Reyhane Shahrokhian 99521361

HomeWork2 of Computational Intelligence Course

Dr. Mozayeni

Q1:

x_1	x_2	Output
0	0	1
0	1	1
1	0	1
1	1	0

$\alpha = 0.1$, bias = 0.1

$W_1 = 0.5$, $W_2 = 0.5$

$y = x_1 \times W_1 + x_2 \times W_2 + \text{bias}$, Error = (output - y)

New W_i = old $W_i + \alpha \times \text{Error} \times x_i$

New bias = old bias + $\alpha \times$ Error

Round1:

$y = 0 \times 0.5 + 0 \times 0.5 + 0.1 = 0.1$, Error = $1 - 0.1 = 0.9$

$W_1 = 0.5 + 0.1 \times 0.9 \times 0 = 0.5$

$W_2 = 0.5 + 0.1 \times 0.9 \times 0 = 0.5$

bias = $0.1 + 0.1 \times 0.9 = 0.19$

Round2:

$y = 0 \times 0.5 + 1 \times 0.5 + 0.19 = 0.69$, Error = $1 - 0.69 = 0.31$

$W_1 = 0.5 + 0.1 \times 0.31 \times 0 = 0.5$

$W_2 = 0.5 + 0.1 \times 0.31 \times 1 = 0.531$

bias = $0.19 + 0.1 \times 0.31 = 0.221$

Round3:

$y = 1 \times 0.5 + 0 \times 0.531 + 0.221 = 0.721$, Error = $1 - 0.721 = 0.279$

$W_1 = 0.5 + 0.1 \times 0.279 \times 1 = 0.5279$

$W_2 = 0.531 + 0.1 \times 0.279 \times 0 = 0.531$

bias = $0.221 + 0.1 \times 0.279 = 0.2489$

Round4:

$y = 1 \times 0.5279 + 1 \times 0.531 + 0.2489 = 1.3078$

Error = $0 - 1.3078 = -1.3078$

$W_1 = 0.5279 + 0.1 \times (-1.3078) \times 1 = 0.39712$

$W_2 = 0.531 + 0.1 \times (-1.3078) \times 1 = 0.40022$

bias = $0.2489 + 0.1 \times (-1.3078) = 0.11812$

…

Q2:

2-1:

The non-linear activation functions allow neural networks to model and learn complex and non-linear relationships in the data. While the linear activation functions produce an output that is a linear combination of the inputs which results in a network that can't capture and model complex patterns and relationships in the data.

2-2:

- Random Bias and Zero Weights:

  Initializing the bias randomly can help the model escape local minimums during training. If weights are initialized as zero, it can lead to symmetric weights in the network. In each layer, the weights would start the same, and as a result, the neurons in that layer would

compute the same values which makes it difficult for the model to learn different features from the data.

In general, training in this scenario might be slow and could lead to convergence issues.

- Zero Bias and Random Weights:

  If bias is initialized as zeros, this means that neurons in the network initially have no bias, which can affect the network's representational power. Initializing the weights randomly helps in breaking symmetry and allows each neuron to learn different features from the data.

  Generally, training with this scenario is much better than the previous one, as random weight initialization helps the network learn meaningful features from the data and allows for faster and more stable convergence.

In conclusion, random weight initialization and small non-zero bias values is mostly the best practice in neural network training because it helps the network to start learning effectively and continues faster without leading to convergence problems.

2-3:

The ability of a neural network model to generalize effectively to unseen data is influenced by various factors, including the model architecture, the amount of training data, and the presence of regularization techniques. In general, neural network models that are simple and less complex, have the suitable regularization, and are trained on a sufficient amount of diverse data tend to have better generalization ability.

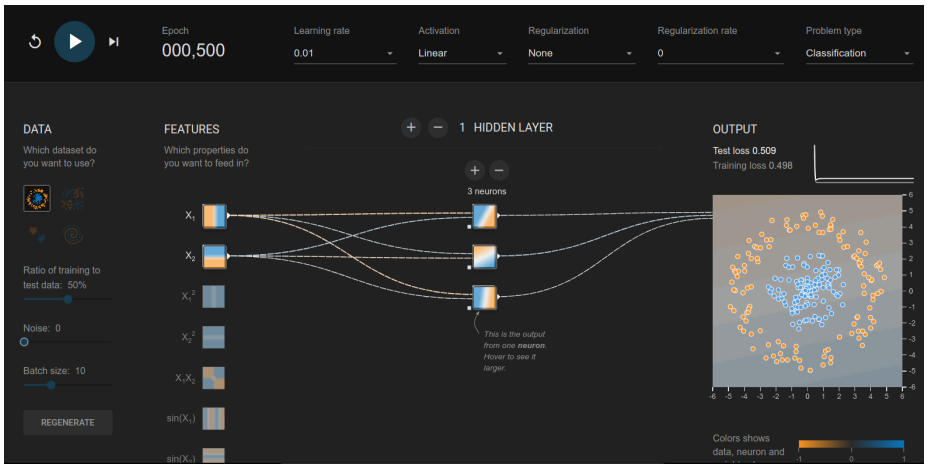
2-4:

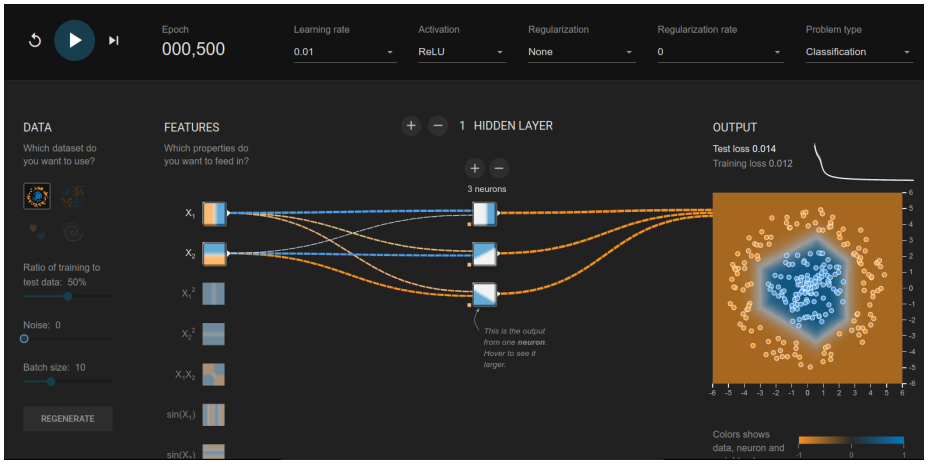
Actually this formula is a second-order optimization method. Due to that, it can converge faster

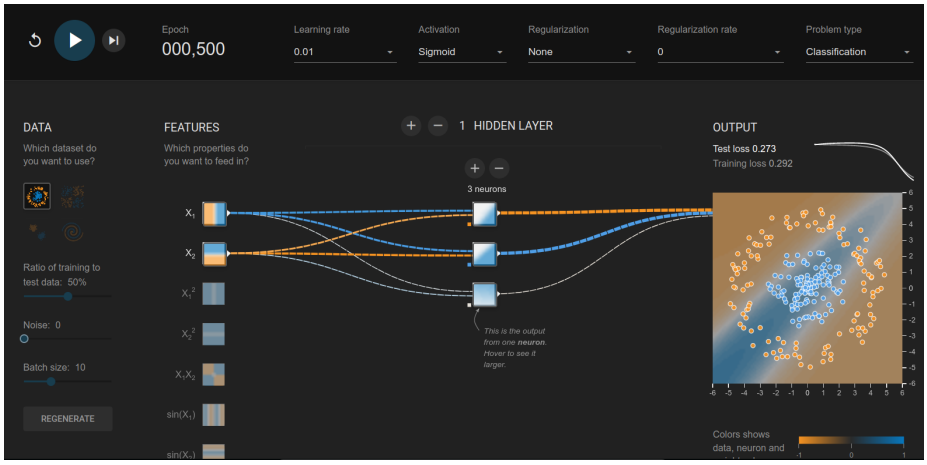
that leads to quicker training. Another benefit of using this method is that they are less likely to get stuck in shallow local minima which causes the model to have more accurate solutions. However, the more computational power is needed that makes this method impractical for models with a large number of parameters.

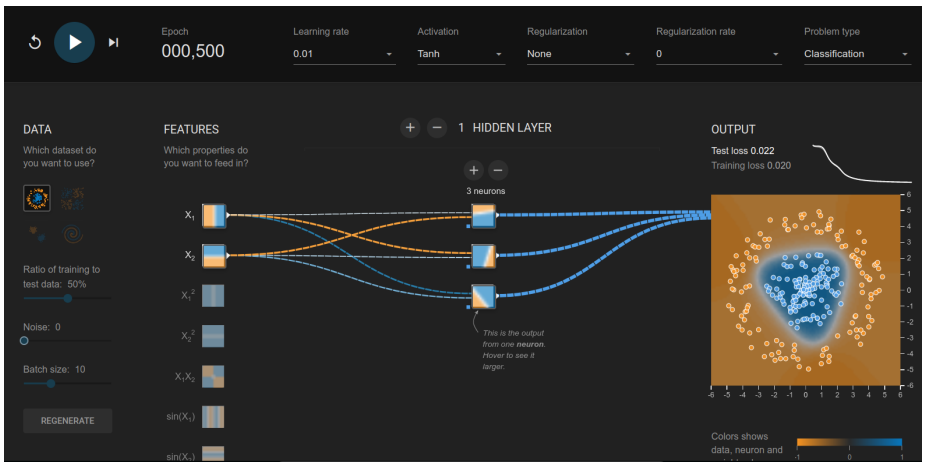
Q3:

Dataset 1:



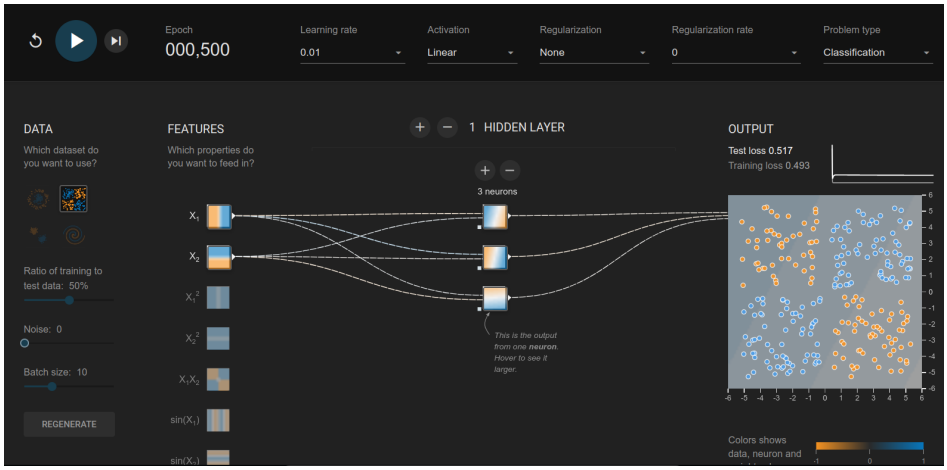


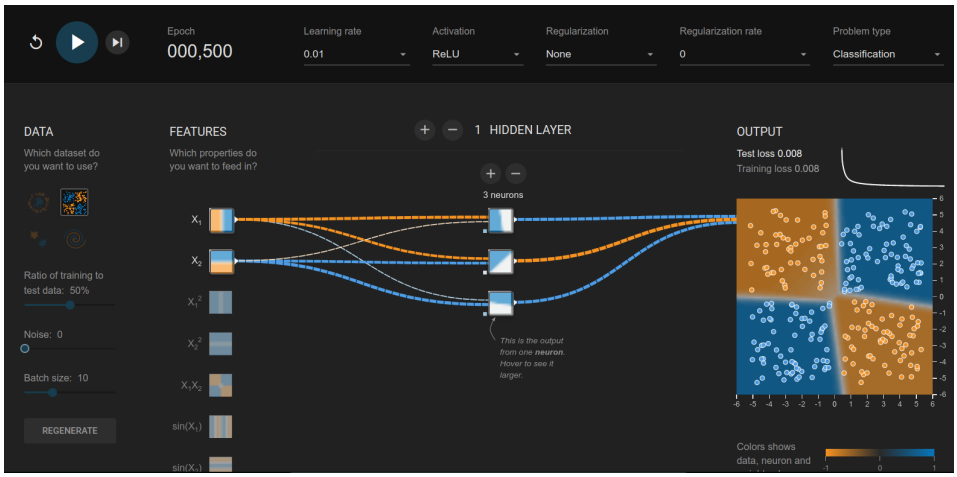


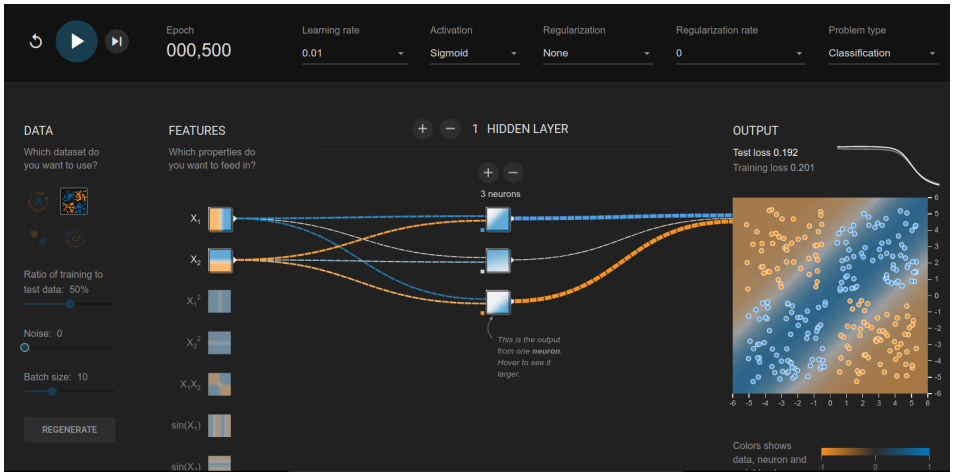


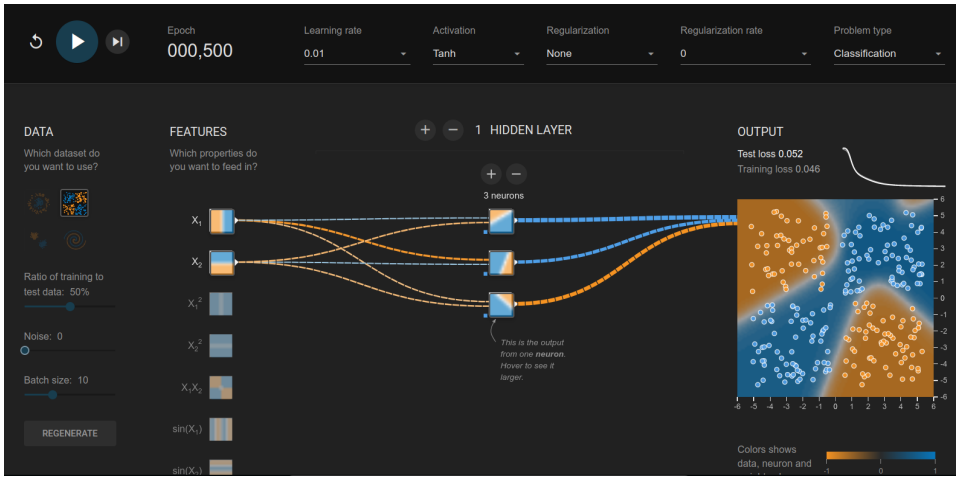
As you can see, the loss value with ReLU activation function is the best among others for the wanted model train on the first dataset. From the data distribution we could easily guess that using linear activation function is unprofitable and the results also prove it. According to the fact that by using sigmoid, outputs are not centered around zero, in the scenarios where data should be like that, tanh activation function is often used. Thus, tanh performs better than sigmoid function.

Dataset 2:


Epoch
000,500
Learning rate
0.01
Activation
Linear
Regularization
None
Regularization rate
0
Problem type
Classification
DATA
Which dataset do you want to use?
Ratio of training to test data: 50%
Noise: 0
Batch size: 10
REGENERATE
FEATURES
Which properties do you want to feed in?
1 HIDDEN LAYER
3 neurons
This is the output from one neuron. Hover to see it larger.
OUTPUT
Test loss 0.517
Training loss 0.493
Colors shows data, neuron and

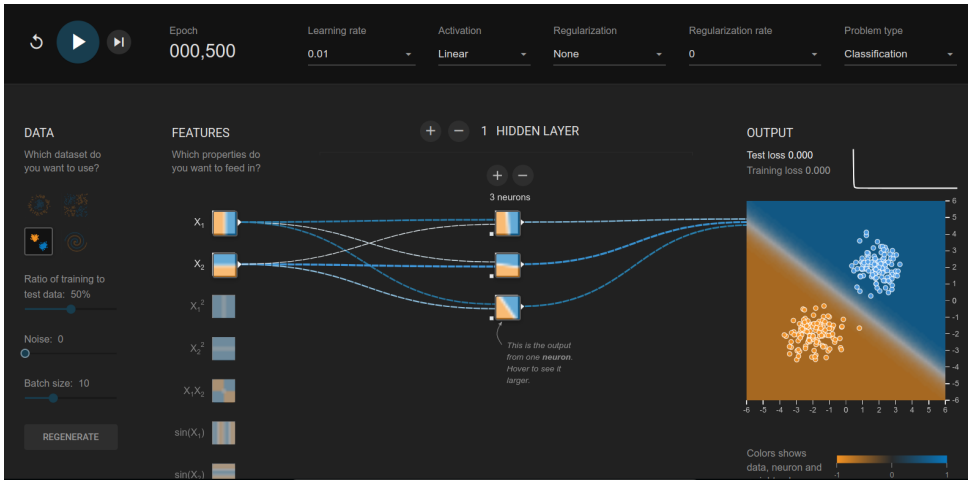

Epoch
000,500
Learning rate
0.01
Activation
ReLU
Regularization
None
Regularization rate
0
Problem type
Classification
DATA
Which dataset do you want to use?
Ratio of training to test data: 50%
Noise: 0
Batch size: 10
REGENERATE
FEATURES
Which properties do you want to feed in?
1 HIDDEN LAYER
3 neurons
This is the output from one neuron. Hover to see it larger.
OUTPUT
Test loss 0.008
Training loss 0.008
Colors shows data, neuron and

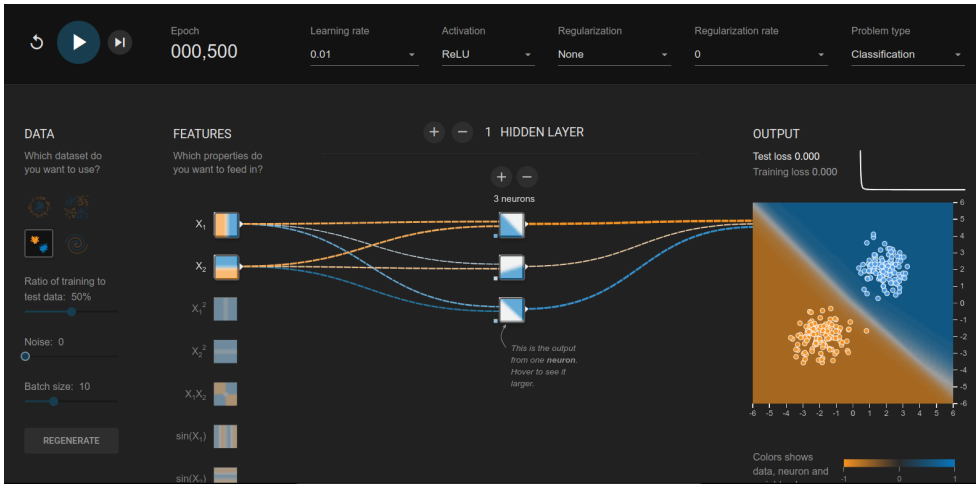

Epoch
000,500
Learning rate
0.01
Activation
Sigmoid
Regularization
None
Regularization rate
0
Problem type
Classification
DATA
Which dataset do you want to use?
Ratio of training to test data: 50%
Noise: 0
Batch size: 10
REGENERATE
FEATURES
Which properties do you want to feed in?
1 HIDDEN LAYER
3 neurons
This is the output from one neuron. Hover to see it larger.
OUTPUT
Test loss 0.192
Training loss 0.201
Colors shows data, neuron and

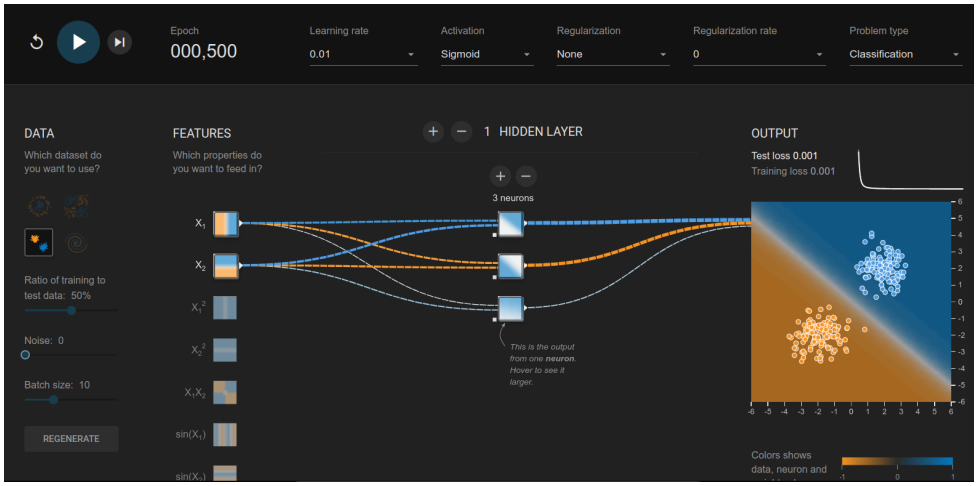


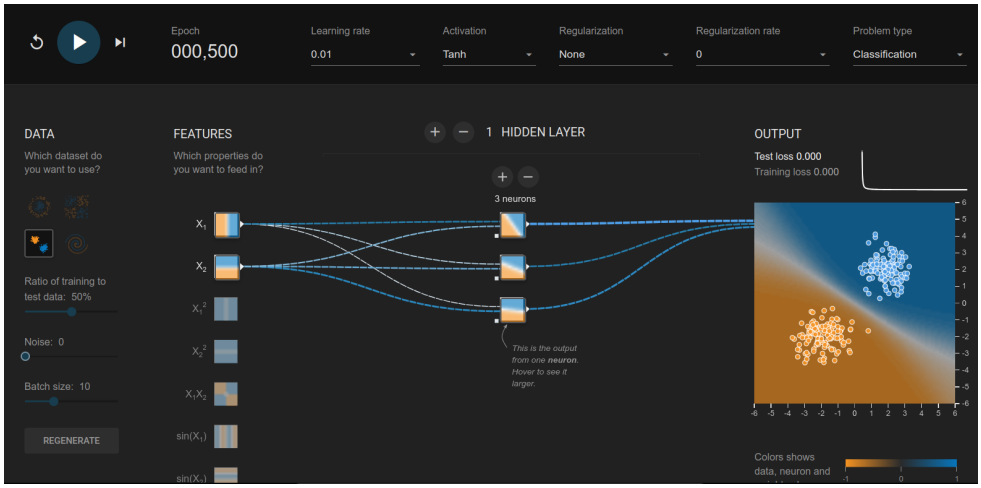
As you can see, the loss value with ReLU activation function is the best among others for the wanted model train on the second dataset. Like the previous one, from the data distribution we could easily guess that using linear activation function is unprofitable and the results also prove it. We know that using tanh is mostly better than sigmoid due to the fact that it is simillar to sigmoid but the significant problems of using sigmoid are solved by the introduction of tanh. Therefore, tanh is working better on this dataset.

Dataset 3:



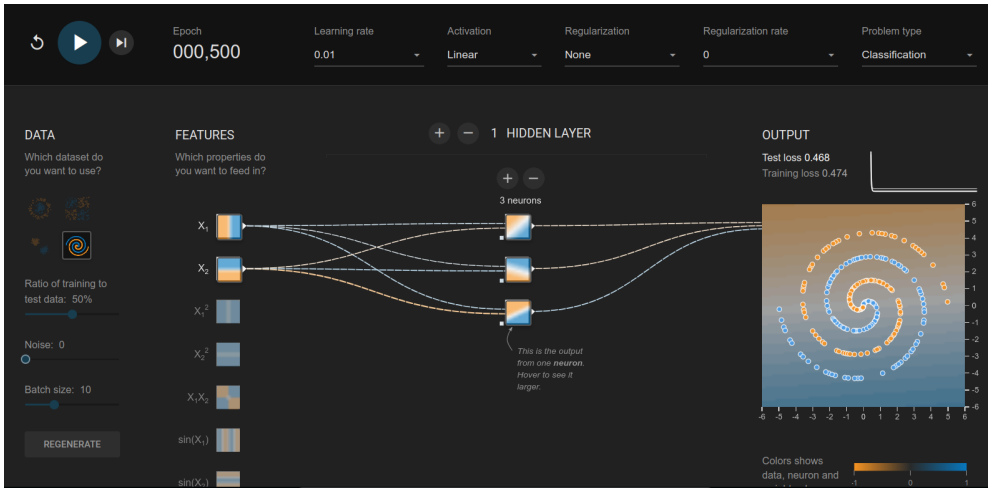

Epoch
000,500
Learning rate
0.01
Activation
ReLU
Regularization
None
Regularization rate
0
Problem type
Classification
DATA
Which dataset do you want to use?
FEATURES
Which properties do you want to feed in?
1 HIDDEN LAYER
3 neurons
OUTPUT
Test loss 0.000
Training loss 0.000
Ratio of training to test data: 50%
Noise: 0
Batch size: 10
REGENERATE
This is the output from one neuron. Hover to see it larger.
Colors shows data, neuron and

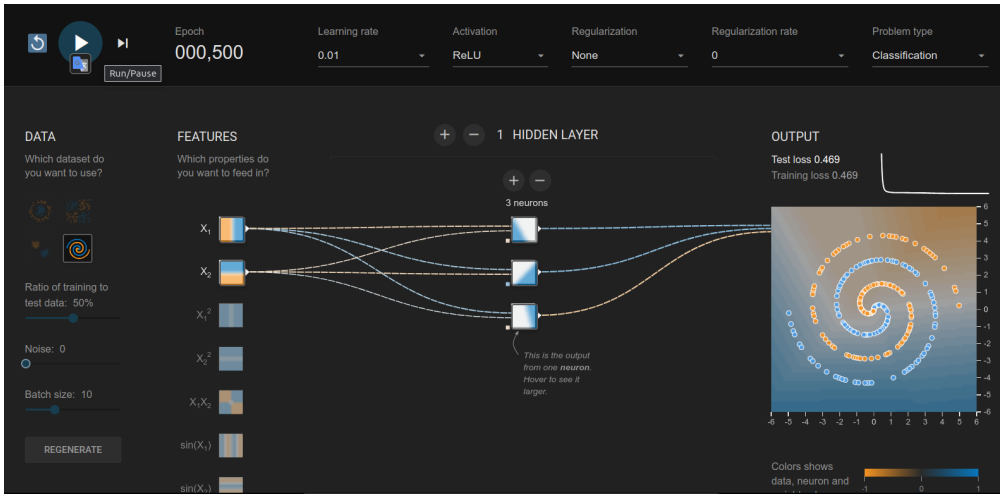

Epoch
000,500
Learning rate
0.01
Activation
Sigmoid
Regularization
None
Regularization rate
0
Problem type
Classification
DATA
Which dataset do you want to use?
FEATURES
Which properties do you want to feed in?
1 HIDDEN LAYER
3 neurons
OUTPUT
Test loss 0.001
Training loss 0.001
Ratio of training to test data: 50%
Noise: 0
Batch size: 10
REGENERATE
This is the output from one neuron. Hover to see it larger.
Colors shows data, neuron and

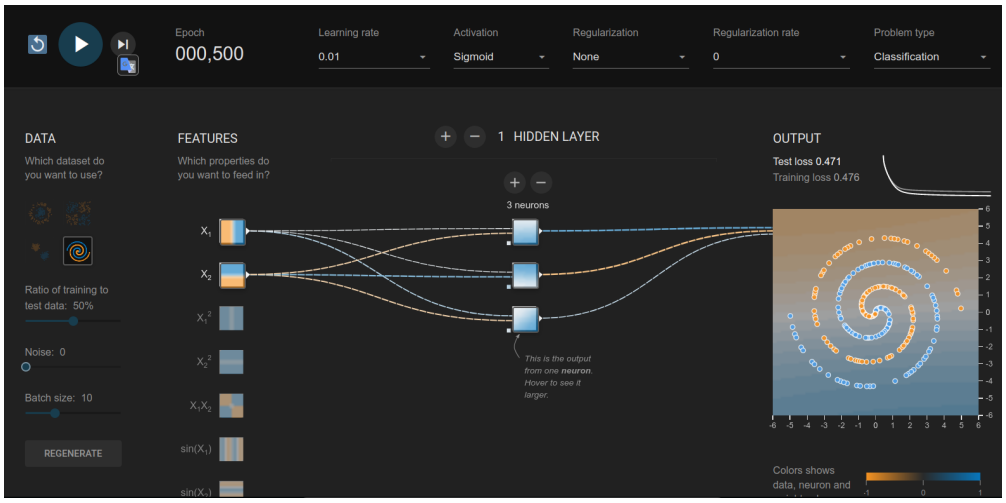

Epoch
000,500
Learning rate
0.01
Activation
Tanh
Regularization
None
Regularization rate
0
Problem type
Classification
DATA
Which dataset do you want to use?
FEATURES
Which properties do you want to feed in?
1 HIDDEN LAYER
3 neurons
OUTPUT
Test loss 0.000
Training loss 0.000
Ratio of training to test data: 50%
Noise: 0
Batch size: 10
REGENERATE
This is the output from one neuron. Hover to see it larger.
Colors shows data, neuron and

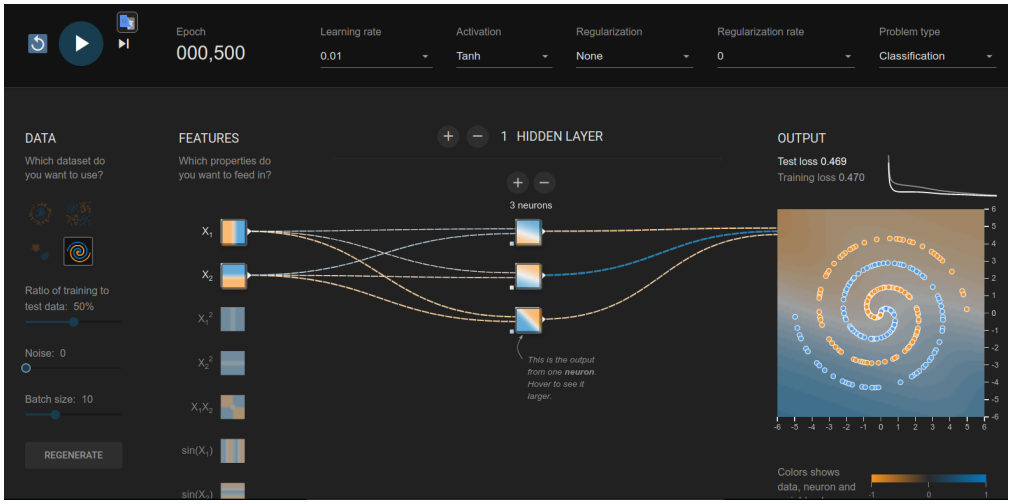
In this Scenario, despite other datasets, data distribution is completely linear. So, by using all 4 activation functions, we can quickly classify the data with the minimum loss that is possible (almost zero).

Dataset 4:









In that case, we have a complex data distribution in a way that none of the activation functions couldn't decrease the loss. Hence, If we want to get acceptable results, we have to implement a more complex model by adding more hidden layers with further neurons.

Q4:

The model that I've created for this problem is as below:

```
 6 model = Sequential([
 7     Dense(25),
 8     Dense(16, activation='relu'),
 9     Dense(10, activation='softmax')
10 ])
11 model.build(input_shape=(None, 25))
```

```
 1 model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 25)                650

 dense_1 (Dense)             (None, 16)                416

 dense_2 (Dense)             (None, 10)                170

=================================================================
Total params: 1236 (4.83 KB)
Trainable params: 1236 (4.83 KB)
Non-trainable params: 0 (0.00 Byte)
```

The input dimension is 25 and the output dimension is 10 (Hoad dataset is a Persian written digit of 0 to 9). I tried 16 and 32 neurons for the hidden layer and the result with 16 neurons was better than 32, so I chose 16.

For the next step, the adam as optimizer and the categorical_crossentropy as loss function is chosen because it is a multi class classification task. I also set an accuracy metric to observe the utility of the model easily.

```
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

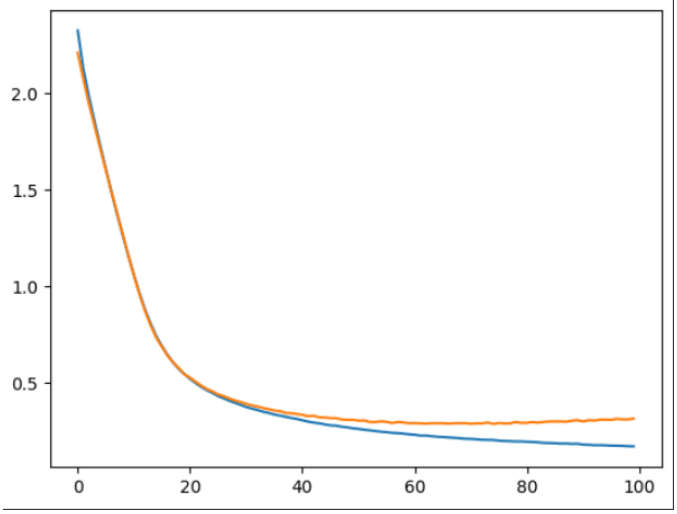
The result at the last epoch is as below:

```
Epoch 100/100
16/16 [==============================] - 0s 6ms/step - loss: 0.1719 - accuracy: 0.9450 - val_loss: 0.3156 - val_accuracy: 0.8950
```

And here are the plots:

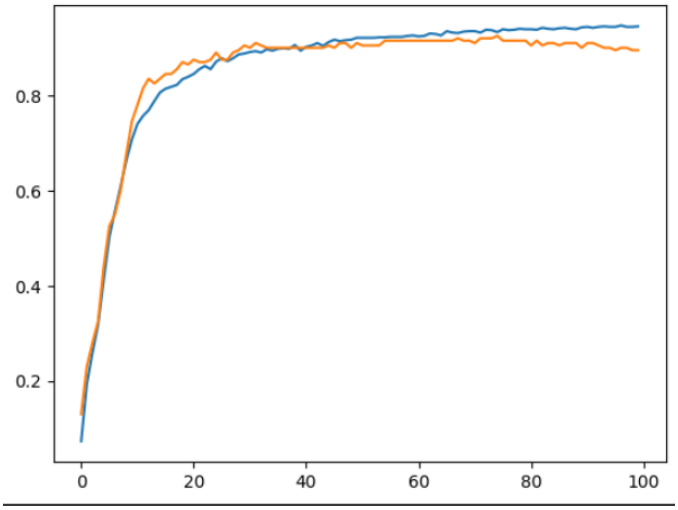
```
2 plt.plot(MLP_model.history["loss"])
3 plt.plot(MLP_model.history["val_loss"])
```

```
[<matplotlib.lines.Line2D at 0x7a77fa5cbbb0>]
```




```
2 plt.plot(MLP_model.history["accuracy"])
3 plt.plot(MLP_model.history["val_accuracy"])
```

```
[<matplotlib.lines.Line2D at 0x7a77f948b040>]
```



From the results, we can find that our model is overfitted because the loss value with training data is much less than the loss value with validation data.

Q5:

First we need to initialize layer size (input layer size = 2, output layer size= 1 and I choose hidden layer size = 4), weights and biases randomly in order to get started.

```
1 # train data
2 x = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
3 y = np.array([1, 0, 0, 1])
```

```
1 # layer sizes
2 input_size = 2
3 hidden_size = 4
4 output_size = 1
5
6 # initialize weights and biases randomly
7 input_weights = np.random.rand(input_size, hidden_size)
8 hidden_weights = np.random.rand(hidden_size, output_size)
9 hidden_bias = np.random.rand(1, hidden_size)
10 output_bias = np.random.rand(1, output_size)
```

Then, I defined the sigmoid activation function and the derivative of that.

```python
# sigmoid activation function
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# derivative of the sigmoid
def sigmoid_derivative(x):
    return x * (1 - x)
```

After that, It's time for the training loop with 10000 epochs and learning_rate = 0.1. In training loop forward propagation, backpropagation and updating weights and biases are the steps that should be implemented.

```python
# learning rate
learning_rate = 0.1

# training loop
for epoch in range(10000):
    # forward propagation
    hidden_layer_output = sigmoid(np.dot(x, input_weights) + hidden_bias)
    output_layer_output = sigmoid(np.dot(hidden_layer_output, hidden_weights) + output_bias)

    # loss
    loss = y.reshape(-1, 1) - output_layer_output

    # backpropagation
    d_output = loss * sigmoid_derivative(output_layer_output)
    loss_hidden = d_output.dot(hidden_weights.T)
    d_hidden = loss_hidden * sigmoid_derivative(hidden_layer_output)

    # update weights and biases
    input_weights += x.T.dot(d_hidden) * learning_rate
    hidden_weights += hidden_layer_output.T.dot(d_output) * learning_rate
    hidden_bias += np.sum(d_hidden, axis=0, keepdims=True) * learning_rate
    output_bias += np.sum(d_output, axis=0, keepdims=True) * learning_rate
```

At the end of the training, the output of output_layer is printed.

```python
# output of the model
print("Output after training:")
print(output_layer_output)
print((output_layer_output >= 0.5).astype(int))
```

```
Output after training:
[[0.94541764]
 [0.04377188]
 [0.07997694]
 [0.92682426]]
[[1]
 [0]
 [0]
 [1]]
```

Q6:

First of all the MNIST dataset should be loaded and then the model is defines and compiled as follows:

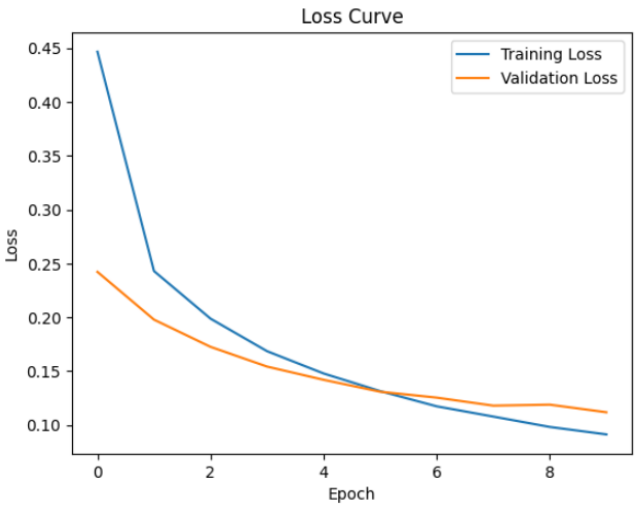
```python
# MLP model
model = keras.Sequential([
    keras.layers.Flatten(input_shape=(28, 28)),
    keras.layers.Dense(32, activation='relu'),
    keras.layers.Dense(10, activation='softmax')
])

# Compile the model
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
```

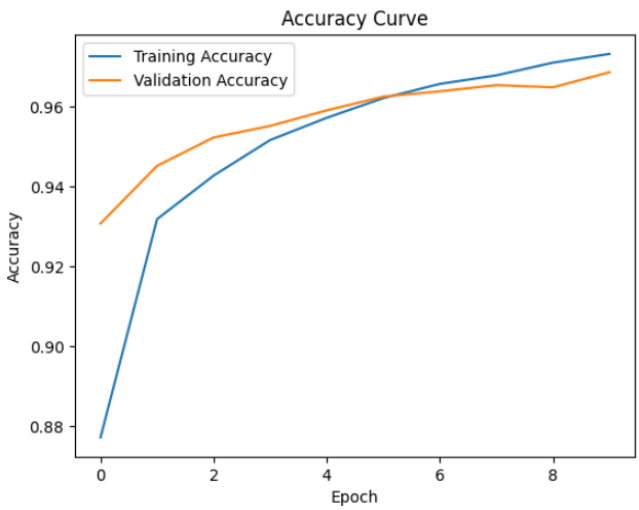
I tried with different hidden layers and with other settings the model overfits. The categorical cross entropy loss function is chosen because it's a multi-class classification task.

Then, I train the model and I set validation_split as 0.15 to also have validation data for checking overfitting issues.

```python
# train the model
history = model.fit(train_x, train_y, epochs=10, batch_size=64, validation_split=0.15)
```

```
Epoch 1/10
797/797 [==============================] - 4s 4ms/step - loss: 0.4465 - accuracy: 0.8772 - val_loss: 0.2422 - val_accuracy: 0.9308
Epoch 2/10
797/797 [==============================] - 2s 3ms/step - loss: 0.2430 - accuracy: 0.9319 - val_loss: 0.1979 - val_accuracy: 0.9452
Epoch 3/10
797/797 [==============================] - 3s 3ms/step - loss: 0.1988 - accuracy: 0.9428 - val_loss: 0.1726 - val_accuracy: 0.9523
Epoch 4/10
797/797 [==============================] - 3s 4ms/step - loss: 0.1686 - accuracy: 0.9517 - val_loss: 0.1543 - val_accuracy: 0.9552
Epoch 5/10
797/797 [==============================] - 3s 3ms/step - loss: 0.1479 - accuracy: 0.9573 - val_loss: 0.1420 - val_accuracy: 0.9591
Epoch 6/10
797/797 [==============================] - 3s 3ms/step - loss: 0.1315 - accuracy: 0.9622 - val_loss: 0.1310 - val_accuracy: 0.9626
Epoch 7/10
797/797 [==============================] - 2s 3ms/step - loss: 0.1174 - accuracy: 0.9657 - val_loss: 0.1255 - val_accuracy: 0.9639
Epoch 8/10
797/797 [==============================] - 3s 4ms/step - loss: 0.1079 - accuracy: 0.9679 - val_loss: 0.1181 - val_accuracy: 0.9654
Epoch 9/10
797/797 [==============================] - 2s 3ms/step - loss: 0.0983 - accuracy: 0.9711 - val_loss: 0.1190 - val_accuracy: 0.9649
Epoch 10/10
797/797 [==============================] - 3s 3ms/step - loss: 0.0914 - accuracy: 0.9733 - val_loss: 0.1119 - val_accuracy: 0.9687
```





At the end, It’s the time for testing the model:

```
# Evaluate the model on the test dataset
test_loss, test_accuracy = model.evaluate(test_x, test_y)

print("Test Loss:", test_loss)
print("Test Accuracy:", test_accuracy)
```

```
313/313 [==============================] - 1s 2ms/step - loss: 0.1136 - accuracy: 0.9674
Test Loss: 0.11356817930936813
Test Accuracy: 0.9674000144004822
```