One-Dimensional Convolutional Neural Network Nonlinear Regression Using the California House Prices Data Set

1st Shouxi Li
Department of Electrical and Computer Engineering
Lakehead University
sli45@lakeheadu.ca

***Abstract*—This report is conducted after the experiment of building a one-dimensional convolutional neural network to predict the house price with the 'California house price' data set. The programming platform is on the Google Colab, and the programming language is Python3 with the Keras library. By loading the raw 'csv' file from GitHub, the data can be obtained in a data frame. Apart from that, there are several steps before building the artificial neural network, which includes showing the first ten records, plotting each attribute, preprocessing the data, etc. The performance of the network is measured by the metrics of the least absolute deviations (L1 loss) and the score of the coefficient of determination (R2 score). For each training epoch, there is a corresponding evaluation in order to monitor the status of the model so that problems such as overfitting and underfitting could be diagnosed, and the learning progress of the model could be obtained by the user.**

***Index Terms*—convolutional neural network, nonlinear regression, California house price data set, Keras**

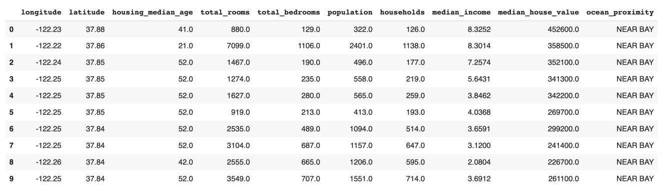
I. Introduction

The growing and developing industry of graphics hardware promises the industry of machine learning and deep learning [1]. Now the powerful chips inside GPUs have significantly boost the speed of training and testing of the artificial neural networks, which makes it possible, in this experiment, to choose GPUs as acceleration hardware to make save the training time. The network built in this experiment is a one-dimensional convolutional neural network that includes several layers, which will be explained in detail in the coming section. The data set that has been chosen in this experiment is the 'California house price', which contains data drawn from the 1990 U.S. Census [2]. The raw data set can be obtained at GitHub with the file extension of 'csv.' The data set is formatted in a data frame, which contains ten columns indicates longitude (a measure of how far west a house is; a higher value is farther west), latitude (a measure of how far north a house is; a higher value is farther north), housingMedianAge (median age of a house within a block; a lower number is a newer building), totalRooms (total number of rooms within a block), totalBedrooms (total number of bedrooms within a block), population (total number of people residing within a block), households (total number of households, a group of people residing within a home unit, for a block), medianIncome (median income for households within a block of houses; measured in tens of thousands of U.S. dollars), and medianHouseValue (median house value for households within a block; measured in U.S. dollars) [2].

II. Methodology

A. Data Preparation

The first step in this part is to load and show the data in a data frame in order to determine whether to convert the data type is necessary in another step, which is demonstrated in the following figure (Fig. 1). To have a direct observation of



	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	median_house_value	ocean_proximity
0	-122.23	37.88	41.0	880.0	129.0	322.0	126.0	8.3252	452600.0	NEAR BAY
1	-122.22	37.86	21.0	7099.0	1106.0	2401.0	1138.0	8.3014	358500.0	NEAR BAY
2	-122.24	37.85	52.0	1467.0	190.0	496.0	177.0	7.2574	352100.0	NEAR BAY
3	-122.25	37.85	52.0	1274.0	235.0	558.0	219.0	5.6431	341300.0	NEAR BAY
4	-122.25	37.85	52.0	1627.0	280.0	565.0	259.0	3.8462	342200.0	NEAR BAY
5	-122.25	37.85	52.0	919.0	213.0	413.0	193.0	4.0368	269700.0	NEAR BAY
6	-122.25	37.84	52.0	2535.0	489.0	1094.0	514.0	3.6591	299200.0	NEAR BAY
7	-122.25	37.84	52.0	3104.0	687.0	1157.0	647.0	3.1200	241400.0	NEAR BAY
8	-122.26	37.84	42.0	2555.0	665.0	1206.0	595.0	2.0804	226700.0	NEAR BAY
9	-122.25	37.84	52.0	3549.0	707.0	1551.0	714.0	3.6912	261100.0	NEAR BAY

Fig. 1. Example of the Data Frame.

the data set, it is common to plot it (Fig. 2). Apart from that, there are several plotting methods supported in Python, and for easiness, the 'ggplot' style is used in the experiment. However, this data set contains more than twenty thousand rows of data, it would be a mess if choose to plot all of them, so only the first nineteenth rows of data will be plotted here started from the index 0 to 18.

After loading this data frame, it is obvious that the first ninth columns are in numerical representation while the last column is in alphabetic representation. Since this experiment aims at using the first eight columns to predict the ninth column, there is no need to convert the data type of the tenth column. However, this data set contains more than twenty thousand rows, it is possible that there are some missing values. To handle this problem, Python provides a solution, a function named 'dropna' is called to clean all the rows that have unclear data. After the cleaning, the rows of the data frame reduced, which indicates that there are some missing values.

Additionally, looking at the value of each attribute, they differed a lot, for example, for the second row, the housing-MedianAge is 21.0 while at the same time the totalRooms is 7099.00, which shows a great gap between these two features.

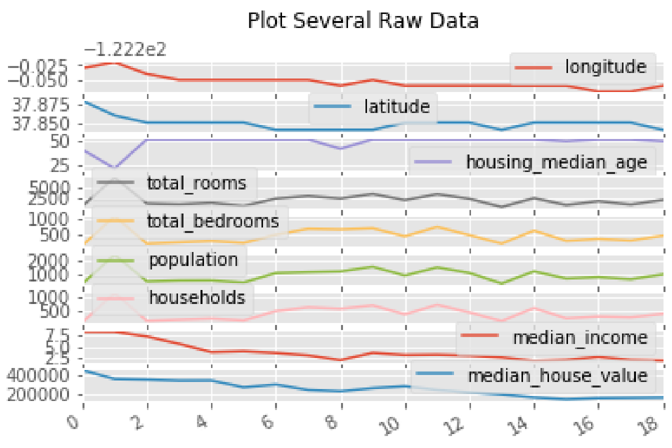


Fig. 2. Plot the Data Set.

In order to reduce the effect of the totalRooms attribute, a normalization is required in this phase. Moreover, the meadianIncome is measured in tens of thousands of U.S. dollar while the target value in the experiment (medianHouseValue) is measured in U.S. dollar, this gap should also be taken into consideration. To start with, it is necessary to match the scale of the income with the house value, which makes more sense than the original representation.

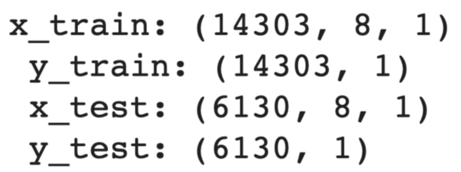
Following is to split the data set to get the training and testing data for further use, in this case, the function from 'sklearn' named 'train_test_split' is called. The ratio of training and testing is set to be 0.7, which means 70% of the data will be used in the training part, while the remaining 30% part will be in the testing part. To make sure that each time the result of the spilt operation will not change when executing the code for reproduction, a random state of 2003 is fixed.

Since the Keras library is used in this experiment, it is required to change the type of the data into Numpy arrays. To achieve this, a function named 'to_numpy()' is called to do the job. After calling this function, the data type of the original data frame will now changes to arrays. As mentioned above, this data set contains data that have great gap between different feature attributes, it is necessary to normalize them to reduce the effect of data points with large a number in order to save the time in the coming training and testing part and possibly to increase the performance of the model. For the easiness of use, here a function from Sklearn named 'StandardScaler()' is called to perform the normalization. The mathematical expression of this technique can be denoted as:

$$y = \frac{(x - \mu)}{\sigma} \tag{1}$$

where y is the output of the equation, x is the input, μ is the mean of the data group, and σ stands for the standard deviation of the group of data.

The final step for the data preparation is to reshape the size of the arrays, since the one-dimensional convolution operation will be conducted in the neural network, it is necessary to reshape them in order to feed one row of data for each iteration. To accomplish this, the function named 'reshape()' will be called to add another dimension in the data set, for example, the original (14303, 8) sized training data will now be changed to (14303, 8, 1). For a more direct expression, the size of the reshaped data is shown in this figure (Fig. 3).



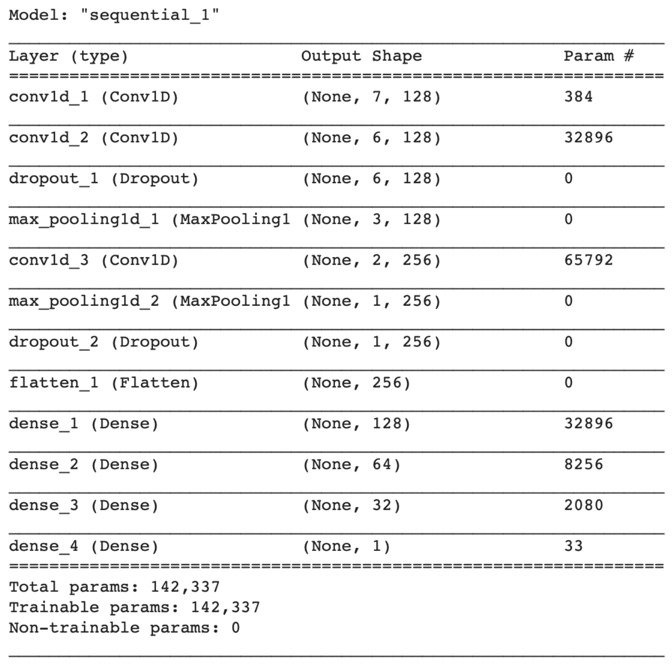
```
x_train: (14303, 8, 1)
 y_train: (14303, 1)
 x_test: (6130, 8, 1)
 y_test: (6130, 1)
```

Fig. 3. The Size of the Reshaped Data.

B. Build the Model

As mentioned above, this model is built using the Keras library. From the aspect of user friendly, Keras offers a method called sequential model building, which is basically adding different layers to build a neural network and indexing each layer to make further analysis and observation more easily. After adding enough layers to the neural network, the user could call a function named 'compile()' to set up the model and check whether the network is legit or not. In this compiling process, the metrics to evaluate the model should be fixed, and there are a few predefined and acknowledged functions available. In this case, the 'mean_absolute_error' represents the L1 loss, however, the R2 score is not predefined in Keras, which means the user should customize this metric. After compiling the model, it is necessary to call the function 'summary()' to check how many parameters are in this model and how many of them are trainable or non-trainable.

As it shows in 'Fig. 4', the model in this experiment contains twelve layers, which contains three convolution layers, two drop out layers, two pooling layers, one flatten layer four dense layer. In detail, the first layer is a one-dimensional convolution layer with the input shape of (8, 1), which specifies that the input data in each iteration have eight feature attributes and one time step; the kernel size is 2 with stride equals to 1, and the number of filters are 128, which means that after this layer, the raw (8, 1) data will change to (7, 128). The second layer has the same arguments as the first layer, which will the data to (6, 128). Following is a dropout layer which would drop 50% of the data to avoid possible overfitting phenomena. The coming maxpooling layer will pool the greater value with the window size of 2, which means now the data would change to (3, 128). The third convolution layer increases the number of filters to 256 while the other arguments remain unchanged, which would change the data to (2, 256). Then the last pooling layer will extract the greater value using the maxpooling operation with a window size of 2, which would result in the data becoming (1, 256). Since now the first dimension is only one, it is impossible to conduct any convolution or pooling operation, instead, a flatten layer is added here to reformat the stacked data. Then, the first



```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
conv1d_1 (Conv1D)            (None, 7, 128)            384       
_________________________________________________________________
conv1d_2 (Conv1D)            (None, 6, 128)            32896     
_________________________________________________________________
dropout_1 (Dropout)          (None, 6, 128)            0         
_________________________________________________________________
max_pooling1d_1 (MaxPooling1 (None, 3, 128)            0         
_________________________________________________________________
conv1d_3 (Conv1D)            (None, 2, 256)            65792     
_________________________________________________________________
max_pooling1d_2 (MaxPooling1 (None, 1, 256)            0         
_________________________________________________________________
dropout_2 (Dropout)          (None, 1, 256)            0         
_________________________________________________________________
flatten_1 (Flatten)          (None, 256)               0         
_________________________________________________________________
dense_1 (Dense)              (None, 128)               32896     
_________________________________________________________________
dense_2 (Dense)              (None, 64)                8256      
_________________________________________________________________
dense_3 (Dense)              (None, 32)                2080      
_________________________________________________________________
dense_4 (Dense)              (None, 1)                 33        
=================================================================
Total params: 142,337
Trainable params: 142,337
Non-trainable params: 0
_________________________________________________________________
```

Fig. 4. The Architecture of the Model.

dense layer with 128 neurons is added here to fully connect the data in the former layer, and another dense layer with 64 neurons is added, and another dense layer with 32 neurons is added. Since the object is to predict the house values using eight input feature attributes, then logically there should be only one neuron in the last layer as an output layer to give the predicting value of the house prices.

Since the goal here is to perform a nonlinear regression, and the activation function of the convolution layers in a neural network is usually set to be the ReLU function, then for the easiness, all the activation functions here in the convolution layers and dense layers are set to be the ReLU function, which is as simple as:

$$max(0, x) \tag{2}$$

where x is the input of the function.

C. Train and Test the Model

As requested, 70% of the data will be used in the training phase and the remaining 30% will be used in the testing part, which as it shows in ‘Fig. 3’, 14303 sets of data are used in training, and 6130 sets of data are used in testing. Since there is no requirement on the number of epochs, then it is set to be 150 to provide the model with a possible sufficient training. To make the training faster, a batch size of 64 is set. Since Google Colab offers a default RAM of 12GB, there would be no problem for such size of a batch.

Apart from that, Keras offers an argument in the training part called ‘validation_data’, which is to let the user provide some data as evaluation for each training epoch, and those evaluation data will not be used for training in the process. Thus, it is possible to test the performance of the model after each epoch of training, and obtain the change of the testing performance so that the user could save the model at its peak performance within a provided number of training epochs, and determine whether there exists an overfitting or underfitting phenomena. In this experiment, evaluation data are provided in the training part, and the corresponding performing metrics will be plotted in the next section.

III. Model Evaluation

As mentioned above, the changes of the performing metrics are shown in ‘Fig. 5’ and ‘Fig. 6’. It is obvious that with

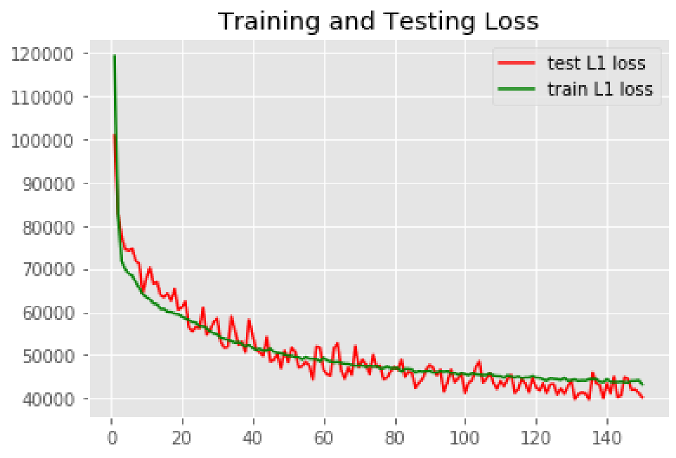


Fig. 5. The Training and Testing L1 Loss of the Model.

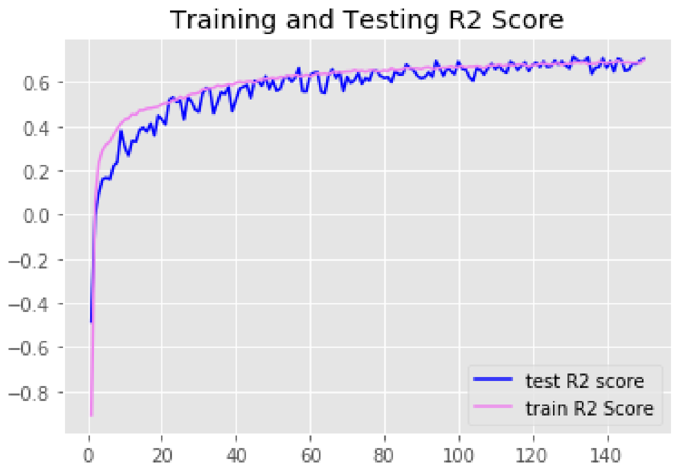
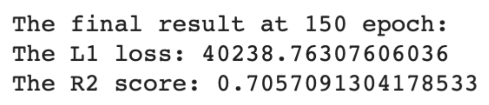


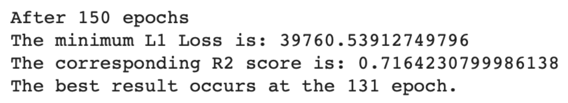
Fig. 6. The Training and Testing R2 Score of the Model.

the increasing number of epochs, the model tends to have a lower L1 loss and a higher R2 score, which means that the gap between the predicted value and the ground truth is narrowing, and the ability for the model to predict the house prices is becoming better. The performance of each epoch will not be demonstrated in specific numbers in the paper due to the limited typing space, while it can be found in the source code. The performance metrics of the model at the last (150th) epoch are shown in ‘Fig. 7’, and the best performance metrics over all training epochs are shown in ‘Fig. 8’.



```
The final result at 150 epoch:
The L1 loss: 40238.76307606036
The R2 score: 0.7057091304178533
```

Fig. 7. The Performance in the Last Epoch.



```
After 150 epochs
The minimum L1 Loss is: 39760.53912749796
The corresponding R2 score is: 0.7164230799986138
The best result occurs at the 131 epoch.
```

Fig. 8. The Best Performance Obtained within 150 Epochs.

From ‘Fig. 8’, the conclusion is that within the limit of 150 epochs of training, the model performs best at the 131st epoch with the L1 loss equals to 39760.5391, and the R2 score equals to 0.7164; the performance after this epoch will not surpass this one. However, it is unclear if a better performance would occur for a new selection number of training epochs that is greater than 150.

Additionally, the testing metrics show that the performance of the model is oscillating along with the increasing epochs of training, which means the network is not robust enough, and the reason why would this happen could be related to the architecture of the neural network, or the parameters selected in the drop out layers and pooling layers, or the choices of activation functions.

References

[1] Razzak, M. I., Naz, S., Zaib, A. (2018). Deep learning for medical image processing: Overview, challenges and the future. In Classification in BioApps (pp. 323-350). Springer, Cham.

[2] Pace, R. K., Barry, R. (1997). Sparse spatial autoregressions. Statistics Probability Letters, 33(3), 291-297.