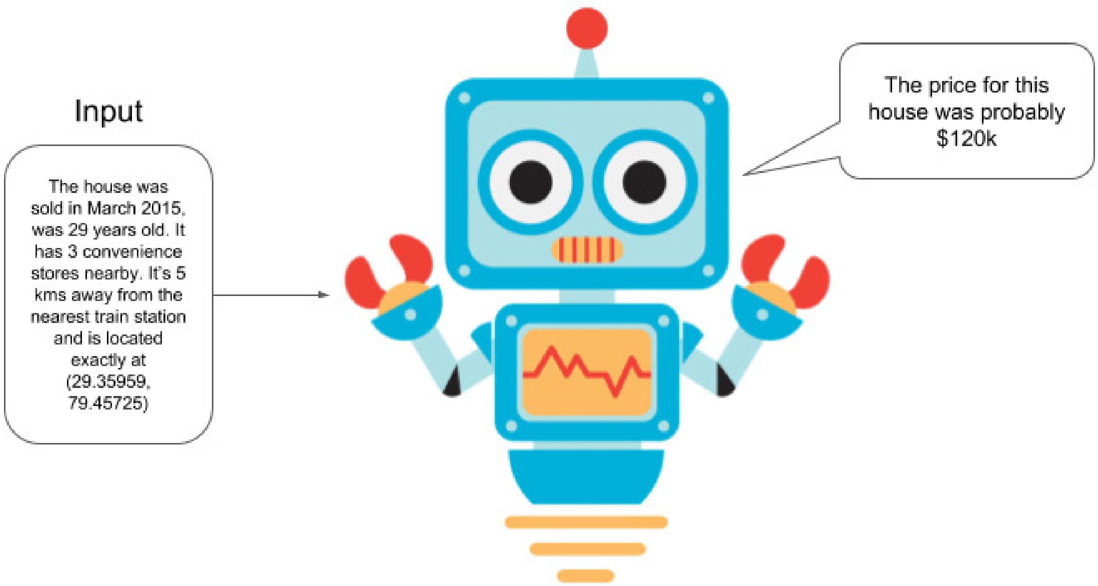
House Price Prediction Using Keras

For this project, I am going to predict price of houses given the following features:

1. Year of sale of the house
2. The age of the house at the time of sale
3. Distance from city center
4. Number of stores in the locality
5. The latitude
6. The longitude



Note: This notebook uses `python 3` and these packages: `tensorflow`, `pandas`, `matplotlib`, `scikit-learn`.

Importing Libraries & Helper Functions

First of all, we will need to import some libraries and helper functions. This includes TensorFlow and some utility functions that I've written to save time.

In [1]:

```
#Importing the relevant libraries
import pandas as pd
import matplotlib.pyplot as plt
import tensorflow as tf

from utils import *
from sklearn.model_selection import train_test_split
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.callbacks import EarlyStopping, LambdaCallback

%matplotlib inline
tf.logging.set_verbosity(tf.logging.ERROR)

print('Libraries imported.')
```

```
Libraries imported.
```

Importing the Data

2.1: Importing the Data

The dataset is saved in a `data.csv` file.

In [2]:
```
df = pd.read_csv("data.csv",names = column_names)
df.head()
```

Out[2]:

	serial	date	age	distance	stores	latitude	longitude	price
0	0	2009	21	9	6	84	121	14264
1	1	2007	4	2	3	86	121	12032
2	2	2016	18	3	7	90	120	13560
3	3	2002	13	2	2	80	128	12029
4	4	2014	25	5	8	81	122	14157

2.2: Check Missing Data

Now, let us check if the data has any missing values.

In [4]:
```
#len(df) # 5000 rows

df.isna().sum()
```

Out[4]:
```
serial       0
date         0
age          0
distance     0
stores       0
latitude     0
longitude    0
price        0
dtype: int64
```

Data Normalization

3.1: Data Normalization

Here I am making it easier for optimization algorithms to find minimas by normalizing the data before training a model.

In [5]:
```
df = df.iloc[:, 1:] # Ignoring the "Serial" column
df_norm = (df - df.mean())/df.std()
df_norm.head()
```

Out[5]:

	date	age	distance	stores	latitude	longitude	price
0	0.015978	0.181384	1.257002	0.345224	-0.307212	-1.260799	0.350088
1	-0.350485	-1.319118	-0.930610	-0.609312	0.325301	-1.260799	-1.836486
2	1.298598	-0.083410	-0.618094	0.663402	1.590328	-1.576456	-0.339584
3	-1.266643	-0.524735	-0.930610	-0.927491	-1.572238	0.948803	-1.839425
4	0.932135	0.534444	0.006938	0.981581	-1.255981	-0.945141	0.245266

3.2: Convert Label Value

Because we are using normalized values for the labels, we will get the predictions back from a trained model in the same distribution. So, we need to convert the predicted values back to the original distribution if we want predicted prices.

In [6]:
```
y_mean = df['price'].mean()
y_std = df['price'].std()

def convert_label_value(pred):
    return int(pred*y_std + y_mean)

print(convert_label_value(0.350088))
```

```
14263
```

This price, $14,263 is the same (approximately) as the original price value.

Create Training and Test Sets

4.1: Select Features

Now, we will remove the column **price** from the list of features as it is the label and should not be used as a feature.

In [7]:
```
X = df_norm.iloc[:,:6]
X.head()
```

Out[7]:

	date	age	distance	stores	latitude	longitude
0	0.015978	0.181384	1.257002	0.345224	-0.307212	-1.260799
1	-0.350485	-1.319118	-0.930610	-0.609312	0.325301	-1.260799
2	1.298598	-0.083410	-0.618094	0.663402	1.590328	-1.576456
3	-1.266643	-0.524735	-0.930610	-0.927491	-1.572238	0.948803
4	0.932135	0.534444	0.006938	0.981581	-1.255981	-0.945141

4.2: Select Labels

In [9]:
```
y = df_norm.iloc[:,-1]
y.head()
```

Out[9]:
```
0    0.350088
1   -1.836486
2   -0.339584
3   -1.839425
4    0.245266
Name: price, dtype: float64
```

4.3: Feature and Label Values

We will need to extract just the numeric values for the features and labels as the TensorFlow model will expect just numeric values as input.

In [10]:
```
#Now, we will convert X and y into numpy arrays
x_arr = X.values
y_arr = y.values
print('Features Array shape: ',x_arr.shape)
print("Label Array shape : ",y_arr.shape)
```

```
Features Array shape:  (5000, 6)
Label Array shape :  (5000,)
```

4.4: Train and Validation Split

We will keep some part of the data aside as a **validation** set. The model will not use this set during training and it will be used only for checking the performance of the model in trained and un-trained states. This way, we can make sure that we are going in the right direction with our model training.

In [11]:
```
x_train,x_val,y_train,y_val = train_test_split(x_arr,y_arr,test_size = 0.05,
                                               random_state = 0)
print('Training Set Features: ', x_train.shape)
print('Training Set Labels: ', y_train.shape)
print('Validation Set Features: ', x_val.shape)
print('Validation Set Labels: ', y_val.shape)
```

```
Training Set Features:  (4750, 6)
Training Set Labels:  (4750,)
Validation Set Features:  (250, 6)
Validation Set Labels:  (250,)
```

Create the Model

5.1: Create the Model

Let's write a function that returns an untrained model of a certain architecture.

In [14]:
```python
def get_model():
    model = Sequential([
        Dense(10,input_shape = (6,),activation = 'relu'),
        Dense(50,activation = 'relu'),
        Dense(30, activation = 'relu'),
        Dense(1) # As this is a regression problem, we don't use activation
        #function at this layer
    ])

    model.compile(
        loss = 'mse',
        optimizer = 'adam'
    )

    return model

get_model().summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 10)                70
_________________________________________________________________
dense_1 (Dense)              (None, 50)                550
_________________________________________________________________
dense_2 (Dense)              (None, 30)                1530
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 31
=================================================================
Total params: 2,181
Trainable params: 2,181
Non-trainable params: 0
_________________________________________________________________
```

Model Training

6.1: Model Training

We can use an `EarlyStopping` callback from Keras to stop the model training if the validation loss stops decreasing for a few epochs.

In [18]:
```
es_cb = EarlyStopping(monitor = 'val_loss',patience = 5)

model = get_model()
preds_on_untrained = model.predict(x_val)

history = model.fit(
    x_train, y_train,
    validation_data = (x_val,y_val),
    epochs = 500,
    callbacks = [es_cb]
)
```

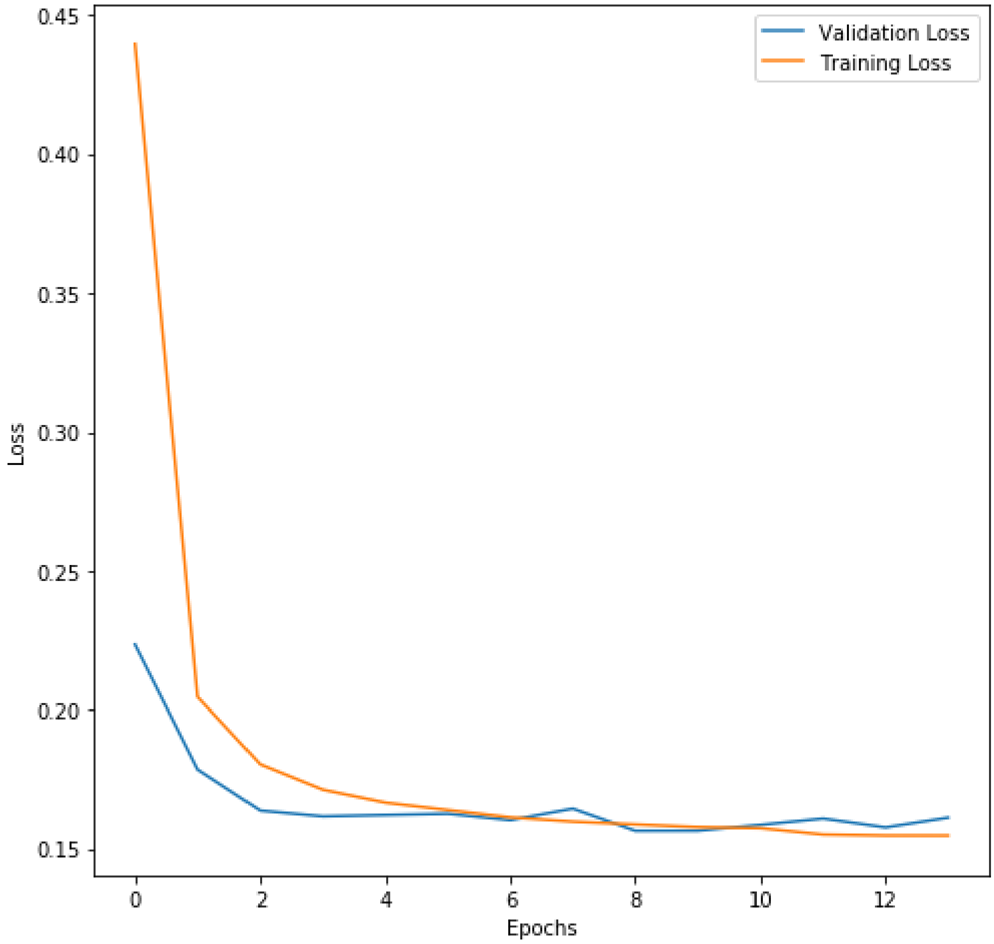
```
Train on 4750 samples, validate on 250 samples
Epoch 1/500
4750/4750 [==============================] - 1s 125us/sample - loss: 0.4397 - val_los
s: 0.2235
Epoch 2/500
4750/4750 [==============================] - 0s 47us/sample - loss: 0.2047 - val_loss:
0.1786
Epoch 3/500
4750/4750 [==============================] - 0s 49us/sample - loss: 0.1804 - val_loss:
0.1638
Epoch 4/500
4750/4750 [==============================] - 0s 47us/sample - loss: 0.1713 - val_loss:
0.1617
Epoch 5/500
4750/4750 [==============================] - 0s 44us/sample - loss: 0.1667 - val_loss:
0.1622
Epoch 6/500
4750/4750 [==============================] - 0s 47us/sample - loss: 0.1640 - val_loss:
0.1627
Epoch 7/500
4750/4750 [==============================] - 0s 46us/sample - loss: 0.1614 - val_loss:
0.1603
Epoch 8/500
4750/4750 [==============================] - 0s 43us/sample - loss: 0.1598 - val_loss:
0.1645
Epoch 9/500
4750/4750 [==============================] - 0s 43us/sample - loss: 0.1588 - val_loss:
0.1565
Epoch 10/500
4750/4750 [==============================] - 0s 47us/sample - loss: 0.1578 - val_loss:
0.1565
Epoch 11/500
4750/4750 [==============================] - 0s 50us/sample - loss: 0.1574 - val_loss:
0.1586
Epoch 12/500
4750/4750 [==============================] - 0s 44us/sample - loss: 0.1552 - val_loss:
0.1609
Epoch 13/500
4750/4750 [==============================] - 0s 44us/sample - loss: 0.1548 - val_loss:
0.1577
Epoch 14/500
4750/4750 [==============================] - 0s 44us/sample - loss: 0.1548 - val_loss:
0.1612
```

We can note here, that our model has stopped at Epoch 14/500, which suggests that there was not considerable change in the validation loss value which is why the model stopped training and did not run on all the epochs

6.2: Plot Training and Validation Loss

Let's use the `plot_loss` helper function to take a look training and validation loss.

In [19]:
```
plot_loss(history)
```



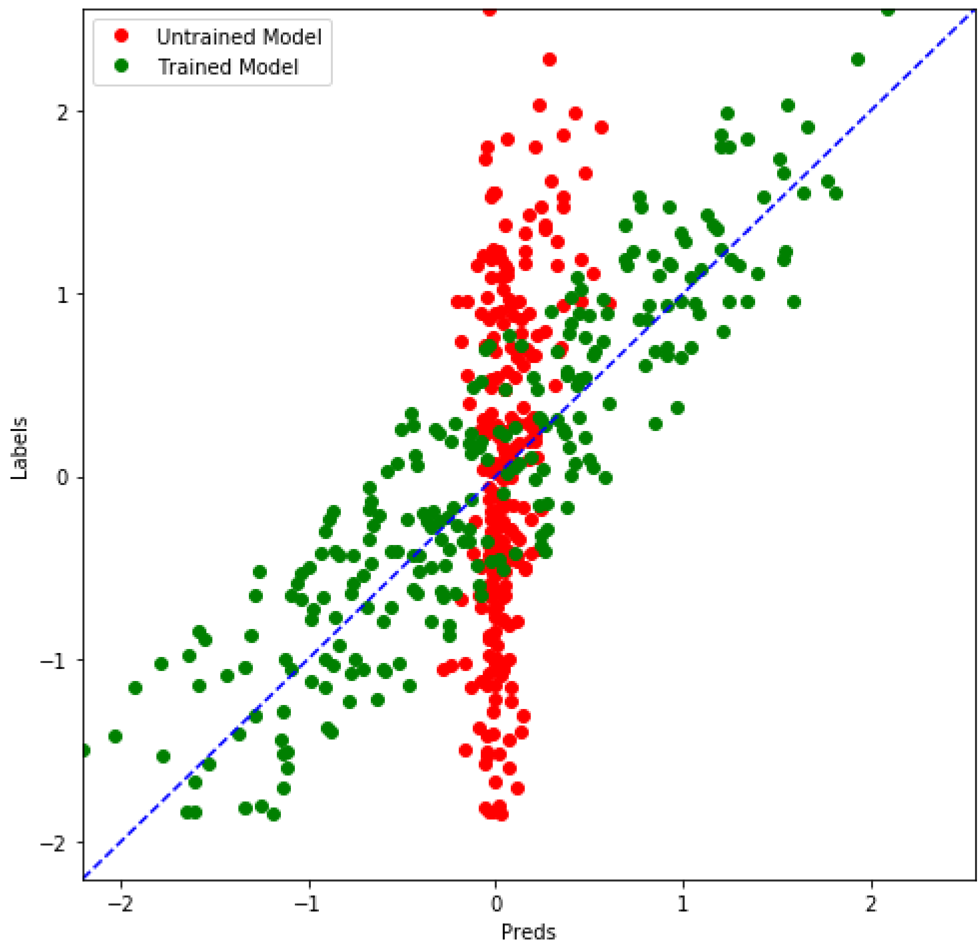
Predictions

7.1: Plot Raw Predictions

Let's use the `compare_predictions` helper function to compare predictions from the model when it was untrained and when it was trained.

In [20]:
```
preds_on_trained = model.predict(x_val)

compare_predictions(preds_on_untrained, preds_on_trained,y_val)
```



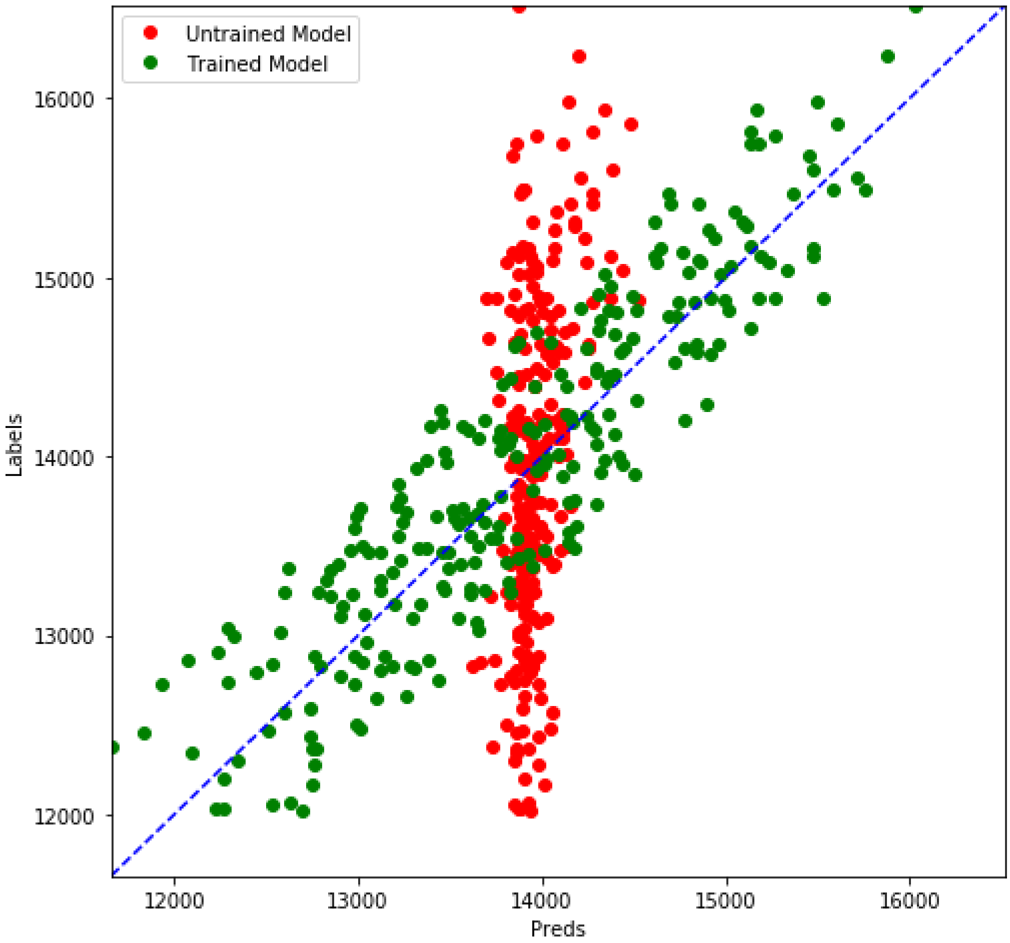
Insight : Untrained model makes random predictions which are all over the place,but it is a linear plot for the trained predictions. Surely, it has a high residual sum, but not as high as in the case of our untrained model

7.2: Plot Price Predictions

The plot for price predictions and raw predictions will look the same with just one difference: The x and y axis scale is changed.

In [21]:
```
price_untrained = [convert_label_value(y) for y in preds_on_untrained]
price_trained = [convert_label_value(y) for y in preds_on_trained]
price_val = [convert_label_value(y) for y in y_val]

compare_predictions(price_untrained, price_trained,price_val)
```



Insight : We pretty much get the same graph, but the ranges are now different and correspond to the original housing prices

In [ ]: