# **Problem Formulation**

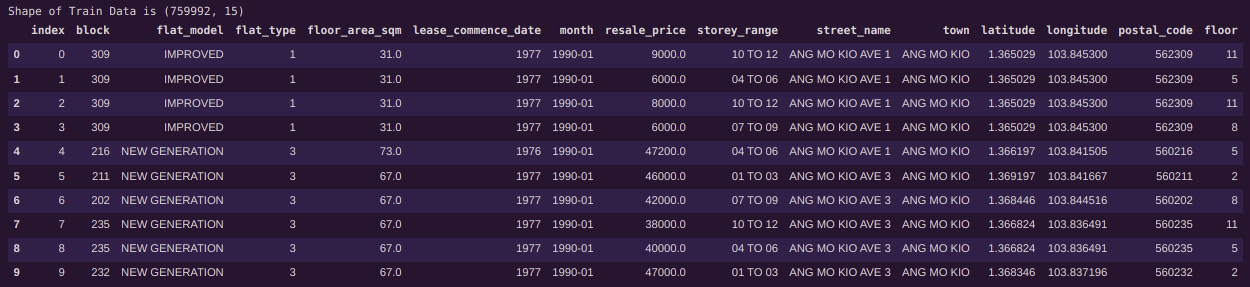
Traditional statistical techniques are being utilized for prediction of continuous values including Linear Regression and multi-variate regression. These techniques are serving the purpose since many decades. But there performance is limited beside feeding them with larger datasets as these models are not complex enough to learn all features inside data and generalize well on new test data. With the rise of large datasets available, and high performing GPU available for training Large neural networks; utility of big data and deep neural network has gain much attention since last decade. In this experiment we are conducting a *HousePricePrediction* experiment. This dataset is collected from real world stats of Singapore houses. It has two modes, *HDB* that contain information about houses, and *Private mode* that contain information about private houses. We’re utilizing HDB dataset. As fortunately, we have large dataset available of around 700K+ houses in HDB mode, it is expected to apply to some Deep Learning models to learn patterns out of this data and predict house price based on given input features. Convolutional Neural Network are much famous because of their capability of capturing non-linear patterns out of data. So instead of applying simple straight forward Fully Connected (Linear Layers) we are going to use CNN for feature extraction. As our input data in a vector of numbers, we are going to use CNN for feature extraction and some Fully Connected layers at end to predict regression value. We’re using *MeanSquaredError* as our loss function and *StochasticGradientDescent* as our model optimizer.

# **System/Algorithm Design (Just explain all the cells in details)**

# Data Preprocessing

As our input has vast variation in features and objects, we need to perform some analysis on it first. Here is attached a snap of dataset diagram loaded in using *pandas pd.read\_csv* function. It’s initial shape is 759K rows and each row has 15 columns.

As it is clearly visible, some columns were not adding any information in estimating target variable (*resale\_price*). We removed those columns at first:

Figure 1: Pandas Dataframe of initial data in raw form

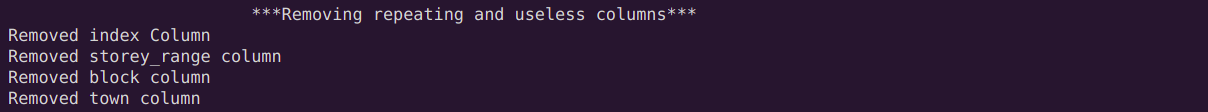
*index* → Just index of datapoints

*block* → Block number of house’s town

*storey\_range* → Repeating one – same information is more precisely available in floor

*town* → Repeating

We calculated Frequency table of *town* variable and plotted it. After getting it’s unique values, We tried to check either if street address do contain same information embedded in it; yes. So we deleted *town* variable to. Output of Preprocessing log is attached.

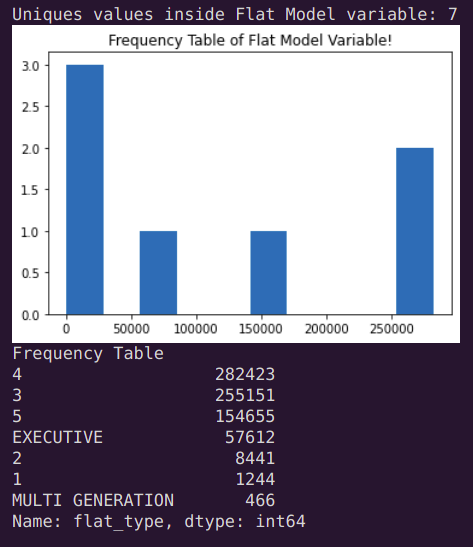
Figure 2: Column removed using *del* python command

Next step is to convert string word features to numeric ones, as machine learning model understand feature values only in numeric forms. We devised three strategies to convert string features into numeric ones and used them according to data of feature values.

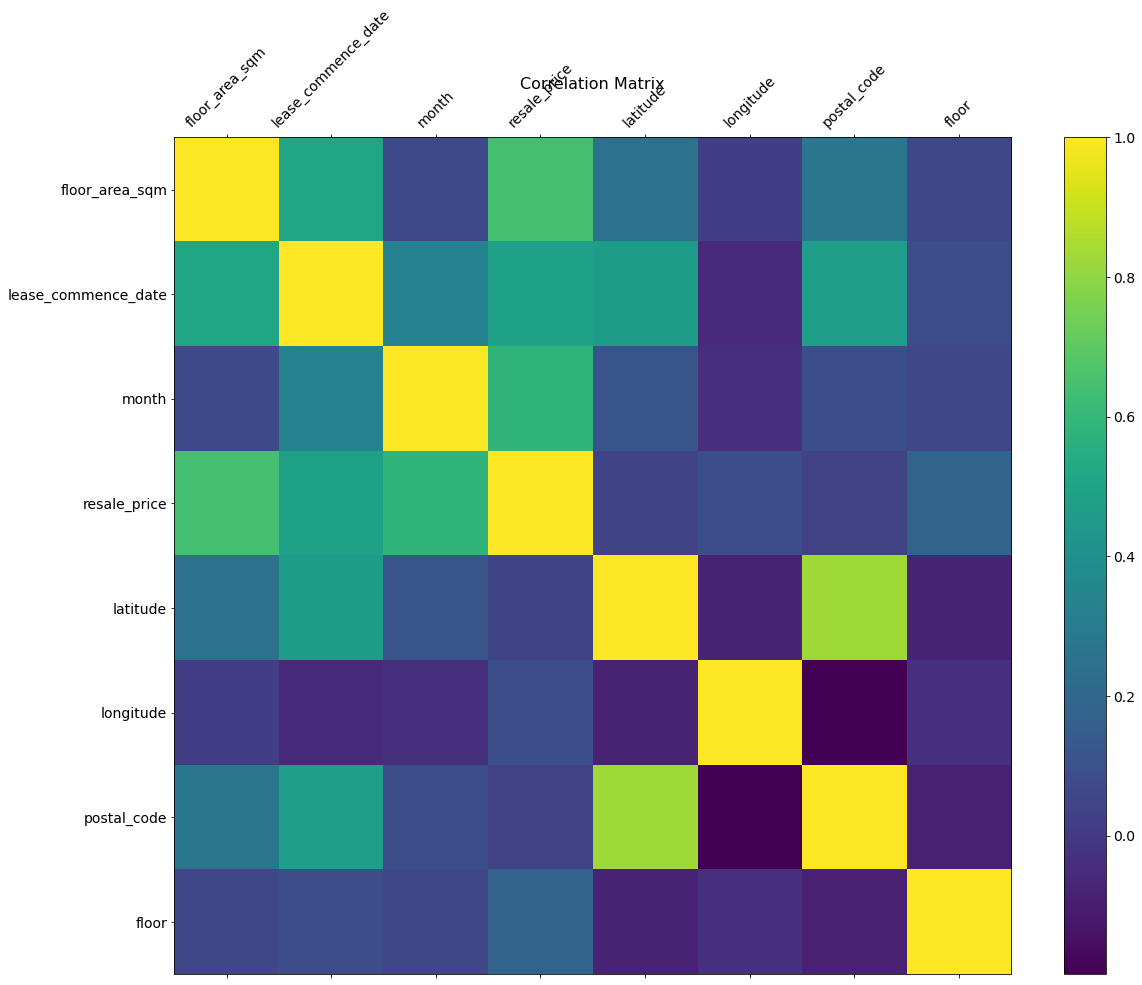
* Replace string with it’s ASCII code
* Replace string with ordinal integers
* One-hot encoding

Column *flat\_model* was converted to numeric values by replacing each data row point with it’s ASCII sum. Python provides a function named *ord()* which returns ASCII of single character. We got into a loop and summed up all ASCII codes of individual characters. Column *flat\_type* has ordinal coding e.g. 1,2,3 but it also has English keywords which were detected during Frequency table analysis. Snapshot is attached, these English words were replaced with ordinal coding 6 and 7.

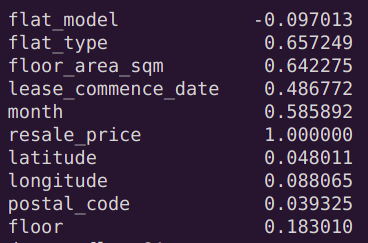
Executive and Multi Generation was replaced with 6 and 7 respectively! We also tried to convert *street\_num* variable to numeric ones using both a) replace by ASCII and b) One-hot encoding. But, in both cases, correlation score of *street\_num* variable was less than 0.0 . Some other features which has less than 0.0 impact of their variation on target variable were also removed. A snap of correlation matrix and score is being attached.

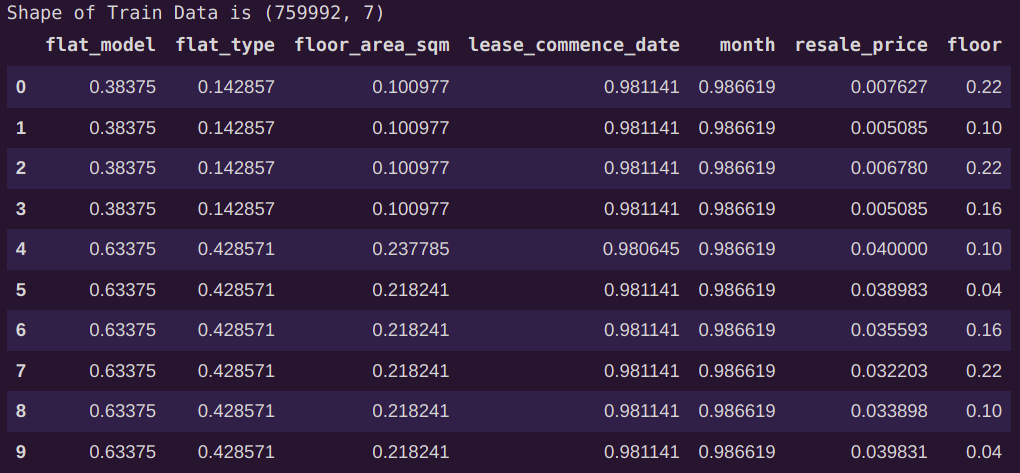
Figure 3: Frequency Table of *flat\_type* Column

Correlation of score of other features:

Figure 4: Correlation Matrix

Features with score less than 0.0 was removed. To scale all feature values between 0-1, max scaling was also done. Those max values were saved to local memory to be used during testing/evaluation process. After all kind of Preprocessing, new data looks like below data snap. It has left with only & features now.

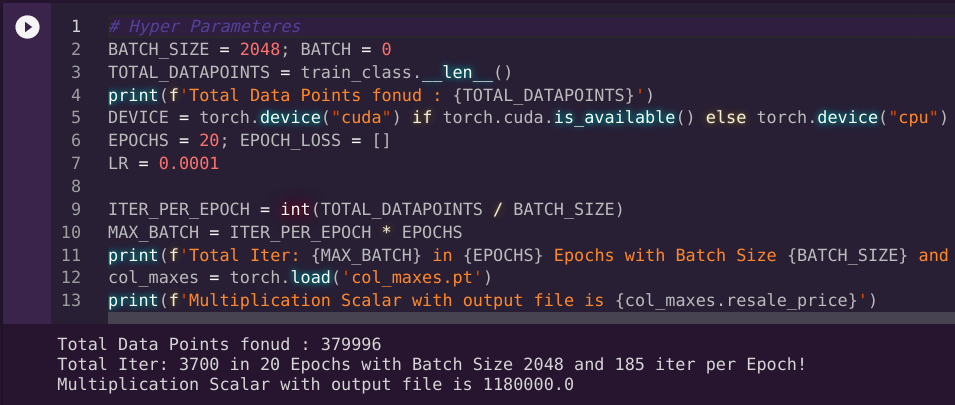
Figure 5: Numeric Representation of Correlation score taken with respect to target variable (*reslae\_price*)

Figure 6: Final Look of data after all Preprocessings

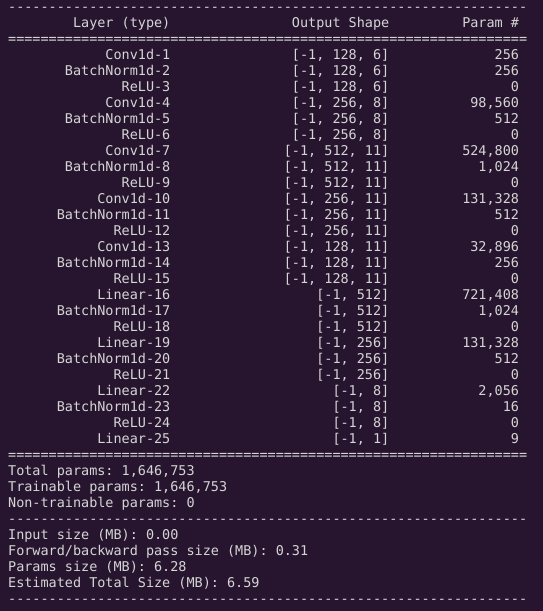
# DataClass and DataLoaders

I wrote a custom data class, and used Pytorch’s builtin Dataloader Class to load data into the model. I split data into train/test 50% each partition. Some hyper parameters snap has been also attached.

# Model

Figure 7: Hyper Parameters

I used model with 5 Convolutional Layers and 4 Fully Connected Linear layers. Each layers except of final output neuron layer; is being followed a *BatchNormalization layer* and *ReLU activation Function.* Summary of model is given below:

Figure 8: Summary of Model written in Pytorch framework

# **Experiments and Evaluation**

The below pasted code was used to for training the model and conducting core of this experiment. Vector of numeric values (data point) is given to 1-D Convolutional layer. Input data has 1 channel, convolutional layers applies a lot filter resulting in increasing number of channel for each single point. This helps model to learn non-linear patterns of data. Later 4 fully connected layer is also which are heading toward single neuron output.

model.train()

for EPOCH in range(EPOCHS): *# Run for 20 Epochs*

for ITER, (data, target) in enumerate(train\_loader):

optim.zero\_grad()

predictions = model(data.float().to(DEVICE)) *# get predictions form model*

target = target.view\_as(predictions) *# Reshape them*

loss = loss\_fn(predictions, target.float().to(DEVICE)) *# Calculate loss*

loss.float().backward() *# Launch backpropagation on loss*

optim.step() *# Change derivatives*

*# Print Evaluatoin Log after each EPOCH*

model.eval() *# Invoke evaluation mode of model*

local\_eval\_loss = [] *# List to store loss of each validation batch*

for valiter, (valdata, valtarget) in enumerate(test\_loader):

if valiter == 20: *# For sampling, just pass 20 batchs, it will save time*

break

valpred = model(valdata.float().to(DEVICE))

valtarget = valtarget.view\_as(valpred)

valloss = loss\_fn(valpred, valtarget.float().to(DEVICE))

local\_eval\_loss.append(valloss.item())

EPOCH\_LOSS.append(torch.mean(torch.tensor(local\_eval\_loss, dtype=torch.float))) *# Take mean of 20 batchs loss*

print(f'Test Data Loss: {EPOCH\_LOSS[-1]}')

model.train() *# Invoke training mode of model again*

if (EPOCH+1) % 5 == 0: *# Plot the Evaluation (unseen data loss) each 5 epoch*

plt.plot(EPOCH\_LOSS)

plt.xlabel('Epochs'); plt.ylabel('MSE LOSS')

plt.title('Model Performance!')

plt.show()

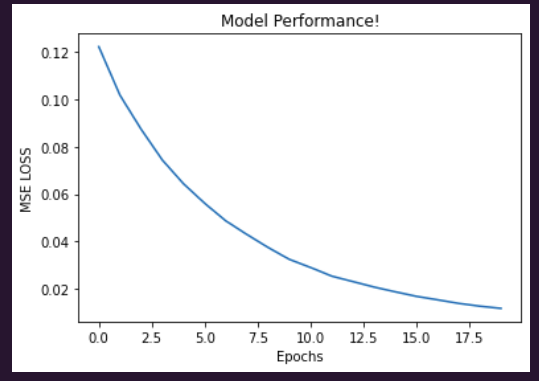
As you can see, whole prediction and loss calculation was done on test\_data which was unseen by model itself.

# **Results**

Our model worked well and produced much lesser error. Earlier error was high and then went to *nan*. On some inspections, I found out that between before calculating loss and after getting output from model, I was multiplying output with max\_value calculated earlier to scale up data as it was in original form. But, it produced huge number when going backward and all gradients tuned to huge number that is represented like Nan. We inspected gradients and came to solution of using scaled feature even for loss function.

We trained our model with 20 Epochs only and on evaluation test data, loss was dropped to 0.0188 on unseen test\_data. Here is the loss curve being shown:

# **Conclusion and Work Division**

Figure 9: CNN based Regressor

We first downloaded hdb dataset and loaded it using padnas csv file. Preprocessing of data involves removing unused column, converting string to numeric values, and feature scaling. In this case, we used max scaling. We also calculated correlation matirx with reference to target variable resale\_price. We removed features that has less than 0.0 impact score on target variable. After all kind of Preprocessing, we were left with 7 features including target variable. We wrote Custom dataclass to load data into memory, and then used pytorch’s buitlin dataloader class to load and feed data to model. We also wrote custom model class with 5 CNN and 4 FC layers. Training was done for 20 Epochs with Learning rate of 0.0001, optimizer as SGD, and Mean Squared Error as loss function. All the evaluations were done on test data which was unseen by model. At 20th Epoch, our loss was dropped to 0.0188 which was still decreasing.