Deepika Singh

Regression with Keras

Deepika Singh

Mar 20, 2019 · 14 Min read · 37,069 Views

Mar 20, 2019 · 14 Min read · 37,069 Views

Data Keras

Introduction

114

- Introduction
- Problem Statement
- Steps
- Evaluation of the Model Performance
- Conclusion
- Top ^

Introduction

Deep Learning is one of the hottest topics in data science today. This is not surprising given the tremendous amount of fascinating applications being developed using deep learning, such as self-driving cars, color restoration, natural language processing, automatic machine translation, image classification, and many more.

There are many deep learning libraries out there, but the most popular ones are TensorFlow, Keras, and PyTorch. Although TensorFlow and Pytorch are immensely popular, they are not easy to use and have a steep learning curve. So, for many practitioners, Keras is the preferred choice.

The Keras library is a high-level API for building deep learning models that has gained favor for its ease of use and simplicity facilitating fast development. Often, building a very complex deep learning network with Keras can be achieved with only a few lines of code.

In this guide, we will focus on how to use the Keras library to build regression models.

Regression with Keras

Regression is a type of supervised machine learning algorithm used to predict a continuous label. The goal is to produce a model that represents the 'best fit' to some observed data, according to an evaluation criterion.

The basic architecture of the deep learning neural network, which we will be following, consists of three main components.

1) Input Layer: This is where the training observations are fed. The number of predictor variables is also specified here through the neurons.

2) Hidden Layers: These are the intermediate layers between the input and output layers. The deep neural network learns about the relationships involved in data in this component.

3) Output Layer: This is the layer where the final output is extracted from what's happening in the previous two layers. In case of regression problems, the output later will have one neuron.

Problem Statement

Unemployment is a major socio-economic and political issue for any country and, hence, managing it is a chief task for any government. But to manage unemployment within an economy, it is imperative to predict it as well. This is what this guide will aim to achieve. The guide will be building a deep learning regression model using Keras to predict unemployment.

The data used in this project was produced from US economic time series data available from http://research.stlouisfed.org/fred2. The data contains 574 rows and 5 variables, as described below:

```
- psavert - personal savings rate.
- pce     - personal consumption expenditures, in billions of dollars.
- uempmed - median duration of unemployment, in weeks.
- pop     - total population, in thousands.
- unemploy- number of unemployed in thousands (dependent variable).
```

Evaluation Metrics

We will evaluate the performance of the model using Root Mean Squared Error (RMSE), a commonly used metric for regression problems. In simple terms, RMSE measures the average magnitude of the residuals or error. Mathematically, it is computed as the square root of the average of squared differences between predicted and actual values.

Steps

Following are the steps which are commonly followed while implementing Regression Models with Keras.

Step 1 - Loading the required libraries and modules.

Step 2 - Loading the data and performing basic data checks.

Step 3 - Creating arrays for the features and the response variable.

Step 4 - Creating the training and test datasets.

Step 5 - Define, compile, and fit the Keras regression model.

Step 6 - Predict on the test data and compute evaluation metrics.

The following sections will cover these steps.

Step 1 - Loading the Required Libraries and Modules

```python
# Import required libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import sklearn

# Import necessary modules
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
from math import sqrt

# Keras specific
import keras
from keras.models import Sequential
from keras.layers import Dense
```

Step 2 - Reading the Data and Performing Basic Data Checks

The first line of code reads in the data as pandas dataframe, while the second line of code prints the shape - 574 observations of 5 variables. The third line gives summary statistics of the numerical variables. We can see that all the variables have 574 as 'count' which is equal to the number of records in the dataset. That means we don't have missing values.

```python
df = pd.read_csv('regressionexample.csv')
print(df.shape)
df.describe()
```

```
(574, 5)
```

	pce	pop	psavert	uempmed	unemploy
count	574	574	574	574	574
mean	4,844	2,57,189	8	9	7,772
std	3,579	36,731	3	4	2,642
min	507	1,98,712	2	4	2,685
25%	1,582	2,24,896	6	6	6,284
50%	3,954	2,53,060	8	8	7,494
75%	7,667	2,90,291	11	9	8,691
max	12,162	3,20,887	17	25	15,352

Step 3 - Creating Arrays for the Features and the Response Variable

The first line of code creates an object of the target variable, while the second line of code gives us the list of all the features, excluding the target variable 'unemploy'.

The third line normalizes the predictors. This is important because the units of the variables differ significantly and may influence the modeling process. To prevent this, we will do normalization via scaling of the predictors between 0 and 1.

The fourth line displays the summary of the normalized data. We can see that all the independent variables have now been scaled between 0 and 1. The target variable remains unchanged.

python
```python
target_column = ['unemploy']
predictors = list(set(list(df.columns))-set(target_column))
df[predictors] = df[predictors]/df[predictors].max()
df.describe()
```

	pce	pop	psavert	uempmed	unemploy
count	574	574	574	574	574
mean	0.40	0.80	0.47	0.34	7,772
std	0.29	0.11	0.18	0.16	2,642
min	0.04	0.62	0.11	0.16	2,685
25%	0.13	0.70	0.32	0.24	6,284
50%	0.33	0.79	0.45	0.30	7,494
75%	0.63	0.90	0.62	0.36	8,691
max	1	1	1	1	15,352

Step 4 - Creating the Training and Test Datasets

The first couple of lines creates arrays of independent (X) and dependent (y) variables, respectively. The third line splits the data into training and test dataset, while the fourth line prints the shape of the training set (401 observations of 4 variables) and test set (173 observations of 4 variables).

python
```python
X = df[predictors].values
y = df[target_column].values

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.30, r
print(X_train.shape); print(X_test.shape)
```

```
(401, 4)
(173, 4)
```

Step 5 - Building the Deep Learning Regression Model

We will build a regression model using deep learning in Keras. To begin with, we will define the model. The first line of code below calls for the Sequential constructor. Note that we would be using the Sequential model because our network consists of a linear stack of layers. The second line of code represents the first layer which specifies the activation function and the number of input dimensions, which in our case is 4 predictors. Then we repeat the same process in the third and fourth line of codes for the hidden layers, this time without the input_dim parameter. The last line of code creates the output layer with one node that is supposed to output the number of unemployed in thousands.

The activation function used in the hidden layers is a rectified linear unit, or ReLU. It is the most widely used activation function because of its advantages of being nonlinear, as well as the ability to not activate all the neurons at the same time. In simple terms, this means that at a time, only a few neurons are activated, making the network sparse and very efficient.

```python
# Define model
model = Sequential()
model.add(Dense(500, input_dim=4, activation= "relu"))
model.add(Dense(100, activation= "relu"))
model.add(Dense(50, activation= "relu"))
model.add(Dense(1))
#model.summary() #Print model Summary
```

The next step is to define an optimizer and the loss measure for training. The mean squared error is our loss measure and the "adam" optimizer is our minimization algorithm. The main advantage of the "adam" optimizer is that we don't need to specify the learning rate as is the case with gradient descent; thereby saving us the task of optimizing the learning rate for our model. We achieve this task with the first line of the code below.

The second line of code fits the model on the training dataset. We also provide the argument, epochs, which represents the number of training iterations. We have taken 20 epochs.

```python
model.compile(loss= "mean_squared_error" , optimizer="adam", metrics=["mean
model.fit(X_train, y_train, epochs=20)
```

```
Epoch 1/20
401/401 [==============================] - 0s 1ms/step - loss: 68136318.3441
Epoch 2/20
401/401 [==============================] - 0s 133us/step - loss: 68101432.06
Epoch 3/20
401/401 [==============================] - 0s 125us/step - loss: 67985495.10
Epoch 4/20
401/401 [==============================] - 0s 134us/step - loss: 67665023.05
Epoch 5/20
401/401 [==============================] - 0s 127us/step - loss: 66899397.2
Epoch 6/20
Epoch 7/20
401/401 [==============================] - 0s 120us/step - loss: 62432633.3
Epoch 8/20
401/401 [==============================] - 0s 128us/step - loss: 57537882.0
Epoch 9/20
401/401 [==============================] - 0s 150us/step - loss: 50086165.6
Epoch 10/20
401/401 [==============================] - 0s 119us/step - loss: 39984370.9
Epoch 11/20
401/401 [==============================] - 0s 97us/step - loss: 28126145.28
Epoch 12/20
401/401 [==============================] - 0s 110us/step - loss: 16095036.0
Epoch 13/20
401/401 [==============================] - 0s 126us/step - loss: 7629222.01
Epoch 14/20
401/401 [==============================] - 0s 107us/step - loss: 4147607.16
Epoch 15/20
401/401 [==============================] - 0s 107us/step - loss: 3668975.75
Epoch 16/20
401/401 [==============================] - 0s 111us/step - loss: 3646548.08
Epoch 17/20
401/401 [==============================] - 0s 126us/step - loss: 3563563.13
Epoch 18/20
401/401 [==============================] - 0s 117us/step - loss: 3533091.93
Epoch 19/20
401/401 [==============================] - 0s 123us/step - loss: 3496560.11
Epoch 20/20
401/401 [==============================] - 0s 132us/step - loss: 3467280.01
```

Step 6 - Predict on the Test Data and Compute Evaluation Metrics

The first line of code predicts on the train data, while the second line prints the RMSE value on the train data. The same is repeated in the third and fourth lines of code which predicts and prints the RMSE value on test data.

```python
pred_train= model.predict(X_train)
print(np.sqrt(mean_squared_error(y_train,pred_train)))

pred= model.predict(X_test)
print(np.sqrt(mean_squared_error(y_test,pred)))
```

```
1856.4850642445354
1825.5904063232729
```

Evaluation of the Model Performance

The output above shows that the RMSE, which is our evaluation metric, was 1856 thousand for train data and 1825 thousand for test data. Ideally, the lower the RMSE value, the better the model performance. However, in contrast to accuracy, it is not straightforward to interpret RMSE as we would have to look at the unit which in our case is in thousands.

Conclusion

In this guide, we have built Regression models using the deep learning framework, Keras. The guide used the US economics time series data and built a deep learning regression model to predict the number of unemployed population in thousands.

Our model is achieving a stable performance with not much variance in the train and test set RMSE. The most ideal result would be an RMSE value of zero, but that's almost impossible in real economic datasets. Also, since the unit of the target variable is in thousands, that also affects the RMSE value.

There are other iterations such as changing the number of neurons, adding more hidden layers, or increasing the number of epochs, which can be tried out to see the impact on model performance.

This regression problem could also be modeled using other algorithms such as Decision Tree, Random Forest, Gradient Boosting or Support Vector Machines. However, that is not in the scope of this guide which is aimed at enabling individuals to solve Regression problems using deep learning library Keras.

114