Introduction to Tensorflow Estimators

Tahsin Mayeesha Follow
Mar 16, 2018 · 22 min read

Tensorflow is an open source numerical computing library for implementing production-ready machine learning models as well as experimenting with novel architectures released by Google. It was originally developed by Google Brain team. Since it's release it has been steadily taking over other deep learning libraries like Theano or Caffe.

Several high level user-friendly deep learning libraries like Keras, TfLearn have been built on top of tensorflow already. It's flexible architecture allows users to create and deploy machine learning and deep learning models in CPU, GPU, distributed machines and even mobile devices. It also supports interfaces for many programming languages including Python and C++. This tutorial will use Python though.

You can use Tensorflow Playground to see an implementation of a neural network in playground for practical experience right now if you like. Google is using Tensorflow for search ranking, computer vision[Inception model], speech recognition, Youtube recommendations, machine translation for Google translate and in many other areas.

In this article , we'll explore tensorflow and work on a regression problem to predict Airbnb rental listing prices from Boston Airbnb Open Data. We'll learn about basic concepts of tensorflow like Tensors and computational graph, learn how to execute simple programs and implement a linear regression model from scratch first. Then we will learn how to use the high level estimator API for quickly building and testing models on the Airbnb data.

Airbnb is an online marketplace that helps peope to lease or rent short-term lodging including vacation and apartment rentals, homestays, hotels.The data is publicly

released in Kaggle. This dataset is part of Airbnb Inside, and the original source can be found here.

Visualization from popular 'Deepdrem' project,implemented with tensorflow, Image credit- Inceptionism, going deeper into neural networks, Google research blog

Installation

Tensorflow can be installed by pip using `pip install –upgrade tensorflow` command in terminal. We are using tensorflow 1.4(Dec 2017 latest release) for this tutorial. We don't have to get the GPU version of tensorflow for this tutorial because the dataset we'll be using is small, so the CPU version is adequate.

Note that in Windows tensorflow only supports 64-bit python 3.5.x and it does not support python 3.6 or 2.7. I'm using Anaconda(with python 3.5) because it comes with many important data science libraries and jupyter notebook.

To learn how to get tensorflow for your specific OS here are the official installation guides.

- Windows
- Linux
- Mac OS X

The guides contain instructions on how to install tensorflow using `pip`, `virtualenv`, Docker and from source. For other information see the installation home page.

Prerequisites

Since it's an introductory article, this tutorial does not expect previous experience with tensorflow. However it does expect basic programming skills in python, knowledge of general machine learning workflow related concepts such as feature preprocessing, loss functions, model training and evaluation etc.

We will use `pandas` library for data preprocessing and some `scikit-learn` helper functions here, but we'll explain them as we go.

Tensorflow fundamentals

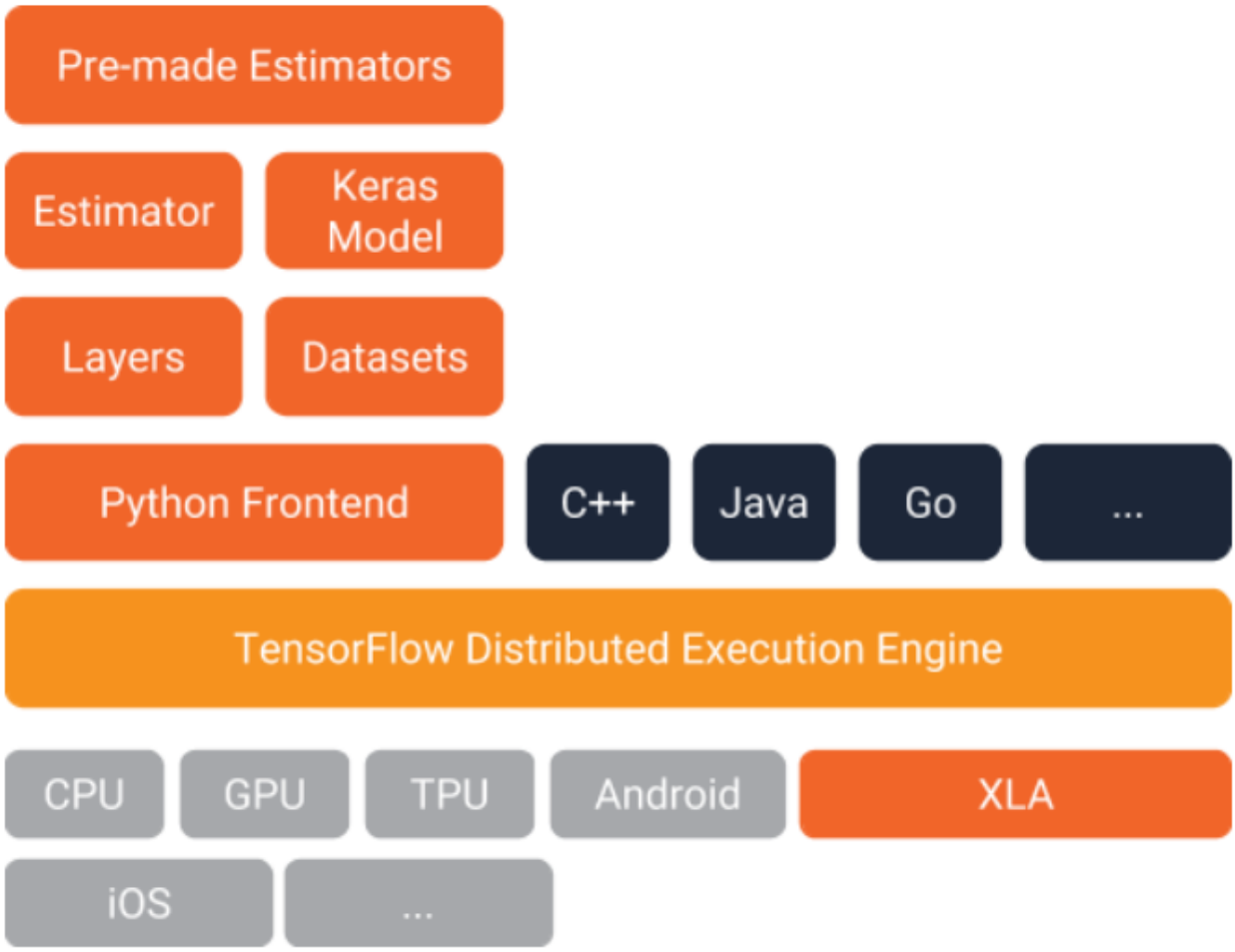
Machine learning is advancing at a rapid rate and to remain relevant a good machine learning framework has to hit the balance between flexibility and simplicity. To implement novel architectures created by researchers, ML frameworks should be extendable and flexible, but regular users often want built in models which they can readily use on their own dataset.

So, Tensorflow has to deal with many different classes of userbase with varied interests, users who want to build their custom models, users who want to use common models and users who don't care much about the specifics of the model, but wants to integrate the results in their own code infrastructure.

In order to handle these different user demographics, Tensorflow provides multiple API's in different level of abstraction. The high level API's like Keras, tf.estimator are built on top of the tensorflow core functionalities. Tensorflow estimators have been integrated with the core Tensorflow like Keras.

For users who just want to use the common models, Tensorflow provides pre-made estimators or "**Canned Estimators**" which refer to implementations of common machine learning models. We will use the canned estimators in this tutorial to predict airbnb rental prices with linear regression and learn how to use the estimator API. However, before using estimators we'll go through the basic building blocks in tensorflow.

We can see how different parts fit into tensorflow architecture in this diagram from Google developer blog post.



Tensors and Computational Graphs

Tensors are the central unit of data in tensorflow. Tensors are like numpy arrays, we can conceptually think of them as the n-dimensional abstraction of matrices. A zero-dimensional tensor is a scalar or a constant. A 1-dimensional tensor is a list or vector. A 2-D tensor is same as a n x m matrix where n = rows and m = columns. Above that we can just say n-dimensional tensors. For example,

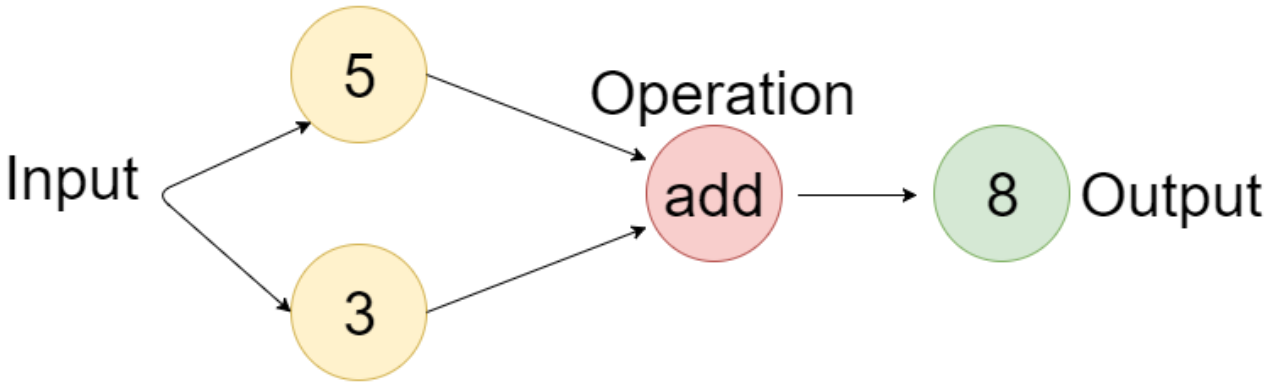
```
a = 3 (treated as 0 dimensional tensor or scalars)
a = [3,5] (treated as 1D tensor or vector)
a = [[3,5],[1,1]] (treated as 2D tensor or a matrix)
```

These tensors are passed to **operations** that perform computations on them. An **'operation'** is commonly known as an **'op'** . Operations take zero or more tensors as

inputs, performs computation with them and outputs zero or more tensors. For example, an operation like `tf.add` may take two inputs 3 and 5 and output their summation 8.

The tensors and the operations are connected to each other in a computational graph. A computational graph is defined by considering the operations as nodes and the tensors as edges. The operations are mathematical operations that are done on data and the data is passed to the operations with the tensors.

We can visualize a computational graph like this.



Here we have two input tensors that pass the constants 5 and 3 to the add operation and the operation outputs 8.

We'll code it in the following section.

Constants and Running a Session :

```
import tensorflow as tf
print(tf.__version__)
```

We will define two constant tensors a and b with `tf.constant` with constants 5 and 3 and add them up with `tf.add` as shown in the computational graph.

```
a = tf.constant(5,name = "a")
b = tf.constant(3, name = "b")
```

```
result = tf.add(a,b,name='add_a_b')
result
# Output

<tf.Tensor 'add_a_b:0' shape=() dtype=int32>
```

Unfortunately enough, our code has not produced the expected output. We can think of tensorflow core programs as having two distinct sections, first we have to define a computational graph that specifies the computations we want to do, then we have to run the code to get our actual results. We have defined our computational graph in this case, but we have not run the graph yet.

To evaluate `result` and get the output we have to run the code under a 'session'. A session takes a computational graph or part of a graph and executes it. It also holds the intermediate values and the results of performing the computation. We can create an instance of a session object from `tf.Session` class.

Following code creates a session, and evaluates the output.

```
sess = tf.Session()
sess.run(result)

# Output
8
```

Variables and Placeholders

Since in machine learning we want to update the paramaters of the models when training simply using constants whose values don't change is not enough, we need some mechanism to add trainable parameters to the computational graph. In Tensorflow we accomplish it using variables. Variables require us to specify an initial value and the data type. We create variables with the `tf.variable` op.

A graph can also be fed external inputs using placeholders so that we can feed arbitrary number of inputs from the training sets to the model. Placeholders act like tensor objects that do not have their values specified and are created using the op `tf.placeholder` .

Placeholder values must be fed when we run them. We can use `sess.run` method's `feed_dict` argument to feed the values to the placeholders. We have to specifiy the shape and datatype of the placeholder when we add them to the graph.A shape of `None` indicates that the placeholder can get any arbitrary input.

`feed_dict` is just a dictionary that maps graph elements like variables,constants or placeholders to values. We use it to overwrite the existing values of tensors. We can also use it to change variable values when running them.

```
c = tf.Variable(3,dtype=tf.float32)
d = tf.placeholder(dtype = tf.float32,shape=None)
```

Unlike constants, variables are not initialized when we call tf.Variable. We'll have to run a special operation called `tf.global_variables_initializer` to initialize the variables by a session.

```
sess.run(tf.global_variables_initializer())
print(sess.run(c,feed_dict = {c:14}))
print(sess.run(d,feed_dict = {d:[1,1,3,5]}))
# OUTPUT
14.0
[ 1.  1.  3.  5.]
```

The variable c was initialized with 3, but we have changed it to 14 with the `feed_dict` parameter. The placeholders didn't have any specified value when we initialized it, but we fed it a list of values when we ran the code.

Tensorboard

Tensorboard is a visualization tool that comes packaged with tensorflow.It's very useful to visualize large scale machine learning models to debug them and understand what's going on under the hood. With tensorboard we can also track our loss metrics and other values to see how they are changing over training steps.

For using tensorboard, we can save our graphs with by writing summaries about them with summary writers. Summaries are like condensed information about models.

Tensorboard creates visualizations out of this information.

We have to pass the directory name where our graph log files will be saved and the computational graph we want to save into the summary writer object when calling it.. `sess.graph` contains the default computational graph for this session and `writer` writes it into the directory provided in `logdir` parameter.

```
writer = tf.summary.FileWriter(logdir= "../first_graph",graph =
sess.graph)
writer.close()
```

To run TensorBoard, use the following command in the terminal of Linux or MACOS.

```
tensorboard --logdir=path/to/log-directory
```

In windows, use `tensorboard --logdir path/to/log-directory` .

Here logdir is `../first_graph` , so we can simply type

```
tensorboard --logdir=first_graph
```

Tensorboard's default port is 6006. So if you go to http://localhost:6006/#graphs tensorboard will be there.

If we go under the graphs section we can see this visualization of our tiny computational graph. The node is a mathematical operation sum that takes in two inputs a and b.

NOTE : Delete the summary folder if you want to re-run the code to run from a clean state.

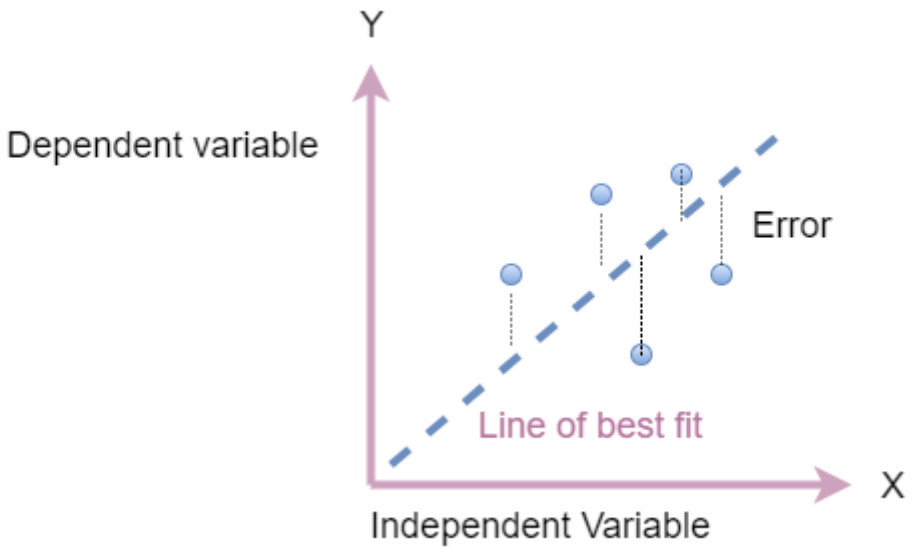
Linear Regression From Scratch :

Linear regression is a simple but powerful commonly used technique used in both statistics and machine learning . It's a 'linear' model, in other words it assumes a linear relationship between the inputs(x) and the outputs(y). Formally, the model assumes that we can get the output value using a linear combination of the input values.

In a simple dataset with only one feature and one output to predict, the form of the equation looks like

$$Y = W * X + B$$

We can see that for different values of input X we can get the predictions by using the equation. We can also visualize it.



We try to find out the best possible value for the weight and bias parameters using optimization technique against a loss function in order to fit a line through the data using the weight and the bias parameter when it comes to single feature.Loss functions tell us how good our predicted value is compared to the actual output. If we have n features the general equation is like this.

$$Y = B + W_1X_1 + W_2X_2 + ...W_nX_n$$

Here we are going to implement linear regression from scratch for a random dataset of one feature. First we are going to create our model parameters, W and b which stands for weight and bias.

```
# Model Parameters

W = tf.Variable([3.0],name='weight')
b = tf.Variable([-2.0],name='bias')
```

We create two placeholders X and y for the training set and labels. We'll feed the inputs to them during training.

```
# Model inputs

# training data
X = tf.placeholder(tf.float32)
# y
Y = tf.placeholder(tf.float32)
```

We define the model like the equations specified above for a single feature.

```
# Model definitation

predictions = W*X + b
```

We'll use sum of squared errors as our loss function. To implement it, for each instance we calculate the error first. Error is the difference between the prediction we get from the model and the original values we were supposed to get. Then we square the error and add them up.

Since we are summing the squared errors or residuals(another name for errors) it's also called the OLS or ordinary least squares method. Note that errors are calculated for each sample or instance, while the loss tells us how good our model is performing on the whole dataset.

```
# loss function. Here we use sum of squared errors.

loss = tf.reduce_sum(tf.square(predictions-Y))
```

Optimizers are used for finding the best value for some parameters with respect to some loss function in machine learning models. There's many kind of optimizers, the one we are using here is called Gradient Descent.We start with a random value for the weight and the bias. The optimizer updates the weight and the bias parameter in a direction(increasing or decreasing the numbers) to minimize the value of the loss. We also provide a learning rate to use as a scale factor while updating.

```
# training op

train = tf.train.GradientDescentOptimizer(0.001).minimize(loss)
```

We create some random values for training set and label and feed them into the model during running the code. As before we initialize all our variables before running them usin `global_variables_initializer` . We run the model for 2000 steps here.

```
# train data

x = [1.1,2.0,3.5,4.8]
y = [2.0,3.4,4.2,5.1]

sess.run(tf.global_variables_initializer())
for train_step in range(2000):
    sess.run(train,{X:x,Y:y})
```

We can see the final results after executing the model.

```
weight, bias, loss = sess.run([W, b, loss], {X:x, Y:y})
print("W: %s b: %s loss: %s"%(weight,bias,loss))

# Output

W: [ 0.84224832] b: [ 1.23781681] loss: 0.288
```

The next part will cover the tensorflow estimators.

Introducing Tensorflow Estimators

As mentioned earlier, estimators is a high level API integrated with Tensorflow that allows us to work with pre-implemented models and provides tools for quickly creating new models as need by customizing them. The interface is loosely scikit-learn inspired and follows a train-evaluate-predict loop similar to scikit-learn. `Estimators` is the base class, canned estimators or pre-implemented models are the sub-class. We are using the canned estimators in this tutorial.

Also, please consult the excellent research paper : "Tensorflow Estimators : Managing Simplicity Vs Flexibility In High Level Machine Learning FrameWorks" for further learning. Note that I have understood most of the stuff from the paper to begin with.

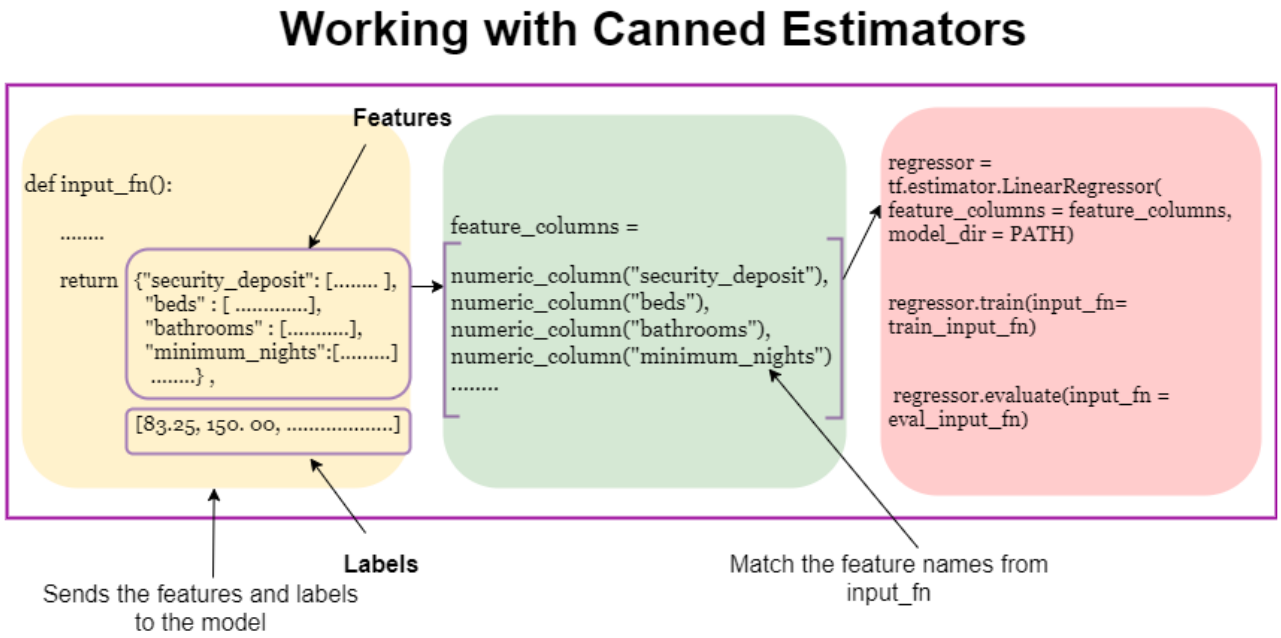
Estimators deal with all the details of creating computational graphs, initializing variables, training the model and saving checkpoint and logging files for Tensorboard behind the scene. But to work with the estimators, we've to become comfortable with two new concepts, feature columns and input functions. Input functions are used for passing input data to the model for training and evaluation. Feature columns are specifications for how the model should interpret the input data. We will cover the feature columns and input function in detail in the later sections.

Since we are going to learn via doing, we can start analyzing from beginning where we are now and where we have to go. We know that we have some training data on airbnb rental pricings and their prices. We want to predict the prices of those rentals from the features in the dataset. We also know that we need a machine learning model to do that. Tensorflow is offering pre-made model implementations for doing it and giving functionalities for representing our features in different ways using the feature columns. We just need to build an input function and send our data to the estimator. Feature columns will connect the data from the input function to the estimators for training and evaluating the model. Before that, we'll have to do some minimal preprocessing on the raw dataset.

So our general workflow will be to follow these steps :

1. Loading the libraries and dataset.
2. Data proprocessing.
3. Defining the feature columns.
4. Building input function.
5. Model instantiation, training and evaluation.
6. Generating prediction.
7. Visualizing the model and the loss metrics using Tensorboard.

Visually, we can see the workflow here.



Inspired by Diagrams from : https://developers.googleblog.com/2017/11/introducing-tensorflow-feature-columns.html

Load the libraries

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
```

```
plt.style.use("seaborn-colorblind")
%matplotlib inline

# only displays the most important warnings

tf.logging.set_verbosity(tf.logging.FATAL)
```

Data Overview

The dataset has `3585` rows and `95` columns. However we will only use a subset of the features.These features will be used :

- `property_type` : Categorical. Describes the type of the property.
- `room_type` : Categorical. Describes the type of the room.
- `bathrooms` : Number of Bathrooms.
- `bedrooms` : Number of bedrooms.
- `beds` : Number of beds.
- `bed_type` : Type of the bed in listing. Categorical feature.
- `accommodates` : Number of people the rental will accommodate.
- `host_total_listings_count` : Number of listings posted by the host.
- `number_of_reviews` : Number of reviews received by the rental.
- `review_scores_value` : Average value of the reviews for this listing.
- `neighbourhood_cleansed` : Categorical feature. Describes the neighbourhood of the rental.
- `cleaning_fee` : Cleaning fee for the rental.
- `minimum_nights` : Minimum nights the guest must stay to be considered for rental.
- `security_deposit` : Amount of security deposit the guest has to pay to the host before renting.

- `host_is_superhost` : Categorical. True if the host is a superhost. False otherwise.
- `instant_bookable` : Categorical.True if the rental is instantly bookable. False Otherwise.
- `price` : Continuous target variable for the regression task.

Now we'll load the dataset with `pandas` . Tensorflow integrates well with `pandas` and provides many useful functions to work with it.

```
used_features =
['property_type','room_type','bathrooms','bedrooms','beds','bed_type
','accommodates','host_total_listings_count'

,'number_of_reviews','review_scores_value','neighbourhood_cleansed',
'cleaning_fee','minimum_nights','security_deposit','host_is_superhos
t',
                'instant_bookable', 'price']

boston = pd.read_csv('boston_listings.csv', usecols = used_features)
print(boston.shape)
boston.head(2)
```

```
(3585, 17)
```

	host_is_superhost	host_total_listings_count	neighbourhood_cleansed	property_type	room_type	accommodates	bathrooms	bedrooms	beds	bed_type	
0	f	1	Roslindale	House	Entire home/apt	4	1.5	2.0	3.0	Real Bed	$2
1	f	1	Roslindale	Apartment	Private room	2	1.0	1.0	1.0	Real Bed	$

dataframe.head() method outputs the first few rows of the dataset

Data Preprocessing

To use the features in our tensorflow model, we need to convert them to Tensors. But before that we need to take some more data cleaning steps to prepare our data for machine learning. Here's a brief overview of the cleaning steps.

- `price` , `security_deposit` and `cleaning_fee` are numbers, but they are loaded as strings.These numbers also contain some non-numeric characters (`$` and `,`). We'll

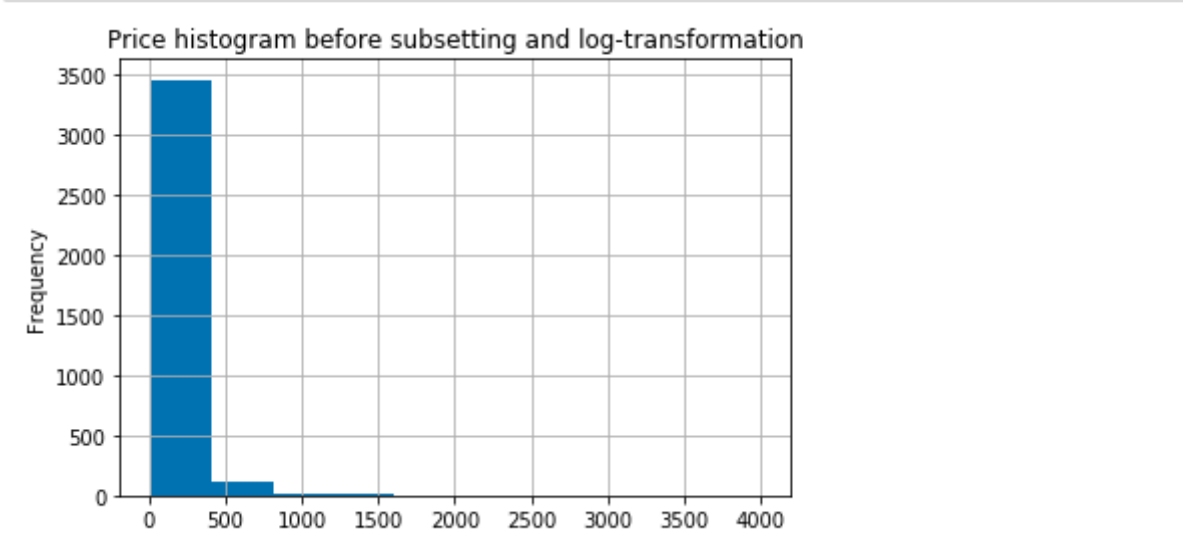
remove the non-numeric characters from the features, convert them to `float` and fill the missing values with the median value of each feature.

- Similarly, some other features like `bathroom`, `bed` has some missing values which are also filled with the median.
- `property_type` is a categorical variable. The missing values are filled with the most common category `Apartment`.

```
for feature in ["cleaning_fee","security_deposit","price"]:
    boston[feature] = boston[feature].map(lambda
x:x.replace("$",'').replace(",",''),na_action = 'ignore')
    boston[feature] = boston[feature].astype(float)
    boston[feature].fillna(boston[feature].median(),inplace = True)


for feature in
["bathrooms","bedrooms","beds","review_scores_value"]:
    boston[feature].fillna(boston[feature].median(),inplace = True)

boston['property_type'].fillna('Apartment',inplace = True)
```

Before moving on to the other features, we check our target variable `price`.

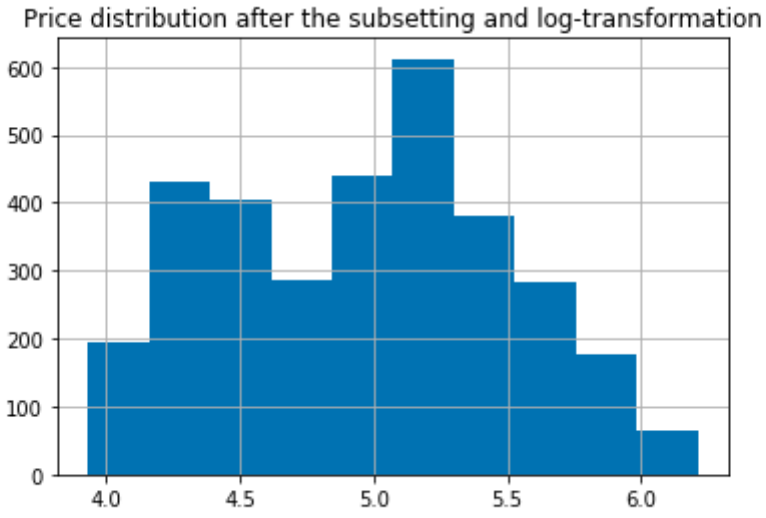
```
boston["price"].plot(kind = 'hist',grid = True)
plt.title("Price histogram before subsetting and log-
transformation");
```



```
boston['price'].skew()

# Output

8.5214835656839512
```

Prices are heavily skewed in this dataset. In the histogram we can also see most of the data falls under 500. So we'll use a subset of our dataset where the price ranges from 50–500 for now to remove very large and very small prices. We'll also log-transform the target variable to reduce skewness.

```
boston = boston[(boston["price"]>50)&(boston["price"]<500)]
target = np.log(boston.price)
target.hist()
plt.title("Price distribution after the subsetting and log-
transformation");
```




```
features = boston.drop('price',axis=1)
features.head()
```

Out[11]:

Out[11]:

	host_is_superhost	host_total_listings_count	neighbourhood_cleansed	property_type	room_type	accommodates	bathrooms	bedrooms	beds	bed_type	se
0	f	1	Roslindale	House	Entire home/apt	4	1.5	2.0	3.0	Real Bed	
1	f	1	Roslindale	Apartment	Private room	2	1.0	1.0	1.0	Real Bed	
2	t	1	Roslindale	Apartment	Private room	2	1.0	1.0	1.0	Real Bed	
3	f	1	Roslindale	House	Private room	4	1.0	1.0	2.0	Real Bed	
4	t	1	Roslindale	House	Private room	2	1.5	1.0	2.0	Real Bed	

Split the data into train and test set

Now that we have our features and the target, we can use scikit-learn to split the data for convinience. Scikit-learn provides a `train_test_split` function for splitting a pandas DataFrame into a training and testing set.

`train_test_split` accepts the features and the target as parameters and returns the randomly shuffled data. We have set 33% of the data in the test set and the rest are in training set. The `test_size` parameter controls what portion of the data should be in the test set. We also set a `random_state` for reproducability.

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(
     features, target, test_size=0.33, random_state=42)
```

Introducing Feature Columns

Generally, machine learning models take numbers as input and outputs numbers for efficient implementation. In Tensorflow, the models take Dense `Tensor` s as input and outputs Dense `Tensor` s. But real world datasets have sparse features like product id, category, location, video id etc. For large datasets, converting each of the categorical features to numerical representations often consume a huge amount of time and the process is also error prone. There are also other feature preprocessing steps like bucketization, scaling, crossing features, embedding etc people often take before feeding the data to the models. To simplify this process Tensorflow offers `FeatureColumn` abstraction.

When we instantiate a canned estimator we have to pass it a list of `FeatureColumn` s for the `feature_column` parameter. `FeatureColumn` s handle the conversion of the spars or dense(numerical) features to a dense `` `Tensor `` usable by the core model.

The type of Feature column to use depends on the feature type and the model type.

- Feature type : Numeric and categorical features need to be handled differently.
- Model type : Linear models and the neural network models handle categorical features differently.

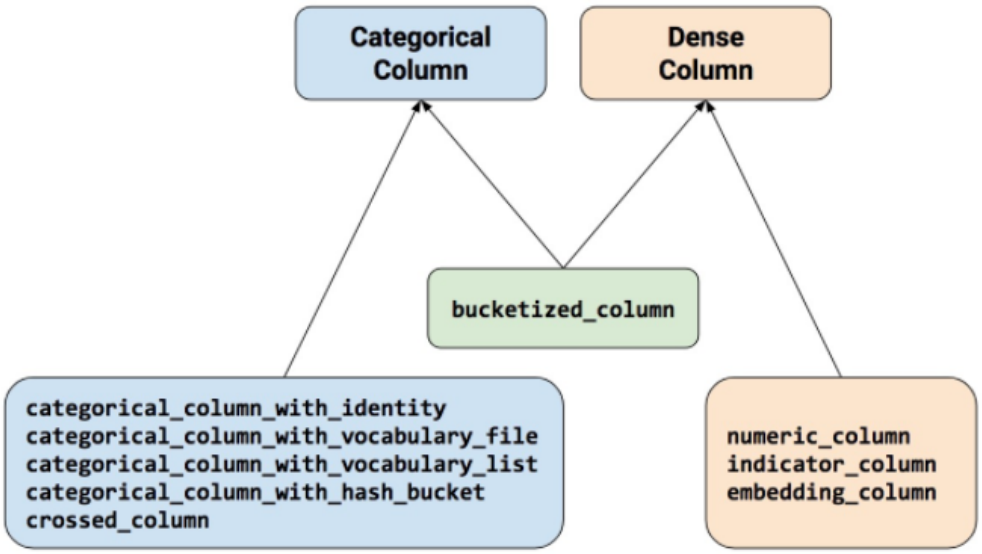


Image Credit : https://developers.googleblog.com/2017/11/introducing-tensorflow-feature-columns.html

In this tutorial we'll show how to handle numeric and the categorical features with two different `FeatureColumn` . To learn more about this topic, you can consult a great tutorial published by the Tensorflow team in Google Research Blog.

First we'll separate the column names of the numeric features and the categorical features.

```
# Get all the numeric feature names

numeric_columns =
['host_total_listings_count','accommodates','bathrooms','bedrooms','
beds',
```

```
'security_deposit','cleaning_fee','minimum_nights','number_of_review
s',
 'review_scores_value']

# Get all the categorical feature names that contains strings

categorical_columns =
['host_is_superhost','neighbourhood_cleansed','property_type','room_
type','bed_type','instant_bookable']
```

Numeric Feature Columns

Numeric features can be represented by `numeric_column` which is used for real valued features. When we create a `numeric_column` we have to pass a unique string to its e `key` parameter. The value of `key` will be used as the name of the column. We can also specify the data type or shape of the `numeric_column` if necessary, but here we will just use the defaults.

We use a list comprehension to create `numeric_columns` for all the numeric features. We basically create a `numeric_column` for each column in the `numeric_columns` list we made before.

```
numeric_features = [tf.feature_column.numeric_column(key = column)
for column in numeric_columns]
print(numeric_features[0])

# Output

_NumericColumn(key='host_total_listings_count', shape=(1,),
default_value=None, dtype=tf.float32, normalizer_fn=None)
```

Categorical Feature Columns

There are many ways to handle categorical features in tensorflow.
`categorical_column_with_vocabulary_list` is just one of them. For small number of categories we give the categorical column the fixed list of values the column will take. It represents each categorical feature in it's one-hot vector representation.

In the one hot representation we replace each categorical instance with a n-dimensional boolean vector which has the size of the number of categories in the feature and marks

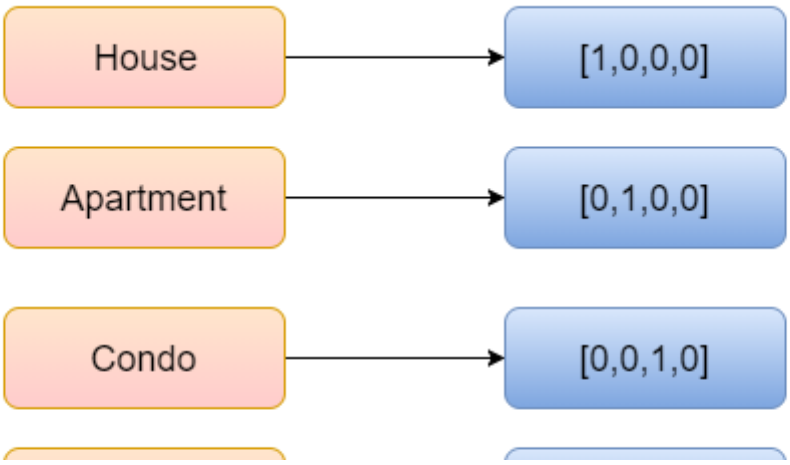
the presence and absence of each category with 1 and 0. For example if we have a feature “Gender” with two categories {male,female}, each time we see “male” we will replace it with a vector [1,0] and each time we see “female” we’ll replace it with a vector [0,1].

`categorical_column_with_vocabulary_list` must have following inputs :

- key : a unique string identifying the input feature which would be used as the name of the column.
- vocabulary_list : An ordered list defining the possible values for this categorical feature

The syntax goes like this.

```
Property_type =
["apartment","condo","apartment","villa","house","house".......]
gender_column =
tf.feature_column.categorical_column_with_vocabulary_list(key =
"Property_type",

vocabulary_list = ["house",

"apartment",

"condo",

"villa"]

)
```



Categorical column representation of "Property_type" Feature

Other feature columns for categorical features are :

- `categorical_column_with_identity` : Returns the column as it is.
- `categorical_column_from_vocabulary_file` : Instad of giving the column values in a list, we read it from a file.
- `categorical_column_with_hash_bucket` : If the number of values a categorical column can take is really large, instead of writing all the possible values in a list and giving it , we can use hashing to let tensorflow take care of it behind the scene. But there's a chance of 'hash collision' where two or more categories can be mapped to same place.
- `crossed_column` : We can cross a categorical feature with a numerical or another categorical feature. For example, let's say we take a categorical feature "Gender" and another feature "Education", we can create new features like "female_x_phd" or "female_x_bachelors".

```
categorical_features =
[tf.feature_column.categorical_column_with_vocabulary_list(key =
column,

vocabulary_list = features[column].unique())

for column in categorical_columns]

print(categorical_features[3])

# output

_VocabularyListCategoricalColumn(key='room_type', vocabulary_list=
('Entire home/apt', 'Private room', 'Shared room'), dtype=tf.string,
default_value=-1, num_oov_buckets=0)
```

In this case I just used another list comprehension to create all the categorical feature columns directly instead of writing out the vocabulary list for all of them. `features[column].unique` returns a list containining all the unique values the categorical feature may take and we set `vocabulary_list` parameter to those values. Then we combine all the numeric and the categorical feature columns to one list so that we can pass it to our canned estimator.

```
linear_features = numeric_features + categorical_features
```

Build Input Function

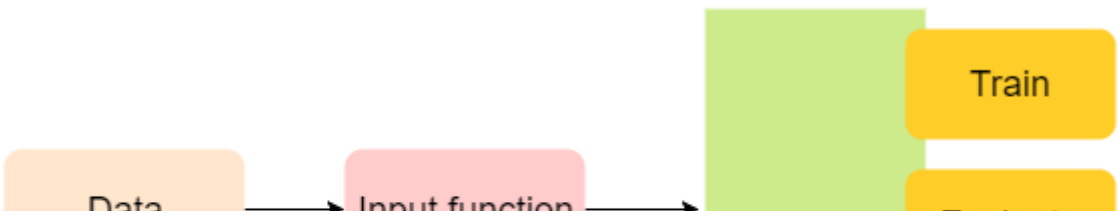
When we train our model we have to pass the features and the labels to the model. Estimators require that we use an input function for this.

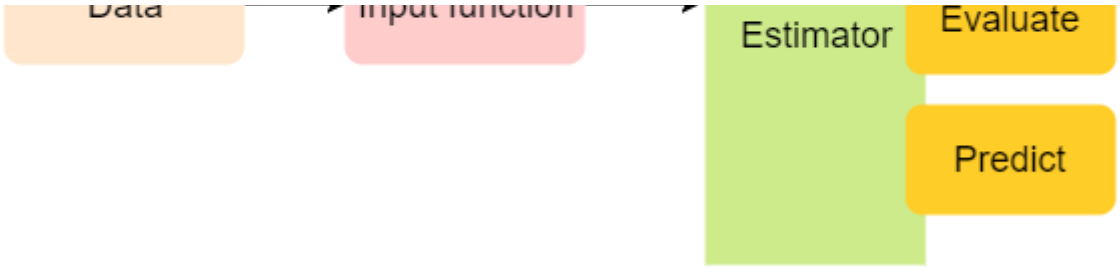
The input function must return a tuple containing two elements.

1. A dictionary that contains the feature column names as key and maps them to the tensors containing the feature data for a training batch.
2. A list of labels for the training batch.

The structure of an input function looks like this.

```
def my_input_fn():


  # Preprocess your data here...

  # ...then return 1) a mapping of feature columns to Tensors with
  # the corresponding feature data, and 2) a Tensor containing labels

  return feature_cols, labels
```





For learning more about the input functions, see this tutorial in the documentation named "Building Input functions".

Luckily tensorflow provides functionality for feeding a pandas Dataframe straight into a tensorflow estimator with the `pandas_input_fn` function. `pandas_input_fn` has many parameters but we will use the following.

- `x` : pandas `DataFrame` object that has the features.
- `y` : pandas `Series` object that has the labels.
- `batch_size` : a number specifying the batch size
- `shuffle` : boolean. whether to shuffle the data or not
- `num_epoch` : int. number of epoches to iterate over the data. one epoch means going over all the training data once. None means it will cycle through input data forever.

Note that `pandas_input_fn` returns an input function that would feed the data to the tensorflow model. Here we create two input functions `training_input_fn` and `eval_input_fn` that takes the training and test set features and labels respectively.

`num_epoch` is set to `None` in the `training_input_fn` because we want to go over the training dataset multiple times as the model trains. We want to go over the test dataset only once to evaluate the model, so we set `num_epoch` to 1.

```
# Create training input function

training_input_fn = tf.estimator.inputs.pandas_input_fn(x = X_train,
                                                        y=y_train,

batch_size=32,
```

```
                                                        shuffle=
True,
                                                        num_epochs =
None)

# create testing input function

eval_input_fn = tf.estimator.inputs.pandas_input_fn(x=X_test,
                                                    y=y_test,
                                                    batch_size=32,
                                                    shuffle=False,
                                                    num_epochs = 1)
```

Instantiate Model

We instantiate the linear model by passing the list containing the feature columns to the `feature_columns` parameter. We also specify a model directory with the `model_dir` parameter where tensorflow will store the model graph and other information. We'll be able to visualize the model architecture and the loss metrics later with tensorboard using them.

We can choose different optimizers by using the `optimizer` parameter, but we'll go with the defaults here. The default loss function is sum of squared errors. If you want to customize the loss function or extend other properties, you can check out Creating Estimators in Tensorflow guide.

```
linear_regressor =
tf.estimator.LinearRegressor(feature_columns=linear_features,
                                                  model_dir =
"linear_regressor")
```

Train Model

We pass the `training_input_fn` to the `input_fn` parameter of the `train` method and specify the number of steps we want to train the model. Note that we have to pass the function object(`input_fn = training_input_fn`) to the `input_fn` parameter, not the return value of the function call.(`input_fn = training_input_fn()`).

```
linear_regressor.train(input_fn = training_input_fn,steps=2000)
```

```
# Output

<tensorflow.python.estimator.canned.linear.LinearRegressor at
0x2087173aac8>
```

Evaluate Model

For evaluating the model we simply pass the `eval_input_fn` to the `input_fn` parameter in `regressor.evaluate` method as we did earlier with the `fit` method. It returns a `dict` containing loss after evaluating the model on the test set. Currently loss is defined as mean squared error(MSE) in the `LinearRegressor` model. If you want to build your own estimator see this tutorial.

```
linear_regressor.evaluate(input_fn = eval_input_fn)
# print("Loss is " + str(loss))

# Output

{'average_loss': 0.15274006, 'global_step': 2000, 'loss': 4.8472509}
```

`LinearRegressor.predict` method returns a `generator` for yielding predictions. Since we have set the `num_epoch` to 1 the input function we are using to feed the data to the evaluation function will go over the test dataset only once. We can easily convert it to a list for getting the predictions.

```
pred = list(linear_regressor.predict(input_fn = eval_input_fn))
pred = [p['predictions'][0] for p in pred]
```

Since we had log-transformed the price before we've to use the exponential function to inverse it and get our prices for the rental values in original dollar terms.

```
prices = np.exp(pred)
print(prices)
# Output
array([  73.30127716,   76.89142609,   83.49259949, ...,
133.32876587,
         66.23993683,  135.94793701], dtype=float32)
```

Visualizing Loss and Model Architecture with Tensorboard

As mentioned earlier, Tensorboard makes visualizing any model really easy. We simply need to set the `logdir` parameter to the directory we have saved our estimator model when we want to visualize it. When we instantiated the model we can set the `model_dir` parameter to store the model. Tensorflow automatically save checkpoints and the details of the models under the hood in the model directory and we visualize them with tensorboard.

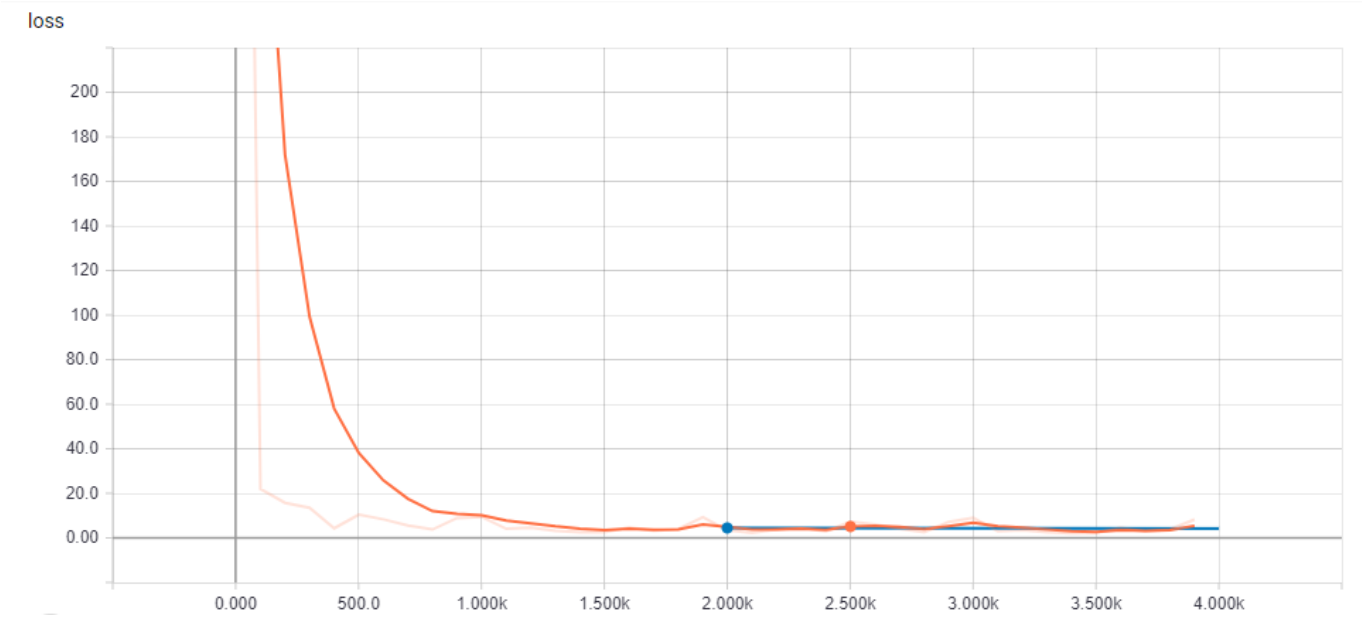
Let's visualize our linear model with Tensorboard first. Type the following commands in the terminal

```
tensorboard --logdir=linear_regressor
```

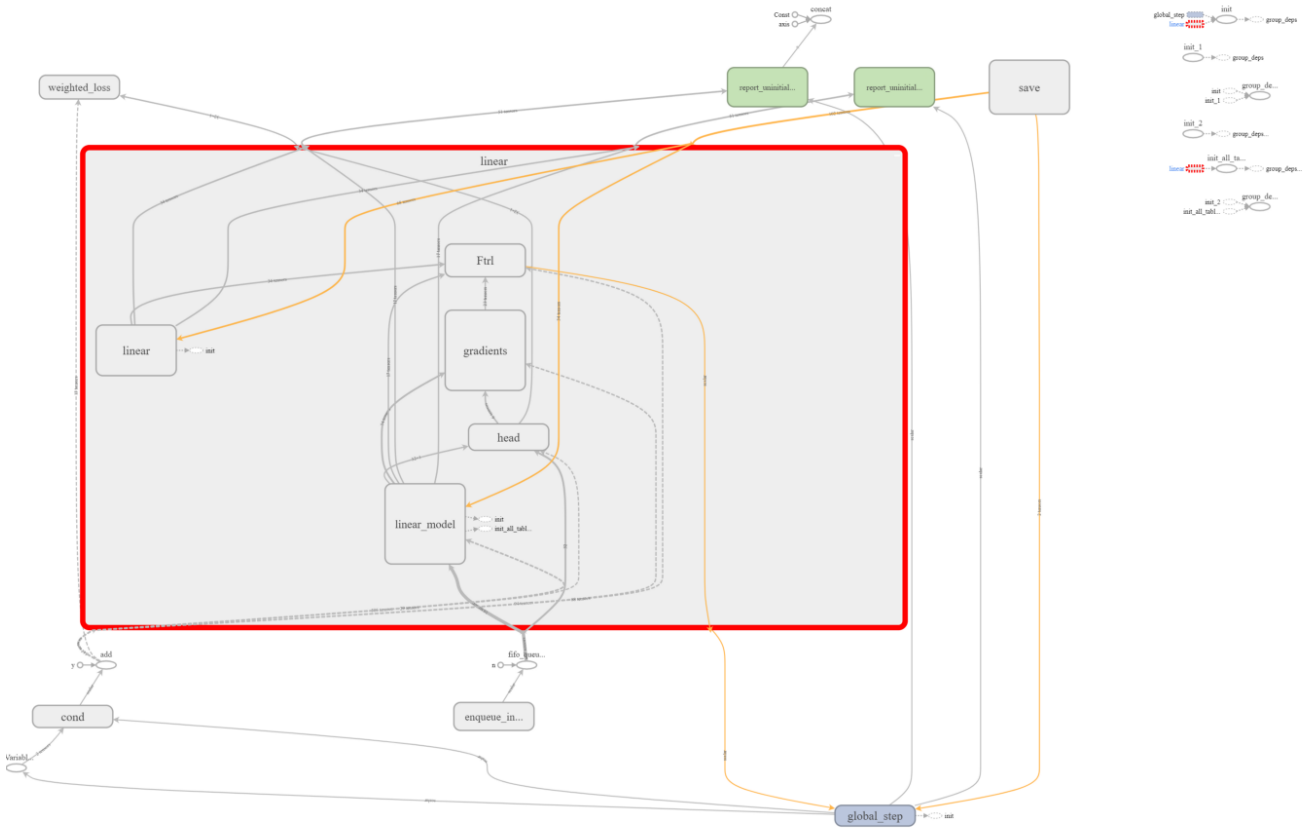
or

```
tensorboard --logdir linear_regressor
```
(for windows)

We can visualize the loss curve under scalars section. Tensorboard also outputs some other metrics like average loss for estimators. Here is the loss curve for the linear model which is decreasing over the number of steps.



The model architecture under the hood can be seen under the graphs section. By clicking on the nodes and the subgraphs we can explore different parts of the computational graph. It's very helpful for writing our own estimators too.



Note: Again, if you want to re-run the code delete the directory containing the checkpoints for the model to start from a clean state.

Visualizing Feature Weights

We may want to check the feature weights for the `linear_regressor` model. We can find the variable names for each feature from the model by using the `get_variable_names` method first and then get the weights by using those names. Here we demonstrate how to get the feature weights for the neighbourhoods and visualize them with a `pandas` dataframe for convinience.

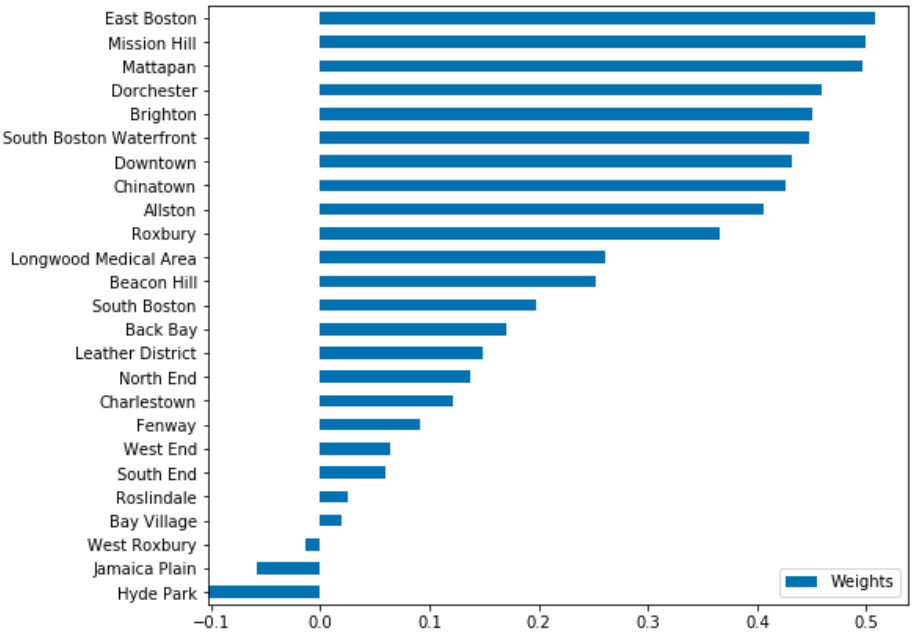
First we check how the variable names look like by printing a few names.

```
linear_regressor.get_variable_names()[5:10]
```

```
# Output

['linear/linear_model/bathrooms/weights/part_0/Ftrl',
 'linear/linear_model/bathrooms/weights/part_0/Ftrl_1',
 'linear/linear_model/bed_type/weights',
 'linear/linear_model/bed_type/weights/part_0/Ftrl',
 'linear/linear_model/bed_type/weights/part_0/Ftrl_1']
```

Then we get the feature weights for the `neighbourhood_cleansed` categorical feature and store them in `property_weights`. I've made a Dataframe with the feature weights as values and the different neighborhood names as index for convenience.

```
property_weights =
linear_regressor.get_variable_value('linear/linear_model/neighbourho
od_cleansed/weights').flatten()
property_keys =
boston["neighbourhood_cleansed"].value_counts().keys()
pd.DataFrame({"Weights":property_weights},index=property_keys).sort_
values("Weights",ascending=True).plot(kind="barh",

figsize= (8,7),

grid = False

);
```



Next steps :

- Use the same techniques on a different dataset.
- Use other estimators like `DNNRegressor` or `DNNLinearCombinedRegressor`.
- Create your own estimator!

References

- https://www.tensorflow.org/get_started/estimator
- https://www.tensorflow.org/get_started/input_fn
- Effective Tensorflow for non experts
- https://research.google.com/pubs/pub46369.html
- https://developers.googleblog.com/2017/11/introducing-tensorflow-feature-columns.html
- https://developers.googleblog.com/2017/09/introducing-tensorflow-datasets.html

Machine Learning TensorFlow Tensorboard Deep Learning Data Science