Design/Practical Experience [MEN1010]
Department of Mechanical Engineering
(Final Report)

Academic Year: 2021-2022

Semester: 4

Date of Submission of Report: 4/5/2022

1. Name of the Student: Prathmesh Gaonkar
2. Roll Number: B20ME032
3. Title of the Project : Signal Processing based Understanding of Machine Learning Models.
4. Project Category: 3
5. Targeted Deliverables: 1) Documentation of the final report.

Motivation :

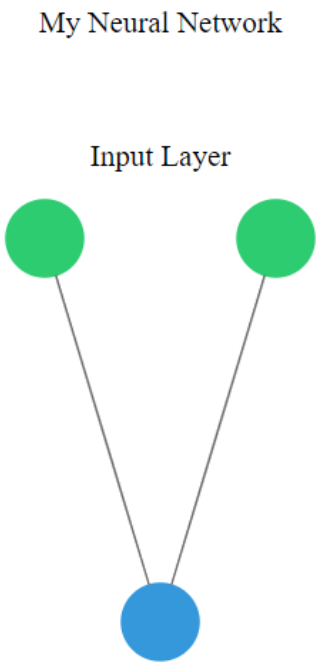
Machine learning is now a pretty common name in almost every field of research. It also impacts our daily life whether we realize it or not. It determines what we see while we scroll on Facebook, what we see on the company's website, and how we interact with brands on the internet. But however, when we work with machine learning models, we are only concerned about input and output. The inner workings of ML are quite complex and not easily understood. Overall we can say, machine learning is a black box system. This is because these black-box models are created directly from data by an algorithm, meaning that even those who design them cannot understand how variables are combined to make predictions.
In machine learning, particularly the most widely used algorithm is neural networks. This algorithm takes the input and connects it to the output by learning the weights. These weights are random numbers that do not help in making any physical interpretation.
This project focuses on understanding what these weights represent and how we can understand them.
The problem statement that we have chosen to understand the working better is noise cancellation. We have selected this example since we know its inner workings, and we will try to learn how the ML algorithm can learn this.

Experiments:

In noise cancellation, if we are given the signals, we take the average of the value corresponding to the same points on the signal, and the resultant denoised signal will take the average value. So our problem reduces to taking an average of numbers.
For simplicity, we have considered only two signals. Hence we have to provide a dataset with two inputs and the output will be the average of the two inputs.

Firstly we tested our model by providing an input of 100 data points.
The training dataset with 100 data points is as follows
https://docs.google.com/spreadsheets/d/1SKjyuQth-1YI_l8IVHtJ1jJe3T2TDWuxIhUrGW6yypY/edit?usp=sharing
The test dataset in each of our models is as follows
https://docs.google.com/spreadsheets/d/10zi_psLeZiiU9OWBm2y6dLr5QcJeXaRTu-oTd0v4g0k/edit?usp=sharing

So we trained our model to learn these hundred data points with a single layer of input and output without any hidden layer.



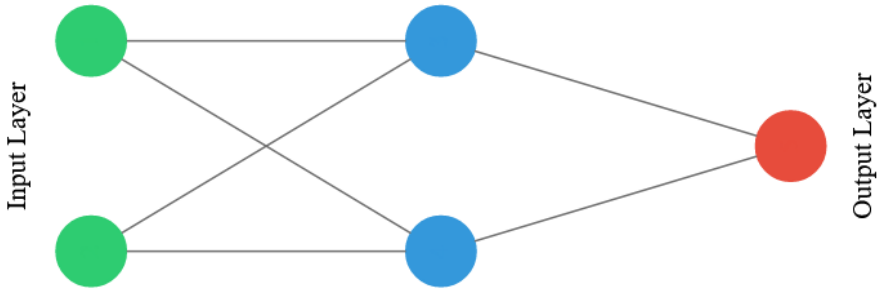
Also, we have not introduced any non-linearity in this model.
The code for the same is
https://colab.research.google.com/drive/179sRwzH6vTVNj5k0-OJHSk_nyfjF6mdK?usp=sharing

The weights obtained were

```
[array([[0.49990925],
        [0.49988994]]
```

Which are the desired weights as we take the average of two numbers. So we had to obtain 0.5, which is also obtained as weights of the neural network.

Next, we introduce a hidden layer with two neurons in them without introducing any non-linearity

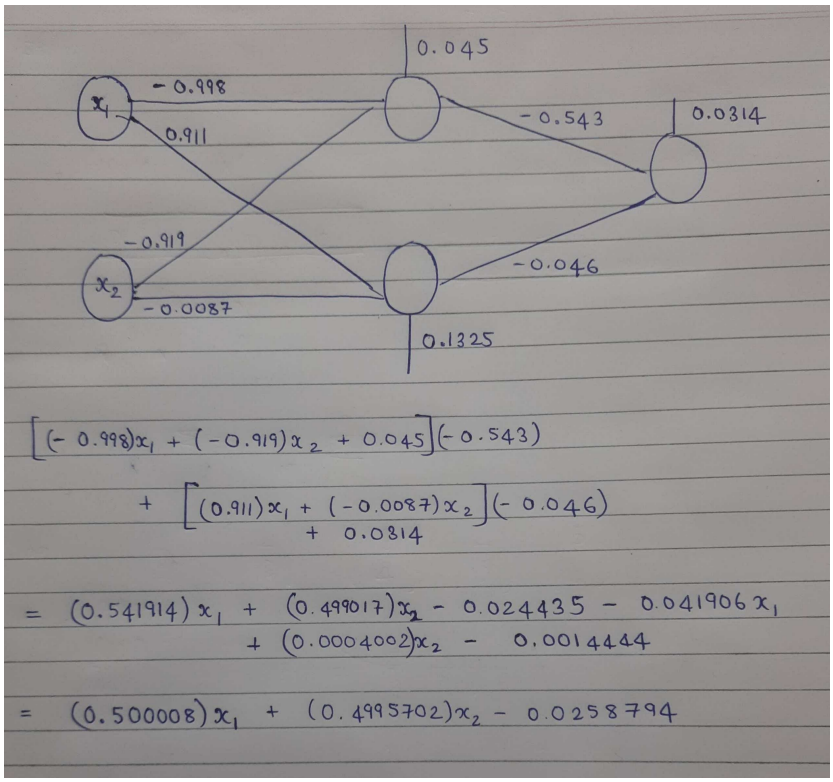


Here, after training the above model for 100 data points we obtain the following 6 weights

```
[array([[-0.9988957 ,  0.9115907 ],
        [-0.91960335, -0.00878043]], dtype=float32),
 array([0.04557306, 0.1325276 ], dtype=float32),
 array([[-0.5432646 ],
        [-0.04680281]], dtype=float32),
 array([0.0314841], dtype=float32)]
```

The code for this model is
https://colab.research.google.com/drive/1-NhcQxrN1bWEjpT6jw-W8b-4S_W7oc8z?usp=sharing

Here we can observe that these weights, after simplification eventually limited down to 0.5 x1 + 0.5 x2.



Further, the following experiment was to check how the weights vary with different sizes of datasets.
The dataset size used is 100 data points, 1000 data points, and 50000 data points. Here we kept the model to have one hidden layer with two neurons with linear activation function, same as in the previous case.

Here we have made sure that we have the same initial weights in each case. Also, the epochs in the case of 100 data points are the highest, i.e., 1500 epochs; in the case of 1000 data

points, we have chosen it to be 290 epochs, and for 50000 data points, it is ten epochs so that we obtain somewhat similar validation loss.
The test dataset for each of the cases is below:
https://docs.google.com/spreadsheets/d/10zi_psLeZiiU9OWBm2y6dLr5QcJeXaRTu-oTd0v4g0k/edit?usp=sharing

The link for the train data set and the code for it is :

1. 100 data points
   Train dataset:
   https://docs.google.com/spreadsheets/d/1SKjyuQth-1YI_l8IVHtJ1jJe3T2TDWuxIhUrGW6yypY/edit?usp=sharing
   Code :
   https://colab.research.google.com/drive/1-NhcQxrN1bWEjpT6jw-W8b-4S_W7oc8z?usp=sharing

2. 1000 data points
   Train dataset:
   https://docs.google.com/spreadsheets/d/1OdZi2FBqspHN6rGix4ltdPq5gvjISF8no3zSQD6dkio/edit?usp=sharing
   Code:
   https://colab.research.google.com/drive/1guMSLFkqgj3DZhNwWSenqtj2D6cUVW6_?usp=sharing

3. 50,000 data points
   Train dataset:
   https://docs.google.com/spreadsheets/d/1vNZuL4_PnYEZPXBNb9lFwzmAhtsOxINekadS4iTMH-M/edit?usp=sharing
   Code:
   https://colab.research.google.com/drive/18mrY__x7NUDju8OQT12exT48VY3tq5zF?usp=sharing

The weights obtained after training the model

1. 100 data points

```
[array([[-0.9988957 ,  0.9115907 ],
        [-0.91960335, -0.00878043]], dtype=float32),
 array([0.04557306, 0.1325276 ], dtype=float32),
 array([[-0.5432646 ],
        [-0.04680281]], dtype=float32),
 array([0.0314841], dtype=float32)]
```

2. 1000 data points

```
[array([[-1.1814133,  0.9100471],
        [-0.7558331, -0.4891397]], dtype=float32),
 array([0.00658798, 0.25712112], dtype=float32),
 array([[-0.5527237 ],
        [-0.16811784]], dtype=float32),
 array([0.04686903], dtype=float32)]
```

3. 50,000 data points

```
[array([[-1.1765639 ,  0.91340226],
        [-0.7538843 , -0.48477736]], dtype=float32),
 array([0.02285358, 0.02589329], dtype=float32),
 array([[-0.5552866 ],
        [-0.16786695]], dtype=float32),
 array([0.01703758], dtype=float32)]
```

We can observe that the weights in the case of 1000 data points and 50,000 data points are the same, whereas, in the case of 100 data points, there is a slight difference in weight in the last weight of each array.

The next experiment was about normalizing the data between 1 to -1 and training the dataset on the same model. We find that we get very similar weights in this case.

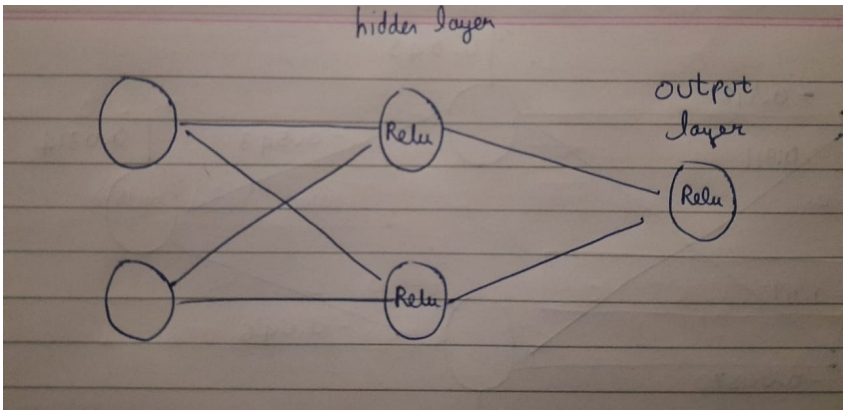
Normalized dataset

```
[array([[-1.2009094 ,  0.90526986],
        [-0.73556   , -0.5304637 ]], dtype=float32),
 array([ 0.01083755, -0.00360826], dtype=float32),
 array([[-0.5510847 ],
        [-0.17851634]], dtype=float32),
 array([0.00519986], dtype=float32)]
```

The code for the same is :
https://colab.research.google.com/drive/1hFnhJxPhMxhUg47QL7IGmM-doIWLRUmS?usp=sharing

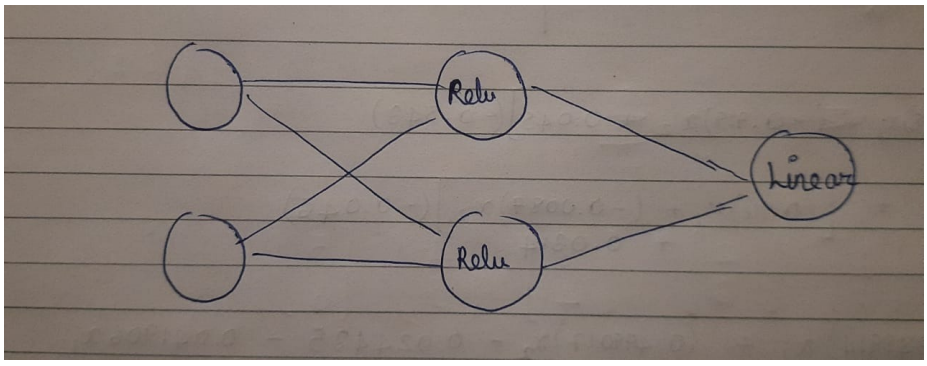
In the next experiment, we introduced nonlinearity using the activation function ReLu. The data set used for this experiment is 50,000 data points.

Firstly we train our model in which every neuron of the hidden layer and output layer has ReLu as the activation function.



The code for the same is :
https://colab.research.google.com/drive/1nlfshUVULnLRcMA9YFRcqNZB0cVn7BdL?usp=sharing

Here we can observe that the model cannot train, which can be understood as the validation loss remains the same. This can be explained as taking out the average is entirely linear, and introducing nonlinearity in all the neurons will not help the model learn.
We introduce a linear activation function in the output layer and ReLu activation function in the hidden layer to solve this problem.

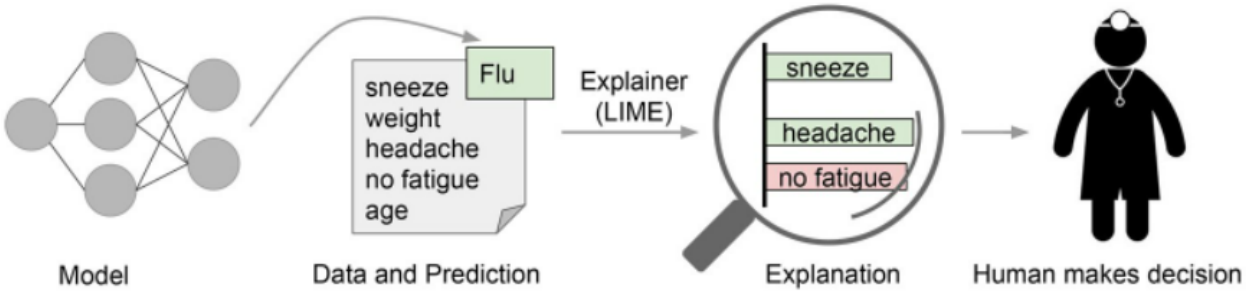


The model thus can learn and provide output.
This model also gave the same weights when we changed the size of the dataset and also in case of normalized data. However as we increase the number of nodes and layers the complexity of the model increases and we lose interpretability of the model.

Next we tried using some established method in the context of understandable AI.

LIME

LIME, the acronym for local interpretable model-agnostic explanations, is a technique that approximates any black box machine learning model with a local, interpretable model to explain each individual prediction. It tries to explain 'why was this prediction made or which variables caused the prediction?'.

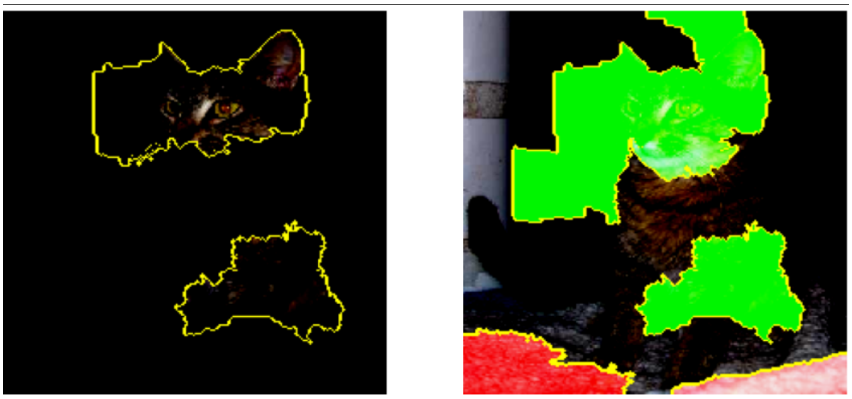


LIME attempts to play the role of the 'explainer', explaining predictions for each data sample. Source

The output of LIME is the list of explanations, reflecting the contribution of each feature to the prediction of a data sample. This provides local interpretability and it also allows us to determine which feature changes will have the most impact on the prediction.

The idea behind lime is that it creates an explanation by approximating the underlying model locally by an interpretable one. These interpretable models are the linear models. They are trained on small perturbations of the original instance and should only provide a good local approximation. The dataset is created by adding noise to continuous features. By only approximating the black box locally the task is significantly simplified.

Using LIME on Images



In the case of images, LIME will generate several samples that are similar with our input image by turning on and off some of the super-pixels of the image. Next, LIME will predict the class of each of the artificial data points that has been generated using our trained model. Further the weight of each artificial data point is calculated and a linear classifier is fit to explain the most important features.

We used this lime analysis for our current model which is averaging and our expectation was that the importance of each feature will be the same.
So first we trained our model for 50,000 data points and ran it for 1250 epochs.
The code instances for LIME is as follows:

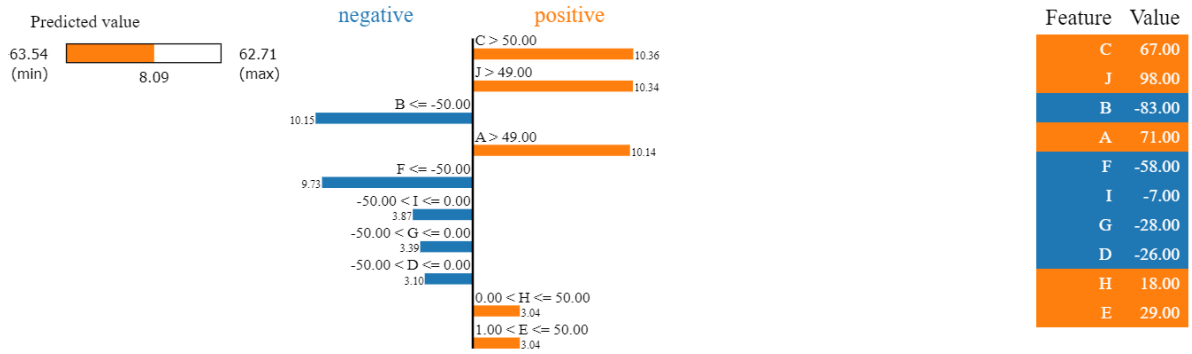
```
import lime
import lime.lime_tabular
```

```
explainer = lime.lime_tabular.LimeTabularExplainer(train_arr, feature_names=arr_1, verbose=True, mode='regression')
```

```
i=14
exp = explainer.explain_instance(test_arr[i], model.predict, num_features=10)
```

```
Intercept -1.7948094887827608
Prediction_local [4.88323357]
Right: 8.094635
```

```
exp.show_in_notebook(show_table=True)
```



We tried to understand what this graph represents and these are some points that we understood.
It has separated the negative and positive values as blue and orange. Basically it is indicating what effect the feature has on our output, whether it is positive or negative.

Furthermore, each feature is associated with a value which for eg. is 10.36 for C or 10.34 for J. This magnitude of the values indicates the impact of that particular feature on the output that we obtained.

Then we used the random forest which is a more understandable model and tried to understand the feature importance.
First we trained the model using a random forest model.
Further we used some inbuilt functions to find the feature importance

```
from sklearn.metrics import r2_score
from rfpimp import permutation_importances

def r2(rf, X_train, y_train):
    return r2_score(y_train, rf.predict(X_train))

perm_imp_rfpimp = permutation_importances(rf, train, labels_train, r2)
```

The feature importance came out to be as expected to be the same for all features.

Feature	Importance
I	0.199334
A	0.198119
D	0.197316
C	0.195021
E	0.194979
F	0.192422
B	0.189569
G	0.189490
H	0.185793
J	0.181855

So for the white box models like random forest it becomes easy for us to interpret what the model is doing and why we are getting a certain output. We can also predict what changes we can expect if we change any parameter.

Conclusion:

The black box model does work well and give us good accuracy however the interpretation part is what we lose.
So we should consider this while working with such models in different kinds of applications.