6

Regression

**Regression is a supervised learning method for predicting a continuous output** of an event based on the relationship between variables (or features) obtained from a dataset. A **continuous** outcome is a real value such as an integer or floating point value often quantified as amounts and sizes. Regression is a widely popular type of deep learning modeling.

Since regression predicts a quantity, performance is measured error in those predictions. Regression performance can be measured in many ways. But, the most common are mean square error (MSE), mean absolute error (MAE), and root mean squared error (RMSE).

MSE is one of the most commonly used metrics, but least useful when a single bad prediction would ruin the entire model's predicting abilities. That is, when the dataset contains a lot of noise. It is most useful when the dataset contains outliers or unexpected values. Unexpected values are those that are too high or too low.

MAE is not very sensitive to outliers in comparison to MSE since it doesn't punish huge errors. It is typically used when performance is measured on continuous variable data. It provides a linear value that averages the weighted individual differences equally.

RMSE errors are squared before they are averaged. As such, RMSE assigns a higher weight to larger errors. So, RMSE is much more useful when large errors are present and they drastically affect the model's performance. A benefit of RMSE is that units of error score are the same as the predicted value.

We thoroughly work through the famous Boston Housing dataset. We demonstrate how to load the data, build the input pipeline, and model the data. We also show the reader how to use the model to make predictions. We end by modeling a different dataset. The Cars dataset might not be as popular, but we want to give the reader experience with another set of data.

Notebooks for this chapter are located at the following URL: <https://github.com/paperd/tensorflow>.

We need to set up the GPU for each notebook. So, click the Runtime tab, click Change runtime type from the drop-down menu, choose GPU from the Hardware Accelerator drop-down menu, and click Save.

Display the current version of TensorFlow and enable the GPU in Google Colab:

import tensorflow as tf

# display tf version and test if GPU is active

tf.\_\_version\_\_, tf.test.gpu\_device\_name()

The GPU is active if you see '/device:GPU:0' displayed.

# Boston Housing Dataset

**Boston Housing** is a dataset derived from information collected by the U.S. Census Service concerning housing in the area of Boston, Massachusetts. It was obtained from the StatLib archive (<http://lib.stat.cmu.edu/datasets/boston>), and has been used extensively throughout the machine learning literature to benchmark algorithms. The dataset is small in size with only 506 cases.

The name of this dataset is boston. It contains 12 features and 1 outcome (or target). The features are as follows:

1. CRIM - per capita crime rate by town
2. ZN - proportion of residential land zoned for lots over 25,000 sq.ft.
3. INDUS - proportion of non-retail business acres per town.
4. CHAS - Charles River dummy variable (1 if tract bounds river; 0 otherwise)
5. NOX - nitric oxides concentration (parts per 10 million)
6. RM - average number of rooms per dwelling
7. AGE - proportion of owner-occupied units built prior to 1940
8. DIS - weighted distances to five Boston employment centres
9. RAD - index of accessibility to radial highways
10. TAX - full-value property-tax rate per $10,000
11. PTRATIO - pupil-teacher ratio by town
12. LSTAT - % lower status of the population

The target is:

* MEDV - median value of owner-occupied homes in $1000's

Data was collected in the '70s, so don't be shocked by the low median value of homes.

## Boston Data

You can access any dataset for this book directly from GitHub with a few simple steps:

1. visit the book URL: <https://github.com/paperd/tensorflow>
2. locate the dataset and click on it
3. click the Raw button
4. copy the URL to Colab and assign it to a variable
5. read the dataset with the Pandas \*\*read\_csv\*\* method

For convenience, we've already located the appropriate URL and assigned to a variable:

url = 'https://raw.githubusercontent.com/paperd/tensorflow/\

master/chapter6/data/boston.csv'

Read the dataset into a Pandas DataFrame:

import pandas as pd

data = pd.read\_csv(url)

Verify that the dataset was read properly:

data.head()

## Explore the Dataset

Get datatypes:

data.dtypes

All features are datatype float64 or int64. The label MEDV is datatype float64.

Get general information:

data.info()

The dataset contains 506 records.

Create a DataFrame that holds basic statistics with the describe method and transpose it for easier viewing:

data\_t = data.describe()

desc = data\_t.T

desc

Target specific statistics from the transposed DataFrame:

desc[['mean', 'std']]

Describe a specific feature from the original DataFrame:

data.describe().LSTAT

Get columns:

cols = list(data)

cols

## Create Feature and Target Sets

Create the target:

# create a copy of the DataFrame

df = data.copy()

# create the target

target = df.pop('MEDV')

print (target.head())

Since we popped MEDV from the copy, it should only contain the features:

df.head()

## Get Feature Names from the Features DataFrame

Since the target is no longer part of the DataFrame, it’s easy to get the features:

feature\_cols = list(df)

feature\_cols

Get the number of features:

len(feature\_cols)

## Convert Features and Labels

Convert Pandas DataFrame values to float with the values method:

features = df.values

labels = target.values

type(features), type(labels)

## Split Dataset into Train and Test Sets

Listing 6-1 creates train and test sets:

from sklearn.model\_selection import train\_test\_split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

    features, labels, test\_size=0.33, random\_state=0)

br = '\n'

print ('X\_train shape:', end=' ')

print (X\_train.shape, br)

print ('X\_test shape:', end=' ')

print (X\_test.shape)

Listing 6-1. Train and test sets

## Scale Data and Create TensorFlow Tensors

With image data, we scale by dividing each element by 255.0 to ensure that each input parameter (a pixel, in our case) has a similar data distribution. However, features represented by continuous values are scaled differently. We scale continuous data to have a mean (μ) of 0 and standard deviation (σ) of 1. A σ of 1 is called unit variance.

Listing 6-2 scales train and test data:

# scale feature data and create TensorFlow tensors

from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

X\_train\_std = scaler.fit\_transform(X\_train)

X\_test\_std = scaler.fit\_transform(X\_test)

train = tf.data.Dataset.from\_tensor\_slices(

    (X\_train\_std, y\_train))

test = tf.data.Dataset.from\_tensor\_slices(

    (X\_test\_std, y\_test))

Listing 6-2. Scale train and test sets

Let’s view the first tensor as shown in Listing 6-3:

def see\_samples(data, num):

  for feat, targ in data.take(num):

    print ('Features: {}'.format(feat), br)

    print ('Target: {}'.format(targ))

n = 1

see\_samples(train, n)

Listing 6-3. Display the first tensor

The first sample looks as we expect.

## Prepare Tensors for Training

Shuffle (where appropriate), batch, and prefetch train and test data:

BATCH\_SIZE, SHUFFLE\_BUFFER\_SIZE  = 16, 100

train\_bs = train.shuffle(

    SHUFFLE\_BUFFER\_SIZE).batch(BATCH\_SIZE).prefetch(1)

test\_bs = test.batch(BATCH\_SIZE).prefetch(1)

## Create a Model

If we don't have a lot of training data, one technique to avoid overfitting is to create a small network with few hidden layers. We do just that!

The 64 neuron input layer accommodates our 12 input features. We have one hidden layers with 64 neurons. The output layer has 1 neuron because we are using regression with one target.

Create a model as shown in Listing 6-4:

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense, Flatten

from tensorflow import keras

import numpy as np

# clear any previous model

keras.backend.clear\_session()

# plant a random seed for replication purposes

np.random.seed(0)

tf.random.set\_seed(0)

# notice input shape accommodates 12 features!

model = Sequential([

  Dense(64, activation='relu', input\_shape=[12,]),

  Dense(64, activation='relu'),

  Dense(1)

])

Listing 6-4. Create a model

## Model Summary

model.summary()

Output shape of the first layer is (None, 64) because we have 64 neurons at this layer. We get parameters of 832 by multiplying 64 neurons at this layer by 12 features, and adding 64 neurons at this layer.

Output shape of the second layer is (None, 64) because we have 64 neurons at this layer. We get parameters of 4,160 by multiplying 64 neurons at this layer by 64 neurons from the previous layer, and adding 64 at this layer.

Output shape of the third layer is (None, 1) because we have one target. We get parameters of 65 by adding 64 neurons from the previous layer to 1 neuron at this layer.

## Compile the Model

rmse = tf.keras.metrics.RootMeanSquaredError()

model.compile(loss='mse', optimizer='RMSProp',

              metrics=[rmse, 'mae', 'mse'])

Mean Squared Error (MSE) is a common loss function used for regression problems. Mean Absolute Error (MAE) and RMSE are also commonly used metrics. With some experimentation, we found that optimizer RMSProp performed pretty well with this dataset.

## Train the Model

Train the model for 50 epochs:

history = model.fit(train\_bs, epochs=50,

                    validation\_data=test\_bs)

## Visualize Training

Create variable hist to holds the model's history as a Pandas DataFrame. Create another variable hist['epoch'] to hold epoch history. Display the last five rows to get an idea about performance.

Here’s the code:

hist = pd.DataFrame(history.history)

hist['epoch'] = history.epoch

hist.tail()

Build the plots as shown in Listing 6-5:

import matplotlib.pyplot as plt

def plot\_history(history, limit1, limit2):

  hist = pd.DataFrame(history.history)

  hist['epoch'] = history.epoch

  plt.figure()

  plt.xlabel('epoch')

  plt.ylabel('MAE [MPG]')

  plt.plot(hist['epoch'], hist['mae'],

           label='Train Error')

  plt.plot(hist['epoch'], hist['val\_mae'],

           label = 'Val Error')

  plt.ylim([0, limit1])

  plt.legend()

  plt.title('MAE by Epoch')

  plt.show()

  plt.clf()

  plt.figure()

  plt.xlabel('Epoch')

  plt.ylabel('MSE [MPG]')

  plt.plot(hist['epoch'], hist['mse'],

           label='Train Error')

  plt.plot(hist['epoch'], hist['val\_mse'],

           label = 'Val Error')

  plt.ylim([0, limit2])

  plt.legend()

  plt.title('MSE by Epoch')

  plt.show()

  plt.clf()

  plt.figure()

  plt.xlabel('Epoch')

  plt.ylabel('RMSE [MPG]')

  plt.plot(hist['epoch'], hist['root\_mean\_squared\_error'],

           label='Train Error')

  plt.plot(hist['epoch'], hist['val\_root\_mean\_squared\_error'],

           label = 'Val Error')

  plt.ylim([0, limit2])

  plt.legend()

  plt.title('RMSE by Epoch')

  plt.show()

# set limits to make plot readable

mae\_limit, mse\_limit = 10, 100

plot\_history(history, mae\_limit, mse\_limit)

Listing 6-5. Visualize training performance

Since the validation error is a worse than the train error, the model is overfitting. What can we do? The first step is to estimate when performance begins to degrade. From the visualizations, can you see when this happens?

## Early Stopping

With classification, our goal is to maximize accuracy. Of course, we also want to minimize loss. With regression, our goal is to minimize MSE or one of the other common error metrics. From the visualizations, we see that our model is overfitting because validation error is higher than training error. We also see that once training error and validation error cross, performance begins to degrade.

There is one simple tuning experiment we can run to make this model more useful. We can stop the model when training and validation error are very close to each other. This technique is called early stopping. **Early stopping** is a widely used approach that stops training at the point when performance on a validation dataset starts to degrade.

Let's modify our training experiment to automatically stop training when the validation score doesn't improve. We use an EarlyStopping callback that tests a training condition for every epoch. If a set amount of epochs elapse without showing improvement, training is automatically stopped.

All we need to do is update the fit method and retrain as shown in Listing 6-6:

# clear the previous model

keras.backend.clear\_session()

# plant a random seed for replication purposes

np.random.seed(0)

tf.random.set\_seed(0)

# monitor 'val\_loss' for early stopping

early\_stop = keras.callbacks.EarlyStopping(monitor='val\_loss')

history = model.fit(train\_bs, epochs=50,

                    validation\_data=test\_bs,

                    callbacks=[early\_stop])

Listing 6-6. Early stopping

Instead of allowing the algorithm to automatically early stop, we can add some control. Just include a parameter that forces the model to continue to a point that gives us the best performance. The patience parameter can be set to a given a number of epochs after which training will be stopped if there is no improvement.

Let's try this and see what happens as shown in Listing 6-7:

# clear the previous model

keras.backend.clear\_session()

# plant a random seed for replication purposes

np.random.seed(0)

tf.random.set\_seed(0)

# set number of patience epochs

n = 4

early\_stop = keras.callbacks.EarlyStopping(

    monitor='val\_loss', patience=n)

history = model.fit(train\_bs, epochs=50,

                    validation\_data=test\_bs,

                    callbacks=[early\_stop])

Listing 6-7. Early stopping with patience

Experiment with the patience parameter to find better results.

Let’s visualize:

hist = pd.DataFrame(history.history)

hist['epoch'] = history.epoch

train\_limit, test\_limit = 10, 100

plot\_history(history, train\_limit, test\_limit)

## Remove Bad Data

The Boston dataset has some inherent bad data. What is wrong with the data? Prices of homes are capped at $50,000 because the Census Service censored the data. They decided to set the maximum value of the price variable to 50k USD, so no price can go beyond that value.

What do we do? While maybe not ideal, we can remove data with prices at or above 50k USD. This is not ideal because we may be removing perfectly good data, but there is no way to know this. Another reason is because the dataset is so small to begin with. Neural nets are meant to perform at their best with larger datasets.

To explore this topic further, we recommend this URL:

<https://towardsdatascience.com/things-you-didnt-know-about-the-boston-housing-dataset-2e87a6f960e8>

## Get Data

We don’t want to attempt to clean a dataset processed for TensorFlow consumption. So, just reload the raw dataset:

# get the raw data

url = 'https://raw.githubusercontent.com/paperd/tensorflow/\

master/chapter6/data/boston.csv'

boston = pd.read\_csv(url)

Verify data:

boston.head()

## Remove Noise

Remove the bad data, which hopefully reduces inherent noise:

print ('data set before removing noise:', boston.shape)

# remove noise

noise = boston.loc[boston['MEDV'] >= 50]

data = boston.drop(noise.index)

print ('data set without noise:', data.shape)

**Noise** is the irrelevant information or randomness in a dataset. We removed several records from the dataset.

## Create Feature and Target Data

Create feature and target sets:

# create a copy of the DataFrame

df = data.copy()

# create feature and target sets

target, features = df.pop('MEDV'), df.values

labels = target.values

## Build the Input Pipeline

Create the input pipeline by splitting data into train and test sets, scaling feature data, and slicing data into TensorFlow consumable pieces. Finish the pipeline by shuffling (where appropriate), batching, and prefetching data.

Listing 6-8 includes the code to build the input pipeline:

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

    features, labels, test\_size=0.33, random\_state=0)

# standardize feature data and create TensorFlow tensors

X\_train\_std = scaler.fit\_transform(X\_train)

X\_test\_std = scaler.fit\_transform(X\_test)

# slice data for TensorFlow consumption

train = tf.data.Dataset.from\_tensor\_slices(

    (X\_train\_std, y\_train))

test = tf.data.Dataset.from\_tensor\_slices(

    (X\_test\_std, y\_test))

# shuffle, batch, prefetch

BATCH\_SIZE = 16

SHUFFLE\_BUFFER\_SIZE = 100

train\_n = train.shuffle(

    SHUFFLE\_BUFFER\_SIZE).batch(BATCH\_SIZE).prefetch(1)

test\_n = test.batch(BATCH\_SIZE).prefetch(1)

Listing 6-8. Build the input pipeline

## Compile and Train

Listing 6-9 includes the code for compiling and training the model:

rmse = tf.keras.metrics.RootMeanSquaredError()

model.compile(loss='mse', optimizer='RMSProp',

              metrics=[rmse, 'mae', 'mse'])

keras.backend.clear\_session()

# plant a random seed for replication purposes

np.random.seed(0)

tf.random.set\_seed(0)

n = 4

early\_stop = keras.callbacks.EarlyStopping(

    monitor='val\_loss', patience=n)

history = model.fit(train\_n, epochs=50,

                    validation\_data=test\_n,

                    callbacks=[early\_stop])

Listing 6-9. Compile and train the model

## Visualize

Plot results:

hist = pd.DataFrame(history.history)

hist['epoch'] = history.epoch

train\_limit, test\_limit = 10, 100

plot\_history(history, train\_limit, test\_limit)

Our model is not perfect, but we did improve performance.

## Generalize on Test Data

loss, rmse, mae, mse = model.evaluate(test\_n, verbose=2)

print ()

print('"Testing set Mean Abs Error: {:5.2f} thousand dollars'.

      format(mae))

MAE provides a good idea of performance for linear continuous data in an easy-to-understand manner. With this dataset, we can expect model predictions to be off by the MAE value in thousands of dollars on average.

## Make Predictions

Use the predict method to make predictions from processed test data test\_n:

predictions = model.predict(test\_n)

Display the first prediction:

# predicted housing price

first = predictions[0]

print ('predicted price:', first[0], 'thousand')

# actual housing price

print ('actual price:', y\_test[0], 'thousand')

Compare predicted and actual prices to gauge model performance.

Display the first five predictions and compare against actual target values:

five = predictions[:5]

print (five, br)

actuals = y\_test[:5]

print (actuals)

## Visualize Predictions

Listing 6-10 displays predictions against actual housing prices:

plt.scatter(y\_test, predictions)

plt.xlabel('True Values [price]')

plt.ylabel('Predictions [price]')

plt.axis('equal')

plt.axis('square')

plt.xlim([0,plt.xlim()[1]])

plt.ylim([0,plt.ylim()[1]])

\_ = plt.plot([-100, 100], [-100, 100])

Listing 6-10. Predictions against actual prices plot

The diagonal line is a plot of the actual housing prices. The further away a prediction is from the diagonal, the more erroneous it is.

## Load Boston Data from Scikit-Learn

Since Boston data is included in sklearn.datasets, we can load it from this environment:

from sklearn import datasets

dataset = datasets.load\_boston()

data, target = dataset.data, dataset.target

Access keys:

dataset.keys()

The list of keys informs us about accessing feature names:

feature\_names = dataset.feature\_names

feature\_names

Notice that the Sklearn dataset has an extra feature B. This column might be considered by some to be controversial because it singles out Black (or African American) people in a township.

We want to remove noise from the entire dataset, so build a DataFrame with feature data and add target data:

df\_sklearn = pd.DataFrame(dataset.data, columns=feature\_names)

df\_sklearn['MEDV'] = dataset.target

df\_sklearn.head()

Check information:

df\_sklearn.info()

## Remove Noise

# remove noisy data

print ('data set before removing noise:', df\_sklearn.shape)

noise = df\_sklearn.loc[df\_sklearn['MEDV'] >= 50]

df\_clean = df\_sklearn.drop(noise.index)

print ('data set without noise:', df\_clean.shape)

## Build the Input Pipeline

Build the pipeline as shown in Listing 6-11:

# create a copy of the DataFrame

df = df\_clean.copy()

# create the target

target = df.pop('MEDV')

# convert features and target data

features = df.values

labels = target.values

# create train and test sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

    features, labels, test\_size=0.33, random\_state=0)

X\_train\_std = scaler.fit\_transform(X\_train)

X\_test\_std = scaler.fit\_transform(X\_test)

# slice data into a TensorFlow consumable form

train = tf.data.Dataset.from\_tensor\_slices(

    (X\_train\_std, y\_train))

test = tf.data.Dataset.from\_tensor\_slices(

    (X\_test\_std, y\_test))

# finalize the pipeline

BATCH\_SIZE = 16

SHUFFLE\_BUFFER\_SIZE = 100

train\_sk = train.shuffle(

    SHUFFLE\_BUFFER\_SIZE).batch(BATCH\_SIZE).prefetch(1)

test\_sk = test.batch(BATCH\_SIZE).prefetch(1)

Listing 6-11. Build the input pipeline

## Model Data

Model data as shown in Listing 6-12:

# clear any previous model

keras.backend.clear\_session()

# plant a random seed for replication purposes

np.random.seed(0)

tf.random.set\_seed(0)

# new model with 13 input features

model = Sequential([

  Dense(64, activation='relu', input\_shape=[13,]),

  Dense(64, activation='relu'),

  Dense(1)

])

# compile the new model

rmse = tf.keras.metrics.RootMeanSquaredError()

model.compile(loss='mse', optimizer='RMSProp',

              metrics=[rmse, 'mae', 'mse'])

# train

n = 4

early\_stop = keras.callbacks.EarlyStopping(monitor='val\_loss', patience=n)

history = model.fit(train\_sk, epochs=50,

                    validation\_data=test\_sk,

                    callbacks=[early\_stop])

Listing 6-12. Model data

# Model Cars Data

Let’s practice with another dataset.

## Get Cars Data from GitHub

We’ve already located the URL and assigned it to a variable:

cars\_url = 'https://raw.githubusercontent.com/paperd/tensorflow/\

master/chapter6/data/cars.csv'

Read data into a Pandas DataFrame:

cars = pd.read\_csv(cars\_url)

Verify data:

cars.head()

Get information about the dataset:

cars.info()

## Convert Categorical Column to Numeric

Machine learning algorithms can only train numeric data. So, we must convert any non-numeric feature. The Origin column is categorical, not numeric. To remedy, one solution is to encode the data as one-hot. **One hot encoding** is a process that converts categorical data into a numeric form for machine learning algorithm consumption.

We start by slicing off the Origin feature column from the original DataFrame into its own DataFrame. We then use this DataFrame as a template to build a new feature column in the original DataFrame for each category from the original Origin feature.

Create a copy of the DataFrame:

# create a copy of DataFrame

df = cars.copy()

origin = df.pop('Origin')

Define one-hot encoded feature columns for US, Europe, and Japanese cars:

df['US'] = (origin == 'US') \* 1.0

df['Europe'] = (origin == 'Europe') \* 1.0

df['Japan'] = (origin == 'Japan') \* 1.0

df.tail(8)

For US origin, we assign 1.0 0.0 0.0. For Europe origin, we assign 0.0 1.0 0.0. For Japan origin, we assign 0.0 0.0 1.0. So, we replace the Origin feature with three one-hot encoded features.

## Slice Extraneous Data

Since the name of each car has no impact on any predictions we might want to make, we can tuck it away into its own DataFrame in case we want to revisit it in the future:

try:

  name = df.pop('Car')

except:

  print("An exception occurred")

If an exception occurs, the Car column has already been removed. You can run this piece of code several times with no impact results.

Verify:

df.tail(8)

## Create Features and Labels

Our goal is to predict miles per gallon for cars in this dataset. So, the target is MPG and the features are the remaining feature columns.

Create feature and target sets as shown in Listing 6-13:

# create data sets

features = df.copy()

target = features.pop('MPG')

# get feature names

feature\_cols = list(features)

print (feature\_cols)

# get number of features

num\_features = len(feature\_cols)

print (num\_features)

# convert feature and target data to float

features = features.values

labels = target.values

(type(features), type(labels))

Listing 6-13. Create feature and target sets

## Build the Input Pipeline

Build the input pipeline as shown in Listing 6-14:

# split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

    features, labels, test\_size=0.33, random\_state=0)

print ('X\_train shape:', end=' ')

print (X\_train.shape, br)

print ('X\_test shape:', end=' ')

print (X\_test.shape)

# scale

X\_train\_std = scaler.fit\_transform(X\_train)

X\_test\_std = scaler.fit\_transform(X\_test)

# slice

train = tf.data.Dataset.from\_tensor\_slices(

    (X\_train\_std, y\_train))

test = tf.data.Dataset.from\_tensor\_slices(

    (X\_test\_std, y\_test))

# shuffle, batch, prefetch

BATCH\_SIZE = 16

SHUFFLE\_BUFFER\_SIZE = 100

train\_cars = train.shuffle(

    SHUFFLE\_BUFFER\_SIZE).batch(BATCH\_SIZE).prefetch(1)

test\_cars = test.batch(BATCH\_SIZE).prefetch(1)

Listing 6-14. Build the input pipeline

## Model Data

Model data as shown in Listing 6-15:

# clear any previous model

keras.backend.clear\_session()

# create the model

model = Sequential([

  Dense(64, activation='relu', input\_shape=[num\_features]),

  Dense(64, activation='relu'),

  Dense(1)

])

# compile

rmse = tf.keras.metrics.RootMeanSquaredError()

optimizer = tf.keras.optimizers.RMSprop(0.001)

model.compile(loss='mse',

              optimizer=optimizer,

              metrics=[rmse, 'mae', 'mse'])

# train

keras.backend.clear\_session()

n = 4

early\_stop = keras.callbacks.EarlyStopping(

    monitor='val\_loss', patience=n)

car\_history = model.fit(train\_cars, epochs=100,

                        validation\_data=test\_cars,

                        callbacks=[early\_stop])

Listing 6-15. Model data

## Inspect the Model

model.summary()

Output shape of the first layer is (None, 64) because we have 64 neurons at this layer. We get parameters of 640 by multiplying 64 neurons at this layer by 9 features, and adding 64 neurons at this layer.

Output shape of the second layer is (None, 64) because we have 64 neurons at this layer. We get parameters of 4,160 by multiplying 64 neurons at this layer by 64 neurons from the previous layer, and adding 64 at this layer.

Output shape of the third layer is (None, 1) because we have one target. We get parameters of 65 by adding 64 neurons from the previous layer to 1 neuron at this layer.

## Visualize Training

hist = pd.DataFrame(history.history)

hist['epoch'] = history.epoch

train\_limit, test\_limit = 10, 100

plot\_history(history, train\_limit, test\_limit)

## Generalize on Test Data

loss, rmse, mae, mse = model.evaluate(test\_cars, verbose=2)

print ()

print('"Testing set Mean Abs Error: {:5.2f} MPG'.format(mae))

## Make Predictions

predictions = model.predict(test\_cars)

## Visualize Predictions

Visualize predictions as shown in Listing 6-16:

plt.scatter(y\_test, predictions)

plt.xlabel('True Values [MPG]')

plt.ylabel('Predictions [MPG]')

plt.axis('equal')

plt.axis('square')

plt.xlim([0,plt.xlim()[1]])

plt.ylim([0,plt.ylim()[1]])

\_ = plt.plot([-100, 100], [-100, 100])

Listing 6-16. Visualize predictions for cars data

The further the prediction is away from the diagonal true values line, the more erroneous it is.