learning-using-keras-regression

November 21, 2024

Importing the necessary libraries and packages

[12]:
```
import warnings
warnings.filterwarnings('ignore')
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
import tensorflow as tf
```

[31]:
```
print(tf.__version__)
```

```
2.18.0
```

Downloading the dataset

- California Housing dataset

[2]:
```
housing = fetch_california_housing()
```

Inspecting the data and it's shape

[3]:
```
housing.data, housing.data.shape
```

[3]:
```
(array([[   8.3252    ,   41.        ,    6.98412698, …,    2.55555556,
          37.88      , -122.23      ],
        [   8.3014    ,   21.        ,    6.23813708, …,    2.10984183,
          37.86      , -122.22      ],
        [   7.2574    ,   52.        ,    8.28813559, …,    2.80225989,
          37.85      , -122.24      ],
        …,
        [   1.7       ,   17.        ,    5.20554273, …,    2.3256351 ,
          39.43      , -121.22      ],
        [   1.8672    ,   18.        ,    5.32951289, …,    2.12320917,
          39.43      , -121.32      ],
        [   2.3886    ,   16.        ,    5.25471698, …,    2.61698113,
          39.37      , -121.24      ]]),
 (20640, 8))
```

Inspecting the data target and it's shape

[4]:
```
housing.target, housing.target.shape
```

[4]: (array([4.526, 3.585, 3.521, …, 0.923, 0.847, 0.894]), (20640,))

Splitting the dataset

- Splitting the dataset into training and testing dataset (75:25 percentage split)

[5]:
```
X_train_full, X_test, y_train_full, y_test = train_test_split(housing.data,␣
↪housing.target, random_state=42)
```

[6]:
```
print(X_train_full.shape, y_train_full.shape)
print(X_test.shape, y_test.shape)
```

```
(15480, 8) (15480,)
(5160, 8) (5160,)
```

- Splitting the training dataset into training and validation dataset (75:25 percentage split)

[7]:
```
X_train, X_valid, y_train, y_valid = train_test_split(X_train_full,␣
↪y_train_full, random_state=42)
```

[8]:
```
print(X_train.shape, y_train.shape)
print(X_valid.shape, y_valid.shape)
```

```
(11610, 8) (11610,)
(3870, 8) (3870,)
```

0.0.1 Type 1 Model : Initializing the model

- Flatten() layer is not used in this example.
- Instead a Normalization layer is used as the first layer and does the same thing as Scikit-Learn's StandardScaler().
- Must be fitted to the training data using it's adapt() methods before calling the model's fit() method

The **Normalization layer** learns the feature means and standard deviations in the training data when you call the adapt() method. Yet when you display the model's summary, these statistics are listed as non trainable. This is because these parameters are not affected by gradient descent.

[9]:
```
tf.random.set_seed(42)
```

[21]:
```
norm_layer = tf.keras.layers.Normalization(input_shape=X_train.shape[1:])
model = tf.keras.Sequential([
    norm_layer,
    tf.keras.layers.Dense(50, activation='relu'),
    tf.keras.layers.Dense(50, activation='relu'),
    tf.keras.layers.Dense(50, activation='relu'),
    tf.keras.layers.Dense(1)])
```

Setting the optimizer

[22]:
```
optimizer = tf.keras.optimizers.Adam(learning_rate=1e-3)
```

Setting the model's training metrics

[23]:
```
model.compile(loss="mse", optimizer=optimizer, metrics=["RootMeanSquaredError"])
```

Adapt the training data to the normalization layer

[24]:
```
norm_layer.adapt(X_train)
```

Train the model on training data using specified number of epochs

[25]:
```
history = model.fit(X_train, y_train, epochs=20, validation_data=(X_valid,
    y_valid))
```

```
Epoch 1/20
363/363          1s 1ms/step -
RootMeanSquaredError: 1.1591 - loss: 1.4237 - val_RootMeanSquaredError: 0.9103 -
val_loss: 0.8287
Epoch 2/20
363/363          0s 1ms/step -
RootMeanSquaredError: 0.6435 - loss: 0.4146 - val_RootMeanSquaredError: 0.6084 -
val_loss: 0.3701
Epoch 3/20
363/363          0s 1ms/step -
RootMeanSquaredError: 0.6060 - loss: 0.3674 - val_RootMeanSquaredError: 0.8837 -
val_loss: 0.7809
Epoch 4/20
363/363          0s 1ms/step -
RootMeanSquaredError: 0.5923 - loss: 0.3510 - val_RootMeanSquaredError: 1.1057 -
val_loss: 1.2226
Epoch 5/20
363/363          0s 1ms/step -
RootMeanSquaredError: 0.5820 - loss: 0.3389 - val_RootMeanSquaredError: 1.0742 -
val_loss: 1.1540
Epoch 6/20
363/363          0s 1ms/step -
RootMeanSquaredError: 0.5759 - loss: 0.3317 - val_RootMeanSquaredError: 1.5071 -
val_loss: 2.2712
Epoch 7/20
363/363          0s 934us/step -
RootMeanSquaredError: 0.5718 - loss: 0.3271 - val_RootMeanSquaredError: 0.9234 -
val_loss: 0.8526
Epoch 8/20
363/363          0s 1ms/step -
RootMeanSquaredError: 0.5648 - loss: 0.3191 - val_RootMeanSquaredError: 0.8959 -
val_loss: 0.8026
Epoch 9/20
```

```
363/363              0s 988us/step -
RootMeanSquaredError: 0.5563 - loss: 0.3096 - val_RootMeanSquaredError: 0.5539 -
val_loss: 0.3068
Epoch 10/20
363/363              0s 1ms/step -
RootMeanSquaredError: 0.5518 - loss: 0.3046 - val_RootMeanSquaredError: 0.5721 -
val_loss: 0.3273
Epoch 11/20
363/363              0s 1ms/step -
RootMeanSquaredError: 0.5476 - loss: 0.3000 - val_RootMeanSquaredError: 0.5610 -
val_loss: 0.3147
Epoch 12/20
363/363              0s 1ms/step -
RootMeanSquaredError: 0.5432 - loss: 0.2952 - val_RootMeanSquaredError: 0.5622 -
val_loss: 0.3161
Epoch 13/20
363/363              0s 1ms/step -
RootMeanSquaredError: 0.5392 - loss: 0.2909 - val_RootMeanSquaredError: 0.5681 -
val_loss: 0.3227
Epoch 14/20
363/363              0s 1ms/step -
RootMeanSquaredError: 0.5364 - loss: 0.2878 - val_RootMeanSquaredError: 0.5309 -
val_loss: 0.2819
Epoch 15/20
363/363              0s 976us/step -
RootMeanSquaredError: 0.5329 - loss: 0.2841 - val_RootMeanSquaredError: 0.5499 -
val_loss: 0.3023
Epoch 16/20
363/363              0s 1ms/step -
RootMeanSquaredError: 0.5316 - loss: 0.2827 - val_RootMeanSquaredError: 0.5311 -
val_loss: 0.2821
Epoch 17/20
363/363              0s 1ms/step -
RootMeanSquaredError: 0.5323 - loss: 0.2834 - val_RootMeanSquaredError: 0.6966 -
val_loss: 0.4852
Epoch 18/20
363/363              0s 1ms/step -
RootMeanSquaredError: 0.5275 - loss: 0.2783 - val_RootMeanSquaredError: 0.5416 -
val_loss: 0.2933
Epoch 19/20
363/363              0s 1ms/step -
RootMeanSquaredError: 0.5250 - loss: 0.2757 - val_RootMeanSquaredError: 0.5675 -
val_loss: 0.3221
Epoch 20/20
363/363              0s 1ms/step -
RootMeanSquaredError: 0.5224 - loss: 0.2730 - val_RootMeanSquaredError: 0.5398 -
val_loss: 0.2914
```

Evaluate the model's performance on testing data

[26]:
```
mse_test, rmse_test = model.evaluate(X_test, y_test)
```

```
162/162              0s 652us/step -
RootMeanSquaredError: 0.5293 - loss: 0.2803
```

[27]:
```
print(mse_test, rmse_test)
```

```
0.2853373885154724 0.5341697931289673
```

Predict the output values for the first 3 datapoints

[28]:
```
X_new = X_test[:3]
y_pred = model.predict(X_new)
```

```
1/1              0s 60ms/step
```

[29]:
```
y_pred
```

[29]:
```
array([[0.45630956],
       [1.4869951 ],
       [5.008497  ]], dtype=float32)
```

0.0.2 Type 2 Model : Building a Wide and Deep neural network

- Non sequential neural network is a Wide and Deep Neural Network
- Heng-Tze Cheng et al (2016 paper)
- It connects all or part of the inputs directly to the output layer.
- This architecture makes it possible for the neural network to learn both deep patterns (using the deep path) and simple rules(through the short path)
- A regular MLP forces all the data to flow through the full stack of layers, thus simple patterns in the data may end up being distorted by this sequence of transformations

Defining layers of the model

1. Normalization layer to standardize the inputs
2. Two Dense layers with 30 neurons each using the ReLU activation function
3. Concatenate layer
4. Dense layer with a single neuron for the output layer, without any activation function

[33]:
```
normalization_layer = tf.keras.layers.Normalization()
hidden_layer1 = tf.keras.layers.Dense(30, activation='relu')
hidden_layer2 = tf.keras.layers.Dense(30, activation='relu')
concat_layer = tf.keras.layers.Concatenate()
output_layer = tf.keras.layers.Dense(1)
```

Establishing the flow of data in model layers

1. input_ - Input object. It is a specification of the kind of input the model will get, including it's shape and optionally its dtype, which defaults to 32 bits floats. A model may actually have multiple inputs. Input object is just a data specification.
2. Used the Normalization layer just like a function, passing it the Input object. It's called the functional API. No actual data is being processed yet. Only keras is being told how it should connect the layers together.The input and output are both symbolic. Normalized doesn't store any actual data, it's just used to construct the model.
3. concat_layer - concatenate the input and the second hidden layer's output.

[34]:
```
input_ = tf.keras.layers.Input(shape=X_train.shape[1:])
normalized = normalization_layer(input_)
hidden1 = hidden_layer1(normalized)
hidden2 = hidden_layer2(hidden1)
concat = concat_layer([normalized, hidden2])
output = output_layer(concat)
```

Creation of Keras model from inputs and outputs

[38]:
```
model = tf.keras.Model(inputs=[input_], outputs=[output])
```

Defining the optimizer

[39]:
```
optimizer1 = tf.keras.optimizers.Adam(learning_rate=1e-3)
```

Model compilation using model training metrics

[40]:
```
model.compile(loss="mse", optimizer=optimizer1,␣
↪metrics=["RootMeanSquaredError"])
```

Fitting the model

[41]:
```
history = model.fit(X_train, y_train, epochs=20, validation_data=(X_valid,␣
↪y_valid))
```

```
Epoch 1/20
363/363              1s 1ms/step -
RootMeanSquaredError: 98.5840 - loss: 13144.9893 - val_RootMeanSquaredError:
13.7108 - val_loss: 187.9848
Epoch 2/20
363/363              0s 1ms/step -
RootMeanSquaredError: 2.6266 - loss: 6.9281 - val_RootMeanSquaredError: 12.9485
- val_loss: 167.6648
Epoch 3/20
363/363              0s 950us/step -
RootMeanSquaredError: 1.9082 - loss: 3.6802 - val_RootMeanSquaredError: 11.5206
- val_loss: 132.7237
Epoch 4/20
363/363              0s 882us/step -
RootMeanSquaredError: 1.5835 - loss: 2.5444 - val_RootMeanSquaredError: 10.2225
```

```
- val_loss: 104.4997
Epoch 5/20
363/363              0s 966us/step -
RootMeanSquaredError: 1.3602 - loss: 1.8849 - val_RootMeanSquaredError: 8.9722 -
val_loss: 80.5004
Epoch 6/20
363/363              0s 1ms/step -
RootMeanSquaredError: 1.2263 - loss: 1.5357 - val_RootMeanSquaredError: 7.7842 -
val_loss: 60.5934
Epoch 7/20
363/363              0s 988us/step -
RootMeanSquaredError: 1.1364 - loss: 1.3156 - val_RootMeanSquaredError: 6.6950 -
val_loss: 44.8229
Epoch 8/20
363/363              0s 988us/step -
RootMeanSquaredError: 1.0486 - loss: 1.1081 - val_RootMeanSquaredError: 5.6728 -
val_loss: 32.1808
Epoch 9/20
363/363              0s 998us/step -
RootMeanSquaredError: 1.0188 - loss: 1.0466 - val_RootMeanSquaredError: 4.7339 -
val_loss: 22.4094
Epoch 10/20
363/363              0s 969us/step -
RootMeanSquaredError: 0.9584 - loss: 0.9214 - val_RootMeanSquaredError: 3.9133 -
val_loss: 15.3142
Epoch 11/20
363/363              0s 989us/step -
RootMeanSquaredError: 0.9036 - loss: 0.8168 - val_RootMeanSquaredError: 3.2180 -
val_loss: 10.3554
Epoch 12/20
363/363              0s 1ms/step -
RootMeanSquaredError: 0.8940 - loss: 0.7995 - val_RootMeanSquaredError: 2.6714 -
val_loss: 7.1362
Epoch 13/20
363/363              0s 1ms/step -
RootMeanSquaredError: 0.9756 - loss: 0.9531 - val_RootMeanSquaredError: 2.2724 -
val_loss: 5.1636
Epoch 14/20
363/363              0s 988us/step -
RootMeanSquaredError: 0.9971 - loss: 0.9992 - val_RootMeanSquaredError: 2.0833 -
val_loss: 4.3401
Epoch 15/20
363/363              0s 1ms/step -
RootMeanSquaredError: 0.9919 - loss: 0.9905 - val_RootMeanSquaredError: 2.2558 -
val_loss: 5.0888
Epoch 16/20
363/363              0s 977us/step -
RootMeanSquaredError: 1.0314 - loss: 1.0712 - val_RootMeanSquaredError: 1.9443 -
```

```
val_loss: 3.7805
Epoch 17/20
363/363              0s 945us/step -
RootMeanSquaredError: 0.9496 - loss: 0.9035 - val_RootMeanSquaredError: 2.4768 -
val_loss: 6.1343
Epoch 18/20
363/363              0s 979us/step -
RootMeanSquaredError: 1.3026 - loss: 1.7384 - val_RootMeanSquaredError: 2.3679 -
val_loss: 5.6067
Epoch 19/20
363/363              0s 965us/step -
RootMeanSquaredError: 1.4613 - loss: 2.2241 - val_RootMeanSquaredError: 2.0035 -
val_loss: 4.0142
Epoch 20/20
363/363              0s 1ms/step -
RootMeanSquaredError: 1.0357 - loss: 1.0998 - val_RootMeanSquaredError: 2.1053 -
val_loss: 4.4321
```

Evaluate the model's performance on testing data

[42]:
```
mse_test, rmse_test = model.evaluate(X_test, y_test)
```

```
162/162              0s 845us/step -
RootMeanSquaredError: 1.3568 - loss: 1.8413
```

[43]:
```
print(mse_test, rmse_test)
```

```
1.860201120376587 1.3638919591903687
```

0.0.3 Type 3 Model : Subset features different paths

- Send a subset of the features through the wide path and a different subset (possibly overlapping) through the deep path

Defining layers of the model

[52]:
```
input_wide = tf.keras.layers.Input(shape=[5]) # features 0 to 4
input_deep = tf.keras.layers.Input(shape=[6]) # features 2 to 7

norm_layer_wide = tf.keras.layers.Normalization()
norm_layer_deep = tf.keras.layers.Normalization()

norm_wide = norm_layer_wide(input_wide)
norm_deep = norm_layer_deep(input_deep)

hidden1 = tf.keras.layers.Dense(30, activation='relu')(norm_deep)
hidden2 = tf.keras.layers.Dense(30, activation='relu')(hidden1)

concat = tf.keras.layers.concatenate([norm_wide, hidden2])
```

```
output = tf.keras.layers.Dense(1)(concat)

model = tf.keras.Model(inputs=[input_wide, input_deep], outputs=[output])
```

Defining the optimizer

[53]:
```
optimizer2 = tf.keras.optimizers.Adam(learning_rate=1e-3)
```

Compiling the model with training metrics

[54]:
```
model.compile(loss="mse", optimizer=optimizer2,␣
↪metrics=["RootMeanSquaredError"])
```

Seperating the datasets

[58]:
```
X_train_wide, X_train_deep = X_train[:, :5], X_train[:, 2:]
X_valid_wide, X_valid_deep = X_valid[:, :5], X_valid[:, 2:]
X_test_wide, X_test_deep = X_test[:, :5], X_test[:, 2:]
X_new_wide, X_new_deep = X_test_wide[:3], X_test_deep[:3]
```

Training the model

[56]:
```
norm_layer_wide.adapt(X_train_wide)
norm_layer_deep.adapt(X_train_deep)

history = model.fit((X_train_wide, X_train_deep), y_train, epochs=20,␣
↪validation_data=((X_valid_wide, X_valid_deep), y_valid))
```

```
Epoch 1/20
363/363              1s 1ms/step -
RootMeanSquaredError: 1.4704 - loss: 2.2448 - val_RootMeanSquaredError: 1.4442 -
val_loss: 2.0857
Epoch 2/20
363/363              0s 991us/step -
RootMeanSquaredError: 0.7428 - loss: 0.5526 - val_RootMeanSquaredError: 0.8360 -
val_loss: 0.6988
Epoch 3/20
363/363              0s 905us/step -
RootMeanSquaredError: 0.6676 - loss: 0.4461 - val_RootMeanSquaredError: 0.6239 -
val_loss: 0.3893
Epoch 4/20
363/363              0s 958us/step -
RootMeanSquaredError: 0.6452 - loss: 0.4165 - val_RootMeanSquaredError: 0.6018 -
val_loss: 0.3621
Epoch 5/20
363/363              0s 1ms/step -
RootMeanSquaredError: 0.6309 - loss: 0.3983 - val_RootMeanSquaredError: 0.5956 -
val_loss: 0.3547
```

```
Epoch 6/20
363/363              0s 1ms/step -
RootMeanSquaredError: 0.6214 - loss: 0.3864 - val_RootMeanSquaredError: 0.6294 -
val_loss: 0.3961
Epoch 7/20
363/363              0s 999us/step -
RootMeanSquaredError: 0.6138 - loss: 0.3770 - val_RootMeanSquaredError: 0.5915 -
val_loss: 0.3499
Epoch 8/20
363/363              0s 1ms/step -
RootMeanSquaredError: 0.6070 - loss: 0.3687 - val_RootMeanSquaredError: 0.6080 -
val_loss: 0.3697
Epoch 9/20
363/363              0s 996us/step -
RootMeanSquaredError: 0.6021 - loss: 0.3628 - val_RootMeanSquaredError: 0.5725 -
val_loss: 0.3278
Epoch 10/20
363/363              0s 1ms/step -
RootMeanSquaredError: 0.5970 - loss: 0.3566 - val_RootMeanSquaredError: 0.6704 -
val_loss: 0.4494
Epoch 11/20
363/363              0s 1ms/step -
RootMeanSquaredError: 0.5933 - loss: 0.3521 - val_RootMeanSquaredError: 0.6333 -
val_loss: 0.4011
Epoch 12/20
363/363              0s 1ms/step -
RootMeanSquaredError: 0.5908 - loss: 0.3491 - val_RootMeanSquaredError: 0.8573 -
val_loss: 0.7349
Epoch 13/20
363/363              0s 955us/step -
RootMeanSquaredError: 0.5876 - loss: 0.3454 - val_RootMeanSquaredError: 0.7427 -
val_loss: 0.5515
Epoch 14/20
363/363              0s 915us/step -
RootMeanSquaredError: 0.5855 - loss: 0.3429 - val_RootMeanSquaredError: 1.1110 -
val_loss: 1.2343
Epoch 15/20
363/363              0s 1ms/step -
RootMeanSquaredError: 0.5862 - loss: 0.3438 - val_RootMeanSquaredError: 0.8765 -
val_loss: 0.7682
Epoch 16/20
363/363              0s 1ms/step -
RootMeanSquaredError: 0.5816 - loss: 0.3384 - val_RootMeanSquaredError: 1.2828 -
val_loss: 1.6457
Epoch 17/20
363/363              0s 1ms/step -
RootMeanSquaredError: 0.5817 - loss: 0.3385 - val_RootMeanSquaredError: 1.1154 -
val_loss: 1.2440
```

```
Epoch 18/20
363/363              0s 1ms/step -
RootMeanSquaredError: 0.5795 - loss: 0.3359 - val_RootMeanSquaredError: 1.0683 -
val_loss: 1.1412
Epoch 19/20
363/363              0s 1ms/step -
RootMeanSquaredError: 0.5771 - loss: 0.3332 - val_RootMeanSquaredError: 0.7588 -
val_loss: 0.5758
Epoch 20/20
363/363              0s 1ms/step -
RootMeanSquaredError: 0.5730 - loss: 0.3285 - val_RootMeanSquaredError: 0.7698 -
val_loss: 0.5926
```

Evaluating the model against test dataset

[59]:
```
mse_test = model.evaluate((X_test_wide, X_test_deep), y_test)
y_pred = model.predict((X_new_wide, X_new_deep))
```

```
162/162              0s 574us/step -
RootMeanSquaredError: 0.5748 - loss: 0.3305
1/1              0s 44ms/step
```

0.0.4 Type 4 Model : Seperate Outputs

- Extra output is quite easy: connect it to the appropriate layer and add it to the model's list of outputs

Defining the 2 model outputs in output layer

[71]:
```
output = tf.keras.layers.Dense(1)(concat)
aux_output = tf.keras.layers.Dense(1)(hidden2)

model = tf.keras.Model(inputs=[input_wide, input_deep], outputs=[output,␣
 ↪aux_output])
```

Defining the optimizer

[72]:
```
optimizer3 = tf.keras.optimizers.Adam(learning_rate=1e-3)
```

Define the model compilation with training parameters

[73]:
```
model.compile(loss=("mse", "mse"), loss_weights=(0.9, 0.1), optimizer =␣
 ↪optimizer3, metrics=["RootMeanSquaredError", "RootMeanSquaredError"])
```

Adapting the training data with normalization layer

[74]:
```
norm_layer_wide.adapt(X_train_wide)
norm_layer_deep.adapt(X_train_deep)
```

Fitting the model

[75]:
```
history = model.fit((X_train_wide, X_train_deep), (y_train, y_train),␣
    ↪epochs=20, validation_data=((X_valid_wide, X_valid_deep), (y_valid,␣
    ↪y_valid)))
```

```
Epoch 1/20
363/363                2s 2ms/step -
dense_29_RootMeanSquaredError: 1.4116 - dense_29_loss: 1.8856 -
dense_30_RootMeanSquaredError: 1.8370 - dense_30_loss: 0.3484 - loss: 2.2340 -
val_dense_29_RootMeanSquaredError: 0.7331 - val_dense_29_loss: 0.4836 -
val_dense_30_RootMeanSquaredError: 1.0084 - val_dense_30_loss: 0.1017 -
val_loss: 0.5854
Epoch 2/20
363/363                0s 1ms/step -
dense_29_RootMeanSquaredError: 0.7011 - dense_29_loss: 0.4427 -
dense_30_RootMeanSquaredError: 0.8633 - dense_30_loss: 0.0746 - loss: 0.5173 -
val_dense_29_RootMeanSquaredError: 0.6377 - val_dense_29_loss: 0.3659 -
val_dense_30_RootMeanSquaredError: 0.8684 - val_dense_30_loss: 0.0754 -
val_loss: 0.4414
Epoch 3/20
363/363                1s 1ms/step -
dense_29_RootMeanSquaredError: 0.6355 - dense_29_loss: 0.3636 -
dense_30_RootMeanSquaredError: 0.7202 - dense_30_loss: 0.0519 - loss: 0.4155 -
val_dense_29_RootMeanSquaredError: 0.6087 - val_dense_29_loss: 0.3334 -
val_dense_30_RootMeanSquaredError: 0.7227 - val_dense_30_loss: 0.0522 -
val_loss: 0.3857
Epoch 4/20
363/363                1s 1ms/step -
dense_29_RootMeanSquaredError: 0.6065 - dense_29_loss: 0.3311 -
dense_30_RootMeanSquaredError: 0.6699 - dense_30_loss: 0.0449 - loss: 0.3760 -
val_dense_29_RootMeanSquaredError: 0.5841 - val_dense_29_loss: 0.3071 -
val_dense_30_RootMeanSquaredError: 0.6526 - val_dense_30_loss: 0.0426 -
val_loss: 0.3497
Epoch 5/20
363/363                1s 1ms/step -
dense_29_RootMeanSquaredError: 0.5936 - dense_29_loss: 0.3172 -
dense_30_RootMeanSquaredError: 0.6485 - dense_30_loss: 0.0421 - loss: 0.3592 -
val_dense_29_RootMeanSquaredError: 0.5906 - val_dense_29_loss: 0.3139 -
val_dense_30_RootMeanSquaredError: 0.6445 - val_dense_30_loss: 0.0415 -
val_loss: 0.3555
Epoch 6/20
363/363                1s 1ms/step -
dense_29_RootMeanSquaredError: 0.5862 - dense_29_loss: 0.3093 -
dense_30_RootMeanSquaredError: 0.6372 - dense_30_loss: 0.0406 - loss: 0.3499 -
val_dense_29_RootMeanSquaredError: 0.6883 - val_dense_29_loss: 0.4263 -
val_dense_30_RootMeanSquaredError: 0.8057 - val_dense_30_loss: 0.0649 -
val_loss: 0.4913
```

```
Epoch 7/20
363/363             1s 1ms/step -
dense_29_RootMeanSquaredError: 0.5817 - dense_29_loss: 0.3046 -
dense_30_RootMeanSquaredError: 0.6302 - dense_30_loss: 0.0397 - loss: 0.3443 -
val_dense_29_RootMeanSquaredError: 0.6775 - val_dense_29_loss: 0.4130 -
val_dense_30_RootMeanSquaredError: 0.6516 - val_dense_30_loss: 0.0424 -
val_loss: 0.4556
Epoch 8/20
363/363             0s 1ms/step -
dense_29_RootMeanSquaredError: 0.5791 - dense_29_loss: 0.3018 -
dense_30_RootMeanSquaredError: 0.6240 - dense_30_loss: 0.0390 - loss: 0.3408 -
val_dense_29_RootMeanSquaredError: 0.8057 - val_dense_29_loss: 0.5841 -
val_dense_30_RootMeanSquaredError: 0.8235 - val_dense_30_loss: 0.0678 -
val_loss: 0.6521
Epoch 9/20
363/363             1s 1ms/step -
dense_29_RootMeanSquaredError: 0.5762 - dense_29_loss: 0.2989 -
dense_30_RootMeanSquaredError: 0.6193 - dense_30_loss: 0.0384 - loss: 0.3372 -
val_dense_29_RootMeanSquaredError: 0.6289 - val_dense_29_loss: 0.3559 -
val_dense_30_RootMeanSquaredError: 0.6241 - val_dense_30_loss: 0.0389 -
val_loss: 0.3949
Epoch 10/20
363/363             1s 2ms/step -
dense_29_RootMeanSquaredError: 0.5744 - dense_29_loss: 0.2970 -
dense_30_RootMeanSquaredError: 0.6154 - dense_30_loss: 0.0379 - loss: 0.3348 -
val_dense_29_RootMeanSquaredError: 0.7995 - val_dense_29_loss: 0.5751 -
val_dense_30_RootMeanSquaredError: 0.8740 - val_dense_30_loss: 0.0764 -
val_loss: 0.6517
Epoch 11/20
363/363             1s 1ms/step -
dense_29_RootMeanSquaredError: 0.5722 - dense_29_loss: 0.2947 -
dense_30_RootMeanSquaredError: 0.6129 - dense_30_loss: 0.0376 - loss: 0.3323 -
val_dense_29_RootMeanSquaredError: 0.5750 - val_dense_29_loss: 0.2975 -
val_dense_30_RootMeanSquaredError: 0.6114 - val_dense_30_loss: 0.0374 -
val_loss: 0.3350
Epoch 12/20
363/363             1s 1ms/step -
dense_29_RootMeanSquaredError: 0.5704 - dense_29_loss: 0.2929 -
dense_30_RootMeanSquaredError: 0.6089 - dense_30_loss: 0.0371 - loss: 0.3300 -
val_dense_29_RootMeanSquaredError: 0.5857 - val_dense_29_loss: 0.3087 -
val_dense_30_RootMeanSquaredError: 0.6805 - val_dense_30_loss: 0.0463 -
val_loss: 0.3551
Epoch 13/20
363/363             1s 1ms/step -
dense_29_RootMeanSquaredError: 0.5683 - dense_29_loss: 0.2908 -
dense_30_RootMeanSquaredError: 0.6068 - dense_30_loss: 0.0368 - loss: 0.3276 -
val_dense_29_RootMeanSquaredError: 0.5485 - val_dense_29_loss: 0.2707 -
val_dense_30_RootMeanSquaredError: 0.6166 - val_dense_30_loss: 0.0380 -
```

```
val_loss: 0.3088
Epoch 14/20
363/363             1s 1ms/step -
dense_29_RootMeanSquaredError: 0.5673 - dense_29_loss: 0.2898 -
dense_30_RootMeanSquaredError: 0.6040 - dense_30_loss: 0.0365 - loss: 0.3262 -
val_dense_29_RootMeanSquaredError: 0.5585 - val_dense_29_loss: 0.2807 -
val_dense_30_RootMeanSquaredError: 0.6493 - val_dense_30_loss: 0.0421 -
val_loss: 0.3229
Epoch 15/20
363/363             1s 1ms/step -
dense_29_RootMeanSquaredError: 0.5661 - dense_29_loss: 0.2885 -
dense_30_RootMeanSquaredError: 0.6023 - dense_30_loss: 0.0363 - loss: 0.3247 -
val_dense_29_RootMeanSquaredError: 0.5467 - val_dense_29_loss: 0.2690 -
val_dense_30_RootMeanSquaredError: 0.6175 - val_dense_30_loss: 0.0381 -
val_loss: 0.3072
Epoch 16/20
363/363             0s 1ms/step -
dense_29_RootMeanSquaredError: 0.5643 - dense_29_loss: 0.2867 -
dense_30_RootMeanSquaredError: 0.6007 - dense_30_loss: 0.0361 - loss: 0.3228 -
val_dense_29_RootMeanSquaredError: 0.5477 - val_dense_29_loss: 0.2699 -
val_dense_30_RootMeanSquaredError: 0.6214 - val_dense_30_loss: 0.0386 -
val_loss: 0.3086
Epoch 17/20
363/363             0s 1ms/step -
dense_29_RootMeanSquaredError: 0.5629 - dense_29_loss: 0.2852 -
dense_30_RootMeanSquaredError: 0.5984 - dense_30_loss: 0.0358 - loss: 0.3210 -
val_dense_29_RootMeanSquaredError: 0.5455 - val_dense_29_loss: 0.2678 -
val_dense_30_RootMeanSquaredError: 0.6209 - val_dense_30_loss: 0.0386 -
val_loss: 0.3064
Epoch 18/20
363/363             1s 2ms/step -
dense_29_RootMeanSquaredError: 0.5617 - dense_29_loss: 0.2840 -
dense_30_RootMeanSquaredError: 0.5972 - dense_30_loss: 0.0357 - loss: 0.3197 -
val_dense_29_RootMeanSquaredError: 0.5472 - val_dense_29_loss: 0.2694 -
val_dense_30_RootMeanSquaredError: 0.6014 - val_dense_30_loss: 0.0362 -
val_loss: 0.3056
Epoch 19/20
363/363             1s 1ms/step -
dense_29_RootMeanSquaredError: 0.5603 - dense_29_loss: 0.2826 -
dense_30_RootMeanSquaredError: 0.5961 - dense_30_loss: 0.0355 - loss: 0.3182 -
val_dense_29_RootMeanSquaredError: 0.5542 - val_dense_29_loss: 0.2764 -
val_dense_30_RootMeanSquaredError: 0.6508 - val_dense_30_loss: 0.0424 -
val_loss: 0.3188
Epoch 20/20
363/363             1s 1ms/step -
dense_29_RootMeanSquaredError: 0.5593 - dense_29_loss: 0.2816 -
dense_30_RootMeanSquaredError: 0.5947 - dense_30_loss: 0.0354 - loss: 0.3170 -
val_dense_29_RootMeanSquaredError: 0.6114 - val_dense_29_loss: 0.3363 -
```

```
val_dense_30_RootMeanSquaredError: 0.5949 - val_dense_30_loss: 0.0354 -
val_loss: 0.3718
```

Evaluating the model performance against testing dataset

[76]:
```python
eval_results = model.evaluate((X_test_wide, X_test_deep), (y_test, y_test))

weighted_sum_of_losses, main_loss, aux_loss, main_rmse, aux_rmse = eval_results
```

```
162/162              0s 935us/step -
dense_29_RootMeanSquaredError: 0.5648 - dense_29_loss: 0.2871 -
dense_30_RootMeanSquaredError: 0.6071 - dense_30_loss: 0.0369 - loss: 0.3240
```

[77]:
```python
print(weighted_sum_of_losses, main_loss, aux_loss, main_rmse, aux_rmse)
```

```
0.32088109850883484 0.28391894698143005 0.036691226065158844 0.5619576573371887
0.6055104732513428
```

Predicting the results using model

[78]:
```python
y_pred_main, y_pred_aux = model.predict((X_new_wide, X_new_deep))
```

```
1/1              0s 65ms/step
```

[79]:
```python
y_pred_tuple = model.predict((X_new_wide, X_new_deep))

y_pred = dict(zip(model.output_names, y_pred_tuple))
```

```
1/1              0s 21ms/step
```

[80]:
```python
y_pred
```

[80]:
```
{'dense_29': array([[0.49514577],
        [1.2503316 ],
        [3.6080847 ]], dtype=float32),
 'dense_30': array([[0.4965441],
        [1.1439607],
        [3.5039604]], dtype=float32)}
```