Python手搓神经网络

来源于：https://b23.tv/FvTGUJW
参考代码：https://github.com/JunlingWang/Neuronetwork_with_python/tree/main
进阶：https://github.com/ZHE2018/network/blob/master/network3.py

以下是重要内容的笔记

目录

00.Numpy与线性代数

非常重要的是这里要注意Numpy的数组的切片是原始数组的视图。这意味着数据不会被复制，视图上的任何修改都会 直接反映到源数组上。

重要的语法：
矩阵乘法 `np.dot(a, b)`；
轴向平均 `np.mean(a, axis)`；
最值函数 `np.maximum(number, a)`；
最大值 `np.max(a)`；

01.神经网络（结构）简介

整个网络：输入层、隐藏层、输出层；
两个神经元之间为一层：权重矩阵，偏置；
层的前馈：激活函数；
反向传播；
训练（训练集、测试集）、推理；

02～09.搭建前馈部分

激活函数ReLU：输入值小于0时输出0，大于0时输出输入值；
激活函数softmax：将输入值平移到负值区后变为指数再做概率归一化；

10~15.反向传播算法

In [1]:
```
# 导入必要的库

import numpy as np
import matplotlib.pyplot as plt
import math
import random
import copy
```

In [2]:
```
# 参数

NETWORK_SHAPE = [2, 100, 200, 100, 2]# 神经网络的结构
BATCH_SIZE = 30 # 一批次数据的数据总数
LEARNING_RATE = 0.003 # 学习率
LOSS_THRESHOLD = 0.1 # 损失临界值
FORCE_TRAIN_THRESHOLD = 0.05 # 强制训练临界值

force_train = False # 强制训练开关
n_improved = 0
n_not_improved = 0
current_loss = 1
```

In [3]:
```
# 函数

# ReLU激活函数
def activation_ReLU(inputs):
    """输入值小于0时输出0，大于0时输出输入值"""
    return np.maximum(0, inputs)

# softmax激活函数
def activation_softmax(inputs):
    """将输入值平移到负值区后变为指数再做概率归一化"""
    max_value = np.max(inputs, axis=1, keepdims=True)
    slided_inputs = inputs - max_value
    exp_value = np.exp(slided_inputs)
    norm_base = np.sum(exp_value, axis=1, keepdims=True)
    norm_values = exp_value / norm_base
    return norm_values

# 归一化函数
def normalize(array):
    """将矩阵进行归一化"""
    max_number = np.max(np.absolute(array), axis=1, keepdims=True)
```

```
    scale_rate = np.where(max_number == 0, 1, 1/max_number)
    norm = array * scale_rate
    return norm

# 分类函数
def classify(probabilities):
    """按照网络的预测结果（第二列）打标签"""
    return np.rint(probabilities[:, 1]).reshape(-1, 1)

# 需求函数
def demands(predicted_values, targets_vector):
    """根据网络当次预测结果（也就是最后一层输出,这将是一个n行2列的矩阵）和标准答案（这将是
    来返回一个用于调整参数的demands（这个demands只是最后一层的，还需要向前传递）"""
    demands = np.zeros((len(targets_vector), 2))
    demands[:, 1] = targets_vector
    demands[:, 0] = 1 - targets_vector

    for i in range(len(targets_vector)):
        if np.dot(predicted_values[i], demands[i]) > 0.5 :
            demands[i] = np.array([0, 0])
        else:
            demands[i] = 2 * (demands[i] - 0.5)
    return demands

# 向量标准化函数
def vector_normalize(array):
    max_number = np.max(np.absolute(array))
    scale_rate = np.where(max_number == 0, 1, 1/max_number)
    norm = array * scale_rate
    return norm

# 精确损失函数
def precise_loss_function(predicted, real):
    real_matrix = np.zeros((len(real), 2))
    real_matrix[:, 1] = real
    real_matrix[:, 0] = 1 - real
    product = np.sum(predicted*real_matrix, axis=1)
    return 1 - product

# 损失函数
def loss_function(predicted, real):
    condition = (predicted > 0.5)
    binary_predicted = np.where(condition, 1, 0)
    real_matrix = np.zeros((len(real), 2))
    real_matrix[:, 1] = real
    real_matrix[:, 0] = 1 - real
    product = np.sum(predicted*real_matrix, axis=1)
    return 1 - product
```

In [4]:

```
# 神经网络

# 定义一个层类
class Layer:
    """两个神经元之间的层"""
    def __init__(self, n_inputs, n_neurons):
        self.weights = np.random.randn(n_inputs, n_neurons) # 权重矩阵
        self.biases = np.random.randn(n_neurons) # 偏置

    def layer_forward(self, inputs):
        """层的前馈"""
        return np.dot(inputs, self.weights) + self.biases

    def layer_backward(self, prelayer_outputs, currentlayer_demands):
        """层的反向传播，输入前一层的输出结果，和本层的需求；
```

```
        调整矩阵：根据反向传播与梯度下降的想法，根据链式法则，分别输入本层的输入a（也就是
        返回本层权重矩阵的调整矩阵；

        返回归一化的前一层的需求和归一化的本层的调整矩阵"""

        # ReLU函数的导数
        condition = (self.layer_forward(prelayer_outputs) > 0)
        derivatives_ReLU = np.where(condition, 1, 0)

        # 权重的调整矩阵
        weights_adjust_matrix = np.full(self.weights.shape, 0.0)
        for i in range(BATCH_SIZE):
            weights_adjust_matrix = weights_adjust_matrix + (prelayer_outpu
        weights_adjust_matrix = weights_adjust_matrix / BATCH_SIZE

        # 前一层的需求
        prelayer_demands = np.dot((currentlayer_demands * derivatives_ReLU),

        # 归一化
        norm_prelayer_demands = normalize(prelayer_demands)
        norm_weights_adjust_matrix = normalize(weights_adjust_matrix)

        return (norm_prelayer_demands, norm_weights_adjust_matrix)

# 定义一个网络类
class Network:
    """整个神经网络"""
    def __init__(self, network_shape):
        self.shape = network_shape
        self.layers = []
        for i in range(len(network_shape) - 1):
            self.layers.append(Layer(network_shape[i], network_shape[i + 1])

    def network_forward(self, inputs):
        """前馈运算函数，返回每一层的预测结果"""
        outputs = [inputs]
        for i in range(len(self.layers)):
            if i < len(self.layers) - 1:
                layer_outputs = activation_ReLU(self.layers[i].layer_forward
                layer_outputs = normalize(layer_outputs)
            else:
                layer_outputs = activation_softmax(self.layers[i].layer_forw
            outputs.append(layer_outputs)

            #print(layer_outputs)
            #print('-------------------------------------------')

        return outputs

    def network_backward(self, outputs, targets_vector):
        """反向传播函数，传入各层预测值和标准答案，对参数做调整，返回调整后的新网络（一个
        new_network = copy.deepcopy(self) # 备用网络
        layer_demands = demands(outputs[-1], targets_vector) # 最后一层的需求函

        # 先调整最后一层
        final_layer = new_network.layers[-1]
        # 调整偏置
        final_layer.biases = final_layer.biases + LEARNING_RATE * np.mean(la
        final_layer.biases = vector_normalize(final_layer.biases)
        # 调整权重
        final_weights_adjust_matrix = np.full(final_layer.weights.shape, 0.0
        for i in range(BATCH_SIZE):
```

```
            #print(outputs[-2][i].reshape(len(outputs[-2][i]), 1))
            #print(layer_demands[i])
            #print("outputs[-2][i] shape:", outputs[-2][i].shape)
            #print("layer_demands[i] shape:", layer_demands[i].shape)

            final_weights_adjust_matrix = final_weights_adjust_matrix + (ou
        final_weights_adjust_matrix = final_weights_adjust_matrix / BATCH_S
        norm_final_weights_adjust_matrix = normalize(final_weights_adjust_ma
        final_layer.weights = final_layer.weights + LEARNING_RATE * norm_fi
        final_layer.weights = normalize(final_layer.weights)

        # 调整其它层
        layer_demands = normalize(np.dot(layer_demands, final_layer.weights

        for i in range(len(new_network.layers) - 1):
            layer = new_network.layers[-(i + 2)]

            #print(layer_demands)
            #print('-------------------------------------------')
            #print("layer_demands[i] shape:", layer_demands[i].shape)
            #print('-------------------------------------------')
            #print(np.mean(layer_demands, axis=0))

            # 调整偏置
            layer.biases = layer.biases + LEARNING_RATE * np.mean(layer_dema
            layer.biases = vector_normalize(layer.biases)

            # 获得前一层的需求和本层的权重调整矩阵
            layer_demands, weights_adjust_matrix = layer.layer_backward(outp

            # 调整权重
            layer.weights = layer.weights + LEARNING_RATE * weights_adjust_m
            layer.weights = normalize(layer.weights)

        return new_network

    def one_batch_training(self, data_batch):
        """进行单批次训练"""
        global force_train, n_improved, n_not_improved

        inputs = data_batch[:, (0, 1)]
        targets_vector = copy.deepcopy(data_batch[:, 2]).astype(int) # 标准答
        outputs = self.network_forward(inputs)
        precise_loss = precise_loss_function(outputs[-1], targets_vector) #
        loss = loss_function(outputs[-1], targets_vector) # 损失

        if np.mean(loss) <= LOSS_THRESHOLD:
            print('No need for training') # 不需要进行进一步训练
            return 0

        else:
            # 进行训练，生成新网络
            new_network = self.network_backward(outputs, targets_vector)
            new_outputs = new_network.network_forward(inputs)
            new_precise_loss = precise_loss_function(new_outputs[-1], target
            new_loss = loss_function(new_outputs[-1], targets_vector)

            # 利用两种损失比较新旧网络的准确度，若新网络更好则将旧网络替换为新网络
            if np.mean(precise_loss) >= np.mean(new_precise_loss) or np.mea
                for i in range(len(self.layers)):
                    self.layers[i].weights = new_network.layers[i].weights.c
                    self.layers[i].biases = new_network.layers[i].biases.cop
                #print('Improved')
                n_improved += 1
```

```
                #print('--------------------------------------------------')

            else:
                if force_train:
                    for i in range(len(self.layers)):
                        self.layers[i].weights = new_network.layers[i].weigh
                        self.layers[i].biases = new_network.layers[i].biases
                    print('Force train')
                else:
                    #print('No improvement')
                    n_not_improved += 1
                #print('--------------------------------------------------')

    def train(self, n_entries):
        """多批次训练"""
        global force_train, n_improved, n_not_improved
        n_improved = 0
        n_not_improved = 0

        # 训练集
        n_batches = n_entries // BATCH_SIZE + 1
        for i in range(n_batches):
            data_batch = generate_data(BATCH_SIZE)
            self.one_batch_training(data_batch)
            if self.one_batch_training(data_batch) == 0:
                break

        improvement_rate = n_improved / (n_improved + n_not_improved)
        print('improvement_rate is')
        print(improvement_rate)
        #print('--------------------------------------------------')
        print('--------------------------------------------------')

        if improvement_rate <= FORCE_TRAIN_THRESHOLD:
            force_train = True
        else:
            force_train = False

        # 测试集
        data = generate_data(1000)
        inputs = data[:, (0, 1)]
        outputs = self.network_forward(inputs)
        classification = classify(outputs[-1])
        data[:, 2] = classification.T
        plot_data(data, 'After Training')
```

In [5]:

```
# 生成数据与可视化

def tag_entry(x, y):
    """打标函数，在单位圆外标为1，反之标为0"""
    if x**2 + y**2 > 1:
        tag = 1
    else:
        tag = 0
    return tag

def generate_data(num_of_data):
    """随机生成数据"""
    entry_list = []
    for i in range(num_of_data):
        x = random.uniform(-2, 2)
        y = random.uniform(-2, 2)
        tag = tag_entry(x, y)
        entry = [x, y, tag]
```

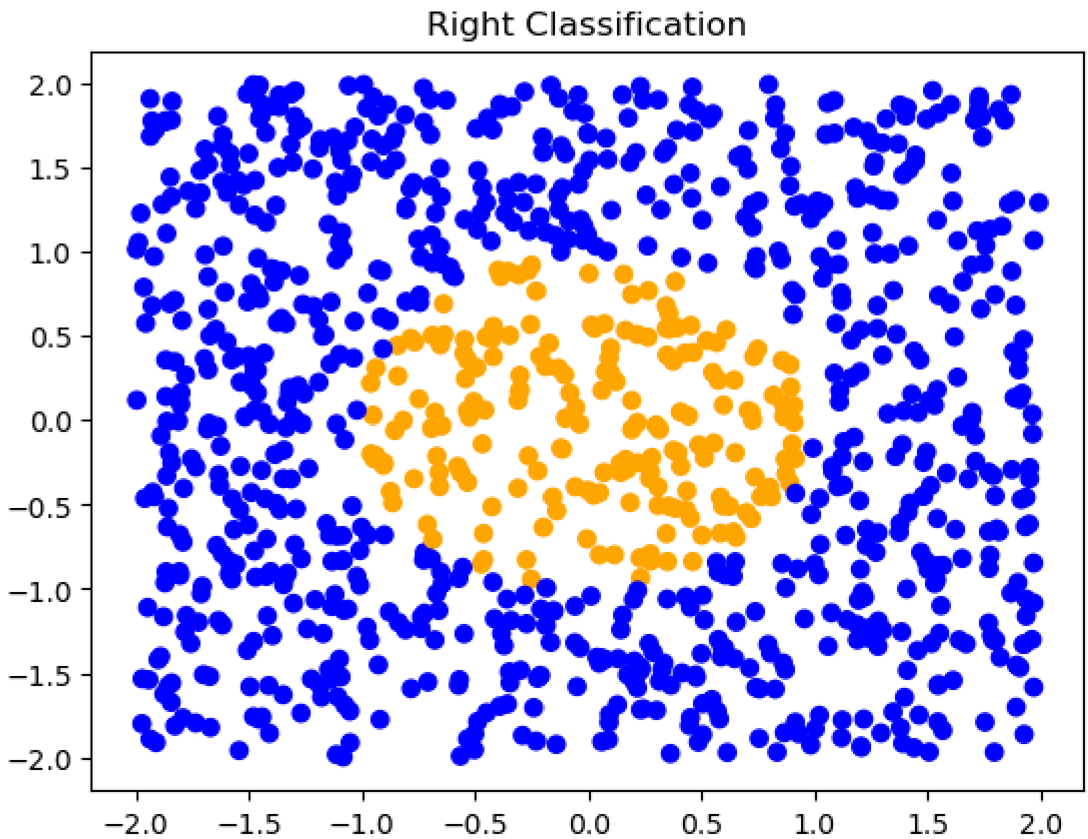
```python
        entry_list.append(entry)
    return np.array(entry_list)

def plot_data(data, title):
    """将数据可视化"""
    color = []
    for i in data[:, 2]:
        if i == 0:
            color.append('orange')
        else:
            color.append('blue')
    plt.scatter(data[:, 0], data[:, 1], c=color)
    plt.title(title)
    plt.show()
```

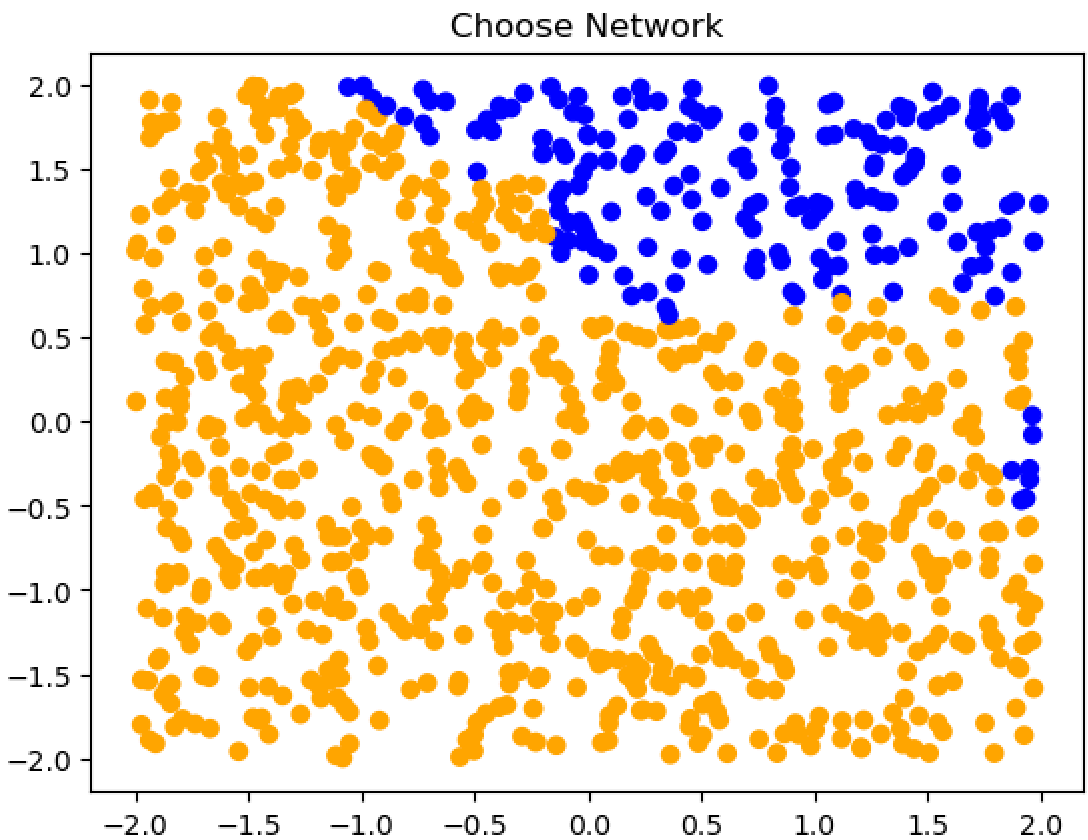
In [6]:

```python
# 训练流程控制

def main():
    global current_loss
    data = generate_data(1000) # 生成数据
    plot_data(data, "Right Classification")

    # 选择起始网络
    use_this_network = 'n' # No
    while use_this_network != 'Y' and use_this_network != 'y':
        network = Network(NETWORK_SHAPE)
        inputs = data[:, (0, 1)]
        outputs = network.network_forward(inputs)
        classification = classify(outputs[-1])
        data[:, 2] = classification.T
        plot_data(data, "Choose Network")
        use_this_network = input("Use this network? Y to yes, N to No \n")

    # 进行训练
    do_train = input("Train? Y to yes, N to No \n")
    while do_train == 'Y' or do_train == 'y' or do_train.isnumeric() == True
        if do_train.isnumeric() == True:
            n_entries = int(do_train)
        else:
            n_entries = int(input("Enter the number of data entries used to

        network.train(n_entries)
        do_train = input("Train? Y to yes, N to No \n")

    #演示训练效果
    inputs = data[:, (0, 1)]
    outputs = network.network_forward(inputs)
    classification = classify(outputs[-1])
    data[:, 2] = classification.T
    plot_data(data, "After training")
    print("谢谢，再见！")
```

In [7]:

```python
main()
```

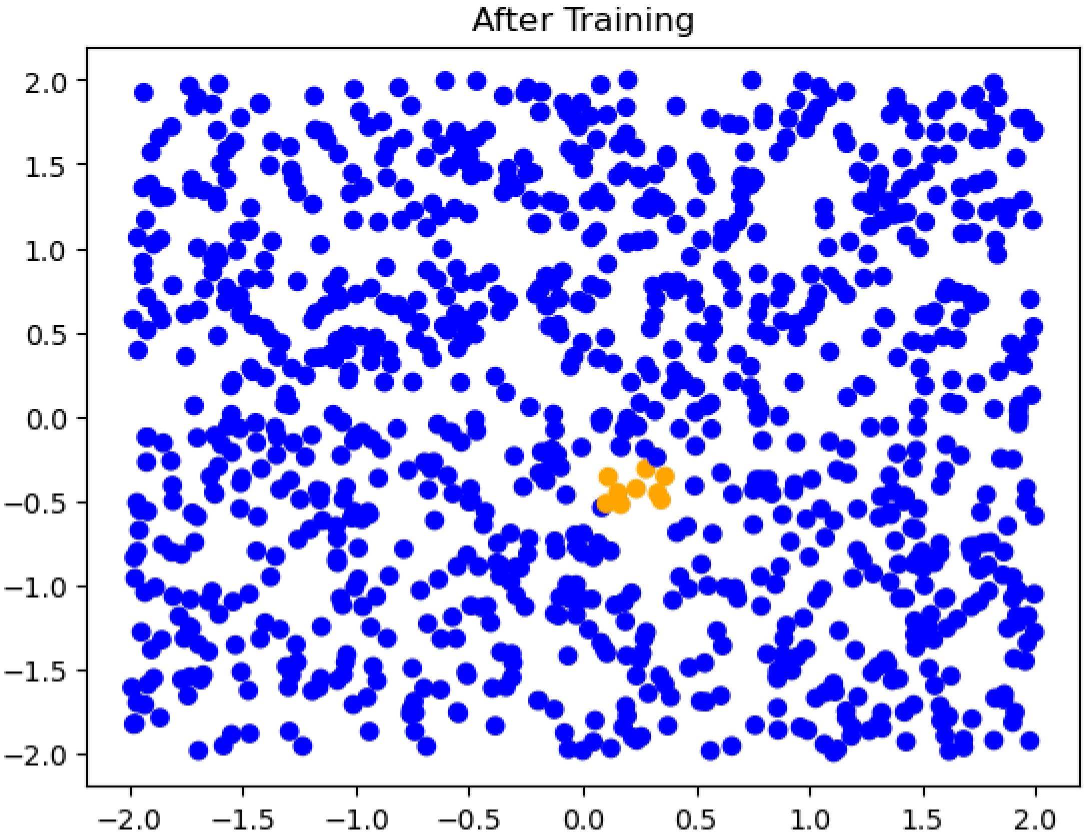




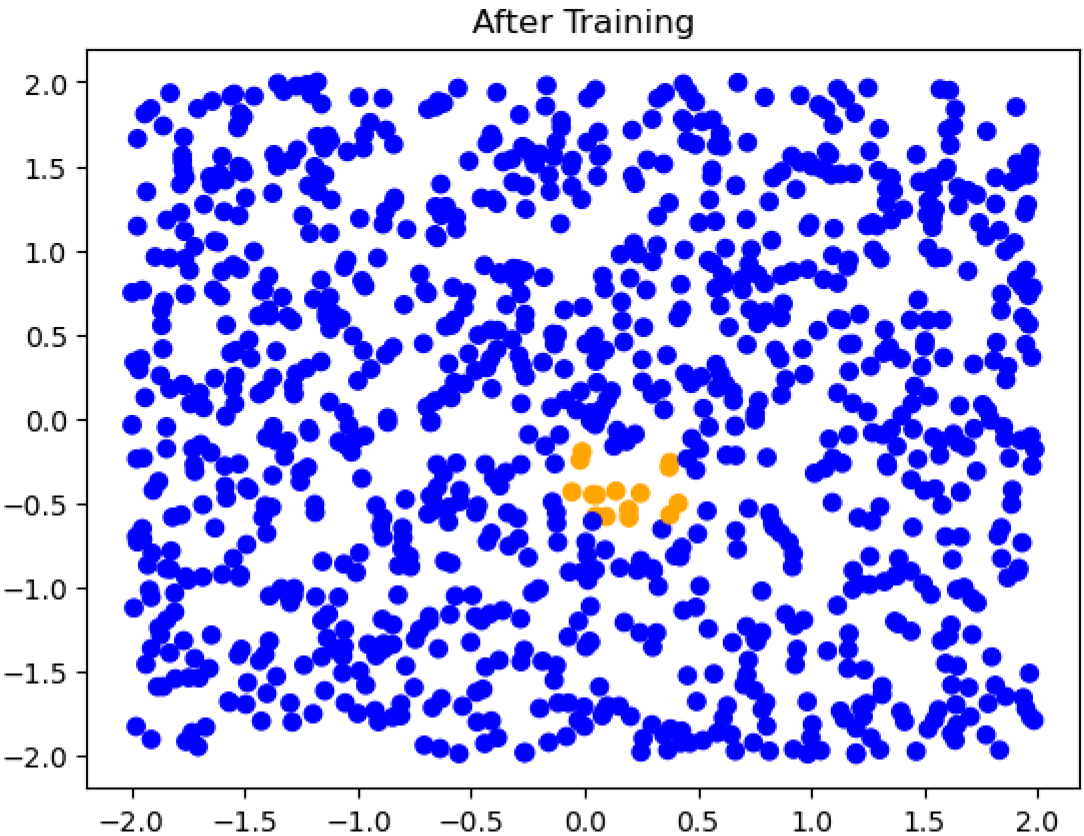

```
Use this network? Y to yes, N to No
y
Train? Y to yes, N to No
y
Enter the number of data entries used to train.
2500
```

```
/var/folders/7l/1kcst3m15b3f1jyrwvsyrckw0000gn/T/ipykernel_15134/144325295
2.py:22: RuntimeWarning: divide by zero encountered in divide
  scale_rate = np.where(max_number == 0, 1, 1/max_number)
```

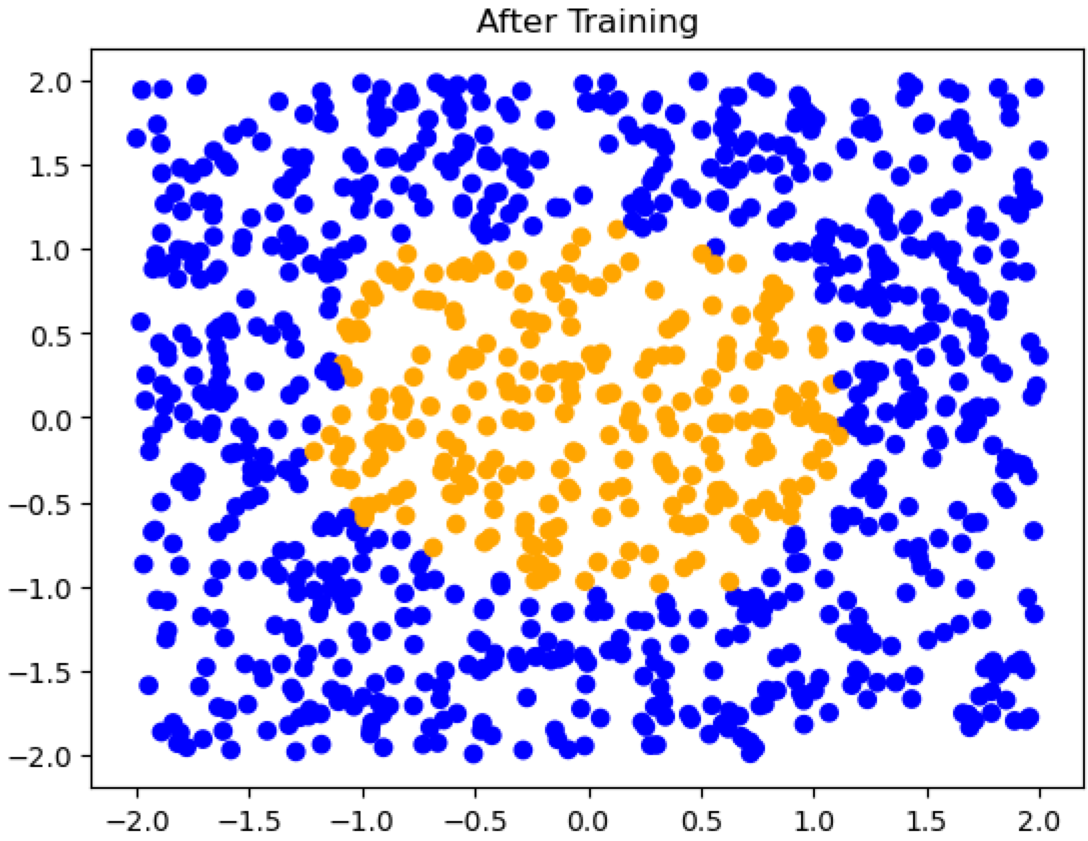
```
improvement_rate is
0.07142857142857142
--------------------------------------------------
```



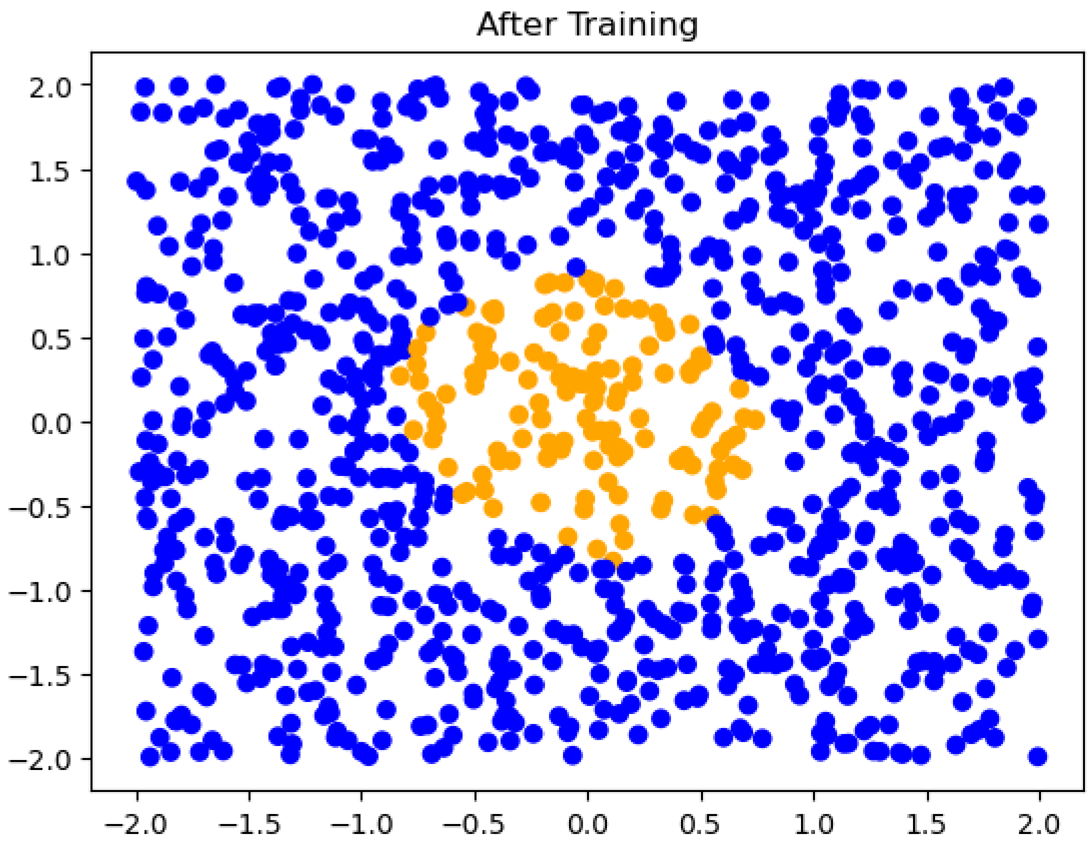

```
Train? Y to yes, N to No
y
Enter the number of data entries used to train.
3000
improvement_rate is
0.019801980198019802
--------------------------------------------------
```



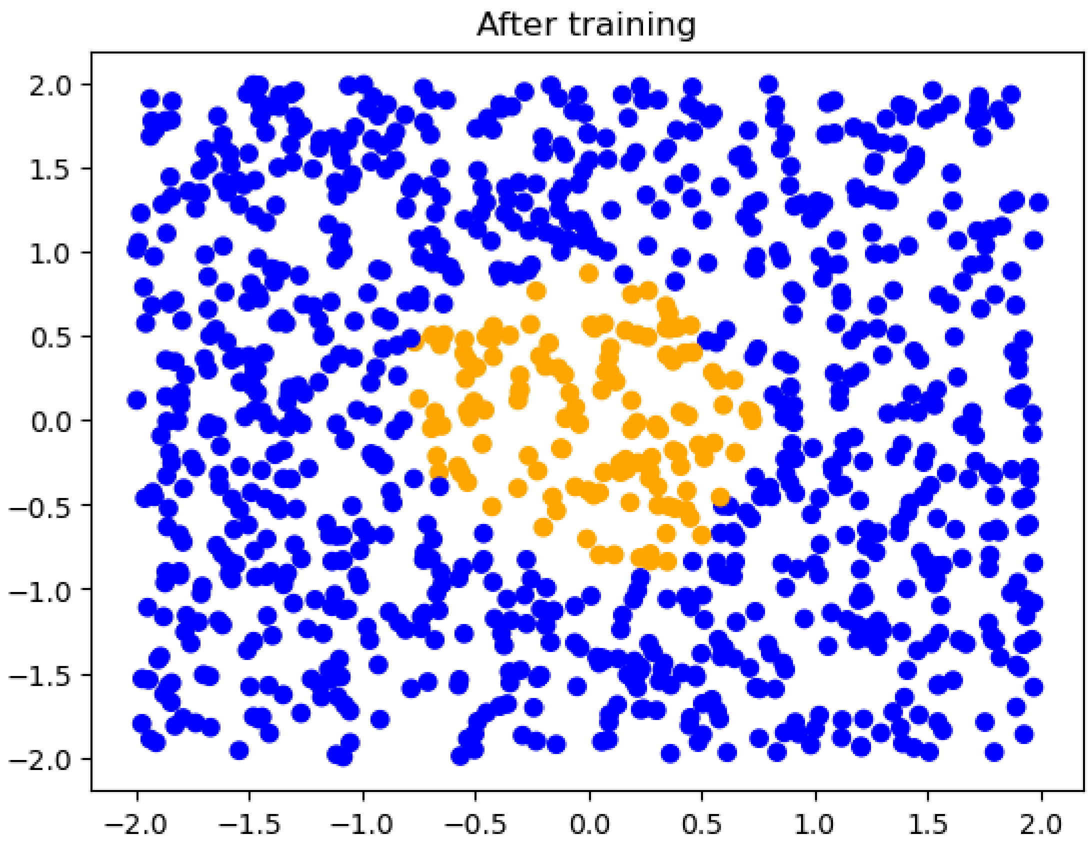
```
Train? Y to yes, N to No
y
Enter the number of data entries used to train.
3000
Force train
Force train
Force train
Force train
Force train
Force train
Force train
Force train
Force train
Force train
Force train
Force train
Force train
Force train
Force train
Force train
Force train
Force train
Force train
Force train
Force train
Force train
Force train
Force train
Force train
Force train
Force train
Force train
Force train
Force train
Force train
Force train
Force train
Force train
Force train
Force train
Force train
Force train
Force train
Force train
Force train
Force train
improvement_rate is
1.0
-------------------------------------------------
```




```
Train? Y to yes, N to No
y
Enter the number of data entries used to train.
1000
No need for training
improvement_rate is
0.2727272727272727
--------------------------------------------------
```




```
Train? Y to yes, N to No
n
```



谢谢，再见！

In [ ]: