Introduction to Data Analytics
ECE9603

#Assignment-2 on Neural Networks
Paris Housing Classification

By Vivitha AnandaKrishnan (251274261)

Problem Description:

The problem focuses on classifying houses in the urban environment in Paris. This dataset gives an insight of Parisian Real Estate market on both Luxury and Basic housing options.

To make it simpler for purchasers to choose which home best suits their needs, it is crucial to categorise, label, and display the properties in their appropriate categories. Given the amount of data available, manually classifying them into their respective categories can be very challenging.

We are attempting to categorize a home as either a Luxury or a Basic property based on the data that is currently available. This is a Binary classification problem and issues like these can be effectively solved to a huge extent through the effective use of Neural Networks.

Data Description:

The available data is obtained from an open-source dataset from Kaggle. The dataset consists of 10,000 entries were then classified into the following 2 classes:

- Luxury
- Basic

The various attributes that will help in predicting the house type are:

- squareMeters
- numberOfRooms
- hasYard
- hasPool
- floors
- cityCode
- cityPartRange
- numPrevOwners
- hasGuestRoom
- price
- category

Overview of the Network Architecture:

I have chosen to work with a basic Deep Neural Network architecture to resolve this problem. Here I have used 4 hidden layers, and with the output layer having one neuron that gives the probability of class “1”.

Since this is a binary classification problem, the output layer only has one neuron. If there are multiple classes, we must pick the number of neurons to use based on the number of classes. For example, if there are five classes, the output layer will have five neurons, each of which provides the likelihood of that class; the class with the highest probability becomes the final answer.

Neurons can be thought of as nodes through which information and computations flow. It is a mathematical operation that gathers and categorizes data in accordance with a particular design.

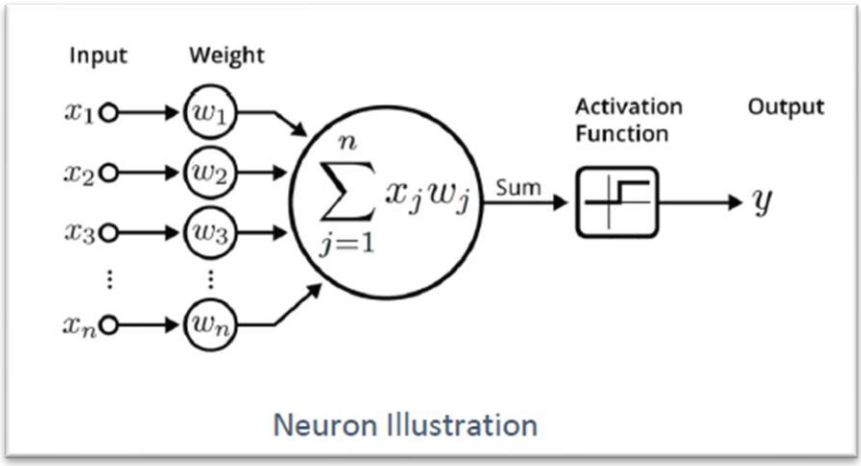

Figure 1 Illustration of a Neuron

As Deep neural network is nothing but a neural Network with at least two layers and some degree of complexity, let's try to briefly understand how neural networks work.

A neural network is loosely modelled after the human brain and is made up of thousands of simple processing nodes that are densely interconnected. They are structured into layers of nodes and are "feed-forward," which means that data only flows in one direction through them. An individual node may be linked to multiple nodes in the layer beneath it, from which it receives data, as well as multiple nodes in the layer above it, to which it sends data.

A node will assign a numerical value known as a "weight" to each of its incoming connections. It multiplies the number received in each of its connections by the associated "weight". The node then sums the resulting products to get a single number. If that number falls below a certain threshold value, it does not send data to the next layer. It "fires" if the number exceeds the threshold value.

The training data is fed to the input layer, which then traverses through the following layers, getting multiplied and added together in intricate ways, until it finally arrives at the output layer. The layers between the input and the output are called Hidden layers. A neural network with multiple hidden layers is called a Deep Neural Network.

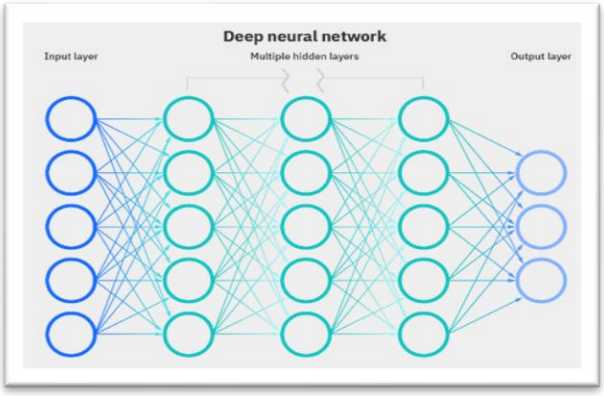

Figure 2 Illustration of a Deep Neural Network (DNN)

For this dataset, I have used 5 hidden layers initially and tuned their hyperparameters which controls and determines the learning process. Now let us look at all the important hyperparameters that affects our network operation:

- units=10: This means the layer is fed with 10 neurons as input.
- input_dim=10: This denotes predictors in the input data which is expected by the first layer. The Sequential model passes this information to the further layers and hence, we don't specify this parameter for other layers expect for the input.
- kernel_initializer='he_normal': This parameter denotes the algorithm that decides the value for each weight. We can choose different values for it like 'uniform' or 'glorot_uniform'.
- activation='relu': This is the activation function which squishes a range of values into a specific range inside each neuron. We can choose values like 'sigmoid', 'tanh', 'relu', etc.
- optimizer='adam': This parameter aids in determining the best values for each weight in the neural network. Adam is a stochastic gradient descent replacement optimization algorithm for deep learning model training.
- batch_size=10: This specifies the number of rows that will be passed to the Network in a single pass before the calculation begins and the neural network begins tweaking its weights based on the errors.
  1-epoch is when all the rows are passed in batches of ten rows as specified in this parameter, which is also called as mini-batch gradient descent. A small batch size value causes the ANN to examine the data slowly, which may result in overfitting, whereas a large value causes the ANN to examine the data quickly, which may result in underfitting. Hence an appropriate value must be determined through hyperparameter tuning.
- Epochs=50: As specified by this parameter, the same activity of adjusting weights is repeated 50 times. Simply put, the ANN examines the entire training data set 50 times and adjusts its weights.

For best results, here I have tuned Units, batch_size and Epochs using Manual Grid Search Method which is easily adaptable to meet our needs. This method gives us the flexibility to decide how many iterations are required and add another nested for-loop if needed.

Specifics about how algorithms were applied and the evaluation procedure:

Data Pre-processing:

First of all, to access data from the CSV file, I used the read_csv() method of Pandas library that retrieves data in the form of the Dataframe.

Next, I checked for duplicate rows and dropped them if any were present. We can see from below that there were no duplicate rows present.

```
Shape before deleting duplicate values: (10000, 11)
Shape After deleting duplicate values: (10000, 11)
```

Figure 3 Output showing that there are no duplicate records

Then I checked for missing /null data. The output shows that no null values are present in any of the columns

```
squareMeters      0
numberOfRooms     0
hasYard           0
hasPool           0
floors            0
cityCode          0
cityPartRange     0
numPrevOwners     0
hasGuestRoom      0
price             0
category          0
dtype: int64
```

Figure 4 Output showing that there are no Null data

In the next step, using the info() method of the same library, I verified again if there are only non-null values present for all of the features. The output clearly indicates that there are no null values in the dataset.

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 10000 entries, 0 to 9999
Data columns (total 11 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   squareMeters    10000 non-null  int64
 1   numberOfRooms   10000 non-null  int64
 2   hasYard         10000 non-null  int64
 3   hasPool         10000 non-null  int64
 4   floors          10000 non-null  int64
 5   cityCode        10000 non-null  int64
 6   cityPartRange   10000 non-null  int64
 7   numPrevOwners   10000 non-null  int64
 8   hasGuestRoom    10000 non-null  int64
 9   price           10000 non-null  float64
 10  category        10000 non-null  int64
dtypes: float64(1), int64(10)
memory usage: 937.5 KB
```

Figure 5 Non-null count and datatype of each feature

Further, to understand the dataset more and to know about its distribution, I used data.describe().T. It can be noticed from the output that all the features fall into different ranges. This must be scaled to the same range in order to fit into a model.

	count	mean	std	min	25%	50%	75%	max
squareMeters	10000.0	4.987013e+04	2.877438e+04	89.0	25098.50	50105.5	74609.75	99999.0
numberOfRooms	10000.0	5.035840e+01	2.881670e+01	1.0	25.00	50.0	75.00	100.0
hasYard	10000.0	5.087000e-01	4.999493e-01	0.0	0.00	1.0	1.00	1.0
hasPool	10000.0	4.968000e-01	5.000148e-01	0.0	0.00	0.0	1.00	1.0
floors	10000.0	5.027630e+01	2.888917e+01	1.0	25.00	50.0	76.00	100.0
cityCode	10000.0	5.022549e+04	2.900668e+04	3.0	24693.75	50693.0	75683.25	99953.0
cityPartRange	10000.0	5.510100e+00	2.872024e+00	1.0	3.00	5.0	8.00	10.0
numPrevOwners	10000.0	5.521700e+00	2.856667e+00	1.0	3.00	5.0	8.00	10.0
hasGuestRoom	10000.0	4.994600e+00	3.176410e+00	0.0	2.00	5.0	8.00	10.0
price	10000.0	4.993448e+06	2.877424e+06	10313.5	2516401.95	5016180.3	7469092.45	10006771.2
category	10000.0	1.265000e-01	3.324286e-01	0.0	0.00	0.0	0.00	1.0

Figure 6 Summary of data in each feature

The feature scaling step is implemented as part of data preprocessing and is achieved using preprocessing.StandardScaler() method which standardizes the features and fits them into the same range.

Next, I plotted the heatmap for the features using the seaborn library to visualize the correlation. Each cell in the matrix depicts the relationship between two variables.

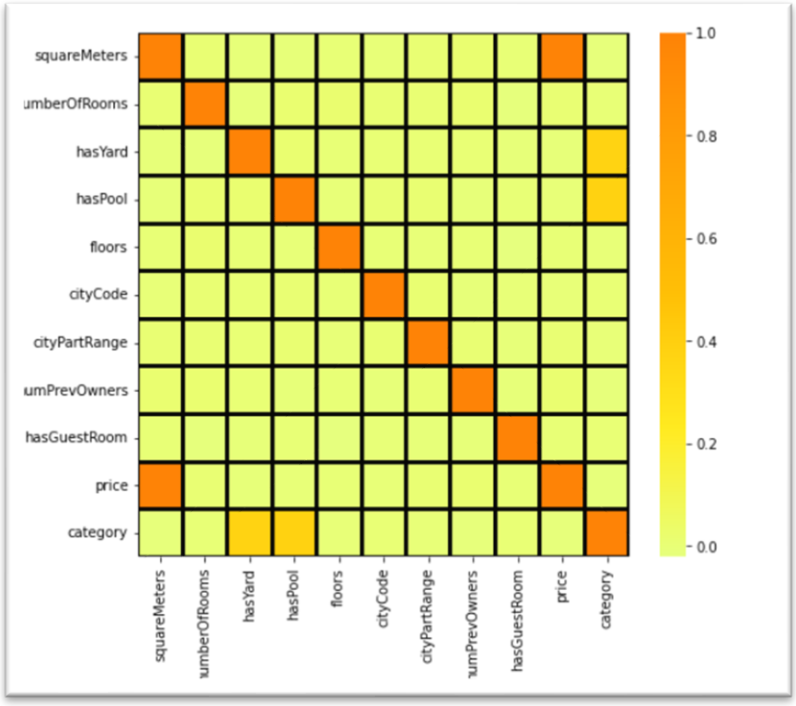

Figure 7 Correlation Heatmap

Model Training/Testing/ fitting:

Now moving on to fitting and training the model, the data is first separated into “Features and “Target”, which are then split into two parts – a training set and a testing set, with test size =0.33. This means that 33 percent of the data will be set aside for testing and the model will be trained on the rest of the

set. i.e, out of the 10000 records that we have, 3300 records are put in testing set and 6700 are put into the training set. When we print the shape of both sets, we get the below output:

```
(6700, 9)
(6700,)
(3300, 9)
(3300.)
```

Figure 8 Splitting of dataset into Training and Testing sets

The model is fit into Random Forest Classifier and the below target values are predicted on the testing set. Results are as below:

```
---------Random Forest Classifier Results ----------
Accuracy       :  0.866969696969697
Recall         :  0.866969696969697
Precision      :  0.8665582399988916
F1 Score       :  0.866969696969697
```

Figure 9 Random Forest Classifier result (untuned model)

The quality of the model is evaluated using hold-out validation process, in which a predictive model is built using only the training set.

Deep Neural Network Model:

I have implemented the Deep Neural Network using sequential model of Keras library, which is appropriate for stacking up multiple layers.

Initially, I built the NN with 5 layers – 1 input layer, 3 hidden layers and 1 output layer. The hyperparameters defined for each layer is as below:

Layers	Units (neurons)	Activation function
1	10	relu
2	15	relu
3	20	relu
4	10	relu
5	1	softmax

Table 1 Hyperparameter values for untuned DNN model

I have used “he_normal” kernel for all the layers, define the way to set the initial random weights of Keras layers.

After defining the model, I compiled it and fit it on the dataset with batch_size =32, and plotted the Accuracy and loss values against epoch using matplotlib library.

The first graph below shows the plot of Accuracy values and the one beneath that depicts the loss values:

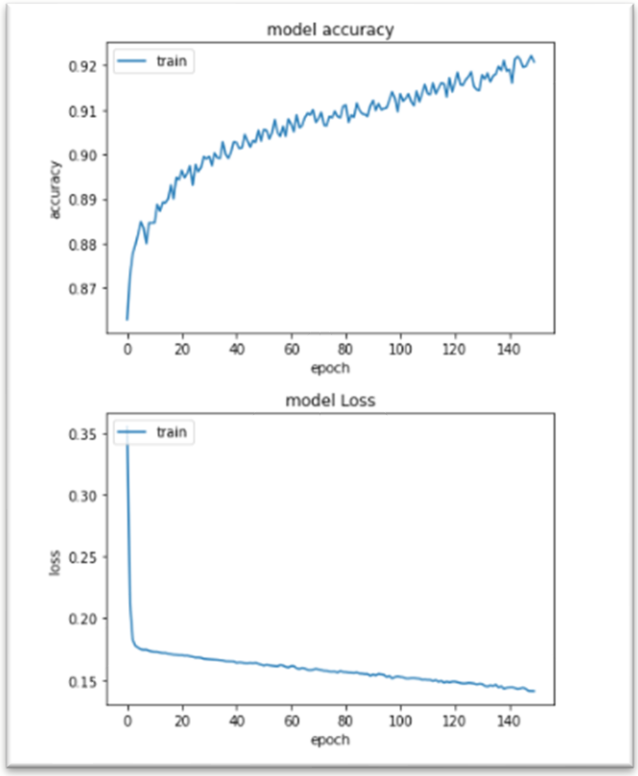

Figure 10 Accuracy and Loss values of an Untuned DNN Model

When evaluated on the testing set, the model Accuracy turns out to be 0.872.

Hyper Parameter tuning of Deep Neural Network Model:

We have tuned the above model by adding one more hidden layer and evaluating it for various combination of values for *units, batch_size and epochs.*

Values considered for the above mentioned hyperparameters are as follows

Hyperparameter	Values
batch_size	5, 10, 15, 20
epoch	5, 10, 50, 100
unit - input layer (n)	3, 5, 10, 15, 20

Table 2 Hyperparameter values for tuning the DNN model

For the hidden layers, the unit varies with respect to the unit fed to input layer in the below fashion for each iteration:

1st hidden layer – n+5

2nd hidden layer – n+10

3rd hidden layer – n+5

4th hidden layer – n

The parameters are tuned by a manual grid search algorithm that stores results of each iteration for both testing and training sets in respective data frames. The accuracy from the stored results is plotted against the parameters and the best of them is displayed as an output.

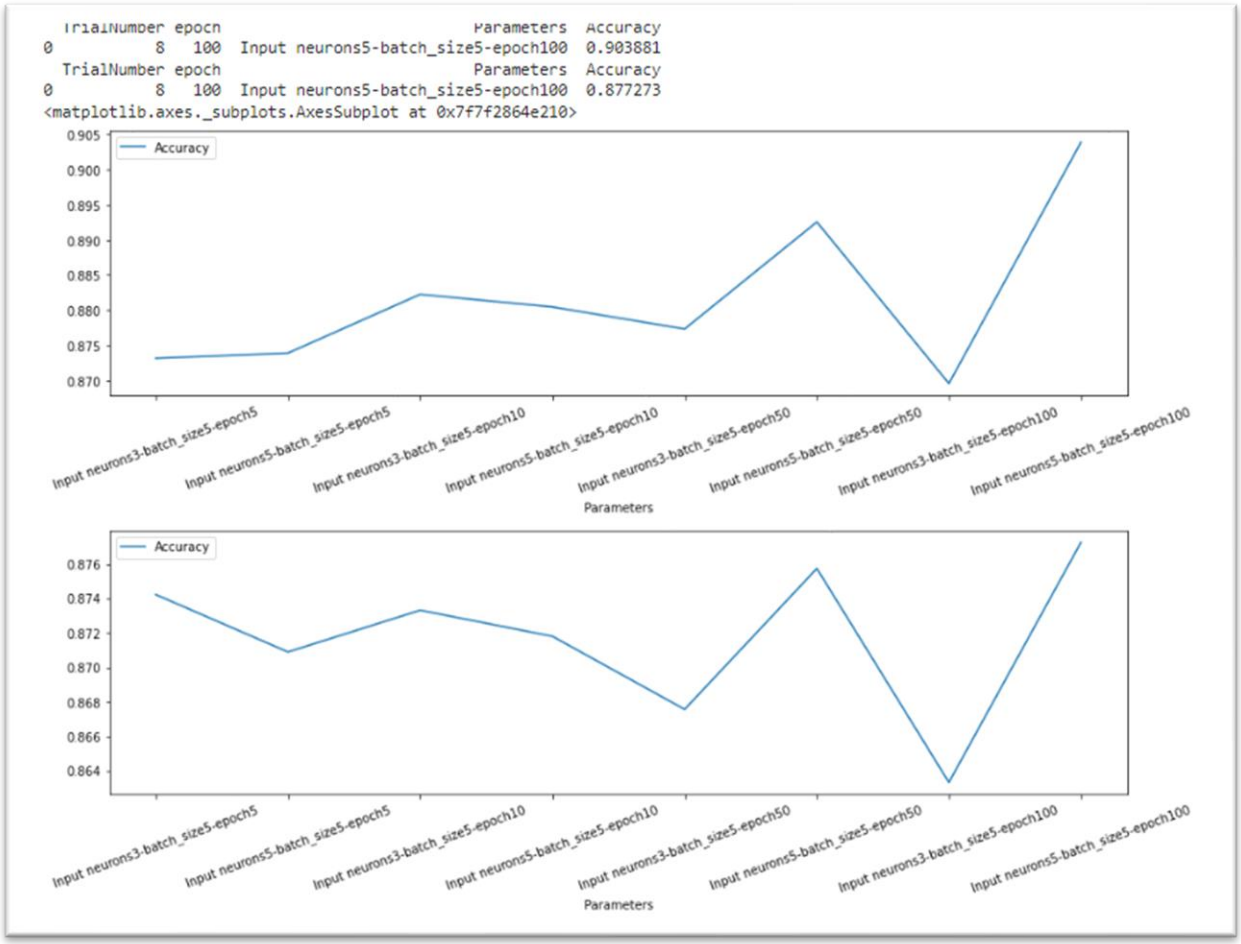

Figure 11 Accuracy values plotted against Hyperparameters for Training and testing sets respectively

It can be seen the output that the best results are achieved for accuracy for the following parameter values:

Unit- Input layer – 5

Batch_ size – 5

Epoch – 100

Layers – 6

Comparison of results:

Models	Accuracy
Random Tree Classifier	86.69
DNN model - Before Tuning	87.72
DNN model - After Tuning	90.38

Table 3 Comparison of results obtained from multiple models

From the above table it is evident that a tuned DNN model delivers the best results than an untuned model. Random Tree Classifier has the least accuracy rate of all the three models taken into consideration.

Though hyperparameter tuning is cumbersome and time-consuming process, it definitely enhances the performance of the model to a great extent.