CSE 343: Machine Learning

Assignment 2 | Report

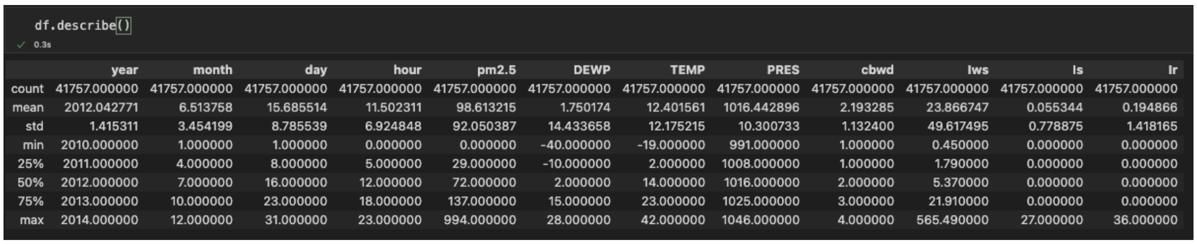
Question 1. Decision Trees and Random Forest

1 **Loading the dataset**

- The dataset was loaded and converted to dataframe using the pandas library.

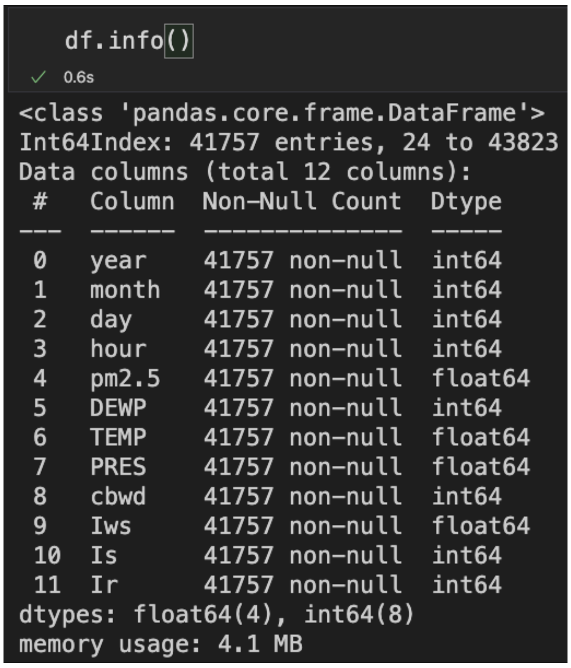
2 **Preprocessing the data**

- ‘No’ column was dropped from the dataframe
- A total of 2067 entries corresponding to ‘na’ in the pm2.5 column were found.
- There were no other column with any nan values.
- The model was trained by:
  - Removing the rows corresponding to the nan values.
  - Replacing the nan values with their mean values.
  - Replacing the nan values with their median values.
- Since the number of nan values were relatively small, all three of the above gave approximately similar results.
- In the end, I decided to follow the first approach, i.e. to remove the rows corresponding to the nan values as the data was spread over different months, hours of the day, and years.
- Replacing the values with mean or median did not feel suitable to me as the pm2.5 vales depend a lot on the above 3 mentioned factors.
- Thus the decision to remove them completely was taken.
- Final dataset shape was found to be (41757, 12)
- The entries in the ‘cbwd’ column had textual entries rather than numerical ones.
- These were changed to numeric data.
- Basic Info about the dataset was obtained:



```
df.describe()
✓ 0.3s
```

	year	month	day	hour	pm2.5	DEWP	TEMP	PRES	cbwd	Iws	Is	Ir
count	41757.000000	41757.000000	41757.000000	41757.000000	41757.000000	41757.000000	41757.000000	41757.000000	41757.000000	41757.000000	41757.000000	41757.000000
mean	2012.042771	6.513758	15.685514	11.502311	98.613215	1.750174	12.401561	1016.442896	2.193285	23.866747	0.055344	0.194866
std	1.415311	3.454199	8.785539	6.924848	92.050387	14.433658	12.175215	10.300733	1.132400	49.617495	0.778875	1.418165
min	2010.000000	1.000000	1.000000	0.000000	0.000000	-40.000000	-19.000000	991.000000	1.000000	0.450000	0.000000	0.000000
25%	2011.000000	4.000000	8.000000	5.000000	29.000000	-10.000000	2.000000	1008.000000	1.000000	1.790000	0.000000	0.000000
50%	2012.000000	7.000000	16.000000	12.000000	72.000000	2.000000	14.000000	1016.000000	2.000000	5.370000	0.000000	0.000000
75%	2013.000000	10.000000	23.000000	18.000000	137.000000	15.000000	23.000000	1025.000000	3.000000	21.910000	0.000000	0.000000
max	2014.000000	12.000000	31.000000	23.000000	994.000000	28.000000	42.000000	1046.000000	4.000000	565.490000	27.000000	36.000000

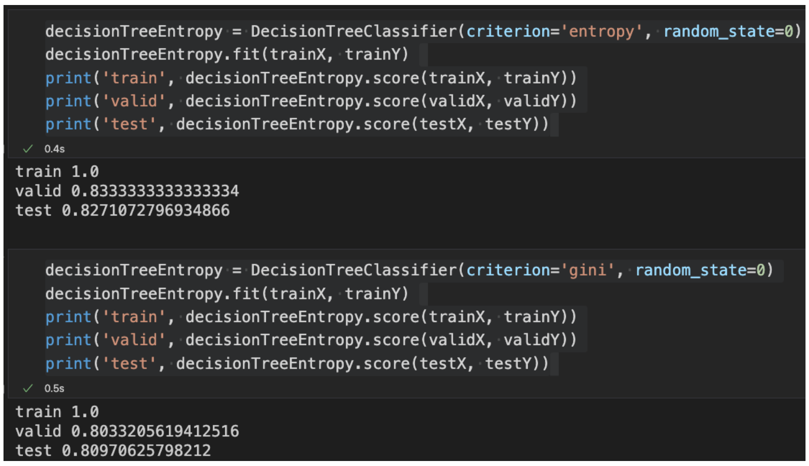


```
   df.info()
 ✓ 0.6s
<class 'pandas.core.frame.DataFrame'>
Int64Index: 41757 entries, 24 to 43823
Data columns (total 12 columns):
 #   Column  Non-Null Count  Dtype  
---  ------  --------------  -----  
 0   year    41757 non-null  int64  
 1   month   41757 non-null  int64  
 2   day     41757 non-null  int64  
 3   hour    41757 non-null  int64  
 4   pm2.5   41757 non-null  float64
 5   DEWP    41757 non-null  int64  
 6   TEMP    41757 non-null  float64
 7   PRES    41757 non-null  float64
 8   cbwd    41757 non-null  int64  
 9   Iws     41757 non-null  float64
 10  Is      41757 non-null  int64  
 11  Ir      41757 non-null  int64  
dtypes: float64(4), int64(8)
memory usage: 4.1 MB
```

- The data was divided into train:validation:test set in the ration 70:15:15
- After this, the above sets were divided into x and y sets for each.
- Since the decision tree and random forest are immune to the variance in the data and they perform in similar fashion for both normalized and non normalized data, the given dataset was not normalized.

3 Decision trees with gini index and entropy

- Decision Trees having criterion as entropy and gini index were trained.
- The accuracies on train, valid and test sets were calculated and were as follows:



```
decisionTreeEntropy = DecisionTreeClassifier(criterion='entropy', random_state=0)
decisionTreeEntropy.fit(trainX, trainY)
print('train', decisionTreeEntropy.score(trainX, trainY))
print('valid', decisionTreeEntropy.score(validX, validY))
print('test', decisionTreeEntropy.score(testX, testY))
```

```
train 1.0
valid 0.8333333333333334
test 0.8271072796934866
```

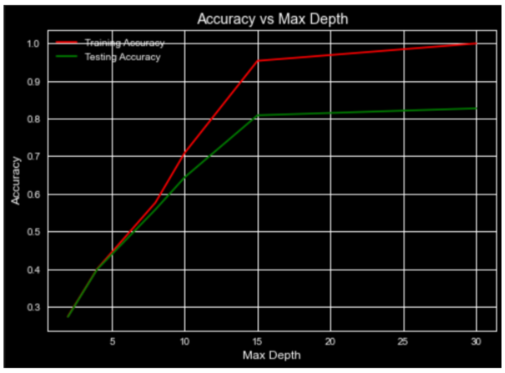
```
decisionTreeEntropy = DecisionTreeClassifier(criterion='gini', random_state=0)
decisionTreeEntropy.fit(trainX, trainY)
print('train', decisionTreeEntropy.score(trainX, trainY))
print('valid', decisionTreeEntropy.score(validX, validY))
print('test', decisionTreeEntropy.score(testX, testY))
```

```
train 1.0
valid 0.8033205619412516
test 0.80970625798212
```

- Courtesy to the above values obtained, 'entropy' was used for all the further evaluations.

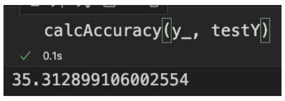
4 Decision Tree on different max depths

- The trees were trained with different max depths and corresponding accuracies were calculated.
- The curve for test and train set were plotted and were as follows.



5 Ensembling

- 100 different decision trees were created with max depth 3.
- 50% of the data was randomly selected and that was used to train the models.
- Predictions were made after taking the majority vote of the decision stumps.
- Using the above predictions, the accuracies were calculated on the test data which came out to be:



```
calcAccuracy(y_, testY)
```

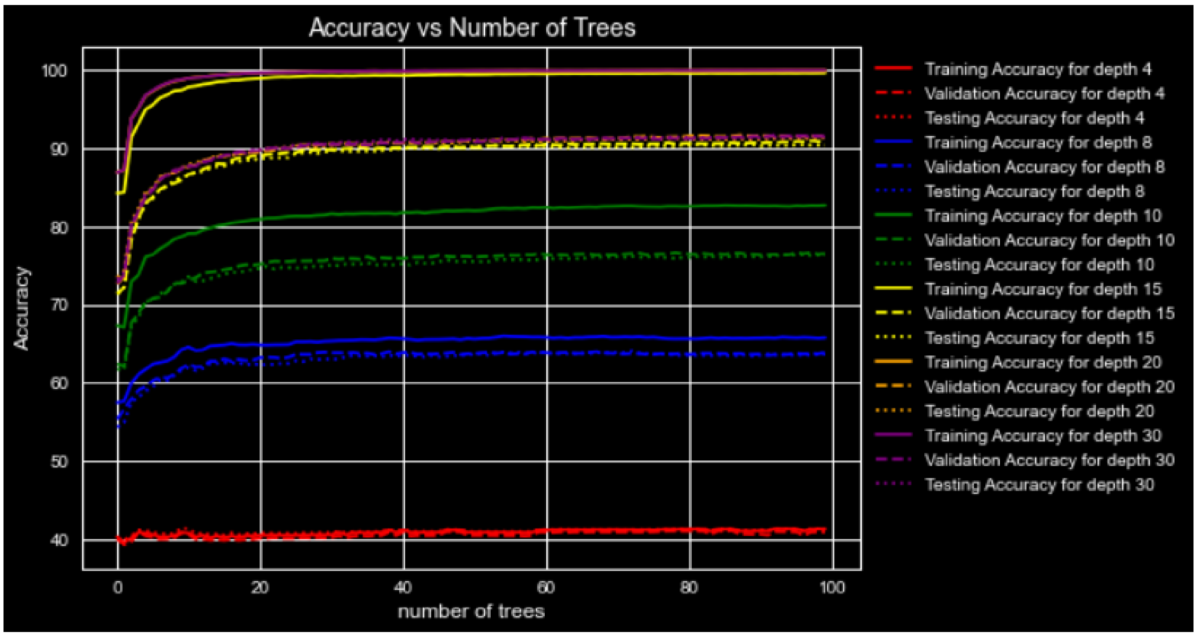
```
35.312899106002554
```

- In the first part, the model was free to chose the depth for itself. This even though could have lead to overfitting, still gave a pretty good accuracy for the given test set.
- In the second part, the max depth of the trees were fixed. For lower depths, the accuracies obtained were pretty low, but as the max depth increased, the training and the testing accuracies increased as well.
- In the current part, even though ensembling was done with 100 stumps, but the accuracy

obtained was not good enough as the max depth was restricted to just three. This restricted the model from applying various combinations on the input parameters, thus decreasing the accuracy. Still if we extrapolate the graph from the 2nd part, we can see that the accuracy obtained by the ensemble is significantly higher than the one obtained using just a single tree.

6 Random Forest with Different Depths

- In the b part, best accuracies were obtained on the max depth of 30, so that was included in the max depths of the current part.
- The ensemble model was trained on the different value of max depths and number of trees. For the output majority vote was obtained and the accuracies obtained were plotted as following:

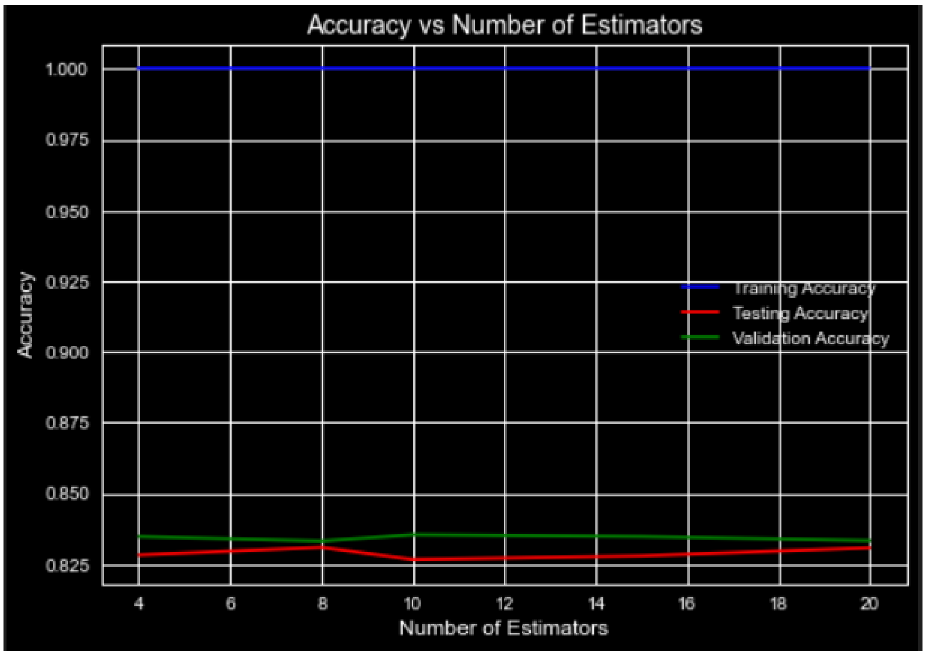


- As it is evident from the graph above, the best accuracies obtained were for the data with more number of depths.
- Results were similar for the model having about 20 or more trees and each tree having depth of about 15 or more.

7 Adaboost

- Sklearn’s implementation of adaboost was used.
- Different number of estimators were used to get the best values from the adaboost classifier.

- The different models were trained and the corresponding accuracies were generated in the form of a graph:



- The above graph shows the effect of estimators on the accuracies when using the adaboost model.
- The model here shows that there was likely overfitting as the underlying model of decision trees did not have any check on the number of depth of the trees being generated.
- Irrespective of that, the results were pretty good.
- There was a difference between the best results obtained on random forest and here because of the underlying decision tree. In the previous part, there was a check on the max depth which was not present here. This lead to excessive overfitting here which resulted in the evident loss of accuracy on the validation and testing set.

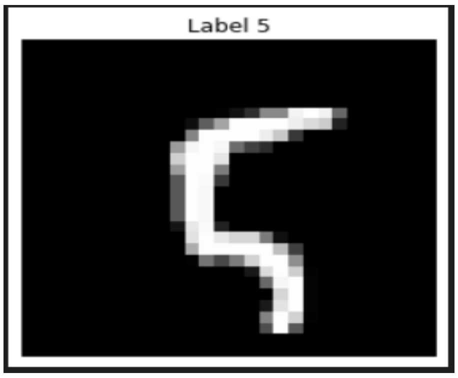
Question 2. MLP

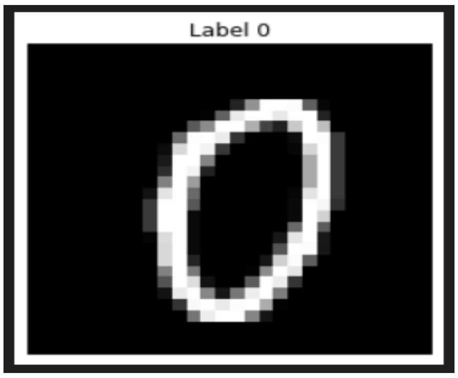
1 Neural Network Class

- The neural network class was implemented from scratch containing all the desired parameters, activation functions, weight initializations and class methods.

2 Dataset and Preprocessing

- The dataset was read using numpy library.
- The training set consisted of 60,000 samples and the testing set consisted of 10,000 samples.
- The two sets were merged together and the resultant set was divided into the train:test:validation sets in the ration 70:20:10
- Each image was a matrix of 28*28 and represented one number.
- Draw image function can be used to visualize any of the data samples.
- The images look like:





- The data was then normalized for using in the models.

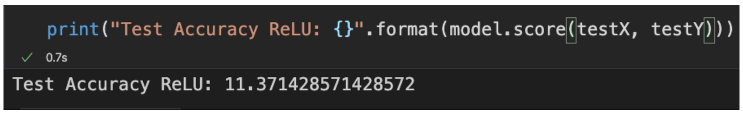
3 Training Model with different Activation Functions

- The model was trained with various activation functions.
- The parameters of the model were:
  - N_inputs = 784
  - N_outputs = 10
  - N_layers = 4 (Number of hidden layers)
  - Layer_sizes = [256, 128, 64, 32]
  - activation = activation function being used
  - learning_rate = 0.08
  - weight_init = normal

  - num_epochs = 150
  - batch_size = len(X)//2 (X is the vector)
- Each model's final weights were also stored.
- Their evaluations are as follows:

a ReLU:

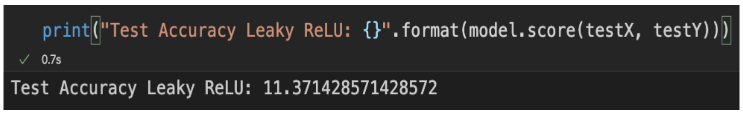
  - Test Accuracy obtained was:



```
print("Test Accuracy ReLU: {}".format(model.score(testX, testY)))
✓ 0.7s
Test Accuracy ReLU: 11.371428571428572
```

b Leaky ReLU:

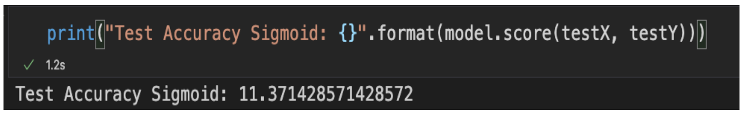
  - Test Accuracy obtained was:



```
print("Test Accuracy Leaky ReLU: {}".format(model.score(testX, testY)))
✓ 0.7s
Test Accuracy Leaky ReLU: 11.371428571428572
```

c Sigmoid:

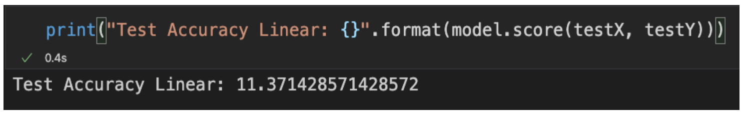
  - Test Accuracy obtained was:



```
print("Test Accuracy Sigmoid: {}".format(model.score(testX, testY)))
✓ 1.2s
Test Accuracy Sigmoid: 11.371428571428572
```

d Linear:

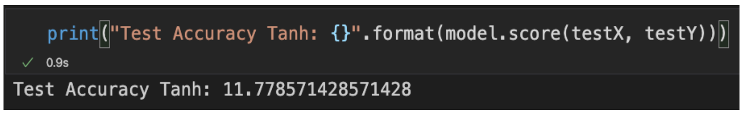
  - Test Accuracy obtained was:



```
print("Test Accuracy Linear: {}".format(model.score(testX, testY)))
✓ 0.4s
Test Accuracy Linear: 11.371428571428572
```

e tanh:

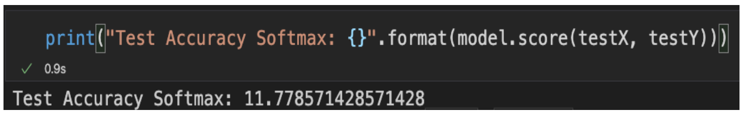
  - Before running tanh and softmax, the data was shuffled once again.
  - Test Accuracy obtained was:



```
print("Test Accuracy Tanh: {}".format(model.score(testX, testY)))
✓ 0.9s
Test Accuracy Tanh: 11.778571428571428
```

f softmax:

  - For softmax, due to my computational limitations, I was not able to use the actual softmax gradient to output the values. Instead I used the gradient format of sigmoid on softmax.
  - Both the codes however are present in the code file submitted.
  - Test Accuracy obtained was:



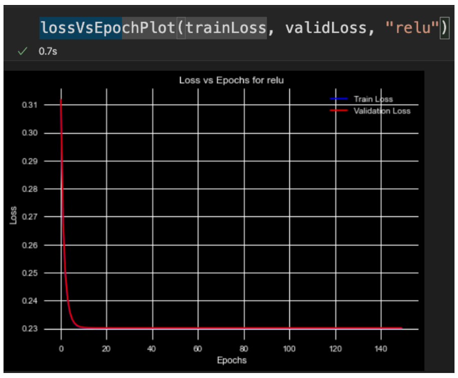
```
print("Test Accuracy Softmax: {}".format(model.score(testX, testY)))
✓ 0.9s
Test Accuracy Softmax: 11.778571428571428
```

4 Training vs Epochs Plot

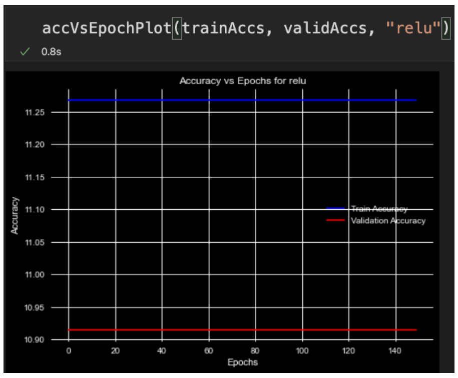
- The train vs epochs and accuracy vs epochs plots are as follows:

a ReLU

- The graphs can be seen below.
- By looking at the graphs one can deduce that the given model has converged, but by looking at the accuracy graph, it is evident that the performance of the model is not upto the mark.
- This scenario might have arised as the function might have found something of a flat plane with very less slope, but since the learning rate is a bit smaller, it might have not been able to emerge out of the same.
- Since any kind of optimizers are absent, the model just learns from the preset parameters and is not able to modify them during runtime.
- However, if the model is allowed to run for quite a few iterations more, it might just tip out from the current surface and achieve better results.



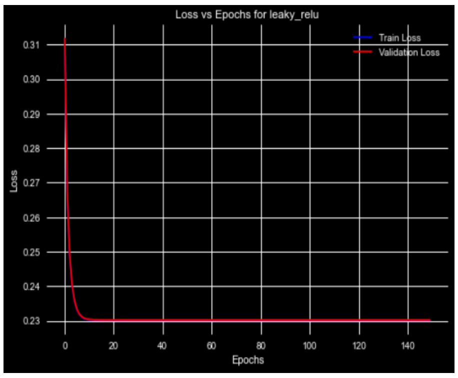
```
lossVsEpochPlot(trainLoss, validLoss, "relu")
✓ 0.7s
```

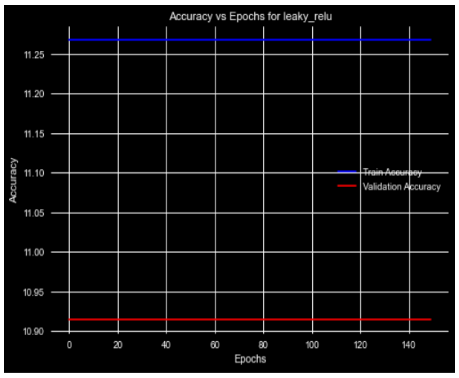


```
accVsEpochPlot(trainAccs, validAccs, "relu")
✓ 0.8s
```

b Leaky ReLU

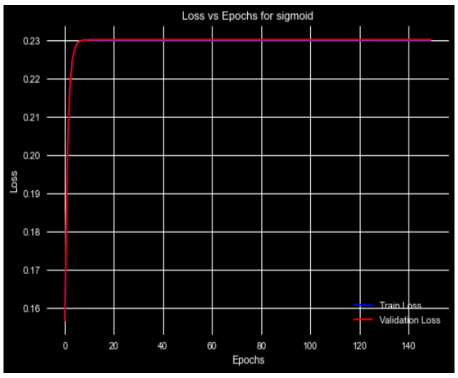
- The graphs can be seen below.
- By looking at the graphs one can deduce that the given model has converged, but by looking at the accuracy graph, it is evident that the performance of the model is not upto the mark.
- This scenario might have arised as the function might have found something of a flat plane with very less slope, but since the learning rate is a bit smaller, it might have not been able to emerge out of the same.
- Since any kind of optimizers are absent, the model just learns from the preset parameters and is not able to modify them during runtime.
- However, if the model is allowed to run for quite a few iterations more, it might just tip out from the current surface and achieve better results.

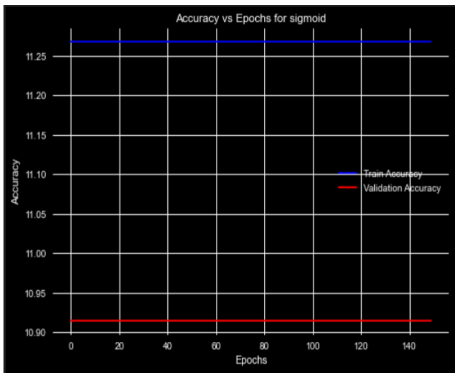




c Sigmoid

- The graphs can be seen below.
- Since the model is trained using mini batches, it might have happened that some of the batch might have increased the error a bit.
- As evident from the graph, the error was increased by a mere value of 0.07 in the initial iterations.
- I tried the same configurations with a higher learning rate and the convergence was being reached and the graph was opposite of what it is currently.
- After that, the scenario might have arised as the function might have found something of a flat plane with very less slope, but since the learning rate is a bit smaller, it might have not been able to emerge out of the same.
- Since any kind of optimizers are absent, the model just learns from the preset parameters and is not able to modify them during runtime.
- However, if the model is allowed to run for quite a few iterations more, it might just tip out from the current surface and achieve better results as it is already decreasing in the absolute loss values.

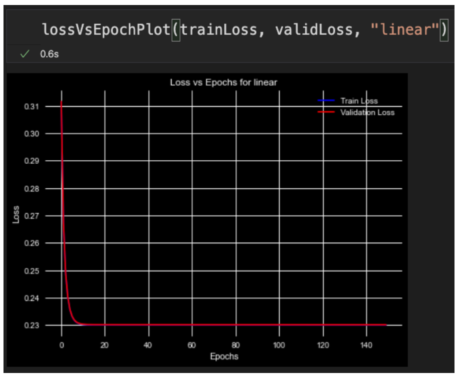




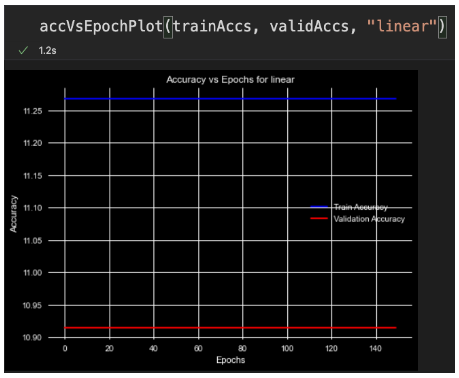
d Linear

- The graphs can be seen below.
- By looking at the graphs one can deduce that the given model has converged, but by looking at the accuracy graph, it is evident that the performance of the model is not upto the mark.

- This scenario might have arised as the function might have found something of a flat plane with very less slope, but since the learning rate is a bit smaller, it might have not been able to emerge out of the same.
- Since any kind of optimizers are absent, the model just learns from the preset parameters and is not able to modify them during runtime.
- However, if the model is allowed to run for quite a few iterations more, it might just tip out from the current surface and achieve better results.



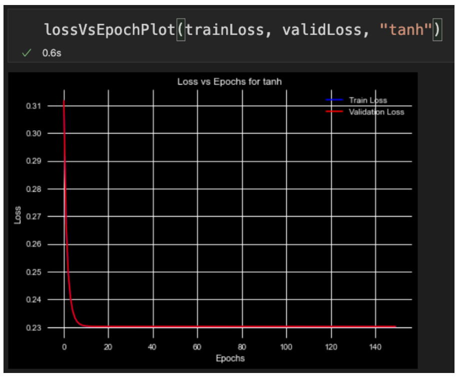
```
lossVsEpochPlot(trainLoss, validLoss, "linear")
```



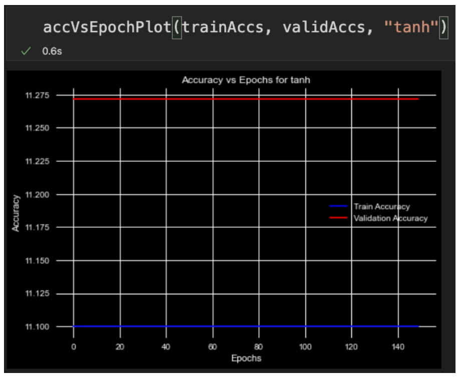
```
accVsEpochPlot(trainAccs, validAccs, "linear")
```

e Tanh:

- The graphs can be seen below.
- By looking at the graphs one can deduce that the given model has converged, but by looking at the accuracy graph, it is evident that the performance of the model is not upto the mark.
- This scenario might have arised as the function might have found something of a flat plane with very less slope, but since the learning rate is a bit smaller, it might have not been able to emerge out of the same.
- Since any kind of optimizers are absent, the model just learns from the preset parameters and is not able to modify them during runtime.
- However, if the model is allowed to run for quite a few iterations more, it might just tip out from the current surface and achieve better results.



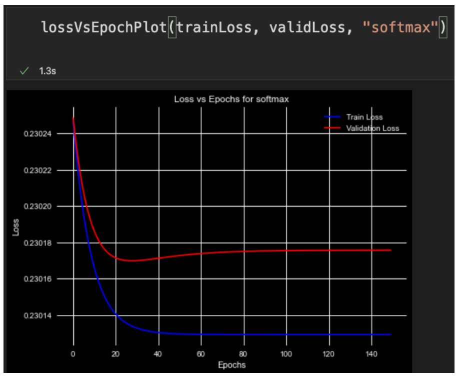
```
lossVsEpochPlot(trainLoss, validLoss, "tanh")
```



```
accVsEpochPlot(trainAccs, validAccs, "tanh")
```

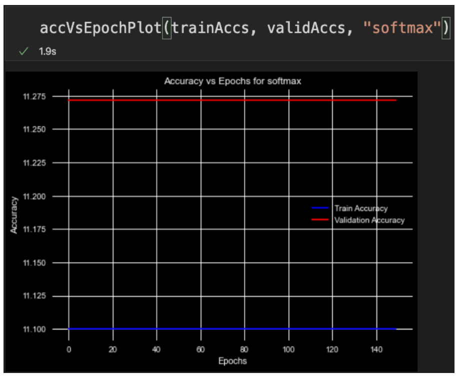
f Softmax:

- The graphs can be seen below.
- By looking at the graphs one can deduce that the given model has converged, but by looking at the accuracy graph, it is evident that the performance of the model is not upto the mark.
- This scenario might have arised as the function might have found something of a flat plane with very less slope, but since the learning rate is a bit smaller, it might have not been able to emerge out of the same.
- Since any kind of optimizers are absent, the model just learns from the preset parameters and is not able to modify them during runtime.
- However, if the model is allowed to run for quite a few iterations more, it might just tip out from the current surface and achieve better results.



```
lossVsEpochPlot(trainLoss, validLoss, "softmax")
```

✓ 1.3s



```
accVsEpochPlot(trainAccs, validAccs, "softmax")
```

✓ 1.9s

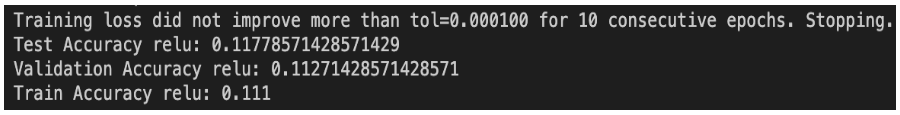
- Since none of the above have converged, it would not be correct to judge any one function from just the above data.
- However, I ran the functions for a few more epochs and found that for my current distribution, tanh starts converging the fastest with higher learning rates.
- Please note that here I am not even considering softmax for the best activation funtion as due to computation limitations, I had to use a wrong derivative of the function.

5 Best Function for the Output Layer

- In every case, the best function for the output layer should be softmax.
- This is because:
  - a Each output value ranges between 0 and 1
  - b Sum of all values is equal to 1
  - c Thus it can be considered as a probabilistic output.
  - d Due to the presence of exponents in its calculations, the difference between any 2 output values before the application of softmax would be profound in the final probabilistic like output.
  - e Thus the target class would have the highest probability thus making it easier for the computations and making the final predictions.

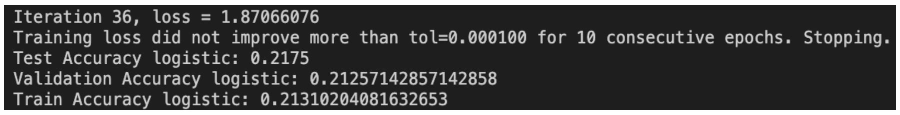
6 Sklearn Implementation

- sklearn's implementations were used for the following functions.
  - a ReLU:
    - Accuracies obtained:



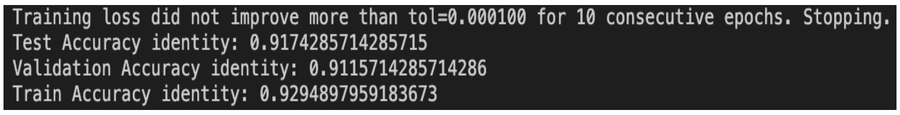
```
Training loss did not improve more than tol=0.000100 for 10 consecutive epochs. Stopping.
Test Accuracy relu: 0.11778571428571429
Validation Accuracy relu: 0.11271428571428571
Train Accuracy relu: 0.111
```

    - The accuracies obtained are more or less similar to what my model obtained on the data.
  - b Sigmoid:
    - Accuracies obtained:



```
Iteration 36, loss = 1.87066076
Training loss did not improve more than tol=0.000100 for 10 consecutive epochs. Stopping.
Test Accuracy logistic: 0.2175
Validation Accuracy logistic: 0.21257142857142858
Train Accuracy logistic: 0.21310204081632653
```

    - The accuracies obtained are a bit better than the ones my model obtained on the data.
    - This might be because my model might have stuck on some local minima.
    - Even though the loss values were slowly decreasing in my model, they were not fast enough to give meaningful results in just 150 epochs.
    - The better performance can also be credited to the high level of optimizations in the inbuilt model, along with the presence of optimizers and solvers in the inbuilt model.
  - c Linear:
    - Accuracies obtained:

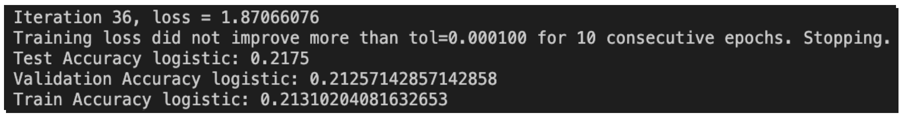


```
Training loss did not improve more than tol=0.000100 for 10 consecutive epochs. Stopping.
Test Accuracy identity: 0.9174285714285715
Validation Accuracy identity: 0.9115714285714286
Train Accuracy identity: 0.9294897959183673
```

- The accuracies obtained are a lot better than the ones my model obtained on the data.
- This might be because my model might have stuck on some local minima.
- Even though the loss values were slowly decreasing in my model, they were not fast enough to give meaningful results in just 150 epochs.
- The better performance can also be credited to the high level of optimizations in the inbuilt model, along with the presence of optimizers and solvers in the inbuilt model.

d Tanh:

- Accuracies obtained:



```
Iteration 36, loss = 1.87066076
Training loss did not improve more than tol=0.000100 for 10 consecutive epochs. Stopping.
Test Accuracy logistic: 0.2175
Validation Accuracy logistic: 0.21257142857142858
Train Accuracy logistic: 0.21310204081632653
```

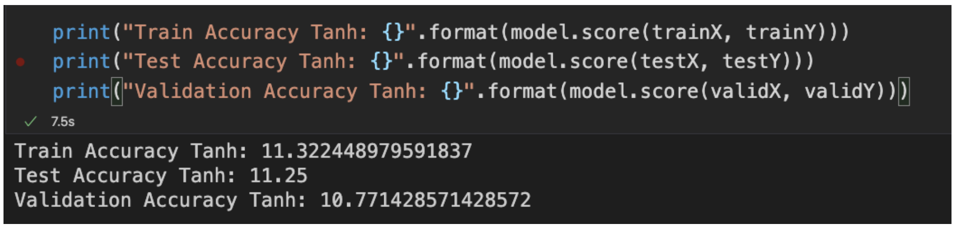
- The accuracies obtained are a lot better than the ones my model obtained on the data.
- This might be because my model might have stuck on some local minima.
- Even though the loss values were slowly decreasing in my model, they were not fast enough to give meaningful results in just 150 epochs.
- The better performance can also be credited to the high level of optimizations in the inbuilt model, along with the presence of optimizers and solvers in the inbuilt model.

7 Bonus Part (Changing the learning rates)

- tanh function was used

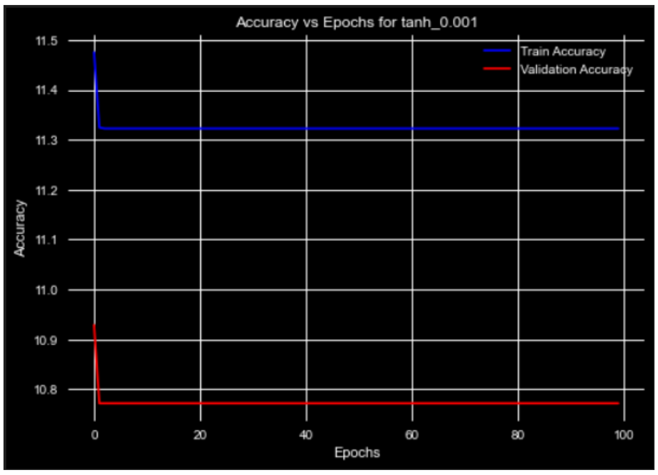
a learning_rate = 0.001, epochs = 100

- The Following accuracies were obtained:



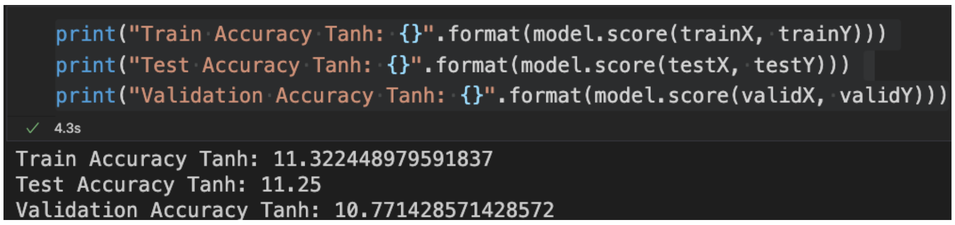
```
print("Train Accuracy Tanh: {}".format(model.score(trainX, trainY)))
print("Test Accuracy Tanh: {}".format(model.score(testX, testY)))
print("Validation Accuracy Tanh: {}".format(model.score(validX, validY)))
✓ 7.5s
Train Accuracy Tanh: 11.322448979591837
Test Accuracy Tanh: 11.25
Validation Accuracy Tanh: 10.771428571428572
```

- Accuracy vs epoch plot being:



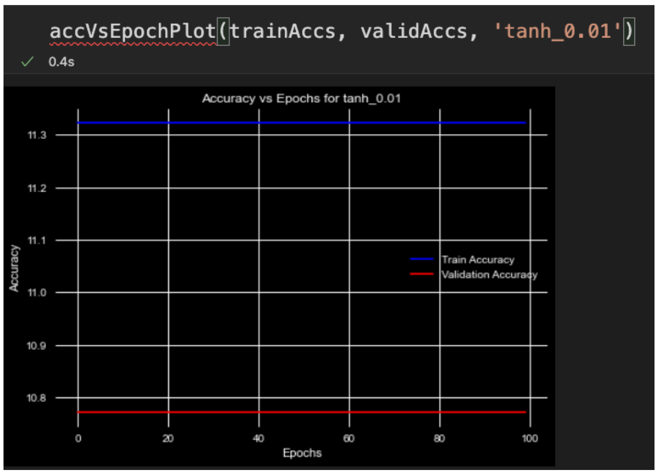
b learning_rate = 0.01, epochs = 100

- The Following accuracies were obtained:



```
print("Train Accuracy Tanh: {}".format(model.score(trainX, trainY)))
print("Test Accuracy Tanh: {}".format(model.score(testX, testY)))
print("Validation Accuracy Tanh: {}".format(model.score(validX, validY)))
✓ 4.3s
Train Accuracy Tanh: 11.322448979591837
Test Accuracy Tanh: 11.25
Validation Accuracy Tanh: 10.771428571428572
```

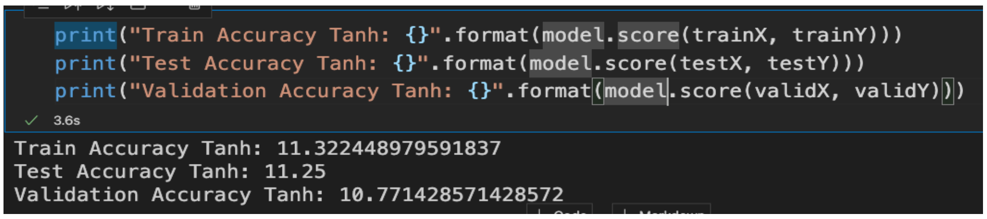
- Accuracy vs epoch plot being:



```
accVsEpochPlot(trainAccs, validAccs, 'tanh_0.01')
✓ 0.4s
```

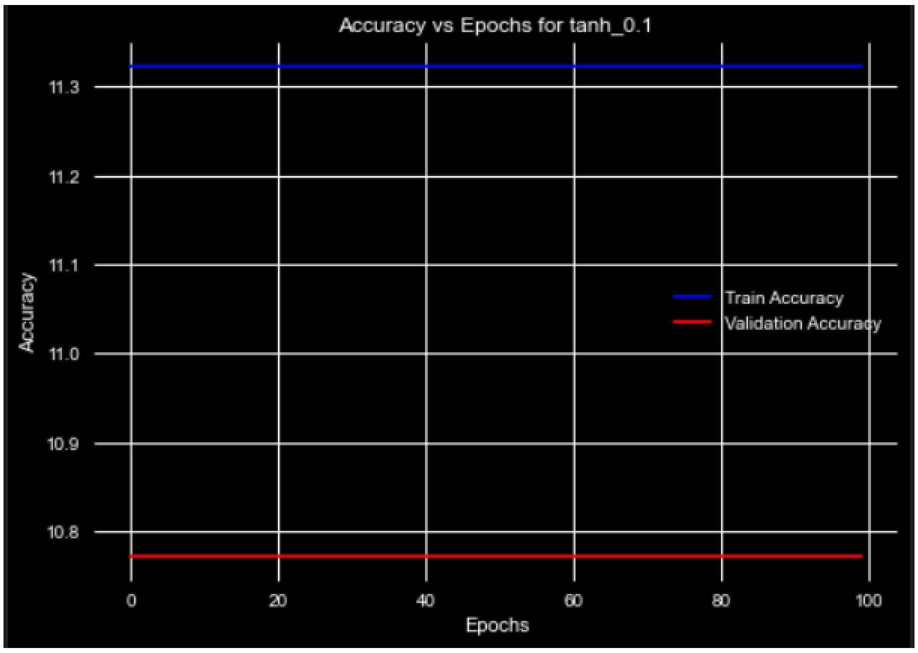
c learning_rate = 0.1, epochs = 100

- The Following accuracies were obtained:



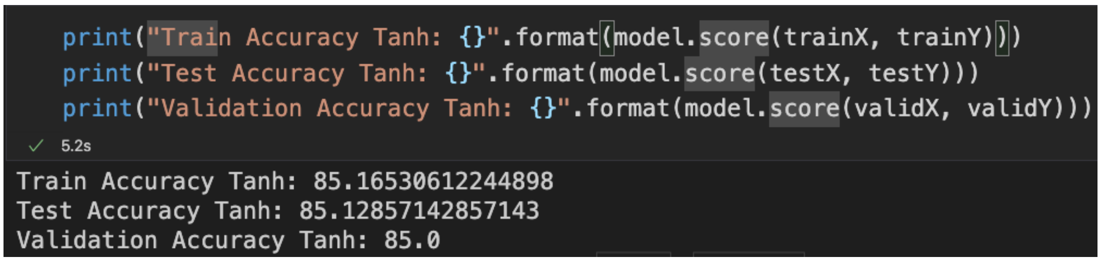
```
print("Train Accuracy Tanh: {}".format(model.score(trainX, trainY)))
print("Test Accuracy Tanh: {}".format(model.score(testX, testY)))
print("Validation Accuracy Tanh: {}".format(model.score(validX, validY)))
✓ 3.6s
Train Accuracy Tanh: 11.322448979591837
Test Accuracy Tanh: 11.25
Validation Accuracy Tanh: 10.771428571428572
```

- Accuracy vs epoch plot being:



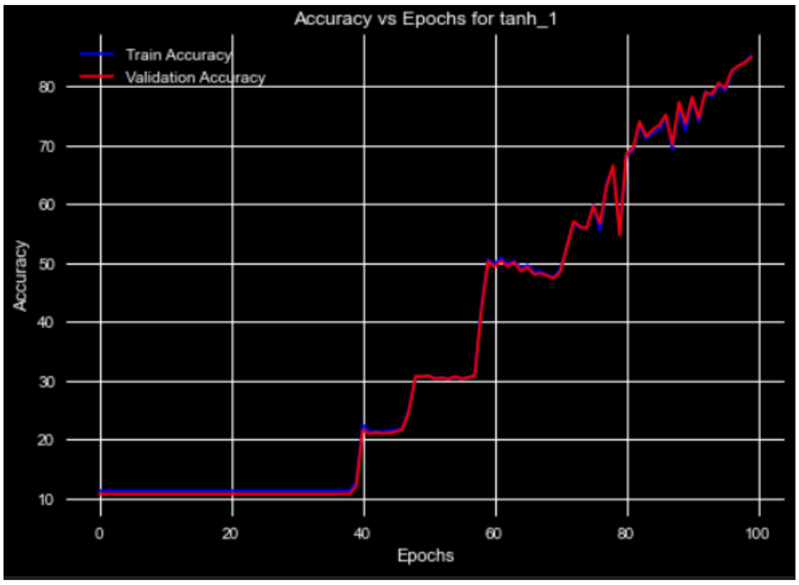
d learning_rate = 1, epochs = 100

- The Following accuracies were obtained:



```
print("Train Accuracy Tanh: {}".format(model.score(trainX, trainY)))
print("Test Accuracy Tanh: {}".format(model.score(testX, testY)))
print("Validation Accuracy Tanh: {}".format(model.score(validX, validY)))
✓ 5.2s
Train Accuracy Tanh: 85.16530612244898
Test Accuracy Tanh: 85.12857142857143
Validation Accuracy Tanh: 85.0
```

- Accuracy vs epoch plot being:



- As it is clearly evident, the model with learning rate 1 on tanh performs the best. Had i still let the model go on training for some more epochs, the accuracy would have improved further. As a matter of fact, for about 150-200 epochs, the model gives testing accuracy of as high as 95-97%.