

学习笔记

Zhengjv

2021 年 6 月 4 日

目录

| | | |
|----------|---------------------------|----------|
| 1 | 深度学习 | 2 |
| 1.1 | 神经网络基础学习 | 2 |
| 1.1.1 | 人工神经网络 | 2 |
| 1.1.2 | 前向传播 | 2 |
| 1.1.3 | 损失函数 | 2 |
| 1.1.4 | 梯度下降 | 3 |
| 1.1.5 | 反向传播 | 3 |
| 1.2 | 实验 (基于 pytorch) | 3 |
| 1.2.1 | 利用神经网络进行回归分析 | 3 |
| 1.2.2 | 利用神经网络进行简单分类 | 5 |
| 2 | 刷题笔记 | 7 |
| 2.1 | 回文数 | 7 |
| 2.1.1 | 常规解法 | 8 |
| 2.1.2 | 不转换字符串解法 | 8 |
| 2.2 | 正则表达式匹配 | 9 |
| 2.2.1 | 解决方法 | 10 |
| 2.3 | 盛最多水的容器 | 12 |
| 2.3.1 | 解决方法 | 12 |
| 2.4 | 整数转罗马数字 | 13 |
| 2.4.1 | 解决方法 | 13 |
| 2.4.2 | 优化方法 | 14 |

| | | |
|-------|--------------------|----|
| 2.5 | 罗马数字转整数 | 15 |
| 2.5.1 | 解决方法 | 15 |
| 2.6 | 最长公共前缀 | 15 |
| 2.6.1 | 纵向扫描 | 16 |
| 2.7 | 三数之和 | 17 |
| 2.7.1 | 解决方法 | 17 |
| 2.8 | 最接近的三数之和 | 19 |
| 2.8.1 | 解决方法 | 19 |

1 深度学习

1.1 神经网络基础学习

1.1.1 人工神经网络

人工神经网络是一种模仿人类大脑的工作机制的系统，通过对系统内参数的更新，从而达到人们想得到的最优效果。简而言之，它的作用就是，给予若干个输入的特征 x ，就能得到输出的预测值 y 。而中间过程隐藏单元都相当于是一个黑盒子，它会自动完成相应的计算，得到从 x 到 y 的精准映射函数，我们只需要喂给它足够多的数据，诸如 (x, y) 的训练样本。只要数据足够，就可得出好效果的网络模型。当然，由于非线性的激活函数的存在，神经网络可以将许多线性变换和非线性变化叠加起来，使得其可以完成几乎任何任务。

1.1.2 前向传播

前向传播负责在神经网络中将输入层层运算并逐层加权计算出输出。其本质就是利用非线性激活函数改变输入矩阵与权重矩阵的相乘结果、并将其结果作为输出传递给下一层。

1.1.3 损失函数

当使用前向传播算法逐层计算到输出层，获得输出层的结果之后，接下来神经网络要做的就是获得学习到的信息，然后通过这些信息来更新网络中的权重参数。通常，我们用一个非负的实值函数来表示，即为损失函数。主要用来表示输出层的输出结果与真实结果之间的差距。当这个差距越小

时,说明神经网络模型效果越好。训练网络的过程,则是不断减小损失以获得更好模型效果的过程。

1.1.4 梯度下降

通过损失函数得到了预测值与真实值差距后,我们要最小化这个差距,通常采用梯度下降法。其思想是,求得初始点的梯度,向着梯度相反的方向移动,如梯度很大,则大步前进,如梯度很小则慢步前进,通过设置学习率来控制前进的步长,以此迭代靠近梯度为零的点,即最小值点(或局部最小点)。如跨过了极值点,由于梯度方向改变,移动方向也会改变,因此可以慢慢逼近极小值点。同时我们得到最优的一组参数。如学习率设置过大,导致步长过大,可能会在极值点附近振荡左右摆动,无法达到极小值点,如学习率设置过小,导致步长过小,可能会使达到极小值点的时间过长,使训练速度过于缓慢。

1.1.5 反向传播

前面已经介绍过神经网络的一般训练过程,分别通过前向传播算法计算预测输出,损失函数计算差距,梯度下降去优化损失函数。梯度下降是真正的学习算法,而反向传播算法只是计算梯度的方法。可以用来对任意函数求导。它使用链式法则从后往前推导,从而简化神经网络节点的梯度计算。最后梯度下降算法使用反向传播计算出的梯度去更新网络参数。

1.2 实验(基于 pytorch)

1.2.1 利用神经网络进行回归分析

代码

```
import torch
import numpy as np
from torch.autograd import Variable
import torch.nn.functional as f
import matplotlib.pyplot as plt
x = torch.unsqueeze(torch.linspace(-1,1,100),dim=1)
y=x.pow(2)+0.2*torch.rand(x.size())
x=Variable(x)
```

```

y=Variable(y) #variable 已弃用
x_np=x.data.numpy()
y_np=y.data.numpy()
plt.scatter(x.data.numpy(), y.data.numpy())
plt.show()
class Net(torch.nn.Module):
    def __init__(self, n_features, n_hidden, n_output):
        super(Net, self).__init__()
        self.hidden = torch.nn.Linear(n_features, n_hidden) #隐藏层
        self.predict = torch.nn.Linear(n_hidden, n_output)

    def forward(self, x): #网络搭建
        x = f.relu(self.hidden(x))
        x = self.predict(x)
        return x

net=Net(1,10,1) #10个隐藏层的神经网络
print(net)
optimizer = torch.optim.SGD(net.parameters(), lr=0.5) #梯度更新方法
loss_func = torch.nn.MSELoss() #loss函数选择MSE
for t in range(500):
    prediction = net(x) # 利用神经网络预测x值
    loss = loss_func(prediction, y) #计算真实值和预测值的 loss
    optimizer.zero_grad() # 梯度清零
    loss.backward() # 反向传播
    optimizer.step() # 梯度更新

```

```

D:\aconada\envs\python36\python.exe E:/pytorch01/回归.py
Net(
  (hidden): Linear(in_features=1, out_features=10, bias=True)
  (predict): Linear(in_features=10, out_features=1, bias=True)
)

```

图 1: 网络结构.

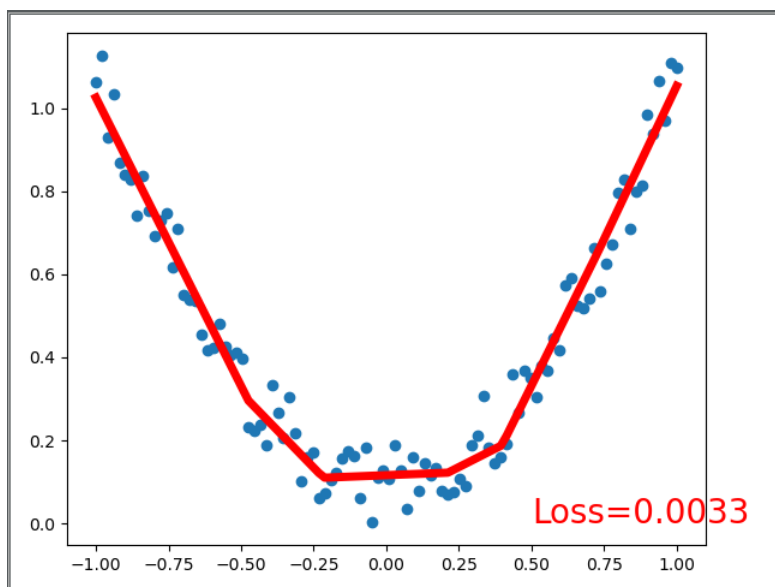


图 2: 回归结果

结果

1.2.2 利用神经网络进行简单分类

代码

```
import torch
import numpy as np
from torch.autograd import Variable
import torch.nn.functional as F
import matplotlib.pyplot as plt

n_data = torch.ones(100, 2)
x0 = torch.normal(2*n_data, 1)
y0 = torch.zeros(100)
x1 = torch.normal(-2*n_data, 1)
y1 = torch.ones(100)
x = torch.cat((x0, x1), 0).type(torch.FloatTensor)
y = torch.cat((y0, y1), ).type(torch.LongTensor)
```

```

x, y = Variable(x), Variable(y)
print(y)
plt.scatter(x.data.numpy()[ :, 0],
            x.data.numpy()[ :, 1], c=y.data.numpy(), s=100, lw=0, cmap='RdYlGn')
plt.show()
class Net(torch.nn.Module):
    def __init__(self, n_features, n_hidden, n_output):
        super(Net, self).__init__()
        self.hidden = torch.nn.Linear(n_features, n_hidden)
        self.predict = torch.nn.Linear(n_hidden, n_output)
    def forward(self, x):
        x = F.relu(self.hidden(x))
        x = self.predict(x)
        return x
net = Net(n_features=2, n_hidden=10, n_output=2)
print(net)
optimizer = torch.optim.SGD(net.parameters(), lr=0.02)
loss_func = torch.nn.CrossEntropyLoss()
# 交叉熵损失
for t in range(100):
    out = net(x)
    loss = loss_func(out, y)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

```

```

Net(
  (hidden): Linear(in_features=2, out_features=10, bias=True)
  (predict): Linear(in_features=10, out_features=2, bias=True)
)

```

图 3: 网络结构.

结果

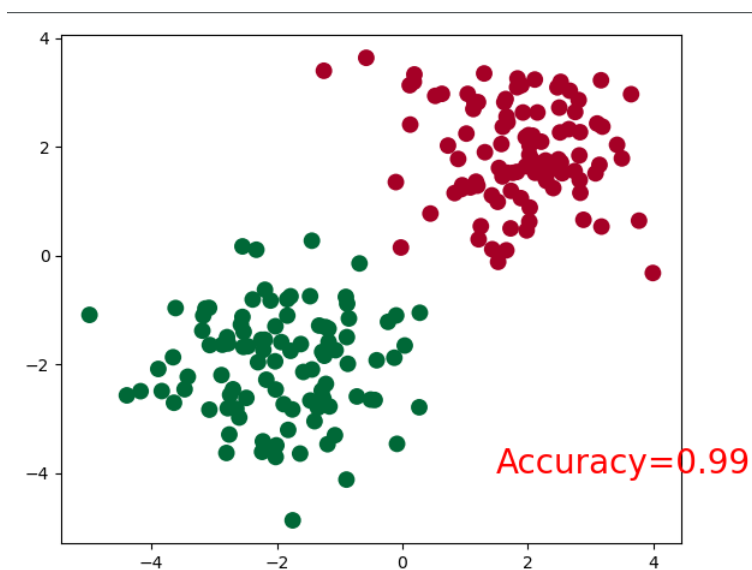


图 4: 回归结果

2 刷题笔记

2.1 回文数

给你一个整数 x ，如果 x 是一个回文整数，返回 `true`；否则，返回 `false`。

回文数是指正序（从左向右）和倒序（从右向左）读都是一样的整数。例如，121 是回文，而 123 不是。

示例 1:

输入: $x = 121$

输出: `true`

示例 2:

输入: $x = -121$

输出: `false`

解释: 从左向右读, 为 -121 。从右向左读, 为 $121-$ 。因此它不是一个回文数。

示例 3:

输入: $x = 10$

输出: false

解释: 从右向左读, 为 01 。因此它不是一个回文数。

示例 4:

输入: x = -101

输出: false

2.1.1 常规解法

思路 看到题目, 首先负数不可能为回文, 则先将负数排除, 然后将其转换为字符串。比较正序字符串数组和倒序字符串数组是否相同, 若相同, 则是一个回文数。

代码

```
class Solution(object):
    def isPalindrome(self, x):
        if x < 0:
            temp = False
        else:
            st = str(x)
            st2 = st[::-1]
            if st == st2:
                temp = True
            else:
                temp = False
        return temp
```

2.1.2 不转换字符串解法

思路 若不转换为字符串, 则需要不断地取余取整操作将不同数位上的数字分离。此时若数字太大则会产生溢出, 因此将其分为两半, 首先排除负数, 和除 0 以外个位是 0 的数字。其次定义一个新数字, 表示后半部分翻转后的数字, 即为将每次取余得到的数字翻转, 将原始数字逐位减小, 当原始数字小于新数字时停止。此时再来判断不奇偶情况下两数字是否相同。从而判断是否为回文。

代码

```
class Solution(object):
    def isPalindrome(self, x):
        reverNumber=0
        if x<0 or (x%10==0 and x!=0):
            return False
        else:
            while x>reverNumber:
                reverNumber=reverNumber*10+x%10
                x=x/10
            return x==reverNumber or x==reverNumber/10
```

2.2 正则表达式匹配

给你一个字符串 s 和一个字符规律 p ，请你来实现一个支持 '.' 和 '*' 的正则表达式匹配。 '.' 匹配任意单个字符 '*' 匹配零个或多个前面的那一个元素所谓匹配，是要涵盖整个字符串 s 的，而不是部分字符串。

示例 1:

输入: $s = "aa"$ $p = "a"$

输出: false

解释: "a" 无法匹配 "aa" 整个字符串。

示例 2:

输入: $s = "aa"$ $p = "a^*"$

输出: true

解释: 因为 '*' 代表可以匹配零个或多个前面的那一个元素, 在这里前面的元素就是'a'。因此，字符串"aa" 可被视为'a' 重复了一次。

示例 3:

输入: $s = "ab"$ $p = ".*"$

输出: true

解释: ".*" 表示可匹配零个或多个 ('*') 任意字符 ('.')。

示例 4:

输入: $s = "aab"$ $p = "c^*a^*b"$

输出: true

解释：因为'*' 表示零个或多个，这里'c' 为 0 个，'a' 被重复一次。因此可以匹配字符串"aab"。

示例 5:

输入: s = "mississippi" p = "mis*is*p*."

输出: false

2.2.1 解决方法

思路 动态规划，定义动态数组 $dp[m][n]$ ，确定初始状态。

确定动态转移方程：

$$f[i][j] = \begin{cases} \text{if } (p[j] \neq '*') = \begin{cases} f[i-1][j-1], & \text{matches}(s[i], p[j]) \\ \text{false}, & \text{otherwise} \end{cases} \\ \text{otherwise} = \begin{cases} f[i-1][j] \text{ or } f[i][j-2], & \text{matches}(s[i], p[j-1]) \\ f[i][j-2], & \text{otherwise} \end{cases} \end{cases}$$

图 5: 状态转移方程

确定边界条件：首先我们要确定 $dp[0][0]$ ，当 p 为空,s 为空时，肯定是匹配成功的！那么 $dp[0][0]=\text{true}$

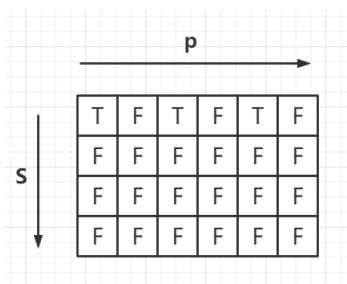
当 p 为空字符串，而 s 不为空时，dp 数组必定为 False，正好初始化 dp 数组的时候设置的是 False；即 dp 数组的第一列为 False 可以确定

当 s 为空字符串，而 p 不为空时，我们无需判断 p 里面的第一个值是否为""，如果为""，那肯定匹配不到为 False，原数组正好是 False，所以直接从 2 开始判断即可。如果遇到了*，只要判断其对应的前面两个元素的 dp 值

代码

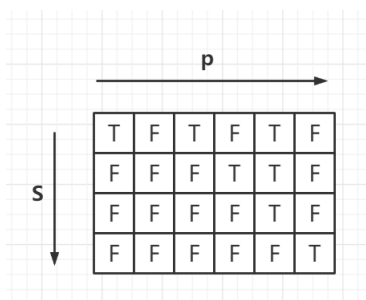
`class Solution:`

```
def isMatch(self, s: str, p: str) -> bool:
    m, n = len(s) + 1, len(p) + 1
    dp = [[False] * n for _ in range(m)]
    dp[0][0] = True
    # 初始化首行
```



| | | | | | | |
|--|---|---|---|---|---|---|
| | T | F | T | F | T | F |
| | F | F | F | F | F | F |
| | F | F | F | F | F | F |
| | F | F | F | F | F | F |

图 6: 初始化后的 dp 数组为



| | | | | | | |
|--|---|---|---|---|---|---|
| | T | F | T | F | T | F |
| | F | F | F | T | T | F |
| | F | F | F | F | T | F |
| | F | F | F | F | F | T |

图 7: 最终结果

```

for j in range(2, n, 2):
    dp[0][j] = dp[0][j - 2] and p[j - 1] == '*'
# 状态转移
for i in range(1, m):
    for j in range(1, n):
        if p[j-1] == '*':
            if dp[i][j - 2]: dp[i][j] = True
            elif dp[i-1][j] and s[i-1]==p[j-2]: dp[i][j] = True
            elif dp[i-1][j] and p[j-2]=='.': dp[i][j] = True
        else:
            if dp[i-1][j-1] and s[i-1] == p[j-1]: dp[i][j] = True
            elif dp[i-1][j-1] and p[j-1] == '.': dp[i][j] = True
    return dp[-1][-1]
s=Solution()
print(s.isMatch("saaa","a.*"))

```

2.3 盛最多水的容器

给你 n 个非负整数 a_1, a_2, \dots, a_n ，每个数代表坐标中的一个点 (i, a_i) 。在坐标内画 n 条垂直线，垂直线 i 的两个端点分别为 (i, a_i) 和 $(i, 0)$ 。找出其中的两条线，使得它们与 x 轴共同构成的容器可以容纳最多的水。

说明：你不能倾斜容器。

2.3.1 解决方法

思路 双指针，初始时指针分别在左右两端，每次都移动数字较小的那个指针（假设左端数字最小，那么无论右端如何移动，最大容量即是在右边指针在最右端时），双指针代表的是可以作为容器边界的所有位置的范围。一开始，双指针指向数组的左右边界，表示数组中所有的位置都可以作为容器的边界，因为我们还没有进行过任何尝试。在这之后，我们每次将对应的数字较小的那个指针往另一个指针的方向移动一个位置，就表示我们认为这个指针不可能再作为容器的边界了。那么，只要经过比较使得每次移动的指针都是较小的那一个，便可以得出系列最大容量，最后对移动中产生的最大容量作比较即可。

代码

```
class Solution(object):
    def maxArea(self, height):

        maxv=0
        i=0
        j=len(height)-1
        while i!=j:
            volume = min(height[i],height[j])*(j-i)
            if height[i]<height[j]:
                i=i+1
            else:
                j=j-1
            if maxv<volume:
                maxv=volume
        return maxv
```

2.4 整数转罗马数字

罗马数字包含以下七种字符：I，V，X，L，C，D 和 M。

通常情况下，罗马数字中小的数字在大的数字的右边。但也存在特例，例如 4 不写做 IIII，而是 IV。数字 1 在数字 5 的左边，所表示的数等于大数 5 减小数 1 得到的数值 4。同样地，数字 9 表示为 IX。这个特殊的规则只适用于以下六种情况：

I 可以放在 V (5) 和 X (10) 的左边，来表示 4 和 9。

X 可以放在 L (50) 和 C (100) 的左边，来表示 40 和 90。

C 可以放在 D (500) 和 M (1000) 的左边，来表示 400 和 900。

给你一个整数，将其转为罗马数字。

2.4.1 解决方法

思路 对数字进行递减，让数字与罗马字母所代表的数值从大到小逐一比较。大于罗马数字时，则减去其所代表的值，并将罗马字符添加入列表中，直到 numbers 小于当前罗马字符所代表数值时结束，跳出循环。进入下一个更小的字符。

代码

```
class Solution:
```

```
    VALUE_SYMBOLS = [  
        (1000, "M"),  
        (900, "CM"),  
        (500, "D"),  
        (400, "CD"),  
        (100, "C"),  
        (90, "XC"),  
        (50, "L"),  
        (40, "XL"),  
        (10, "X"),  
        (9, "IX"),  
        (5, "V"),  
        (4, "IV"),
```

```

        (1, "I"),
    ]

    def intToRoman(self, num):
        roman = list()
        for value, symbol in Solution.VALUE_SYMBOLS:
            while num >= value:
                num -= value
                roman.append(symbol)
            if num == 0:
                break
        return "".join(roman)

```

2.4.2 优化方法

思路 对罗马字符数字之间建立编码表。

代码

```

class Solution:

    THOUSANDS =
        ["", "M", "MM", "MMM"]
    HUNDREDS =
        ["", "C", "CC", "CCC", "CD", "D", "DC", "DCC", "DCCC", "CM"]
    TENS =
        ["", "X", "XX", "XXX", "XL", "L", "LX", "LXX", "LXXX", "XC"]
    ONES =
        ["", "I", "II", "III", "IV", "V", "VI", "VII", "VIII", "IX"]

    def intToRoman(self, num: int) -> str:
        return Solution.THOUSANDS[num // 1000] + \
            Solution.HUNDREDS[num % 1000 // 100] + \
            Solution.TENS[num % 100 // 10] + \
            Solution.ONES[num % 10]

```

2.5 罗马数字转整数

给定一个罗马数字，将其转换成整数。输入确保在 1 到 3999 的范围内。

2.5.1 解决方法

思路 不同于整数转罗马数字，因为涉及罗马数字会有两个字符代表一个数字的情况，所以不好遍历。可以在计算整数时将其忽略掉。具体的方法为，依然每次遍历一个字母，但是每次和下一位进行比较，很容易得出当前是 IV or VI，若比下一位小，则整体的数值将当前的数值减去即可。

代码

```
class Solution(object):
    def romanToInt(self, s):
        """
        :type s: str
        :rtype: int
        """
        dct = {'M':1000, 'D':500, 'C':100, 'L':50, 'X':10, 'V':5, 'I':1}
        i, res = 0, 0
        while i < len(s)-1:
            if dct[s[i]] < dct[s[i+1]]:
                res -= dct[s[i]]
            else:
                res += dct[s[i]]
            i += 1
        return res+dct[s[-1]]
```

2.6 最长公共前缀

编写一个函数来查找字符串数组中的最长公共前缀。
如果不存在公共前缀，返回空字符串""。

2.6.1 纵向扫描

思路 每次遍历列表中每个字符串的一个字母，比较它们是否相同，若不同则结束循环，若相同则继续往下比较。

不同时，前面的字符即为公共前缀。另外，会需要几种特殊情况进行处理，当列表中的字符串只有一个时。

另外，还需注意遍历时候的长度，需要以最短的字符串作为便利的长度。

代码

```
class Solution(object):
    def longestCommonPrefix(self, strs):
        k,t=0,0
        if len(strs)==1:
            return strs[0]
        else:
            while True:
                if k<min(len(i) for i in strs):
                    temp=strs[0][k]
                    for s in strs:
                        if s[k]==temp:
                            temp=s[k]
                        else:
                            t=1
                            break
                    if t==1:
                        break
                    k = k + 1
                else:
                    break
            if k==0:
                return ''
            else:
```



```
return strs[0][:k]
```

2.7 三数之和

给你一个包含 n 个整数的数组 `nums`，判断 `nums` 中是否存在三个元素 a, b, c ，使得 $a + b + c = 0$ ？请你找出所有和为 0 且不重复的三元组。

注意：答案中不可以包含重复的三元组。

2.7.1 解决方法

思路 为了减小运算量，将其拆分为正负两组数并对其分别进行计数（这样做的好处是减少运算量，并且进行了一部分剔除重）。在排序后，对第一组数遍历，并对第二组数使用双指针遍历，使得第二组数中两数之和等于第一组正在遍历的数时结束，将其添加。另外，因为数字已经有序，可设定在相等之后又遇到重复数字时直接结束当前循环，又进行一部分剔除重。

代码

```
class Solution(object):
```

```
    def iszero(self, nums):
```

```
        T=0
```

```
        for i in nums:
```

```
            if i!=0:
```

```
                T=1
```

```
        if T==0:
```

```
            return True
```

```
        else:
```

```
            return False
```

```
    def threeSum(self, nums):
```

```
        """
```

```
        :type nums: List[int]
```

```
        :rtype: List[List[int]]
```

```
        """
```

```
        if self.iszero(nums) and len(nums)>=3:
```

```
        return [[0,0,0]]
    elif len(nums)<3:
        return []
    elif len(nums)==3:
        if sum(nums)==0:
            return [nums]
        else:
            return []
    else:

        res=[]
        a=[]
        b=[]

        for i in nums:
            if i<0:
                a.append(-i)
            else:
                b.append(i)
        b.append(999999)
        a.append(111111)
        a.sort()
        b.sort()

        for q,i in enumerate(a):
            if i==a[q-1]:
                continue
            j, k = 0, len(b) - 1
            while j<k:
                if b[j]+b[k]<i:
                    j = j + 1
                elif b[j]+b[k]>i:
```

```

        k = k - 1
    else:
        res.append([b[j], b[k], -i])
        j = j + 1
        k = k - 1
for q, i in enumerate(b):
    if i == b[q-1]:
        continue
    j, k = 0, len(a) - 1
    while j < k:
        if a[j] + a[k] < i:
            j = j + 1
        elif a[j] + a[k] > i:
            k = k - 1
        else:
            res.append([-a[j], -a[k], i])
            j = j + 1
            k = k - 1

return res

```

2.8 最接近的三数之和

给定一个包括 n 个整数的数组 `nums` 和一个目标值 `target`。找出 `nums` 中的三个整数，使得它们的和与 `target` 最接近。返回这三个数的和。假定每组输入只存在唯一答案。

2.8.1 解决方法

思路 依然可以采用双指针的办法，考虑一个元素 a ，对于剩下的两个元素 b 和 c ，希望它们的和最接近 $target - a$ 。使用双指针之前，我们依然需要对数组先进行排序，

在进行枚举时，为了避免重复，将 a 所在位置跳过。初始时，左指针指向位置 $i+1$ ，即左边界；右指针指向位置 $n-1$ ，即右边界。每次比较 $a+b+c$

与 `target`，若大于，则移动左指针，

若小于，则移动右指针。由于数组已经有序，所以在编码的过程中，使其每次移动到下一个不相等的元素

代码

```
class Solution(object):
    def threeSumClosest(self, nums, target):
        nums.sort()
        # print(nums)
        n = len(nums)
        res = float("inf")
        for i in range(n):
            if i > 0 and nums[i] == nums[i - 1]:#妙啊
                continue
            left = i + 1
            right = n - 1
            while left < right:
                cur = nums[i] + nums[left] + nums[right]
                if cur == target: return target
                if abs(res - target) > abs(cur - target):
                    res = cur
                if cur > target:
                    right -= 1
                elif cur < target:
                    left += 1
        return res
```