Project2

1.Team member

- **m5268101 Liu Jiahe**
- **m5251140 Ken Sato**
- **m5271051 Keisuke Utsumi**

2.Team Project II

Team Project II

- Write a computer program for the BP algorithm.
- Test your program using the 4-bit parity check problem.
- The number of inputs is 5 (4 original inputs plus one

  dummy input) and the number of output is 1 (a real number in [0,1] or [-1,1]).
- The desired output is 1 if the number of ones in the inputs is even; otherwise, the output is 0 or -1.
- Check the performance of the network by changing the number of hidden neurons from 4, 6, 8, and 10.
- Provide a summary of your results in your report (txt-file).

3.数学原理

输入4bit的数字 (0或1)，如果数字1的个数是偶数，则输出1，否则输出0。

For example:

- (0, 0, 0, 0, -1) -> 1
- (0, 1, 1, 1, -1) -> 0
- (1, 0, 0, 1, -1) -> 1
- (1, 0, 1, 1, -1) -> 0

为了完成代码，首先需要**推导出前向传播和反向传播的数学公式**

关于四位数奇偶校验问题，输入神经元有五个，隐藏层有若干个神经元，输出层有一个神经元，其中隐藏层神经元和输出层神经元的激活函数都为sigmoid函数,他的原函数和导函数如下

$$\sigma(x) = \frac{1}{1+e^{-x}} \qquad \frac{d\sigma(x)}{dx} = \sigma(x)(1-\sigma(x)) \tag{2}$$

首先是前向传播， X表示输入矩阵，形状为(m, n_0)，其中 m 是样本数量，n_0 是输入特征数量，在这里 $n_0 = 5$。Y 表示目标输出矩阵，大小为 (m, n_2)，其中 n_2 是输出层神经元的数量，这里 $n_2 = 1$。权重矩阵 $W^{(1)}$ 和 $W^{(2)}$ 分别表示输入层到隐藏层和隐藏层到输出层的权重，大小分别为 (n_0, n_1) 和 (n_1, n_2)，其中 n_1 是隐藏层神经元的数量。

$A^{(1)}$ 表示隐藏层的输入矩阵被激活后的矩阵，大小为 (m, n_1)，隐藏层的输入矩阵 $Z^{(1)}$，大小为 (m, n_1)：

$$Z^{(1)} = XW^{(1)} \qquad A^{(1)} = \sigma(Z^{(1)}) \tag{3}$$

输出层的激活值矩阵为 $A^{(2)}$，大小为 (m, n_2)，输出层的输入矩阵 $Z^{(2)}$，大小为 (m, n_2)：

$$Z^{(2)} = A^{(1)}W^{(2)} \qquad A^{(2)} = \sigma(Z^{(2)}) \tag{4}$$

均方差损失函数：

$$L = \frac{1}{2m}\sum_{i=1}^{m}(Y^{(i)} - A^{(2)(i)})^2 \tag{5}$$

误差反向传播，找到损失函数L相对于权重W的偏导数：

$$\frac{\partial L}{\partial W^{(2)}} = \frac{\partial L}{\partial A^{(2)}}\frac{\partial A^{(2)}}{\partial Z^{(2)}}\frac{\partial Z^{(2)}}{\partial W^{(2)}} \tag{6}$$

$$\frac{\partial L}{\partial W^{(1)}} = \frac{\partial L}{\partial A^{(2)}}\frac{\partial A^{(2)}}{\partial Z^{(2)}}\frac{\partial Z^{(2)}}{\partial A^{(1)}}\frac{\partial A^{(1)}}{\partial Z^{(1)}}\frac{\partial Z^{(1)}}{\partial W^{(1)}} \tag{7}$$

$$(4) \Rightarrow \frac{\partial L}{\partial A^{(2)}} = \frac{1}{m}(Y - A^{(2)}) = \frac{1}{m}E \tag{8}$$

上面的Y,$A^{(2)}$,E,都是向量

$$\frac{\partial A^{(2)}}{\partial Z^{(2)}} = \frac{\partial \sigma(Z^{(2)})}{\partial Z^{(2)}} = \sigma^{'}(Z^{(2)})$$
$$\frac{\partial Z^{(2)}}{\partial W^{(2)}} = A^{(1)}$$

得到$\frac{\partial L}{\partial W^{(2)}}$

$$\Rightarrow \frac{\partial L}{\partial W^{(2)}} = \frac{1}{m}E * \sigma^{'}(Z^{(2)}) * A^{(1)T} \tag{9}$$

对应代码：

```
output_layer_error_term = error * sigmoid_derivative(output_layer_input)
dL_dW2 = np.dot(hidden_layer_output.T, output_layer_error_term) / len(inputs)
weights_hidden_output += learning_rate * dL_dW2
```

$$\frac{\partial Z^{(2)}}{\partial A^{(1)}} = W^{(2)}$$
$$\frac{\partial A^{(1)}}{\partial Z^{(1)}} = \sigma^{'}(Z^{(1)})$$
$$\frac{\partial Z^{(1)}}{\partial W^{(1)}} = X$$

得到$\frac{\partial L}{\partial W^{(1)}}$

$$\Rightarrow \frac{\partial L}{\partial W^{(1)}} = \frac{1}{m} E * \sigma^{'}(Z^{(2)}) * W^{(2)T} * \sigma^{'}(Z^{(1)}) * X^T \tag{10}$$

对应代码:

```
hidden_layer_error_term = np.dot(output_layer_error_term, weights_hidden_output.T) * sigmoid_derivative(hidden_layer_input)
dL_dW1 = np.dot(inputs.T, hidden_layer_error_term) / len(inputs)
weights_input_hidden += learning_rate * dL_dW1
```

4.代码实现

代码文件共三个:

utils.py

实现必要的功能函数

```
import numpy as np
from itertools import product

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

def sigmoid_derivative(x):
    s = sigmoid(x)
    return s * (1 - s)

def generate_dataset():
    inputs = []
    outputs = []

    for a in '01':
        for b in '01':
            for c in '01':
                for d in '01':
                    input_vector = [int(a), int(b), int(c), int(d), -1]
                    inputs.append(input_vector)

                    num_of_ones = sum(input_vector[:-1])
                    output = 0 if num_of_ones % 2 else 1
                    outputs.append([output])
```

```python

    inputs = np.array(inputs)
    outputs = np.array(outputs)

    return inputs, outputs

def generate_parity_dataset(n_parity):
    inputs = []
    outputs = []

    # Generate all possible n-bit binary combinations
    for binary_combination in product('01', repeat=n_parity): # Cartesian product
        input_vector = [int(bit) for bit in binary_combination]
        input_vector.append(-1)  # Add bias term
        inputs.append(input_vector)

        num_of_ones = sum(input_vector[:-1])
        output = 0 if num_of_ones % 2 == 0 else 1
        outputs.append([output])

    inputs = np.array(inputs)
    outputs = np.array(outputs)

    return inputs, outputs



def train(inputs, outputs, weights_input_hidden, weights_hidden_output, learning_rate, num_epochs):
    '''
    shape(inputs) = (2^n_parity, n_parity+1), shape(outputs) = (2^n_parity, 1)
    shape(weights_input_hidden) = (n_parity+1, hidden_neurons), shape(weights_hidden_output) = (hidden_neurons, 1)
    '''
    loss_list = []
    for epoch in range(num_epochs):

        # Forward pass
        hidden_layer_input = np.dot(inputs, weights_input_hidden) # shape(hidden_layer_input) = (2^n_parity, hidden_neurons)
        hidden_layer_output = sigmoid(hidden_layer_input)
        output_layer_input = np.dot(hidden_layer_output, weights_hidden_output) # shape(output_layer_input) = (2^n_parity, 1)
        output_layer_output = sigmoid(output_layer_input)

        # Calculate error and loss
        error = outputs - output_layer_output # shape(error) = (2^n_parity, 1), error is also the derivative of loss
        loss = 0.5 * np.mean(error ** 2)
        loss_list.append(loss)
        # print(f"Epoch {epoch + 1}: Loss: {loss}")
```

```python

        # Backpropagation
        output_layer_error_term = error * sigmoid_derivative(output_layer_input) # (2^n_parity, 1) = (2^n_parity, 1) * (2^n_parity, 1)
        dL_dW2 = np.dot(hidden_layer_output.T, output_layer_error_term) / len(inputs) # (hidden_neurons, 1) = (hidden_neurons, 2^n_parity)(2^n_parity, 1)

        # (2^n_parity, hidden_neurons) = (2^n_parity, 1)(1, hidden_neurons) * (2^n_parity, hidden_neurons)
        hidden_layer_error_term = np.dot(output_layer_error_term, weights_hidden_output.T) * sigmoid_derivative(hidden_layer_input)
        dL_dW1 = np.dot(inputs.T, hidden_layer_error_term) / len(inputs) # (n_parity+1, hidden_neurons) = (n_parity+1, 2^n_parity)(2^n_parity, hidden_neurons)

        # Update weights
        weights_hidden_output += learning_rate * dL_dW2
        weights_input_hidden += learning_rate * dL_dW1

    return weights_input_hidden, weights_hidden_output, loss_list

def test(inputs, weights_input_hidden, weights_hidden_output):
    hidden_layer_input = np.dot(inputs, weights_input_hidden)
    hidden_layer_output = sigmoid(hidden_layer_input)
    output_layer_input = np.dot(hidden_layer_output, weights_hidden_output)
    output_layer_output = sigmoid(output_layer_input)

    return output_layer_output




```

project2.py

实现一次训练，输出权重，损失函数图像，同时对训练好的模型进行一次测试 ,因为四位奇偶校验出现的情况有限，就直接用训练的全部16种数据进行测试

```python
import numpy as np
import matplotlib.pyplot as plt

from utils import *


np.set_printoptions(linewidth=np.inf)


def main():
```

```python

    n_parity = 4
    inputs, outputs = generate_parity_dataset(n_parity) # shape(inputs) = (16, 5), shape(outputs) = (16, 1)
    print( outputs )

    hidden_neurons = 8
    learning_rate = 2
    num_epochs = 50000

    input_size = inputs.shape[1] # shape(inputs) = (16, 5)
    output_size = outputs.shape[1] # shape(outputs) = (16, 1)

    weights_input_hidden = np.random.uniform(-1, 1, size=(input_size, hidden_neurons)) # shape(weights_input_hidden) = (5, 8)
    weights_hidden_output = np.random.uniform(-1, 1, size=(hidden_neurons, output_size)) # shape(weights_hidden_output) = (8, 1)

    weights_input_hidden, weights_hidden_output, loss_list = train(inputs, outputs, weights_input_hidden, weights_hidden_output, learning_rate, num_epochs)

    print(f"There are {hidden_neurons} hidden_neurons.")
    print("Training complete.")
    print("Weights from input layer to hidden layer:")
    print(weights_input_hidden)
    print("Weights from hidden layer to output layer:")
    print(weights_hidden_output)

    # Test the model on training data
    predictions = test(inputs, weights_input_hidden, weights_hidden_output) # shape(predictions) = (16, 1)

    # Print actual and predicted outputs
    print("\nTest the accuracy:\n")
    for i in range(inputs.shape[0]): # inputs.shape[0] = 16
        print(f"Input: {inputs[i]} | Desired Output: {outputs[i]} | Predicted Output: {predictions[i]}=>{np.round(predictions[i])},{np.round(predictions[i]) == outputs[i]}")

    # Calculate accuracy
    accuracy = np.mean(np.round(predictions) == outputs) * 100
    print(f"Accuracy on training data: {accuracy}%")

    plt.plot(range(num_epochs), loss_list)
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.show()

if __name__ == "__main__":
    main()

```

draw_plot.py

在不同的隐藏层神经元数量，不同的学习率下进行测试，该文件实现的功能：给出若干学习率和不同的隐含层神经元数量组合进行训练，考虑到每次训练测试的准确率有所波动，对于每一对学习率和隐含层神经元数量的组，训练十次，测试十次准确率取平均值作为最终的准确率。

每一张图表都会显示十个损失函数曲线，图表的标题显示对应的隐藏神经元数量，学习率，模型的准确率。

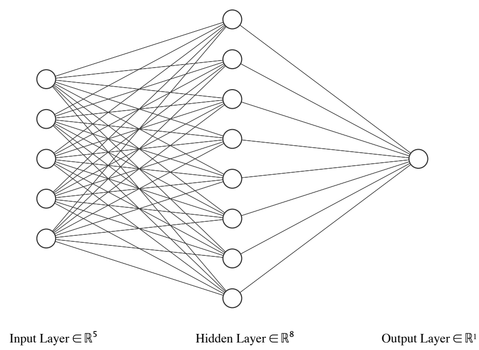
```python
import numpy as np
import matplotlib.pyplot as plt
from tqdm import tqdm
from utils import *

n_parity = 4
inputs, outputs = generate_parity_dataset(4)

learning_rates = [0.5, 1.0, 1.5, 2, 2.5, 3.0]
hidden_neurons_list = [4, 6, 8, 10, 12, 14]
num_epochs = 5000

num_repeats = 10


fig, axes = plt.subplots(len(learning_rates), len(hidden_neurons_list), figsize=(15, 10), sharex=True, sharey=True)
fig.tight_layout(pad=4.0)

total_combinations = len(learning_rates) * len(hidden_neurons_list) * num_repeats
progress_bar = tqdm(total=total_combinations, desc="Training progress")

for i, lr in enumerate(learning_rates):
    for j, hidden_neurons in enumerate(hidden_neurons_list):
        accuracies = []
        max_loss = -np.inf
        min_loss = np.inf
        for repeat in range(num_repeats):
            input_size = inputs.shape[1]
            output_size = outputs.shape[1]
            weights_input_hidden = np.random.uniform(-1, 1, size=(input_size, hidden_neurons))
            weights_hidden_output = np.random.uniform(-1, 1, size=(hidden_neurons, output_size))

            weights_input_hidden, weights_hidden_output, loss_list = train(inputs, outputs, weights_input_hidden, weights_hidden_output, lr, num_epochs)

            axes[i, j].plot(range(num_epochs), loss_list, alpha=0.3)
```

```
            max_loss = max(max_loss, np.max(loss_list))
            min_loss = min(min_loss, np.min(loss_list))

            predictions = test(inputs, weights_input_hidden, weights_hidden_output)
            accuracy = np.mean(np.round(predictions) == outputs)
            accuracies.append(accuracy)

            progress_bar.update(1)

        mean_accuracy = np.mean(accuracies)
        axes[i, j].set_title(f"LR:{lr}, Hidden:{hidden_neurons}, Acc:{mean_accuracy:.2f}")
        axes[i, j].set_ylim(min_loss, max_loss)
        axes[i, j].set_xlabel("Epoch")
        axes[i, j].set_ylabel("Loss")

progress_bar.close()
plt.show()
```

5.结果讨论

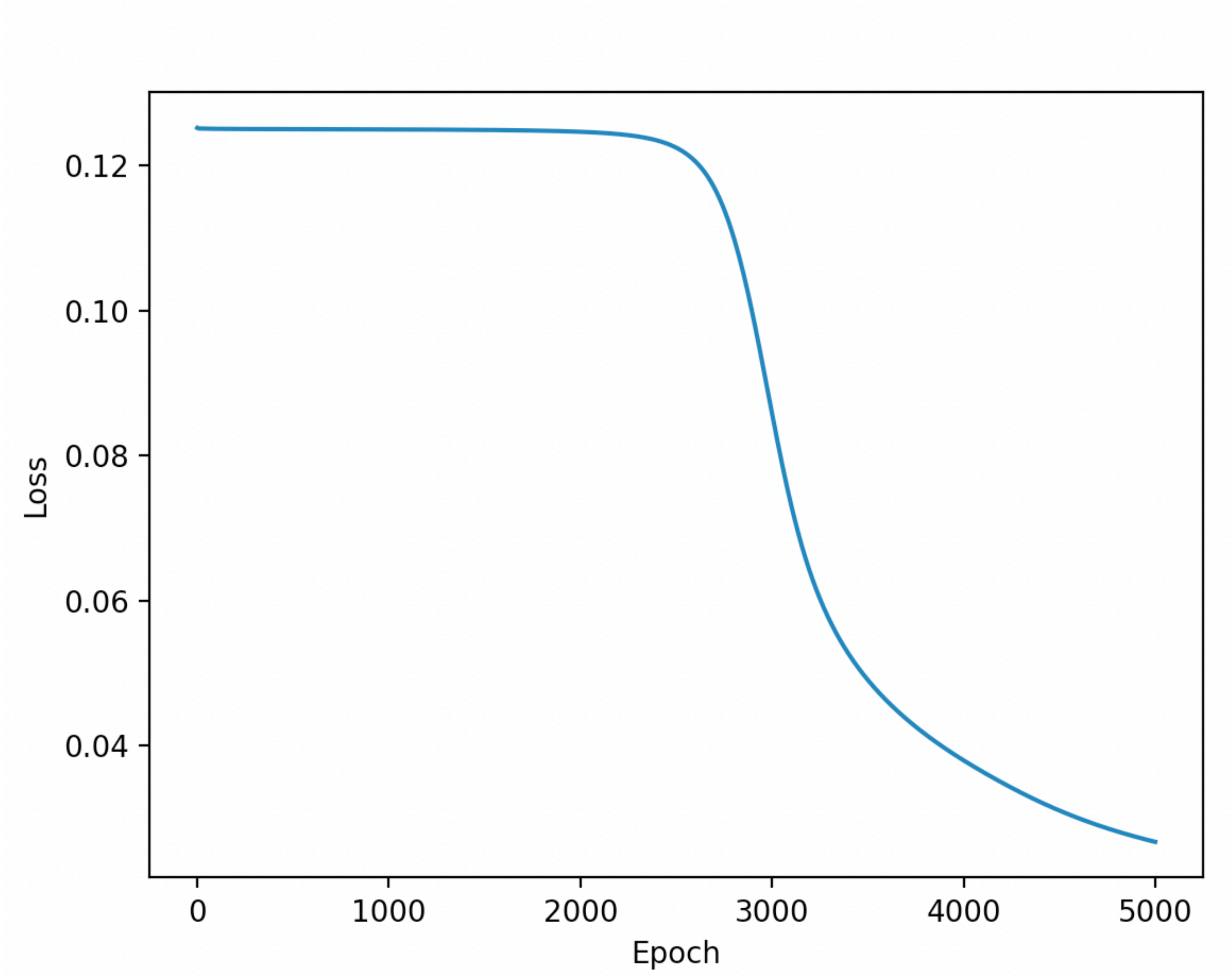
```
hidden_neurons = 8
learning_rate = 2
num_epochs = 5000
```

用以上参数测试，神经网络结构如下：



1.project2

使用[project2.py](project2.py)，输出结果如下，首先是损失函数图像



可以看出在某一阶段Loss值明显下降，说明模型学到了数据之间的规律

之后程序会输出模型的权重矩阵

```
Training complete.
Weights from input layer to hidden layer:
[[-2.31644425 -0.63039741 -3.57731934 -1.04625514 -1.25227723 -6.18501899
   1.431915    4.77349946]
 [-2.15748164 -0.31308126 -3.57955329 -0.65912048 -0.5021715  -6.14898936
   1.47622154  4.73967536]
 [ 0.32974164 -0.05176128  3.95640313 -0.64162932 -0.16245872  5.72925796
  -1.64498763 -4.69839852]
 [ 1.48133044  0.05978481 -1.31046318 -0.45520148 -0.92089128  5.43126401
   3.67404465 -4.12666975]
 [ 1.14864812  0.63524123  0.44982406  0.6552721   0.81828722  2.13236256
   0.3013925   1.47702114]]
Weights from hidden layer to output layer:
[[ 2.74413683]
 [ 0.79679478]
 [ 5.64768705]
```

```
 [-0.90614131]
 [-1.00069039]
 [-8.97386301]
 [ 4.40344738]
 [-5.17288463]]
```

再之后程序会采用训练集作为测试数据，输出模型输出值和训练集的期望值，同时计算准确率

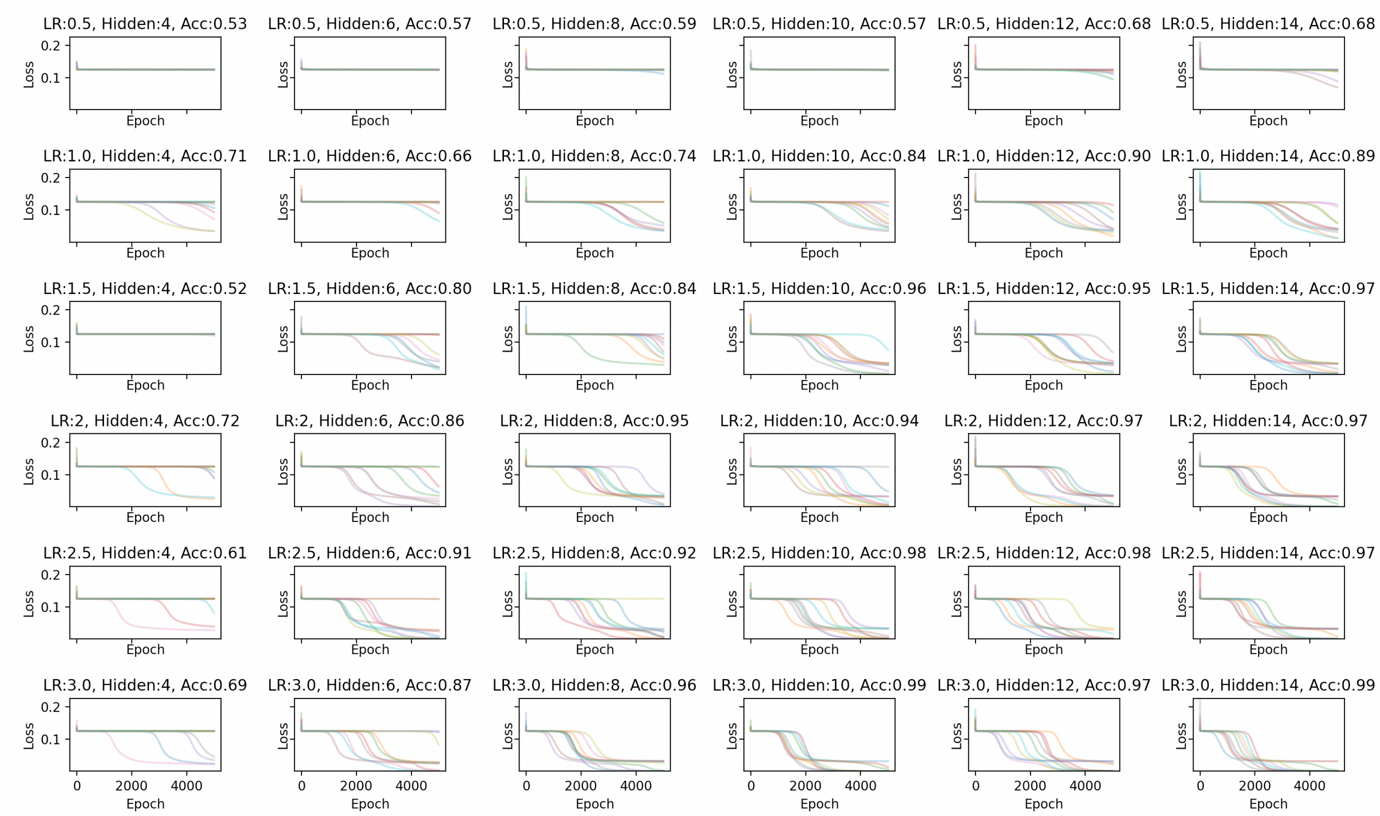
```
Test the accuracy:

Input: [ 0  0  0  0 -1] | Desired Output: [1] | Predicted Output: [0.9268844]=>[1.],[ True]
Input: [ 0  0  0  1 -1] | Desired Output: [0] | Predicted Output: [0.19078016]=>[0.],[ True]
Input: [ 0  0  1  0 -1] | Desired Output: [0] | Predicted Output: [0.08886483]=>[0.],[ True]
Input: [ 0  0  1  1 -1] | Desired Output: [1] | Predicted Output: [0.94171521]=>[1.],[ True]
Input: [ 0  1  0  0 -1] | Desired Output: [0] | Predicted Output: [0.04732206]=>[0.],[ True]
Input: [ 0  1  0  1 -1] | Desired Output: [1] | Predicted Output: [0.91286132]=>[1.],[ True]
Input: [ 0  1  1  0 -1] | Desired Output: [1] | Predicted Output: [0.89465911]=>[1.],[ True]
Input: [ 0  1  1  1 -1] | Desired Output: [0] | Predicted Output: [0.04239285]=>[0.],[ True]
Input: [ 1  0  0  0 -1] | Desired Output: [0] | Predicted Output: [0.16491239]=>[0.],[ True]
Input: [ 1  0  0  1 -1] | Desired Output: [1] | Predicted Output: [0.19063355]=>[0.],[False]
Input: [ 1  0  1  0 -1] | Desired Output: [1] | Predicted Output: [0.94086615]=>[1.],[ True]
Input: [ 1  0  1  1 -1] | Desired Output: [0] | Predicted Output: [0.26320072]=>[0.],[ True]
Input: [ 1  1  0  0 -1] | Desired Output: [1] | Predicted Output: [0.90727252]=>[1.],[ True]
Input: [ 1  1  0  1 -1] | Desired Output: [0] | Predicted Output: [0.25959473]=>[0.],[ True]
Input: [ 1  1  1  0 -1] | Desired Output: [0] | Predicted Output: [0.05040591]=>[0.],[ True]
Input: [ 1  1  1  1 -1] | Desired Output: [1] | Predicted Output: [0.88746894]=>[1.],[ True]
Accuracy on training data: 93.75%
```

2.draw plot

同时训练多个模型以发现规律，接下来使用draw_plot.py，采用如下不同的隐含层神经元数量

```
learning_rates = [0.5, 1.0, 1.5, 2, 2.5, 3.0]
hidden_neurons_list = [4, 6, 8, 10, 12, 14]
num_epochs = 10000
```

输出图像如下:



基本可以看出，在一定范围内，隐藏层神经元数量越多，学习率越高，准确率越高，模型性能越好

3.问题

问题：损失函数值在0.125和0.03的时候在很长一段epoch里都几乎平稳不下降，这是为什么

猜想：损失函数等于这俩个值的时候，神经网络陷入了局部最优解的阶段，因此梯度较小，权重更新很慢，损失函数就会几乎处于一个定值。继续训练，圣经网络跳出局部最优解，梯度增大，权重开始更新，损失函数值继续下降。

4.尝试使用更多隐含层

代码见more_hidden_neurous.py结果并没有比单层神经网络好