NLP Assignment 1: Non-Linear Regression w/t DL

Palak Bhatia
Lakehead University)
Email-pbhatia3@lakeheadu.ca
Student Id-1116552

***Abstract*—This paper targets to implement 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 on a housing dataset. The solution predicted is a non-linear regression model.**

I. Introduction

In deep learning, a convolutional neural network (CNN, or ConvNet) is a class of deep neural networks which has been used in the assignmnet. The term "convolutionary neural network" implies the network is using a logical operation called convolution. Recently, Convolutionary neural networks have become the popular solution for numerous machine learning tasks, like object detection,natural language processing, time series classification and many more applications.
Throughout mathematics and science, a nonlinear method is a mechanism in which output changes are not equal to input changes.Given the difficulty of solving nonlinear dynamic equations, nonlinear systems are usually approximated by linear equations (linearization)[4].
Here, I have implemented a 1-Dimensional convolution layer combined with ReLU activation function to make the solution non-linear and .Regression analysis consists of a set of machine learning methods that allow us to predict a continuous outcome variable (y) based on the value of one or multiple predictor variables (x)[3].Since the median value is a numerical figure, hence this is a regression problem. The dataset used here is called California Housing Dataset which is taken from github.

II. Literature Review

A Convolutionary Neural Network (ConvNet / CNN) is a Deep Learning algorithm capable of capturing an input image, assigning significance (learnable weights and biases) to different aspects / objects in the image, and being able to distinguish one from another. The pre-processing needed in a ConvNet is significantly lower than in other classification algorithms.A ConvNet's design is similar to that of the communication structure of Neurons in the Human Brain and was influenced by the Visual Cortex organisation. Convolutionary layer consists of a series of convolutionary kernels (every neuron functions as a kernel).Convolutionary kernel functions by slicing the data into tiny slices commonly known as receptive regions. Kernel transforms to a specific set of weights on the images by multiplying its components with the corresponding sensitive field elements. Division of a picture into small blocks helps to remove motifs for the element. Different sets of functions within the image are removed with the same collection of weights by sliding kernel on the pixel. After the features have been extracted, the output is passed to a pooling layer which reduces the dimensions of the data.Activation mechanism acts as a tool for decision making, which allows to understand a complex pattern[1]. In literature, different activation functions such as sigmoid, tanh, maxout, SWISH, ReLU, and variants of ReLU such as leaky ReLU, ELU, and PReLU are used to inculcate non-linear combination of features. However, ReLU and its variants are preferred as they help in overcoming the vanishing gradient problem[4]. Fully connected layer is mostly used for grouping purposes at network level. It is a regional process, in comparison to pooling and convolution. It takes feedback from the abstraction of function stages and analyses the performance of all of the previous layers globally. It allows a non-linear mix of chosen functions that are used to identify data[1].

III. Proposed Model

The proposed model makes use of different layers of convolution network combined with an optimizer and activation function to predict the median house value of the dataset.The model has been trained and tested using the python's pytorch library. The algorithm begins by splitting the dataset in the ratio of 70:30 of training and testing using the sklearn.model selection library with random state is set to 2003. The model consists of two convolution layers having batch size as 32, kernel size of 1x1, stride rate of 1 and zero padding. I have also used a batch normalization layer to normalize the inputs. After the final max pooling layer, the output is passed to a flattening layer and linear layers to convert them into a column vector. Finally after some reshaping, the ReLu activation function is applied which changes all the negative values to zero. By using the Conv1d layers with the ReLU activation function, the model becomes non-linear. ReLU is significant and is intended for inserting non-linearity in the CNN. ReLU allows for faster and more effective training by mapping negative values to zero and maintaining positive values. This is sometimes referred to as activation, because only the activated features are carried forward into the next layer.The following subsections discuss the various elements of the model.
The proposed model has been trained over 500 epochs and for each size the batch size have been set to 32. The optimizer used

Identify applicable funding agency here. If none, delete this.

in the model is Adam optimizer having learning rate as 0.001. The first input convolution layer takes input of dimension 8x32 and passed it to the second convolution layer which takes dimensions of 32x128 and then to the linear layer after flattening. The actual code snipped for the trained convolution class can be found in Appendix 6.1

A. Over/under fitting issue

Overfitting occurs when the variability of the data is identified by a mathematical model or machine learning algorithm. Intuitively, overfitting occurs when the model or algorithm blends in too well with the data. In particular, overfitting happens where the model or algorithm experiences low bias yet large variance.The best solution for preventing overfitting is to use more accurate training data. The dataset would cover the full spectrum of inputs required for the model to manage. It has been observed that reducing the batch size, increases the overfitting. Also, using the batch-normalization decreases overfitting. The use of max-pool layer besides reducing dimensions also avoids overfitting. Since,the max-pooling layer pools out only the maximum data from the kernel, thus reducing the data and eventually minimising the overfitting
Underfitting arises when the fundamental pattern of the data is not detected by a statistical model or machine learning algorithm. Intuitively, if the pattern or algorithm does not suit the data well enough, underfitting happens. Specifically, underfitting occurs if the model or algorithm shows low variance but high bias. Underfitting is often the product of an oversimplified layout project. To avoid underfitting, I have added two layers of convolution and two layers of max-pooling, an activation layer and linear layers.
The model achieves the accuracy of 70% in tetsting phase and almost same in training phase, hence it is evident that there is no overfitting or underfitting observed in the proposed model.

B. Vanishing/exploding gradient issue

The vanishing gradient problem in machine learning is a challenge that is involved in teaching artificial neural networks with gradient-based learning methods and backpropagation. In such approaches, in each training iteration each of the weights of the neural network receives an adjustment equal to the partial derivative of the error function with respect to the current weight. The concern is that the differential will in some situations be vanishingly small, effectively preventing the weight from increasing its value. In the worst scenario, this may absolutely halt further preparation for the neural network.Neural networks are equipped using the optimisation technique for stochastic gradient descent.It involves making a forecast using the current state of the experiment, contrasting the result with the predicted values and using the discrepancy as an approximation of the gradient of the error. This gradient of error is then used to change the weights of the variable, and repeat the process.
The amount of training instances used to estimate the error gradient is a hyperparameter for the learning algorithm named the "batch size," or literally the "batch size." A batch size of 32 indicates that 32 observations from the testing dataset are used to approximate the error gradient before adjusting the model weights. One training epoch indicates that one trip through the testing dataset was created by the learning algorithm, where examples were divided into randomly selected "sample scale" classes. Along with the batch size, Adam optimizer also resolves the problem of gradient.

C. Number of trainable parameters.

Various learning parameters have been used tuned in the model which are discussed below:

- Kernel: - In machine learning, a "kernel" is usually used to refer to the kernel trick, a method of using a linear classifier to solve a non-linear problem. The model uses a kernel size of 1
- Batch Size:- A smaller mini-batch size (not too small) usually leads not only to a smaller number of iterations of a training algorithm, than a large batch size, but also to a higher accuracy overall, i.e, a neural network that performs better, in the same amount of training time, or less.Therefore, the batch size used in the model is 32.
- Number of Epochs:- The number of epochs is the number of complete passes through the training dataset. It is set to 500 for the model.
- Stride Rate:-In CNNs, a stride represents the number of shifts of pixels in the input matrix. Here the stride rate is set to 1
- Padding:- Padding is a term relevant to convolutional neural networks as it refers to the amount of pixels added to an image when it is being processed by the kernel of a CNN. No padding has been used in the model.
- Learning rate:-The amount that the weights are updated during training is referred to as the step size or the "learning rate." Specifically, the learning rate is a configurable hyperparameter used in the training of neural networks that has a small positive value, often in the range between 0.0 and 1.0 . For this model, the learning rate is set to 0.001.
- Optimizer:- Optimizers are algorithms or methods used to change the attributes of your neural network such as weights and learning rate in order to reduce the losses. Adam optimizer have been used in the model.

IV. Experimental Analysis

The model gives a testing accuracy of 73.09% which is observed as the R2 Score as shown in fig1. While training the model, I altered various training parameters such as kernel size, padding, stride rate, batch size, learning rate, number of epochs etc to achieve higher accuracy. Along with the training parameters, I also changed the optimizers and activation functions to predict the model withn highest testing accuracy. I also tried to compare the accuracy by adding and removing various layers. Some of these comparisons have been analyzed below:-

- Optimizer- Using different optimizers such as Adam, Adamax and SGD gave different accuracies, where Adam

```
        Loss = 106338.15706954656
        R^2 Score = -0.6904850650134757
Epoch 79:
        Loss = 105974.68970588235
        R^2 Score = -0.7136254716674314
Epoch 80:
        Loss = 105647.3200291054
        R^2 Score = -0.7278070620276612
Epoch 81:
        Loss = 105344.70274203431
        R^2 Score = -0.6749463347401387
Epoch 82:
        Loss = 104983.26231617646
        R^2 Score = -0.7214243941246323
Epoch 83:
        Loss = 104667.28238357844
        R^2 Score = -0.6765654685879606
Epoch 84:
        Loss = 104335.97342218137
        R^2 Score = -0.6167100654247308
Epoch 85:
        Loss = 104005.13606770833
        R^2 Score = -0.6562966380403888
Epoch 86:
        Loss = 103598.17306985294
        R^2 Score = -0.6299881894975875
Epoch 87:
        Loss = 103184.59111519608
```

```
# Convert the testing set into torch variables for our model using the GPU
# as floats
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()
# Create a DataLoader instance to work with our batches
tensor = TensorDataset(inputs, outputs)
loader = DataLoader(tensor, batch_size, shuffle=True, drop_last=True)
# Output the average performance of the model
avg_loss, avg_r2_score = model_loss(model, loader)
print("The model's L1 loss is: " + str(avg_loss))
print("The model's R^2 score is: " + str(avg_r2_score))
```

```
The model's L1 loss is: 41656.8226808563
The model's R^2 score is: 0.730905120580973
```

Fig. 1. Testing Accuracy

being the best. It gave the testing accuracy of 73.09%. SGD optimizer gave just 29.55% accuracy whereras adamax gave 40.63% as shown in fig 2 and fig 3.

:

```
+ Code  + Text
epochs = 200
# Define the performance measure and optimizer
optimizer = torch.optim.SGD(model.parameters(), lr=1e-5)
# Convert the training set into torch variables for our model using the GPU
# as floats. The reshape is to remove a warning pytorch outputs otherwise.
inputs = torch.from_numpy(x_train_np).cuda().float()
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)
# Start the training loop
for epoch in range(epochs):
# Cycle through the batches and get the average loss
  avg_loss, avg_r2_score = model_loss(model, loader, train=True, optimizer=optimizer)
# Output the average loss
  print("Epoch " + str(epoch + 1) + ":\n\tLoss = " + str(avg_loss) + "\n\tR^2 Score = " + str(avg
#### Time for training the model
time_taken = time() - start_time
print("Inference time: %.4f s" % time_taken)
```

```
[43] # Convert the testing set into torch variables for our model using the GPU
# as floats
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()
# Create a DataLoader instance to work with our batches
tensor = TensorDataset(inputs, outputs)
loader = DataLoader(tensor, batch_size, shuffle=True, drop_last=True)
# Output the average performance of the model
avg_loss,avg_r2_score = model_loss(model, loader)
print(str(model_loss))
print("The model's L1 loss is: " + str(avg_loss))
print("The model's R^2 score is: " + str(avg_r2_score))
# Plot in blue color the predicted adata and in green color the
# actual data to verify visually the accuracy of the model.
```

```
<function model_loss at 0x7f219c1d4f28>
The model's L1 loss is: 73251.39966459424
The model's R^2 score is: 0.29555455344556986
```

Fig. 2. Testing Accuracy with SGD Optimizer

- Batch Size:- As shown in fig 4, increasing the batch size to 64 reduced the accuracy to 54.14%.
- Kernel:- Increasing the kernel size to 2 decreases the accuracy to 65.1%(fig 5).

```
# Define the performance measure and optimizer
optimizer = torch.optim.Adamax(model.parameters())
# Convert the training set into torch variables for our model using the GPU
# as floats. The reshape is to remove a warning pytorch outputs otherwise.
inputs = torch.from_numpy(x_train_np).cuda().float()
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)
# Start the training loop
for epoch in range(epochs):
# Cycle through the batches and get the average loss
  avg_loss, avg_r2_score = model_loss(model, loader, train=True, optimizer=optimizer)
# Output the average loss
  print("Epoch " + str(epoch + 1) + ":\n\tLoss = " + str(avg_loss) + "\n\tR^2 Score = " + str(avg_r2_score))
#### Time for training the model
time_taken = time() - start_time
print("Inference time: %.4f s" % time_taken)
```

```
[45] # Convert the testing set into torch variables for our model using the GPU
# as floats
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()
# Create a DataLoader instance to work with our batches
tensor = TensorDataset(inputs, outputs)
loader = DataLoader(tensor, batch_size, shuffle=True, drop_last=True)
# Output the average performance of the model
avg_loss,avg_r2_score = model_loss(model, loader)
print(str(model_loss))
print("The model's L1 loss is: " + str(avg_loss))
print("The model's R^2 score is: " + str(avg_r2_score))
# Plot in blue color the predicted adata and in green color the
# actual data to verify visually the accuracy of the model.
```

```
<function model_loss at 0x7f219c1d4f28>
The model's L1 loss is: 67074.8595795157
The model's R^2 score is: 0.40636487565650153
```

Fig. 3. Testing Accuracy with Adamax Optimizer

```
+ Code  + Text
start_time = time()
# Define the number of epochs to train for

epochs = 20
# Define the performance measure and optimizer
optimizer = torch.optim.Adam(model.parameters())
# Convert the training set into torch variables for our model using the GPU
# as floats. The reshape is to remove a warning pytorch outputs otherwise.
inputs = torch.from_numpy(x_train_np).cuda().float()
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)
# Start the training loop
for epoch in range(epochs):
# Cycle through the batches and get the average loss
  avg_loss, avg_r2_score = model_loss(model, loader, train=True, optimizer=optimizer)
# Output the average loss
  print("Epoch " + str(epoch + 1) + ":\n\tLoss = " + str(avg_loss) + "\n\tR^2 Score = " + str(avg_r2_score))
#### Time for training the model
time_taken = time() - start_time
print("Inference time: %.4f s" % time_taken)
```

```
[50] # Convert the testing set into torch variables for our model using the GPU
# as floats
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()
# Create a DataLoader instance to work with our batches
tensor = TensorDataset(inputs, outputs)
loader = DataLoader(tensor, batch_size, shuffle=True, drop_last=True)
# Output the average performance of the model
avg_loss,avg_r2_score = model_loss(model, loader)
print(str(model_loss))
print("The model's L1 loss is: " + str(avg_loss))
print("The model's R^2 score is: " + str(avg_r2_score))
# Plot in blue color the predicted adata and in green color the
# actual data to verify visually the accuracy of the model.
```

```
<function model_loss at 0x7f219c1d4f28>
The model's L1 loss is: 58877.27754934211
The model's R^2 score is: 0.5414786748672215
```

Fig. 4. Testing Accuracy with Batch Size=64

- Inference Time:- This inference time obtained for the best-trained model which gives accuracy of 71.23% is 359.3267 sec.
- Accuracy measures:- The model's accuracy has been calculated using two parameters which are R2 Score and L1Loss.R-squared is a statistical measure of how close the data are to the fitted regression line.[2] It is

```
[57] # Define the number of epochs to train for

     epochs = 500
     # Define the performance measure and optimizer
     optimizer = torch.optim.Adam(model.parameters())
     # Convert the training set into torch variables for our model using the GPU
     # as floats. The reshape is to remove a warning pytorch outputs otherwise.
     inputs = torch.from_numpy(x_train_np).cuda().float()
     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)
     # Start the training loop
     for epoch in range(epochs):
     # Cycle through the batches and get the average loss
       avg_loss, avg_r2_score = model_loss(model, loader, train=True, optimizer=optimizer)
     # Output the average loss
       print("Epoch " + str(epoch + 1) + ":\n\tLoss = " + str(avg_loss) + "\n\tR^2 Score = " + str(avg_r2_scor
     #### Time for training the model
     time_taken = time() - start_time
     print("Inference time: %.4f s" % time_taken)
```

```
     # Convert the testing set into torch variables for our model using the GPU
     # as floats
     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()
     # Create a DataLoader instance to work with our batches
     tensor = TensorDataset(inputs, outputs)
     loader = DataLoader(tensor, batch_size, shuffle=True, drop_last=True)
     # Output the average performance of the model
     avg_loss,avg_r2_score = model_loss(model, loader)
     print(str(model_loss))
     print("The model's L1 loss is: " + str(avg_loss))
     print("The model's R^2 score is: " + str(avg_r2_score))
     # Plot in blue color the predicted adata and in green color the
     # actual data to verify visually the accuracy of the model.
```

```
<function model_loss at 0x7f219c1d4f28>
The model's L1 loss is: 48579.060773026315
The model's R^2 score is: 0.6513321123984293
```

Fig. 5. Testing Accuracy with kernel=2

also known as the coefficient of determination, or the coefficient of multiple determination for multiple regression.L1 Score is the sum of the all the absolute differences between the true value and the predicted value. The model achieves the R2 score of 0.7123 and L1 Loss is 44429.089 as shown in Fig1.

V. Plots

- Features of dataset:- The below plot[Fig 6] depicts all features of the dataset on separate sublplot. It signifies the range of values a particular feature has. The plot has been created using the matplot library of python.//
- Training R2 Score:- Fig7 shows the relation between number of epochs and the R2 Score for training dataset. This depicts the increase in accuracy on increasing number of epochs. However,the R2 Score becomes almost constant after 450.

Conclusion

A non-linear regression model has been implemented using convolution neural network for predicting the median house on California housing dataset. Appending a non-linear regression layer at the end of convolution layer provides a descent solution to this regression problem. On the basis of above observations, we see that a CNN model having 2 convolutional layers, max-pooling layers and ReLU activation function with batch size = 32, optimizer = Adam, number of epochs=500 and kernel size=1 provide the highest testing R2 Score of 0.7366.

Fig. 6. Features of the Dataset

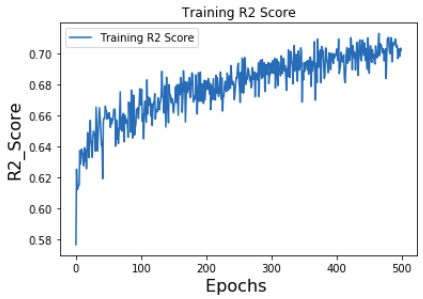

Fig. 7. Epoch V/s Training R2 Score

References

[1] A Survey of the Recent Architectures of Deep Convolutional Neural Networks by Asifullah Khan,Anabia Sohail, Umme Zahoora1 and Aqsa Saeed Qureshi1 IEEE 2018

[2] https://afteracademy.com/blog/what-are-l1-and-l2-loss-functions

[3] https://towardsdatascience.com/deep-neural-networks-for-regression-problems-81321897ca33

[4] Efficient and Accurate Approximations of Nonlinear Convolutional Networks,Xiangyu Zhang, Jianhua Zou, Xiang Ming, Kaiming He, Jian Sun; The IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2015, pp. 1984-1992

VI. Appendix

The github link for the code can be accessed using https://github.com/palakbh18/NLP

6.1 the_trained_model

```
# Our class MUST be a subclass of torch.nn.Modul
class 1116552_1dconv reg(torch.nn.Module):
# Define the initialization method
```

```
  def __init__(self, batch_size, inputs, outputs):
# Initialize the superclass and store the parameters
    super(CnnRegressor, self).__init__()
    self.batch_size = batch_size
    self.inputs = inputs
    self.outputs = outputs
# Define the input layer
# (input channels, output channels, kernel size)
    self.input_layer = Conv1d(inputs, batch_size,1)
    # Batch normalization
    self.input= torch.nn.BatchNorm1d(inputs)
# Define a max pooling layer
    self.max_pooling_layer = MaxPool1d(1)
# Define another convolution layer
    self.conv_layer = Conv1d(batch_size, 128,1)
# Define a max pooling layer
    self.max_pooling_layer2 = MaxPool1d(1)
    self.flatten_layer = Flatten()
# Define a linear layer
# (inputs, outputs)
    self.linear_layer = Linear(128, 64)
# Finally, define the output layer
    self.output_layer = Linear(64, outputs)

# Define a method to feed inputs through the model
  def feed(self, input):
# Reshape the entry so it can be fed to the input layer
# Although we re using 1D convolution, it still expects a 3D array to process in a 1D fashio
    input = input.reshape((self.batch_size, self.inputs, 1))
# Get the output of the first layer and run it through the
# the ReLU activation function
    output = relu(self.input_layer(input))
# Get the output of the max pooling layer
    output = self.max_pooling_layer(output)
# Get the output of the second convolution layer and run it
# through the ReLU activation function
    output = relu(self.conv_layer(output))
# Get the output of the flatten layer
    output = self.flatten_layer(output)
# Get the output of the linear layer and run it through the
# ReLU activation function
    output = self.linear_layer(output)
# Finally, get the output of the output layer and return it
    output = self.output_layer(output)
    return output
```