a.

I was not able to pass the test_custom_transform case. There were two methods that I tried: 1) using trigonometric functions and 2) a logarithmic spiral. I want to showcase the code that I used for both methods:

1) Trigonometric Functions

```
def custom_transform(data):
    #81.2%
    transformed_data = np.zeros((data.shape[0], 2))
    transformed_data[:, 0] = np.sin(np.pi*data[:, 0])
    transformed_data[:, 1] = np.sin(np.pi*data[:, 1])
    return transformed_data
```

This was my first approach. I honestly was messing around with applying sinusoidal functions ‘sin’, ‘cos’, and ‘tan’ to the data. Truthfully, once I applied a sine wave transform to the data, with a scaling of pi towards each datapoint, this one had the greatest accuracy that I have achieved. I am not really sure why this is the case, however, this was only using 2 features which made it even more of an interesting behavior. I tried out many other combinations involving what is mentioned above, from computing distances r = x^2 + y^2 or the angle theta = np.tan2(y/x), but nothing achieved that 90%.

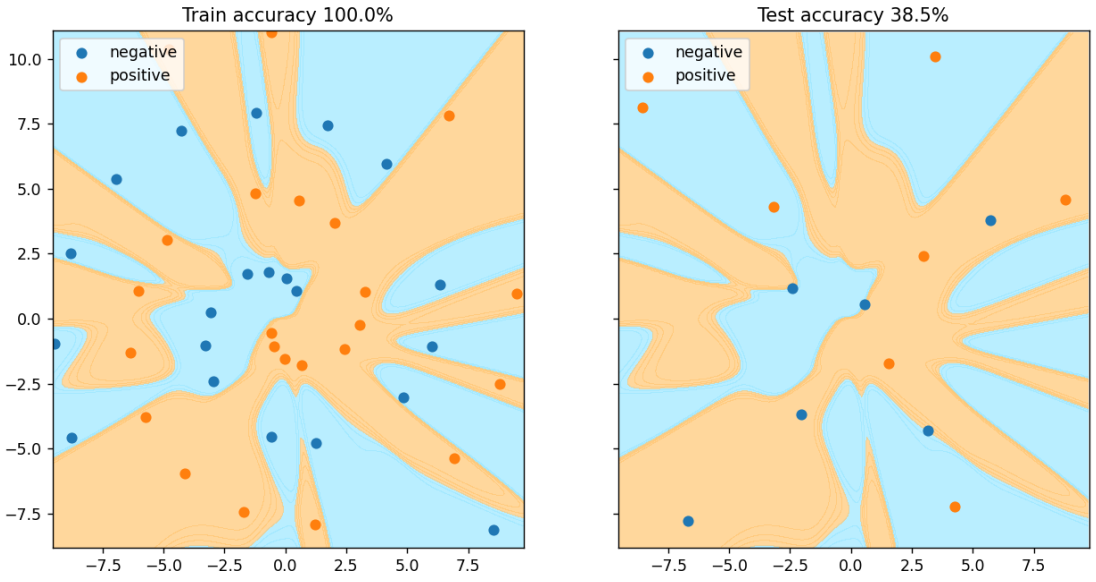
2) Logarithmic Spiral

```
def polar_to_cartesian(r, theta):
    return r * np.cos(theta), r * np.sin(theta)

def logarithmic_spiral(a, b, theta):
    r = a * np.exp(b * theta)
    return polar_to_cartesian(r, (1/b) * np.log(r/a))

def custom_transform(data):
    #48-53%
    transformed_data = np.zeros((data.shape[0], 3))
    a, b = 1, 0.01
    for i in range(data.shape[0]):
        x, y = data[i]
        r, theta = np.sqrt(x**2 + y**2), np.arctan2(y, x)
        r_new, theta_new = logarithmic_spiral(a, b, theta)
        transformed_data[i, 0] = r_new
        transformed_data[i, 1] = theta_new
        transformed_data[i, 2] = abs(x - y)
    return transformed_data
```

In the second approach, using Logarithmic Spirals seemed to be more promising and more logical rather than simply testing out with trial and error. Though, this did not perform as intended. I wasn't able to get 90% accuracy, nor even close to the accuracy from the first approach of 81.2%. As we are working with spirals, it only makes sense to transform the data into the language of spirals: polar. The spirals have an increasing radius given the angle at which the data points find themselves. The data transformed will give us the radius at the given point, the angle, and the difference between the x and y coordinate values as it could help distinguish the spiral arms as they curve in opposite directions. Along with that, I tried **many** different values for a and b in the radius equation for a logarithmic spiral. Nonetheless, even with this more logical approach, I wasn't able to achieve 90% accuracy.

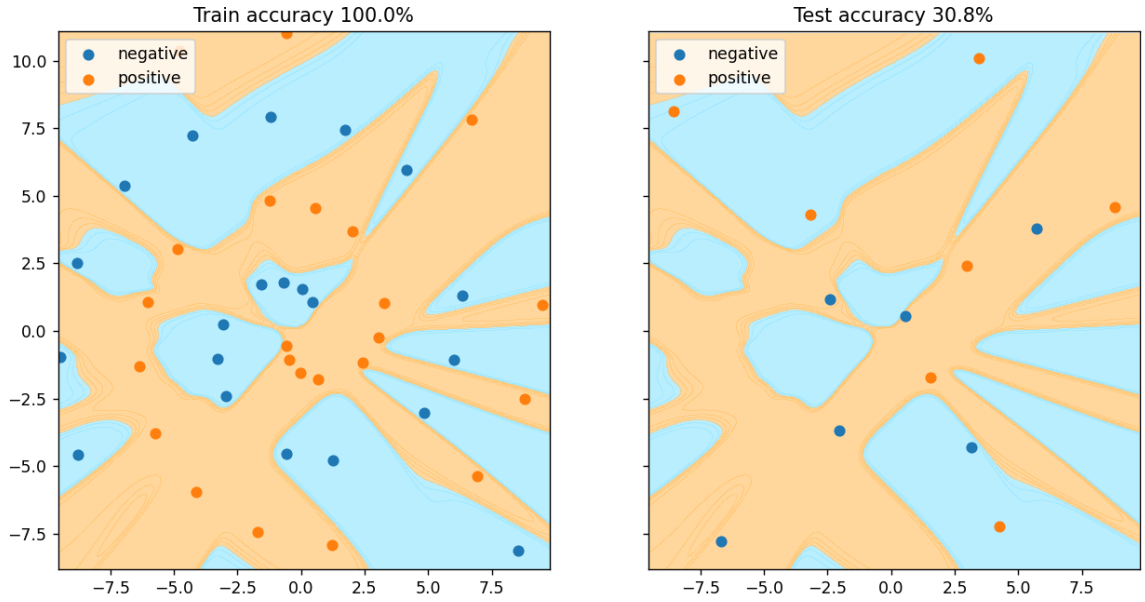
b.

```
# Try changing the layer structure and hyperparameters
    n_iters = 10000
    learning_rate = 1
    reg = Regularizer(alpha=0, penalty="l2")
    layers = [
        FullyConnected(2, 32, regularizer=reg),
        SigmoidActivation(),
        FullyConnected(32, 1, regularizer=reg),
        SigmoidActivation()
    ]
```

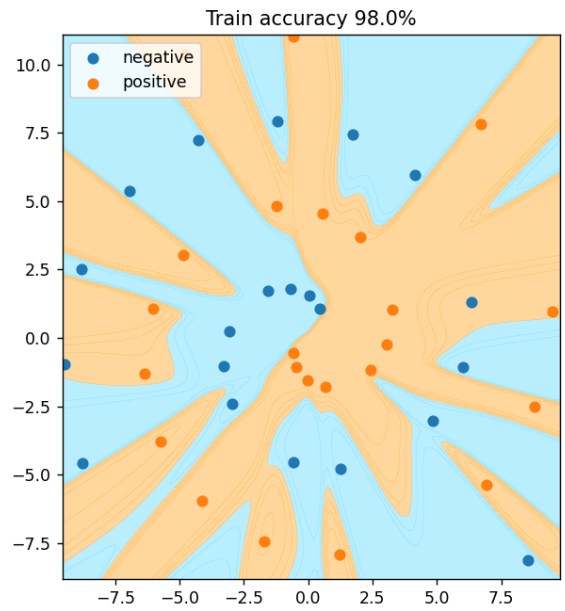


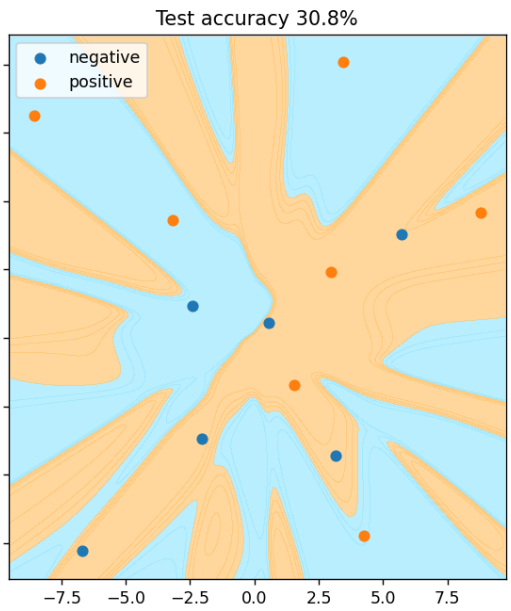

```
    n_iters = 10000
    learning_rate = 2
```

```
reg = Regularizer(alpha=0, penalty="l2")
layers = [
    FullyConnected(2, 32, regularizer=reg),
    SigmoidActivation(),
    FullyConnected(32, 1, regularizer=reg),
    SigmoidActivation()
]
```

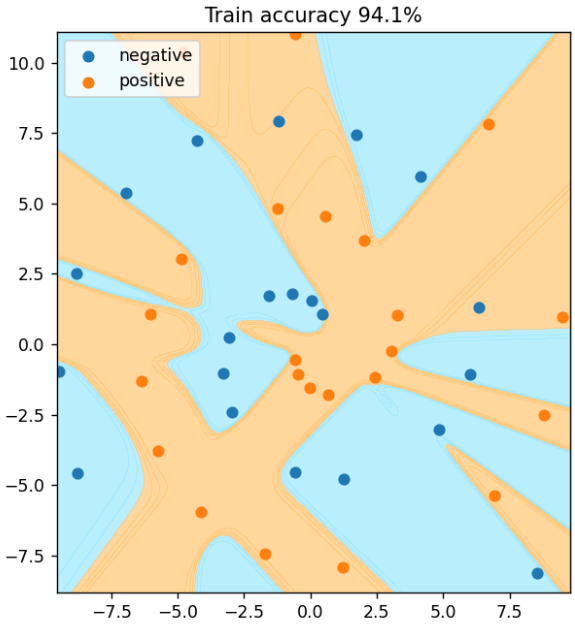


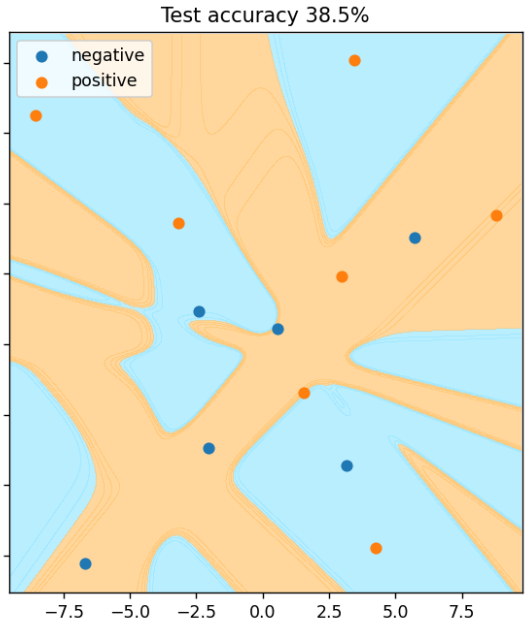

```
n_iters = 10000
learning_rate = 0.5
reg = Regularizer(alpha=0, penalty="l2")
layers = [
    FullyConnected(2, 32, regularizer=reg),
    SigmoidActivation(),
    FullyConnected(32, 1, regularizer=reg),
    SigmoidActivation()
]
```



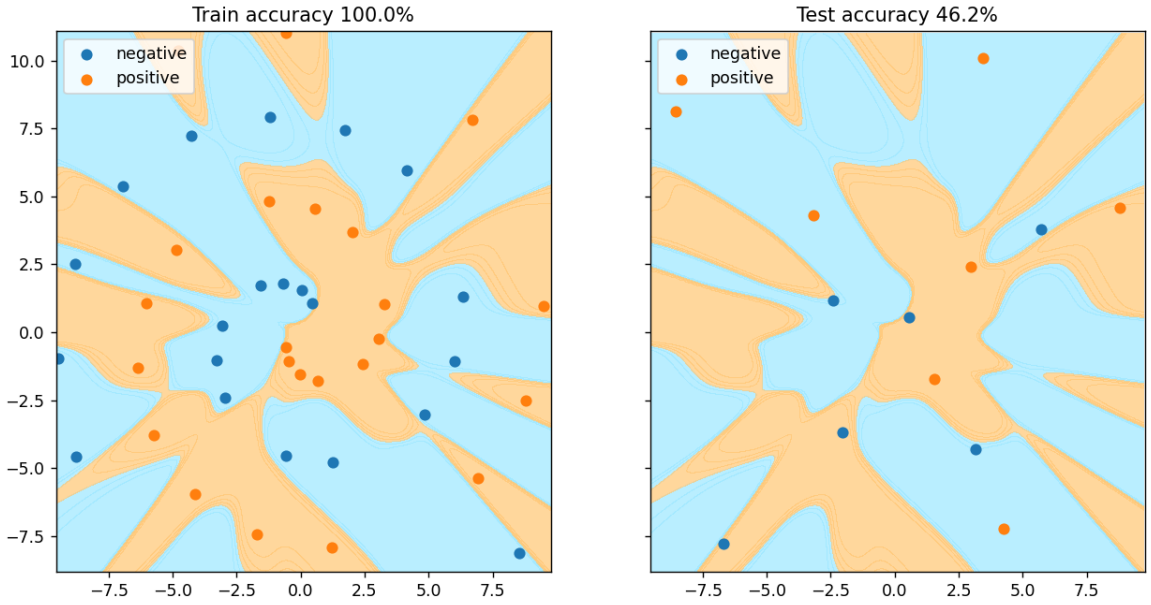



```
n_iters = 10000
learning_rate = 1
reg = Regularizer(alpha=0, penalty="l2")
layers = [
    FullyConnected(2, 16, regularizer=reg),
    SigmoidActivation(),
    FullyConnected(16, 1, regularizer=reg),
    SigmoidActivation()
]
```

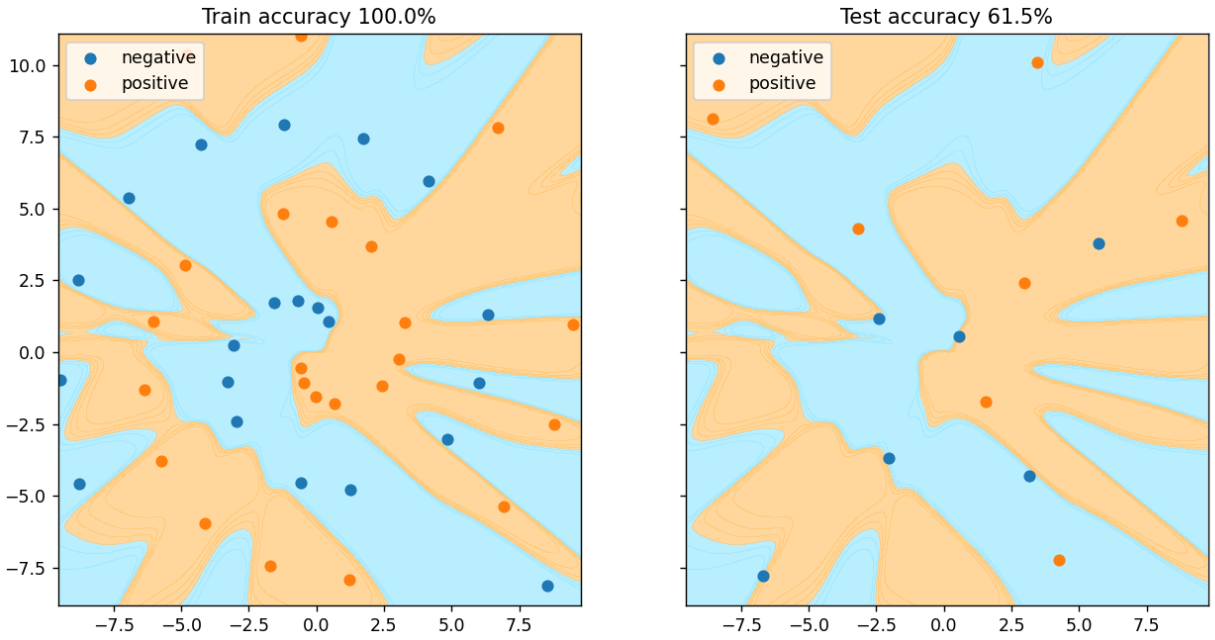




```
n_iters = 10000
learning_rate = 1
reg = Regularizer(alpha=0, penalty="l2")
layers = [
    FullyConnected(2, 64, regularizer=reg),
    SigmoidActivation(),
    FullyConnected(64, 1, regularizer=reg),
    SigmoidActivation()
]
```

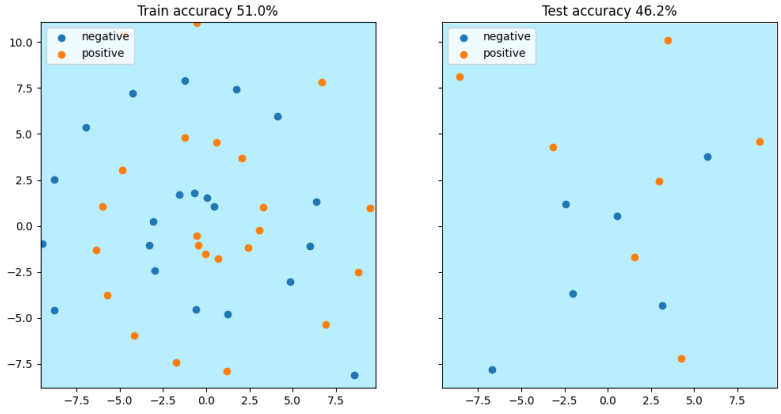



```
n_iters = 10000
learning_rate = 2
reg = Regularizer(alpha=0, penalty="l2")
layers = [
    FullyConnected(2, 64, regularizer=reg),
    SigmoidActivation(),
    FullyConnected(64, 1, regularizer=reg),
    SigmoidActivation()
]
```



Upon experimenting with the parameters for the spiral data, one prominent aspect was for the model to overfit the data, rather than being a continuous “blob” for a decision boundary, propagating from one cluster of one to another, the model would only *highlight* one particular data point whether or not that data point is next to another. That’s at least the trend that I was noticing when one increases the number of nodes within a hidden layer. Of course, tuning the parameters of the learning rate and alpha to accommodate such behavior of the neural network can lead to a more controlled output, to have a more defined shape that can perform well against the testing data.

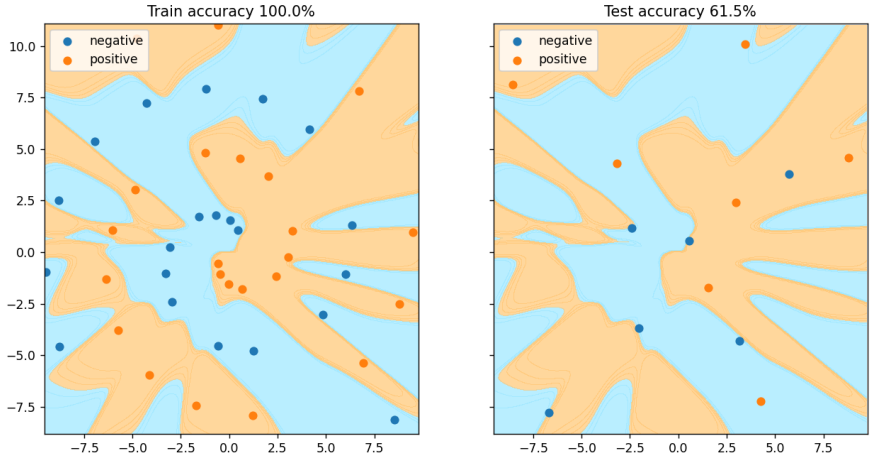
I attempted to use ReluActivation(), however, I was met with an improper graph no matter the settings of parameters:



c.

The following is the best that I was able to tune:

```
n_iters = 10000
learning_rate = 2
reg = Regularizer(alpha=0, penalty="l2")
layers = [
    FullyConnected(2, 64, regularizer=reg),
    SigmoidActivation(),
    FullyConnected(64, 1, regularizer=reg),
    SigmoidActivation()
]
```



I used a Sigmoid activation function with a fully connected neural network of 64, 2, and 1 nodes, utilizing a learning rate of 2 with 10000 example iterations. This graph compared to the other experiments showcases the formation of the spiral shape, which is what I had intended to capture. One is able to see the creation of the ‘hook’ within the spiral. I found that having a decent learning rate as well as a high *enough* amount of nodes within the layers was best to generalize the shape of the decision boundary within the data.