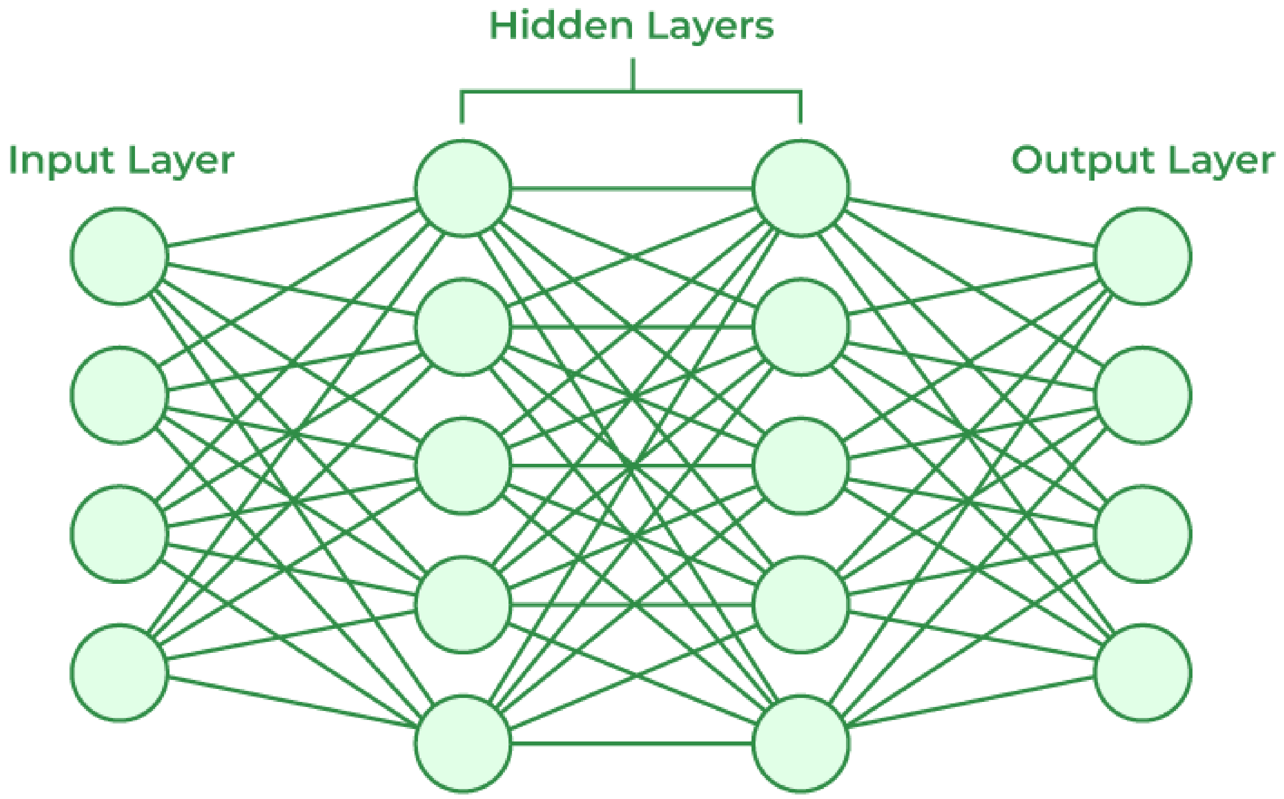
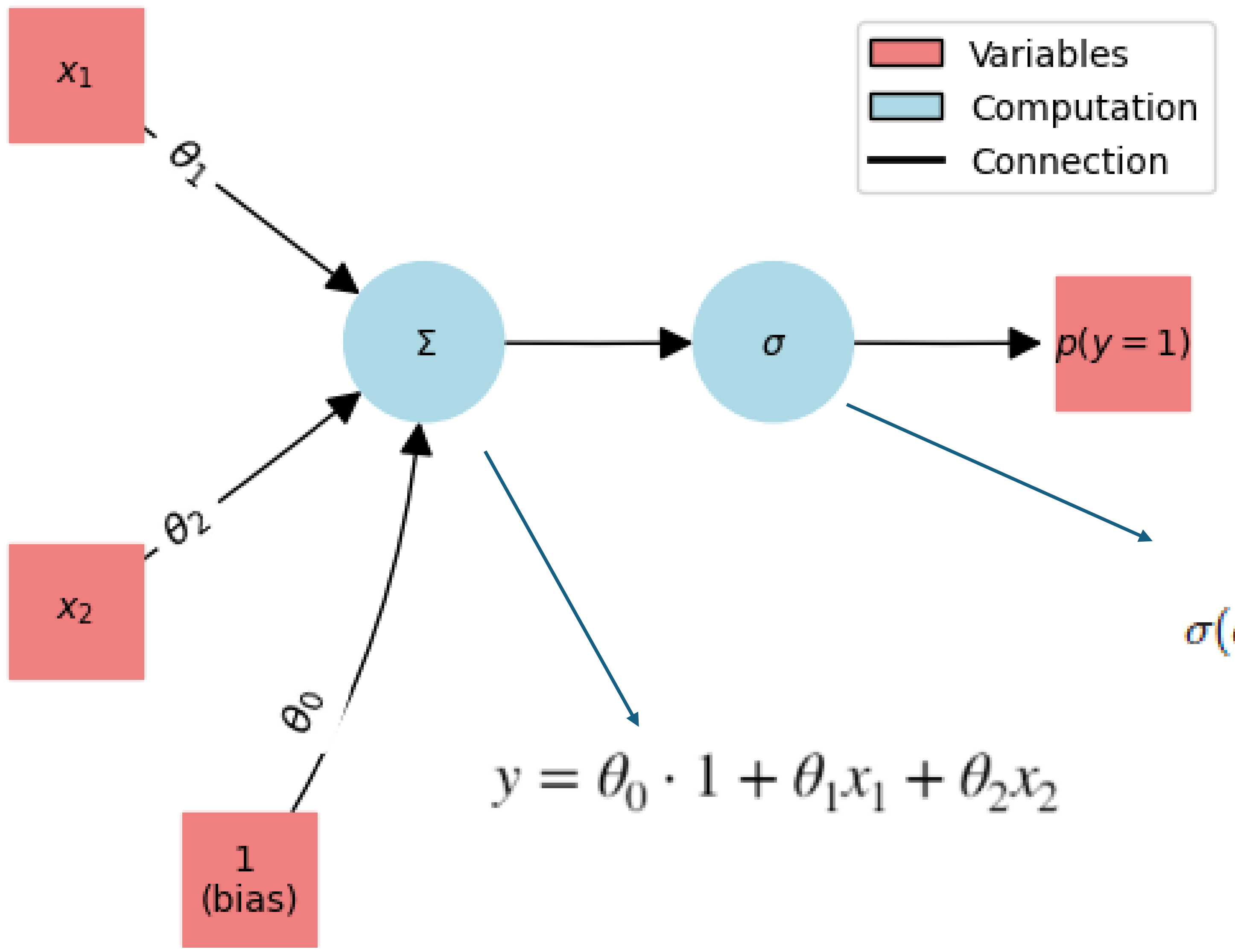
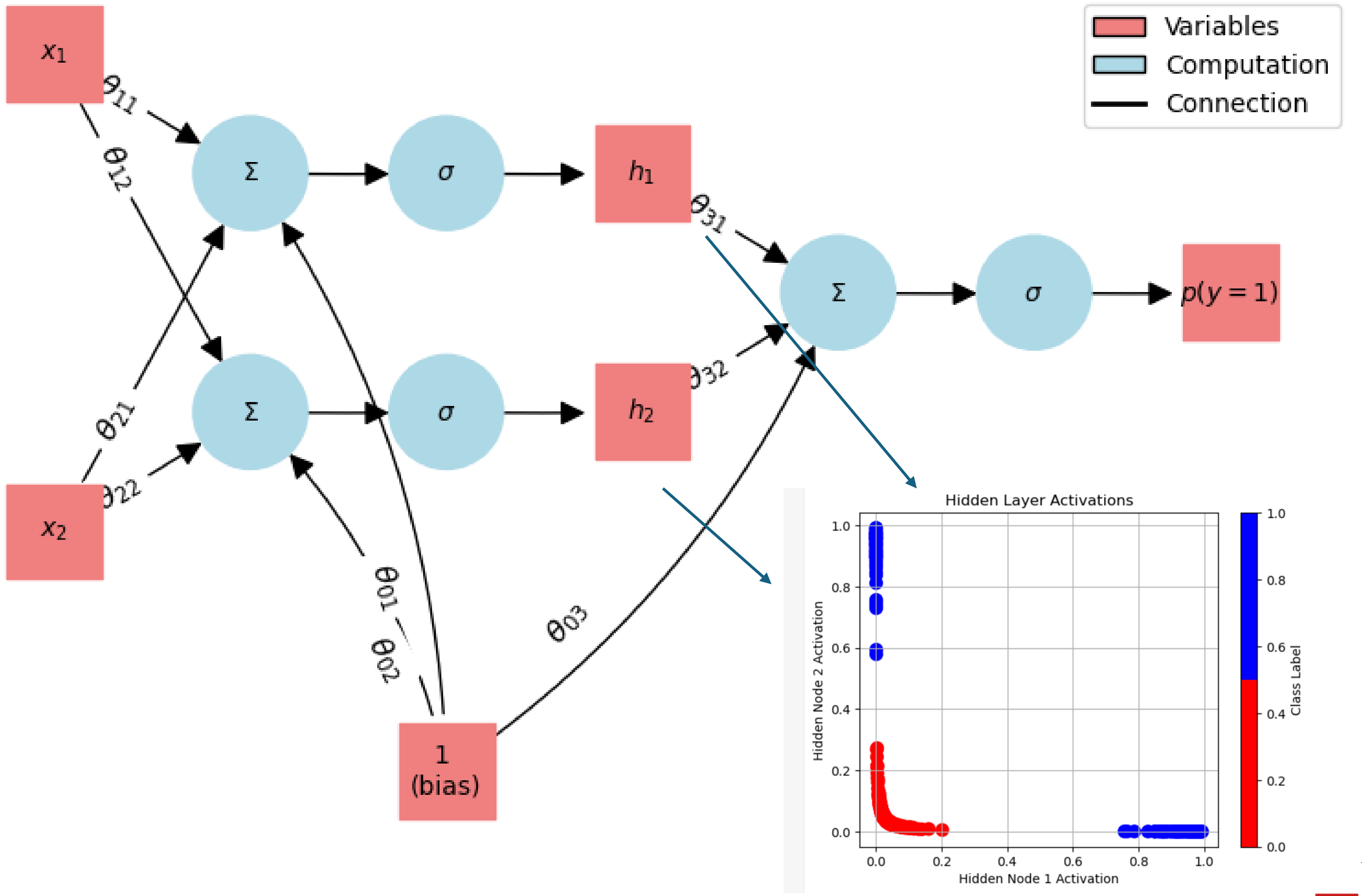
Neural Networks part 2



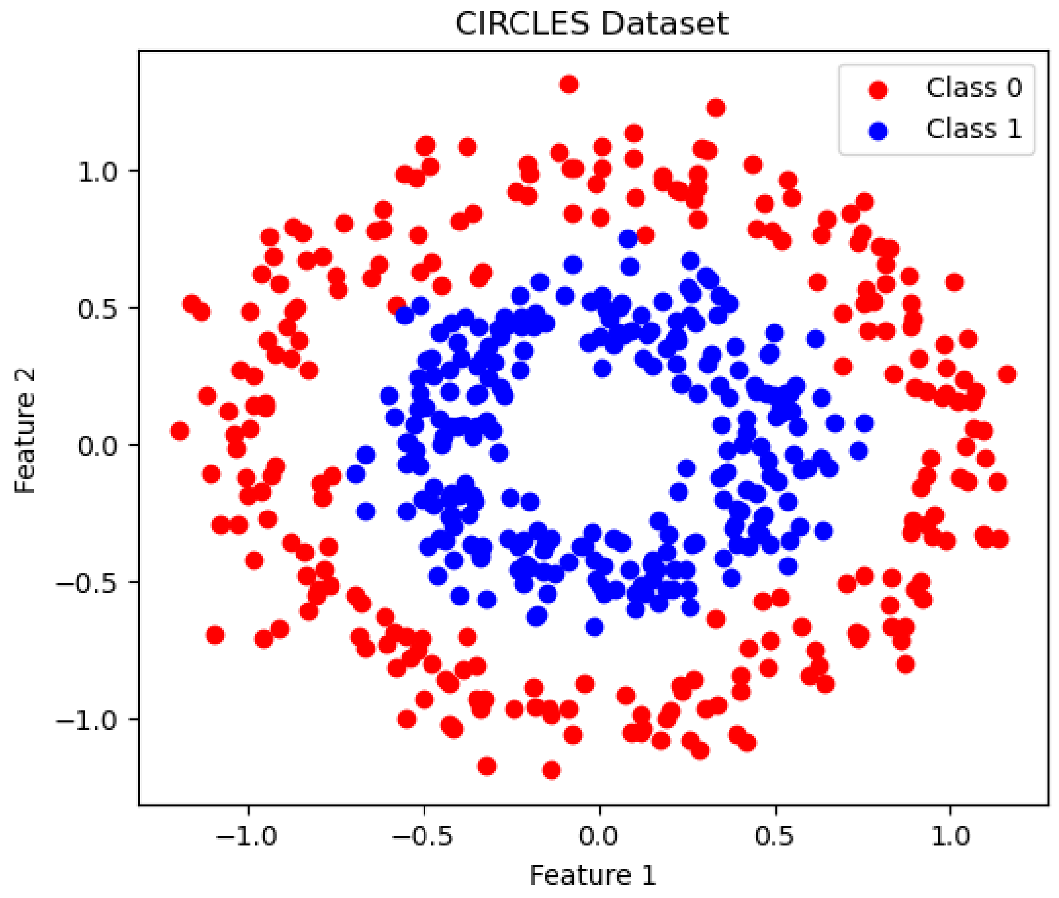


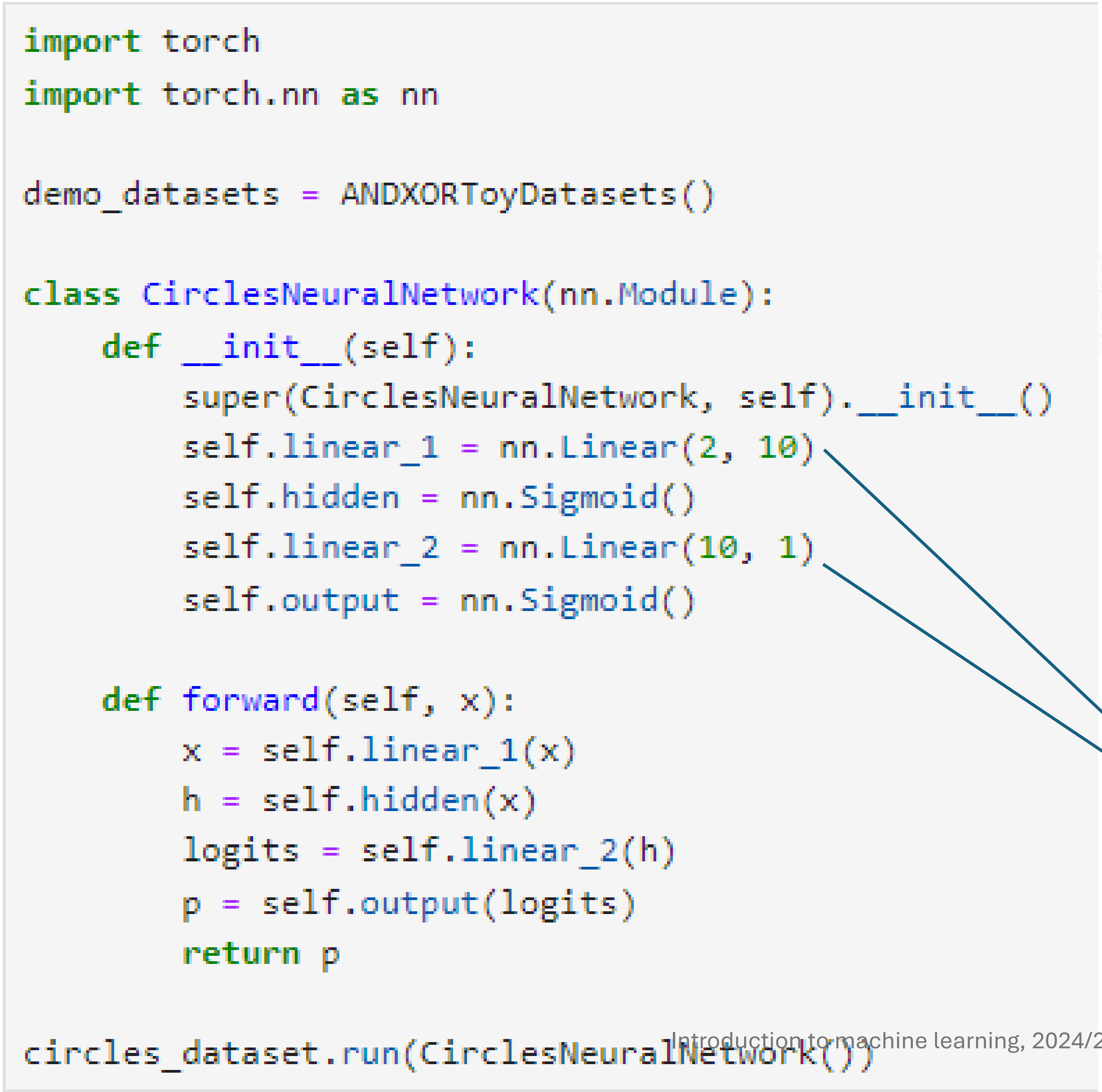
$$y = \theta_0 \cdot 1 + \theta_1 x_1 + \theta_2 x_2$$

$$\sigma(a) = \frac{1}{1 + e^{-a}}$$


x_1
x_2
θ_{11}
θ_{12}
θ_{21}
θ_{22}
Σ
σ
h_1
h_2
θ_{31}
θ_{32}
θ_{01}
θ_{02}
θ_{03}
1 (bias)
$p(y=1)$
Variables
Computation
Connection
Hidden Layer Activations
Hidden Node 1 Activation
Hidden Node 2 Activation
Class Label
0.0
0.2
0.4
0.6
0.8
1.0

Circles data





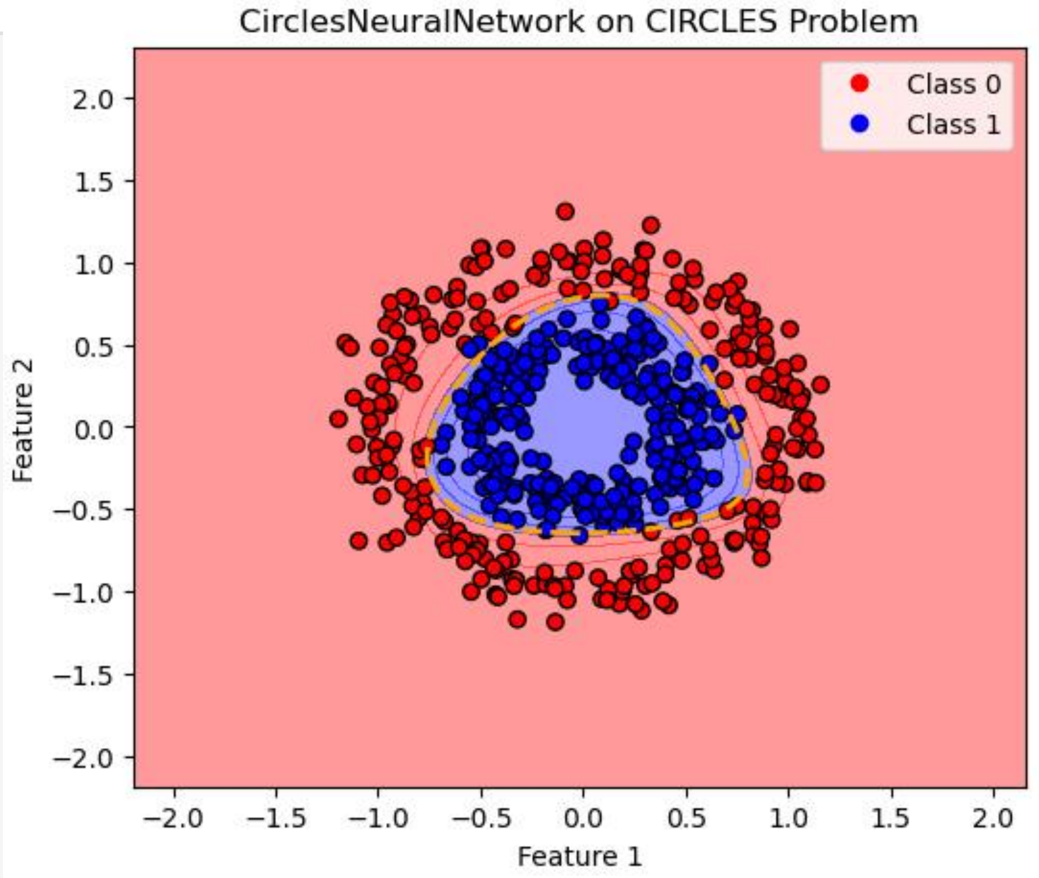
```
import torch
import torch.nn as nn

demo_datasets = ANDXORToyDatasets()

class CirclesNeuralNetwork(nn.Module):
    def __init__(self):
        super(CirclesNeuralNetwork, self).__init__()
        self.linear_1 = nn.Linear(2, 10)
        self.hidden = nn.Sigmoid()
        self.linear_2 = nn.Linear(10, 1)
        self.output = nn.Sigmoid()

    def forward(self, x):
        x = self.linear_1(x)
        h = self.hidden(x)
        logits = self.linear_2(h)
        p = self.output(logits)
        return p

circles_dataset.run(CirclesNeuralNetwork())
```



More neurons in hidden layers!

What challenges did you encounter when designing neural networks for these datasets compared to simpler datasets like XOR?

In simpler datasets we can visually see how we would draw (linear) lines to separate the data, even only two lines with XOR are relatively easy to see. This helps us understand what kind of "minimal" architecture would be necessary to draw these lines. However, in a more complex dataset, it becomes more difficult to see how many lines we would need or it is even impossible to do!

Perhaps certain data transformations are necessary (this happens in the hidden layers!); however, this is almost impossible to see. It is also very difficult to initially make a good guess of what type of minimal model architecture is necessary for learning all the complexities.

How did the complexity of the datasets influence your choice of network architecture and training parameters?

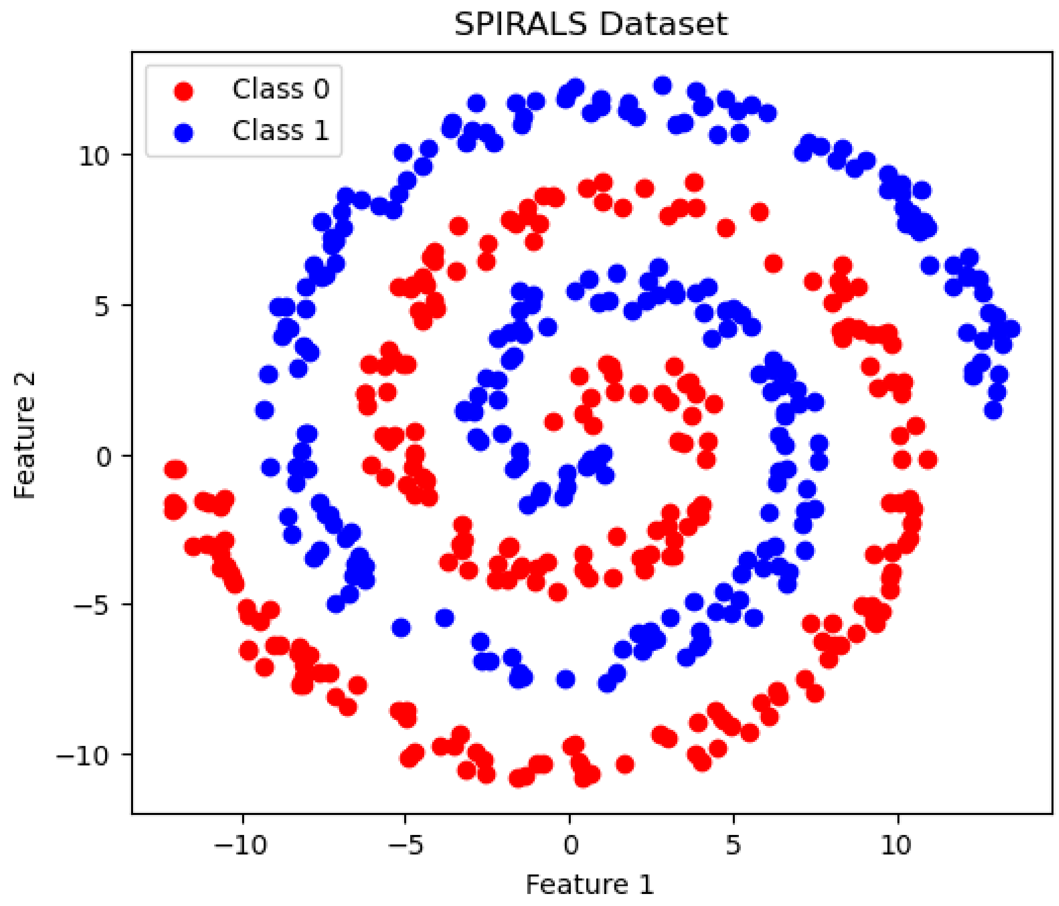
Generally speaking, more complexity would need more complex network architecture, which also means more trainable parameters. When looking at a dataset it is good to try and see how many linear lines you would need to separate classes. If this is doable, we can try a simple network. If this is almost impossible (like spirals dataset) this indicates that there is a lot of complexity in the dataset and we would thus need a more complex architecture to learn all the relations in the data.

How do depth & width influence model’s ability to learn complex patterns?

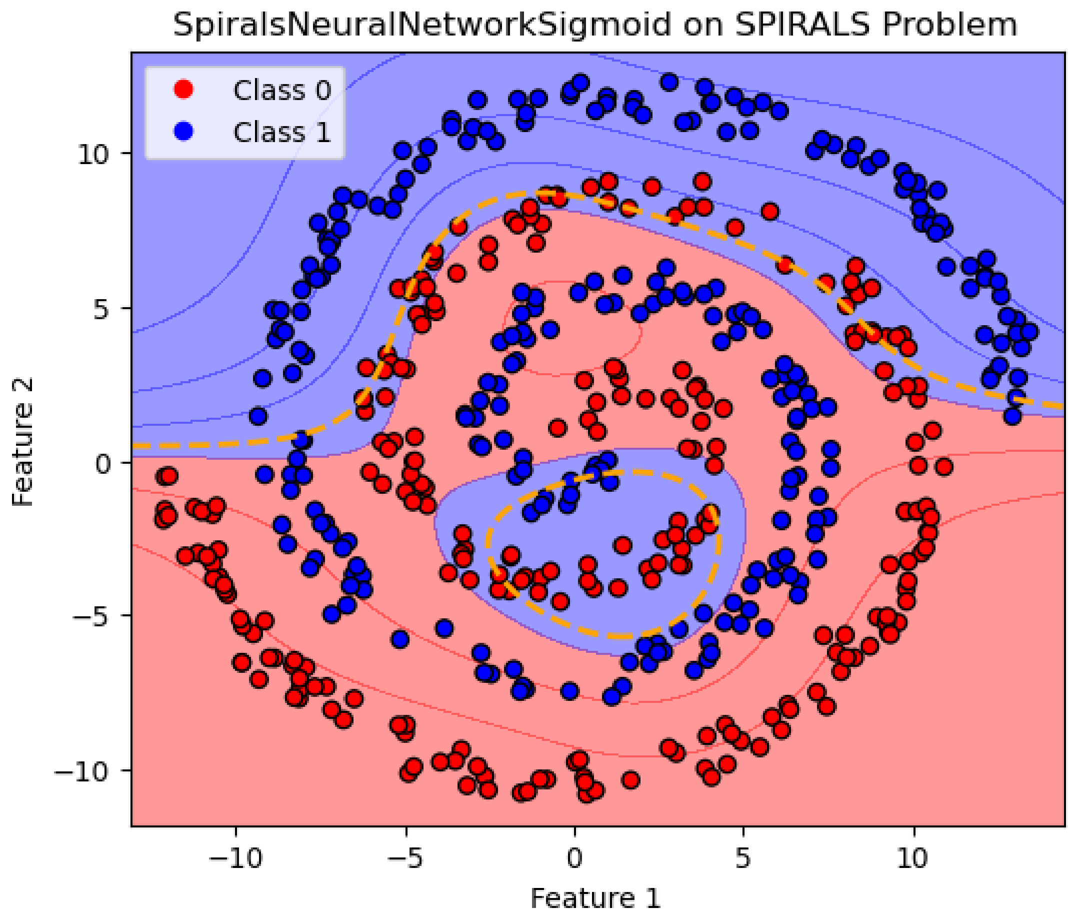
Depth means adding more layers with hidden activations. Generally speaking, deeper networks can transform the input features in more ways and thus find more complex dynamics (such as spirals dataset). Width means adding more neurons within a layer, meaning every layer can capture more detail and complexity within the layer (think of XOR where we needed two lines to make the problem linearly separable, we did this by adding)

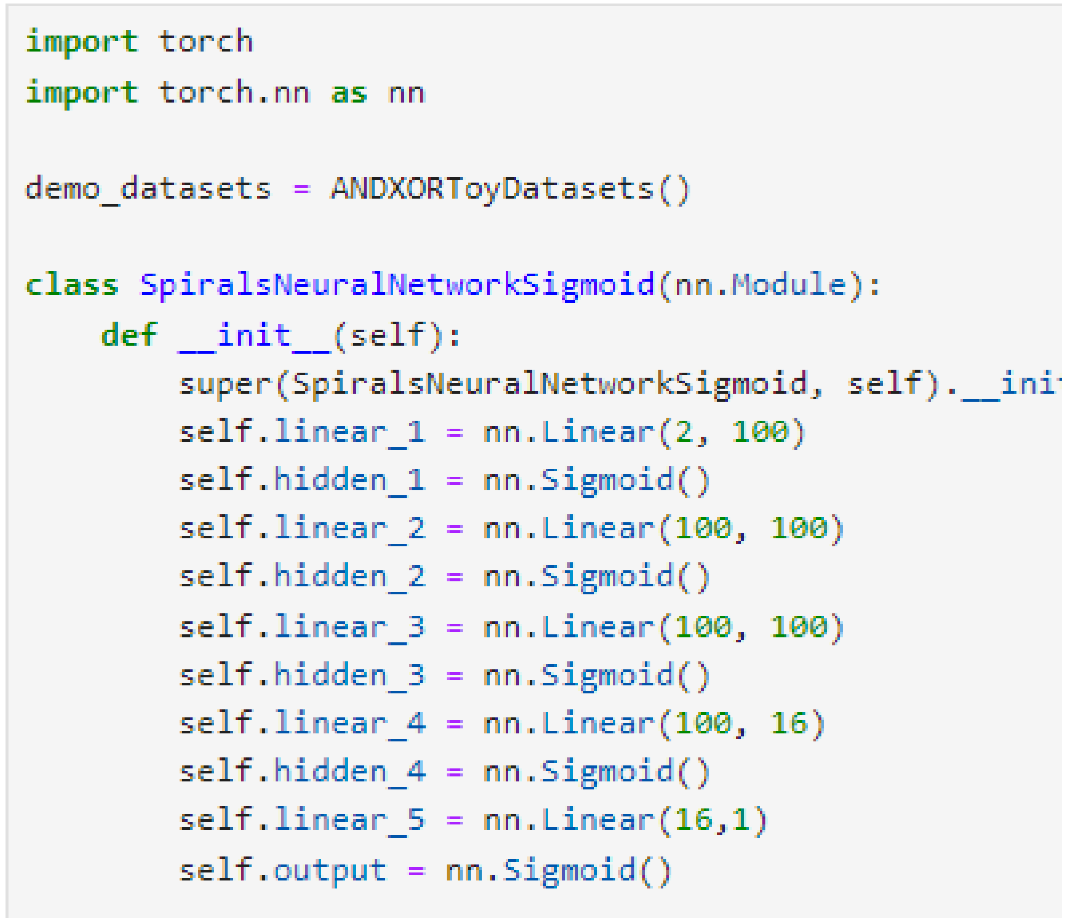
Width & Depth both influence the model’s capacity to learn the complexities of the dataset and do not selectively influence one thing the other does not. This makes designing a model architecture from scratch when looking at data very difficult and a bit of trial/error work.

Spirals data



Vanishing Gradient!



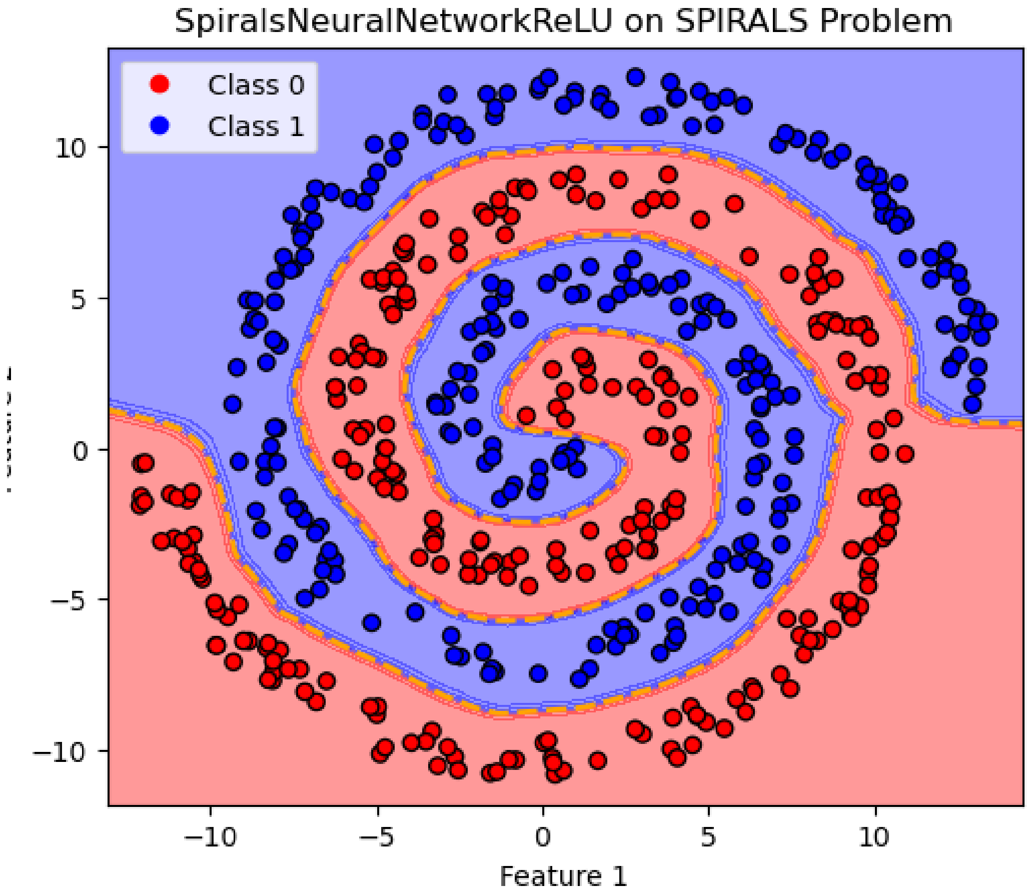


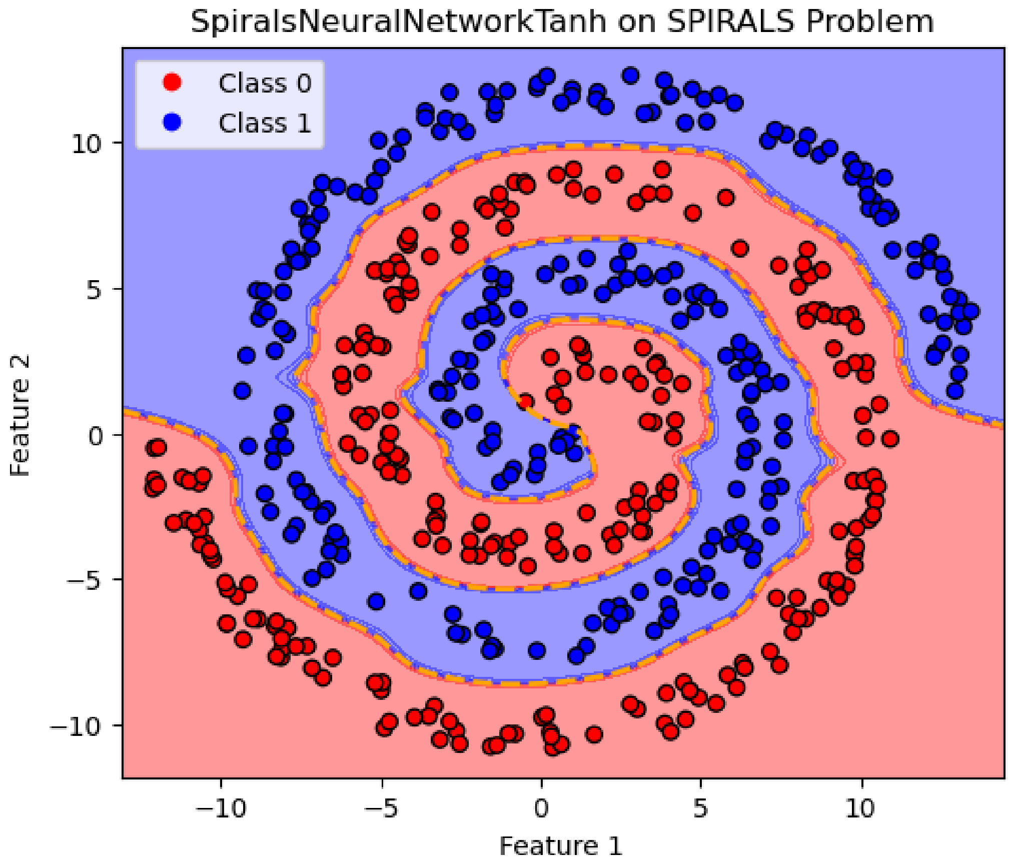
```
import torch
import torch.nn as nn

demo_datasets = ANDXORToyDatasets()

class SpiralsNeuralNetworkSigmoid(nn.Module):
    def __init__(self):
        super(SpiralsNeuralNetworkSigmoid, self).__ini
        self.linear_1 = nn.Linear(2, 100)
        self.hidden_1 = nn.Sigmoid()
        self.linear_2 = nn.Linear(100, 100)
        self.hidden_2 = nn.Sigmoid()
        self.linear_3 = nn.Linear(100, 100)
        self.hidden_3 = nn.Sigmoid()
        self.linear_4 = nn.Linear(100, 16)
        self.hidden_4 = nn.Sigmoid()
        self.linear_5 = nn.Linear(16,1)
        self.output = nn.Sigmoid()
```

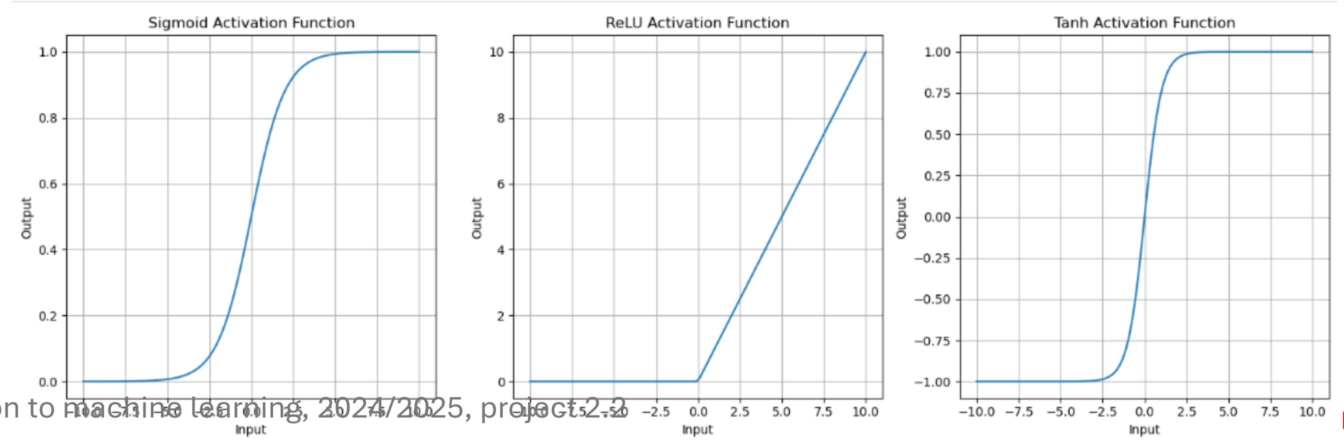
Tanh & ReLU spirals





1. Why is the sigmoid activation function less suitable for deeper neural networks?

Because the sigmoid function in general has low derivatives from output to input (what we calculate during backpropagation). In deep networks this leads to the multiplication of many low numbers, eventually reaching 0. This causes the model parameters to change very little depending on a change in our cost function (what we are minimising/maximising), meaning our model becomes untrainable and is very easily stuck in local minima.



How did changing the activation function to ReLU or Tanh affect the training process and performance of your neural networks?

- Mitigates Vanishing Gradient problem!!

Based on your observations, which activation function would you recommend for deeper networks, and why?

- Tanh and ReLU are more appropriate for deeper networks, especially the ReLU function as Tanh can still suffer from a vanishing gradient. This is due to ReLU holding higher values for its gradients (gradient of output to input).
- You can see that Sigmoid (and tanh) have lower derivatives by looking at what the derivative would be for very high and low values. The line there is flat, meaning the derivative will be close to 0. ReLU has a lot more parts where the gradient is larger/non-zero.
- Tanh being centred around zero is the main factor why it performs better than Sigmoid. This causes the gradients to average to zero, helping with training.

Can ReLU or TanH be used as output nonlinearies? Motivate your answer.

If we are classifying, we want to interpret the model output (y) as a probability. This way, we can do maximum likelihood estimation (MLE) to train the model parameters. For us to be able to interpret the output as probability, the output of the last hidden layer must be between 0 & 1. ReLU and TanH do not have an output between 0 & 1 so we would generally not want to use them as output non-linearity.

What will happen if you remove all nonlinearities from the model, except the output nonlinearity?

Non-linearities (our activation functions like sigmoid) in the model allow the model to learn non-linear relations between input features. The output non-linearity scales the input of the preceding neurons to the output (y, which if it is a probability we want between 0 and 1), but does not aid in learning non-linearities in the data (look at linear regression as a NN, it has a non-linearity in the end but does not learn any non-linear dynamics).

So removing all nonlinearities essentially reverts our model back to a linear regression model!

NOTE:

Non-linearity here relates to the fact that the activation functions we use relate input to output in a **non-linear way!** They are not named that because they learn the non-linearities in the data perse (but this is of course the reason why we choose non-linear functions)