representation_with_a_feature_cross

August 26, 2021

[1]:
```
from jupyterthemes import jtplot
jtplot.style(theme="onedork", figsize=(14, 5))
```

[ ]:
```
#@title Copyright 2020 Google LLC. Double-click for license information.
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

1 Representation with a Feature Cross

In this exercise, you'll experiment with different ways to represent features.

1.1 Learning Objectives:

After doing this Colab, you'll know how to:

- Use `tf.feature_column` methods to represent features in different ways.
- Represent features as bins.
- Cross bins to create a feature cross.

1.2 The Dataset

Like several of the previous Colabs, this exercise uses the California Housing Dataset.

1.3 Call the import statements

The following code imports the necessary code to run the code in the rest of this Colaboratory.

[2]:
```
#@title Load the imports
```

```
# from __future__ import absolute_import, division, print_function,␣
↪unicode_literals

import numpy as np
import pandas as pd
import tensorflow as tf
from tensorflow import feature_column
from tensorflow.keras import layers

from matplotlib import pyplot as plt

# The following lines adjust the granularity of reporting.
pd.options.display.max_rows = 10
pd.options.display.float_format = "{:.1f}".format

tf.keras.backend.set_floatx('float32')

print("Imported the modules.")
```

```
Imported the modules.
```

1.4 Load, scale, and shuffle the examples

The following code cell loads the separate .csv files and creates the following two pandas DataFrames:

- `train_df`, which contains the training set
- `test_df`, which contains the test set

The code cell then scales the `median_house_value` to a more human-friendly range and then suffles the examples.

[3]:
```
# Load the dataset
train_df = pd.read_csv("https://download.mlcc.google.com/mledu-datasets/
↪california_housing_train.csv")
test_df = pd.read_csv("https://download.mlcc.google.com/mledu-datasets/
↪california_housing_test.csv")

# Scale the labels
scale_factor = 1000.0
# Scale the training set's label.
train_df["median_house_value"] /= scale_factor

# Scale the test set's label
test_df["median_house_value"] /= scale_factor

# Shuffle the examples
train_df = train_df.reindex(np.random.permutation(train_df.index))
```

1.5 Represent latitude and longitude as floating-point values

Previous Colabs trained on only a single feature or a single synthetic feature. By contrast, this exercise trains on two features. Furthermore, this Colab introduces **feature columns**, which provide a sophisticated way to represent features.

You create feature columns as possible:

- Call a `tf.feature_column` method to represent a single feature, single feature cross, or single synthetic feature in the desired way. For example, to represent a certain feature as floating-point values, call `tf.feature_column.numeric_column`. To represent a certain feature as a series of buckets or bins, call `tf.feature_column.bucketized_column`.
- Assemble the created representations into a Python list.

A neighborhood's location is typically the most important feature in determining a house's value. The California Housing dataset provides two features, `latitude` and `longitude` that identify each neighborhood's location.

The following code cell calls `tf.feature_column.numeric_column` twice, first to represent `latitude` as floating-point value and a second time to represent `longitude` as floating-point values.

This code cell specifies the features that you'll ultimately train the model on and how each of those features will be represented. The transformations (collected in `fp_feature_layer`) don't actually get applied until you pass a DataFrame to it, which will happen when we train the model.

[4]:
```
# Create an empty list that will eventually hold all feature columns.
feature_columns = []

# Create a numerical feature column to represent latitude.
latitude = tf.feature_column.numeric_column("latitude")
feature_columns.append(latitude)

# Create a numerical feature column to represent longitude.
longitude = tf.feature_column.numeric_column("longitude")
feature_columns.append(longitude)

# Convert the list of feature columns into a layer that will ultimately become
# part of the model. Understanding layers is not important right now.
fp_feature_layer = layers.DenseFeatures(feature_columns)
```

When used, the layer processes the raw inputs, according to the transformations described by the feature columns, and packs the result into a numeric array. (The model will train on this numeric array.)

1.6 Define functions that create and train a model, and a plotting function

The following code defines three functions:

- `create_model`, which tells TensorFlow to build a linear regression model and to use the `feature_layer_as_fp` as the representation of the model's features.
- `train_model`, which will ultimately train the model from training set examples.
- `plot_the_loss_curve`, which generates a loss curve.

[5]:

```python
#@title Define functions to create and train a model, and a plotting function
def create_model(my_learning_rate, feature_layer):
  """Create and compile a simple linear regression model."""
  # Most simple tf.keras models are sequential.
  model = tf.keras.models.Sequential()

  # Add the layer containing the feature columns to the model.
  model.add(feature_layer)

  # Add one linear layer to the model to yield a simple linear regressor.
  model.add(tf.keras.layers.Dense(units=1, input_shape=(1,)))

  # Construct the layers into a model that TensorFlow can execute.
  model.compile(optimizer=tf.keras.optimizers.
↪RMSprop(learning_rate=my_learning_rate),
                loss="mean_squared_error",
                metrics=[tf.keras.metrics.RootMeanSquaredError()])

  return model


def train_model(model, dataset, epochs, batch_size, label_name):
  """Feed a dataset into the model in order to train it."""

  features = {name:np.array(value) for name, value in dataset.items()}
  label = np.array(features.pop(label_name))
  history = model.fit(x=features, y=label, batch_size=batch_size,
                      epochs=epochs, shuffle=True)

  # The list of epochs is stored separately from the rest of history.
  epochs = history.epoch

  # Isolate the mean absolute error for each epoch.
  hist = pd.DataFrame(history.history)
  rmse = hist["root_mean_squared_error"]

  return epochs, rmse


def plot_the_loss_curve(epochs, rmse):
  """Plot a curve of loss vs. epoch."""

  plt.figure()
  plt.xlabel("Epoch")
  plt.ylabel("Root Mean Squared Error")

  plt.plot(epochs, rmse, label="Loss")
```

```
    plt.legend()
    plt.ylim([rmse.min()*0.94, rmse.max()* 1.05])
    plt.show()

print("Defined the create_model, train_model, and plot_the_loss_curve functions.
 ↪")
```

```
Defined the create_model, train_model, and plot_the_loss_curve functions.
```

1.7 Train the model with floating-point representations

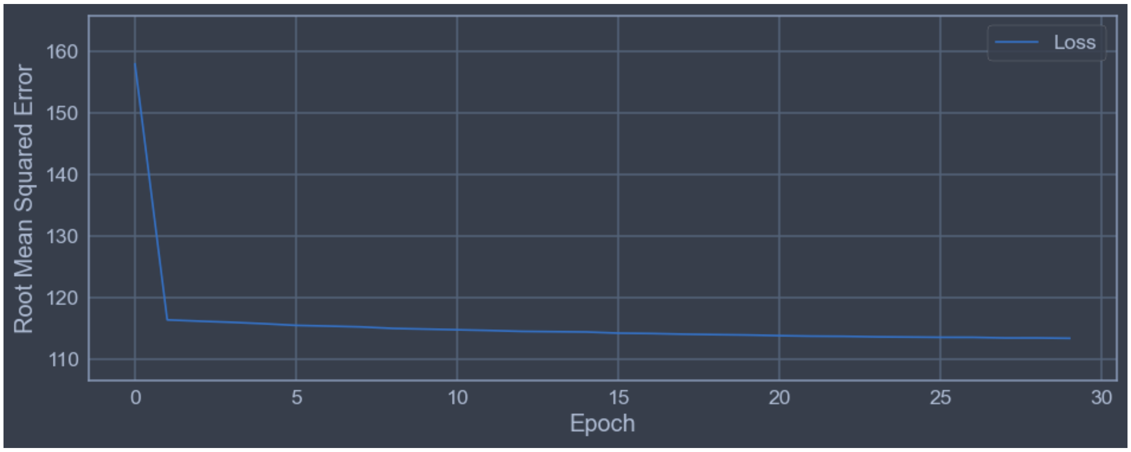
The following code cell calls the functions you just created to train, plot, and evaluate a model.

[6]:
```
# The following variables are the hyperparameters.
learning_rate = 0.05
epochs = 30
batch_size = 100
label_name = 'median_house_value'

# Create and compile the model's topography.
my_model = create_model(learning_rate, fp_feature_layer)

# Train the model on the training set.
epochs, rmse = train_model(my_model, train_df, epochs, batch_size, label_name)

plot_the_loss_curve(epochs, rmse)

print("\n: Evaluate the new model against the test set:")
test_features = {name:np.array(value) for name, value in test_df.items()}
test_label = np.array(test_features.pop(label_name))
my_model.evaluate(x=test_features, y=test_label, batch_size=batch_size)
```

```
Epoch 1/30
WARNING:tensorflow:Layers in a Sequential model should only have a single input
tensor, but we receive a <class 'dict'> input: {'longitude': <tf.Tensor
'ExpandDims_3:0' shape=(100, 1) dtype=float32>, 'latitude': <tf.Tensor
'ExpandDims_2:0' shape=(100, 1) dtype=float32>, 'housing_median_age': <tf.Tensor
'ExpandDims_1:0' shape=(100, 1) dtype=float32>, 'total_rooms': <tf.Tensor
'ExpandDims_7:0' shape=(100, 1) dtype=float32>, 'total_bedrooms': <tf.Tensor
'ExpandDims_6:0' shape=(100, 1) dtype=float32>, 'population': <tf.Tensor
'ExpandDims_5:0' shape=(100, 1) dtype=float32>, 'households': <tf.Tensor
'ExpandDims:0' shape=(100, 1) dtype=float32>, 'median_income': <tf.Tensor
'ExpandDims_4:0' shape=(100, 1) dtype=float32>}
Consider rewriting this model with the Functional API.
WARNING:tensorflow:Layers in a Sequential model should only have a single input
tensor, but we receive a <class 'dict'> input: {'longitude': <tf.Tensor
'ExpandDims_3:0' shape=(100, 1) dtype=float32>, 'latitude': <tf.Tensor
'ExpandDims_2:0' shape=(100, 1) dtype=float32>, 'housing_median_age': <tf.Tensor
```

```
'ExpandDims_1:0' shape=(100, 1) dtype=float32>, 'total_rooms': <tf.Tensor
'ExpandDims_7:0' shape=(100, 1) dtype=float32>, 'total_bedrooms': <tf.Tensor
'ExpandDims_6:0' shape=(100, 1) dtype=float32>, 'population': <tf.Tensor
'ExpandDims_5:0' shape=(100, 1) dtype=float32>, 'households': <tf.Tensor
'ExpandDims:0' shape=(100, 1) dtype=float32>, 'median_income': <tf.Tensor
'ExpandDims_4:0' shape=(100, 1) dtype=float32>}
Consider rewriting this model with the Functional API.
170/170 [==============================] - 4s 1ms/step - loss: 48967.7785 -
root_mean_squared_error: 213.1454
Epoch 2/30
170/170 [==============================] - 0s 1ms/step - loss: 13577.8688 -
root_mean_squared_error: 116.5125
Epoch 3/30
170/170 [==============================] - 0s 1ms/step - loss: 13212.2934 -
root_mean_squared_error: 114.9330
Epoch 4/30
170/170 [==============================] - 0s 1ms/step - loss: 13494.9081 -
root_mean_squared_error: 116.1651
Epoch 5/30
170/170 [==============================] - 0s 1ms/step - loss: 13662.1923 -
root_mean_squared_error: 116.8796
Epoch 6/30
170/170 [==============================] - 0s 1ms/step - loss: 13080.7060 -
root_mean_squared_error: 114.3563
Epoch 7/30
170/170 [==============================] - 0s 1ms/step - loss: 13298.3145 -
root_mean_squared_error: 115.3080
Epoch 8/30
170/170 [==============================] - 0s 1ms/step - loss: 13312.5290 -
root_mean_squared_error: 115.3703
Epoch 9/30
170/170 [==============================] - 0s 1ms/step - loss: 13093.2639 -
root_mean_squared_error: 114.4116
Epoch 10/30
170/170 [==============================] - 0s 1ms/step - loss: 13153.6313 -
root_mean_squared_error: 114.6867
Epoch 11/30
170/170 [==============================] - 0s 1ms/step - loss: 12949.4399 -
root_mean_squared_error: 113.7914
Epoch 12/30
170/170 [==============================] - 0s 1ms/step - loss: 13460.4567 -
root_mean_squared_error: 116.0098
Epoch 13/30
170/170 [==============================] - 0s 1ms/step - loss: 13309.8032 -
root_mean_squared_error: 115.3636
Epoch 14/30
170/170 [==============================] - 0s 1ms/step - loss: 13071.6555 -
root_mean_squared_error: 114.3288
```

```
Epoch 15/30
170/170 [==============================] - 0s 1ms/step - loss: 12903.1461 -
root_mean_squared_error: 113.5825
Epoch 16/30
170/170 [==============================] - 0s 1ms/step - loss: 12975.5904 -
root_mean_squared_error: 113.9045
Epoch 17/30
170/170 [==============================] - 0s 1ms/step - loss: 13140.3450 -
root_mean_squared_error: 114.6255
Epoch 18/30
170/170 [==============================] - 0s 1ms/step - loss: 13404.0652 -
root_mean_squared_error: 115.7643
Epoch 19/30
170/170 [==============================] - 0s 1ms/step - loss: 13043.3169 -
root_mean_squared_error: 114.2046
Epoch 20/30
170/170 [==============================] - 0s 1ms/step - loss: 12829.7842 -
root_mean_squared_error: 113.2589
Epoch 21/30
170/170 [==============================] - 0s 1ms/step - loss: 12946.8356 -
root_mean_squared_error: 113.7780
Epoch 22/30
170/170 [==============================] - 0s 1ms/step - loss: 12973.4305 -
root_mean_squared_error: 113.8938
Epoch 23/30
170/170 [==============================] - 0s 1ms/step - loss: 13020.4727 -
root_mean_squared_error: 114.1027
Epoch 24/30
170/170 [==============================] - 0s 1ms/step - loss: 12923.4633 -
root_mean_squared_error: 113.6781
Epoch 25/30
170/170 [==============================] - 0s 1ms/step - loss: 13132.5606 -
root_mean_squared_error: 114.5908
Epoch 26/30
170/170 [==============================] - 0s 1ms/step - loss: 12766.6098 -
root_mean_squared_error: 112.9832
Epoch 27/30
170/170 [==============================] - 0s 1ms/step - loss: 12964.8454 -
root_mean_squared_error: 113.8580
Epoch 28/30
170/170 [==============================] - 0s 1ms/step - loss: 12933.9153 -
root_mean_squared_error: 113.7139
Epoch 29/30
170/170 [==============================] - 0s 1ms/step - loss: 12627.9517 -
root_mean_squared_error: 112.3555
Epoch 30/30
170/170 [==============================] - 0s 1ms/step - loss: 13061.5998 -
root_mean_squared_error: 114.2813
```




```
: Evaluate the new model against the test set:
WARNING:tensorflow:Layers in a Sequential model should only have a single input
tensor, but we receive a <class 'dict'> input: {'longitude': <tf.Tensor
'ExpandDims_3:0' shape=(100, 1) dtype=float32>, 'latitude': <tf.Tensor
'ExpandDims_2:0' shape=(100, 1) dtype=float32>, 'housing_median_age': <tf.Tensor
'ExpandDims_1:0' shape=(100, 1) dtype=float32>, 'total_rooms': <tf.Tensor
'ExpandDims_7:0' shape=(100, 1) dtype=float32>, 'total_bedrooms': <tf.Tensor
'ExpandDims_6:0' shape=(100, 1) dtype=float32>, 'population': <tf.Tensor
'ExpandDims_5:0' shape=(100, 1) dtype=float32>, 'households': <tf.Tensor
'ExpandDims:0' shape=(100, 1) dtype=float32>, 'median_income': <tf.Tensor
'ExpandDims_4:0' shape=(100, 1) dtype=float32>}
Consider rewriting this model with the Functional API.
30/30 [==============================] - 0s 1ms/step - loss: 12190.5361 -
root_mean_squared_error: 110.4108
```

[6]:
```
[12190.5361328125, 110.41075897216797]
```

1.8 Task 1: Why aren't floating-point values a good way to represent latitude and longitude?

Are floating-point values a good way to represent `latitude` and `longitude`?

[ ]:
```
#@title Double-click to view an answer to Task 1.

# No. Representing latitude and longitude as
# floating-point values does not have much
# predictive power. For example, neighborhoods at
# latitude 35 are not 36/35 more valuable
# (or 35/36 less valuable) than houses at
# latitude 36.
```

```
# Representing `latitude` and `longitude` as
# floating-point values provides almost no
# predictive power. We're only using the raw values
# to establish a baseline for future experiments
# with better representations.
```

1.9 Represent latitude and longitude in buckets

The following code cell represents latitude and longitude in buckets (bins). Each bin represents all the neighborhoods within a single degree. For example, neighborhoods at latitude 35.4 and 35.8 are in the same bucket, but neighborhoods in latitude 35.4 and 36.2 are in different buckets.

The model will learn a separate weight for each bucket. For example, the model will learn one weight for all the neighborhoods in the "35" bin", a different weight for neighborhoods in the"36" bin, and so on. This representation will create approximately 20 buckets:

- 10 buckets for `latitude`.
- 10 buckets for `longitude`.

[7]:
```
resolution_in_degrees = 1.0

# Create a new empty list that will eventually hold the generated feature␣
↪column.
feature_columns = []

# Create a bucket feature column for latitude.
latitude_as_a_numeric_column = tf.feature_column.numeric_column("latitude")
latitude_boundaries = list(np.arange(int(min(train_df['latitude'])),
                                     int(max(train_df['latitude'])),
                                     resolution_in_degrees))
latitude = tf.feature_column.bucketized_column(latitude_as_a_numeric_column,
                                               latitude_boundaries)
feature_columns.append(latitude)

# Create a bucket feature column for longitude.
longitude_as_a_numeric_column = tf.feature_column.numeric_column("longitude")
longitude_boundaries = list(np.arange(int(min(train_df['longitude'])),
                                      int(max(train_df['longitude'])),
                                      resolution_in_degrees))
longitude = tf.feature_column.bucketized_column(longitude_as_a_numeric_column,
                                                longitude_boundaries)
feature_columns.append(longitude)

# Convert the list of feature columns into a layer that will ultimately become
# part of the model. Understanding layers is not important right now.
buckets_feature_layer = layers.DenseFeatures(feature_columns)
```

1.10 Train the model with bucket representations

Run the following code cell to train the model with bucket representations rather than floating-point representations:

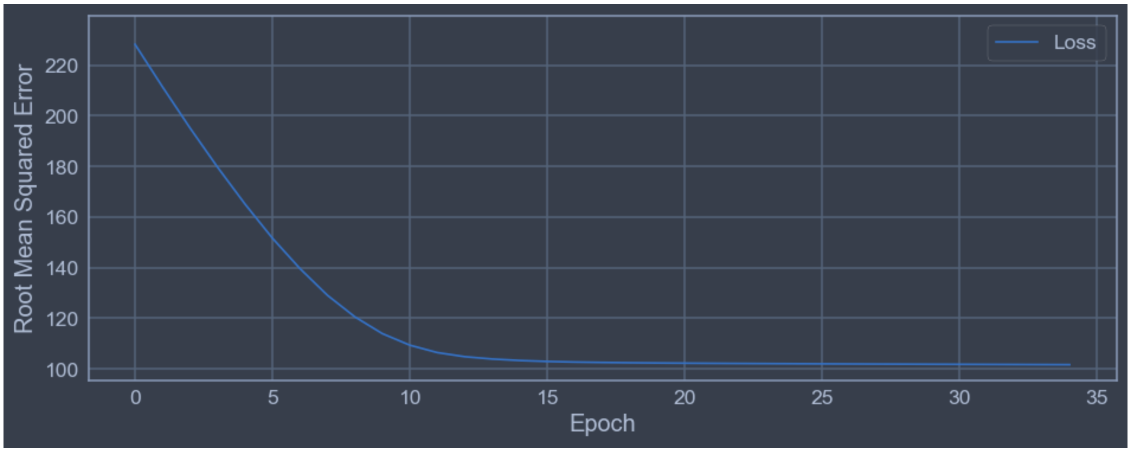
[8]:
```
# The following variables are the hyperparameters.
learning_rate = 0.04
epochs = 35

# Build the model, this time passing in the buckets_feature_layer.
my_model = create_model(learning_rate, buckets_feature_layer)

# Train the model on the training set.
epochs, rmse = train_model(my_model, train_df, epochs, batch_size, label_name)

plot_the_loss_curve(epochs, rmse)

print("\n: Evaluate the new model against the test set:")
my_model.evaluate(x=test_features, y=test_label, batch_size=batch_size)
```

```
Epoch 1/35
WARNING:tensorflow:Layers in a Sequential model should only have a single input
tensor, but we receive a <class 'dict'> input: {'longitude': <tf.Tensor
'ExpandDims_3:0' shape=(100, 1) dtype=float32>, 'latitude': <tf.Tensor
'ExpandDims_2:0' shape=(100, 1) dtype=float32>, 'housing_median_age': <tf.Tensor
'ExpandDims_1:0' shape=(100, 1) dtype=float32>, 'total_rooms': <tf.Tensor
'ExpandDims_7:0' shape=(100, 1) dtype=float32>, 'total_bedrooms': <tf.Tensor
'ExpandDims_6:0' shape=(100, 1) dtype=float32>, 'population': <tf.Tensor
'ExpandDims_5:0' shape=(100, 1) dtype=float32>, 'households': <tf.Tensor
'ExpandDims:0' shape=(100, 1) dtype=float32>, 'median_income': <tf.Tensor
'ExpandDims_4:0' shape=(100, 1) dtype=float32>}
Consider rewriting this model with the Functional API.
WARNING:tensorflow:Layers in a Sequential model should only have a single input
tensor, but we receive a <class 'dict'> input: {'longitude': <tf.Tensor
'ExpandDims_3:0' shape=(100, 1) dtype=float32>, 'latitude': <tf.Tensor
'ExpandDims_2:0' shape=(100, 1) dtype=float32>, 'housing_median_age': <tf.Tensor
'ExpandDims_1:0' shape=(100, 1) dtype=float32>, 'total_rooms': <tf.Tensor
'ExpandDims_7:0' shape=(100, 1) dtype=float32>, 'total_bedrooms': <tf.Tensor
'ExpandDims_6:0' shape=(100, 1) dtype=float32>, 'population': <tf.Tensor
'ExpandDims_5:0' shape=(100, 1) dtype=float32>, 'households': <tf.Tensor
'ExpandDims:0' shape=(100, 1) dtype=float32>, 'median_income': <tf.Tensor
'ExpandDims_4:0' shape=(100, 1) dtype=float32>}
Consider rewriting this model with the Functional API.
170/170 [==============================] - 1s 2ms/step - loss: 53864.0429 -
root_mean_squared_error: 232.0660
Epoch 2/35
170/170 [==============================] - 0s 1ms/step - loss: 46282.1519 -
root_mean_squared_error: 215.1198
```

```
Epoch 3/35
170/170 [==============================] - 0s 1ms/step - loss: 39309.7831 -
root_mean_squared_error: 198.2604
Epoch 4/35
170/170 [==============================] - 0s 1ms/step - loss: 33877.7666 -
root_mean_squared_error: 184.0291
Epoch 5/35
170/170 [==============================] - 0s 1ms/step - loss: 28549.9910 -
root_mean_squared_error: 168.9535
Epoch 6/35
170/170 [==============================] - 0s 2ms/step - loss: 23718.3898 -
root_mean_squared_error: 153.9988
Epoch 7/35
170/170 [==============================] - 0s 1ms/step - loss: 20639.1274 -
root_mean_squared_error: 143.6427
Epoch 8/35
170/170 [==============================] - 0s 2ms/step - loss: 17035.4874 -
root_mean_squared_error: 130.5014
Epoch 9/35
170/170 [==============================] - 0s 1ms/step - loss: 14806.4360 -
root_mean_squared_error: 121.6740
Epoch 10/35
170/170 [==============================] - 0s 2ms/step - loss: 13232.4208 -
root_mean_squared_error: 115.0056
Epoch 11/35
170/170 [==============================] - 0s 2ms/step - loss: 12307.3044 -
root_mean_squared_error: 110.9284
Epoch 12/35
170/170 [==============================] - 0s 2ms/step - loss: 11496.0458 -
root_mean_squared_error: 107.2111
Epoch 13/35
170/170 [==============================] - 0s 2ms/step - loss: 10967.1314 -
root_mean_squared_error: 104.7201
Epoch 14/35
170/170 [==============================] - 0s 1ms/step - loss: 10896.3460 -
root_mean_squared_error: 104.3580
Epoch 15/35
170/170 [==============================] - 0s 1ms/step - loss: 10675.8825 -
root_mean_squared_error: 103.3191
Epoch 16/35
170/170 [==============================] - 0s 2ms/step - loss: 10454.8923 -
root_mean_squared_error: 102.2429
Epoch 17/35
170/170 [==============================] - 0s 1ms/step - loss: 10633.7093 -
root_mean_squared_error: 103.1159
Epoch 18/35
170/170 [==============================] - 0s 1ms/step - loss: 10577.0247 -
root_mean_squared_error: 102.8419
```

```
Epoch 19/35
170/170 [==============================] - 0s 2ms/step - loss: 10478.8993 -
root_mean_squared_error: 102.3543
Epoch 20/35
170/170 [==============================] - 0s 1ms/step - loss: 10594.9817 -
root_mean_squared_error: 102.9278
Epoch 21/35
170/170 [==============================] - 0s 1ms/step - loss: 10650.3214 -
root_mean_squared_error: 103.1961
Epoch 22/35
170/170 [==============================] - 0s 1ms/step - loss: 10623.2918 -
root_mean_squared_error: 103.0575
Epoch 23/35
170/170 [==============================] - 0s 1ms/step - loss: 10443.9517 -
root_mean_squared_error: 102.1854
Epoch 24/35
170/170 [==============================] - 0s 1ms/step - loss: 10314.2343 -
root_mean_squared_error: 101.5552
Epoch 25/35
170/170 [==============================] - 0s 1ms/step - loss: 10348.0475 -
root_mean_squared_error: 101.7225
Epoch 26/35
170/170 [==============================] - 0s 1ms/step - loss: 10183.0090 -
root_mean_squared_error: 100.9049
Epoch 27/35
170/170 [==============================] - 0s 1ms/step - loss: 10454.0507 -
root_mean_squared_error: 102.2394
Epoch 28/35
170/170 [==============================] - 0s 1ms/step - loss: 10255.0272 -
root_mean_squared_error: 101.2531
Epoch 29/35
170/170 [==============================] - 0s 1ms/step - loss: 10355.7453 -
root_mean_squared_error: 101.7452
Epoch 30/35
170/170 [==============================] - 0s 1ms/step - loss: 10585.1057 -
root_mean_squared_error: 102.8796
Epoch 31/35
170/170 [==============================] - 0s 2ms/step - loss: 10429.9680 -
root_mean_squared_error: 102.1255
Epoch 32/35
170/170 [==============================] - 0s 3ms/step - loss: 10358.2837 -
root_mean_squared_error: 101.7723
Epoch 33/35
170/170 [==============================] - 0s 3ms/step - loss: 10294.1062 -
root_mean_squared_error: 101.4425
Epoch 34/35
170/170 [==============================] - 0s 3ms/step - loss: 10359.1232 -
root_mean_squared_error: 101.7778
```

```
Epoch 35/35
170/170 [==============================] - 0s 2ms/step - loss: 10424.9198 -
root_mean_squared_error: 102.0992
```




```
: Evaluate the new model against the test set:
WARNING:tensorflow:Layers in a Sequential model should only have a single input
tensor, but we receive a <class 'dict'> input: {'longitude': <tf.Tensor
'ExpandDims_3:0' shape=(100, 1) dtype=float32>, 'latitude': <tf.Tensor
'ExpandDims_2:0' shape=(100, 1) dtype=float32>, 'housing_median_age': <tf.Tensor
'ExpandDims_1:0' shape=(100, 1) dtype=float32>, 'total_rooms': <tf.Tensor
'ExpandDims_7:0' shape=(100, 1) dtype=float32>, 'total_bedrooms': <tf.Tensor
'ExpandDims_6:0' shape=(100, 1) dtype=float32>, 'population': <tf.Tensor
'ExpandDims_5:0' shape=(100, 1) dtype=float32>, 'households': <tf.Tensor
'ExpandDims:0' shape=(100, 1) dtype=float32>, 'median_income': <tf.Tensor
'ExpandDims_4:0' shape=(100, 1) dtype=float32>}
Consider rewriting this model with the Functional API.
30/30 [==============================] - 0s 2ms/step - loss: 10198.6270 -
root_mean_squared_error: 100.9883
```

[8]:
```
[10198.626953125, 100.98825073242188]
```

1.11 Task 2: Did buckets outperform floating-point representations?

Compare the model's `root_mean_squared_error` values for the two representations (floating-point vs. buckets)? Which model produced lower losses?

[ ]:
```
#@title Double-click for an answer to Task 2.

# Bucket representation outperformed
# floating-point representations.
# However, you can still do far better.
```

1.12 Task 3: What is a better way to represent location?

Buckets are a big improvement over floating-point values. Can you identify an even better way to identify location with `latitude` and `longitude`?

[ ]:
```
#@title Double-click to view an answer to Task 3.

# Representing location as a feature cross should
# produce better results.

# In Task 2, you represented latitude in
# one-dimensional buckets and longitude in
# another series of one-dimensional buckets.
# Real-world locations, however, exist in
# two dimension. Therefore, you should
# represent location as a two-dimensional feature
# cross. That is, you'll cross the 10 or so latitude
# buckets with the 10 or so longitude buckets to
# create a grid of 100 cells.

# The model will learn separate weights for each
# of the cells.
```

1.13 Represent location as a feature cross

The following code cell represents location as a feature cross. That is, the following code cell first creates buckets and then calls `tf.feature_column.crossed_column` to cross the buckets.

[17]:
```
resolution_in_degrees = 0.4

# Create a new empty list that will eventually hold the generated feature␣
↪column.
feature_columns = []

# Create a bucket feature column for latitude.
latitude_as_a_numeric_column = tf.feature_column.numeric_column("latitude")
latitude_boundaries = list(np.arange(int(min(train_df['latitude'])),␣
↪int(max(train_df['latitude'])), resolution_in_degrees))
latitude = tf.feature_column.bucketized_column(latitude_as_a_numeric_column,␣
↪latitude_boundaries)

# Create a bucket feature column for longitude.
longitude_as_a_numeric_column = tf.feature_column.numeric_column("longitude")
longitude_boundaries = list(np.arange(int(min(train_df['longitude'])),␣
↪int(max(train_df['longitude'])), resolution_in_degrees))
longitude = tf.feature_column.bucketized_column(longitude_as_a_numeric_column,␣
↪longitude_boundaries)
```

```
# Create a feature cross of latitude and longitude.
latitude_x_longitude = tf.feature_column.crossed_column([latitude, longitude],␣
 ↪hash_bucket_size=100)
crossed_feature = tf.feature_column.indicator_column(latitude_x_longitude)
feature_columns.append(crossed_feature)

# Convert the list of feature columns into a layer that will later be fed into
# the model.
feature_cross_feature_layer = layers.DenseFeatures(feature_columns)
```

Invoke the following code cell to test your solution for Task 2. Please ignore the warning messages.

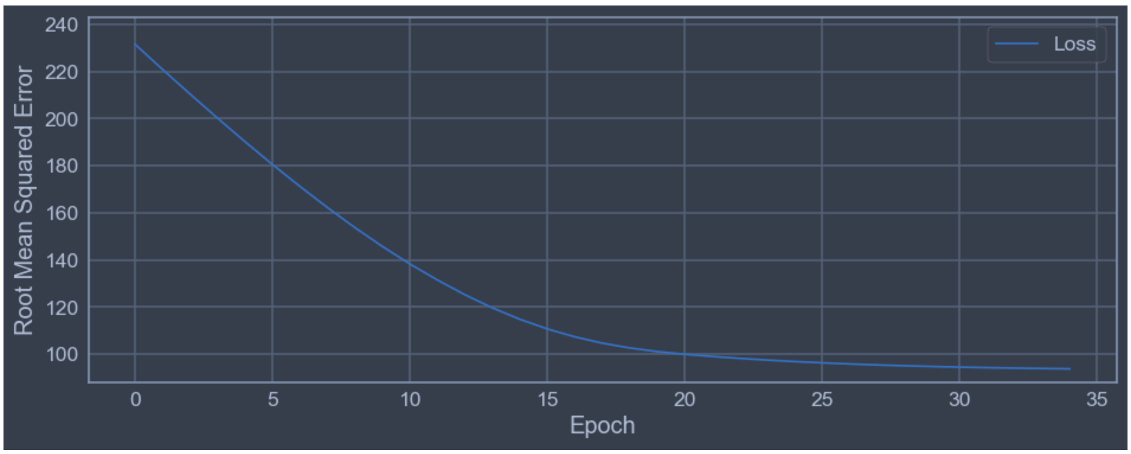
[18]:
```
# The following variables are the hyperparameters.
learning_rate = 0.04
epochs = 35

# Build the model, this time passing in the feature_cross_feature_layer:
my_model = create_model(learning_rate, feature_cross_feature_layer)

# Train the model on the training set.
epochs, rmse = train_model(my_model, train_df, epochs, batch_size, label_name)

plot_the_loss_curve(epochs, rmse)

print("\n: Evaluate the new model against the test set:")
my_model.evaluate(x=test_features, y=test_label, batch_size=batch_size)
```

```
Epoch 1/35
WARNING:tensorflow:Layers in a Sequential model should only have a single input
tensor, but we receive a <class 'dict'> input: {'longitude': <tf.Tensor
'ExpandDims_3:0' shape=(100, 1) dtype=float32>, 'latitude': <tf.Tensor
'ExpandDims_2:0' shape=(100, 1) dtype=float32>, 'housing_median_age': <tf.Tensor
'ExpandDims_1:0' shape=(100, 1) dtype=float32>, 'total_rooms': <tf.Tensor
'ExpandDims_7:0' shape=(100, 1) dtype=float32>, 'total_bedrooms': <tf.Tensor
'ExpandDims_6:0' shape=(100, 1) dtype=float32>, 'population': <tf.Tensor
'ExpandDims_5:0' shape=(100, 1) dtype=float32>, 'households': <tf.Tensor
'ExpandDims:0' shape=(100, 1) dtype=float32>, 'median_income': <tf.Tensor
'ExpandDims_4:0' shape=(100, 1) dtype=float32>}
Consider rewriting this model with the Functional API.
WARNING:tensorflow:Layers in a Sequential model should only have a single input
tensor, but we receive a <class 'dict'> input: {'longitude': <tf.Tensor
'ExpandDims_3:0' shape=(100, 1) dtype=float32>, 'latitude': <tf.Tensor
'ExpandDims_2:0' shape=(100, 1) dtype=float32>, 'housing_median_age': <tf.Tensor
'ExpandDims_1:0' shape=(100, 1) dtype=float32>, 'total_rooms': <tf.Tensor
'ExpandDims_7:0' shape=(100, 1) dtype=float32>, 'total_bedrooms': <tf.Tensor
'ExpandDims_6:0' shape=(100, 1) dtype=float32>, 'population': <tf.Tensor
'ExpandDims_5:0' shape=(100, 1) dtype=float32>, 'households': <tf.Tensor
'ExpandDims:0' shape=(100, 1) dtype=float32>, 'median_income': <tf.Tensor
```

```
'ExpandDims_4:0' shape=(100, 1) dtype=float32>}
Consider rewriting this model with the Functional API.
170/170 [==============================] - 1s 2ms/step - loss: 54619.3328 -
root_mean_squared_error: 233.6996
Epoch 2/35
170/170 [==============================] - 0s 2ms/step - loss: 50226.7875 -
root_mean_squared_error: 224.1012
Epoch 3/35
170/170 [==============================] - 0s 2ms/step - loss: 45037.2616 -
root_mean_squared_error: 212.2039
Epoch 4/35
170/170 [==============================] - 0s 2ms/step - loss: 42016.5040 -
root_mean_squared_error: 204.9477
Epoch 5/35
170/170 [==============================] - 0s 2ms/step - loss: 37625.7285 -
root_mean_squared_error: 193.9441
Epoch 6/35
170/170 [==============================] - 0s 2ms/step - loss: 33634.8849 -
root_mean_squared_error: 183.3722
Epoch 7/35
170/170 [==============================] - 0s 2ms/step - loss: 30226.6555 -
root_mean_squared_error: 173.8476
Epoch 8/35
170/170 [==============================] - 0s 3ms/step - loss: 26821.2219 -
root_mean_squared_error: 163.7662
Epoch 9/35
170/170 [==============================] - 1s 3ms/step - loss: 24001.8744 -
root_mean_squared_error: 154.9179
Epoch 10/35
170/170 [==============================] - 1s 4ms/step - loss: 21735.1485 -
root_mean_squared_error: 147.4095
Epoch 11/35
170/170 [==============================] - 1s 4ms/step - loss: 19754.9901 -
root_mean_squared_error: 140.5310
Epoch 12/35
170/170 [==============================] - 1s 4ms/step - loss: 18155.7249 -
root_mean_squared_error: 134.7181
Epoch 13/35
170/170 [==============================] - 1s 3ms/step - loss: 16080.9936 -
root_mean_squared_error: 126.7975
Epoch 14/35
170/170 [==============================] - 1s 4ms/step - loss: 14836.0994 -
root_mean_squared_error: 121.7662
Epoch 15/35
170/170 [==============================] - 1s 3ms/step - loss: 13467.5284 -
root_mean_squared_error: 116.0459
Epoch 16/35
170/170 [==============================] - 1s 3ms/step - loss: 12560.5400 -
```

```
root_mean_squared_error: 112.0666
Epoch 17/35
170/170 [==============================] - 1s 3ms/step - loss: 11591.2872 -
root_mean_squared_error: 107.6579
Epoch 18/35
170/170 [==============================] - 1s 3ms/step - loss: 10974.3131 -
root_mean_squared_error: 104.7483
Epoch 19/35
170/170 [==============================] - 1s 3ms/step - loss: 10384.7929 -
root_mean_squared_error: 101.8976
Epoch 20/35
170/170 [==============================] - 1s 3ms/step - loss: 10099.9753 -
root_mean_squared_error: 100.4937
Epoch 21/35
170/170 [==============================] - 1s 3ms/step - loss: 10086.0510 -
root_mean_squared_error: 100.4264
Epoch 22/35
170/170 [==============================] - 1s 3ms/step - loss: 9563.2488 -
root_mean_squared_error: 97.7832
Epoch 23/35
170/170 [==============================] - 1s 3ms/step - loss: 9769.0690 -
root_mean_squared_error: 98.8362
Epoch 24/35
170/170 [==============================] - 1s 3ms/step - loss: 9549.9982 -
root_mean_squared_error: 97.7176
Epoch 25/35
170/170 [==============================] - 1s 3ms/step - loss: 9178.3903 -
root_mean_squared_error: 95.7912
Epoch 26/35
170/170 [==============================] - 1s 3ms/step - loss: 9493.9538 -
root_mean_squared_error: 97.4233
Epoch 27/35
170/170 [==============================] - 1s 3ms/step - loss: 9254.3647 -
root_mean_squared_error: 96.1959
Epoch 28/35
170/170 [==============================] - 1s 3ms/step - loss: 9069.2396 -
root_mean_squared_error: 95.2273
Epoch 29/35
170/170 [==============================] - 1s 3ms/step - loss: 9031.9349 -
root_mean_squared_error: 95.0347
Epoch 30/35
170/170 [==============================] - 1s 3ms/step - loss: 8775.9093 -
root_mean_squared_error: 93.6712
Epoch 31/35
170/170 [==============================] - 1s 3ms/step - loss: 8892.9110 -
root_mean_squared_error: 94.2939
Epoch 32/35
170/170 [==============================] - 1s 3ms/step - loss: 8922.0123 -
```

```
root_mean_squared_error: 94.4510
Epoch 33/35
170/170 [==============================] - 1s 3ms/step - loss: 8809.8588 -
root_mean_squared_error: 93.8578
Epoch 34/35
170/170 [==============================] - 1s 3ms/step - loss: 8873.4916 -
root_mean_squared_error: 94.1924
Epoch 35/35
170/170 [==============================] - 1s 3ms/step - loss: 8679.1867 -
root_mean_squared_error: 93.1578
```




```
: Evaluate the new model against the test set:
WARNING:tensorflow:Layers in a Sequential model should only have a single input
tensor, but we receive a <class 'dict'> input: {'longitude': <tf.Tensor
'ExpandDims_3:0' shape=(100, 1) dtype=float32>, 'latitude': <tf.Tensor
'ExpandDims_2:0' shape=(100, 1) dtype=float32>, 'housing_median_age': <tf.Tensor
'ExpandDims_1:0' shape=(100, 1) dtype=float32>, 'total_rooms': <tf.Tensor
'ExpandDims_7:0' shape=(100, 1) dtype=float32>, 'total_bedrooms': <tf.Tensor
'ExpandDims_6:0' shape=(100, 1) dtype=float32>, 'population': <tf.Tensor
'ExpandDims_5:0' shape=(100, 1) dtype=float32>, 'households': <tf.Tensor
'ExpandDims:0' shape=(100, 1) dtype=float32>, 'median_income': <tf.Tensor
'ExpandDims_4:0' shape=(100, 1) dtype=float32>}
Consider rewriting this model with the Functional API.
30/30 [==============================] - 0s 3ms/step - loss: 8645.8350 -
root_mean_squared_error: 92.9830
```

```
[18]: [8645.8349609375, 92.98297882080078]
```

1.14 Task 4: Did the feature cross outperform buckets?

Compare the model's `root_mean_squared_error` values for the two representations (buckets vs. feature cross)? Which model produced lower losses?

[ ]:
```
#@title Double-click for an answer to this question.

# Yes, representing these features as a feature
# cross produced much lower loss values than
# representing these features as buckets
```

1.15 Task 5: Adjust the resolution of the feature cross

Return to the code cell in the "Represent location as a feature cross" section. Notice that `resolution_in_degrees` is set to 1.0. Therefore, each cell represents an area of 1.0 degree of latitude by 1.0 degree of longitude, which corresponds to a cell of 110 km by 90 km. This resolution defines a rather large neighborhood.

Experiment with `resolution_in_degrees` to answer the following questions:

1. What value of `resolution_in_degrees` produces the best results (lowest loss value)?
2. Why does loss increase when the value of `resolution_in_degrees` drops below a certain value?

Finally, answer the following question:

3. What feature (that does not exist in the California Housing Dataset) would be a better proxy for location than latitude X longitude.

[ ]:
```
#@title Double-click for possible answers to Task 5.

#1. A resolution of ~0.4 degree provides the best
#   results.

#2. Below ~0.4 degree, loss increases because the
#   dataset does not contain enough examples in
#   each cell to accurately predict prices for
#   those cells.

#3. Postal code would be a far better feature
#   than latitude X longitude, assuming that
#   the dataset contained sufficient examples
#   in each postal code.
```