323,40.753934,-74.00719,40.741472,1,9
```

Use tf.data to read the CSV files

We wrote these functions for reading data from the csv files above in the previous notebook.

In [ ]:

```
# Defining the feature names into a list `CSV_COLUMNS`
CSV_COLUMNS = [
    'fare_amount',
    'pickup_datetime',
    'pickup_longitude',
    'pickup_latitude',
    'dropoff_longitude',
    'dropoff_latitude',
    'passenger_count',
    'key'
]
LABEL_COLUMN = 'fare_amount'
# Defining the default values into a list `DEFAULTS`
DEFAULTS = [[0.0], ['na'], [0.0], [0.0], [0.0], [0.0], [0.0], ['na']]
UNWANTED_COLS = ['pickup_datetime', 'key']


def features_and_labels(row_data):
# The .pop() method will return item and drop from frame.
    label = row_data.pop(LABEL_COLUMN)
    features = row_data

    for unwanted_col in UNWANTED_COLS:
        features.pop(unwanted_col)

    return features, label


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

# The map() function executes a specified function for each item in an iterable.
# The item is sent to the function as a parameter.
    dataset = dataset.map(features_and_labels)

    if mode == 'train':
# The shuffle() method takes a sequence (list, string, or tuple) and reorganize the ord
        dataset = dataset.shuffle(buffer_size=1000).repeat()

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

Build a simple keras DNN model

We will use feature columns to connect our raw data to our keras DNN model. Feature columns make it easy to perform common types of feature engineering on your raw data. For example, you can one-hot encode categorical data, create feature crosses, embeddings and more. We'll cover these in more detail later in the course, but if you want to a sneak peak browse the official TensorFlow feature columns guide.

In our case we won't do any feature engineering. However, we still need to create a list of feature columns to specify the numeric values which will be passed on to our model. To do this, we use `tf.feature_column.numeric_column()`

We use a python dictionary comprehension to create the feature columns for our model, which is just an elegant alternative to a for loop.

In [ ]:

```
# Defining the feature names into a list `INPUT_COLS`
INPUT_COLS = [
    'pickup_longitude',
    'pickup_latitude',
    'dropoff_longitude',
    'dropoff_latitude',
    'passenger_count',
]

# Create input layer of feature columns
# TODO 1
feature_columns = {
    colname: tf.feature_column.numeric_column(colname)
    for colname in INPUT_COLS
    }
```

Next, we create the DNN model. The Sequential model is a linear stack of layers and when building a model using the Sequential API, you configure each layer of the model in turn. Once all the layers have been added, you compile the model.

In [ ]:

```
# Build a keras DNN model using Sequential API
# TODO 2a
model = Sequential([
    DenseFeatures(feature_columns=feature_columns.values()),
    Dense(units=32, activation="relu", name="h1"),
    Dense(units=8, activation="relu", name="h2"),
    Dense(units=1, activation="linear", name="output")
    ])
```

Next, to prepare the model for training, you must configure the learning process. This is done using the compile method. The compile method takes three arguments:

- An optimizer. This could be the string identifier of an existing optimizer (such as `rmsprop` or `adagrad`), or an instance of the Optimizer class.
- A loss function. This is the objective that the model will try to minimize. It can be the string identifier of an existing loss function from the Losses class (such as categorical_crossentropy or mse), or it can be a custom objective function.
- A list of metrics. For any machine learning problem you will want a set of metrics to evaluate your model. A metric could be the string identifier of an existing metric or a custom metric function.

We will add an additional custom metric called `rmse` to our list of metrics which will return the root mean square error.

In [ ]:
```
# TODO 2b
# Create a custom evalution metric
def rmse(y_true, y_pred):
    return tf.sqrt(tf.reduce_mean(tf.square(y_pred - y_true)))


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

Train the model

To train your model, Keras provides three functions that can be used:

1. `.fit()` for training a model for a fixed number of epochs (iterations on a dataset).
2. `.fit_generator()` for training a model on data yielded batch-by-batch by a generator
3. `.train_on_batch()` runs a single gradient update on a single batch of data.

The `.fit()` function works well for small datasets which can fit entirely in memory. However, for large datasets (or if you need to manipulate the training data on the fly via data augmentation, etc) you will need to use `.fit_generator()` instead. The `.train_on_batch()` method is for more fine-grained control over training and accepts only a single batch of data.

The taxifare dataset we sampled is small enough to fit in memory, so can we could use `.fit` to train our model. Our `create_dataset` function above generates batches of training examples, so we could also use `.fit_generator`. In fact, when calling `.fit` the method inspects the data, and if it's a generator (as our dataset is) it will invoke automatically `.fit_generator` for training.

We start by setting up some parameters for our training job and create the data generators for the training and validation data.

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 [ ]:
```
TRAIN_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=TRAIN_BATCH_SIZE,
    mode='train')

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

There are various arguments you can set when calling the .fit method. Here `x` specifies the input data which in our case is a `tf.data` dataset returning a tuple of (inputs, targets). The `steps_per_epoch` parameter is used to mark the end of training for a single epoch. Here we are training for NUM_EVALS epochs. Lastly, for the `callback` argument we specify a Tensorboard callback so we can inspect Tensorboard after training.

In [ ]:

```
%time
# TODO 3
steps_per_epoch = NUM_TRAIN_EXAMPLES // (TRAIN_BATCH_SIZE * NUM_EVALS)

LOGDIR = "./taxi_trained"
# Train the sequential model
history = model.fit(x=trainds,
                    steps_per_epoch=steps_per_epoch,
                    epochs=NUM_EVALS,
                    validation_data=evalds,
                    callbacks=[TensorBoard(LOGDIR)])
```

```
CPU times: user 0 ns, sys: 0 ns, total: 0 ns
Wall time: 6.44 µs
Train for 1 steps, validate for 10 steps
Epoch 1/50
1/1 [==============================] - 14s 14s/step - loss: 561.1435 - rmse: 23.6885 - m
se: 561.1435 - val_loss: 538.9352 - val_rmse: 23.2127 - val_mse: 538.9352
Epoch 2/50
WARNING:tensorflow:Method (on_train_batch_end) is slow compared to the batch update (0.4
60273). Check your callbacks.
1/1 [==============================] - 1s 656ms/step - loss: 501.7719 - rmse: 22.4003 -
mse: 501.7719 - val_loss: 499.9065 - val_rmse: 22.3571 - val_mse: 499.9065
Epoch 3/50
1/1 [==============================] - 0s 201ms/step - loss: 463.3914 - rmse: 21.5265 -
mse: 463.3914 - val_loss: 466.3172 - val_rmse: 21.5895 - val_mse: 466.3172
Epoch 4/50
1/1 [==============================] - 0s 276ms/step - loss: 430.6084 - rmse: 20.7511 -
mse: 430.6084 - val_loss: 430.4976 - val_rmse: 20.7423 - val_mse: 430.4977
Epoch 5/50
1/1 [==============================] - 0s 253ms/step - loss: 417.5044 - rmse: 20.4329 -
mse: 417.5044 - val_loss: 397.8781 - val_rmse: 19.9455 - val_mse: 397.8781
Epoch 6/50
1/1 [==============================] - 0s 262ms/step - loss: 345.4152 - rmse: 18.5853 -
mse: 345.4152 - val_loss: 366.7722 - val_rmse: 19.1435 - val_mse: 366.7722
Epoch 7/50
1/1 [==============================] - 0s 253ms/step - loss: 363.9706 - rmse: 19.0780 -
mse: 363.9706 - val_loss: 341.5944 - val_rmse: 18.4763 - val_mse: 341.5944
Epoch 8/50
1/1 [==============================] - 0s 232ms/step - loss: 289.2403 - rmse: 17.0071 -
mse: 289.2403 - val_loss: 314.0481 - val_rmse: 17.7171 - val_mse: 314.0481
Epoch 9/50
1/1 [==============================] - 0s 241ms/step - loss: 286.2377 - rmse: 16.9186 -
mse: 286.2377 - val_loss: 303.3531 - val_rmse: 17.4117 - val_mse: 303.3531
Epoch 10/50
1/1 [==============================] - 0s 220ms/step - loss: 257.0701 - rmse: 16.0334 -
mse: 257.0701 - val_loss: 300.9174 - val_rmse: 17.3433 - val_mse: 300.9174
Epoch 11/50
1/1 [==============================] - 0s 185ms/step - loss: 271.4450 - rmse: 16.4756 -
mse: 271.4450 - val_loss: 295.7931 - val_rmse: 17.1898 - val_mse: 295.7932
Epoch 12/50
1/1 [==============================] - 0s 196ms/step - loss: 250.6559 - rmse: 15.8321 -
mse: 250.6559 - val_loss: 291.8766 - val_rmse: 17.0752 - val_mse: 291.8766
Epoch 13/50
```

```
1/1 [==============================] - 0s 198ms/step - loss: 272.0665 - rmse: 16.4944 -
mse: 272.0665 - val_loss: 290.6524 - val_rmse: 17.0397 - val_mse: 290.6524
Epoch 14/50
1/1 [==============================] - 0s 227ms/step - loss: 256.3561 - rmse: 16.0111 -
mse: 256.3561 - val_loss: 288.1296 - val_rmse: 16.9712 - val_mse: 288.1296
Epoch 15/50
1/1 [==============================] - 0s 185ms/step - loss: 250.8681 - rmse: 15.8388 -
mse: 250.8681 - val_loss: 284.6306 - val_rmse: 16.8609 - val_mse: 284.6306
Epoch 16/50
1/1 [==============================] - 0s 198ms/step - loss: 298.4770 - rmse: 17.2765 -
mse: 298.4770 - val_loss: 282.3497 - val_rmse: 16.7934 - val_mse: 282.3497
Epoch 17/50
1/1 [==============================] - 0s 208ms/step - loss: 232.7821 - rmse: 15.2572 -
mse: 232.7821 - val_loss: 276.3227 - val_rmse: 16.6097 - val_mse: 276.3227
Epoch 18/50
1/1 [==============================] - 0s 209ms/step - loss: 233.9045 - rmse: 15.2939 -
mse: 233.9045 - val_loss: 274.2226 - val_rmse: 16.5548 - val_mse: 274.2226
Epoch 19/50
1/1 [==============================] - 0s 190ms/step - loss: 241.5300 - rmse: 15.5412 -
mse: 241.5300 - val_loss: 273.8148 - val_rmse: 16.5399 - val_mse: 273.8148
Epoch 20/50
1/1 [==============================] - 0s 200ms/step - loss: 231.1722 - rmse: 15.2043 -
mse: 231.1722 - val_loss: 270.3832 - val_rmse: 16.4390 - val_mse: 270.3832
Epoch 21/50
1/1 [==============================] - 0s 232ms/step - loss: 244.5873 - rmse: 15.6393 -
mse: 244.5873 - val_loss: 268.6810 - val_rmse: 16.3805 - val_mse: 268.6810
Epoch 22/50
1/1 [==============================] - 0s 203ms/step - loss: 245.4552 - rmse: 15.6670 -
mse: 245.4552 - val_loss: 266.0358 - val_rmse: 16.3049 - val_mse: 266.0358
Epoch 23/50
1/1 [==============================] - 0s 188ms/step - loss: 232.4738 - rmse: 15.2471 -
mse: 232.4738 - val_loss: 263.8402 - val_rmse: 16.2393 - val_mse: 263.8402
Epoch 24/50
1/1 [==============================] - 0s 190ms/step - loss: 221.9046 - rmse: 14.8965 -
mse: 221.9046 - val_loss: 258.3069 - val_rmse: 16.0616 - val_mse: 258.3069
Epoch 25/50
1/1 [==============================] - 0s 211ms/step - loss: 217.2153 - rmse: 14.7382 -
mse: 217.2153 - val_loss: 259.5989 - val_rmse: 16.1060 - val_mse: 259.5989
Epoch 26/50
1/1 [==============================] - 0s 197ms/step - loss: 251.1073 - rmse: 15.8464 -
mse: 251.1073 - val_loss: 257.2844 - val_rmse: 16.0286 - val_mse: 257.2844
Epoch 27/50
1/1 [==============================] - 0s 211ms/step - loss: 204.0089 - rmse: 14.2832 -
mse: 204.0089 - val_loss: 253.0699 - val_rmse: 15.8976 - val_mse: 253.0699
Epoch 28/50
1/1 [==============================] - 0s 231ms/step - loss: 218.9465 - rmse: 14.7968 -
mse: 218.9465 - val_loss: 252.1912 - val_rmse: 15.8734 - val_mse: 252.1912
Epoch 29/50
1/1 [==============================] - 0s 188ms/step - loss: 245.2914 - rmse: 15.6618 -
mse: 245.2914 - val_loss: 250.6115 - val_rmse: 15.8177 - val_mse: 250.6115
Epoch 30/50
1/1 [==============================] - 0s 214ms/step - loss: 221.9627 - rmse: 14.8984 -
mse: 221.9627 - val_loss: 249.7443 - val_rmse: 15.7994 - val_mse: 249.7444
Epoch 31/50
1/1 [==============================] - 0s 215ms/step - loss: 216.0065 - rmse: 14.6972 -
mse: 216.0065 - val_loss: 251.0478 - val_rmse: 15.8414 - val_mse: 251.0477
Epoch 32/50
1/1 [==============================] - 0s 252ms/step - loss: 227.5754 - rmse: 15.0856 -
mse: 227.5754 - val_loss: 250.0740 - val_rmse: 15.8082 - val_mse: 250.0740
Epoch 33/50
1/1 [==============================] - 0s 244ms/step - loss: 230.6432 - rmse: 15.1869 -
mse: 230.6432 - val_loss: 250.5666 - val_rmse: 15.8279 - val_mse: 250.5665
Epoch 34/50
1/1 [==============================] - 0s 205ms/step - loss: 217.8810 - rmse: 14.7608 -
mse: 217.8810 - val_loss: 252.1180 - val_rmse: 15.8701 - val_mse: 252.1180
```

```
Epoch 35/50
1/1 [==============================] - 0s 238ms/step - loss: 209.2079 - rmse: 14.4640 -
mse: 209.2079 - val_loss: 248.9013 - val_rmse: 15.7699 - val_mse: 248.9012
Epoch 36/50
1/1 [==============================] - 0s 270ms/step - loss: 266.9137 - rmse: 16.3375 -
mse: 266.9137 - val_loss: 249.1861 - val_rmse: 15.7715 - val_mse: 249.1861
Epoch 37/50
1/1 [==============================] - 0s 247ms/step - loss: 239.9447 - rmse: 15.4901 -
mse: 239.9447 - val_loss: 248.4395 - val_rmse: 15.7570 - val_mse: 248.4395
Epoch 38/50
1/1 [==============================] - 0s 192ms/step - loss: 220.2974 - rmse: 14.8424 -
mse: 220.2974 - val_loss: 250.9961 - val_rmse: 15.8313 - val_mse: 250.9961
Epoch 39/50
1/1 [==============================] - 0s 240ms/step - loss: 203.5481 - rmse: 14.2670 -
mse: 203.5481 - val_loss: 250.4058 - val_rmse: 15.8156 - val_mse: 250.4058
Epoch 40/50
1/1 [==============================] - 0s 188ms/step - loss: 207.8423 - rmse: 14.4167 -
mse: 207.8423 - val_loss: 249.7776 - val_rmse: 15.7993 - val_mse: 249.7776
Epoch 41/50
1/1 [==============================] - 0s 213ms/step - loss: 199.6897 - rmse: 14.1312 -
mse: 199.6897 - val_loss: 249.4542 - val_rmse: 15.7859 - val_mse: 249.4542
Epoch 42/50
1/1 [==============================] - 0s 172ms/step - loss: 224.3715 - rmse: 14.9790 -
mse: 224.3715 - val_loss: 244.6346 - val_rmse: 15.6317 - val_mse: 244.6346
Epoch 43/50
1/1 [==============================] - 0s 216ms/step - loss: 211.3754 - rmse: 14.5388 -
mse: 211.3754 - val_loss: 248.7197 - val_rmse: 15.7594 - val_mse: 248.7197
Epoch 44/50
1/1 [==============================] - 0s 200ms/step - loss: 208.2830 - rmse: 14.4320 -
mse: 208.2830 - val_loss: 249.0528 - val_rmse: 15.7667 - val_mse: 249.0528
Epoch 45/50
1/1 [==============================] - 0s 199ms/step - loss: 237.6899 - rmse: 15.4172 -
mse: 237.6899 - val_loss: 248.0569 - val_rmse: 15.7402 - val_mse: 248.0570
Epoch 46/50
1/1 [==============================] - 0s 220ms/step - loss: 216.7469 - rmse: 14.7223 -
mse: 216.7469 - val_loss: 247.8345 - val_rmse: 15.7375 - val_mse: 247.8345
Epoch 47/50
1/1 [==============================] - 0s 186ms/step - loss: 262.1243 - rmse: 16.1903 -
mse: 262.1243 - val_loss: 247.1873 - val_rmse: 15.7175 - val_mse: 247.1873
Epoch 48/50
1/1 [==============================] - 0s 173ms/step - loss: 212.1745 - rmse: 14.5662 -
mse: 212.1745 - val_loss: 247.0517 - val_rmse: 15.7084 - val_mse: 247.0517
Epoch 49/50
1/1 [==============================] - 0s 182ms/step - loss: 210.8718 - rmse: 14.5214 -
mse: 210.8718 - val_loss: 249.0218 - val_rmse: 15.7772 - val_mse: 249.0218
Epoch 50/50
1/1 [==============================] - 0s 223ms/step - loss: 243.6624 - rmse: 15.6097 -
mse: 243.6624 - val_loss: 247.9750 - val_rmse: 15.7405 - val_mse: 247.9749
```

High-level model evaluation

Once we've run data through the model, we can call `.summary()` on the model to get a high-level summary of our network. We can also plot the training and evaluation curves for the metrics we computed above.

In [ ]:

```
# The summary() is a generic function used to produce result summaries of the results o
model.summary()
```

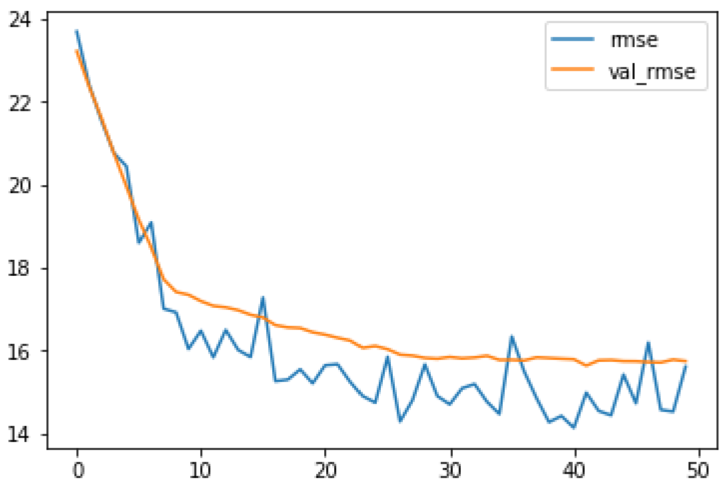
```
Model: sequential
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
```

```
dense_features (DenseFeature multiple                  0
_________________________________________________________________
h1 (Dense)                   multiple                  192
_________________________________________________________________
h2 (Dense)                   multiple                  264
_________________________________________________________________
output (Dense)               multiple                  9
=================================================================
Total params: 465
Trainable params: 465
Non-trainable params: 0
_________________________________________________________________
```

Running `.fit` (or `.fit_generator`) returns a History object which collects all the events recorded during training. Similar to Tensorboard, we can plot the training and validation curves for the model loss and rmse by accessing these elements of the History object.

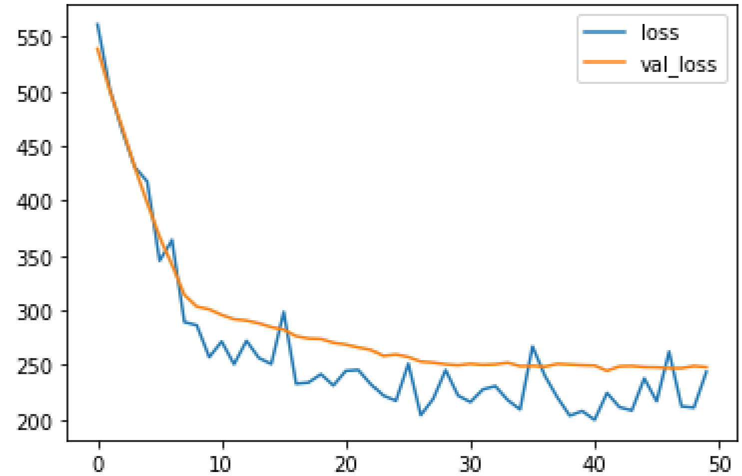
In [ ]:

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

# The history object is returned from calls to the fit() function used to train the mod
# Metrics are stored in a dictionary in the history member of the object returned.
pd.DataFrame(history.history)[RMSE_COLS].plot()
```



In [ ]:

```
LOSS_COLS = ['loss', 'val_loss']

# The history object is returned from calls to the fit() function used to train the mod
# Metrics are stored in a dictionary in the history member of the object returned.
pd.DataFrame(history.history)[LOSS_COLS].plot()
```



Making predictions with our model

To make predictions with our trained model, we can call the predict method, passing to it a dictionary of values. The `steps` parameter determines the total number of steps before declaring the prediction round finished. Here since we have just one example, we set `steps=1` (setting `steps=None` would also work). Note, however, that if x is a `tf.data` dataset or a dataset iterator, and steps is set to None, predict will run until the input dataset is exhausted.

In [ ]:

```
# The predict() method will predict the response for model.
# Using tf.convert_to_tensor() we will convert the given value to a Tensor.
model.predict(x={"pickup_longitude": tf.convert_to_tensor([-73.982683]),
                 "pickup_latitude": tf.convert_to_tensor([40.742104]),
                 "dropoff_longitude": tf.convert_to_tensor([-73.983766]),
                 "dropoff_latitude": tf.convert_to_tensor([40.755174]),
                 "passenger_count": tf.convert_to_tensor([3.0])},
              steps=1)
```

```
array([[0.04512187]], dtype=float32)
```

Export and deploy our model

Of course, making individual predictions is not realistic, because we can't expect client code to have a model object in memory. For others to use our trained model, we'll have to export our model to a file, and expect client code to instantiate the model from that exported file.

We'll export the model to a TensorFlow SavedModel format. Once we have a model in this format, we have lots of ways to "serve" the model, from a web application, from JavaScript, from mobile applications, etc.

In [ ]:

```
# TODO 4a
OUTPUT_DIR = "./export/savedmodel"
shutil.rmtree(OUTPUT_DIR, ignore_errors=True)
# The join() method takes all items in an iterable and joins them into one string.
EXPORT_PATH = os.path.join(OUTPUT_DIR,
```

```
                           datetime.datetime.now().strftime("%Y%m%d%H%M%S"))
tf.saved_model.save(model, EXPORT_PATH)  # with default serving function
```

```
WARNING:tensorflow:From /opt/conda/lib/python3.7/site-packages/tensorflow_core/python/op
s/resource_variable_ops.py:1786: calling BaseResourceVariable.__init__ (from tensorflow.
python.ops.resource_variable_ops) with constraint is deprecated and will be removed in a
future version.
Instructions for updating:
If using Keras pass *_constraint arguments to layers.
INFO:tensorflow:Assets written to: ./export/savedmodel/20200910125428/assets
```

In [ ]:

```
# Export the model to a TensorFlow SavedModel format
# TODO 4b
!saved_model_cli show \
 --tag_set serve \
 --signature_def serving_default \
 --dir {EXPORT_PATH}
!find {EXPORT_PATH}
os.environ['EXPORT_PATH'] = EXPORT_PATH
```

```
2020-09-10 12:55:19.479572: W tensorflow/stream_executor/platform/default/dso_loader.cc:
55] Could not load dynamic library 'libnvinfer.so.6'; dlerror: libnvinfer.so.6: cannot o
pen shared object file: No such file or directory; LD_LIBRARY_PATH: /usr/local/cuda/lib6
4:/usr/local/nccl2/lib:/usr/local/cuda/extras/CUPTI/lib64
2020-09-10 12:55:19.479674: W tensorflow/stream_executor/platform/default/dso_loader.cc:
55] Could not load dynamic library 'libnvinfer_plugin.so.6'; dlerror: libnvinfer_plugin.
so.6: cannot open shared object file: No such file or directory; LD_LIBRARY_PATH: /usr/l
ocal/cuda/lib64:/usr/local/nccl2/lib:/usr/local/cuda/extras/CUPTI/lib64
2020-09-10 12:55:19.479755: W tensorflow/compiler/tf2tensorrt/utils/py_utils.cc:30] Cann
ot dlopen some TensorRT libraries. If you would like to use Nvidia GPU with TensorRT, pl
ease make sure the missing libraries mentioned above are installed properly.
The given SavedModel SignatureDef contains the following input(s):
inputs['dropoff_latitude'] tensor_info:
dtype: DT_FLOAT
shape: (-1, 1)
name: serving_default_dropoff_latitude:0
inputs['dropoff_longitude'] tensor_info:
dtype: DT_FLOAT
shape: (-1, 1)
name: serving_default_dropoff_longitude:0
inputs['passenger_count'] tensor_info:
dtype: DT_FLOAT
shape: (-1, 1)
name: serving_default_passenger_count:0
inputs['pickup_latitude'] tensor_info:
dtype: DT_FLOAT
shape: (-1, 1)
name: serving_default_pickup_latitude:0
inputs['pickup_longitude'] tensor_info:
dtype: DT_FLOAT
shape: (-1, 1)
name: serving_default_pickup_longitude:0
The given SavedModel SignatureDef contains the following output(s):
outputs['output_1'] tensor_info:
dtype: DT_FLOAT
shape: (-1, 1)
name: StatefulPartitionedCall:0
Method name is: tensorflow/serving/predict
./export/savedmodel/20200910125428
./export/savedmodel/20200910125428/assets
./export/savedmodel/20200910125428/saved_model.pb
./export/savedmodel/20200910125428/variables
```

```
./export/savedmodel/20200910125428/variables/variables.index
./export/savedmodel/20200910125428/variables/variables.data-00000-of-00001
```

Deploy our model to AI Platform

Finally, we will deploy our trained model to AI Platform and see how we can make online predicitons.

In [ ]:

```
%%bash
gcloud config set compute/region us-east1
gcloud config set ai_platform/region global
```

```
Updated property [compute/region].
Updated property [ai_platform/region].
```

Below cell will take around 10 minutes to complete.

In [ ]:

```
%%bash

# TODO 5a

PROJECT= # TODO: Change this to your PROJECT
BUCKET=${PROJECT}
REGION=us-east1
MODEL_NAME=taxifare
VERSION_NAME=dnn

# Create GCS bucket if it doesn't exist already...
exists=$(gsutil ls -d | grep -w gs://${BUCKET}/)

if [ -n "$exists" ]; then
    echo -e "Bucket exists, let's not recreate it."
else
    echo "Creating a new GCS bucket."
    gsutil mb -l ${REGION} gs://${BUCKET}
    echo "Here are your current buckets:"
    gsutil ls
fi

if [[ $(gcloud ai-platform models list --format='value(name)' --region=$REGION | grep $
    echo "$MODEL_NAME already exists"
else
    echo "Creating $MODEL_NAME"
    gcloud ai-platform models create --region=$REGION $MODEL_NAME
fi

if [[ $(gcloud ai-platform versions list --model $MODEL_NAME --region=$REGION --format=
    echo "Deleting already existing $MODEL_NAME:$VERSION_NAME ... "
    echo yes | gcloud ai-platform versions delete --model=$MODEL_NAME $VERSION_NAME --r
    echo "Please run this cell again if you don't see a Creating message ... "
    sleep 2
fi

echo "Creating $MODEL_NAME:$VERSION_NAME"
gcloud ai-platform versions create --model=$MODEL_NAME $VERSION_NAME \
       --framework=tensorflow --python-version=3.7 --runtime-version=2.1 \
       --origin=$EXPORT_PATH --staging-bucket=gs://$BUCKET --region=$REGION
```

```
Creating a new GCS bucket.
```

```
Here are your current buckets:
gs://qwiklabs-gcp-00-eae83af0ede5/
taxifare already exists
Creating taxifare:dnn
Creating gs://qwiklabs-gcp-00-eae83af0ede5/...
Using endpoint [https://ml.googleapis.com/]
Using endpoint [https://ml.googleapis.com/]
Using endpoint [https://ml.googleapis.com/]
Creating version (this might take a few minutes)......
.................................................................
..................................................................
...................................................................
...................................................................
.....................................................done.
```

In [ ]:

```
%%writefile input.json
{"pickup_longitude": -73.982683, "pickup_latitude": 40.742104,"dropoff_longitude": -73.
```

```
Writing input.json
```

In [ ]:

```
# The `gcloud ai-platform predict` sends a prediction request to AI Platform for the gi
# TODO 5b
!gcloud ai-platform predict --model taxifare --json-instances input.json --version dnn
```

```
Using endpoint [https://ml.googleapis.com/]
OUTPUT_1
[0.04512186720967293]
```