NonLinear Regression using 1D convolution

Umang Mehta
Student Id: 1117269
Lakehead University

***Abstract*—The report describes Implementation, result, a comparison between approaches of a one-dimensional convolution (Conv1D)-based neural network for predicting median house value using longitude, latitude, housing median age, total number of rooms, total number of bedrooms, population, number of households, and median income.**

***Index Terms*—Keywords- Nonlinear regression, Natural Language Processing, 1D CNN.**

I. Introduction

There is a data of the California Housing to predict the median house value. To achieve such task, we describe the nonlinear regression, one dimensional convolution (Conv1D) based neural Network. The Nonlinear regression is a type of regression analysis in which data can be fit to a model and express with the mathematical function. On the other hand, CNN functions well to find basic patterns within the results, which will then be used to construct more complex patterns within higher layers. A 1D CNN is very useful where you intend to extract interesting features from limited-length segments of the overall data set and where the location of the feature within the segment is not of great significance.

II. PROPOSED MODEL

A. Preparing and Reading data

To implement, first we need to change runtime type to the python 3.0 and GPU. Followed by that, mounted the google drive with the notebook, so we can save and import any data directly from drive. The next step is to import libraries as followed: pandas, sklearn, and numpy. pandas is using to reading comma-separated value (CSV) file and removed incomplete entries from the dataset. Sklearn is used for splitting data into training and testing parts and working with and manipulating data respectively Numpy is used to get numpy arrays converted from dataset.

Here is the output of first 10 rows from the database.

```
dataset = pd.read_csv('/content/drive/My
    Drive/Colab Notebooks/housing.csv')

#dropna use to remove missing values form the
    dataset
dataset = dataset.dropna()

print('here are ten first ten row of the
    dataset')
dataset.head(10)
```

here are te first ten row of the dataset

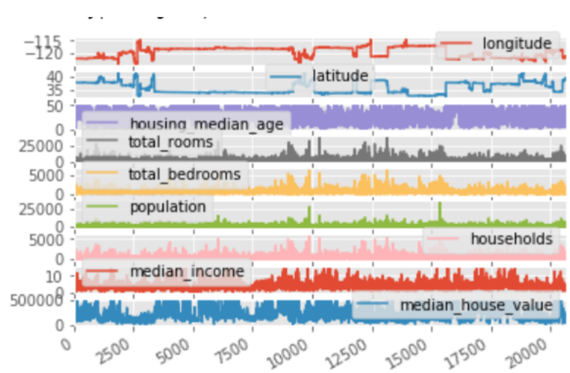
	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	median_house_value	ocean_proximity
0	-122.23	37.88	41	880	129.0	322	126	8.3252	452600	NEAR BAY
1	-122.22	37.86	21	7099	1106.0	2401	1138	8.3014	358500	NEAR BAY
2	-122.24	37.85	52	1467	190.0	496	177	7.2574	352100	NEAR BAY
3	-122.25	37.85	52	1274	235.0	558	219	5.6431	341300	NEAR BAY
4	-122.25	37.85	52	1627	280.0	565	259	3.8462	342200	NEAR BAY
5	-122.25	37.85	52	919	213.0	413	193	4.0368	269700	NEAR BAY
6	-122.25	37.84	52	2535	489.0	1094	514	3.6591	299200	NEAR BAY
7	-122.25	37.84	52	3104	687.0	1157	647	3.1200	241400	NEAR BAY
8	-122.26	37.84	42	2555	665.0	1206	595	2.0804	226700	NEAR BAY
9	-122.25	37.84	52	3549	707.0	1551	714	3.6912	261100	NEAR BAY

Now we are going to plot each feature of the dataset on a separate sub-plot.

```
% matplotlib inline
import matplotlib.pyplot as plt

plt.style.use('ggplot')
dataset.plot(subplots=True)
```

OUTPUT:



Now we are going to split our dataset in 70-30 ratio. So 70% of data goes into training and 30% of data is allotted for testing.

```
# Splits the dataset so 70% is used for
    training and 30% for testing
x_train, x_test, y_train, y_test =
    train_test_split(X, Y, test_size= 0.3)
```

Output:
There are 114424 training entries and 49040 testing entries!

B. Defining Model

Convolutional Neural Networks (ConvNets) works well on such problems just because of the ability to extract features form the input. 1D convolution layers conduct input translation for each patch. Such patches can be extracted from input and output the maximum value that is called Max Pooling. For this implementation, I have used two conv1d layers to extract the features, polling and relu activation function.

First we are Importing Libraries:

```
#Importing the pytorch library
import torch

#Importing the 1D convolution layer
from torch.nn import Conv1d

# Importing the max pooling layer
from torch.nn import MaxPool1d

# Importing the flatten layer
from torch.nn import Flatten

# Importing the linear layer
from torch.nn import Linear

# Importing the ReLU activation function
from torch.nn.functional import relu

# to work with datasets we are importing the
    DataLoader and TensorDataset libraries
    from PyTorch
from torch.utils.data import
    DataLoader,TensorDataset

#Importing L1Loss
from torch.nn import L1Loss

!pip install pytorch-ignite

#Importing R2Score
from
    ignite.contrib.metrics.regression.r2_score
    import R2Score
```

Creating class CnnRegressor with one default constructor and another is feed method to feed our model.

```
#creating a class
class CnnRegressor(torch.nn.Module):
  def __init__(self, batch_size, inputs,
      outputs):
    super(CnnRegressor, self).__init__()
    self.batch_size = batch_sizes
    self.inputs = inputs
    self.outputs = outputs

    self.input_layer = Conv1d(inputs,
       batch_size, 1 , 2)

    # First Max pooling layer
    self.max_pooling_layer = MaxPool1d(1)
    #First conv layer
    self.conv_layer = Conv1d(batch_size, 128,
       1)

    #adding second max pooling layer
    self.max_pooling_layer1 = MaxPool1d(1)

    #adding second conv layer
    self.conv_layer1 = Conv1d(128, 64, 1)

    #adding flatten layer
    self.flatten_layer = Flatten()

    self.linear_layer = Linear(64, 64)
    self.output_layer = Linear(64, outputs)

  #writing feed method
  def feed(self, input):
    input = input.reshape((self.batch_size,
        self.inputs,1))
    output = relu(self.input_layer(input))

    output = self.max_pooling_layer(output)
    output = relu(self.conv_layer(output))

    output = self.max_pooling_layer1(output)
    output = relu(self.conv_layer1(output))

    output = self.flatten_layer(output)
    output = self.linear_layer(output)
    output = self.output_layer(output)

    return output
```

The constructor with parameters batch size, input, and output. Followed by that, assigning the values to the parameters and adding two Max Pooling and Conv1D layer. The max pooling is used to reduce the size by taking the maximum value of elements in the window. In the Flatten layer is used to flatten the input shape into simple vector output.

Now, we set the batch size = 32 and defining a model.

```
#Seting batch size
batch_size = 32

#passing parameters to class constructor
model = CnnRegressor(batch_size, X.shape[1],
    1)

model.cuda()
```

```
CnnRegressor(
  (input_layer): Conv1d(8, 32,
     kernel_size=(1,), stride=(2,))
  (max_pooling_layer):
     MaxPool1d(kernel_size=1, stride=1,
     padding=0, dilation=1, ceil_mode=False)
  (conv_layer): Conv1d(32, 128,
     kernel_size=(1,), stride=(1,))
  (max_pooling_layer1):
     MaxPool1d(kernel_size=1, stride=1,
     padding=0, dilation=1, ceil_mode=False)
  (conv_layer1): Conv1d(128, 64,
     kernel_size=(1,), stride=(1,))
  (flatten_layer): Flatten()
  (linear_layer): Linear(in_features=64,
     out_features=64, bias=True)
  (output_layer): Linear(in_features=64,
     out_features=1, bias=True)
)
```

Before Training model, we need to create a method which can calculate model loss. The R2Score measures the distance of the data to the fitted regression line.

```
def model_loss(model, dataset, train = False,
    optimizer = None):
 performance = L1Loss()
 score_metric = R2Score()

 avg_loss = 0
 avg_score = 0
 count = 0

 for input, output in iter(dataset):
  predections = model.feed(input)

  loss = performance(predections, output)

  score_metric.update([predections, output])
  score = score_metric.compute()

  if(train):
   optimizer.zero_grad()

   loss.backward()

   optimizer.step()

  avg_loss += loss.item()

  avg_score += score
  count += 1

 return avg_loss / count, avg_score / count
```

C. Training the model

As we have created the network, now we need to import our optimizer. The performance of the optimizer is depending on different parameters (such as learning rate) according to the optimizer. I have tried many optimizers with different parameters but now I am using RMSprop.

```
#Training the model
import time

epochs = 500

optimizer = torch.optim.RMSprop (
    model.parameters(), lr = 1e-2 )

inputs =
    torch.from_numpy(x_train_np).cuda().float()

#Reshape is used to remove a warning while
    time of output
outputs = torch.from_numpy(y_train_np.reshape
(y_train_np.shape[0], 1)).cuda().float()

#Create a DataLoader instance to work with
    our batches
tensor = TensorDataset (inputs, outputs)
loader = DataLoader ( tensor, batch_size ,
    shuffle = True, drop_last = True)

time1 = time.time()
for epoch in range(epochs):
 avg_loss, avg_r2_score = model_loss(model,
     loader, train=True, optimizer=optimizer)

 print("Epoch " + str(epoch + 1) + ":\n\tLoss
     = " + str(avg_loss) + "\n\tR^2 Score = "
     + str(avg_r2_score))

time2 = time.time()
```

OUTPUT:
Epoch 500:
Loss = 44868.1178356222
$\hat{R2}$ Score = 0.697891784862267

To train the model, I used sklearn's linear model. Now the training set now is converted into torch variables for model by using GPU. Reshape is used to remove a warning at the time of output. Then as we created loss model, we will get the average loss of L1loss and R2Score.

In the training model, I have set the epochs 500, learning rate 1E-2 and the optimizer RMSprop. To build a dataloader instance to operate with a batch I used TensorDataset and DataLoader, which included shuffling and dropping true while executing.

```
inputs =
    torch.from_numpy(x_test_np).cuda().float()
outputs =
    torch.from_numpy(y_test_np.reshape(y_test_np.
shape[0],1)).cuda().float()

tensor = TensorDataset(inputs, outputs)

loader = DataLoader(tensor,batch_size,
    shuffle=True, drop_last=True)

avg_loss, avg_r2_score = model_loss(model,
    loader)

print("\nTime Taken: "+ str(time2-time1)
    +"\nLoss = " + str(avg_loss) + "\nR^2
    Score = " + str(avg_r2_score))
```

Output:
Time Taken: 614.1602246761322
Loss = 45061.726746564134
$\hat{R2}$ Score = 0.6950189956235179
For the Mean square error:

```
from sklearn import linear_model
from sklearn.metrics import mean_squared_error

model = linear_model.LinearRegression()
model.fit(x_train, y_train)
predictions = model.predict(x_test)
mse = mean_squared_error(y_test, predictions)
```

```
print("The mean squared error is: " +
    str(mse))
```

Output: The mean squared error is: 4927590516.896483
Now we are going to store our model in Google drive:

```
torch.save(model,"/content/drive/My
    Drive/Colab
    Notebooks/1117269_1dconv_reg.pth")
```

III. CONCLUSION

To get better accuracy, I changed many parameters and did trial and error method. Such parameters are optimizer, batch size, kernel size, number of epochs, learning rate, and optimizer. I added two layers to improve the accuracy.

With the optimizer SGD, with batch size 64 and learning rate 1e-5 with 500 epochs I got 40 percentage of accuracy. I keep changed such parameters but still I was not able to get more than 45 percentage of accuracy. However, I changed optimizer and used ADAM with the above parameters and I got 61 percentage of accuracy. Followed by that I change some parameters and I got 69 percentage accuracy. At the end, I tried with RMSprop optimizer and I got 69.5 percentage of accuracy with 500 epochs and learning rate 1E-2, batch size = 32. I have attached the readings.

Epochs	Lr rate	Batch Size	R^2 Score	loss L1	Optimizer
500	1.00E-05	64	0.64	55809	SGD
500	1.00E-02	32	0.69	45394	Adam
500	1.00E-02	32	0.695	46078	RMSprop
100	1.00E-01	32	0.55	54749	Adamax

REFERENCES

[1] https://towardsdatascience.com/understanding-learning-rates-and-how-it-improves-performance-in-deep-learning-d0d4059c1c10 595–603.

[2] bibitemc1S. Albawi, T. A. Mohammed and S. Al-Zawi, "Understanding of a convolutional neural network," 2017 International Conference on Engineering and Technology (ICET), Antalya, 2017, pp. 1-6. doi: 10.1109/ICEngTechnol.2017.8308186

[3] https://towardsdatascience.com/understanding-1d-and-3d-convolution-neural-network-keras-9d8f76e29610