Model Answer Exercise 1
Experimenting with a simple Neural Network

We come back to the playground exercise already presented this morning:

https://playground.tensorflow.org

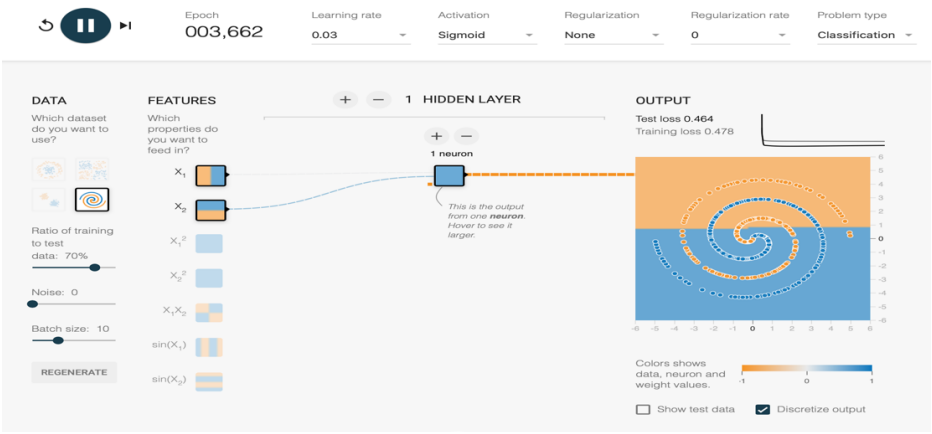
Unless otherwise indicated, we work with the following hyper-parameters:
Learning rate =0.03, Batch Size=10, Noise=0, Training Data=70%.
Since there is a total of 500 data points, there are 350 (or 70%) training data and 150 test data.
We examine the behaviour of the networks until about 3500 epochs.

It is important to note that there will be differences between different runs obtained with the same hyperparameters, because the initial values of the neural network parameters (weights and bias terms) are sampled randomly. The dataset itself may also change if you click the *Regenerate* Button at the bottom left of the screen.

We work on a classification problem on the Spiral dataset. The yellow dots have a value of -1 and the blue dots have a value of +1.
The comments below characterize the most frequent behaviour we tend to observe.

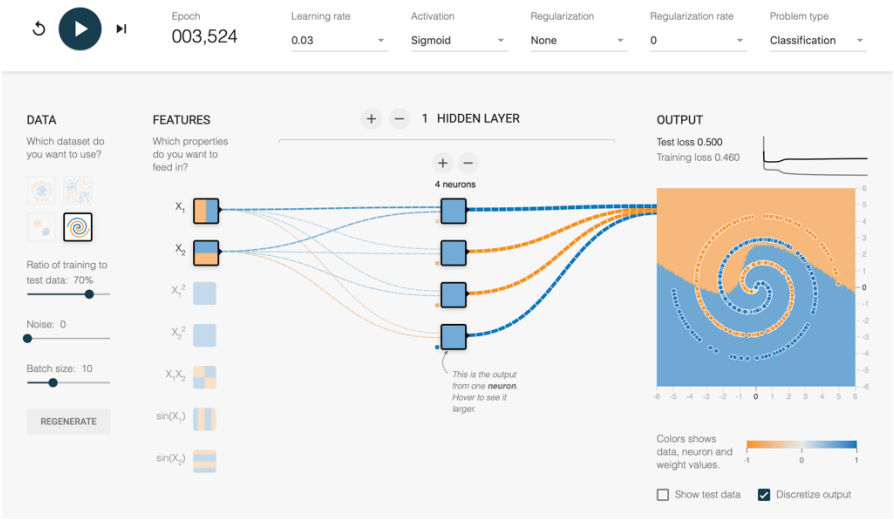
1. Use the sigmoid activation function and no hidden layer. What do you observe? How do you interpret it?

   With a sigmoid activation function and no hidden layer, we are exactly in the situation of logistic regression. We know that with a sigmoid function the decision boundary is linear.

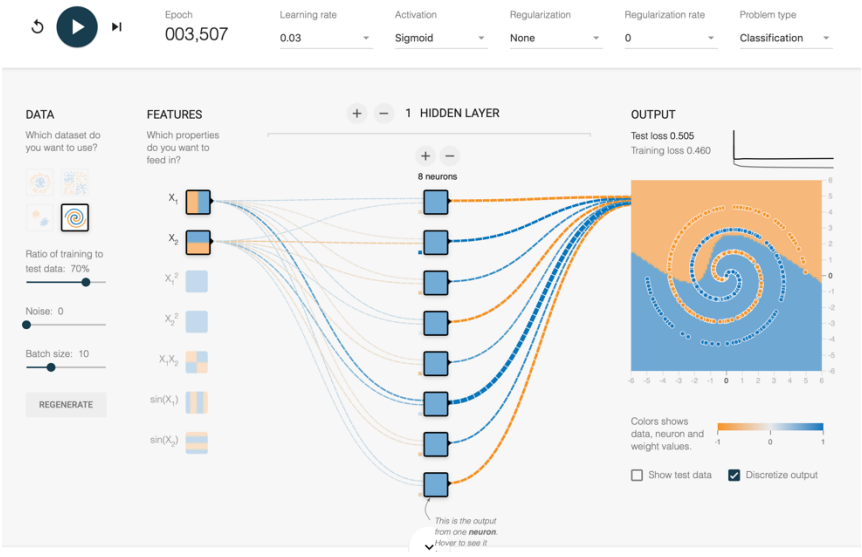


2. Now put just one hidden layer, a sigmoid activation function, and make the number of neurons in the hidden layer equal to 4. Then make it equal to the maximum offered by the application, that is 8. What is the number of trained parameters and what do you observe in each case?

   With 4 neurons in the hidden layer, we see that after 3500 epochs a slight non-linearity appears in the Decision Boundary, but on some of the obtained models, there is no progress at all. The number of parameters (considering that we have bias terms) to train is 3x4+5=17, which is not enough considering that we have 350 data points in the training set.

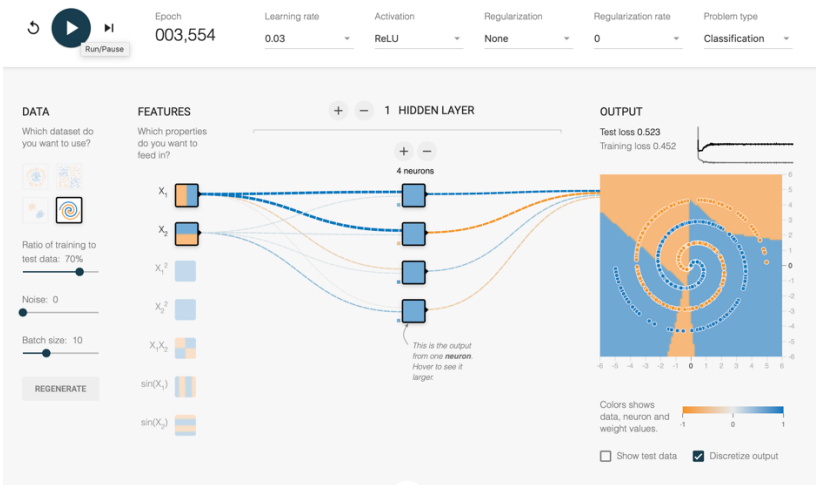


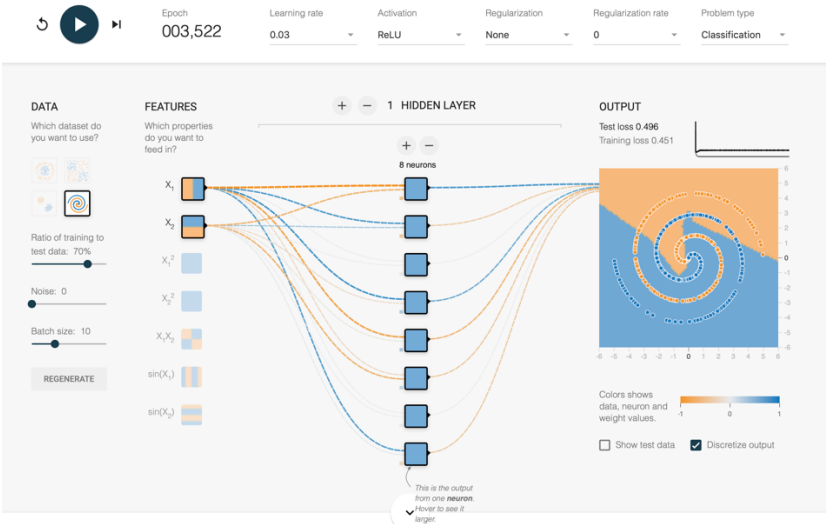
- **With 8 neurons in the hidden layer, and still the sigmoid activation function, we see a slow evolution of the previous non-linearity – but the loss function does not change anymore after a number of iterations - and the Decision Boundary remains simplistic. Sigmoid activations are easier to saturate: once a sigmoid reaches either its left or right plateau, it leads to derivatives that are very close to 0. Also we have only 3x8+9=33 parameters to train, which is still not enough.**



3. Now do the same as in question 3, but using the ReLU activation function. What do you observe?

   The changes are faster and more apparent with the ReLU, because we do not have the saturation effects of the sigmoid function. But the result is still disappointing after 3500 epochs, both with 4 neurons or 8 neurons in the hidden layer. Clearly, more hidden layers are needed in order to increase the number of trained parameters.

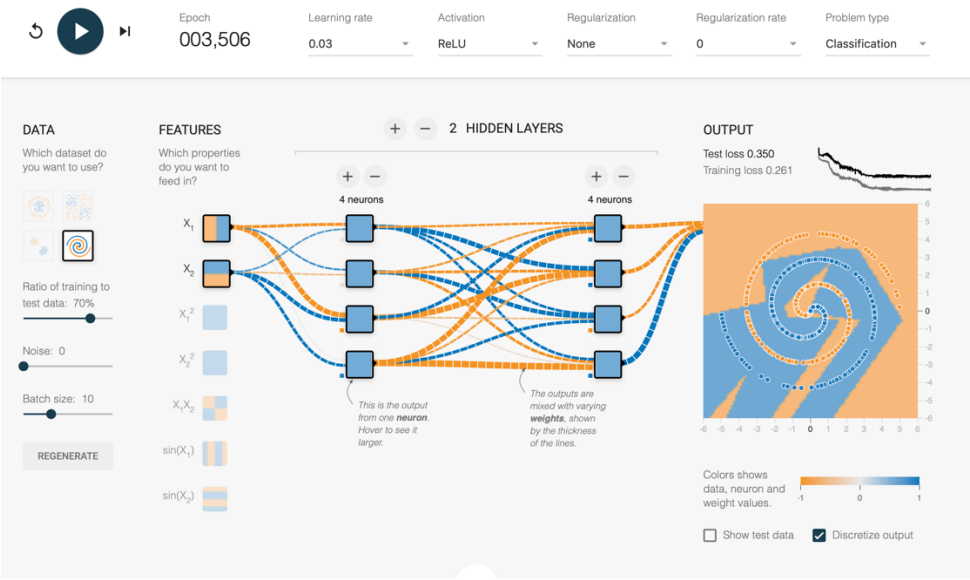




4. Now use two hidden layers each with four neurons and the ReLU activation function. How many trained parameters do you have? What do you observe?

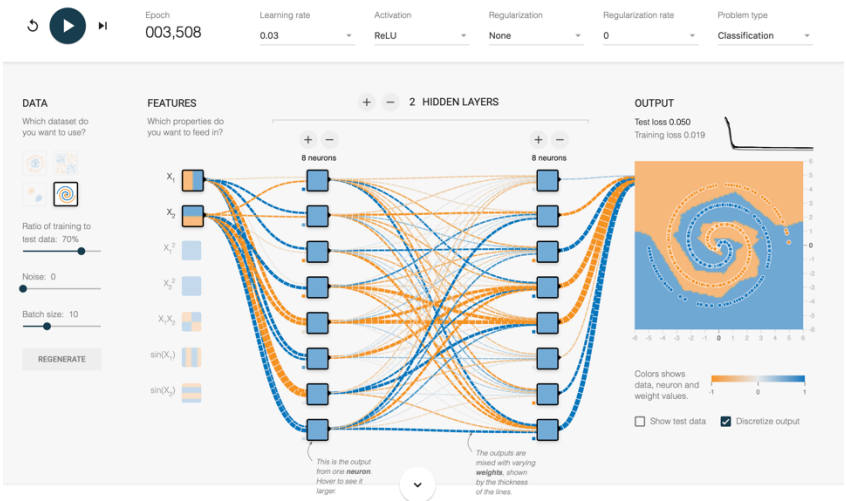
   There is more change occurring with the ReLU activation function, but the result is still far from satisfactory. The number of trained parameters (with bias terms) is

3x4+5x4+5=37, which is still not enough.



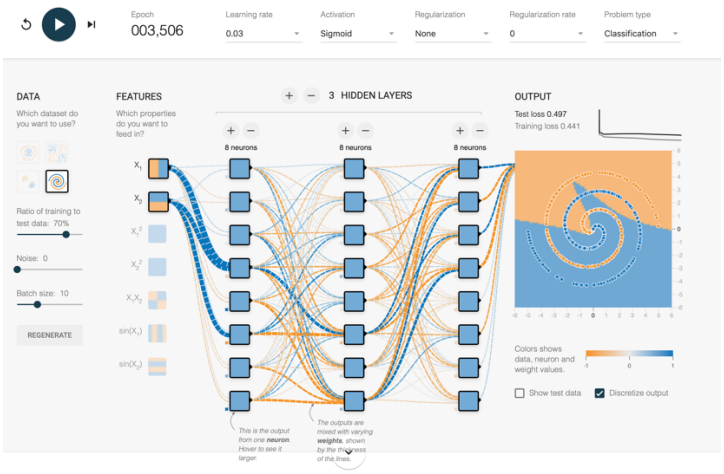
6 Now try two hidden layers with 8 neurons each and a ReLU activation function. How many trained parameters do you have? What do you observe ?

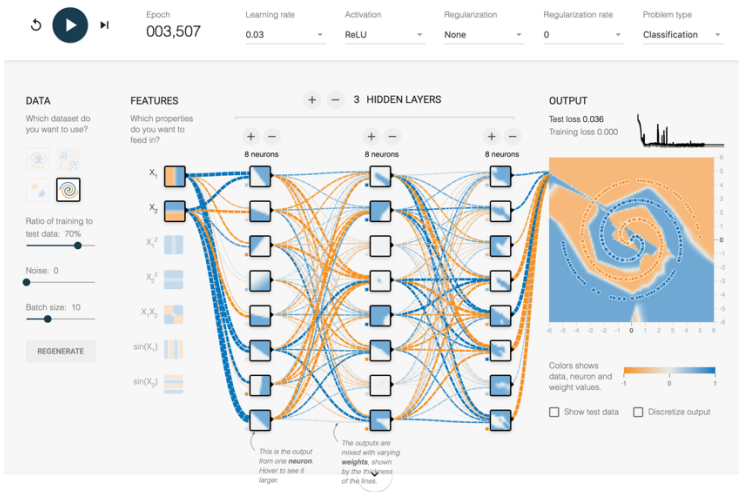
We obtain a more satisfactory result, with a significantly decreased training loss (0.019) and test loss (0.050), which tend to be nicely parallel as a function of the number of epochs, meaning that there is no over-fitting in this example. The number of layers, combined with the number of neurons per layer, has led to a much better result, as the number of trained parameters is now 3x8+9x8+9=105.



7. Now try three hidden layers with 8 neurons each. What happens if you compare sigmoid and ReLU?

With three hidden layers of 8 neurons each (the number of trained parameters is now 3x8+9x8+9x8+9=177), the ReLU does much better than the sigmoid (which shows little progress) even after just about 500 epochs. The ReLU test loss is now lower than with two hidden layers at 0.036. It is interesting (by de-activating the "Discrete Output" visualization) to see the maps corresponding to each neuron of the second and third hidden layers for the ReLU, where some very strongly non-linear shapes appear. The fact that ReLU does consistently better than sigmoid explains why ReLU is almost always preferred, except for the last layer of the network.

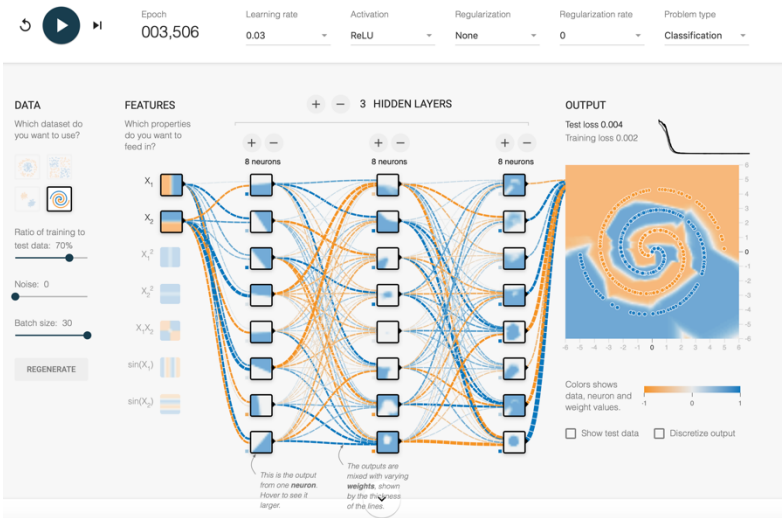




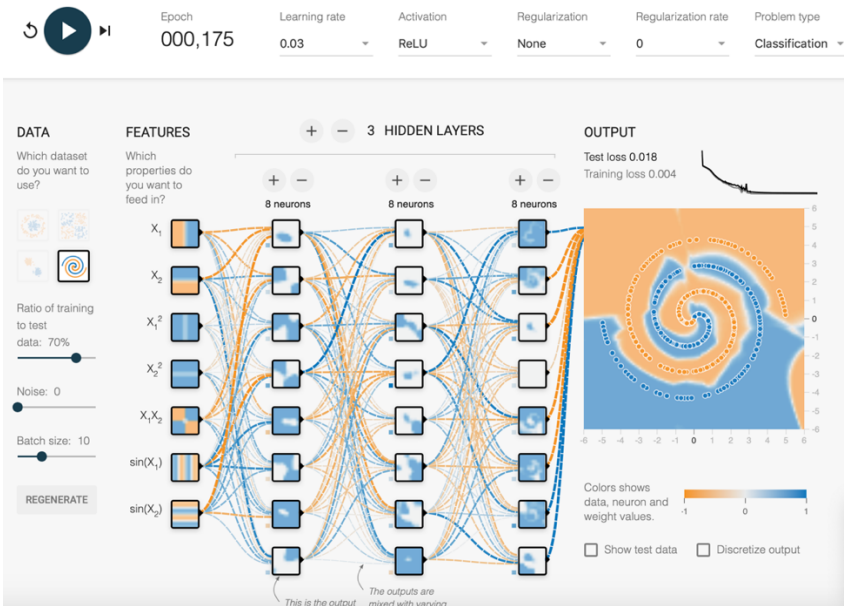
8. Now keep the three layers with 8 neurons and change the batch size to 30, and then 1. What happens if you use ReLU?

Following this morning's discussion, if we use a batch size of 1, we do stochastic gradient descent. In this case (result not shown below) the training loss can evolve in a rather chaotic manner: it can significantly increase after reaching a minimum, because of the impact of individual data points on the gradient used and hence on the training loss after the associated iteration. With a batch size of 30, we obtain a test loss of 0.004, avery good score, and a better one than with a batch size of 10. The larger

number of data used in each iteration tends to produce a smoother and regular path towards a very low test loss.



9. Now that we have obtained a good model see what happens if you introduce all the 7 input variables : $X_1, X_2, X_1^2, X_2^2, X_1X_2, \sin X_1, \sin X_2$.



Please note that the above dataset is different from those of the previous questions, but this is not an issue as the number of input features is also quite different from that of the previous examples. We see that in less than 200 epochs (instead of 3400 in the previous examples) the network converges to a very good model. By using non-linear functions of the input coordinates, we provide input features that are better suited to the task. The use of seven instead of two input features also drastically increases the

number of parameters (to 8x8+9x8+9x8+9= 217) and hence the risk of over-fitting, which may be why the test loss is four times the training loss.

10. Now assume that the data are affected by noise. Just use the maximum noise value of 50. The application is not very clear about the mechanism by which the noise affects the initial data classes. This does not matter, just assume that, after introducing this noise value of 50, you are now dealing with new binary class values at each point. What do you observe?

With noise present, the network still does a good job of producing a fit to the data, even though the difference between test and training loss shows a risk of overfitting.