CPE 313 : Advanced Machine Learning with Deep Learning

HOA 1.1 Using Tensorflow with a Real Dataset

- Name: Navida, Kryslyhr L.

Linear Regression with a Real Dataset

This Colab uses a real dataset to predict the prices of houses in California.

Learning Objectives:

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

- Demonstrate csv file manipulation using Pandas.
- Examine a given data.
- Experiment with different features in building a model.
- Demonstrate tuning a model's hyperparameters.

The Dataset

The dataset for this exercise is based on 1990 census data from California. The dataset is old but still provides a great opportunity to learn about machine learning programming.

Import relevant modules

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

In [1]:
```
#@title Import relevant modules
import pandas as pd
import tensorflow as tf
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
```

The dataset

Datasets are often stored on disk or at a URL in .csv format.

A well-formed .csv file contains column names in the first row, followed by many rows of data. A comma divides each value in each row. For example, here are the first five rows of the .csv file holding the California Housing Dataset:

```
"longitude","latitude","housing_median_age","total_rooms","total_bedrooms'
-114.310000,34.190000,15.000000,5612.000000,1283.000000,1015.000000,472.0(
-114.470000,34.400000,19.000000,7650.000000,1901.000000,1129.000000,463.0(
-114.560000,33.690000,17.000000,720.000000,174.000000,333.000000,117.0000(
-114.570000,33.640000,14.000000,1501.000000,337.000000,515.000000,226.0000
```

Load the .csv file into a pandas DataFrame

This Colab, like many machine learning programs, gathers the .csv file and stores the data in memory as a pandas Dataframe. Pandas is an open source Python library. The primary datatype in pandas is a DataFrame. You can imagine a pandas DataFrame as a spreadsheet in which each row is identified by a number and each column by a name. Pandas is itself built on another open source Python library called NumPy. If you aren't familiar with these technologies, please view these two quick tutorials:

- NumPy
- Pandas DataFrames

The following code cell imports the .csv file into a pandas DataFrame and scales the values in the label (`median_house_value`):

In [2]:
```
# Import the dataset.
training_df = pd.read_csv(filepath_or_buffer="https://download.mlcc.google.com/mled

# Scale the label.
training_df["median_house_value"] /= 1000.0

# Print the first rows of the pandas DataFrame.
training_df.head()
```

Out[2]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	hou
0	-114.3	34.2	15.0	5612.0	1283.0	1015.0	
1	-114.5	34.4	19.0	7650.0	1901.0	1129.0	
2	-114.6	33.7	17.0	720.0	174.0	333.0	
3	-114.6	33.6	14.0	1501.0	337.0	515.0	
4	-114.6	33.6	20.0	1454.0	326.0	624.0	

Scaling `median_house_value` puts the value of each house in units of thousands. Scaling will keep loss values and learning rates in a friendlier range.

Although scaling a label is usually *not* essential, scaling features in a multi-feature model usually *is* essential.

Examine the dataset

A large part of most machine learning projects is getting to know your data. The pandas API provides a `describe` function that outputs the following statistics about every column in the DataFrame:

- `count`, which is the number of rows in that column. Ideally, `count` contains the same value for every column.
- `mean` and `std`, which contain the mean and standard deviation of the values in each column.
- `min` and `max`, which contain the lowest and highest values in each column.
- `25%`, `50%`, `75%`, which contain various quantiles.

In [3]:

```
# Get statistics on the dataset.
training_df.describe()
```

Out[3]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population
count	17000.0	17000.0	17000.0	17000.0	17000.0	17000.0
mean	-119.6	35.6	28.6	2643.7	539.4	1429.6
std	2.0	2.1	12.6	2179.9	421.5	1147.9
min	-124.3	32.5	1.0	2.0	1.0	3.0
25%	-121.8	33.9	18.0	1462.0	297.0	790.0
50%	-118.5	34.2	29.0	2127.0	434.0	1167.0
75%	-118.0	37.7	37.0	3151.2	648.2	1721.0
max	-114.3	42.0	52.0	37937.0	6445.0	35682.0

Task 1: Identify anomalies in the dataset

Do you see any anomalies (strange values) in the data?

When compared to the other quantiles, the maximum value (max) of a few columns appears to be extremely high. Take the total_rooms column, for instance. The quantile values of 25%, 50%, and 75% suggest that the maximum value of total_rooms should be in the range of 5,000 to 10,000. In actuality, though, the maximum value is 37,937. Use caution while utilizing a column as a feature when you notice irregularities in it. Having stated that, anomalies in prospective features can occasionally reflect abnormalities in the label, which could give the impression that the column is a strong feature. Additionally, you may be able to represent (pre-process) raw data in order to turn columns into helpful characteristics, as you will learn later in the course.

In [4]:

```
#@title Double-click to view a possible answer.

# The maximum value (max) of several columns seems very
# high compared to the other quantiles. For example,
# example the total_rooms column. Given the quantile
# values (25%, 50%, and 75%), you might expect the
# max value of total_rooms to be approximately
# 5,000 or possibly 10,000. However, the max value
# is actually 37,937.

# When you see anomalies in a column, become more careful
# about using that column as a feature. That said,
# anomalies in potential features sometimes mirror
# anomalies in the label, which could make the column
# be (or seem to be) a powerful feature.
# Also, as you will see later in the course, you
# might be able to represent (pre-process) raw data
# in order to make columns into useful features.
```

Define functions that build and train a model

The following code defines two functions:

- `build_model(my_learning_rate)`, which builds a randomly-initialized model.
- `train_model(model, feature, label, epochs)`, which trains the model from the examples (feature and label) you pass.

Since you don't need to understand model building code right now, we've hidden this code cell. You may optionally double-click the following headline to see the code that builds and trains a model.

In [5]:
```
def build_model(my_learning_rate):
  """Create and compile a simple linear regression model."""

  model = tf.keras.models.Sequential()

  model.add(tf.keras.layers.Dense(units=1,
                                  input_shape=(1,)))

  model.compile(optimizer=tf.keras.optimizers.experimental.RMSprop(learning_rate=my
                loss="mean_squared_error",
                metrics=[tf.keras.metrics.RootMeanSquaredError()])

  return model


def train_model(model, df, feature, label, epochs, batch_size):
  """Train the model by feeding it data."""

  history = model.fit(x=df[feature],
                      y=df[label],
                      batch_size=batch_size,
                      epochs=epochs)

  trained_weight = model.get_weights()[0]
  trained_bias = model.get_weights()[1]

  epochs = history.epoch

  hist = pd.DataFrame(history.history)

  rmse = hist["root_mean_squared_error"]

  return trained_weight, trained_bias, epochs, rmse

print("Defined the build_model and train_model functions.")
```

```
Defined the build_model and train_model functions.
```

Define plotting functions

The following matplotlib functions create the following plots:

- a scatter plot of the feature vs. the label, and a line showing the output of the trained model
- a loss curve

You may optionally double-click the headline to see the matplotlib code, but note that writing matplotlib code is not an important part of learning ML programming.

In [6]:
```
def plot_the_model(trained_weight, trained_bias, feature, label):
  """Plot the trained model against 200 random training examples."""

  plt.xlabel(feature)
  plt.ylabel(label)

  random_examples = training_df.sample(n=200)
  plt.scatter(random_examples[feature], random_examples[label])

  x0 = 0
  y0 = trained_bias
  x1 = random_examples[feature].max()
  y1 = trained_bias + (trained_weight * x1)
  plt.plot([x0, x1], [y0, y1], c='r')

  plt.show()


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()*0.97, rmse.max()])
  plt.show()

print("Defined the plot_the_model and plot_the_loss_curve functions.")
```

```
Defined the plot_the_model and plot_the_loss_curve functions.
```

Call the model functions

An important part of machine learning is determining which features correlate with the label. For example, real-life home-value prediction models typically rely on hundreds of features and synthetic features. However, this model relies on only one feature. For now, you'll arbitrarily use `total_rooms` as that feature.

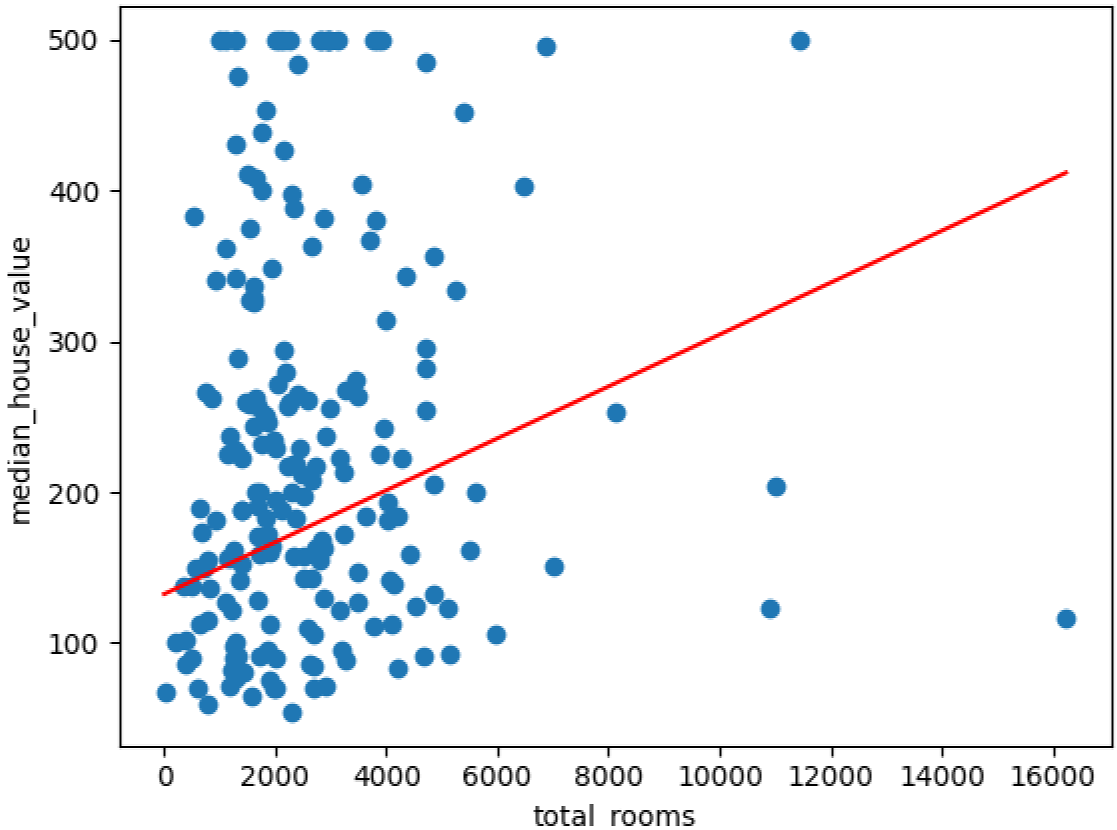
In [16]:
```
learning_rate = 0.01
epochs = 30
```

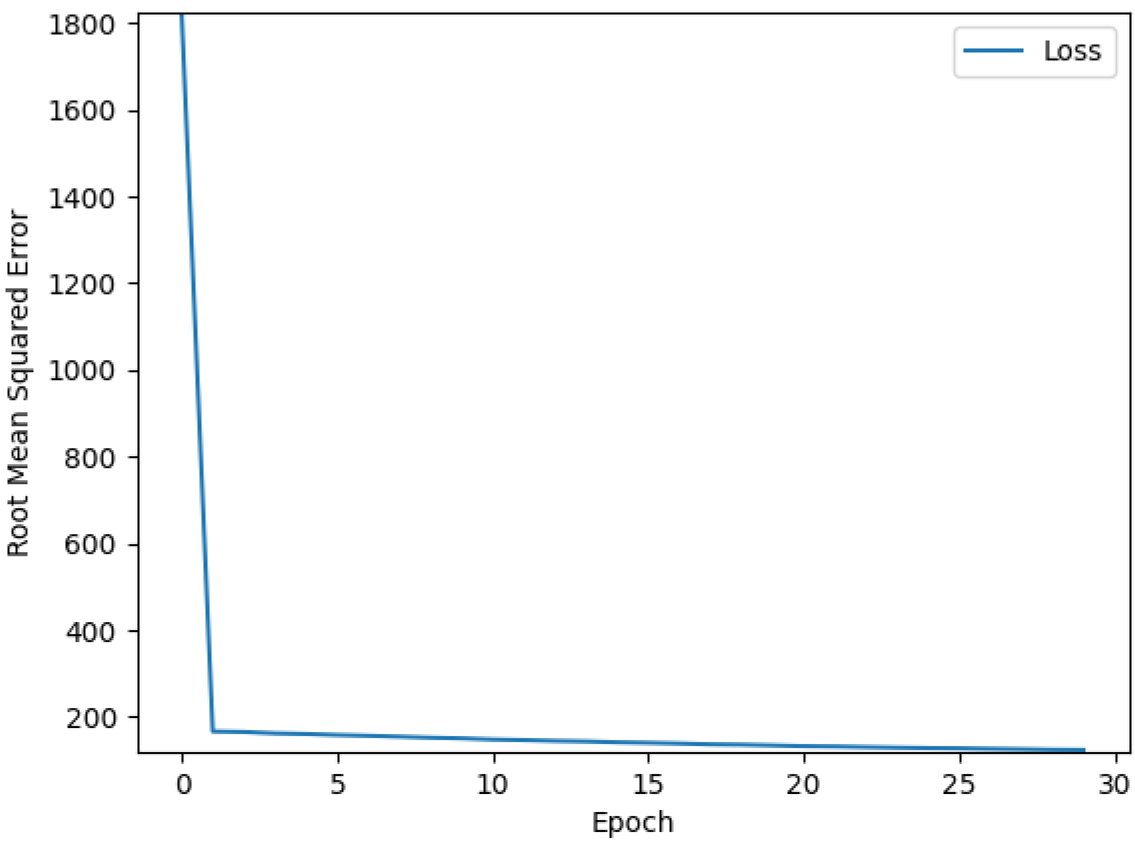
```
batch_size = 30

my_feature = "total_rooms"
my_label="median_house_value"

my_model = None

my_model = build_model(learning_rate)
weight, bias, epochs, rmse = train_model(my_model, training_df,
                                         my_feature, my_label,
                                         epochs, batch_size)

print("\nThe learned weight for your model is %.4f" % weight)
print("The learned bias for your model is %.4f\n" % bias )

plot_the_model(weight, bias, my_feature, my_label)
plot_the_loss_curve(epochs, rmse)
```

```
Epoch 1/30
567/567 [==============================] - 2s 2ms/step - loss: 3327965.7500 - root_m
ean_squared_error: 1824.2712
Epoch 2/30
567/567 [==============================] - 1s 2ms/step - loss: 27577.0234 - root_mea
n_squared_error: 166.0633
Epoch 3/30
567/567 [==============================] - 1s 1ms/step - loss: 27186.8984 - root_mea
n_squared_error: 164.8845
Epoch 4/30
567/567 [==============================] - 1s 1ms/step - loss: 26125.9180 - root_mea
n_squared_error: 161.6351
Epoch 5/30
567/567 [==============================] - 1s 1ms/step - loss: 25644.5098 - root_mea
n_squared_error: 160.1390
Epoch 6/30
567/567 [==============================] - 1s 1ms/step - loss: 24907.0938 - root_mea
n_squared_error: 157.8198
Epoch 7/30
567/567 [==============================] - 1s 1ms/step - loss: 24349.4219 - root_mea
n_squared_error: 156.0430
Epoch 8/30
567/567 [==============================] - 1s 1ms/step - loss: 23705.4980 - root_mea
n_squared_error: 153.9659
Epoch 9/30
567/567 [==============================] - 1s 1ms/step - loss: 23001.5293 - root_mea
n_squared_error: 151.6626
Epoch 10/30
567/567 [==============================] - 1s 1ms/step - loss: 22521.5039 - root_mea
n_squared_error: 150.0717
Epoch 11/30
567/567 [==============================] - 1s 1ms/step - loss: 21727.1172 - root_mea
n_squared_error: 147.4012
Epoch 12/30
567/567 [==============================] - 1s 1ms/step - loss: 21339.2539 - root_mea
n_squared_error: 146.0796
Epoch 13/30
567/567 [==============================] - 1s 1ms/step - loss: 20810.2754 - root_mea
n_squared_error: 144.2577
Epoch 14/30
567/567 [==============================] - 1s 2ms/step - loss: 20455.5000 - root_mea
n_squared_error: 143.0227
Epoch 15/30
567/567 [==============================] - 1s 2ms/step - loss: 19877.7656 - root_mea
n_squared_error: 140.9885
Epoch 16/30
567/567 [==============================] - 1s 2ms/step - loss: 19534.8320 - root_mea
n_squared_error: 139.7671
Epoch 17/30
567/567 [==============================] - 1s 2ms/step - loss: 19099.3750 - root_mea
n_squared_error: 138.2005
Epoch 18/30
567/567 [==============================] - 1s 1ms/step - loss: 18555.0371 - root_mea
n_squared_error: 136.2169
Epoch 19/30
567/567 [==============================] - 1s 1ms/step - loss: 18353.2227 - root_mea
```

```
n_squared_error: 135.4741
Epoch 20/30
567/567 [==============================] - 1s 1ms/step - loss: 17906.9160 - root_mea
n_squared_error: 133.8167
Epoch 21/30
567/567 [==============================] - 1s 1ms/step - loss: 17424.7578 - root_mea
n_squared_error: 132.0029
Epoch 22/30
567/567 [==============================] - 1s 1ms/step - loss: 17247.7207 - root_mea
n_squared_error: 131.3306
Epoch 23/30
567/567 [==============================] - 1s 1ms/step - loss: 16878.2422 - root_mea
n_squared_error: 129.9163
Epoch 24/30
567/567 [==============================] - 1s 1ms/step - loss: 16572.7656 - root_mea
n_squared_error: 128.7353
Epoch 25/30
567/567 [==============================] - 1s 1ms/step - loss: 16317.8896 - root_mea
n_squared_error: 127.7415
Epoch 26/30
567/567 [==============================] - 1s 1ms/step - loss: 16161.2480 - root_mea
n_squared_error: 127.1269
Epoch 27/30
567/567 [==============================] - 1s 1ms/step - loss: 15869.8936 - root_mea
n_squared_error: 125.9758
Epoch 28/30
567/567 [==============================] - 1s 1ms/step - loss: 15648.1504 - root_mea
n_squared_error: 125.0926
Epoch 29/30
567/567 [==============================] - 1s 2ms/step - loss: 15356.6748 - root_mea
n_squared_error: 123.9221
Epoch 30/30
567/567 [==============================] - 1s 2ms/step - loss: 15241.6230 - root_mea
n_squared_error: 123.4570

The learned weight for your model is 0.0172
The learned bias for your model is 132.3119
```

```
/usr/local/lib/python3.10/dist-packages/numpy/core/shape_base.py:65: VisibleDeprecat
ionWarning: Creating an ndarray from ragged nested sequences (which is a list-or-tup
le of lists-or-tuples-or ndarrays with different lengths or shapes) is deprecated. I
f you meant to do this, you must specify 'dtype=object' when creating the ndarray.
  ary = asanyarray(ary)
```





A certain amount of randomness plays into training a model. Consequently, you'll get different results each time you train the model. That said, given the dataset and the hyperparameters, the trained model will generally do a poor job describing the feature's relation to the label.

Use the model to make predictions

You can use the trained model to make predictions. In practice, you should make predictions on examples that are not used in training. However, for this exercise, you'll just work with a subset of the same training dataset. A later Colab exercise will explore ways to make predictions on examples not used in training.

First, run the following code to define the house prediction function:

In [17]:
```
def predict_house_values(n, feature, label):
  """Predict house values based on a feature."""

  batch = training_df[feature][10000:10000 + n]
  predicted_values = my_model.predict_on_batch(x=batch)

  print("feature   label          predicted")
  print("  value   value          value")
  print("          in thousand$   in thousand$")
  print("--------------------------------------")
  for i in range(n):
    print ("%5.0f %6.0f %15.0f" % (training_df[feature][10000 + i],
                                   training_df[label][10000 + i],
                                   predicted_values[i][0] ))
```

Now, invoke the house prediction function on 10 examples:

In [18]:
```
predict_house_values(10, my_feature, my_label)
```

```
feature   label          predicted
  value   value          value
          in thousand$   in thousand$
--------------------------------------
 1960     53             166
 3400     92             191
 3677     69             196
 2202     62             170
 2403     80             174
 5652    295             230
 3318    500             189
 2552    342             176
 1364    118             156
 3468    128             192
```

Task 2: Judge the predictive power of the model

Look at the preceding table. How close is the predicted value to the label value? In other words, does your model accurately predict house values?

In [19]:
```
#It is likely that the trained model has little predictive power because the majori
#significantly different from the label value.
#Nevertheless, it's possible that the first ten cases don't accurately reflect the
```

Task 3: Try a different feature

The `total_rooms` feature had only a little predictive power. Would a different feature have greater predictive power? Try using `population` as the feature instead of `total_rooms`.

Note: When you change features, you might also need to change the hyperparameters.

In [20]:
```
my_feature = "?"   # Replace the ? with population or possibly
                   # a different column name.

# Experiment with the hyperparameters.
learning_rate = 2
epochs = 3
batch_size = 120

# Don't change anything below this line.
my_model = build_model(learning_rate)
weight, bias, epochs, rmse = train_model(my_model, training_df,
                                         my_feature, my_label,
                                         epochs, batch_size)
plot_the_model(weight, bias, my_feature, my_label)
plot_the_loss_curve(epochs, rmse)

predict_house_values(15, my_feature, my_label)
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
/usr/local/lib/python3.10/dist-packages/pandas/core/indexes/base.py in get_loc(self, key, method, tolerance)
   3801             try:
-> 3802                 return self._engine.get_loc(casted_key)
   3803             except KeyError as err:

/usr/local/lib/python3.10/dist-packages/pandas/_libs/index.pyx in pandas._libs.index.IndexEngine.get_loc()

/usr/local/lib/python3.10/dist-packages/pandas/_libs/index.pyx in pandas._libs.index.IndexEngine.get_loc()

pandas/_libs/hashtable_class_helper.pxi in pandas._libs.hashtable.PyObjectHashTable.get_item()

pandas/_libs/hashtable_class_helper.pxi in pandas._libs.hashtable.PyObjectHashTable.get_item()

KeyError: '?'

The above exception was the direct cause of the following exception:

KeyError                                  Traceback (most recent call last)
<ipython-input-20-9c1854baaa30> in <cell line: 11>()
      9 # Don't change anything below this line.
     10 my_model = build_model(learning_rate)
---> 11 weight, bias, epochs, rmse = train_model(my_model, training_df,
     12                                          my_feature, my_label,
     13                                          epochs, batch_size)

<ipython-input-5-b6257f094005> in train_model(model, df, feature, label, epochs, batch_size)
     17   """Train the model by feeding it data."""
     18 
---> 19   history = model.fit(x=df[feature],
     20                       y=df[label],
     21                       batch_size=batch_size,

/usr/local/lib/python3.10/dist-packages/pandas/core/frame.py in __getitem__(self, key)
   3805             if self.columns.nlevels > 1:
   3806                 return self._getitem_multilevel(key)
-> 3807             indexer = self.columns.get_loc(key)
   3808             if is_integer(indexer):
   3809                 indexer = [indexer]

/usr/local/lib/python3.10/dist-packages/pandas/core/indexes/base.py in get_loc(self, key, method, tolerance)
   3802                 return self._engine.get_loc(casted_key)
   3803             except KeyError as err:
-> 3804                 raise KeyError(key) from err
   3805             except TypeError:
   3806                 # If we have a listlike key, _check_indexing_error will raise
```

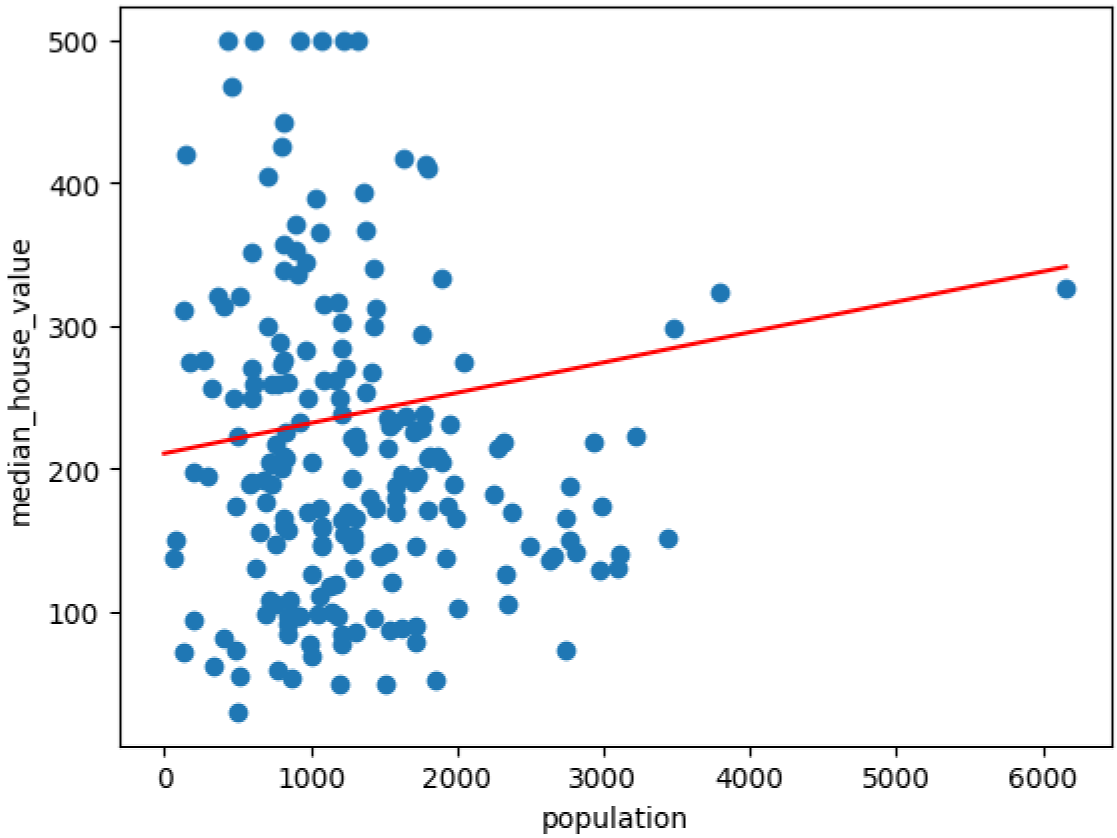
```
KeyError: '?'
```

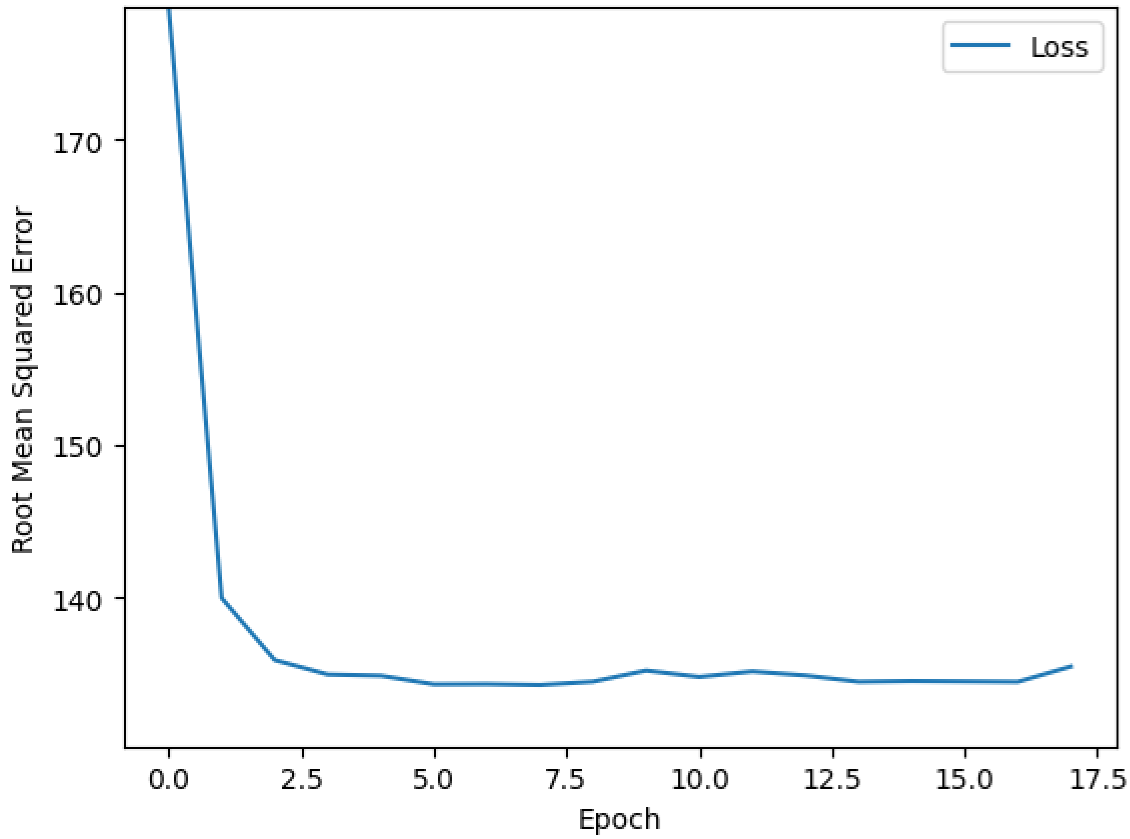
In [15]:

```
my_feature = "population"

learning_rate = 0.05
epochs = 18
batch_size = 3

my_model = build_model(learning_rate)
weight, bias, epochs, rmse = train_model(my_model, training_df,
                                          my_feature, my_label,
                                          epochs, batch_size)

plot_the_model(weight, bias, my_feature, my_label)
plot_the_loss_curve(epochs, rmse)

predict_house_values(10, my_feature, my_label)
```

```
Epoch 1/18
5667/5667 [==============================] - 10s 2ms/step - loss: 31934.4082 - root_
mean_squared_error: 178.7020
Epoch 2/18
5667/5667 [==============================] - 9s 2ms/step - loss: 19593.4434 - root_m
ean_squared_error: 139.9766
Epoch 3/18
5667/5667 [==============================] - 8s 1ms/step - loss: 18471.0781 - root_m
ean_squared_error: 135.9083
Epoch 4/18
5667/5667 [==============================] - 9s 2ms/step - loss: 18215.2734 - root_m
ean_squared_error: 134.9640
Epoch 5/18
5667/5667 [==============================] - 9s 2ms/step - loss: 18193.0625 - root_m
ean_squared_error: 134.8817
Epoch 6/18
5667/5667 [==============================] - 12s 2ms/step - loss: 18042.0117 - root_
mean_squared_error: 134.3206
Epoch 7/18
5667/5667 [==============================] - 10s 2ms/step - loss: 18045.3770 - root_
mean_squared_error: 134.3331
Epoch 8/18
5667/5667 [==============================] - 10s 2ms/step - loss: 18029.5293 - root_
mean_squared_error: 134.2741
Epoch 9/18
5667/5667 [==============================] - 12s 2ms/step - loss: 18088.7266 - root_
mean_squared_error: 134.4943
Epoch 10/18
5667/5667 [==============================] - 9s 2ms/step - loss: 18284.8965 - root_m
ean_squared_error: 135.2217
Epoch 11/18
5667/5667 [==============================] - 9s 2ms/step - loss: 18171.6562 - root_m
ean_squared_error: 134.8023
Epoch 12/18
5667/5667 [==============================] - 9s 2ms/step - loss: 18269.4219 - root_m
ean_squared_error: 135.1644
Epoch 13/18
5667/5667 [==============================] - 8s 1ms/step - loss: 18197.2148 - root_m
ean_squared_error: 134.8970
Epoch 14/18
5667/5667 [==============================] - 9s 2ms/step - loss: 18086.4551 - root_m
ean_squared_error: 134.4859
Epoch 15/18
5667/5667 [==============================] - 9s 2ms/step - loss: 18098.6582 - root_m
ean_squared_error: 134.5312
Epoch 16/18
5667/5667 [==============================] - 8s 1ms/step - loss: 18092.2559 - root_m
ean_squared_error: 134.5075
Epoch 17/18
5667/5667 [==============================] - 10s 2ms/step - loss: 18085.6484 - root_
mean_squared_error: 134.4829
Epoch 18/18
5667/5667 [==============================] - 9s 2ms/step - loss: 18356.4707 - root_m
ean_squared_error: 135.4861
```

```
/usr/local/lib/python3.10/dist-packages/numpy/core/shape_base.py:65: VisibleDeprecat
ionWarning: Creating an ndarray from ragged nested sequences (which is a list-or-tup
le of lists-or-tuples-or ndarrays with different lengths or shapes) is deprecated. I
f you meant to do this, you must specify 'dtype=object' when creating the ndarray.
  ary = asanyarray(ary)
```






```
feature   label          predicted
  value   value          value
          in thousand$   in thousand$
--------------------------------------
 1286      53            238
 1867      92            250
 2191      69            257
 1052      62            233
 1647      80            246
 2312     295            260
 1604     500            245
 1066     342            233
  338     118            218
 1604     128            245
```

Did `population` produce better predictions than `total_rooms` ?

Although population usually converges at a little higher RMSE than total_rooms, training is not totally deterministic. Therefore, population seems to forecast things roughly the same or marginally worse than total_rooms.

Task 4: Define a synthetic feature

You have determined that `total_rooms` and `population` were not useful features. That is, neither the total number of rooms in a neighborhood nor the neighborhood's population successfully predicted the median house price of that neighborhood. Perhaps though, the *ratio* of `total_rooms` to `population` might have some predictive power. That is, perhaps block density relates to median house value.

To explore this hypothesis, do the following:

1. Create a synthetic feature that's a ratio of `total_rooms` to `population`. (If you are new to pandas DataFrames, please study the Pandas DataFrame Ultraquick Tutorial.)
2. Tune the three hyperparameters.
3. Determine whether this synthetic feature produces a lower loss value than any of the single features you tried earlier in this exercise.

In [21]:

```
# Define a synthetic feature named rooms_per_person
training_df["rooms_per_person"] = ? # write your code here.

# Don't change the next line.
my_feature = "rooms_per_person"

# Assign values to these three hyperparameters.
learning_rate = ?
epochs = ?
batch_size = ?

# Don't change anything below this line.
my_model = build_model(learning_rate)
weight, bias, epochs, rmse = train_model(my_model, training_df,
                                         my_feature, my_label,
                                         epochs, batch_size)

plot_the_loss_curve(epochs, rmse)
predict_house_values(15, my_feature, my_label)
```

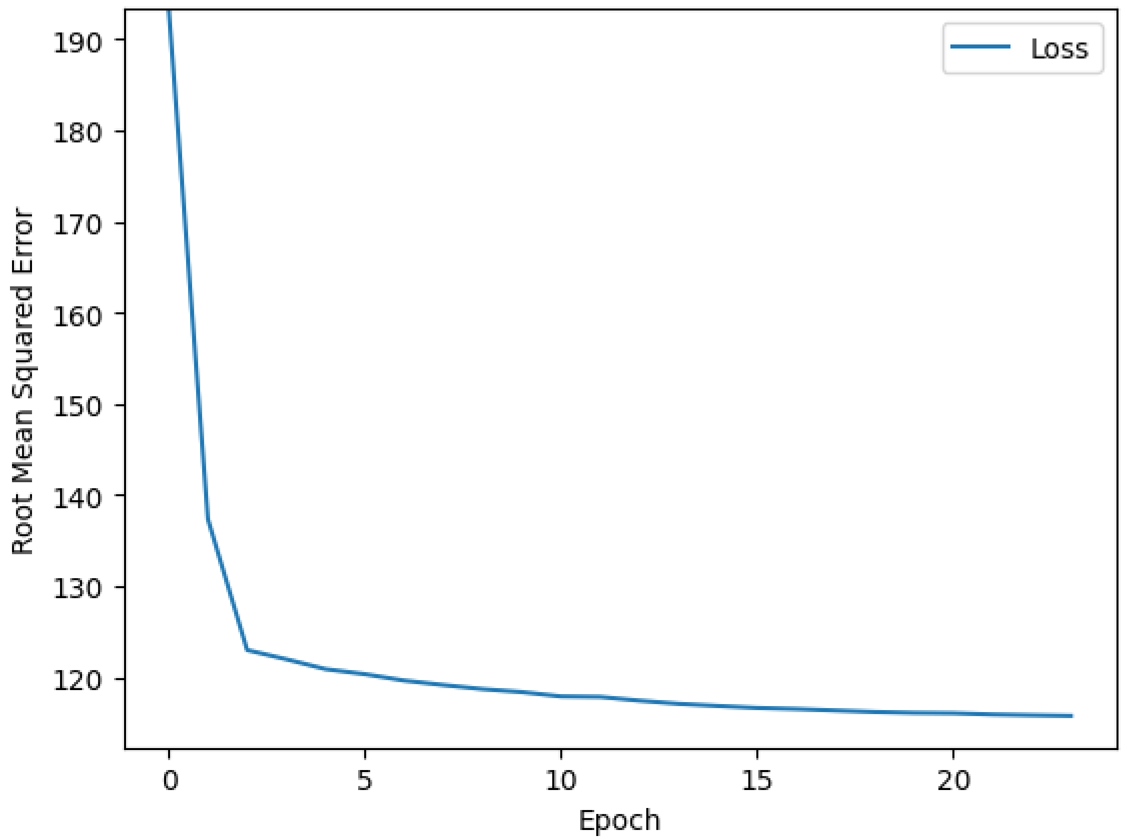
```
  File "<ipython-input-21-b3a0c85a84f7>", line 2
    training_df["rooms_per_person"] = ? # write your code here.
                                      ^
SyntaxError: invalid syntax
```

In [22]:

```
training_df["rooms_per_person"] = training_df["total_rooms"] / training_df["populat
my_feature = "rooms_per_person"

learning_rate = 0.06
epochs = 24
batch_size = 30

my_model = build_model(learning_rate)
weight, bias, epochs, mae = train_model(my_model, training_df,
                                        my_feature, my_label,
                                        epochs, batch_size)

plot_the_loss_curve(epochs, mae)
predict_house_values(15, my_feature, my_label)
```

```
Epoch 1/24
567/567 [==============================] - 2s 2ms/step - loss: 37388.4219 - root_mea
n_squared_error: 193.3609
Epoch 2/24
567/567 [==============================] - 1s 2ms/step - loss: 18866.6270 - root_mea
n_squared_error: 137.3558
Epoch 3/24
567/567 [==============================] - 1s 1ms/step - loss: 15126.0020 - root_mea
n_squared_error: 122.9878
Epoch 4/24
567/567 [==============================] - 1s 2ms/step - loss: 14878.4023 - root_mea
n_squared_error: 121.9771
Epoch 5/24
567/567 [==============================] - 1s 1ms/step - loss: 14615.8623 - root_mea
n_squared_error: 120.8961
Epoch 6/24
567/567 [==============================] - 1s 1ms/step - loss: 14482.7520 - root_mea
n_squared_error: 120.3443
Epoch 7/24
567/567 [==============================] - 1s 2ms/step - loss: 14314.3506 - root_mea
n_squared_error: 119.6426
Epoch 8/24
567/567 [==============================] - 1s 2ms/step - loss: 14197.2119 - root_mea
n_squared_error: 119.1521
Epoch 9/24
567/567 [==============================] - 1s 1ms/step - loss: 14090.7793 - root_mea
n_squared_error: 118.7046
Epoch 10/24
567/567 [==============================] - 1s 1ms/step - loss: 14013.7656 - root_mea
n_squared_error: 118.3798
Epoch 11/24
567/567 [==============================] - 1s 1ms/step - loss: 13900.9570 - root_mea
n_squared_error: 117.9023
Epoch 12/24
567/567 [==============================] - 1s 1ms/step - loss: 13888.5068 - root_mea
n_squared_error: 117.8495
Epoch 13/24
567/567 [==============================] - 1s 2ms/step - loss: 13791.4775 - root_mea
n_squared_error: 117.4371
Epoch 14/24
567/567 [==============================] - 1s 2ms/step - loss: 13710.1006 - root_mea
n_squared_error: 117.0901
Epoch 15/24
567/567 [==============================] - 1s 2ms/step - loss: 13655.5498 - root_mea
n_squared_error: 116.8570
Epoch 16/24
567/567 [==============================] - 1s 1ms/step - loss: 13604.4443 - root_mea
n_squared_error: 116.6381
Epoch 17/24
567/567 [==============================] - 1s 1ms/step - loss: 13577.1875 - root_mea
n_squared_error: 116.5212
Epoch 18/24
567/567 [==============================] - 1s 1ms/step - loss: 13538.8955 - root_mea
n_squared_error: 116.3568
Epoch 19/24
567/567 [==============================] - 1s 1ms/step - loss: 13501.5371 - root_mea
```

```
n_squared_error: 116.1961
Epoch 20/24
567/567 [==============================] - 1s 2ms/step - loss: 13478.9795 - root_mea
n_squared_error: 116.0990
Epoch 21/24
567/567 [==============================] - 1s 1ms/step - loss: 13470.1797 - root_mea
n_squared_error: 116.0611
Epoch 22/24
567/567 [==============================] - 1s 1ms/step - loss: 13437.4932 - root_mea
n_squared_error: 115.9202
Epoch 23/24
567/567 [==============================] - 1s 2ms/step - loss: 13418.6709 - root_mea
n_squared_error: 115.8390
Epoch 24/24
567/567 [==============================] - 1s 1ms/step - loss: 13402.8086 - root_mea
n_squared_error: 115.7705
```



```
feature   label          predicted
  value   value          value
          in thousand$   in thousand$
--------------------------------------
    2      53            190
    2      92            202
    2      69            196
    2      62            213
    1      80            187
    2     295            227
    2     500            212
    2     342            225
    4     118            291
    2     128            215
    2     187            226
    3      80            236
    2     112            227
    2      95            221
    2      69            212
```

Based on the loss values, this synthetic feature produces a better model than the individual features you tried in Task 2 and Task 3. However, the model still isn't creating great predictions.

Task 5. Find feature(s) whose raw values correlate with the label

So far, we've relied on trial-and-error to identify possible features for the model. Let's rely on statistics instead.

A **correlation matrix** indicates how each attribute's raw values relate to the other attributes' raw values. Correlation values have the following meanings:

- `1.0` : perfect positive correlation; that is, when one attribute rises, the other attribute rises.
- `-1.0` : perfect negative correlation; that is, when one attribute rises, the other attribute falls.
- `0.0` : no correlation; the two columns are not linearly related.

In general, the higher the absolute value of a correlation value, the greater its predictive power. For example, a correlation value of -0.8 implies far more predictive power than a correlation of -0.2.

The following code cell generates the correlation matrix for attributes of the California Housing Dataset:

In [23]:
```
# Generate a correlation matrix.
training_df.corr()
```

Out[23]:

	longitude	latitude	housing_median_age	total_rooms	total_bedroom
longitude	1.0	-0.9	-0.1	0.0	0.
latitude	-0.9	1.0	0.0	-0.0	-0.
housing_median_age	-0.1	0.0	1.0	-0.4	-0.
total_rooms	0.0	-0.0	-0.4	1.0	0.
total_bedrooms	0.1	-0.1	-0.3	0.9	1.
population	0.1	-0.1	-0.3	0.9	0.
households	0.1	-0.1	-0.3	0.9	1.
median_income	-0.0	-0.1	-0.1	0.2	-0.
median_house_value	-0.0	-0.1	0.1	0.1	0.
rooms_per_person	-0.1	0.1	-0.1	0.1	0.

The correlation matrix shows nine potential features (including a synthetic feature) and one label (`median_house_value`). A strong negative correlation or strong positive correlation with the label suggests a potentially good feature.

Your Task: Determine which of the nine potential features appears to be the best candidate for a feature?

In [24]:
```
#Given that the label (median house value) and median income have a 0.7 correlation
#The correlations for the other seven possible features are all rather near to zero
#Try using median_income as the feature if you have the time, and see whether the m
```

Correlation matrices don't tell the entire story. In later exercises, you'll find additional ways to unlock predictive power from potential features.

Note: Using `median_income` as a feature may raise some ethical and fairness issues. Towards the end of the course, we'll explore ethical and fairness issues.

In [25]: