Predict Housing Price Using Deep Learning Techniques

Olin Business School

Deep Learning for Prediction of Business Outcomes
Project Group 63

Group members:
Charlie Ling
Yue Wang

1. Abstract

The real estate market is one of the essential parts of the national economy, and the housing price is one of the critical elements of economic growth indicators. On the other hand, an accurate estimate of housing value can be beneficial to property owners and buyers, real estate agents, investors, and banks and financial institutions. However, an accurate prediction can be difficult, as the price of a property may depend on a wide variety of factors. This report illustrates how we could utilize deep learning techniques to predict a property's price using house features and textual data. We build a fully-connected neural network model based on house features and a combination of LSTM and CNN for predicting with textual information. In the end, we combine them together to further improve the model. The results showed that deep learning techniques in general have good performance in predicting housing prices. Including the textual information can help improve the model accuracy to some extent.

2. Introduction

According to the data published by Statista (2021), there were 6.5 million homes sold in the U.S. in 2020, and the number is expected to increase to 7.1 million by 2021. The number of home sales and the average sales price of new homes have both steadily risen since 2011. The market is enormous, and it keeps growing.

At the same time, investigating housing prices has great value for the real estate business. The value of a property is determined by various factors, such as the size, the location, the neighborhood, the condition of the house, etc. This project aims to take advantage of deep learning methods to better estimate the housing price. First, we will build a fully-connected deep neural network model based on the house features. Next, we will use LSTM and CNN methods to build a model based on textual data. In the end, we will combine them to see if we can get a relatively accurate model for housing price prediction.

The results of this study can be useful to both academicians and different parties involved in real estate transactions. Particularly, it can help provide references for real estate agents to list and find properties at a reasonable price. Therefore, when selling properties, the company could set appropriate prices to attract potential buyers. While buying properties, the company could identify reasonable prices based on house conditions so it can make cost-effective decisions.

3. Literature Review

Considering the importance of housing price prediction, it has been a popular research topic, especially in recent years. Researchers have utilized varieties of methods to try predicting housing prices across different countries. Empirical results show that machine learning and deep learning techniques tend to be more accurate and more efficient than regression models (Yazdani, 2021). The fail of regression models could be due to the non-linear relationship between house features and housing price, the lack of some environmental attributes, and the inadequate sample size.

Among all the methods used in previous studies, commonly used deep learning techniques are artificial neural networks (ANNs), recurrent neural networks (RNNs) based on long short-term memory (LSTM), and convolutional neural networks (CNNs). According to Yu et al. (2018), CNN is more accurate in predicting with characteristic factors, while LSTM has an excellent performance in predicting time series data. Nevertheless, model optimization processes are of great significance in building neural network models. The optimal neural network model needs to be created by a trial-and-error strategy with parameter tuning, or the results may not show superiority (Limsombunchai, 2004).

In addition, some studies have not only considered house attributes but also included other kinds of data, such as spatial context, images, and satellite maps. However, few studies took textual data into consideration. Sellers' descriptions of the property might contain important information about the

conditions of the house that is not shown in the house attributes. Zhou et al. (2019) found that including textual information can improve the accuracy of the rental price prediction model. In this paper, we would like to apply similar methods to selling price prediction. We will examine how much the textual information could contribute to predicting the selling price of a property and see if it could help us build a more accurate model.

4. Problem Description

The goal of this project is to build a housing price prediction model with higher accuracy. Compared to commonly used models, our model will include textual data which is easy to collect but can be informative in price prediction. The model will be built in three steps. First, we will use fully-connected neural networks to build a model based on the house features we selected. Next, we will use LSTM and CNN methods to build models based on textual data. In the end, we will combine the models we built in previous steps and examine its performance.

5. Dataset Description & Data Preprocessing

5.1. Dataset Description

The dataset we used represents a sample of homes in Atlanta, GA that were listed for sale within the past year and still actively listed on 4/29/2021. Data is sourced from the local MLS (multiple listing service). The dataset has 8168 rows, each row represents a property. It does not have time-series data but only the information listed at the time the data was collected.

The dataset contains both numerical house attributes and textual data for us to do analysis. Our dependent variable is the list price for the property. The house attributes include the size, the location, the number of rooms, the year built, etc. The textual data is the seller's description of the property. The dataset contains not only single-family homes but also townhouses and condos. The list price has a wide range from $9,999 to $13,950,000, and a majority of the houses lie in the range of $100,000-$600,000.

5.2. Data Preprocessing

This dataset has 60 columns, most of which are hard to use. To prepare the data for modeling, the first step is to choose those most useful attributes. We choose 'list_price' as output variable, and 'lotsize_sqft', 'zip', 'beds', 'baths_full', 'baths_half', 'sqft', 'property_type', 'year_built' as input variables. Variable descriptions are listed in Table 1. Next, we merge the similar values in ‘property_type’, like full names and shorthand, lower and upper case. Then, we use one-hot encoding to transform zip codes and property types into dummy variables so that they can be used as input variables.

In addition, there are some null values in the data. To simplify the problem, we dropped the rows with NAs, so the actual sample size we used to build our model is 7252. Considering the sample size, we divide the dataset into 60% for training, 20% for validation, and the last 20% for testing.

6. Model Description

We construct two models for this project: one is a simple fully-connected dense model, the other is a pre-trained word embedding.

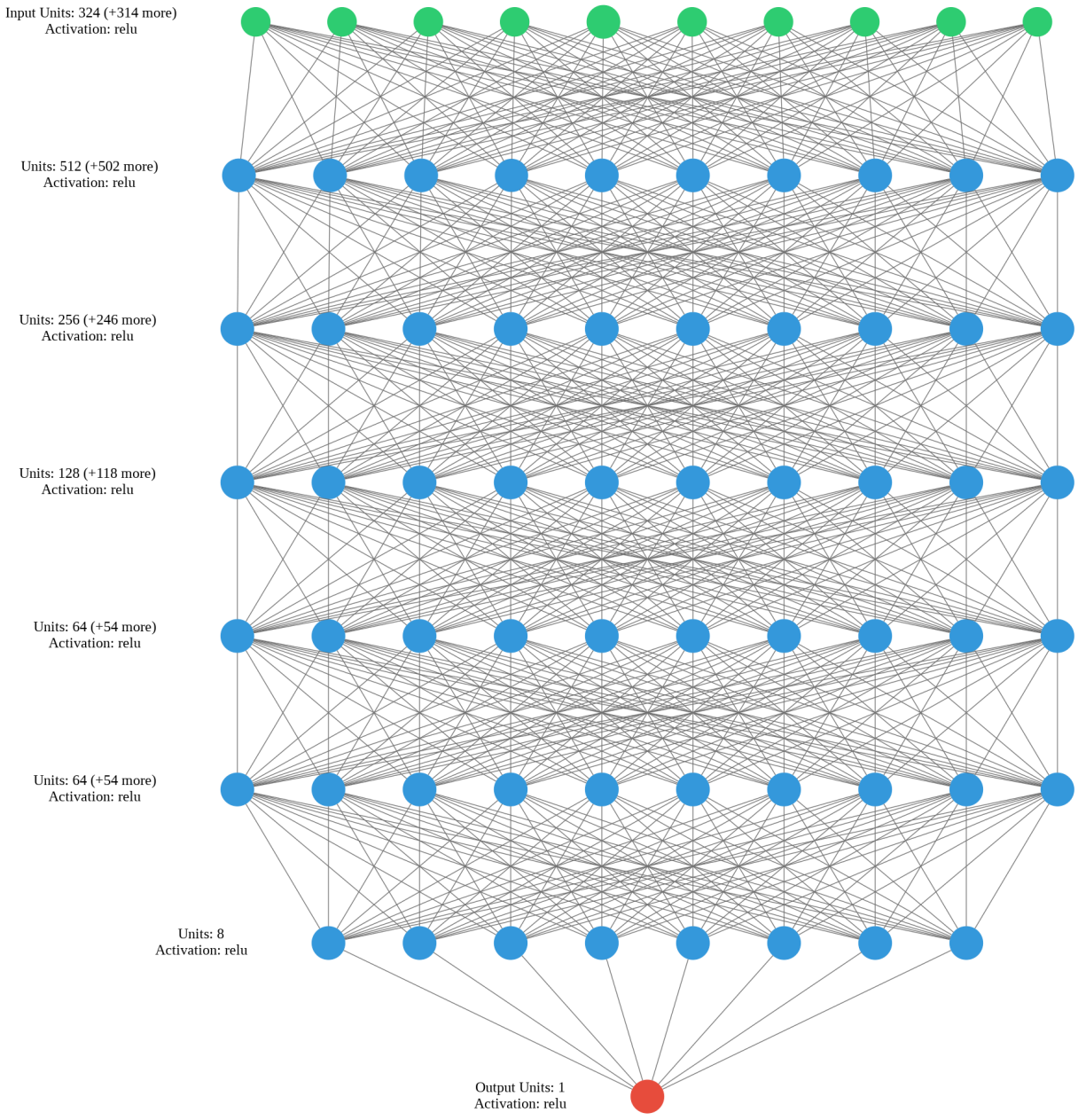
6.1. Fully-connected Neural Network

Because the attributes we choose are numerical and categorical variables, it is suitable to build a simple fully-connected dense model. Dense layer is a basic and powerful structure that can converge to

any nonlinear function. We can use a dense model to find the nonlinear relationship between list price and lotsize_sqft, zip, beds, baths_full, baths_half, sqft, property_type, year_built. While CNN and RNN are used to deal with more complex data types like photos and text, there is no need to use these models to deal with numerical and categorical variables.

After we transform the categorical variables to dummy variables, we use max-min scaling to standardize those numerical variables. Then we can easily use the dense model.

Our initial dense model is like this:



We use 'lotsize_sqft', 'beds', 'baths_full', 'baths_half', 'sqft', 'year_built' and dummy variables for 'zip' and 'property_type' in the input layer. Through several hidden layers using the Relu activation function, we find the mapping to the output variable: list price.

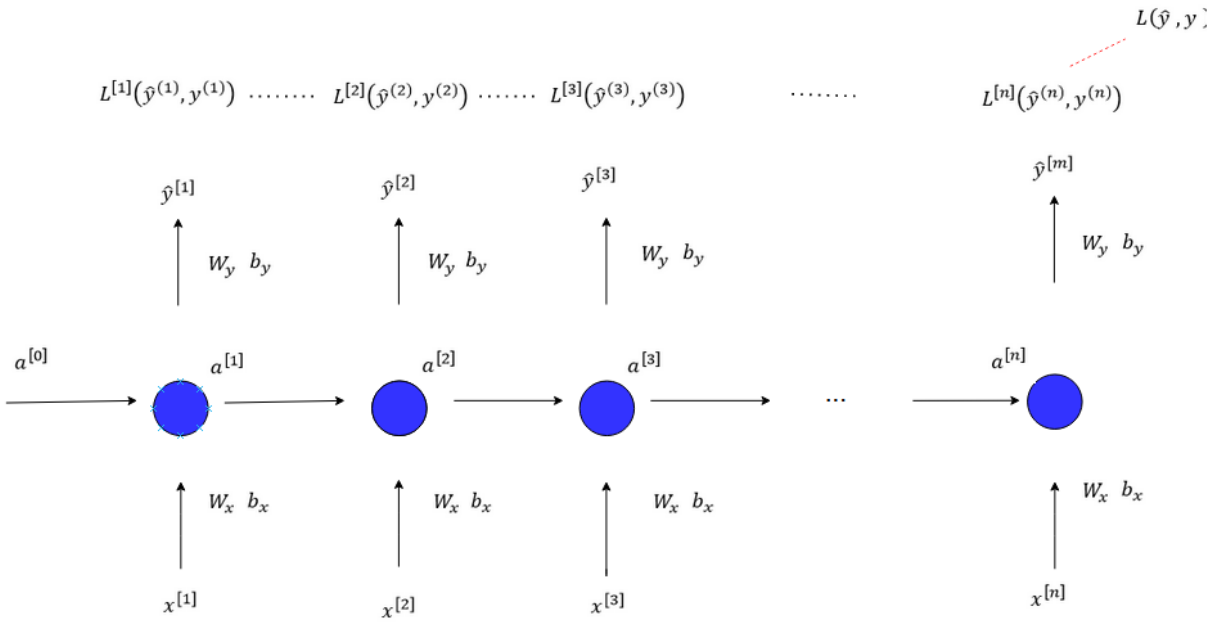
The drawback of this dense model is that we only use a few features in this dataset. A lot of useful information like descriptions and pictures are ignored. Maybe we should find a way to transform more features into dummy variables. But before that, we need to do a lot of data cleaning.

6.2. Pre-trained Word Embedding

We also want to make use of descriptions of those properties, so we need to do text mining. We use descriptions to build a pre-trained word embedding model to learn the position of words in text so that we can extract information from text data. We cannot use Full Connected Networks to deal with text data because inputs and outputs can have different sizes, and it is too complex to use classic NNs.

Firstly, we delete numbers in the text and tokenize the text. Then we use pad_sequences to turn our lists of integers into a 2D integer tensor. Besides, we label list price as 4 groups and transfer list price into a categorical variable so that we can use this label as the output variable for the pre-trained word embedding model.

Initially, we combined Pre-trained Word Embedding with simple RNN. We specify the maximum input length to our Embedding layer so we can later flatten the embedded inputs. After the Embedding layer, we add 3 simple RNN layers as hidden layers and one dense layer as output layer.



$$Z^{[1]} = W^{[1]}a^{[0]} + b^{[1]}, \qquad a^{[1]} = g(Z^{[1]})$$

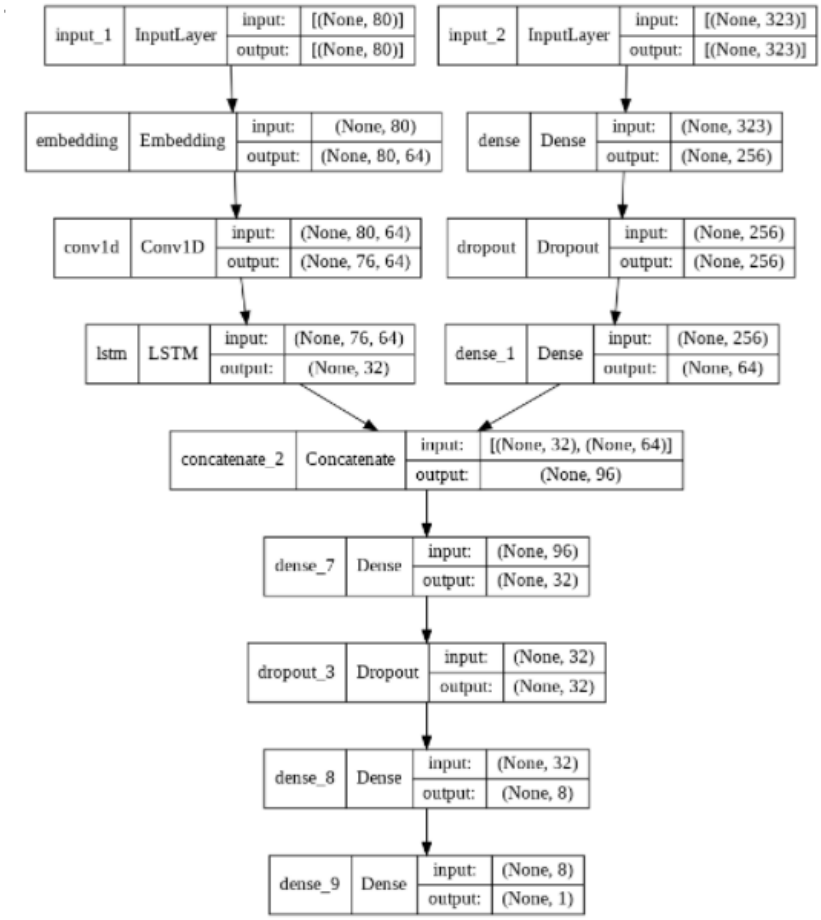
$$L^{[t]}(\hat{y}^{(t)}, y^{(t)}) = -y^{(t)} \log \hat{y}^t - (1 - y^t) \log (1 - \hat{y}^t)$$

$$L(\hat{y}, y) = \sum_{t=1}^{n} L^{[t]}(\hat{y}^{(t)}, y^{(t)})$$

This Pre-trained Word Embedding has an underfitting problem because we only use the text description of properties, which is far from enough. Maybe we can try to combine this word embedding model with the previous dense model so we can make use of more information.

6.3. Combination of the Two Models

In order to further improve our model, we combined the two models we developed in previous steps. First, the two models are developed separately. Then, we add a concatenate layer to combine the two models.



7. Experimental Results

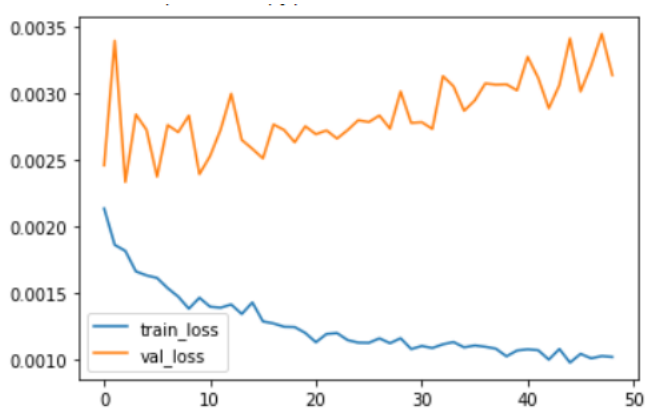
7.1. Fully-connected Neural Network

This is the structure of the initial dense model:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_4 (Dense)              (None, 512)               166400

dense_5 (Dense)              (None, 256)               131328

dense_6 (Dense)              (None, 128)               32896

dense_7 (Dense)              (None, 64)                8256

dense_8 (Dense)              (None, 64)                4160

dense_9 (Dense)              (None, 8)                 520

dense_10 (Dense)             (None, 1)                 9

=================================================================
Total params: 343,569
Trainable params: 343,569
Non-trainable params: 0
_________________________________________________________________
```

It is with 5 hidden layers, and the result is as below:

```
Epoch 40/50
272/272 [==============================] - 2s 8ms/step - loss: 0.0010 - val_loss: 0.0031
Epoch 41/50
272/272 [==============================] - 2s 8ms/step - loss: 0.0011 - val_loss: 0.0030
Epoch 42/50
272/272 [==============================] - 2s 8ms/step - loss: 0.0011 - val_loss: 0.0033
Epoch 43/50
272/272 [==============================] - 2s 8ms/step - loss: 0.0011 - val_loss: 0.0031
Epoch 44/50
272/272 [==============================] - 2s 8ms/step - loss: 9.9771e-04 - val_loss: 0.0029
Epoch 45/50
272/272 [==============================] - 2s 8ms/step - loss: 0.0011 - val_loss: 0.0031
Epoch 46/50
272/272 [==============================] - 2s 8ms/step - loss: 9.7481e-04 - val_loss: 0.0034
Epoch 47/50
272/272 [==============================] - 2s 8ms/step - loss: 0.0010 - val_loss: 0.0030
Epoch 48/50
272/272 [==============================] - 2s 8ms/step - loss: 0.0010 - val_loss: 0.0032
Epoch 49/50
272/272 [==============================] - 2s 8ms/step - loss: 0.0010 - val_loss: 0.0034
Epoch 50/50
272/272 [==============================] - 2s 8ms/step - loss: 0.0010 - val_loss: 0.0031
```



Training loss is smaller than validation loss, but the validation loss is with a lot of oscillation and the distance is increasing, so this is an overfitting problem.

Then we use L2 regularization to solve the overfitting problem, and reduce hidden layers and nodes, and change the activation function of the output layer from relu to sigmoid:

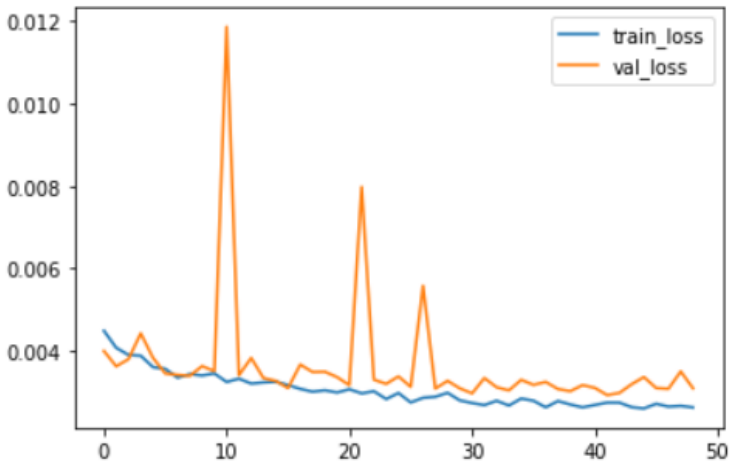
```
from keras.layers import Dense
from keras.layers import Dropout
model=Sequential()
# L2 regularization applied at the first hidden layer
model.add(Dense(256,input_shape=(X_train.shape[1],), activation='relu', kernel_regularizer="12"))
model.add(Dense(64, activation='relu'))
model.add(Dense(32, activation='relu'))
model.add(Dense(8, activation='relu'))
model.add(Dense(1,activation="sigmoid"))
model.compile(optimizer='rmsprop', loss='mean_squared_error')
model.summary()
```

```
Model: "sequential_5"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_21 (Dense)            (None, 256)               83200

 dense_22 (Dense)            (None, 64)                16448

 dense_23 (Dense)            (None, 32)                2080

 dense_24 (Dense)            (None, 8)                 264

 dense_25 (Dense)            (None, 1)                 9

=================================================================
Total params: 102,001
Trainable params: 102,001
Non-trainable params: 0
_________________________________________________________________
```

And this is the new result:

```
Epoch 41/50
272/272 [==============================] - 1s 4ms/step - loss: 0.0026 - val_loss: 0.0032
Epoch 42/50
272/272 [==============================] - 1s 4ms/step - loss: 0.0027 - val_loss: 0.0031
Epoch 43/50
272/272 [==============================] - 1s 4ms/step - loss: 0.0027 - val_loss: 0.0029
Epoch 44/50
272/272 [==============================] - 1s 4ms/step - loss: 0.0027 - val_loss: 0.0030
Epoch 45/50
272/272 [==============================] - 1s 4ms/step - loss: 0.0026 - val_loss: 0.0032
Epoch 46/50
272/272 [==============================] - 1s 4ms/step - loss: 0.0026 - val_loss: 0.0034
Epoch 47/50
272/272 [==============================] - 1s 4ms/step - loss: 0.0027 - val_loss: 0.0031
Epoch 48/50
272/272 [==============================] - 1s 4ms/step - loss: 0.0026 - val_loss: 0.0031
Epoch 49/50
272/272 [==============================] - 1s 4ms/step - loss: 0.0027 - val_loss: 0.0035
Epoch 50/50
272/272 [==============================] - 1s 4ms/step - loss: 0.0026 - val_loss: 0.0031
```


```
Epoch 41/50
272/272 [==============================] - 1s 4ms/step - loss: 0.0026 - val_loss: 0.0032
Epoch 42/50
272/272 [==============================] - 1s 4ms/step - loss: 0.0027 - val_loss: 0.0031
Epoch 43/50
272/272 [==============================] - 1s 4ms/step - loss: 0.0027 - val_loss: 0.0029
Epoch 44/50
272/272 [==============================] - 1s 4ms/step - loss: 0.0027 - val_loss: 0.0030
Epoch 45/50
272/272 [==============================] - 1s 4ms/step - loss: 0.0026 - val_loss: 0.0032
Epoch 46/50
272/272 [==============================] - 1s 4ms/step - loss: 0.0026 - val_loss: 0.0034
Epoch 47/50
272/272 [==============================] - 1s 4ms/step - loss: 0.0027 - val_loss: 0.0031
Epoch 48/50
272/272 [==============================] - 1s 4ms/step - loss: 0.0026 - val_loss: 0.0031
Epoch 49/50
272/272 [==============================] - 1s 4ms/step - loss: 0.0027 - val_loss: 0.0035
Epoch 50/50
272/272 [==============================] - 1s 4ms/step - loss: 0.0026 - val_loss: 0.0031
```

Training loss is smaller than validation loss, but the validation loss is with a lot of oscillation, so this is still an overfitting problem.

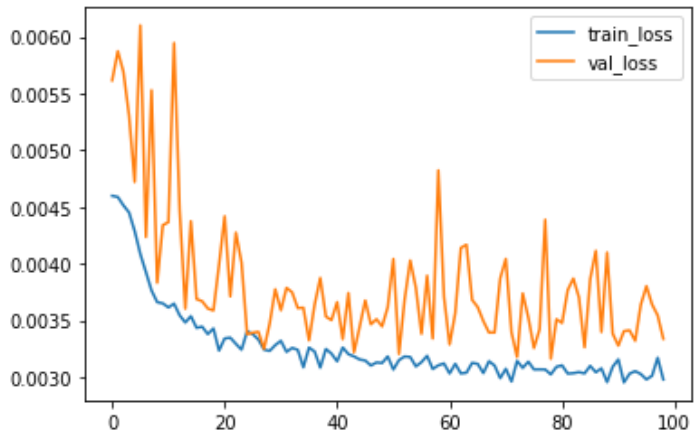
Finally, we use L2, Dropout, Mini-Batch Gradient Descent, RMSprop to improve the results:

```
model=Sequential()
model.add(Dense(256,input_shape=(X_train.shape[1],),  activation='relu',  kernel_regularizer="l2"))
model.add(Dropout(0.15))
model.add(Dense(64,  activation='relu'))
model.add(Dropout(0.15))
#model.add(Dense(32,  activation='relu'))
model.add(Dense(8,  activation='relu'))
model.add(Dense(1,activation="sigmoid"))
model.compile(optimizer='rmsprop',  loss='mean_squared_error')
model.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_11 (Dense)             (None, 256)               83200

dropout_2 (Dropout)          (None, 256)               0

dense_12 (Dense)             (None, 64)                16448

dropout_3 (Dropout)          (None, 64)                0

dense_13 (Dense)             (None, 8)                 520

dense_14 (Dense)             (None, 1)                 9

=================================================================
Total params: 100,177
Trainable params: 100,177
Non-trainable params: 0
_________________________________________________________________
```

This is the final result:

```
Epoch 90/100
272/272 [==============================] - 1s 4ms/step - loss: 0.0030 - val_loss: 0.0041
Epoch 91/100
272/272 [==============================] - 1s 4ms/step - loss: 0.0031 - val_loss: 0.0034
Epoch 92/100
272/272 [==============================] - 1s 4ms/step - loss: 0.0032 - val_loss: 0.0033
Epoch 93/100
272/272 [==============================] - 1s 4ms/step - loss: 0.0030 - val_loss: 0.0034
Epoch 94/100
272/272 [==============================] - 1s 4ms/step - loss: 0.0030 - val_loss: 0.0034
Epoch 95/100
272/272 [==============================] - 1s 4ms/step - loss: 0.0031 - val_loss: 0.0033
Epoch 96/100
272/272 [==============================] - 1s 4ms/step - loss: 0.0030 - val_loss: 0.0036
Epoch 97/100
272/272 [==============================] - 1s 4ms/step - loss: 0.0030 - val_loss: 0.0038
Epoch 98/100
272/272 [==============================] - 1s 4ms/step - loss: 0.0030 - val_loss: 0.0036
Epoch 99/100
272/272 [==============================] - 1s 4ms/step - loss: 0.0032 - val_loss: 0.0035
Epoch 100/100
272/272 [==============================] - 1s 4ms/step - loss: 0.0030 - val_loss: 0.0033
```



Training loss is smaller than validation loss, and the distance between training and validation loss is small, which means this model works well.

Then we save this model. When we meet similar programs, we can just use this architecture.

```
model.save('/content/densemodel.h5')
```

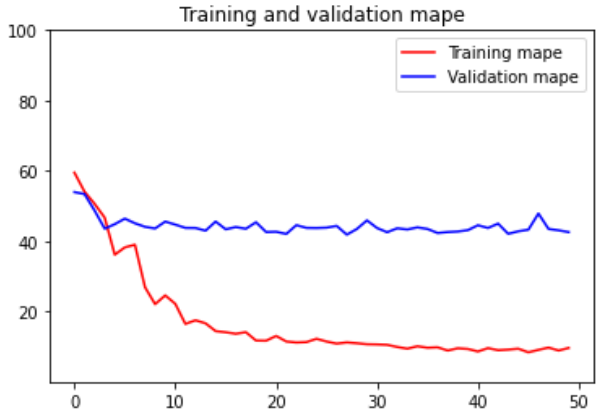
7.2. Pre-trained Word Embedding

This is the structure of the initial Pre-trained Word Embedding model:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (None, 20, 64)            640000

simple_rnn (SimpleRNN)       (None, 20, 32)            3104

simple_rnn_1 (SimpleRNN)     (None, 20, 16)            784

simple_rnn_2 (SimpleRNN)     (None, 32)                1568

dense (Dense)                (None, 1)                 33

=================================================================
Total params: 645,489
Trainable params: 645,489
Non-trainable params: 0
_________________________________________________________________
```

And this is the result:

```
Epoch 40/50
64/64 [==============================] - 2s 27ms/step - loss: 0.0603 - mape: 9.3593 - val_loss: 1.0549 - val_mape: 43.1366
Epoch 41/50
64/64 [==============================] - 3s 40ms/step - loss: 0.0586 - mape: 8.6498 - val_loss: 1.0684 - val_mape: 44.5329
Epoch 42/50
64/64 [==============================] - 3s 42ms/step - loss: 0.0715 - mape: 9.5956 - val_loss: 1.0565 - val_mape: 43.7643
Epoch 43/50
64/64 [==============================] - 2s 26ms/step - loss: 0.0646 - mape: 9.0120 - val_loss: 1.0498 - val_mape: 44.9988
Epoch 44/50
64/64 [==============================] - 2s 25ms/step - loss: 0.0655 - mape: 9.1763 - val_loss: 1.0722 - val_mape: 42.0966
Epoch 45/50
64/64 [==============================] - 2s 24ms/step - loss: 0.0704 - mape: 9.4357 - val_loss: 1.0698 - val_mape: 42.8147
Epoch 46/50
64/64 [==============================] - 2s 26ms/step - loss: 0.0547 - mape: 8.4127 - val_loss: 1.0599 - val_mape: 43.2697
Epoch 47/50
64/64 [==============================] - 3s 41ms/step - loss: 0.0718 - mape: 9.1061 - val_loss: 1.0841 - val_mape: 47.8737
Epoch 48/50
64/64 [==============================] - 3s 41ms/step - loss: 0.0687 - mape: 9.7496 - val_loss: 1.0556 - val_mape: 43.4825
Epoch 49/50
64/64 [==============================] - 3s 48ms/step - loss: 0.0639 - mape: 8.9221 - val_loss: 1.0717 - val_mape: 43.0912
Epoch 50/50
64/64 [==============================] - 3s 46ms/step - loss: 0.0675 - mape: 9.6444 - val_loss: 1.0674 - val_mape: 42.5881
```



Training and validation mape are both high, and the distance is increasing, so we have underfitting and overfitting problems at the same time.

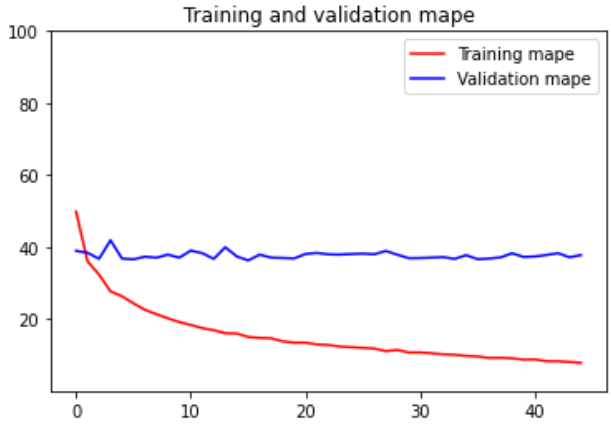
Then we try Regularization with LSTM:

```
model = Sequential()
model.add(Embedding(max_features, 64, input_length=maxlen))
model.add(layers.LSTM(32,dropout=0.2,recurrent_dropout=0.2))
model.add(layers.Dense(1))

model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding (Embedding)       (None, 20, 64)            640000

 lstm (LSTM)                 (None, 32)                12416

 dense (Dense)               (None, 1)                 33

=================================================================
Total params: 652,449
Trainable params: 652,449
Non-trainable params: 0
_________________________________________________________________
```

This is the new result:

```
Epoch 40/50
256/256 [==============================] - 9s 34ms/step - loss: 0.1645 - mape: 8.7122 - val_loss: 0.6871 - val_mape: 37.2464
Epoch 41/50
256/256 [==============================] - 9s 35ms/step - loss: 0.1628 - mape: 8.7636 - val_loss: 0.6987 - val_mape: 37.3757
Epoch 42/50
256/256 [==============================] - 9s 34ms/step - loss: 0.1565 - mape: 8.2673 - val_loss: 0.7042 - val_mape: 37.7791
Epoch 43/50
256/256 [==============================] - 9s 34ms/step - loss: 0.1554 - mape: 8.2587 - val_loss: 0.7124 - val_mape: 38.2863
Epoch 44/50
256/256 [==============================] - 9s 34ms/step - loss: 0.1512 - mape: 8.0549 - val_loss: 0.7094 - val_mape: 37.1497
Epoch 45/50
235/256 [==========================>...] - ETA: 0s - loss: 0.1468 - mape: 7.8359WARNING:tensorflow:Your input ran out of data; interru
256/256 [==============================] - 8s 31ms/step - loss: 0.1468 - mape: 7.8273 - val_loss: 0.7170 - val_mape: 37.7585
```



Training and validation mape decrease a little, but we still have underfitting and overfitting problems.

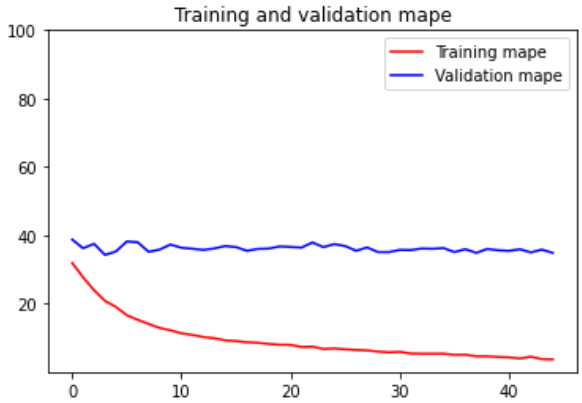
Now we use Bidirectional model:

```
model = Sequential()
model.add(Embedding(max_features, 64, input_length=maxlen))
model.add(layers.Conv1D(64, 5, activation='relu'))
model.add(layers.Bidirectional(
        layers.LSTM(64,dropout=0.2,recurrent_dropout=0.2)))
model.add(layers.Dense(1))

model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding (Embedding)       (None, 20, 64)            640000

 conv1d (Conv1D)             (None, 16, 64)            20544

 bidirectional (Bidirectiona  (None, 128)              66048
 l)

 dense (Dense)               (None, 1)                 129

=================================================================
Total params: 726,721
Trainable params: 726,721
Non-trainable params: 0
_________________________________________________________________
```

This is the new result:

```
Epoch 40/50
256/256 [==============================] - 17s 65ms/step - loss: 0.0865 - mape: 4.4021 - val_loss: 0.6656 - val_mape: 35.6071
Epoch 41/50
256/256 [==============================] - 17s 64ms/step - loss: 0.0815 - mape: 4.2745 - val_loss: 0.6574 - val_mape: 35.4377
Epoch 42/50
256/256 [==============================] - 17s 65ms/step - loss: 0.0753 - mape: 3.9469 - val_loss: 0.6679 - val_mape: 35.9051
Epoch 43/50
256/256 [==============================] - 17s 65ms/step - loss: 0.0813 - mape: 4.4662 - val_loss: 0.6789 - val_mape: 34.9807
Epoch 44/50
256/256 [==============================] - 17s 65ms/step - loss: 0.0728 - mape: 3.7637 - val_loss: 0.6905 - val_mape: 35.7876
Epoch 45/50
236/256 [==========================>...] - ETA: 1s - loss: 0.0689 - mape: 3.7053WARNING:tensorflow:Your input ran out of data; interrupting training.
256/256 [==============================] - 15s 60ms/step - loss: 0.0689 - mape: 3.7053 - val_loss: 0.6949 - val_mape: 34.8583
```



Training and validation mape decrease a little, but we still have underfitting and overfitting problems.

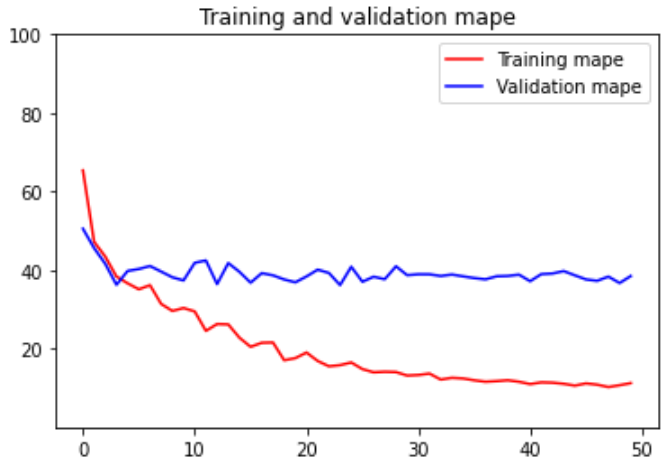
Finally, we try regularization with CNN and LSTM:

```
model = Sequential()
# We specify the maximum input length to our Embedding layer
# so we can later flatten the embedded inputs

model.add(Embedding(max_features, 64, input_length=maxlen))

# After the Embedding layer,
# our activations have shape `(samples, maxlen, 64)`.

model.add(layers.Conv1D(64, 5, activation='relu'))
model.add(layers.LSTM(32,dropout=0.2,recurrent_dropout=0.2))
model.add(Dense(1))
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding (Embedding)       (None, 20, 64)            640000

 conv1d (Conv1D)             (None, 16, 64)            20544

 lstm (LSTM)                 (None, 32)                12416

 dense (Dense)               (None, 1)                 33

=================================================================
Total params: 672,993
Trainable params: 672,993
Non-trainable params: 0
_________________________________________________________________
```

This is our final result:

```
Epoch 44/50
64/64 [==============================] - 2s 35ms/step - loss: 0.0868 - mape: 11.0790 - val_loss: 0.9321 - val_mape: 39.7642
Epoch 45/50
64/64 [==============================] - 2s 36ms/step - loss: 0.0856 - mape: 10.6135 - val_loss: 0.9369 - val_mape: 38.6775
Epoch 46/50
64/64 [==============================] - 2s 36ms/step - loss: 0.0939 - mape: 11.1791 - val_loss: 0.9560 - val_mape: 37.6357
Epoch 47/50
64/64 [==============================] - 2s 36ms/step - loss: 0.0927 - mape: 10.8518 - val_loss: 0.9125 - val_mape: 37.2894
Epoch 48/50
64/64 [==============================] - 2s 36ms/step - loss: 0.0854 - mape: 10.2465 - val_loss: 0.9160 - val_mape: 38.3534
Epoch 49/50
64/64 [==============================] - 2s 35ms/step - loss: 0.0814 - mape: 10.7148 - val_loss: 0.9454 - val_mape: 36.7210
Epoch 50/50
64/64 [==============================] - 2s 35ms/step - loss: 0.0963 - mape: 11.2450 - val_loss: 0.9376 - val_mape: 38.4953
```



The distance between training and validation mape is smaller, and both of them are stable at last. There still exists underfitting and overfitting problems, but the overfitting problem has been reduced.

Then we save this model. When we meet similar programs, we can just use this architecture.

```
model.save('/content/embeddingmodel.h5')
```

7.3. Combination of the Two Models

This is the structure of the combination model:

```
embedding_layer = Embedding(max_features, 64, input_length=maxlen)(input_1)
Layer_1 = Conv1D(64, 5, activation='relu')(embedding_layer)
Layer_2 = LSTM(32,dropout=0.2,recurrent_dropout=0.2)(Layer_1)
```

```
dense_layer_1 = Dense(256,input_shape=(X_train.shape[1],), activation='relu', kernel_regularizer="l2")(input_2)
dense_layer_2 = Dropout(0.15)(dense_layer_1)
dense_layer_3 = Dense(64, activation='relu')(dense_layer_2)
```

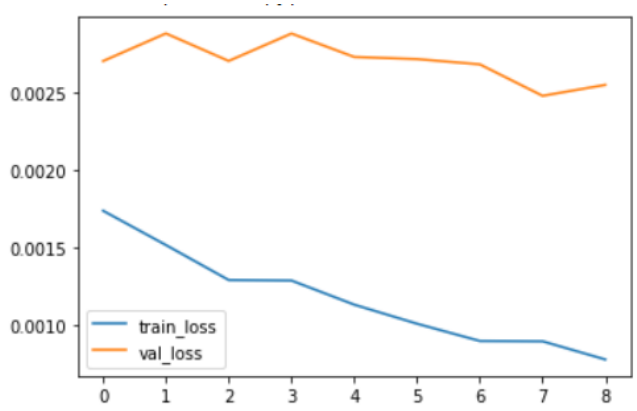
```
from keras.models import Model
from keras.layers import Concatenate

concat_layer = Concatenate()([Layer_2, dense_layer_3])
dense_layer_4 = Dense(32, activation='relu')(concat_layer)
dense_layer_5 = Dropout(0.15)(dense_layer_4)
dense_layer_6 = Dense(8, activation='relu')(dense_layer_5)
output = Dense(1, activation='sigmoid')(dense_layer_6)
model = Model(inputs=[input_1, input_2], outputs=output)
```

```
model.compile(optimizer='rmsprop', loss='mean_squared_error')
print(model.summary())
```

```
Model: "model_29"
__________________________________________________________________________________________________
 Layer (type)                   Output Shape         Param #     Connected to
==================================================================================================
 input_32 (InputLayer)          [(None, 323)]        0           []

 dense_136 (Dense)              (None, 256)          82944       ['input_32[0][0]']

 input_31 (InputLayer)          [(None, 80)]         0           []

 dropout_50 (Dropout)           (None, 256)          0           ['dense_136[0][0]']

 embedding_24 (Embedding)       (None, 80, 64)       640000      ['input_31[0][0]']

 dense_137 (Dense)              (None, 64)           16448       ['dropout_50[0][0]']

 conv1d_24 (Conv1D)             (None, 76, 64)       20544       ['embedding_24[0][0]']

 dropout_51 (Dropout)           (None, 64)           0           ['dense_137[0][0]']

 lstm_24 (LSTM)                 (None, 32)           12416       ['conv1d_24[0][0]']

 dense_138 (Dense)              (None, 32)           2080        ['dropout_51[0][0]']

 concatenate_33 (Concatenate)   (None, 64)           0           ['lstm_24[0][0]',
                                                                  'dense_138[0][0]']

 dense_139 (Dense)              (None, 8)            520         ['concatenate_33[0][0]']

 dense_140 (Dense)              (None, 1)            9           ['dense_139[0][0]']

==================================================================================================
Total params: 774,961
Trainable params: 774,961
Non-trainable params: 0
__________________________________________________________________________________________________
```

And we get the result below:

```
Epoch 1/10
271/271 [==============================] - 31s 101ms/step - loss: 0.0148 - val_loss: 0.0046
Epoch 2/10
271/271 [==============================] - 27s 100ms/step - loss: 0.0017 - val_loss: 0.0027
Epoch 3/10
271/271 [==============================] - 27s 101ms/step - loss: 0.0015 - val_loss: 0.0029
Epoch 4/10
271/271 [==============================] - 27s 98ms/step - loss: 0.0013 - val_loss: 0.0027
Epoch 5/10
271/271 [==============================] - 28s 103ms/step - loss: 0.0013 - val_loss: 0.0029
Epoch 6/10
271/271 [==============================] - 26s 98ms/step - loss: 0.0011 - val_loss: 0.0027
Epoch 7/10
271/271 [==============================] - 27s 98ms/step - loss: 0.0010 - val_loss: 0.0027
Epoch 8/10
271/271 [==============================] - 26s 98ms/step - loss: 8.9818e-04 - val_loss: 0.0027
Epoch 9/10
271/271 [==============================] - 26s 98ms/step - loss: 8.9615e-04 - val_loss: 0.0025
Epoch 10/10
271/271 [==============================] - 27s 99ms/step - loss: 7.7902e-04 - val_loss: 0.0025
```



We can see that the training MSE is lower to around 0.0008. The validation MSE is about 0.0025, which is also lower than the model we developed in Section 7.1. Though there are some overfitting problems in this model, the problem is not severe.

Then we save this model. When we meet similar programs, we can just use this architecture.

```
model.save('/content/combinationmodel.h5')
```

8. Conclusions, Discussions, and Recommendations

The real estate market is a very important part of a country's economy. Many previous studies have proved the feasibility of using machine learning and deep learning methods for housing price prediction. In our project, we tried including the textual data to improve the model performance. We built our models based on over 7000 homes listed for sale in Atlanta.

Based on our results, it is clear that neural networks are powerful tools in predicting housing prices. The fully-connected neural network model based only on house features produced the MSE at around 0.003. As for the model based on textual data, we can see the accuracy of the model is not as high but the textual data does contain information that can help with the prediction. Finally, for the combination model, the test MSE is about 0.0025. It improved a bit compared with the model without textual data. The performance may further improve if we keep tuning the parameters.

There are also some limitations inherent in this study. First, the dataset only contains houses in Atlanta and the sample size may not be enough. Also, the model may be further improved if we include

more spatial context information and maybe images. Finally, the data was collected at a single time and we do not consider the time trend.

Therefore, for the next step, we could include more data from multiple cities and in different years, so our model can capture the time trend and can be more applicable to different areas. We could also include images and more spatial data to examine if they can help improve the performance of the prediction model.

References

[1] Published by Statista Research Department, (2021, May). U.S. home sales 2021. Retrieved from https://www.statista.com/statistics/275156/total-home-sales-in-the-united-states-from-2009/

[2] Yazdani, M. (2021). Machine Learning, Deep Learning, and Hedonic Methods for Real Estate Price Prediction. *arXiv preprint arXiv:2110.07151*.

[3] Yu, L., Jiao, C., Xin, H., Wang, Y., & Wang, K. (2018). Prediction on housing price based on deep learning. *International Journal of Computer and Information Engineering*, *12*(2), 90-99.

[4] Limsombunchai, V. (2004, June). House price prediction: hedonic price model vs. artificial neural network. In *New Zealand agricultural and resource economics society conference* (pp. 25-26).

[5] Zhou, X., Tong, W., & Li, D. (2019). Modeling housing rent in the Atlanta metropolitan area using textual information and deep learning. *ISPRS International Journal of Geo-Information*, *8*(8), 349.

Appendix

Table 1. Variable descriptions

list_price	Most recently posted list price for the property
lotsize_sqft	Size of the entire lot (including outdoor space) in square feet
sqft	Interior size of the house in square feet
beds	Number of bedrooms
baths_full	Number of full bathrooms
baths_half	Number of half bathrooms
property_type	Type of property (e.g. Townhouse, Single Family Residence)
year_built	Year in which the property was constructed
zip	Zipcode
description	Seller's description of the property

Figure 1. List price range of the dataset