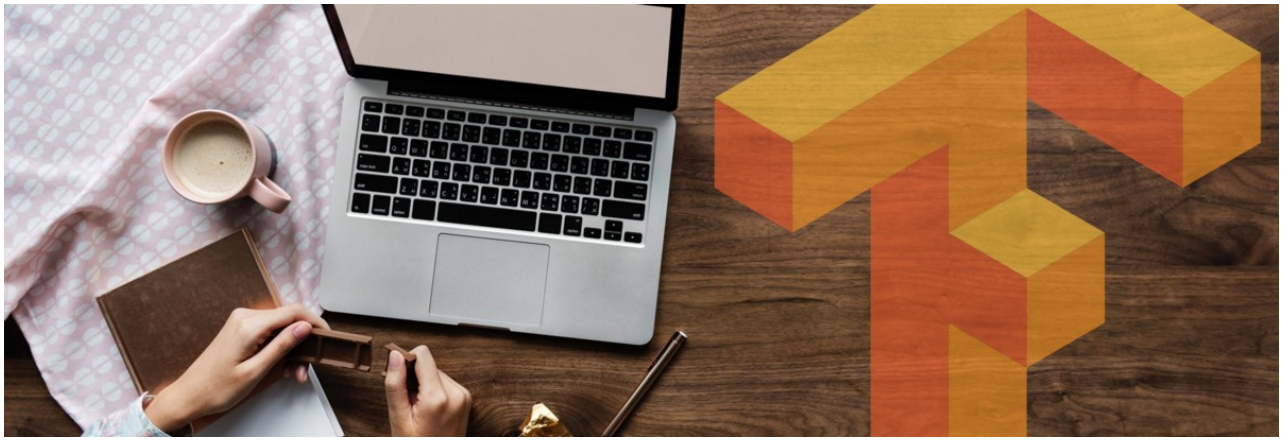
# Multivariate Regression Using Deep Neural Networks in Tensorflow

medium.com/themlblog/multivariate-regression-using-deep-neural-networks-in-tensorflow-f94f42a148b3

July 30, 2018



In this post, we will be discussing a multivariate regression problem and solving it using Google's deep learning library tensorflow.

Don't forget to read the previous post on Getting Started With Tensorflow!

Show your support by

Tensorflow was originally developed to construct the more complex neural networks used in tasks such as time-series analysis , word-embedding , image processing and reinforcement learning. We will be covering some of the mentioned topics in future posts. So, the developers created a estimator API for tensorflow which can seamlessly handle simpler tasks such as Regression and Classification. Today, we will be using the tensorflow estimator API to solve a house value prediction problem.

I know that the house value prediction problem has now become a cliche for regression tasks, but generally this task is associated with one feature, whereas we will be addressing a multivariate task with six features. The dataset can be downloaded here. So, lets get started!


We start by fetching the dataset using pandas read_csv() method. After reading in the dataset, if you want to take a look at the columns and what some of the values look like, you can use the describe() method, which returns the first few values of each column and the column heading. Also, I am not explaining the necessary imports.

The next step is to spit the dataset into a training and test set. We do this using scikit's train_test_split() method. The first two arguments for this method are two dictionaries:

1. x_data is a dictionary with six column values which represent the six features based on which the prediction will be done.
2. y_val contains just one column value which is the label or the target value which we will be predicting.

The next step is to scale the feature set for both training and test set. We will use the MinMaxScaler() method from scikit's preprocessing library. First, we need to create an instance of the MinMaxScler method and then we need to use the fit() and transform() methods. A little trick: you can use the fit_transform() method to perform both the operations at one go.

Now our dataset is ready to be fed into the estimator API. The API has three primary methods:

1. train
2. evaluate
3. predict

But, first we need to create the estimator model. The estimator model does not accept numpy arrays as input, it only takes a specific tensor value which has to be created using the tf.feature_column.numeric_column() method which is then fed into an empty

dictionary. So, we loop over the columns containing the feature columns. The expression df.columns[:-1] slices the dataset to exclude the last column as it is our target variable.

Next, we need to create an input function which has to be passed to the estimator model as an argument. To do this, we use the tf.estimator.inputs.pandas_input_fn() method and pass in the training set and batch size.

Now, we can create the estimator model. As, we will be using a deep neural network to perform the regression task, we use the DNNRegressor() method of the estimator API. There are several models available in the estimator API which can be tested out in different scenarios. Hint: Try out the DNNLinearCombinedRegressor() method which combines a linear and a DNN regressor. We need to pass in the number of neurons for each hidden unit as a dictionary to the above method. You can play around with the number of layers and the number of neurons to improve the model. Also, the feature column dictionary which we created in the earlier step by looping over the sliced column, needs to be passed in as an argument.

The next step is to train our model. The syntax to facilitate training is model.train() and takes the input function created in the previous step and the number of steps or epochs.

Finally, our model is trained and is ready to predict house prices!

But, before actually predict values from the test set we use the evaluate method to perform an assessment of how well our model is performing. We use model.evaluate() method which has the same arguments as the train method. The output comprises of the average loss and the overall loss after the total number of steps mentioned in the argument.

Next, we need to create a new input function called pred_input_func for the predict method. We do this by using the same tf.estimator.inputs.pandas_input_fn() method and we pass in the test set: x_eval and y_eval. The predict method also has the same arguments as the train and evaluate methods.

Notice that all the three primary methods have the same arguments. Makes it easy to remember!

The predict method returns a tensor object which has to be stored in a list in order to properly visualize the results for each value of the test set.

But in order to properly assess the result we need to apply the evaluate method on the pred_input_func input function.

We have all heard the term overfitting. One easy way to check whether our model is overfitting or not is to compare the overall loss from the evaluations on the training and test input functions. If the training loss is low and the test loss high, then our model is

overfitting and the epochs needs to be reduced. If the opposite is true, then our model is underfitting and we need to either increase the epochs or introduce more training data.

The only downside of multivariate regression tasks is that we cannot visualize or plot our results as the number of dimensions exceeds three. Although, if your task only has two parameters, you can use the following library to render a 3D model of your results: `mpl_toolkits.mplot3d`

The complete code can be found by clicking on the banner below. Stay tuned and subscribe to our newsletter for an awesome experience and never missing an update.

That's all, till next time!!