towardsdatascience.com

Regression-based neural networks: Predicting Average Daily Rates for Hotels

Michael Grogan

16–20 minutes

When it comes to hotel bookings, average daily rate (ADR) is a particularly important metric. This reflects the average rate per day that a particular customer pays throughout their stay.

Keras is an API used for running high-level neural networks — the API is now included as the default one under TensorFlow 2.0, which was developed by Google.

The main competitor to Keras at this point in time is PyTorch, developed by Facebook. While PyTorch has a somewhat higher level of community support, it is a particularly verbose language and I personally prefer Keras for greater simplicity and ease of use in building and deploying models.

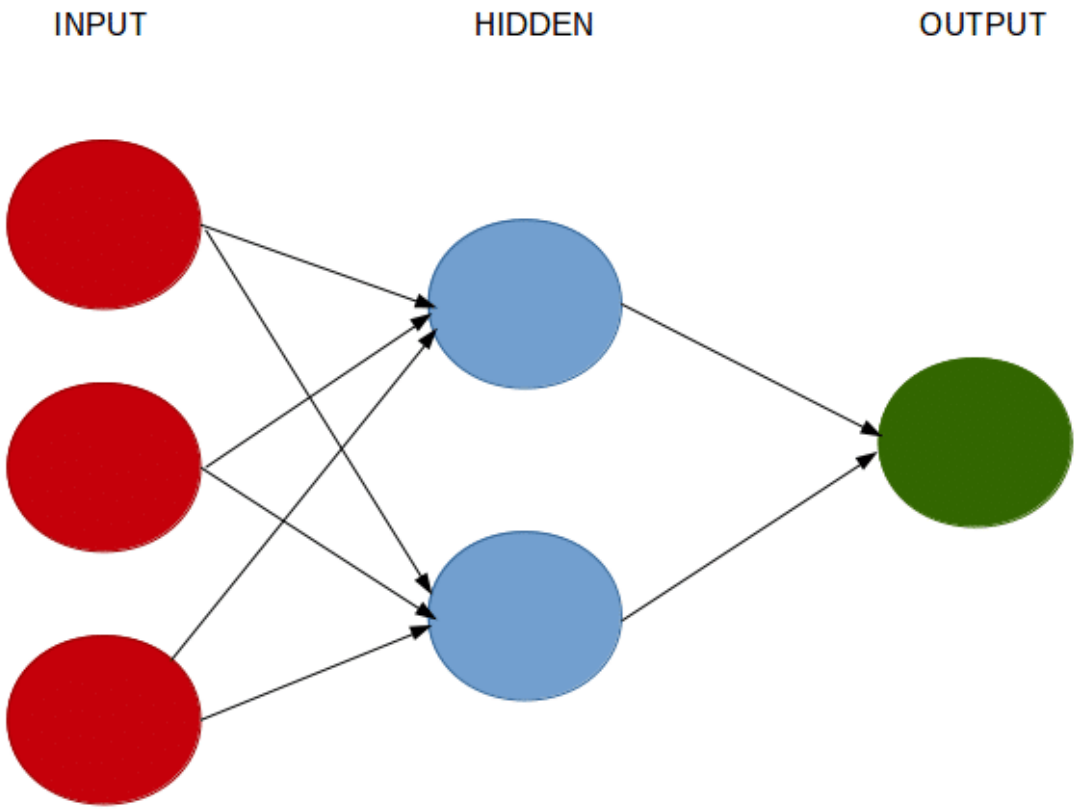
In this particular example, a neural network is built in Keras to solve a regression problem, i.e. one where our dependent variable (y) is in interval format and we are trying to predict the quantity of y with as much accuracy as possible.

What Is A Neural Network?

A neural network is a computational system that creates predictions based on existing data. Let us train and test a neural network using the neuralnet library in R.

A neural network consists of:

- Input layers: Layers that take inputs based on existing data
- Hidden layers: Layers that use backpropagation to optimise the weights of the input variables in order to improve the predictive power of the model
- Output layers: Output of predictions based on the data from the input and hidden layers



Source: Created by Author

Background

This study focuses on hotel booking analysis. When it comes to hotel bookings, **average daily rate (ADR)** is a particularly important metric. This reflects the average rate per day that a particular customer pays throughout their stay.

Gauging ADR allows hotels to more accurately identify its most profitable customers and tailor its marketing strategies accordingly.

The original datasets are available from Antonio, Almedia and Nunes (2019), Hotel Booking Demand Datasets.

For this example, we use a linear activation function within the keras library to create a regression-based neural network. The purpose of this neural network is to predict an **ADR** value for each customer. The chosen features that form the input for this neural network are as follows:

- IsCanceled
- Country of origin
- Market segment
- Deposit type
- Customer type
- Required car parking spaces
- Arrival Date: Week Number

Firstly, the relevant libraries are imported. Note that you will need TensorFlow installed on your system to be able to execute the below code.

```
import math
import matplotlib.pyplot as plt
import numpy as np
from numpy.random import seed
seed(1)
import pandas as pd
import statsmodels.api as sm
import statsmodels.formula.api as smfimport tensorflow
tensorflow.random.set_seed(1)
from tensorflow.python.keras.layers import Dense
from tensorflow.keras.layers import Dropout
from tensorflow.python.keras.models import Sequential
from tensorflow.python.keras.wrappers.scikit_learn import KerasRegressorfrom sklearn.metrics import
mean_absolute_error
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
```

Data Preparation

ADR is set as the y variable in this instance, since this is the feature we are trying to predict. The variable is stored as a numpy array.

```
y1 = np.array(adr)
```

The numerical and categorical variables are distinguished. For instance, a categorical variable such as country of origin is defined as a category (in order to prevent the neural network from assigning an order to the codes, e.g. if 1 = Portugal and 2 = Germany, we do not want a situation whereby Germany is ranked “higher” than Portugal).

```
countrycat=train_df.Country.astype("category").cat.codes
countrycat=pd.Series(countrycat)
```

A numpy stack of the chosen features is created:

```
x1 =
np.column_stack((IsCanceled,countrycat,marketsegmentcat,deposittypecat,customertypecat,rcps,arrivaldateweekno))
x1 = sm.add_constant(x1, prepend=True)
```

As with any neural network, the data needs to be scaled for proper interpretation by the network, a process known as normalization. **MinMaxScaler** is used for this purpose.

However, this comes with a caveat. Scaling must be done **after** the data has been split into training, validation and test sets — with each being scaled separately.

A common mistake when configuring a neural network is to first normalize the data before splitting the data.

The reason this is erroneous is that the normalization technique will use data from the validation and test sets as a reference point when scaling the data as a whole. This will inadvertently influence the values of the training data, essentially resulting in data leakage from the validation and test sets.

Accordingly, the data is first split into training and validation data:

```
X_train, X_val, y_train, y_val = train_test_split(x1, y1)
```

The training and validation data is then scaled using **MinMaxScaler:**

```
y_train=np.reshape(y_train, (-1,1))
y_val=np.reshape(y_val, (-1,1))scaler_x = MinMaxScaler()
scaler_y = MinMaxScaler()print(scaler_x.fit(X_train))
xtrain_scale=scaler_x.transform(X_train)
print(scaler_x.fit(X_val))
xval_scale=scaler_x.transform(X_val)print(scaler_y.fit(y_train))
ytrain_scale=scaler_y.transform(y_train)
print(scaler_y.fit(y_val))
yval_scale=scaler_y.transform(y_val)
```

Neural Network Configuration

One of the most important considerations when training a neural network is choosing the number of neurons to include in the input and hidden layers. Given that the output layer is the result layer, this layer has 1 neuron present by default.

As explained in this article by Farhad Malik, the number of neurons in each layer is configured as follows:

- **Input layer:** The number of neurons in the input layer is calculated as follows:

  `Number of features in the training set + 1`

  In this case, as there were 7 features in the training set to begin with, **8** input neurons are defined accordingly.

- **Hidden layer:** One hidden layer is defined, as a single layer is suitable when working with most datasets. The number of neurons in the hidden layer is determined as follows:

  Training Data Samples/Factor * (Input Neurons + Output Neurons)

  A factor of 1 is set in this case, the purpose of the factor being to prevent overfitting. A factor can take a value between 1 and 10. With 8 neurons in the input layer, 1 neuron in the output layer and 24036 observations in the training set, the hidden layer is assigned 2,670 neurons.

- **Output layer:** As this is the result layer, the output layer takes a value of 1 by default.

```
model = Sequential()
model.add(Dense(8, input_dim=8, kernel_initializer='normal', activation='relu'))
model.add(Dense(2670, activation='relu'))
model.add(Dense(1, activation='linear'))
model.summary()
```

The mean_squared_error (mse) and mean_absolute_error (mae) are our loss functions — i.e. an estimate of how accurate the neural network is in predicting the test data. We can see that with the validation_split set to 0.2, 80% of the training data is used to train the model, while the remaining 20% is used for testing purposes.

```
model.compile(loss='mse', optimizer='adam', metrics=['mse','mae'])
history=model.fit(xtrain_scale, ytrain_scale, epochs=30, batch_size=150, verbose=1, validation_split=0.2)
predictions = model.predict(xval_scale)
```

The model is configured as follows:

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 8)                 72
_________________________________________________________________
dense_1 (Dense)              (None, 2670)              24030
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 2671
=================================================================
Total params: 26,773
Trainable params: 26,773
Non-trainable params: 0
```

Let's now fit our model.

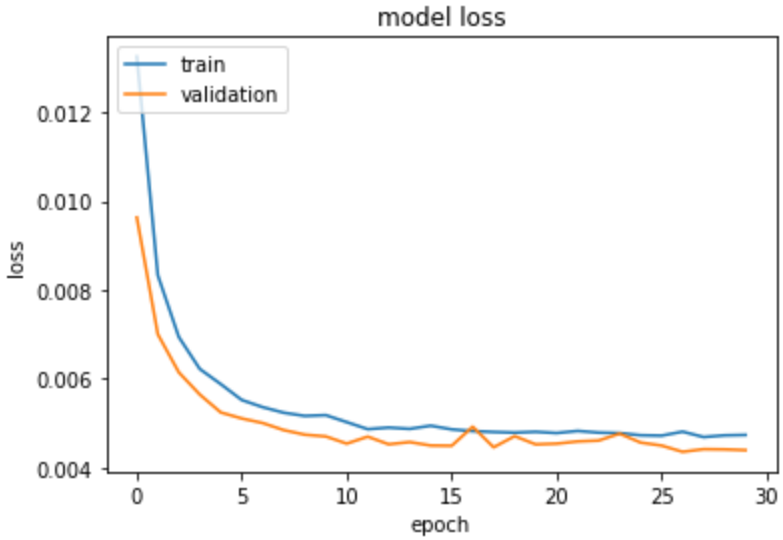
```
Epoch 1/30
161/161 [==============================] - 1s 4ms/step - loss: 0.0133 - mse: 0.0133 - mae: 0.0878 -
val_loss: 0.0096 - val_mse: 0.0096 - val_mae: 0.0714
Epoch 2/30
161/161 [==============================] - 0s 3ms/step - loss: 0.0083 - mse: 0.0083 - mae: 0.0672 -
val_loss: 0.0070 - val_mse: 0.0070 - val_mae: 0.0609
...
Epoch 28/30
161/161 [==============================] - 0s 2ms/step - loss: 0.0047 - mse: 0.0047 - mae: 0.0481 -
val_loss: 0.0044 - val_mse: 0.0044 - val_mae: 0.0474
Epoch 29/30
161/161 [==============================] - 0s 2ms/step - loss: 0.0047 - mse: 0.0047 - mae: 0.0482 -
val_loss: 0.0044 - val_mse: 0.0044 - val_mae: 0.0470
Epoch 30/30
161/161 [==============================] - 0s 2ms/step - loss: 0.0047 - mse: 0.0047 - mae: 0.0484 -
val_loss: 0.0044 - val_mse: 0.0044 - val_mae: 0.0467
```

Here, we can see that both the training loss and validation loss is being calculated, i.e. the deviation between the predicted y and actual y as measured by the mean squared error.

30 epochs have been specified for our model. This means that we are essentially training our model over 30 forward and backward passes, with the expectation that our loss will decrease with each epoch, meaning that our model is predicting the value of y more accurately as we continue to train the model.

Let's see what this looks like when the respective losses are plotted:

```
print(history.history.keys())
# "Loss"
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'validation'], loc='upper left')
plt.show()
```



Source: Jupyter Notebook Output

Both the training and validation loss decrease in an exponential fashion as the number of epochs is increased, suggesting that the model gains a high degree of accuracy as our epochs (or number of forward and backward passes) is increased.

The predictions generated using the features from the validation set can now be compared to the actual ADR values from that validation set.

The predictions are scaled back to the original scale:

```
predictions = scaler_y.inverse_transform(predictions)
predictions
```

```
In [17]: mean_absolute_error(y_val, predictions)
Out[17]: 26.738659201515834

In [18]: mean_squared_error(y_val, predictions)
         math.sqrt(mean_squared_error(y_val, predictions))
Out[18]: 38.94527336983452

In [19]: np.mean(y_val)
Out[19]: 94.62558861707438

In [20]: np.mean(predictions)
Out[20]: 88.01409
```

Source: Jupyter Notebook Output

With an MAE of 26 and an RMSE of 38 compared to the mean ADR value of 94 across the validation set, the model demonstrates significant predictive power. However, the true test is to generate predictions on previously unseen data and compare the results to the actual ADR values from the new dataset. The H2 dataset is used for this purpose.

Predictions — test set (H2)

Using the neural network model generated on the H1 dataset, the features from the H2 dataset are now fed into the network in order to predict ADR values for H2 and compare these predictions with the actual ADR values.

The calculated **mean absolute error** and **root mean squared error** are as follows:

```
>>> mean_absolute_error(btest, bpred)
44.41596076208908>>> mean_squared_error(btest, bpred)
>>> math.sqrt(mean_squared_error(btest, bpred))
57.899202401480025
```

The mean ADR across the H2 dataset was 105.30.

Epochs vs. Batch Size

A key tradeoff when constructing a neural network concerns that of the **number of epochs** used to train the model, and **batch size**.

- **Batch size:** Refers to the number of training examples per one forward/backward pass.
- **Epoch:** One forward and backward pass for all training examples.

This excellent summary on StackOverflow goes into further detail regarding the above definitions.

The key tradeoff faced when constructing a neural network is between the batch size and number of iterations.

For instance, the training data contains 24,036 samples and the batch size is 150. This means that 160 iterations are required to complete 1 epoch.

Therefore, one can either increase the batch size to have less iterations per epoch, or the batch size is reduced which means more iterations are required per epoch.

This means that all else being equal, the neural network either needs a higher batch size to train across a fixed number of epochs, or a lower batch size to train across a higher number of epochs.

150 epochs and batch_size = 50

Here is the model performance on the test set when the number of epochs are increased to 150 and the batch size is lowered to 50.

```
>>> mean_absolute_error(btest, bpred)
45.20461343967321>>> mean_squared_error(btest, bpred)
>>> math.sqrt(mean_squared_error(btest, bpred))
58.47641486637935
```

The MAE and RMSE were slightly lower when using 30 epochs and a batch size of 150 — suggesting that a smaller number of epochs with a larger batch size was superior in predicting ADR.

150 epochs and batch_size = 150

However, what if both the number of epochs and batch size is set to 150? Do the results improve any further?

```
>>> mean_absolute_error(btest, bpred)
45.030375561153335>>> mean_squared_error(btest, bpred)
>>> math.sqrt(mean_squared_error(btest, bpred))
58.43900275965334
```

When compared with a batch size of 150 over 30 epochs, the results are virtually identical, with the RMSE being slightly lower when 30 epochs are used.

In this regard, increasing both the batch size and number of epochs has not resulted in an improvement to the model performance on the test set.

Activation Function

When formulating a neural network, consideration must also be given as to the chosen **activation function**.

The purpose of an activation function in this instance is to induce non-linearity into the input and hidden layers, so as to produce more accurate results as generated by the output layer. When it comes to situations where we are dealing with a regression problem, i.e. the output variable is numerical and not categorical, the ReLU activation function (Rectified Linear Activation Function) is quite popular.

Specifically, this activation function solves what is called the vanishing gradient problem whereby the neural network would not be able to feed back important gradient information from the output layer back to the input layer. More information on the vanishing gradient problem can be found at this tutorial from Machine Learning Mastery.

ReLU was used in the examples above and showed the best accuracy across 30 epochs and a batch size of 150.

However, could there exist a more appropriate activation function for this specific problem?

As an example, the ELU activation function (which stands for Exponential Linear Unit) functions in much the same way that ReLU does, but the main difference is that ELU allows for negative inputs and can also produce negative outputs.

Technically, our dataset does not have negative inputs. However, many of the ADR values in the dataset are 0. After all, if a customer cancels their hotel booking, then the hotel cannot charge them (in the vast majority of cases).

As a result, there are a significant number of **0** entries for ADR, and in fact there is also one instance where a negative observation is recorded for this variable.

In that regard, the neural network is run for 30 epochs once again, and this time the ELU activation function is used in place of ReLU.

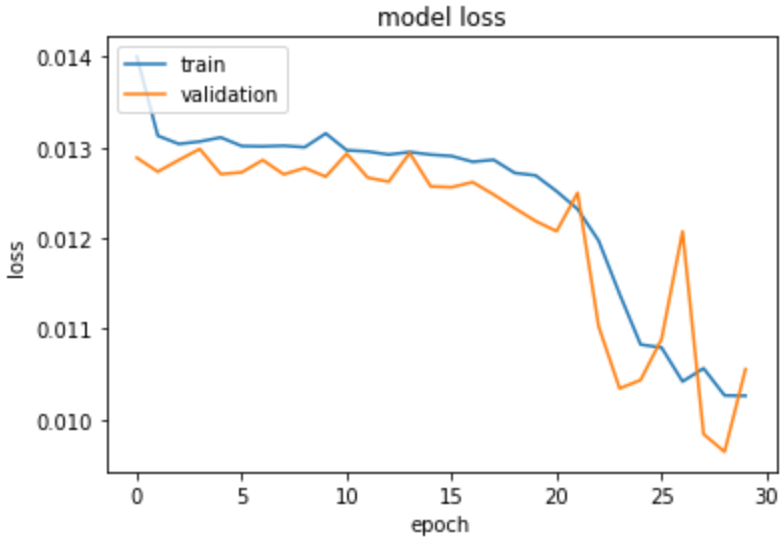
```
model = Sequential()
model.add(Dense(8, input_dim=8, kernel_initializer='normal', activation='elu'))
model.add(Dense(2670, activation='elu'))
```

```
model.add(Dense(1, activation='linear'))
model.summary()
```

Here is the model configuration:

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 8)                 72
_________________________________________________________________
dense_1 (Dense)              (None, 2670)              24030
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 2671
=================================================================
Total params: 26,773
Trainable params: 26,773
Non-trainable params: 0
```

Here is the training and validation loss.



Source: Jupyter Notebook Output

```
>>> mean_absolute_error(btest, bpred)
28.908454264679218>>> mean_squared_error(btest, bpred)
>>> math.sqrt(mean_squared_error(btest, bpred))
43.66170887622355
```

The mean absolute error and root mean squared error are lower when using ELU as opposed to ReLU. This indicates that changing the activation function has resulted in an improvement in accuracy.

Conclusion

This example has illustrated how to:

- Construct neural networks with Keras
- Scale data appropriately with MinMaxScaler
- Calculate training and test losses
- Make predictions using the neural network model
- Importance of choosing the correct activation function
- Considerations of the tradeoff between number of iterations and batch size

You can find the original article here, which includes a link to the GitHub repository containing the relevant notebooks and datasets pertaining to the above examples.

Disclaimer: This article is written on an "as is" basis and without warranty. It was written with the intention of providing an overview of data science concepts, and should not be interpreted as professional advice in any way.