Lab 9 Report

Nakkina Vinay (B21AI023)

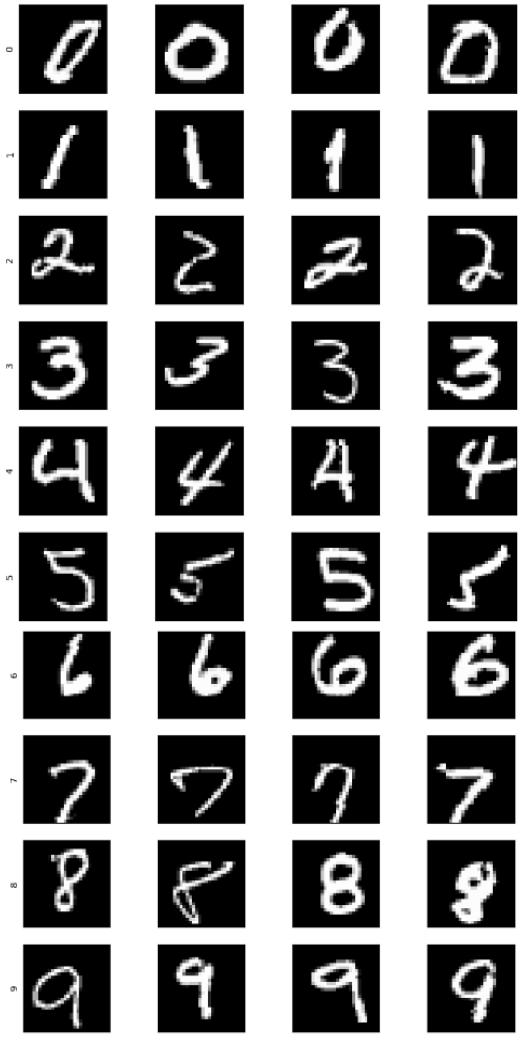
Question 1

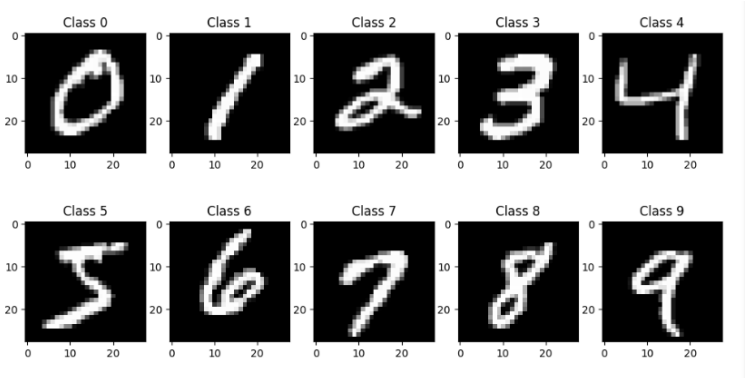
Subpart 1:

- Downloaded the MNIST dataset using torch-vision.
- Applied Augmentations to images of the training dataset: RandomRotation(5 degrees), RandomCrop(size=28, padding = 2), ToTensor and Normalize.
- Defined transformations for the testing dataset and validation datasets: ToTensor and Normalize
- Splitting the dataset into train, test, and validation sets

Subpart 2:

- Plotting a few Images from each class. Created a data loader for training and testing datasets





Subpart 3:

- Implemented 3-Layer MLP using PyTorch all using Linear layers
- The input size is 784, hidden size is 256 and output size is 10
- Printed the number of trainable parameters of the model.

```
Number of trainable parameters: 269322
```

Subpart 4:

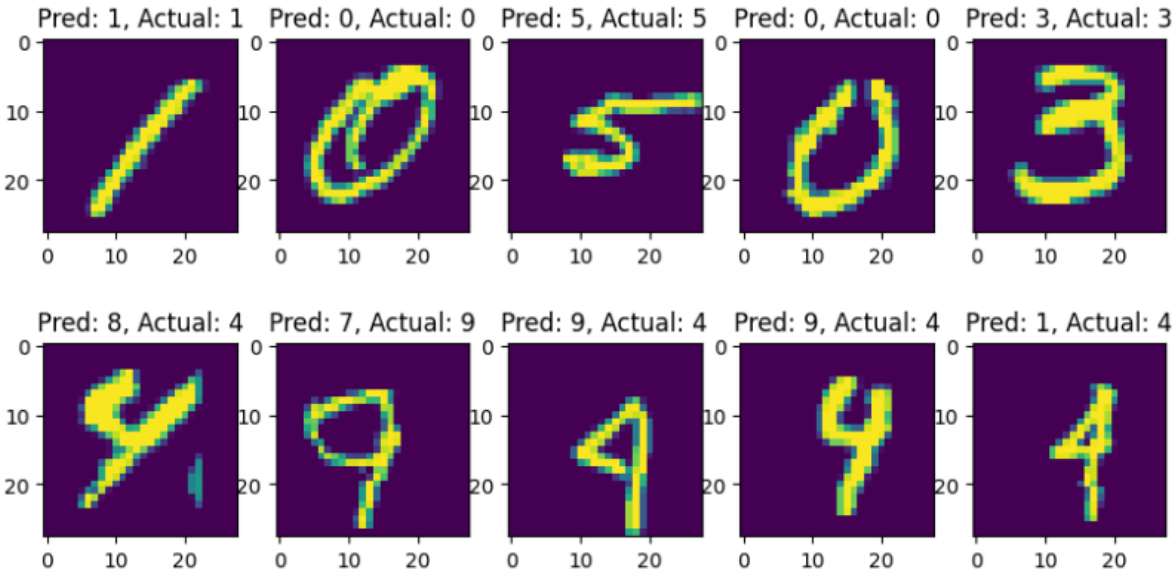
- First defining the training parameters
    - Learning_rate = 0.01
    - Num_epochs = 5
    - Input size = 784
    - Hidden size = 128
    - Output size = 10
- Initialising the model and optimizer and training the model for 5 epochs using Adam as the optimizer and CrossEntropyLoss as the Loss Function.
- Evaluating the model on the validation set after each epoch and saved the best model
- Calculated and printed the accuracy and loss of the model on training and validation data at the end of each epoch.

```
100%|██████████| 750/750 [00:26<00:00, 28.74it/s]
100%|██████████| 188/188 [00:05<00:00, 33.41it/s]
Epoch [1/5], Train Loss: 0.4769, Train Acc: 0.8506, Val Loss: 0.2454, Val Acc: 0.9267

100%|██████████| 750/750 [00:25<00:00, 28.93it/s]
100%|██████████| 188/188 [00:06<00:00, 29.49it/s]
Epoch [2/5], Train Loss: 0.2020, Train Acc: 0.9379, Val Loss: 0.1727, Val Acc: 0.9474

100%|██████████| 750/750 [00:27<00:00, 27.55it/s]
100%|██████████| 188/188 [00:05<00:00, 33.78it/s]
Epoch [3/5], Train Loss: 0.1594, Train Acc: 0.9510, Val Loss: 0.1728, Val Acc: 0.9467

100%|██████████| 750/750 [00:26<00:00, 28.70it/s]
100%|██████████| 188/188 [00:06<00:00, 30.30it/s]
Epoch [4/5], Train Loss: 0.1344, Train Acc: 0.9584, Val Loss: 0.1339, Val Acc: 0.9609

100%|██████████| 750/750 [00:25<00:00, 28.99it/s]
100%|██████████| 188/188 [00:05<00:00, 31.92it/s]Epoch [5/5], Train Loss: 0.1259, Train Acc: 0.9611, Val Loss: 0.1366, Val Acc: 0.9605
```

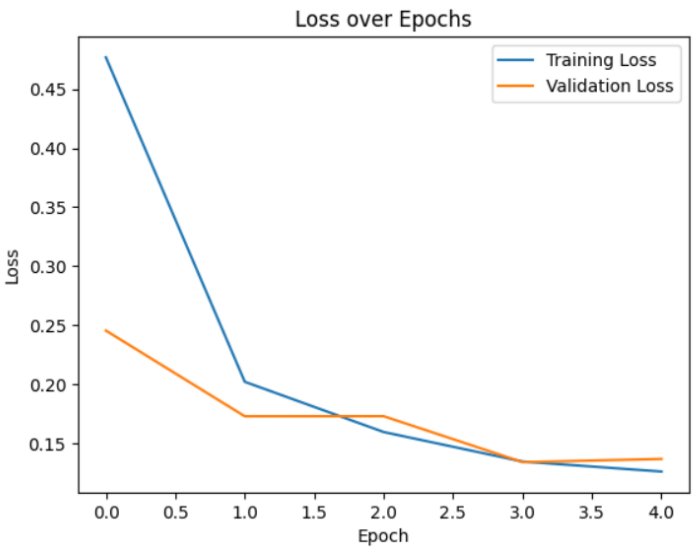
```
Best Validation Accuracy: 0.9609
Final Training Accuracy: 0.9611
Final Validation Accuracy: 0.9605
```

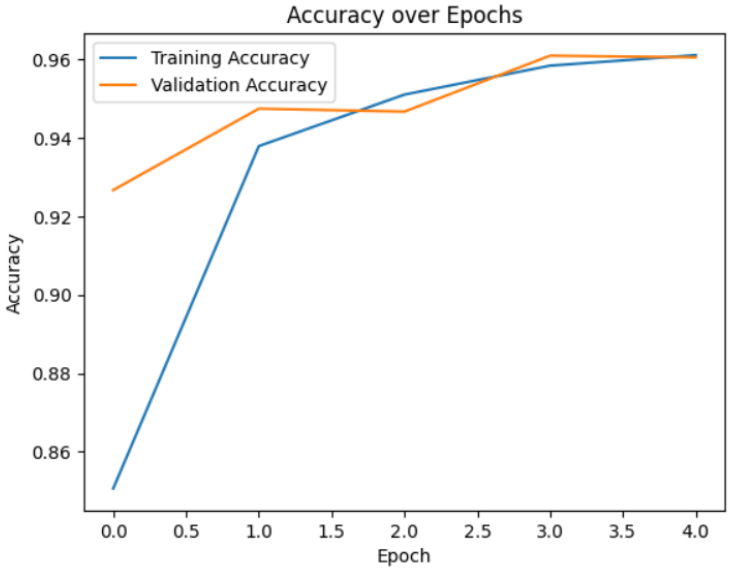
Subpart 5:

- Setting the model to evaluation mode and initializing the variables to keep track of correct and incorrect predictions
- Looping over the validation dataset and appending the correct and incorrect predictions to the lists
- Visualising the correct and incorrect predictions



- Plotting the graphs for loss over the epochs and the accuracy over the epochs





Question 2

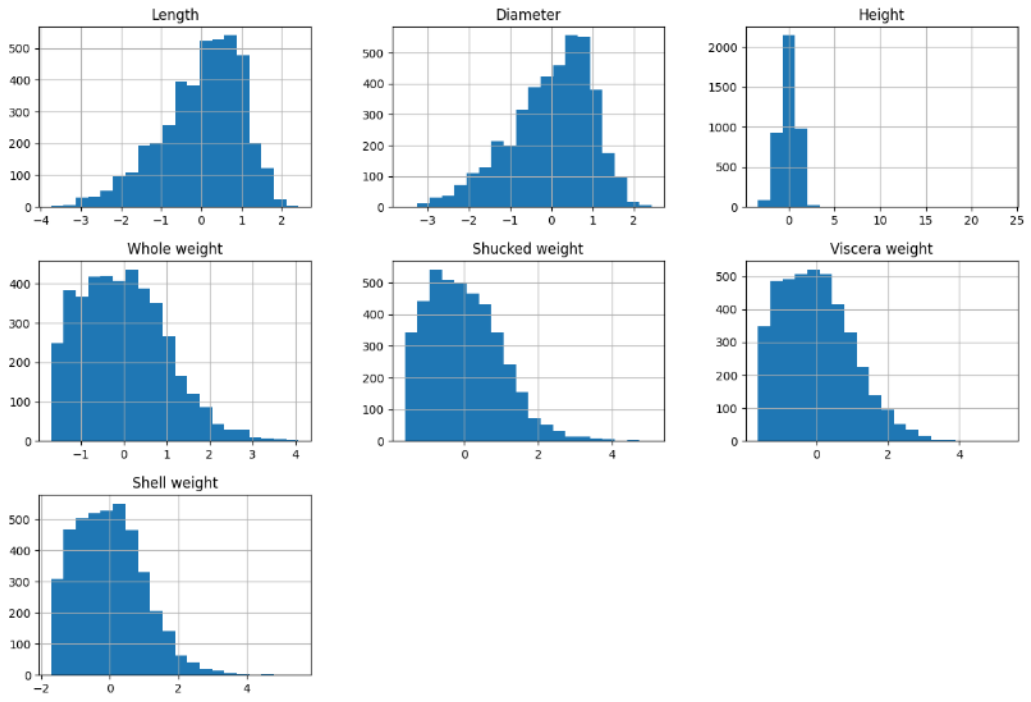
Subpart 1:

- Importing the dataset, giving the column names, and printing it

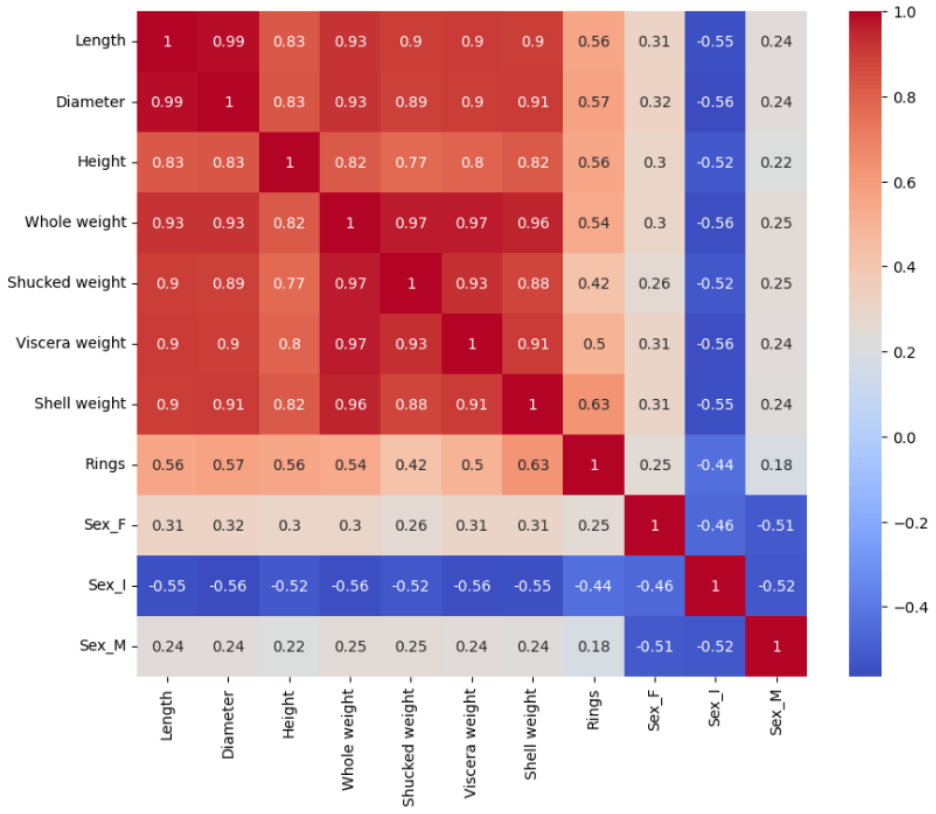
	Sex	Length	Diameter	Height	Whole weight	Shucked weight	Viscera weight	Shell weight	Rings
0	M	0.455	0.365	0.095	0.5140	0.2245	0.1010	0.1500	15
1	M	0.350	0.265	0.090	0.2255	0.0995	0.0485	0.0700	7
2	F	0.530	0.420	0.135	0.6770	0.2565	0.1415	0.2100	9
3	M	0.440	0.365	0.125	0.5160	0.2155	0.1140	0.1550	10
4	I	0.330	0.255	0.080	0.2050	0.0895	0.0395	0.0550	7
...	...	...	...	...	...	...	...	...	...
4172	F	0.565	0.450	0.165	0.8870	0.3700	0.2390	0.2490	11
4173	M	0.590	0.440	0.135	0.9660	0.4390	0.2145	0.2605	10
4174	M	0.600	0.475	0.205	1.1760	0.5255	0.2875	0.3080	9
4175	F	0.625	0.485	0.150	1.0945	0.5310	0.2610	0.2960	10
4176	M	0.710	0.555	0.195	1.9485	0.9455	0.3765	0.4950	12

4177 rows × 9 columns

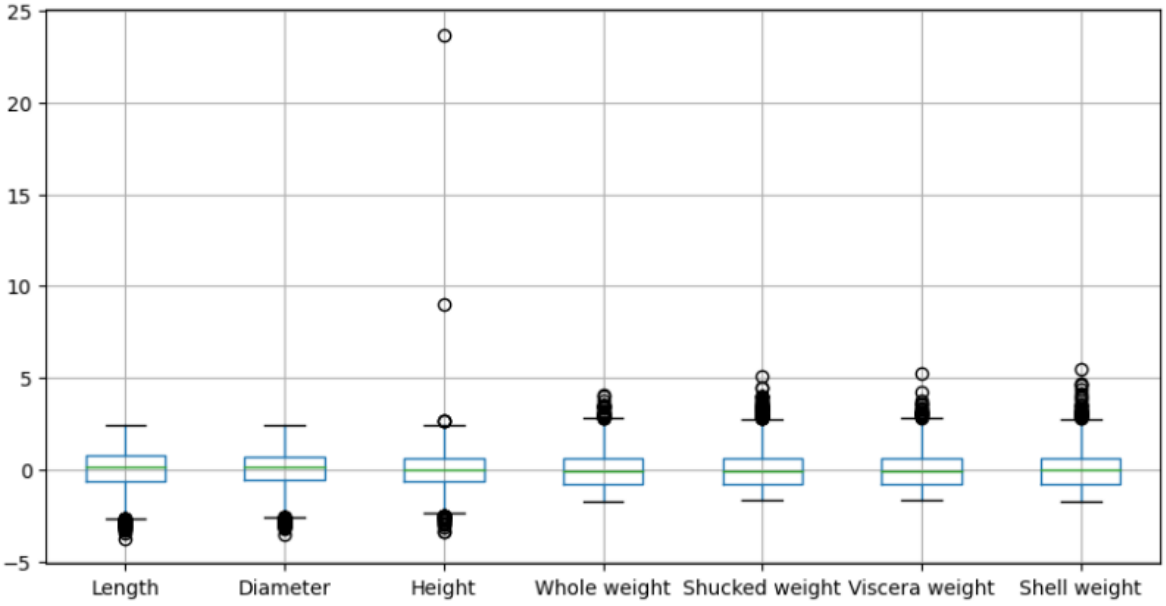
- Converting the categorical variable column “Sex” to numerical using one-hot encoding
- Normalising the numerical variables of columns 'Length', 'Diameter', 'Height', 'Whole weight', 'Shucked weight', 'Viscera weight', 'Shell weight' using StandardScaler
- Visualising the distribution of numerical variables using histograms



- Visualising the correlation between features using a heatmap



- Visualising the boxplot of the features

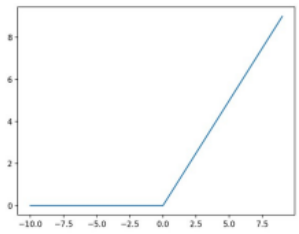


- Splitting the data into X and y and performing sampling on them to increase the occurrence of the least occurred classes and split them into train, test, and validation sets using the stratified splitting

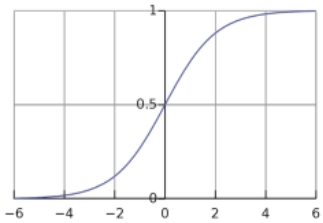
Subpart 2: Implementing MLP from Scratch

Activation functions

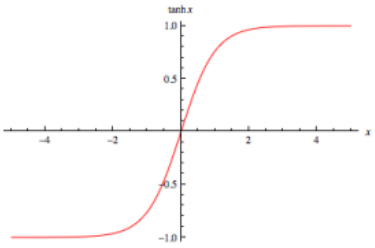
- For our purpose, we defined three activation functions
  - ReLU
  - Sigmoid
  - Tanh



ReLU



Sigmoid



Tanh

Forward Propagation

- Forward propagation was done in three steps
  - **Init(n_inputs, hiddens, n_outputs, active)**
    - n_inputs: number of inputs/neurons in our input layer
    - 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.
    - 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 the 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': [0.9989000154992786, 0.6726347472246018, 0.14056088838086367]}, {'activation': 'relu'}]
[{'weights': [0.28249087013687946, 0.5149752904584157]}, {'weights': [0.18010363747197666, 0.5030694252163678]}, {'activation': 'relu'}]
[{'weights': [0.4946799130002577, 0.8533615132681343, 0.26998885895850366]}, {'weights': [0.022351889175037942, 0.18235304521965845, 0.5237968755356037]}, {'activation': 'relu'}]
```

- **fire(weights, inputs)**
  - weights: it is a list of weights with the last element being the bias
  - inputs: as the name suggests it is the input
    - fire() returns the weighted addition of inputs which is then fed to out activation function
- **forward_propagation(network, data)**
  - network: our initialized neural net
  - 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)
```

```
[2.1435383220761244, 4.927438637994847, 1.7883117629245724]
```

Backward Propagation

- This is done in two steps:
  - transfer derivative
  - error backpropagagtion

- **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

- **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

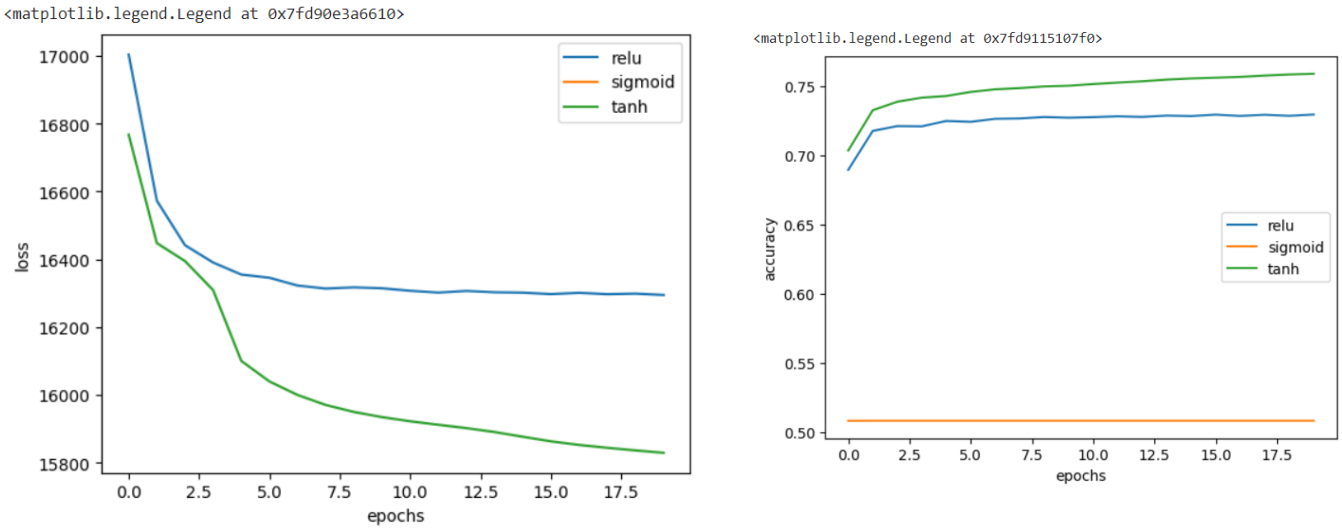
- **train(network, data, l, epochs, output) #**epochs is the number of epochs we want and data is our entire dataset
    - The network is updated via stochastic gradient descent, as previously stated.
    - This entails first looping for a given number of epochs and then updating the network for each 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 utilized 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 are 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' activation below are its results

```
>>>epoch:  0 , learning rate:  0.01 , accuracy:  0.6955244513564887
>>>epoch:  1 , learning rate:  0.01 , accuracy:  0.7201486089510973
>>>epoch:  2 , learning rate:  0.01 , accuracy:  0.7220494211162951
>>>epoch:  3 , learning rate:  0.01 , accuracy:  0.7225678244340764
>>>epoch:  4 , learning rate:  0.01 , accuracy:  0.7234318299637118
>>>epoch:  5 , learning rate:  0.01 , accuracy:  0.7230862277518576
>>>epoch:  6 , learning rate:  0.01 , accuracy:  0.7229998271988941
>>>epoch:  7 , learning rate:  0.01 , accuracy:  0.7238638327285295
>>>epoch:  8 , learning rate:  0.01 , accuracy:  0.7243822360463107
>>>epoch:  9 , learning rate:  0.01 , accuracy:  0.7241230343874201
>>>epoch:  10 , learning rate:  0.01 , accuracy:  0.723950233281493
>>>epoch:  11 , learning rate:  0.01 , accuracy:  0.7238638327285295
>>>epoch:  12 , learning rate:  0.01 , accuracy:  0.723950233281493
>>>epoch:  13 , learning rate:  0.01 , accuracy:  0.7242958354933472
>>>epoch:  14 , learning rate:  0.01 , accuracy:  0.7242094349403836
>>>epoch:  15 , learning rate:  0.01 , accuracy:  0.7240366338344565
>>>epoch:  16 , learning rate:  0.01 , accuracy:  0.7242094349403836
>>>epoch:  17 , learning rate:  0.01 , accuracy:  0.7240366338344565
>>>epoch:  18 , learning rate:  0.01 , accuracy:  0.7236910316226024
>>>epoch:  19 , learning rate:  0.01 , accuracy:  0.7236910316226024
```

Subpart 3:

- We trained our model using different activation functions and the results can be summarised by plotting the graphs



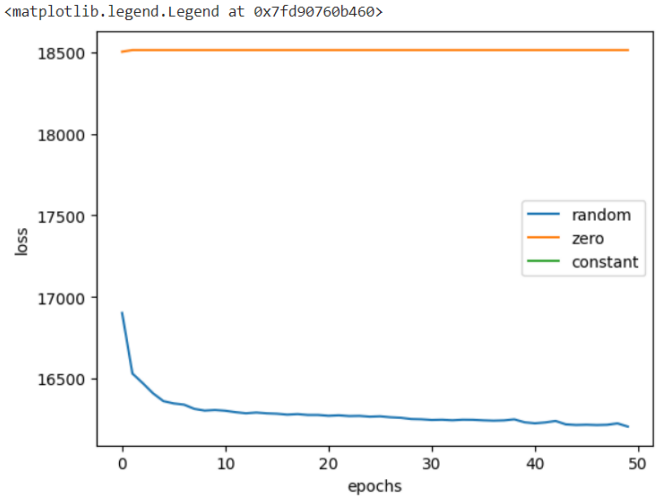
- Sigmoid and tanh loss curves are overlapping
- Accuracies are

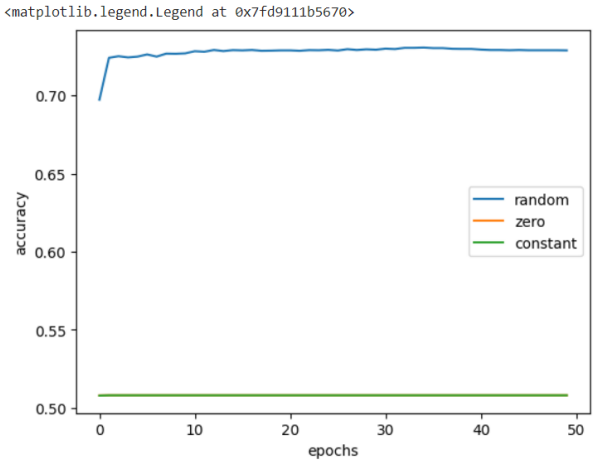
```
Testing accuracies are:

ReLU : 0.7416429126716766
Sigmoid : 0.5107540813682301
Tanh : 0.7452707955428868
```

- Not only ReLU is more biologically inspired but also it is easier and faster to calculate.
- Additional major benefit 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 layer, the more sparse is the 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 a vanishing gradient. Apparently, ReLU can blow the activation us as there is no way to constrain the 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 the sigmoid. This can help in larger weight updates and model can converge faster. Given enough epochs, tanh and sigmoid should theoretically give similar results.

Subpart 4:

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

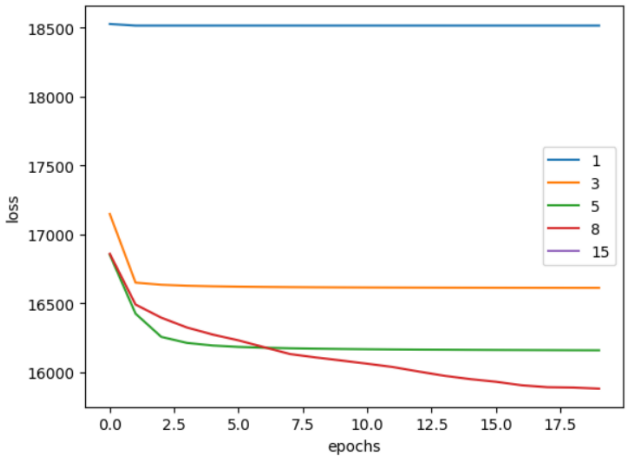


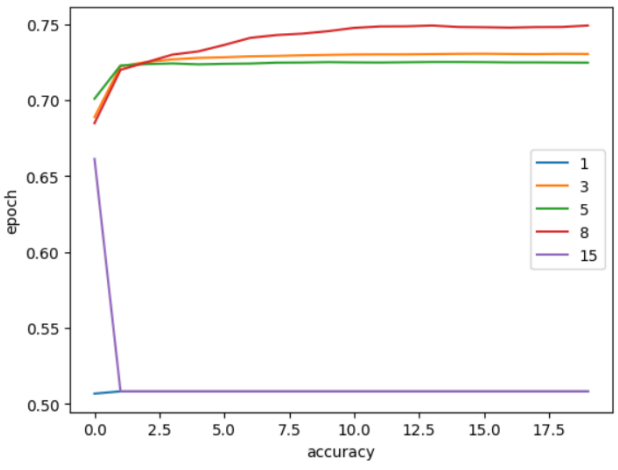



```
Testing accuracies are:

Random : 0.7395698367452708
Zero : 0.5107540813682301
Constant : 0.5107540813682301
```

- As we can see, while constant initializations seem to give better result it is nowhere 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 iteration. This problem is known as network failing to break symmetry. And not only zero, but any constant initialization will also produce a poor result.
- The weights attached to the same neuron, continue to remain the same throughout the training.

Subpart 5:





```
With  1  Test accuracy is : 0.5107540813682301
With  3  Test accuracy is : 0.7429385851256802
With  5  Test accuracy is : 0.7447525265612853
With  8  Test accuracy is : 0.7514900233221041
With  15  Test accuracy is : 0.5107540813682301
```

- 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.
- Saving and loading our model can be done as follows

Saving our model

```
import pickle
model = network
with open('model', 'wb') as f:
  pickle.dump(model, f)
```

Loading model

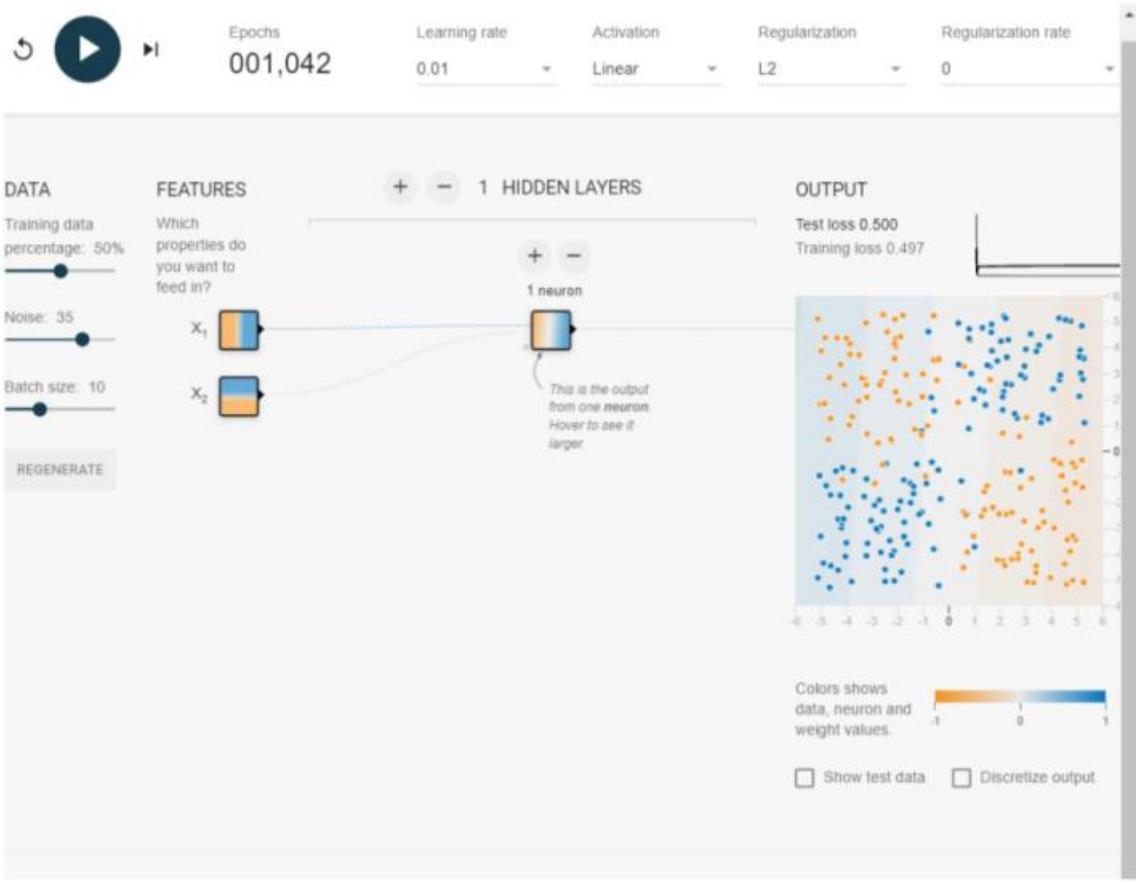
```
[82] my_model= '/content/model'
     with open(my_model, 'rb') as f:
       my_model = pickle.load(f)
```

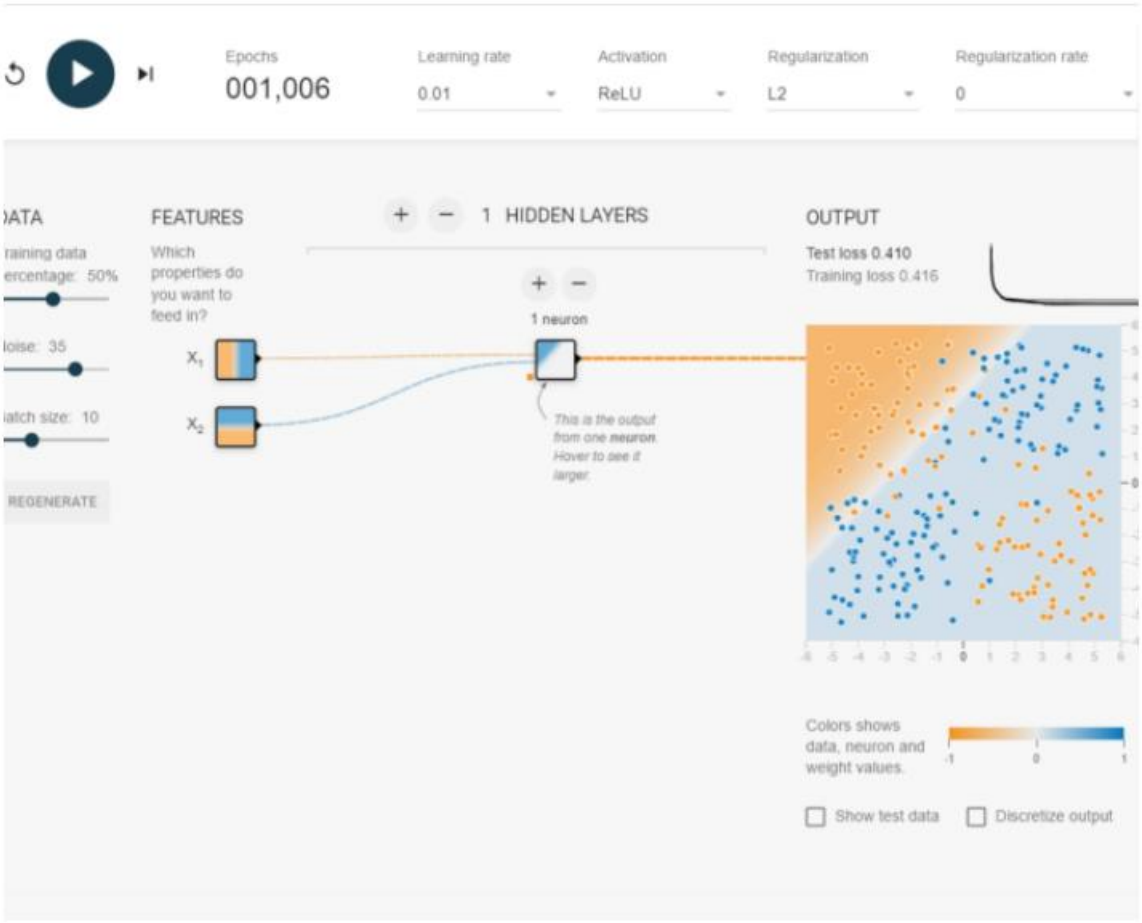
Question 3

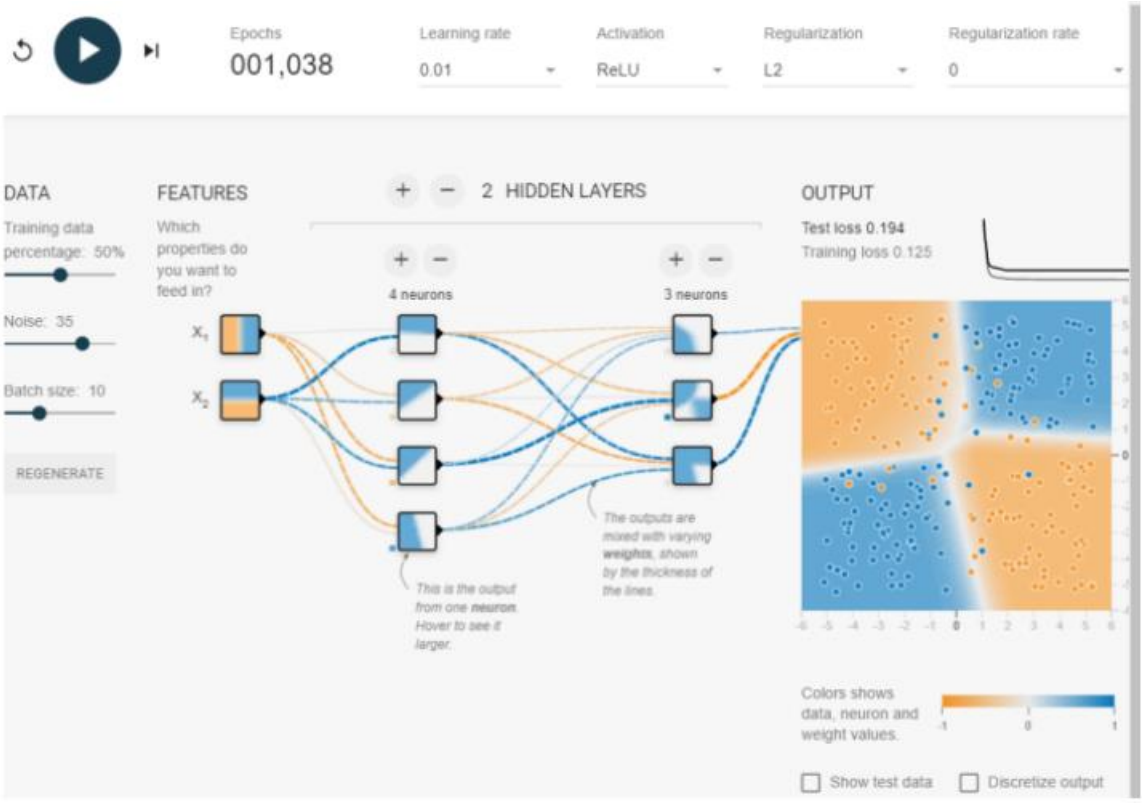
A)Google Playground

a. Increasing the model size does improve the fit and can make the model converge more quickly. However, the effect of model size on convergence depends on the complexity of the problem and the architecture of the network. Increasing the model size can make it easier for the network to fit the training data, but it can also increase the risk

of overfitting, which may lead to poor performance on new, unseen data. The optimal model size depends on the particular problem being solved, and finding the right balance between model size and generalization performance is an important part of neural network design.

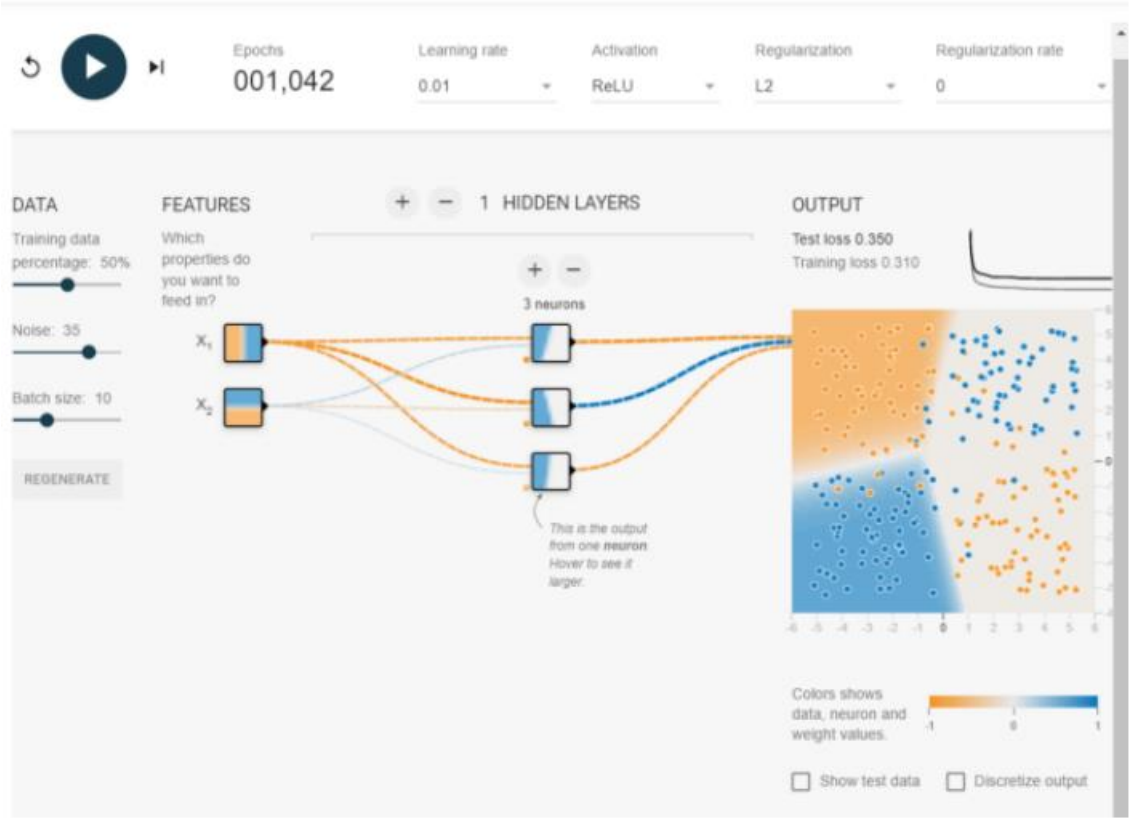


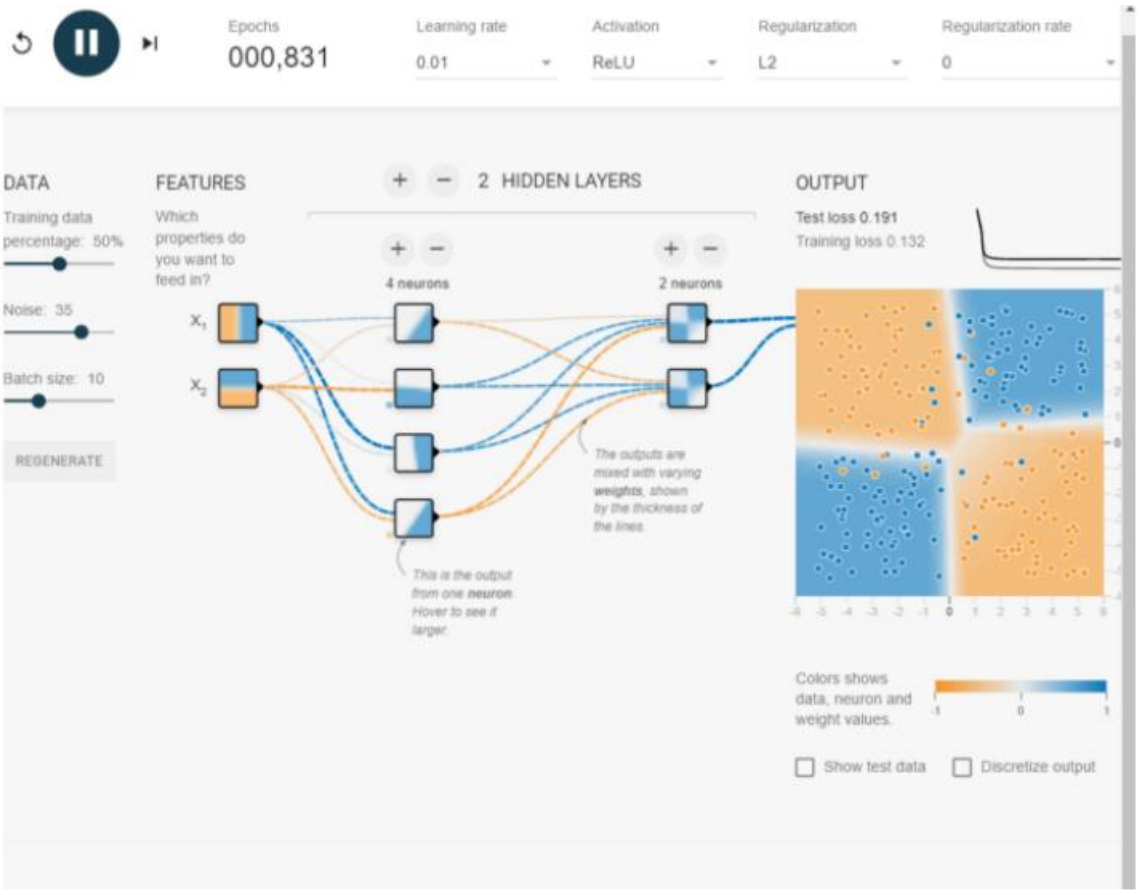




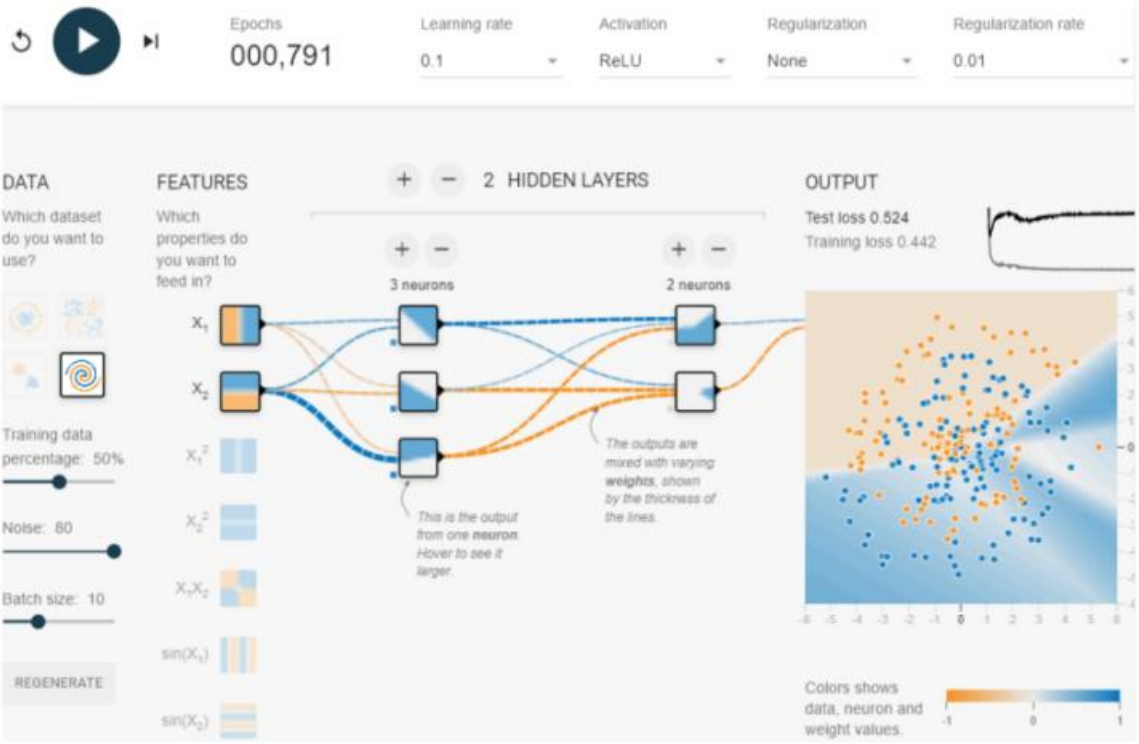
The shape will depend on the complexity of the problem and the initial random

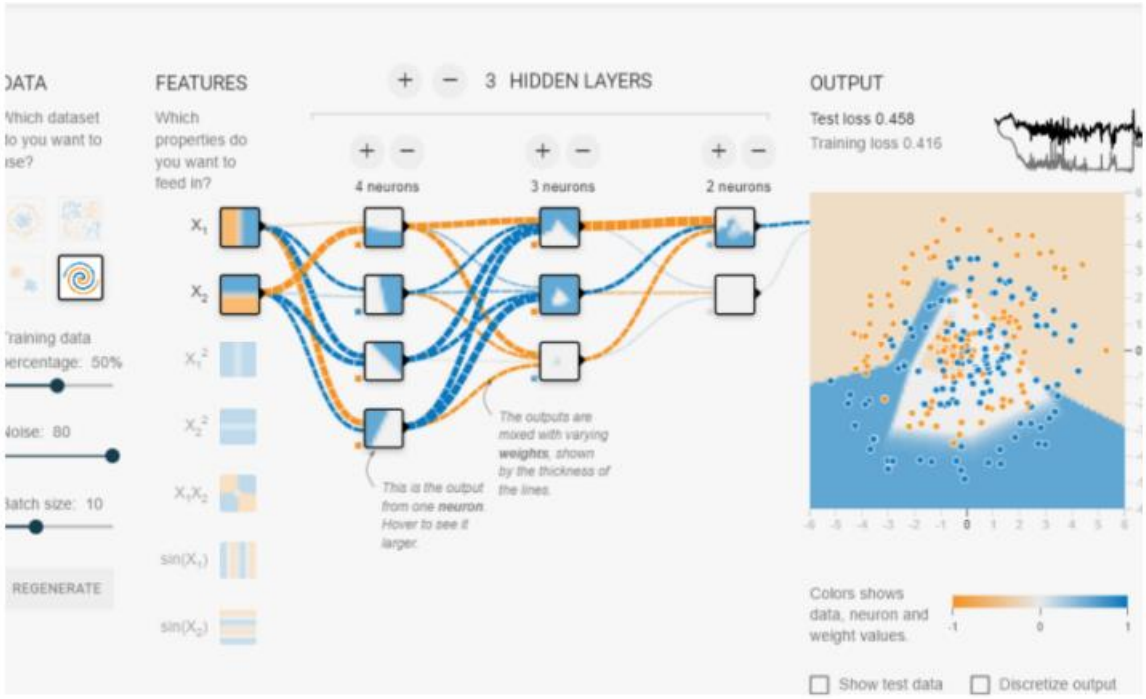
initialization. The additional complexity may improve the model's ability to fit the training data, but it may also increase the risk of overfitting





The additional layer and nodes add any additional stability to the results(loss reduced). The additional complexity may improve the model's ability to fit the training data, but it may also increase the risk of overfitting.





B. Neural Networks Desmos visualized