Exercise 1

Task 1:

$y_i = \hat{y}_i \quad z_j = \tilde{z}_j^{(L)}$

$$\frac{\partial L}{\partial \tilde{z}^{(L)}} = \sum_{i=1}^{C} \frac{\partial L}{\partial \hat{y}_i} \cdot \frac{\partial \hat{y}_i}{\partial \tilde{z}_j^{(L)}} \quad \text{(chain rule)}$$

$$L = -\sum_{i=1}^{C} y_i \log(\hat{y}_i) \Rightarrow \frac{\partial L}{\partial \hat{y}_i} = -y_i \cdot \frac{1}{\hat{y}_i}$$

$$\frac{\partial \hat{y}_i}{\partial \tilde{z}_j^{(L)}} = \hat{y}_i (\delta_{ij} - \hat{y}_j)$$

$$\Rightarrow \frac{\partial L}{\partial \tilde{z}^{(L)}} = \sum^{C} \left(-\frac{y_i}{\hat{y}_i}\right) \cdot \hat{y}_i (\delta_{ij} - \hat{y}_j) = \sum^{C} -y_i (\delta_{ij} - \hat{y}_j)$$

$$= -y_j + \hat{y}_j \underbrace{\sum^{C} y_j}_{=1} = \hat{y}_j - y_j$$

Task 2:

a)

$$\frac{\partial L}{\partial W^{(L-1)}} = \frac{\partial L}{\partial \tilde{z}^{(L)}} \cdot \frac{\partial \tilde{z}^{(L)}}{\partial W^{(L-1)}}$$

$$\nabla_{W^{(L-1)}} L = \frac{\partial L}{\partial \tilde{z}^{(L)}} = \hat{y} - y := \delta^{(L)} \in \mathbb{R}^{C}$$

b)

$$\tilde{z}^{(L)} = \sum_{j=1}^{d_{L-1}} W_{ij}^{(L-1)} \cdot z_j^{(L-1)} + b_i^{(L-1)}$$

$$\frac{\partial L}{\partial W_{ij}^{(L-1)}} = \delta_i^{(L)} \cdot z_j^{(L-1)}$$

Task 2

b)

$$\frac{\partial L}{\partial b^{(L-1)}} = \nabla_{b^{(L-1)}} L = \frac{\partial L}{\partial \tilde{z}^{(L)}} \cdot \underbrace{\frac{\partial \tilde{z}^{(L)}}{\partial b^{(L-1)}}}_{=1} = \delta^{(L)} = \hat{y} - y$$

Because

$$\tilde{z}^{(L)} = w^{(L-1)} z^{(L-1)} + b^{(L-1)}$$

$$\Rightarrow \tilde{z}_i^{(L)} = \sum_{j=1}^{d_{L-1}} w_{ij}^{(L-1)} z_j^{(L-1)} + b_i^{(L-1)} \quad \text{w: Linear} \qquad \frac{\partial \tilde{z}_i^{(L)}}{\partial b_i^{(L-1)}} = 1$$

Dimensions:

a) $\delta^{(L)} \in \mathbb{R}^C$

$w^{(L-1)} \in \mathbb{R}^{C \times d_{L-1}}$ $\Rightarrow$ because $z^{(L-1)} \in \mathbb{R}^{d_{L-1}}$

b) $b^{(L-1)} \in \mathbb{R}^C$

$\delta^{(L)} \in \mathbb{R}^C$

Task 3 $\delta^{(L-1)} = ((w^{(L-1)})^T \delta^{(L)}) \odot \phi'(\tilde{z}^{(L-1)})$

$\phi(x) = \max(0, x)$ $\phi'(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{else} \end{cases}$

When $\tilde{z}_j^{(L-1)} < 0$:

$\rightarrow \phi'(\tilde{z}^{(L-1)}) = 0$

$\rightarrow \delta_j^{(L-1)} = 0$

$\Rightarrow$ If the neuron does not activate, then it does not contribute to the learning

Ex 2

1.

(a) $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$.

let $u(x) = e^x - e^{-x}$, $v(x) = e^x + e^{-x}$

$u'(x) = e^x + e^{-x} = v(x)$ $v'(x) = e^x - e^{-x} = u(x)$

$$\frac{d}{dx}\tanh(x) = \frac{u'(x)v(x) - u(x)v'(x)}{v(x)^2}$$

$$= \frac{v(x)^2 - u(x)^2}{v(x)^2}$$

$$= 1 - \frac{u(x)^2}{v(x)^2}$$

$$= 1 - \tanh^2(x)$$

(b)

$$\frac{\partial L}{\partial \tilde{z}^{(2)}} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial \tilde{z}^{(2)}}$$

$$= \left(\frac{\hat{y} - y}{\hat{y}(1-\hat{y})}\right) \cdot \left(\hat{y}(1-\hat{y})\right)$$

$$= \hat{y} - y$$

MLE25_sheet04

June 1, 2025

1 Machine Learning Essentials SS25 - Exercise Sheet 4

1.1 Instructions

- `TODO`'s indicate where you need to complete the implementations.
- You may use external resources, but write your own solutions.
- Provide concise, but comprehensible comments to explain what your code does.
- Code that's unnecessarily extensive and/or not well commented will not be scored.

1.2 Exercise 2

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_moons
from sklearn.model_selection import train_test_split
from matplotlib.colors import ListedColormap
```

1.2.1 Task 2

[48]:
```
# TODO: Define the needed helper functions
def tanh_prime(x_activated):
    return 1 - np.power(x_activated, 2)

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def binary_cross_entropy(y_hat, y):
    """
    Computes the BCE loss over samples.
    """
    # Hint: Add a small epsilon to y_hat to prevent numerical issues w/ log(0)␣
 ↪issues (that's common practice in these cases)
    epsilon = 1e-8
    return -np.mean(y * np.log(y_hat + epsilon) + (1 - y) * np.log(1 - y_hat +␣
 ↪epsilon))
```

[52]:
```
# MLP Class
class MLP:
```

```
  def __init__(self, layer_dims, initialization_scale=1):
      """
      Initializes the multi-layer perceptron.
      Args:
          layer_dims (list of int): List containing the number of neurons in␣
↪each layer.
                                    [d0, d1, d2] = [2, 10, 1] for the␣
↪exercise.
                                    d0: input dimension
                                    d1: hidden layer dimension
                                    d2: output dimension
          initialization_scale (float): Scaling factor for weight␣
↪initialization (i.e. standard deviation of the normal distribution)
      """
      self.parameters = {}
      self.num_layers = len(layer_dims)

      # Weights are initialized by drawing from a standard normal␣
↪distribution, biases are initialized as zero.
      # For more complex networks, one usually uses techniques like Xavier or␣
↪He initialization. Play around with the initialization_scale parameter to␣
↪see how it affects the training!
      # Layer 0 -> Layer 1
      self.parameters['W0'] = np.random.randn(layer_dims[1], layer_dims[0]) *␣
↪initialization_scale
      self.parameters['b0'] = np.zeros((layer_dims[1], 1))
      # Layer 1 -> Layer 2
      self.parameters['W1'] = np.random.randn(layer_dims[2], layer_dims[1]) *␣
↪initialization_scale
      self.parameters['b1'] = np.zeros((layer_dims[2], 1))

      self.cache = {} # For storing intermediate values (needed for backprop)

  def forward(self, X_batch):
      """
      Forward pass through the network. Store intermediate values in self.
↪cache for backward pass.
      """
      # TODO: Implement the forward pass & store the needed values in self.
↪cache
      z_1 = np.tanh(np.dot(self.parameters['W0'], X_batch.T) + self.
↪parameters['b0'])
      y_hat = sigmoid(np.dot(self.parameters['W1'], z_1) + self.
↪parameters['b1'])
      y_hat = y_hat.T
      self.cache['X_batch'] = X_batch
```

```
        self.cache['z_1'] = z_1
        self.cache['y_hat'] = y_hat

        return y_hat

    def backward(self, Y_batch):
        """
        Performs the backward pass (= backpropagation) to compute gradients of␣
↪the loss with respect to the parameters.
        Gradients are stored in the grads dictionary (see update_params method).
        """
        # TODO: Implement the backward pass
        num_sam = Y_batch.shape[0]
        X_batch = self.cache['X_batch']
        z_1 = self.cache['z_1']
        y_hat = self.cache['y_hat']

        del2 = y_hat - Y_batch
        dW1 = (1 / num_sam) * np.dot(del2.T, z_1.T)
        db1 = (1 / num_sam) * np.sum(del2, axis=0, keepdims=True).T

        del1 = np.dot(self.parameters['W1'].T, del2.T) * tanh_prime(z_1)
        dW0 = (1 / num_sam) * np.dot(del1, X_batch)
        db0 = (1 / num_sam) * np.sum(del1, axis=1, keepdims=True)
        grads = {
        'dW0': dW0,
        'db0': db0,
        'dW1': dW1,
        'db1': db1
        }

        return grads

    def update_params(self, grads, learning_rate):
        """
        Updates the parameters using gradient descent.
        Args:
            grads (dict): Dictionary of gradients.
            learning_rate (float): The learning rate.
        """
        self.parameters['W0'] -= learning_rate * grads['dW0']
        self.parameters['b0'] -= learning_rate * grads['db0']
        self.parameters['W1'] -= learning_rate * grads['dW1']
        self.parameters['b1'] -= learning_rate * grads['db1']
```

1.2.2 Task 3

TODO: Explain why using vectorized operations is generally preferred in ML.

Vectorized operations let us process many data points at once, which is much faster than looping through them one by one. This helps speed up training and makes the code more efficient in machine learning.

1.2.3 Task 4

[53]:
```
#Data loading and preprocessing (predefined)
X, y = make_moons(n_samples=500, noise=0.2, random_state=42)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,␣
 ↪random_state=42)


# NOTE: Different libraries/languages use different conventions for the shape␣
 ↪of the data matrix X, which comes from the way they store data in memory:
#   - In ML textbooks/mathematical notation, X is often (n_features,␣
 ↪n_samples), because each sample is a (n_features, 1) column vector and we␣
 ↪stack them "horizontally".
#   - This is consistent with some languages (e.g. Julia, Matlab), which store␣
 ↪data in column-major order.
#   - However, most ML code (e.g. NumPy, sklearn, Pytorch) is optimized for␣
 ↪row-major order, so you will in code most often see data matrices of shape␣
 ↪(n_samples, n_features).
#     (The reason for that is that most of these libraries run C/C++/CUDA code␣
 ↪under the hood, which is optimized for row-major order)
# --> Juggling the shapes of arrays to be correctly aligned with the used model␣
 ↪implementation / convention is a day-to-day task in practical ML and the␣
 ↪cause of many bugs, so always double-check the expected format.

# TODO: The MLP class you're using expects its inputs in a specific shape, make␣
 ↪sure your X and y match that convention.
print(f"Shape of X_train: {X_train.shape}")
print(f"Shape of y_train: {y_train.shape}")
print(f"Shape of X_test: {X_test.shape}")
print(f"Shape of y_test: {y_test.shape}")

y_train = y_train.reshape(-1, 1)
y_test = y_test.reshape(-1, 1)
```

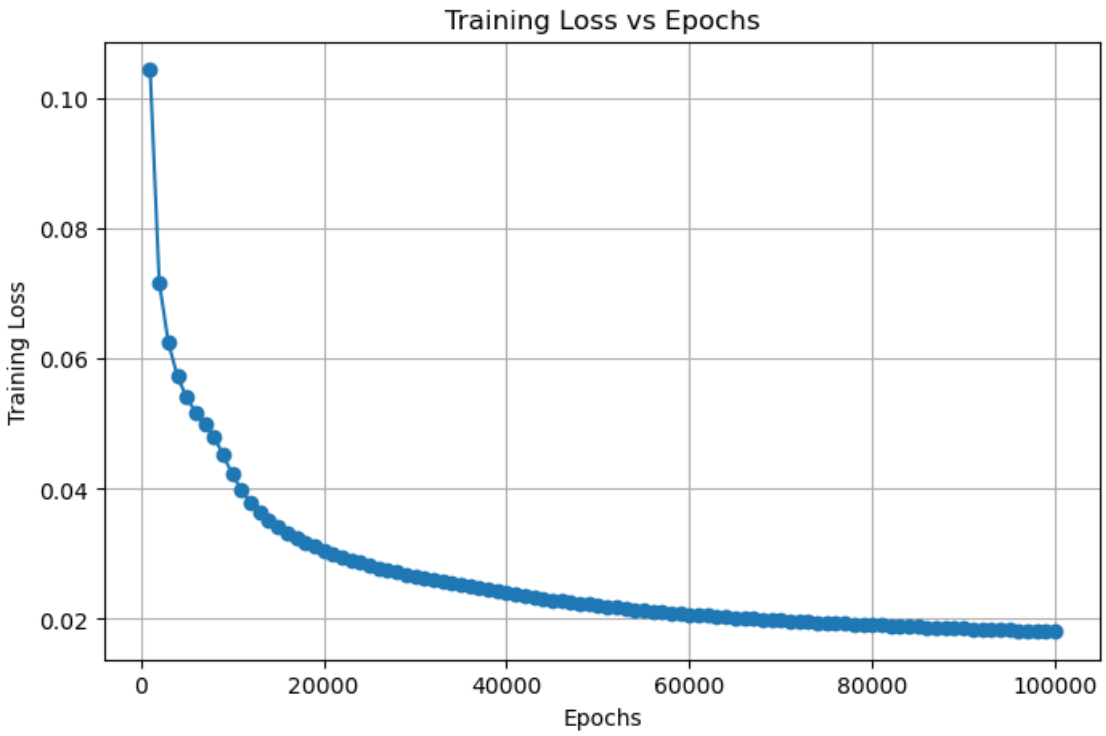
```
Shape of X_train: (350, 2)
Shape of y_train: (350,)
Shape of X_test: (150, 2)
Shape of y_test: (150,)
```

[54]:
```
# Training
layer_dimensions = [X_train.shape[1], 10, 1] # d0, d1, d2 as given in the␣
 ↪exercise
mlp = MLP(layer_dimensions) # Initialize the MLP

# Hyperparameters
learning_rate = 0.3 # TODO: Experiment with this
num_epochs = 100000 # TODO: Experiment with this
print_loss = 1000 # To monitor the training process, print the loss every few␣
 ↪epochs
train_losses = []

for epoch in range(1,num_epochs+1):
    # Forward pass
    y_hat_train = mlp.forward(X_train)
    # Compute loss
    train_loss = binary_cross_entropy(y_hat_train, y_train)
    # Backward pass = backprop
    grads = mlp.backward(y_train)
   # Update parameters by gradient descent
    mlp.update_params(grads, learning_rate)

    if epoch % print_loss == 0 or epoch == num_epochs:
       train_losses.append(train_loss)
       print(f"Epoch {epoch}/{num_epochs} - Training Loss: {train_loss:.4f}")

epochs_recorded = np.arange(1, len(train_losses) + 1) * print_loss

plt.figure(figsize=(8, 5))
plt.plot(epochs_recorded, train_losses, marker='o', linestyle='-')
plt.title('Training Loss vs Epochs')
plt.xlabel('Epochs')
plt.ylabel('Training Loss')
plt.grid(True)
plt.show()
```

```
Epoch 1000/100000 - Training Loss: 0.1045
Epoch 2000/100000 - Training Loss: 0.0718
Epoch 3000/100000 - Training Loss: 0.0625
Epoch 4000/100000 - Training Loss: 0.0573
Epoch 5000/100000 - Training Loss: 0.0540
Epoch 6000/100000 - Training Loss: 0.0518
Epoch 7000/100000 - Training Loss: 0.0500
Epoch 8000/100000 - Training Loss: 0.0480
Epoch 9000/100000 - Training Loss: 0.0452
Epoch 10000/100000 - Training Loss: 0.0422
Epoch 11000/100000 - Training Loss: 0.0398
Epoch 12000/100000 - Training Loss: 0.0379
```

```
Epoch 13000/100000 - Training Loss: 0.0364
Epoch 14000/100000 - Training Loss: 0.0352
Epoch 15000/100000 - Training Loss: 0.0342
Epoch 16000/100000 - Training Loss: 0.0333
Epoch 17000/100000 - Training Loss: 0.0325
Epoch 18000/100000 - Training Loss: 0.0318
Epoch 19000/100000 - Training Loss: 0.0311
Epoch 20000/100000 - Training Loss: 0.0305
Epoch 21000/100000 - Training Loss: 0.0300
Epoch 22000/100000 - Training Loss: 0.0295
Epoch 23000/100000 - Training Loss: 0.0291
Epoch 24000/100000 - Training Loss: 0.0286
Epoch 25000/100000 - Training Loss: 0.0282
Epoch 26000/100000 - Training Loss: 0.0279
Epoch 27000/100000 - Training Loss: 0.0275
Epoch 28000/100000 - Training Loss: 0.0272
Epoch 29000/100000 - Training Loss: 0.0269
Epoch 30000/100000 - Training Loss: 0.0266
Epoch 31000/100000 - Training Loss: 0.0263
Epoch 32000/100000 - Training Loss: 0.0260
Epoch 33000/100000 - Training Loss: 0.0257
Epoch 34000/100000 - Training Loss: 0.0255
Epoch 35000/100000 - Training Loss: 0.0252
Epoch 36000/100000 - Training Loss: 0.0250
Epoch 37000/100000 - Training Loss: 0.0247
Epoch 38000/100000 - Training Loss: 0.0245
Epoch 39000/100000 - Training Loss: 0.0242
Epoch 40000/100000 - Training Loss: 0.0240
Epoch 41000/100000 - Training Loss: 0.0238
Epoch 42000/100000 - Training Loss: 0.0235
Epoch 43000/100000 - Training Loss: 0.0233
Epoch 44000/100000 - Training Loss: 0.0231
Epoch 45000/100000 - Training Loss: 0.0229
Epoch 46000/100000 - Training Loss: 0.0227
Epoch 47000/100000 - Training Loss: 0.0226
Epoch 48000/100000 - Training Loss: 0.0224
Epoch 49000/100000 - Training Loss: 0.0222
Epoch 50000/100000 - Training Loss: 0.0221
Epoch 51000/100000 - Training Loss: 0.0219
Epoch 52000/100000 - Training Loss: 0.0218
Epoch 53000/100000 - Training Loss: 0.0216
Epoch 54000/100000 - Training Loss: 0.0215
Epoch 55000/100000 - Training Loss: 0.0213
Epoch 56000/100000 - Training Loss: 0.0212
Epoch 57000/100000 - Training Loss: 0.0211
Epoch 58000/100000 - Training Loss: 0.0210
Epoch 59000/100000 - Training Loss: 0.0208
Epoch 60000/100000 - Training Loss: 0.0207
```

```
Epoch 61000/100000 - Training Loss: 0.0206
Epoch 62000/100000 - Training Loss: 0.0205
Epoch 63000/100000 - Training Loss: 0.0204
Epoch 64000/100000 - Training Loss: 0.0203
Epoch 65000/100000 - Training Loss: 0.0202
Epoch 66000/100000 - Training Loss: 0.0201
Epoch 67000/100000 - Training Loss: 0.0200
Epoch 68000/100000 - Training Loss: 0.0200
Epoch 69000/100000 - Training Loss: 0.0199
Epoch 70000/100000 - Training Loss: 0.0198
Epoch 71000/100000 - Training Loss: 0.0197
Epoch 72000/100000 - Training Loss: 0.0196
Epoch 73000/100000 - Training Loss: 0.0196
Epoch 74000/100000 - Training Loss: 0.0195
Epoch 75000/100000 - Training Loss: 0.0194
Epoch 76000/100000 - Training Loss: 0.0193
Epoch 77000/100000 - Training Loss: 0.0193
Epoch 78000/100000 - Training Loss: 0.0192
Epoch 79000/100000 - Training Loss: 0.0192
Epoch 80000/100000 - Training Loss: 0.0191
Epoch 81000/100000 - Training Loss: 0.0190
Epoch 82000/100000 - Training Loss: 0.0190
Epoch 83000/100000 - Training Loss: 0.0189
Epoch 84000/100000 - Training Loss: 0.0189
Epoch 85000/100000 - Training Loss: 0.0188
Epoch 86000/100000 - Training Loss: 0.0187
Epoch 87000/100000 - Training Loss: 0.0187
Epoch 88000/100000 - Training Loss: 0.0186
Epoch 89000/100000 - Training Loss: 0.0186
Epoch 90000/100000 - Training Loss: 0.0185
Epoch 91000/100000 - Training Loss: 0.0185
Epoch 92000/100000 - Training Loss: 0.0184
Epoch 93000/100000 - Training Loss: 0.0184
Epoch 94000/100000 - Training Loss: 0.0183
Epoch 95000/100000 - Training Loss: 0.0183
Epoch 96000/100000 - Training Loss: 0.0183
Epoch 97000/100000 - Training Loss: 0.0182
Epoch 98000/100000 - Training Loss: 0.0182
Epoch 99000/100000 - Training Loss: 0.0181
Epoch 100000/100000 - Training Loss: 0.0181
```

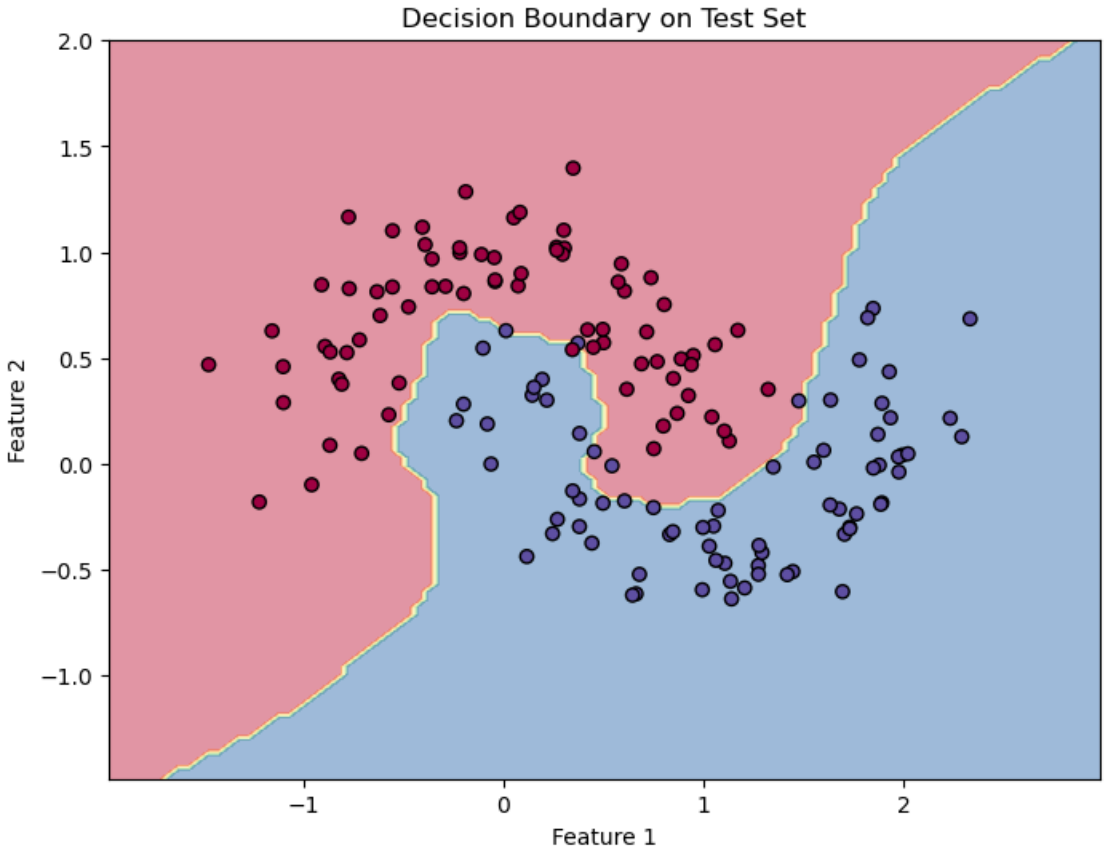


[59]:
```
# Evaluation
# TODO: Compute the accuracy on the test set and plot the decision boundary␣
 ↪over the test set, comment on the performance

y_hat_test = mlp.forward(X_test)

y_pred_test = (y_hat_test >= 0.5).astype(int)
test_accuracy = np.mean(y_pred_test == y_test)
print(f"Test Accuracy: {test_accuracy * 100}%")
```

```
Test Accuracy: 96.0%
```

[62]:
```
xx, yy = np.meshgrid(
    np.linspace(X[:, 0].min() - 0.5, X[:, 0].max() + 0.5, 100),
    np.linspace(X[:, 1].min() - 0.5, X[:, 1].max() + 0.5, 100)
)
grid = np.c_[xx.ravel(), yy.ravel()]

y_hat_grid = mlp.forward(grid)
y_pred_grid = (y_hat_grid >= 0.5).astype(int)
Z = y_pred_grid.reshape(xx.shape)

plt.figure(figsize=(8, 6))
```

```
plt.contourf(xx, yy, Z, cmap=plt.cm.Spectral, alpha=0.5)
plt.scatter(X_test[:, 0], X_test[:, 1], c=y_test.ravel(), edgecolors='k',␣
↪cmap=plt.cm.Spectral)
plt.title('Decision Boundary on Test Set')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.show()
```



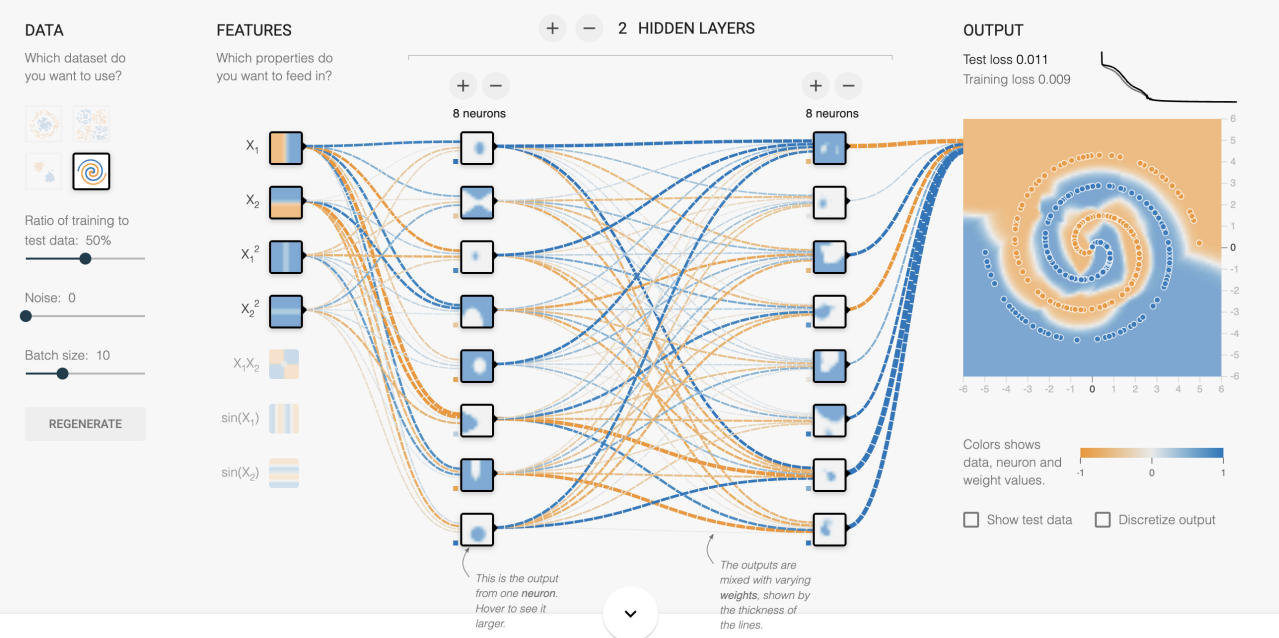
Exercise 3

3.1

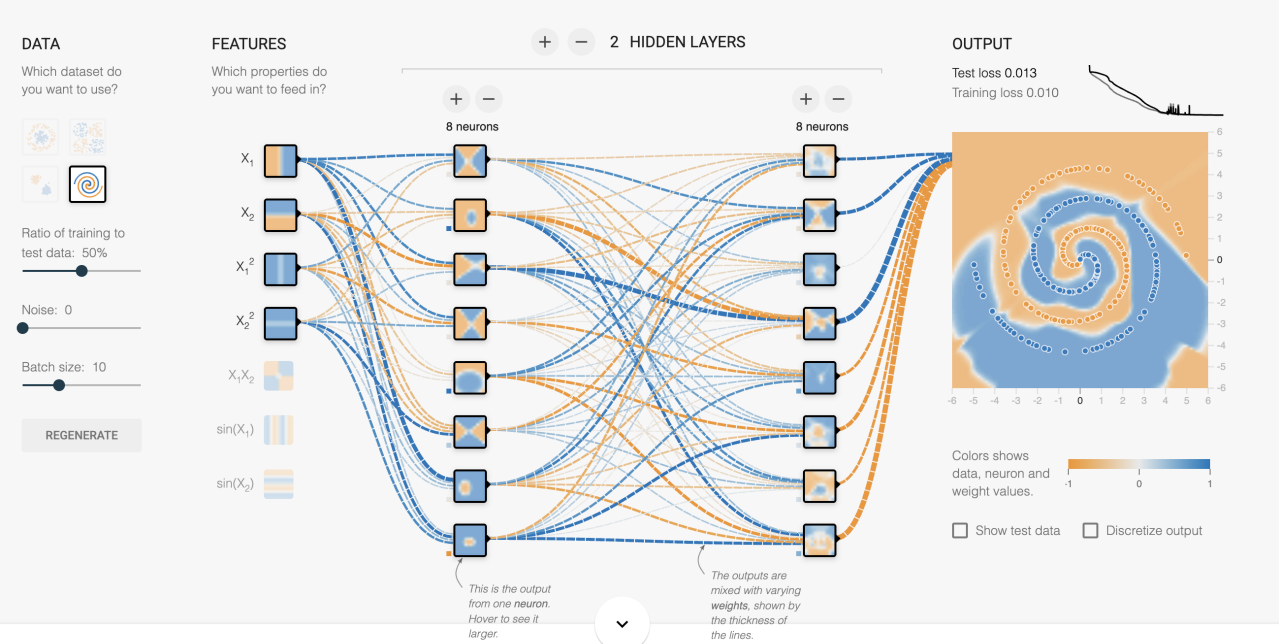
Experiment

For this first problem, I tested all of *ReLU, Tanh, Sigmoid and Linear*. As the **Spiral** dataset cannot be linearly separable, it is also intuitive to think about project the features to higher-dimensional planes by X_1^2, X_2^2 to obtain better results.

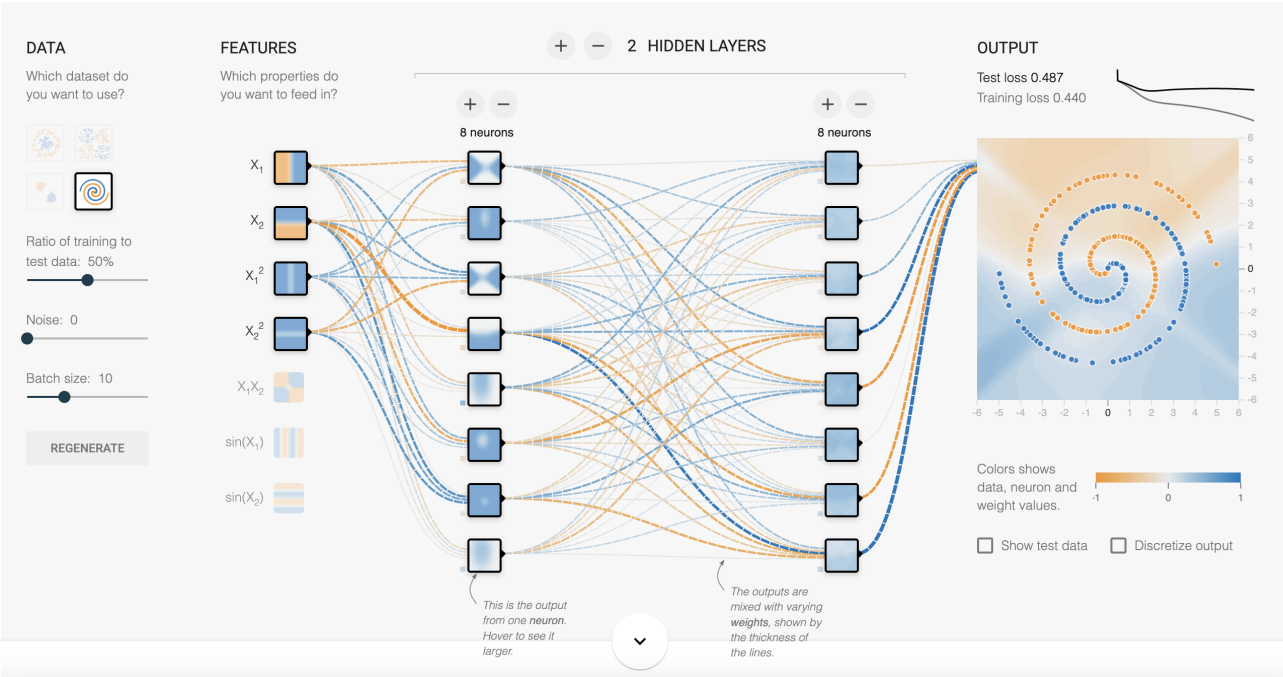
- ReLU w\o Regularization



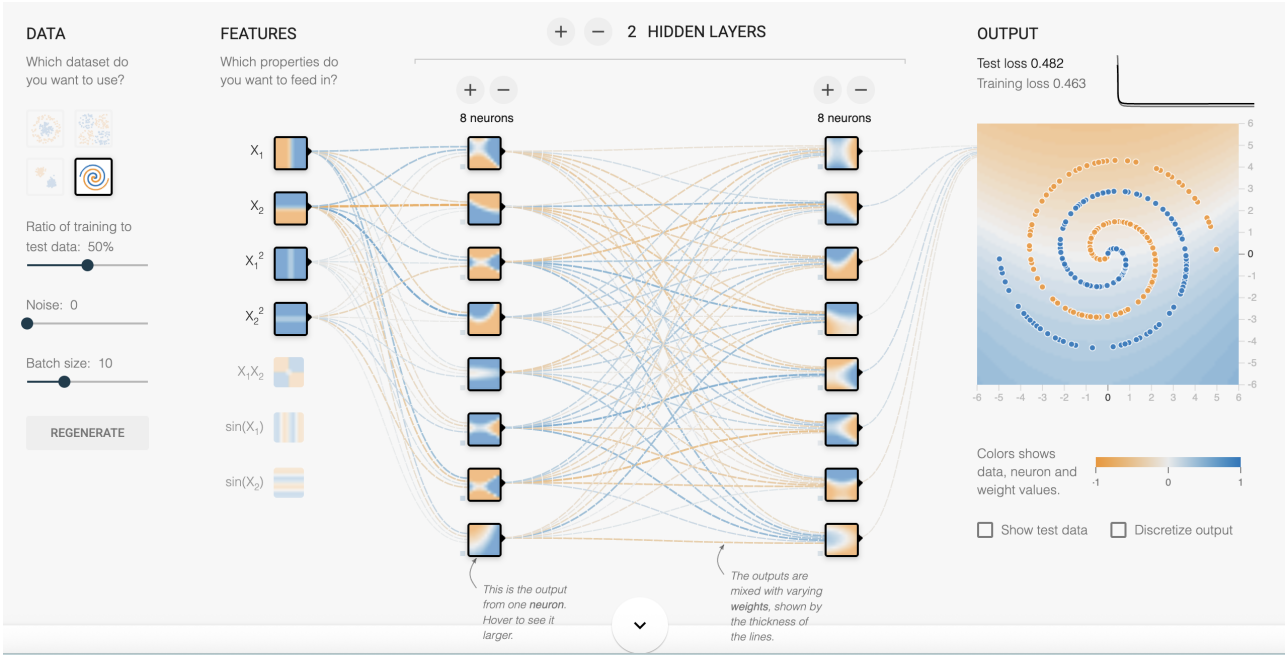
- Tanh w\o Regularization



- Sigmoid w\o Regularization



- Linear w\o Regularization



For this part, I run 1000 epochs on the training dataset for each activation function. It is made apparent from the screenshots above that **ReLU and Tanh** outperform the other activation functions on Spiral dataset.

Insights

Observations on the result:

- With the quaratic input features, the classifier converges faster and gives a smoother decision boundary;
- Under the same setting, *ReLU* converges faster than *Tanh*, whilst *Tanh* has a smoother decision boundary.

Some facts:

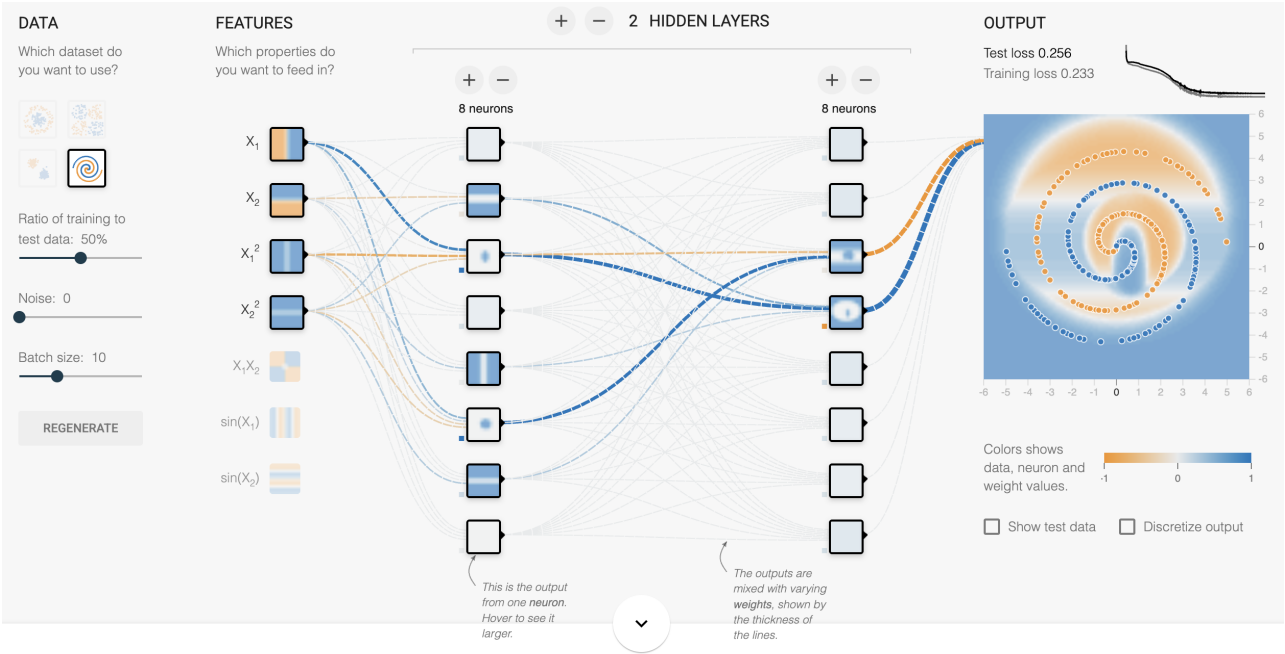
- *ReLU* is not confronted with problems like **Gradient Vanishing** and has a smaller computation overhead, as its expression is quite simple;
- The form of *Tanh* has good symmetric property, thus working well with this **Spiral** dataset.

3.2

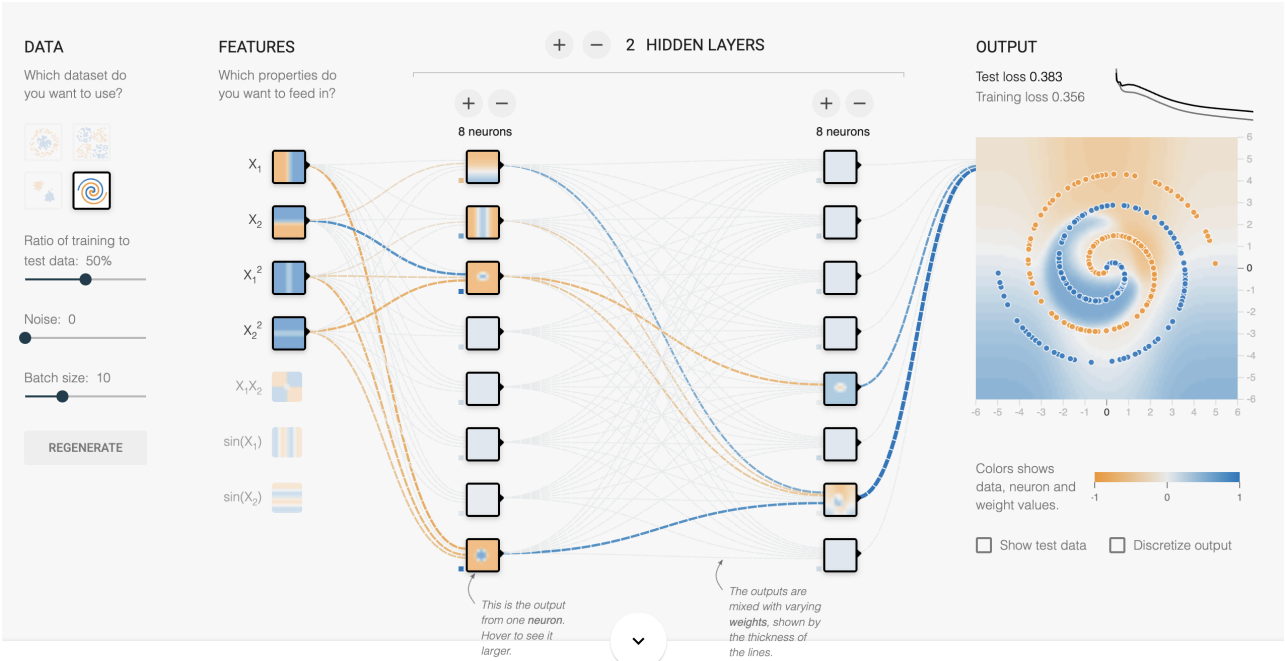
For this part I only test ReLU and Tanh, which win the **3.1 Contest**.

Experiment L1 Regularization

- ReLU



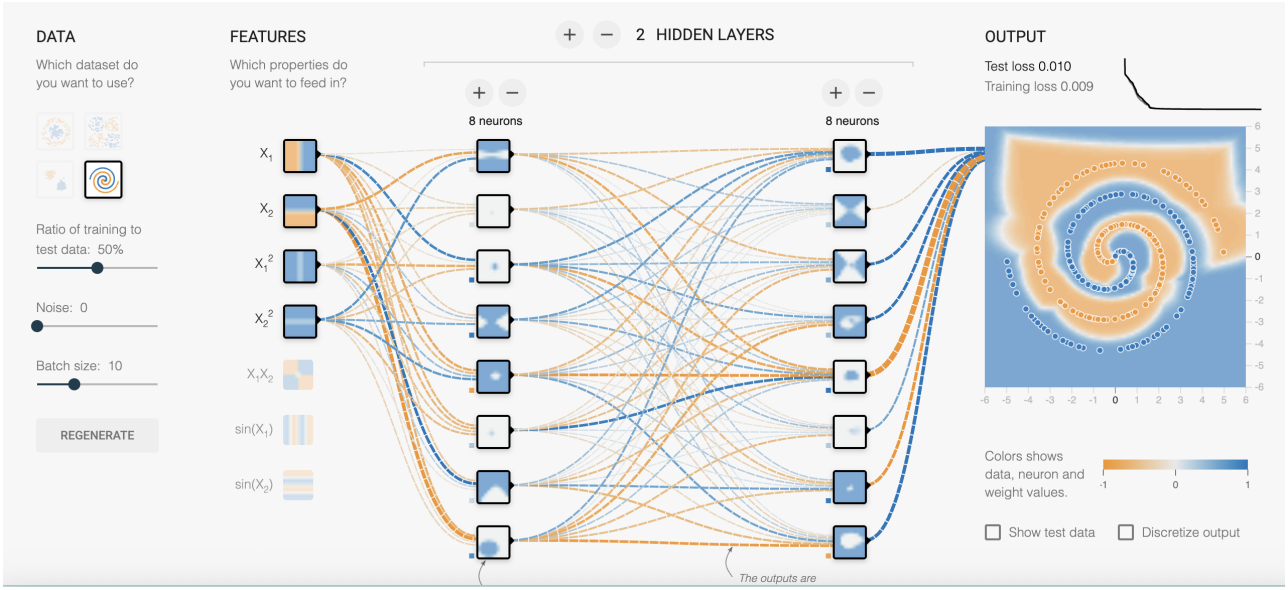
- Tanh



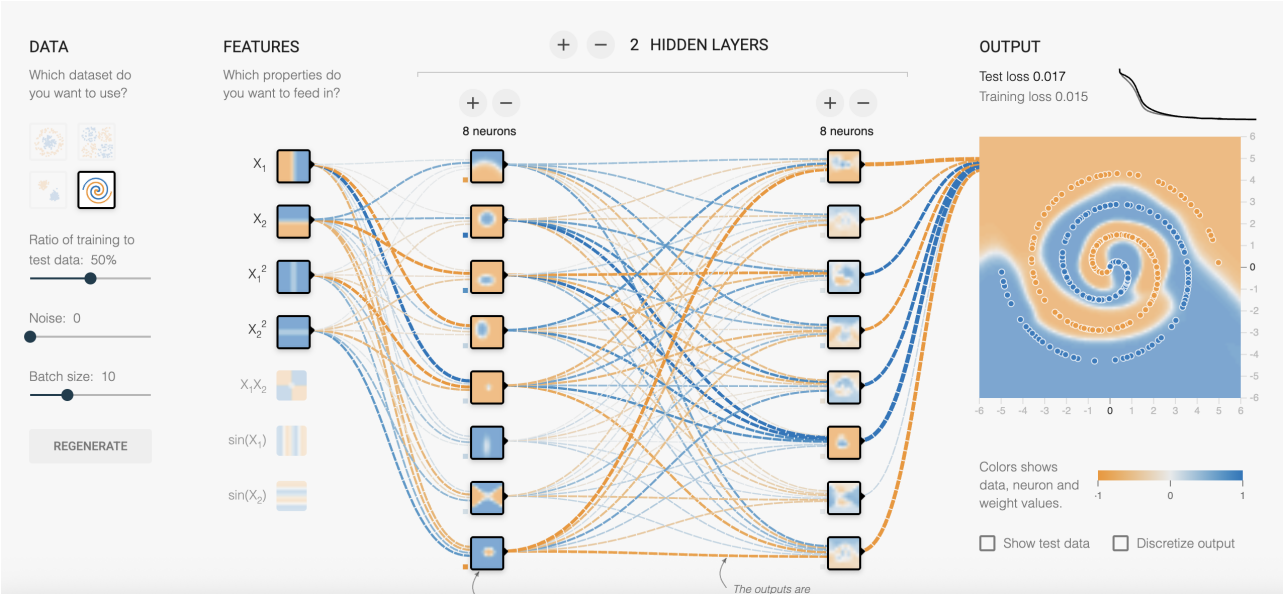
Experiment L2 Regularization

The results of the experiments w\ L2 Regularization are given below:

- ReLU



- Tanh



Insights

1. For L1 Regularization, the promotion of **sparsity** is manifest from the graphs (only a few lines connecting each layers are thick, the others thinner);
2. If we compare the ReLU or Tanh w\o regularization with those w\ regularization, we can note that the former ones give thicker lines overall;
3. For the **Spiral** dataset, the performance of regularization on the loss is approximately: $L_2\approx \text{No Regularization}>>L_1$.

The possible reason for **3.** is that when L1 Regularization is applied, the weights are likely to be more uneven. (As is mentioned in **1.**) However, the curve of our **Spiral** dataset is smooth and symmetric. As a consequence, L1 regularization sees a worse performance in this problem.