Back-Propagation Algorithm Neural Networks

- **Matan Yarin Shimon: 314669342**
- **Or Yitshak: 208936039**
- **Netanel Levine: 312512619**
- **Yahalom Chasid: 208515577**

Dataset:

Our dataset holds 1000 2D points (x, y) and each point has a value that will mark it as v.

x – The value of x is of the form $\frac{m}{100}$ where m is an integer, $-10000 \leq m \leq 10000$.

y – The value of y is of the form $\frac{n}{100}$ where n is an integer, $-10000 \leq n \leq 10000$.

v – The value of v is -1 or 1.

As a reminder in part A, the value of v was determined by:

$$v = f((x,y)) = \begin{cases} 1, & y > 1 \\ -1, & otherwise \end{cases}$$

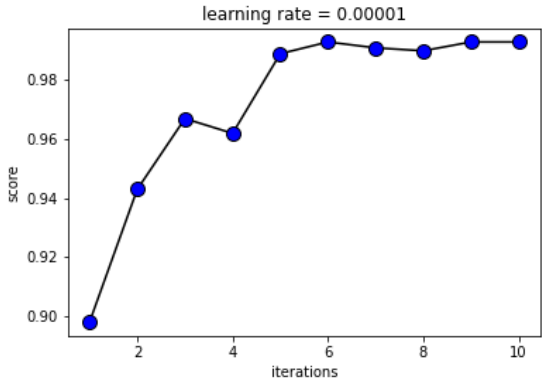
As a reminder in part B, the value of v was determined by:

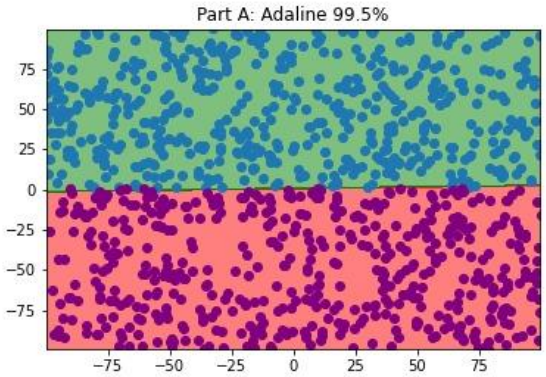
$$v = g((x,y)) = \begin{cases} 1, & 4 \leq x^2 + y^2 \leq 9 \\ -1, & otherwise \end{cases}$$

Part C:

In this part, we will try to improve our scores from parts A and B by using the Back-Propagation Algorithm with the MLP Classifier.

In part A our score was: $\mathbf{99.5}\%$





Explanation of the right scatter:

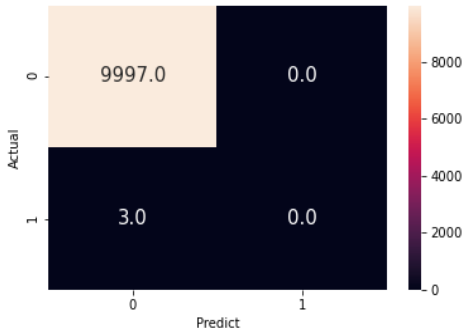
- The **red** area is all the points that their $v = -1$ because $y <= 1$.
- The **green** area is all the points that their $v = 1$ because $y > 1$.
- The **blue** dots mark the points that our model predicted as $v = 1$.
- The **purple** dots mark the points that our model predicted as $v = -1$.

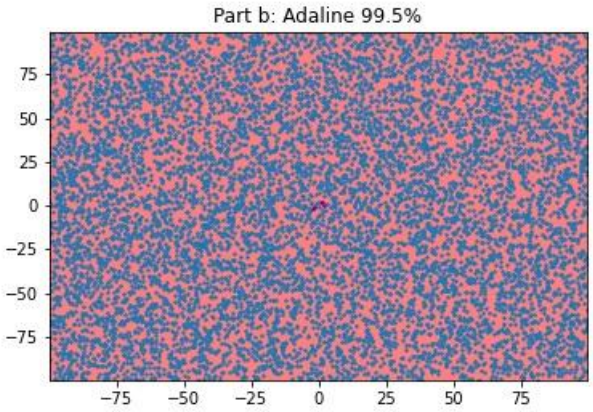
1. Every time our model marked a point as **blue** and this point lies in the range of the **green** area, our model predicted the correct answer.
2. Every time our model marked a point as **purple** and this point lies in the range of the **red** area, our model predicted the correct answer.
3. Every time our model marked a point as **blue** and this point lies in the range of the **red** area, our model predicted the wrong answer.
4. Every time our model marked a point as purple and this point lies in the range of the **green** area, our model predicted the wrong answer.

As we can see, near the line where $y = 0$ the model had a few mistakes.

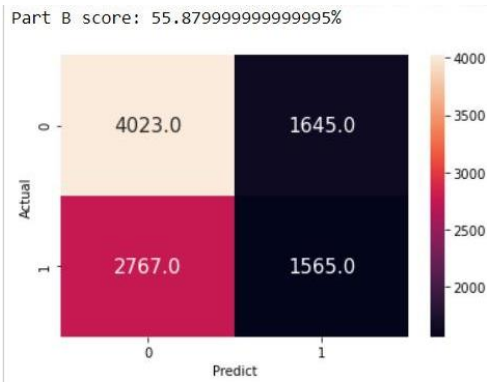
Because our score is almost perfect and Back-Propagation is much more accurate than Adaline we won't perform the Back-Propagation Algorithm on the part A function.

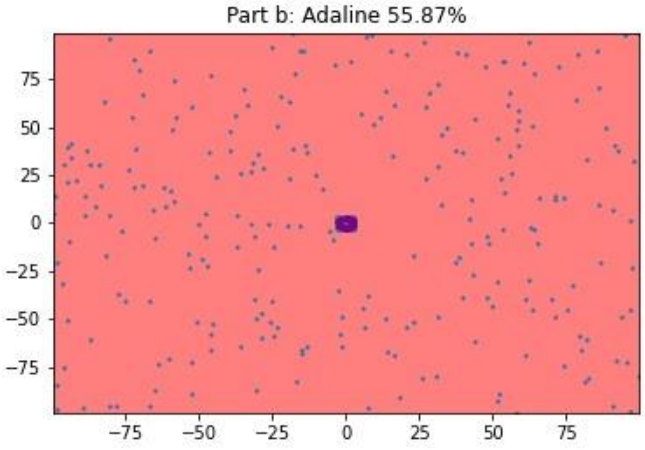
In part B we scored $\mathbf{99.5}\%$ and we thought that this is too good to be true.
We did some digging and we found out that the range of the points that can get a value of 1 is small if we compare it to the range of all the points. This means a high probability of a point ($almost$ 100%) landing in the -1 range which can cause our model to overfit and always predict -1.
As we can see in the confusion matrix below, we got amazing result but the model predict most of the time -1:





So we decided to balance the data by giving our model approximately half instances from each type. After doing so our model accuracy was quite bad $\mathbf{55.87}\%$ with Adaline Algorithm.





Finally, now we will try to improve our model accuracy by using the Back-Propagation Algorithm using the MLP.

Let's recall our problem:

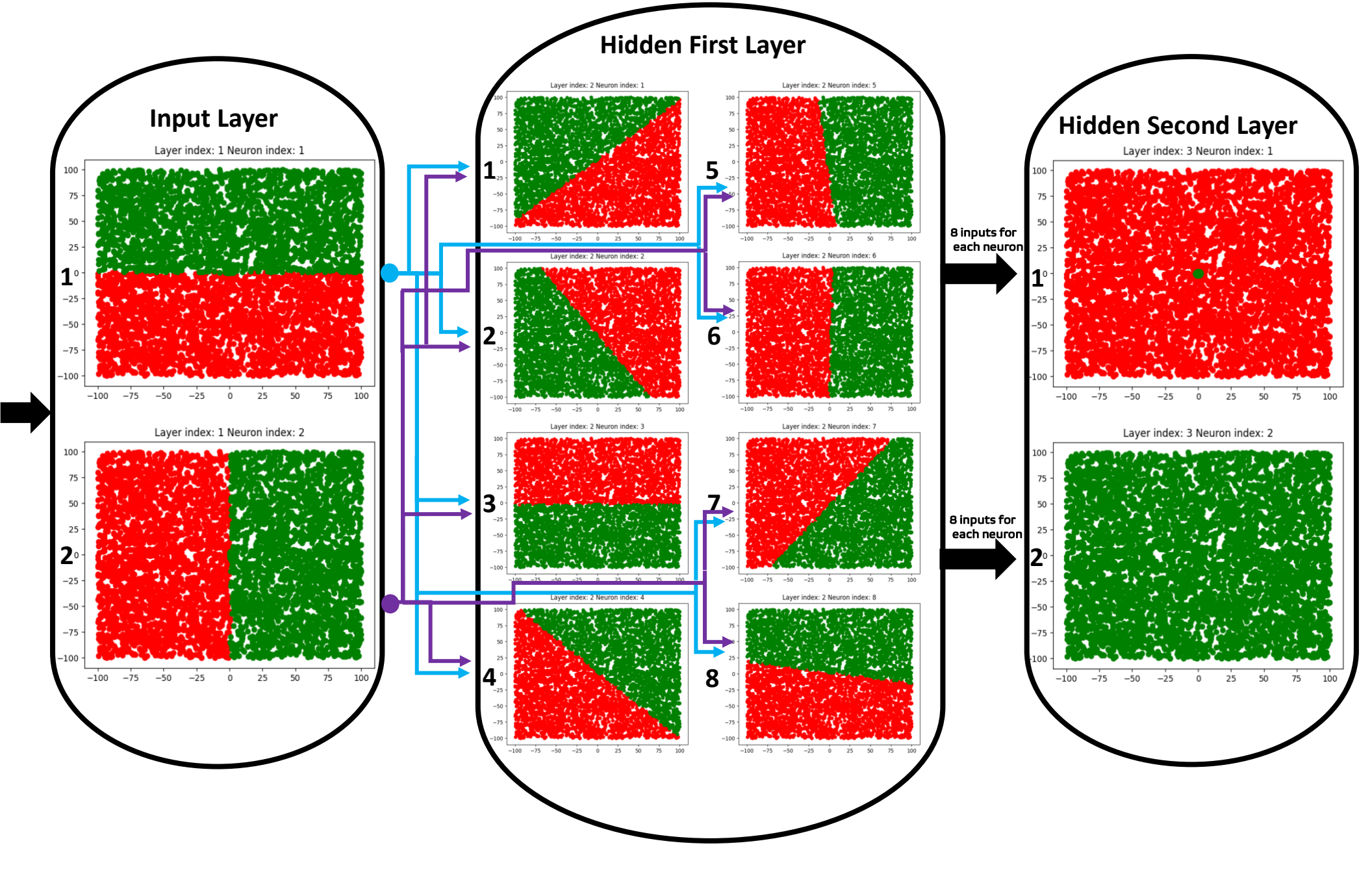
$$v = g((x,y)) = \begin{cases} 1, & 4 \leq x^2 + y^2 \leq 9 \\ -1, & otherwise \end{cases}$$

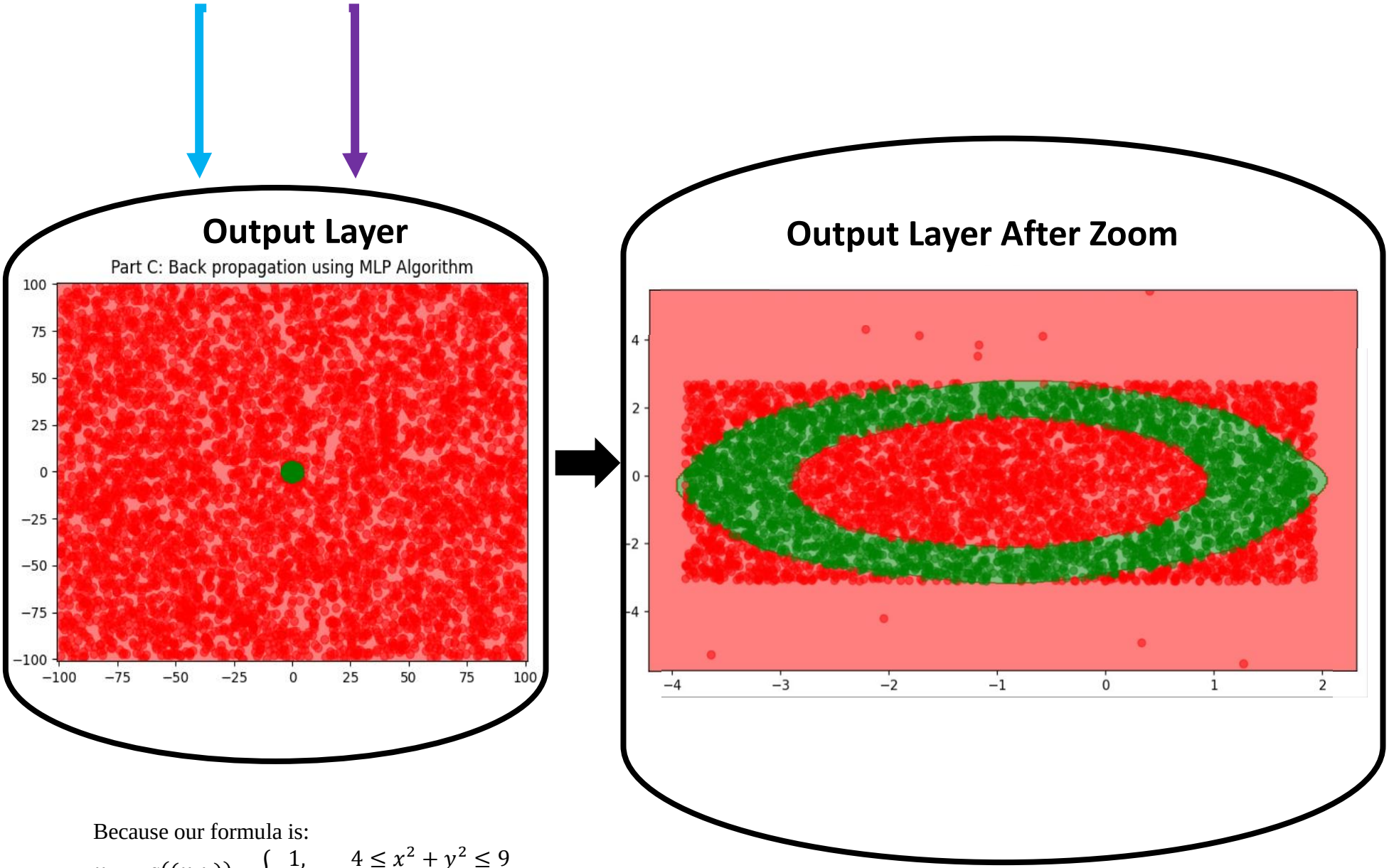
We solved this problem using the MLP Classifier from Sklearn.
The MLPClassifier uses Feed-Forward Networks and Back-propagation, with a given data and desired output, the MLPClassifier train the model and changes the weight and bias accordingly.

We chose to do it with 2 hidden layers:

1. The first hidden layer will have 8 neurons.
2. The second hidden layer will have 2 neurons.

Below we can see a diagram for this network:


Input Layer
Layer index: 1 Neuron index: 1
1
Layer index: 1 Neuron index: 2
2
Hidden First Layer
Layer index: 2 Neuron index: 1
1
Layer index: 2 Neuron index: 2
2
Layer index: 2 Neuron index: 3
3
Layer index: 2 Neuron index: 4
4
Layer index: 2 Neuron index: 5
5
Layer index: 2 Neuron index: 6
6
Layer index: 2 Neuron index: 7
7
Layer index: 2 Neuron index: 8
8
8 inputs for each neuron
8 inputs for each neuron
Hidden Second Layer
Layer index: 3 Neuron index: 1
1
Layer index: 3 Neuron index: 2
2



Because our formula is:

$$v = g((x,y)) = \begin{cases} 1, & 4 \le x^2 + y^2 \le 9 \\ -1, & otherwise \end{cases}$$

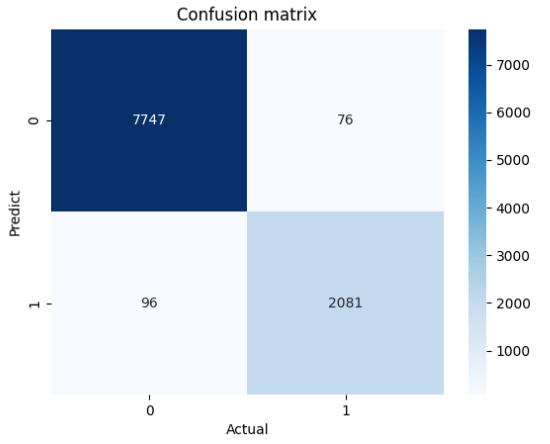
It makes sense that all the green area will look like the space with:
All the circles such that: $2 \le Radius \le 3$, and with the center of $(0,0)$, and with x, y such that: $\{4 \le x^2 + y^2 \le 9\}$.

To conclude the Back-Propagation Algorithm using the MLP result of part B with the balanced dataset was **98.28%.**

Which is almost twice as better as the Adaline score which was **55.87%.**

We can see from the Confusion Matrix that the model doesn't overfit:

Back-Propagation



Adaline

Part D:

In this part, we will use our trained neurons from the next to last level of part C as our input. The last neuron will do the prediction according to the Adaline algorithm and not the Back-Propagation Algorithm that we used in part C, and we will train only the Adaline neuron.

Our problem is the same as before:

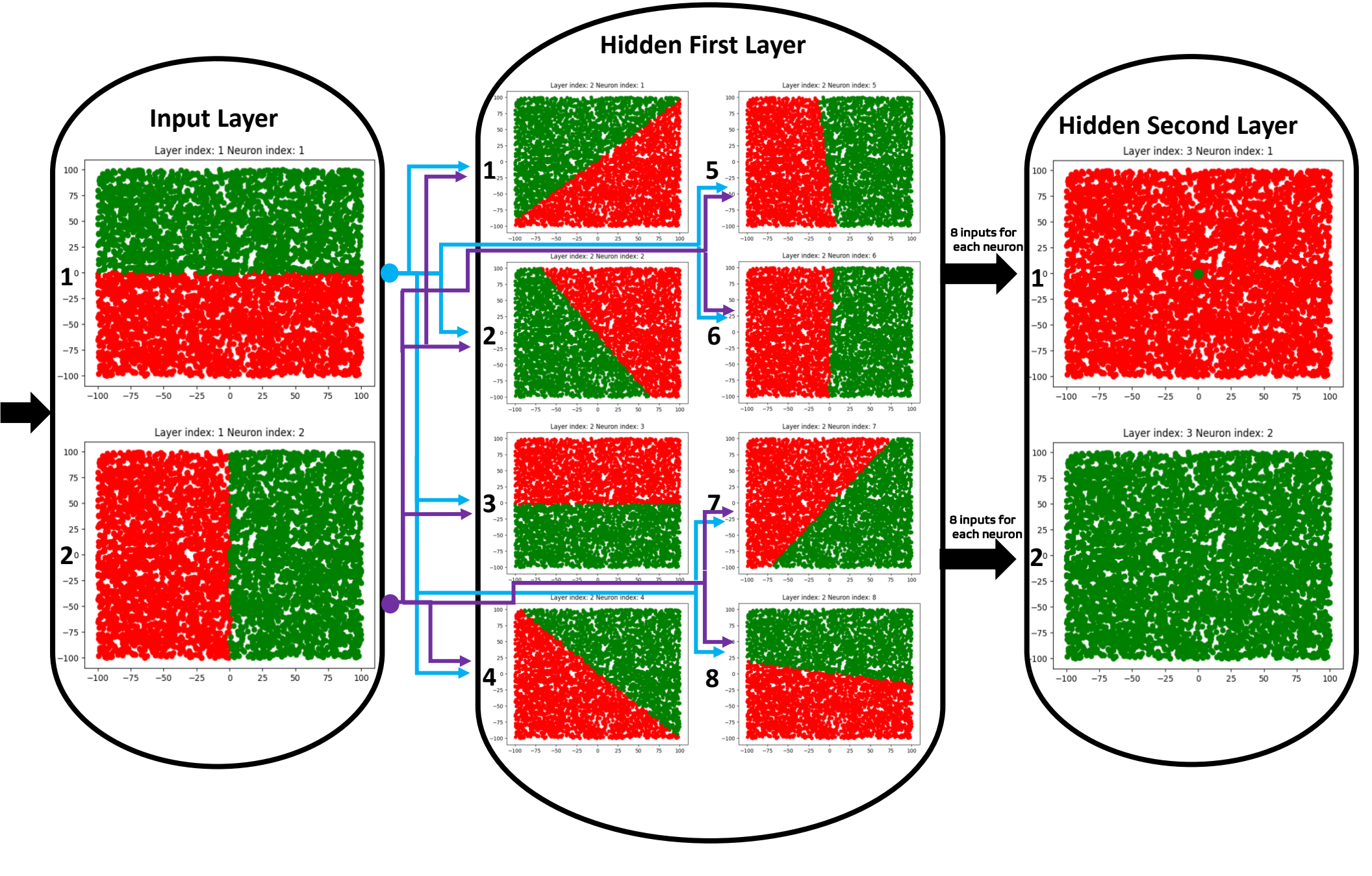
$$v = g((x,y)) = \begin{cases} 1, & 4 \le x^2 + y^2 \le 9 \\ -1, & otherwise \end{cases}$$

After doing so, combining the MLP Classifier with the Adaline Algorithm, our model score was $\mathbf{98.3}\%$.

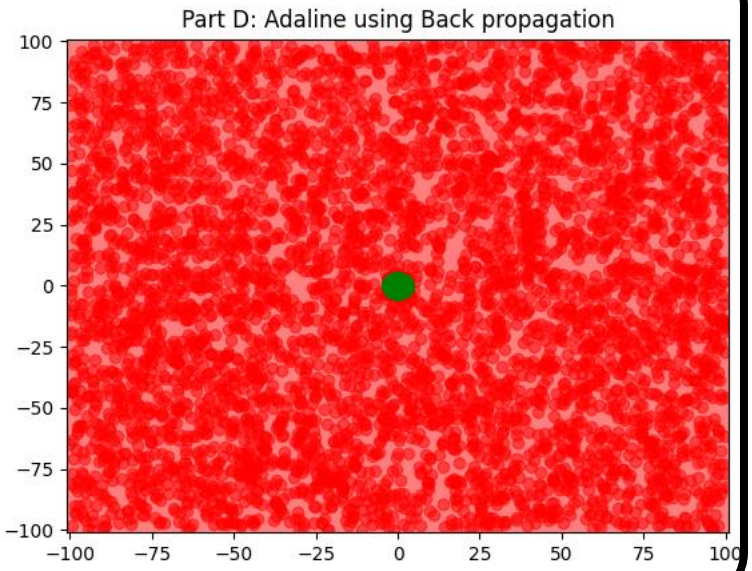
```
Part D

Adaline with MLP test score: 98.3%
[[ -1.25  -1.04]
 [  2.37   2.02]
 [  2.79  -0.89]
 ...
 [-58.08 -37.18]
 [ 85.91  28.23]
 [ 39.8   18.58]]
```

Which is almost the same as the result of the Back-Propagation Algorithm $\mathbf{98.28\%.}$

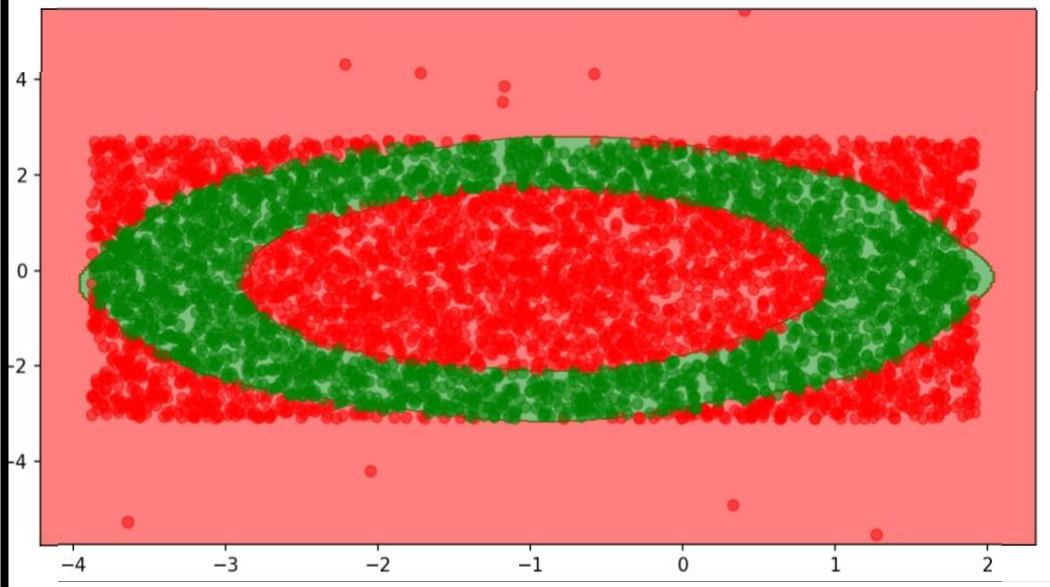
Below we can see a diagram for this network:


Input Layer
Layer index: 1 Neuron index: 1
1
Layer index: 1 Neuron index: 2
2
Hidden First Layer
Layer index: 2 Neuron index: 1
1
Layer index: 2 Neuron index: 2
2
Layer index: 2 Neuron index: 3
3
Layer index: 2 Neuron index: 4
4
Layer index: 2 Neuron index: 5
5
Layer index: 2 Neuron index: 6
6
Layer index: 2 Neuron index: 7
7
Layer index: 2 Neuron index: 8
8
8 inputs for each neuron
8 inputs for each neuron
Hidden Second Layer
Layer index: 3 Neuron index: 1
1
Layer index: 3 Neuron index: 2
2

Output Layer

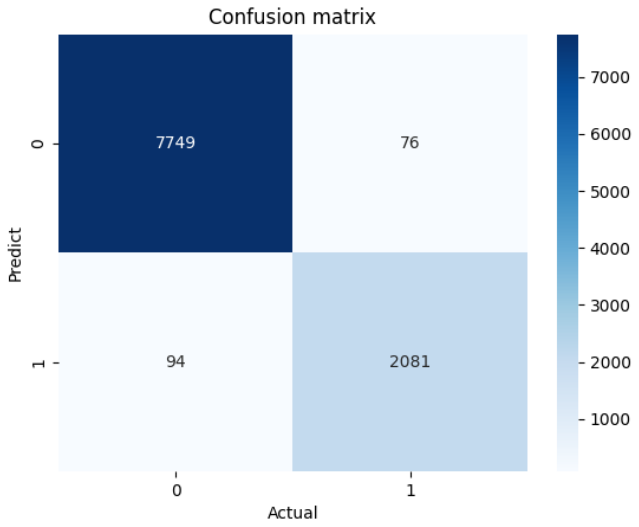


Output Layer After Zoom

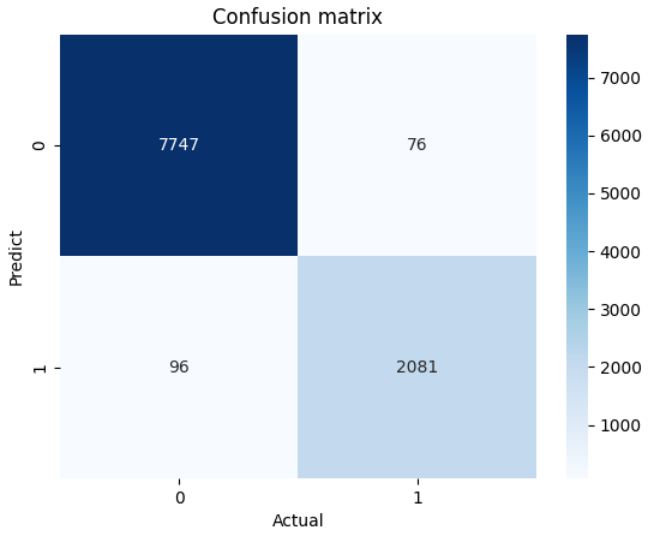


As we saw in the diagram the scatters of the Adaline using the MLP and the Back-Propagation using the MLP are almost identical, which is why both of the scores are almost the same.
From the Confusion Matrix, we can conclude that both models' prediction was quite similar.

Adaline with MLP



Back-Propagation with MLP



We can see that the only difference between those models is that Adaline with MLP predicted 2 points as -1 while Back-Propagation with MLP predicted apparently those points as 1.
To conclude, our first Adaline (from part B) score was: **55.879%**, the Back-Propagation with MLP score was **98.28%,** and the Adaline with MLP score was **98.3%**.
So we saw that MLP was able to improve our model and by using MLP in two different Algorithms we got almost the same scores.

Code:

Functions to create random points the first is balanced the second is not:

```python
# Creating random data - Balanced
def create_random_points(data_size, n):
    points = np.empty((data_size, 2), dtype=object)
    random.seed(7)
    for i in range(data_size // 2):
        points[i, 0] = random.randint(-300, 300) / 100
        points[i, 1] = random.randint(-300, 300) / 100
    for i in range(data_size // 2, data_size):
        points[i, 0] = random.randint(-n, n) / 100
        points[i, 1] = random.randint(-n, n) / 100
    targets = np.array([(1 if 4 <= (pow(points[i][0], 2) + pow(points[i][1], 2)) <= 9 else -1) for i in range(data_size)])
    features = points.astype(np.float64)
    targets = targets.astype(np.float64)
    return features, targets

# Creating random data - unBalanced
def create_random_points():
    return round(random.randint(-10000, 10000) / 100, 15)
```

Main:

```python
if __name__ == '__main__':
    X_train, y_train = create_random_points(data_size, 10000)
    X_test, y_test = create_random_points(data_size, 10000)
    classifier_mlp = MLPClassifier(activation='logistic', learning_rate_init=0.1, hidden_layer_sizes=(8, 2), random_state=7)
    classifier_mlp.fit(X_train, y_train)
    partC(X_train, y_train, X_test, y_test, classifier_mlp)
    partD(X_train, y_train, X_test, y_test, classifier_mlp)
```

partC function:

```python
def partC(X_train, y_train, X_test, y_test, model):
    print("\nPart C\n")
    print("Score of correct prediction: ", model.score(X_test, y_test) * 100, "%")
    diagram(model, X_train)  # present the geometric diagrams
    show_area(X_test, y_test, model)  # present the final test neurons distribution
    get_cm(model.predict(X_test), y_test)  # present the confusion matrix of the test part
```

diagram:

```python
# This function will present the geometric diagram of the model
def diagram(model, X_train):
    for layer_index in range(1, model.n_layers_):  # walking through all the neurons layers
        layer = get_layer(model, X_train, layer_index)  # getting the current layer
        for neuron_index, neuron in enumerate(layer, start=1):  # walking through every neuron
            # plotting the neuron data prediction
            plt.scatter(x=X_train[neuron == -1, 1], y=X_train[neuron == -1, 0], c='red', label=-1.0)
            plt.scatter(x=X_train[neuron == 1, 1], y=X_train[neuron == 1, 0], c='green', label=1.0)
            plt.title(f"Layer index: {str(layer_index)} Neuron index: {str(neuron_index)}")
            plt.show()
```

show_area:

```python
# Presenting the final test neurons distribution for part C
def show_area(X_test, y_test, model):
    x_min, y_min = X_test[:, 0].min() - 1, X_test[:, 1].min() - 1
    x_max, y_max = X_test[:, 0].max() + 1, X_test[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.02), np.arange(y_min, y_max, 0.02))  # getting the axis coordinates
    pred = model.predict(np.array([xx.flatten(), yy.flatten()]).T)
    pred = pred.reshape(xx.shape)
    colors = LCM(('red', 'green'))
    plt.contourf(xx, yy, pred, cmap=colors, alpha=0.5)  # draw contour lines and filled contours, respectively
    # limits of the data plotted
    plt.xlim(xx.min(), xx.max())
    plt.ylim(yy.min(), yy.max())
    plt.scatter(x=X_test[y_test == -1, 1], y=X_test[y_test == -1, 0], c='red', alpha=0.5, label=-1.0)
    plt.scatter(x=X_test[y_test == 1, 1], y=X_test[y_test == 1, 0], c='green', alpha=0.5, label=1.0)
    plt.title("Part C: Back propagation using MLP Algorithm")
    plt.show()
```

get_cm (confusion matrix):

```python
# Presenting the confusion matrix
def get_cm(predictions, y_test):
    c_m = confusion_matrix(predictions, y_test)
    plt.subplots()
    sb.heatmap(c_m, annot=True, fmt=".0f", cmap="Blues")
    plt.title("Confusion matrix")
    plt.xlabel("Actual")
    plt.ylabel("Predict")
    plt.show()
```

partD function:

```
def partD(X_train, y_train, X_test, y_test, model):
    print("\nPart D\n")
    adaline = Adaline(epochs=2)
    Adaline_with_MLP(adaline, model, X_train, y_train, X_test, y_test)
```

Adaline_with_MLP:

```
def Adaline_with_MLP(adaline, MLP, X_train, y_train, X_test, y_test):
    X_train = get_last_bp_layer_for_adaline(MLP, X_train)  # calculating the last layer output with the training data
    y_train[y_train == -1] = 0  # changing to 0 because the packaged adaline can have 0 and 1
    y_train = y_train.astype(int)
    adaline.fit(X_train, y_train)  # fitting the training data
    X_test_copy = X_test
    X_test = get_last_bp_layer_for_adaline(MLP, X_test)
    y_test[y_test == -1] = 0  # changing to 0 because the packaged adaline can have 0 and 1
    y_test = y_test.astype(int)
    score = adaline.score(X_test, y_test)
    print(f'Adaline with MLP test score: {score * 100}%')
    predictions = adaline.predict(X_test)
    get_cm(predictions, y_test)  # presenting the confusion matrix
    show_areaD(X_test_copy, X_test, y_test, adaline, MLP)
```

get_last_bp_layer_for_adaline:

```
# Calculating the last layer output with the training data
def get_last_bp_layer_for_adaline(model, X):
    features = X
    act = Act[model.activation]  # getting the model activation function
    for i in range(model.n_layers_ - 1):  # walking through each layer until the layer_index-1
        weights_i, bias_i = model.coefs_[i], model.intercepts_[i]  # getting the weights and bias of each layer
        features = np.dot(features, weights_i) + bias_i  # multiplying each point by the weights using dot product and adding the bias
        act(features)  # calculating the activation function to everything but the last layer_index
    return features
```

show_areaD:

```
# Presenting the final test neurons distribution for part D
def show_areaD(X_test_orig, X_test, y_test, model, mlp):
    print(X_test_orig)
    x_min, y_min = X_test_orig[:, 0].min() - 1, X_test_orig[:, 1].min() - 1
    x_max, y_max = X_test_orig[:, 0].max() + 1, X_test_orig[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.02), np.arange(y_min, y_max, 0.02))  # getting the axis coordinates
    lss = get_last_bp_layer_for_adaline(mlp, np.array([xx.flatten(), yy.flatten()]).T)  # calculating the last layer output with the training data
    pred = model.predict(lss)
    pred = pred.reshape(xx.shape)
    colors = LCM(('red', 'green'))
    plt.contourf(xx, yy, pred, cmap=colors, alpha=0.5)  # draw contour lines and filled contours, respectively
    # limits of the data plotted
    plt.xlim(xx.min(), xx.max())
    plt.ylim(yy.min(), yy.max())
    plt.scatter(x=X_test_orig[y_test == 0, 1], y=X_test_orig[y_test == 0, 0], c='red', alpha=0.5, label=-1.0)
    plt.scatter(x=X_test_orig[y_test == 1, 1], y=X_test_orig[y_test == 1, 0], c='green', alpha=0.5, label=1.0)
    plt.title("Part D: Adaline using Back propagation")
    plt.show()
```