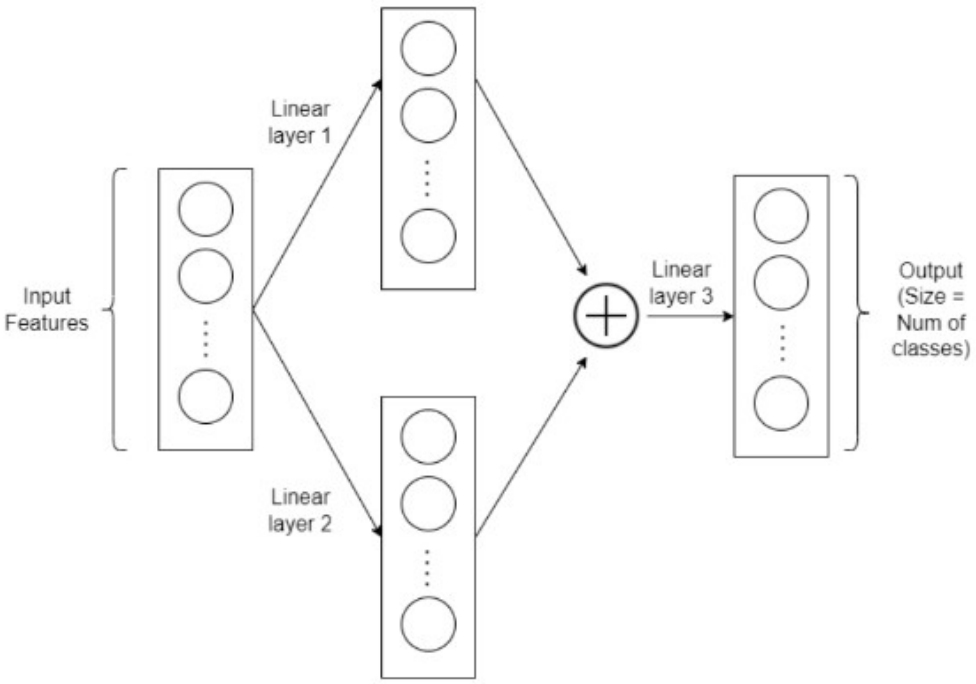
Lab 7
Abu Shahid
B20CS003

Question 1

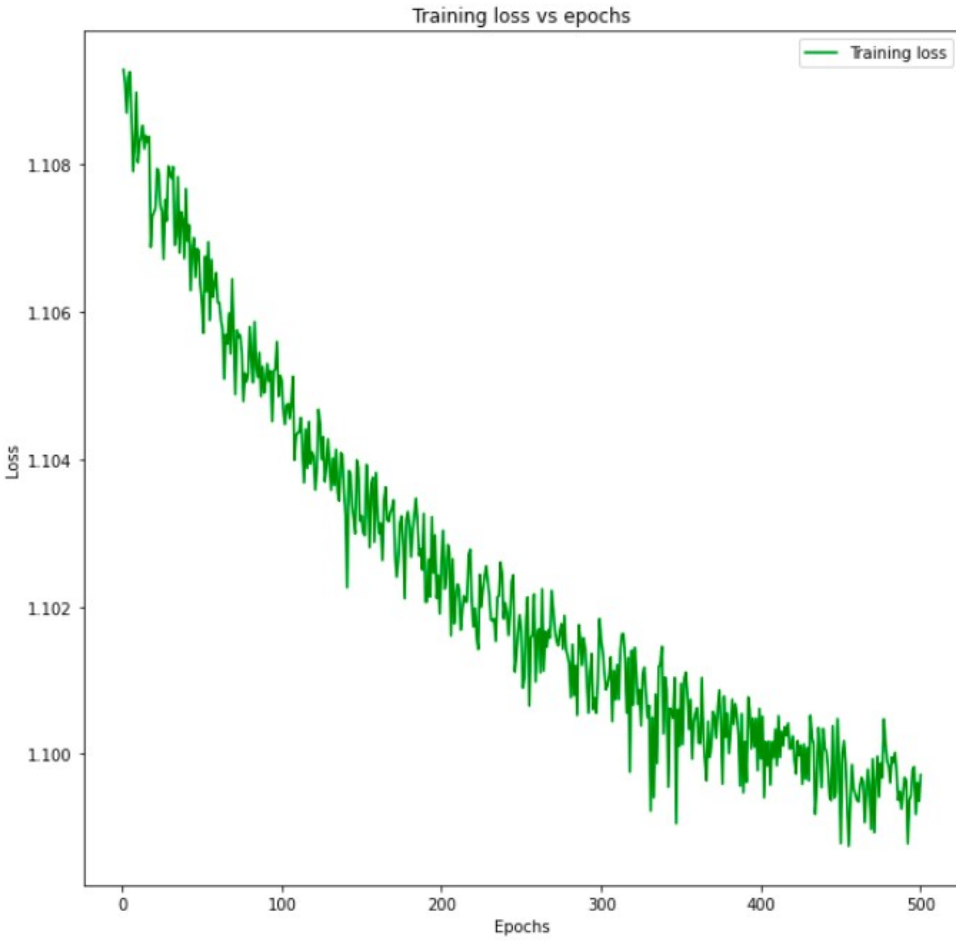
- We first import all the necessary frameworks and libraries.
- Data is loaded. Data cleaning is performed and random errors (object instance in numerical features and vice-versa are taken care of)
- Next label encoding of object columns is performed and data is standardized using MinMaxScaler()
- Further since the output classes are from 0-29, we binerize it in 5 bins uniformly and with this we convene our data pre-processing.
- Data is split in 80:20 train:test ratio.
- **Initializing our network.**



We implemented the above architecture with following specifics.

```
batch_size = 32
num_epochs = 500
size_hidden_1 = 500
size_hidden_2 = 500
learning_rate = 0.05
num_classes = 5
```

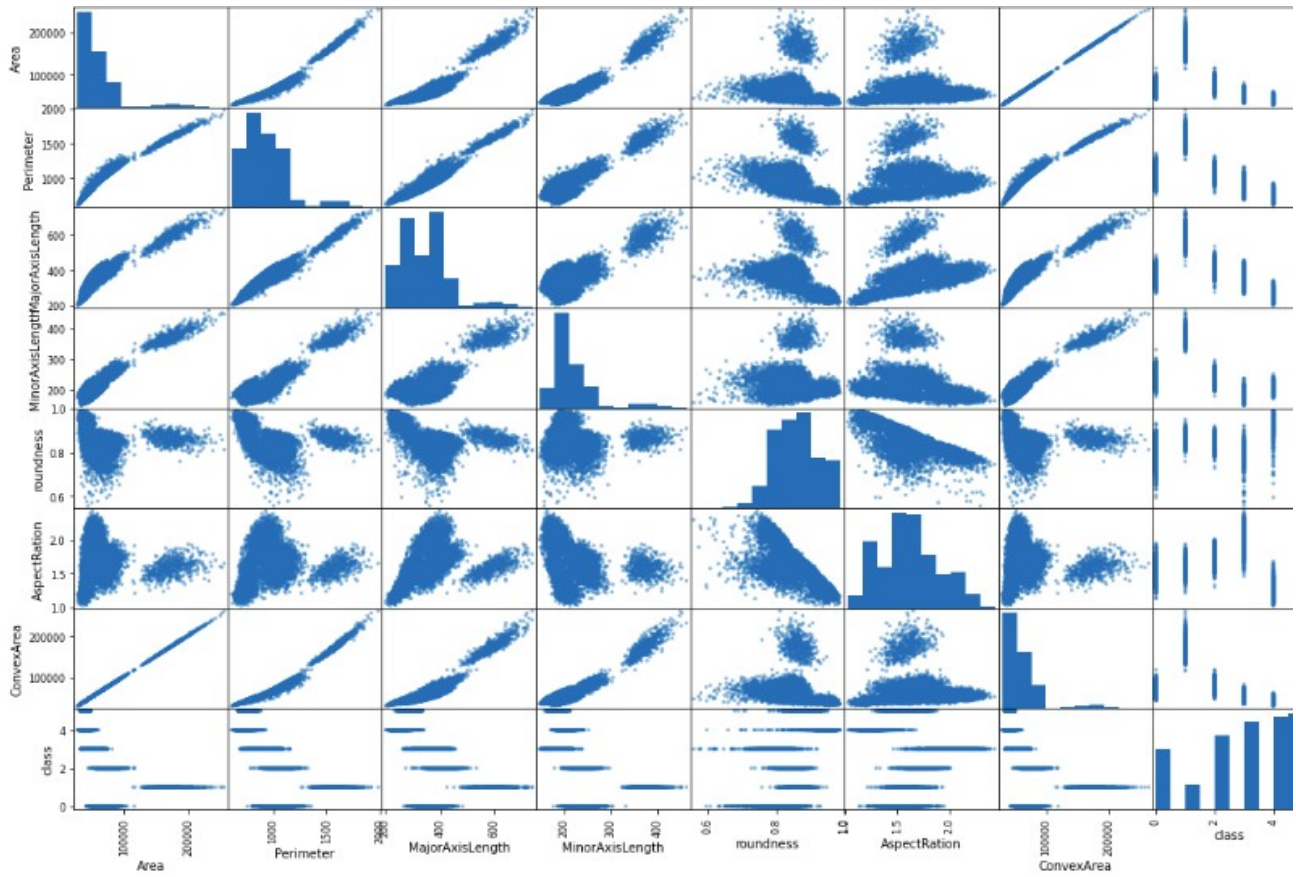
- Activation for hidden layer was kept to be ‘Tanh’ while for output layer, sigmoid was used.
- Calculation CrossEntropy for loss, model was put to training.



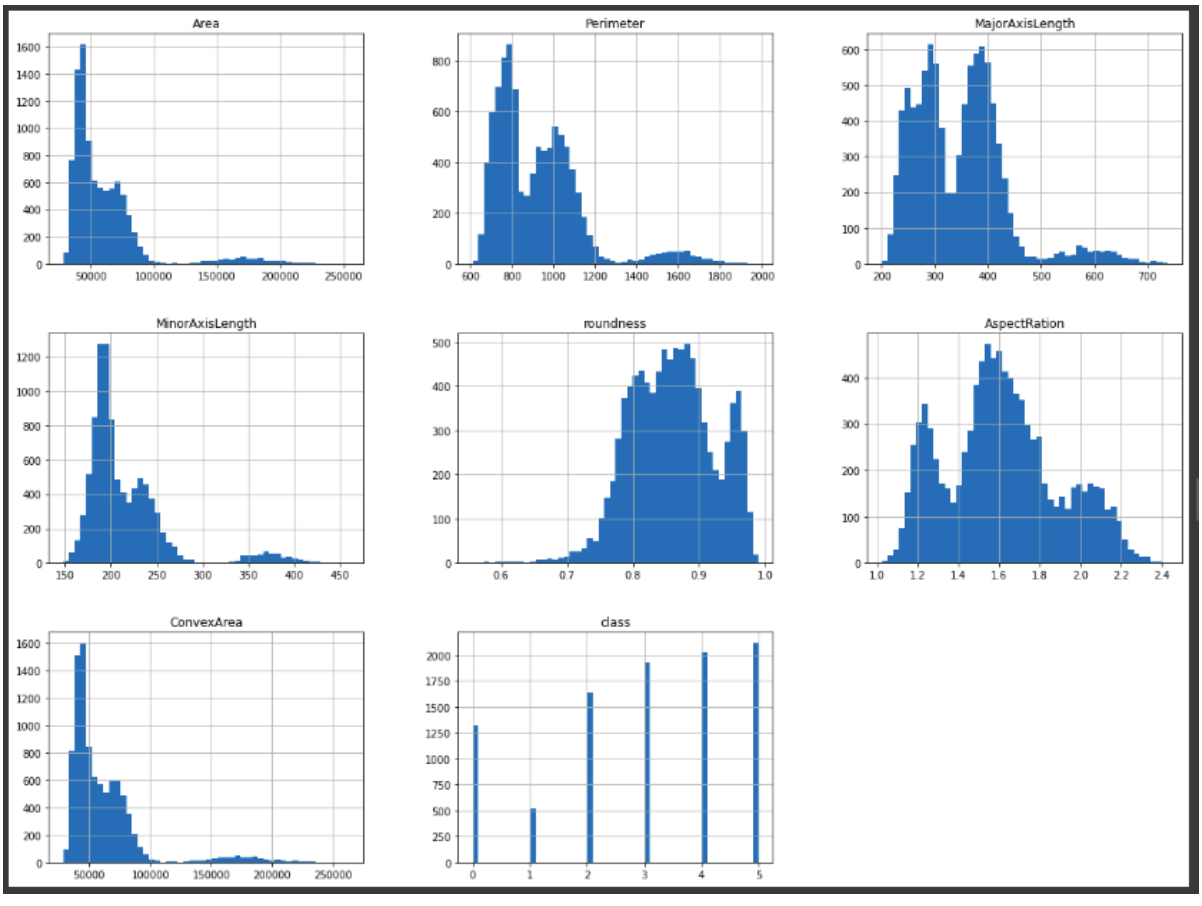
Question 2:

Subpart a: Preprocess & visualize the data. Create train, val, and test splits

- Necessary libraries were imported, data was loaded. Missing feature values were taken care of by dropping corresponding rows.
- As class category was in String, there were ordinally Encoded.
- Further data was then standardized to achieve better results with our network but before that, visualization was performed to get better insight of the data.
    - **Visualization and Drwaing Insights**
- For starters, scatter_matrix was plotted (PFA next page) and as we can see from the last row, there does not seem to exist a simple linear relationship between our ‘class’ and the features.



- Further, histogram with bin size=50 was plotted.

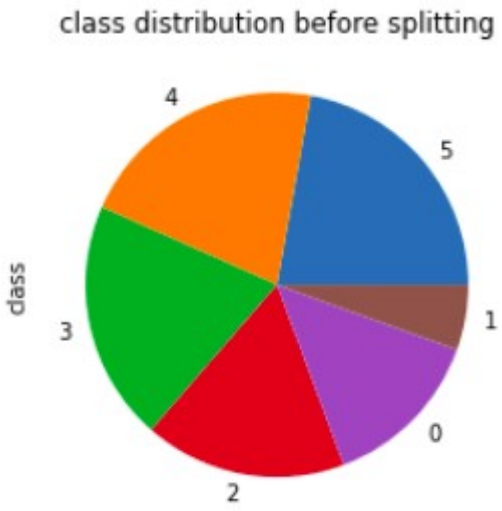


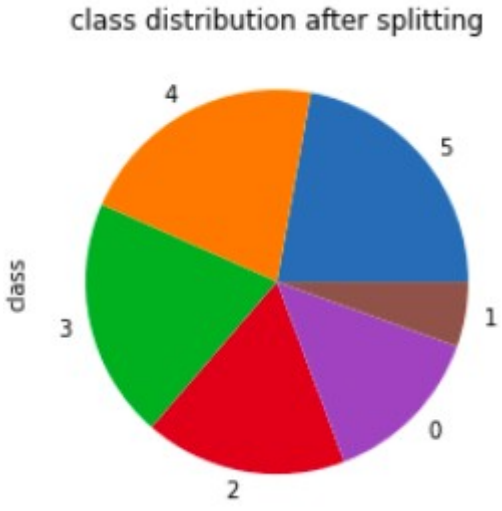
- From the plot above, we can see that data feature distribution is rather skewed.

```
skewness in data features
Area                2.753361
Perimeter           1.596199
MajorAxisLength     1.170257
MinorAxisLength     2.272045
roundness          -0.171924
AspectRation        0.269390
ConvexArea          2.741223
class              -0.450031
dtype: float64
```

```
correlation bw different features and class
class              1.000000
roundness          0.536573
AspectRation      -0.199286
Area              -0.592376
ConvexArea        -0.595030
MinorAxisLength   -0.604100
MajorAxisLength   -0.637025
Perimeter         -0.706528
Name: class, dtype: float64
```

- 'Area', 'MinorAxisLength' and 'ConvexArea' are the most skewed features and yet their correlation with 'class' is almost similar with slight variation. For our case, we will stratify-Split based on 'MinorAxisLength'.
- Same was attempted but as the least populated class in MinorAxisLength, Area and ConvexArea has only 1 member, which is too few, we attempt this with 'class'. This procedure reduces sampling bias.

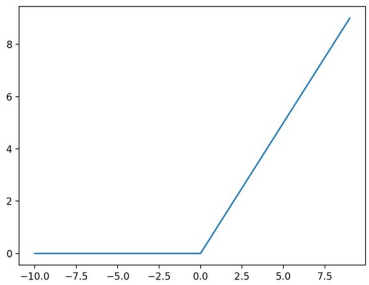


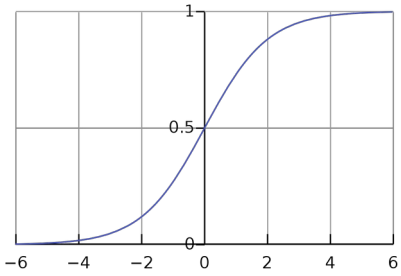


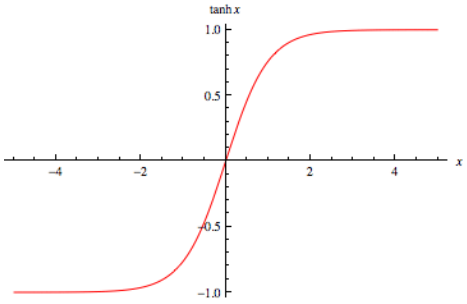
Subpart B: Implementing MLP from scratch

Activation functions

- For our purpose, we define 3 different activation functions.
  - ReLU
  - Sigmoid
  - Tanh







ReLU **Sigmoid** **Tanh**

Forward Propogation

- Our forward propogation task was done in three steps
  - **init(n_inputs, hiddens, n_outputs, activ)**
    1. n_inputs: number of inputs/neurons in our input layer
    2. hiddens: is a list giving us the specification of the hiddens layers in our model. Hiddens= [5, 3] will mean our model has 2 hidden layers with 5 and 3 neurons in each respectively.
    3. n_outputs: number of outputs we want from our network (here it will be equal to number of classes)
  - init() function will return our network as a list of lists.
    - Each element in our list represents a layer is a list of neurons.
    - Neurons are stored in our network as python dictionaries and contain
      - weights of the connections with neurons of previous layer.
      - Last element in layer[] is a dictionary containing the activation specific of that layer.
      - Network= init(2, [1, 2], 2, ['relu','relu','relu','relu']) will make our network as:

```
[{'weights': [0.8092755392960491, 0.11414527663717333, 0.12513648794201415]}, {'activation': 'relu'}]
[{'weights': [0.7333976016748732, 0.6609927430522639]}, {'weights': [0.29311427571280213, 0.480656626087271]}, {'activation': 'relu'}]
[{'weights': [0.773094884873797, 0.17988810248774156, 0.6163789831513413]}, {'weights': [0.25187480421280506, 0.15181969215562108, 0.29064651696348487]}, {'activation': 'relu'}]
```

  - **fire(weights, inputs)**
    1. weights: it is a list of weights with the last element being the bias
    2. inputs: as the name suggest it is the input
       - fire() returns the weighted addition of inputs which is then fed to out activation function

- **forward_propogate(network, data):**
  1. network: our initialized neural net
  2. data is nothing but our input in numpy.ndarray
  - for each layer, inputs are taken and neurons are **fired,** creating input for the corresponding layers.
  - The input to the last layer is finally returned as our output.

```
network= init( 2, [1, 3, 2], 3,['relu','relu','relu','relu','relu'])
output= forward_propagate( network, [2, 4])
print(output)
```

```
[6.42239367487365, 3.7223613716012514, 5.410868649940509]
```

Backward propogation

This is done in two steps:

- transfer derivative
- error backpropogation

1. **back_propogate(network, expected)** #expected is our expected/actual output
   - this function basically calculates the errors for each neuron.
     - This procedure is rather simple for the outermost layer
     - The error signal for a neuron in the hidden layer is calculated as the weighted error of each neuron in the output layer.
   - After having calculated our errors for each neuron in our network, we calculate the gradient (neuron[‘delta’]) for each of the neuron as:
     - **neuron[‘delta’]= error of neuron* f’(output of neuron),** where f’ is the derivation of the activation function

Training

1. **update(network, feed, l)** #feed is basically a row from our dataset, l is learning rate
   - We update weight as, weight= weight – learning_rate* error* input

2. **train(network, data, l, epochs, outputs)**

   # epochs in the number of epochs we want, data is our entire data set

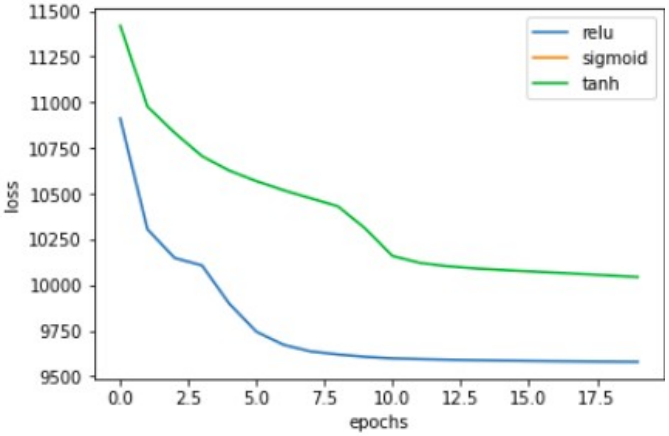
   - The network is updated via stochastic gradient descent, as previously stated.
   - This entails first looping for a given number of epochs and then updating the network for each row in the training dataset inside each epoch.
   - Because each training pattern is updated, this sort of learning is known as online learning.
   - The predicted number of output values is utilised to convert class values in training data into a single hot encoding.
   - To match the output values, this is a binary vector with one column for each class value. This is necessary in order to compute the error for the output layer.
   - The total squared error between the anticipated and network outputs is gathered and shown at each epoch. This is useful for keeping track of how much the network learns and improves at each epoch.
   - Accuracy and loss at the end of each epoch is then stored and returned.

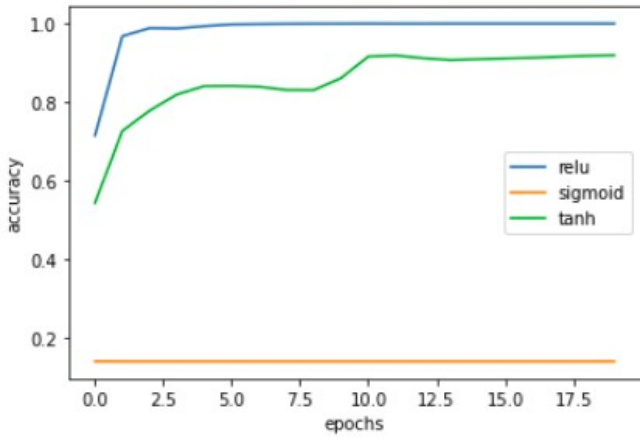
- Finally a function is written to calculate the accuracy of our model.
- Putting all this together,a model was trained with one hidden layer having 5 neurons and ‘ReLU’ activationbelow are its results.

```
>>>>epoch:  0 , learning rate:  0.01 , accuracy:  0.7395556797540869
>>>>epoch:  1 , learning rate:  0.01 , accuracy:  0.9027525499510969
>>>>epoch:  2 , learning rate:  0.01 , accuracy:  0.9707978203157748
>>>>epoch:  3 , learning rate:  0.01 , accuracy:  0.9886824088305156
>>>>epoch:  4 , learning rate:  0.01 , accuracy:  0.9925946625681151
>>>>epoch:  5 , learning rate:  0.01 , accuracy:  0.9925946625681151
>>>>epoch:  6 , learning rate:  0.01 , accuracy:  0.9921754925248009
>>>>epoch:  7 , learning rate:  0.01 , accuracy:  0.9920357691770295
>>>>epoch:  8 , learning rate:  0.01 , accuracy:  0.9925946625681151
>>>>epoch:  9 , learning rate:  0.01 , accuracy:  0.9930138326114294
>>>>epoch:  10 , learning rate:  0.01 , accuracy:  0.9932932793069722
>>>>epoch:  11 , learning rate:  0.01 , accuracy:  0.9934330026547437
>>>>epoch:  12 , learning rate:  0.01 , accuracy:  0.9931535559592007
>>>>epoch:  13 , learning rate:  0.01 , accuracy:  0.9932932793069722
>>>>epoch:  14 , learning rate:  0.01 , accuracy:  0.9934330026547437
>>>>epoch:  15 , learning rate:  0.01 , accuracy:  0.9937124493502865
>>>>epoch:  16 , learning rate:  0.01 , accuracy:  0.9938521726980578
>>>>epoch:  17 , learning rate:  0.01 , accuracy:  0.9938521726980578
>>>>epoch:  18 , learning rate:  0.01 , accuracy:  0.9938521726980578
>>>>epoch:  19 , learning rate:  0.01 , accuracy:  0.9938521726980578
```

Subpart C: Experimenting with activation functions;

- We trained our model using different activation functions and the results can be summarise as follows:

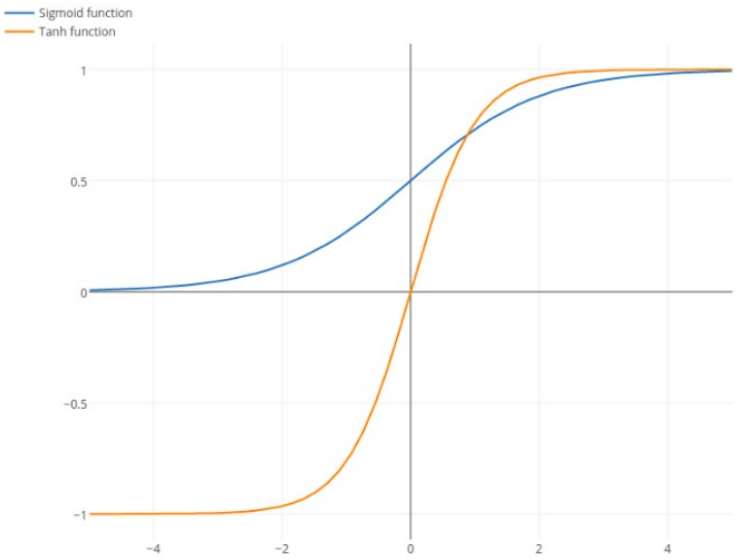




***PS: sigmoid and tanh loss curves are overlapping**

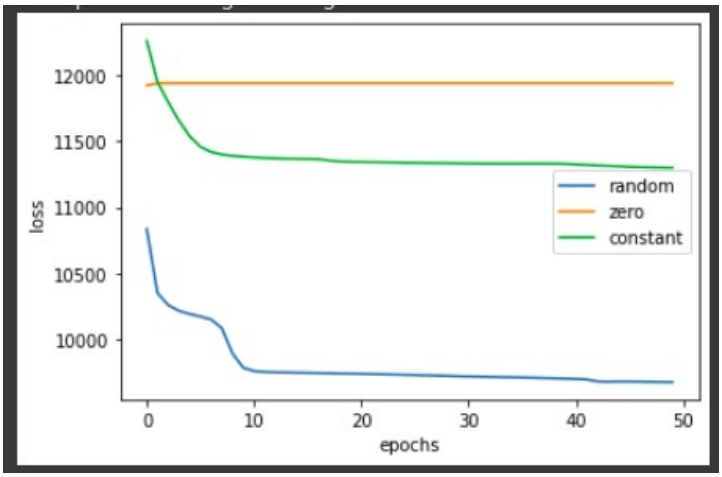
```
testing accuracies are:

reLU : 0.9974853310980721
Sigmoid : 0.13830678960603 52
Tanh : 0.9174350377200335
```

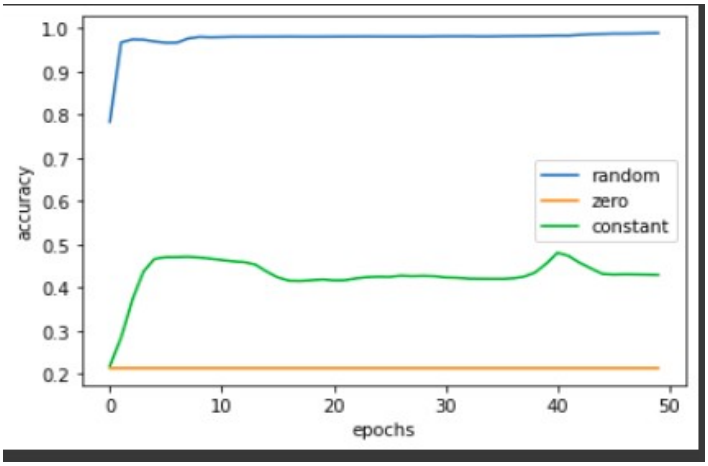
- Not only ReLU is more biologically inspired but also it is easier and fater to calculate.
- Additional major benifit of ReLU is its sparsity and a reduced likelihood to vanishing gradient which seems to be a major concern in other activation functions.
- Sparsity springs up when x<0. If we have more of these in a laye, the more sparse is that corresponding layer. However sigmoid and tanh also give some non-zero value leading to dense representations.
- Sigmoid though may not blow up the activation, can however result in vanishing gradient. Apparently, ReLU can blow the activation us as there is no way to constrain output of a neuron.
- As to the reason why tanh is perform so much better than sigmoid is due to the fact that derivatives of tanh are larger than derivatives of sigmoid. This can help in larger wieight updates and model can converge faster. Given enough epochs, tanh and sigmoid should theoretically give similar results.



Subpart D: Weight Initializations

- We next trained our model with different weight initialization techniques and it all can be summariezed as:



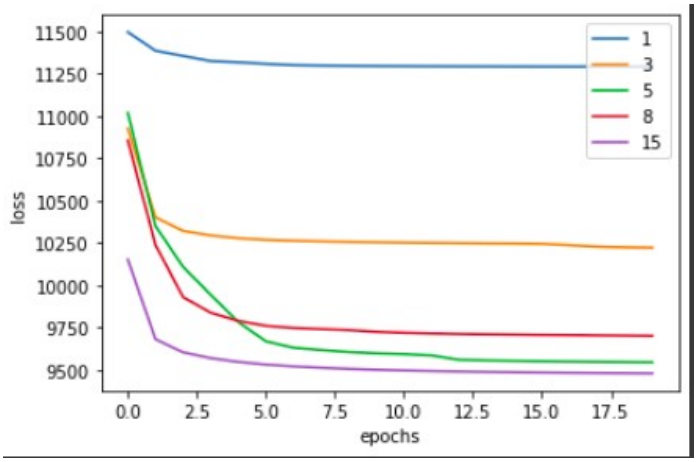


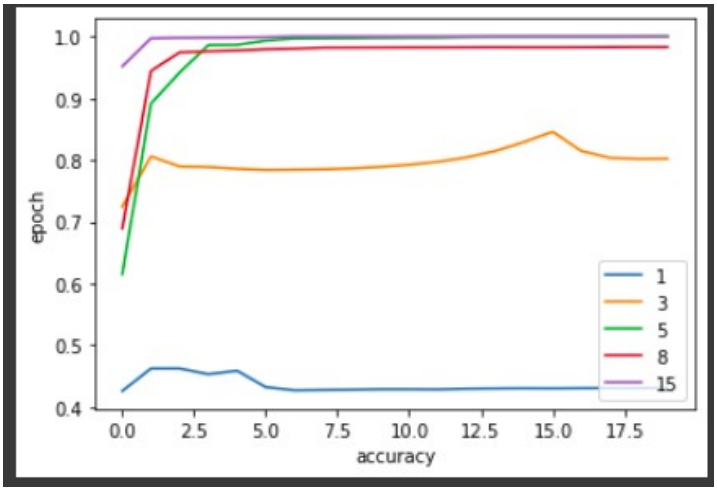

```
testing accuracies are:

random : 0.9874266554903605
zero : 0.22170997485331098
constant : 0.4237217099748533
```

- As we can see, that while constant initializations seem to give better result it is no where close to random initialization.

- If all the weights are initialized to zeros, the derivatives will remain same for every neuron in our network. As a result, neurons will learn same features in each iterations. This problem is known as network failing to break symmetry. And not only zero, any constant initialization will produce a poor result.
- The weights attached to the same neuron, continue to remain the same throughout the training.

Subpart E: Neurons in a Layer






```
with  1  test accuracy is : 0.4237217099748533
with  3  test accuracy is : 0.8021793797150042
with  5  test accuracy is : 0.998742665549036
with  8  test accuracy is : 0.9832355406538139
with  15  test accuracy is : 1.0
```

- In we increase the number of neurons in a hiddedn layer, the complexity of our model increases with increase in complexity. And efficiency may increase.
- The model gets more adaptive, so it can learn smaller details. But this means not necessary, that your classifier is then better on the 'next dataset'. The model gets more affected to over-fitting and so the generalization of you classification model can also decrease, e.g. your classifier will work worse on the next dataset.
- Although in our case so far, increasing neuron count has resulted is greater and better results as it should, its safe to assume that after a certain count the model will start overfitting and we can expected testing accuracy to drop.

Subpart F: Saving and Loading our model

- This is done as follows: