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.

Shape of Train Data is (759992, 15)

	index	block	flat_model	flat_type	floor_area_sqm	lease_commence_date	month	resale_price	storey_range	street_name	town	latitude	longitude	postal_code	floor
0	0	309	IMPROVED	1	31.0	1977	1990-01	9000.0	10 TO 12	ANG MO KIO AVE 1	ANG MO KIO	1.365029	103.845300	562309	11
1	1	309	IMPROVED	1	31.0	1977	1990-01	6000.0	04 TO 06	ANG MO KIO AVE 1	ANG MO KIO	1.365029	103.845300	562309	5
2	2	309	IMPROVED	1	31.0	1977	1990-01	8000.0	10 TO 12	ANG MO KIO AVE 1	ANG MO KIO	1.365029	103.845300	562309	11
3	3	309	IMPROVED	1	31.0	1977	1990-01	6000.0	07 TO 09	ANG MO KIO AVE 1	ANG MO KIO	1.365029	103.845300	562309	8
4	4	216	NEW GENERATION	3	73.0	1976	1990-01	47200.0	04 TO 06	ANG MO KIO AVE 1	ANG MO KIO	1.366197	103.841505	560216	5
5	5	211	NEW GENERATION	3	67.0	1977	1990-01	46000.0	01 TO 03	ANG MO KIO AVE 3	ANG MO KIO	1.369197	103.841667	560211	2
6	6	202	NEW GENERATION	3	67.0	1977	1990-01	42000.0	07 TO 09	ANG MO KIO AVE 3	ANG MO KIO	1.368446	103.844516	560202	8
7	7	235	NEW GENERATION	3	67.0	1977	1990-01	38000.0	10 TO 12	ANG MO KIO AVE 3	ANG MO KIO	1.366824	103.836491	560235	11
8	8	235	NEW GENERATION	3	67.0	1977	1990-01	40000.0	04 TO 06	ANG MO KIO AVE 3	ANG MO KIO	1.366824	103.836491	560235	5
9	9	232	NEW GENERATION	3	67.0	1977	1990-01	47000.0	01 TO 03	ANG MO KIO AVE 3	ANG MO KIO	1.368346	103.837196	560232	2

Figure 1: Pandas Dataframe of initial data in raw form

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

`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.

```
                    ***Removing repeating and useless columns***
Removed index Column
Removed storey_range column
Removed block column
Removed town column
```

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.

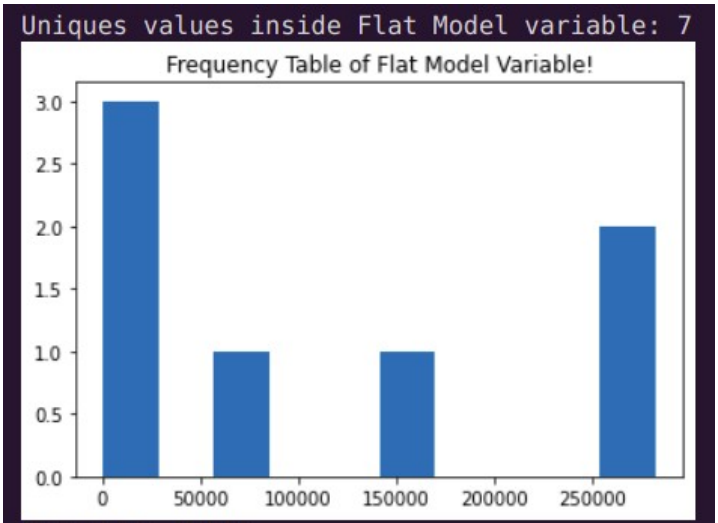


Figure 3: Frequency Table of `flat_type` Column

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.

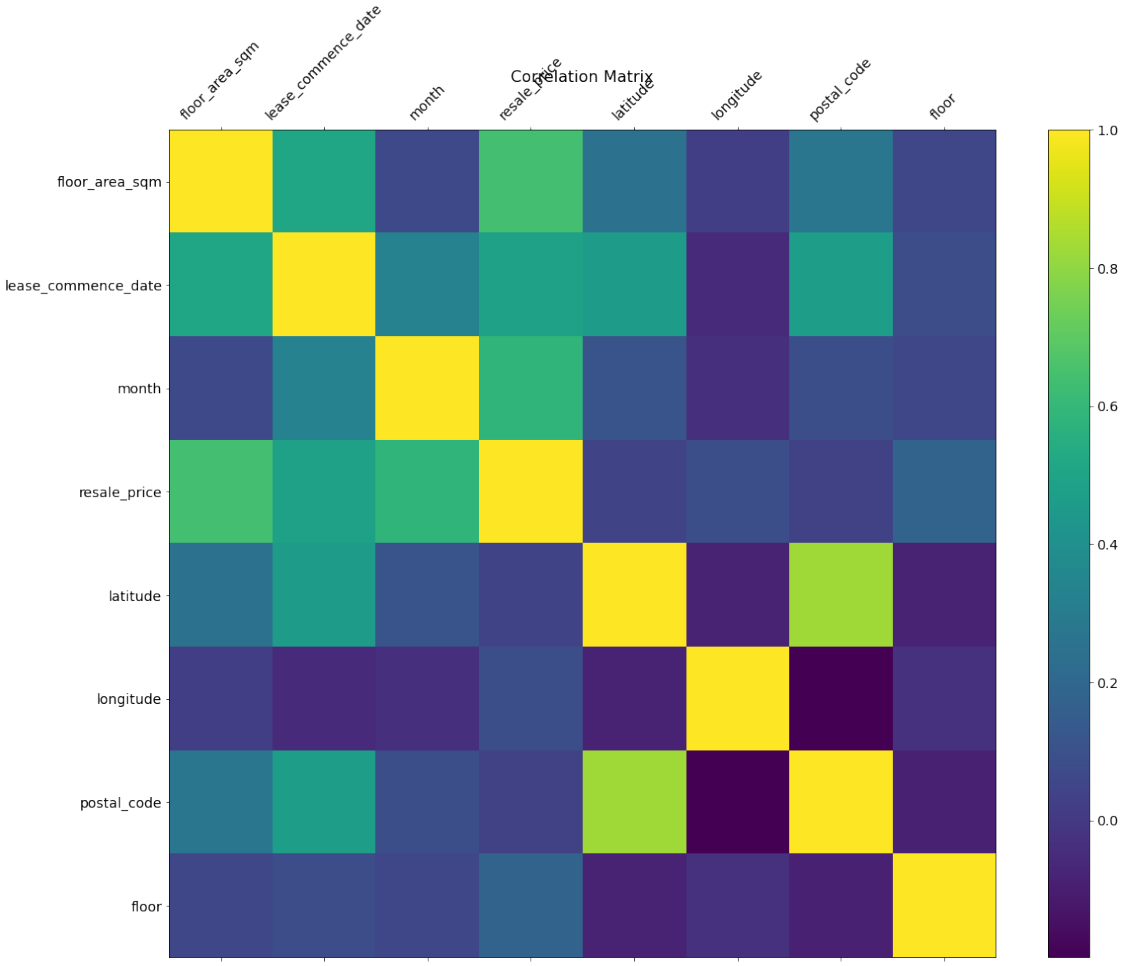


Figure 4: Correlation Matrix

Correlation of score of other features:

flat_model	-0.097013
flat_type	0.657249
floor_area_sqm	0.642275
lease_commence_date	0.486772
month	0.585892
resale_price	1.000000
latitude	0.048011
longitude	0.088065
postal_code	0.039325
floor	0.183010

Figure 5: Numeric Representation of Correlation score taken with respect to target variable (`reslae_price`)

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.

Shape of Train Data is (759992, 7)

	flat_model	flat_type	floor_area_sqm	lease_commence_date	month	resale_price	floor
0	0.38375	0.142857	0.100977	0.981141	0.986619	0.007627	0.22
1	0.38375	0.142857	0.100977	0.981141	0.986619	0.005085	0.10
2	0.38375	0.142857	0.100977	0.981141	0.986619	0.006780	0.22
3	0.38375	0.142857	0.100977	0.981141	0.986619	0.005085	0.16
4	0.63375	0.428571	0.237785	0.980645	0.986619	0.040000	0.10
5	0.63375	0.428571	0.218241	0.981141	0.986619	0.038983	0.04
6	0.63375	0.428571	0.218241	0.981141	0.986619	0.035593	0.16
7	0.63375	0.428571	0.218241	0.981141	0.986619	0.032203	0.22
8	0.63375	0.428571	0.218241	0.981141	0.986619	0.033898	0.10
9	0.63375	0.428571	0.218241	0.981141	0.986619	0.039831	0.04

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.

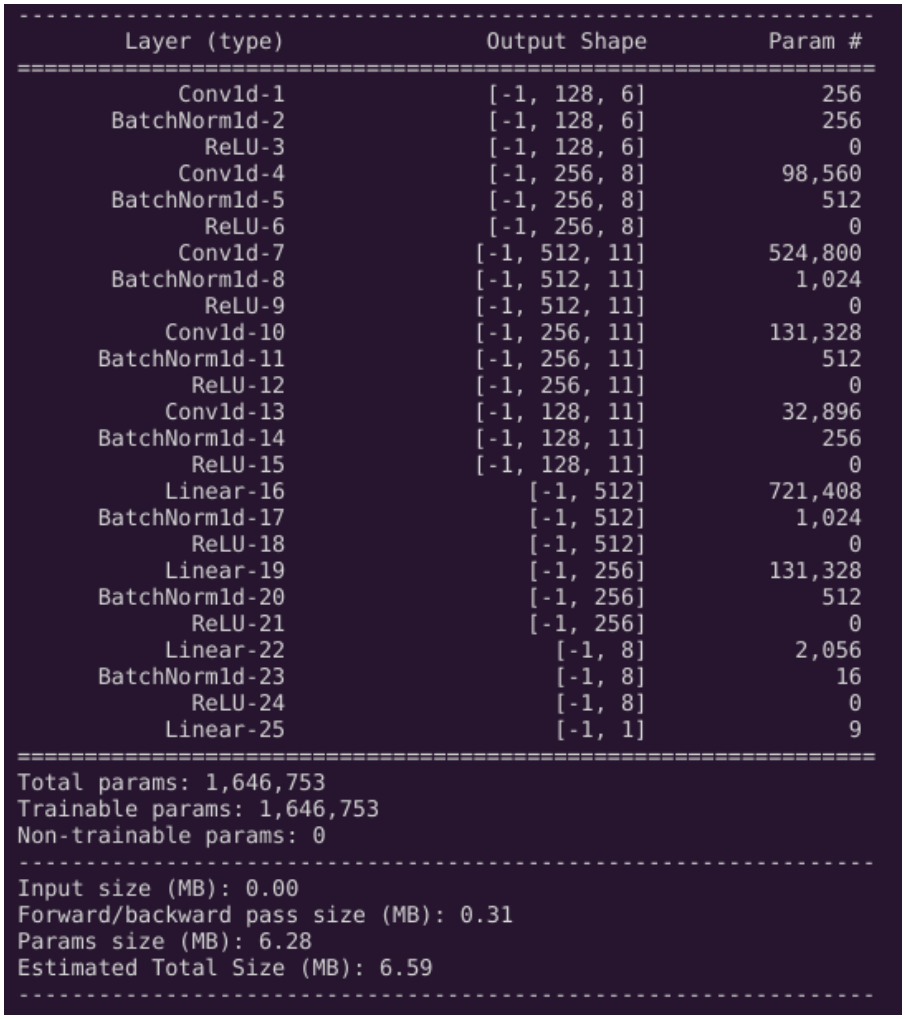
```
# Hyper Parameteres
BATCH_SIZE = 2048; BATCH = 0
TOTAL_DATAPOINTS = train_class.__len__()
print(f'Total Data Points fonud : {TOTAL_DATAPOINTS}')
DEVICE = torch.device("cuda") if torch.cuda.is_available() else torch.device("cpu")
EPOCHS = 20; EPOCH_LOSS = []
LR = 0.0001

ITER_PER_EPOCH = int(TOTAL_DATAPOINTS / BATCH_SIZE)
MAX_BATCH = ITER_PER_EPOCH * EPOCHS
print(f'Total Iter: {MAX_BATCH} in {EPOCHS} Epochs with Batch Size {BATCH_SIZE} and
col_maxes = torch.load('col_maxes.pt')
print(f'Multiplication Scalar with output file is {col_maxes.resale_price}')
```

```
Total Data Points fonud : 379996
Total Iter: 3700 in 20 Epochs with Batch Size 2048 and 185 iter per Epoch!
Multiplication Scalar with output file is 1180000.0
```

Figure 7: Hyper Parameters

Model

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:



Layer (type)	Output Shape	Param #
Conv1d-1	[-1, 128, 6]	256
BatchNorm1d-2	[-1, 128, 6]	256
ReLU-3	[-1, 128, 6]	0
Conv1d-4	[-1, 256, 8]	98,560
BatchNorm1d-5	[-1, 256, 8]	512
ReLU-6	[-1, 256, 8]	0
Conv1d-7	[-1, 512, 11]	524,800
BatchNorm1d-8	[-1, 512, 11]	1,024
ReLU-9	[-1, 512, 11]	0
Conv1d-10	[-1, 256, 11]	131,328
BatchNorm1d-11	[-1, 256, 11]	512
ReLU-12	[-1, 256, 11]	0
Conv1d-13	[-1, 128, 11]	32,896
BatchNorm1d-14	[-1, 128, 11]	256
ReLU-15	[-1, 128, 11]	0
Linear-16	[-1, 512]	721,408
BatchNorm1d-17	[-1, 512]	1,024
ReLU-18	[-1, 512]	0
Linear-19	[-1, 256]	131,328
BatchNorm1d-20	[-1, 256]	512
ReLU-21	[-1, 256]	0
Linear-22	[-1, 8]	2,056
BatchNorm1d-23	[-1, 8]	16
ReLU-24	[-1, 8]	0
Linear-25	[-1, 1]	9

Total params: 1,646,753
Trainable params: 1,646,753
Non-trainable params: 0

Input size (MB): 0.00
Forward/backward pass size (MB): 0.31
Params size (MB): 6.28
Estimated Total Size (MB): 6.59

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:

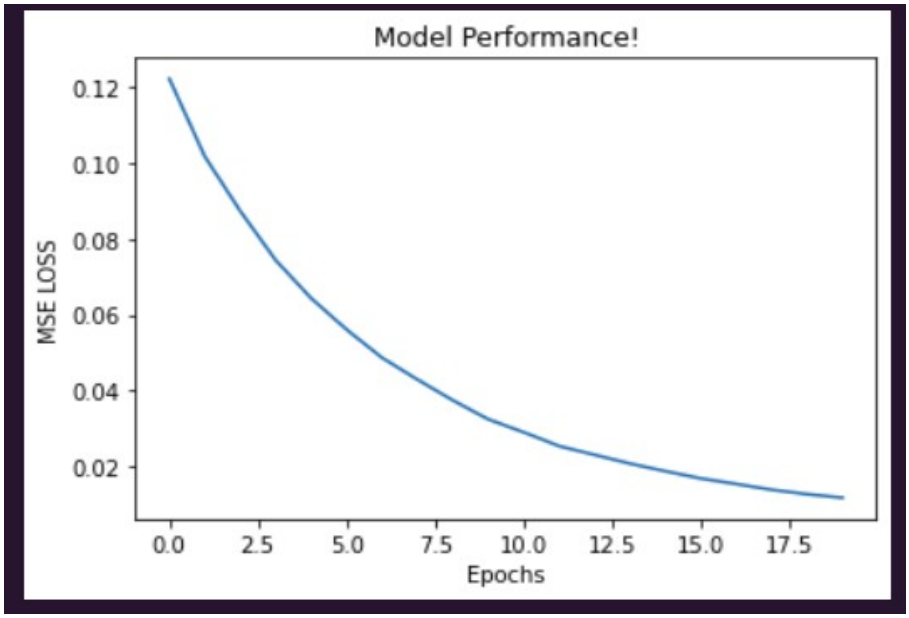


Figure 9: CNN based Regressor

Conclusion and Work Division

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 20^{th} Epoch, our loss was dropped to 0.0188 which was still decreasing.