一、 python基础

1. python类

Python 中使用 class 关键字来定义类，类要遵循下述格式（模板）。

```
class 类名:
    def __init__(self, 参数, …): # 构造函数
        ...
    def 方法名1(self, 参数, …):   # 方法1
        ...
    def 方法名2(self, 参数, …):   # 方法2
        ...
```

这里有一个特殊的 __init__ 方法，这是进行初始化的方法，也称为**构造函数**（constructor），只在生成类的实例时被调用一次。此外，在方法的第一个参数中明确地写入表示自身（自身的实例）的 self 是 Python 的一个特点（学过其他编程语言的人可能会觉得这种写 self 的方式有一点奇怪）。

在上述类的方法当中，函数参数的 `self` 参数是必须要有，其余参数是可选的

2. Numpy库

这里的Numpy数组的算术运算，需要保持元素个数相同，即 element-wise product

element-wise product 对应元素相乘

当然，也可以让各个元素和单个标量元素进行运算 如 `x / 2` 就是整体元素都除以2

```
import numpy as np #导入numpy库
x = np.array([1,2,3])
y = np.array([2,4,18])
print(x + y)
print(x - y)
print(x * y)
print(x / y)
```

2.1 numpy的多维数组

矩阵的乘法

！注意！ `*` 是代表对应元素相乘，`numpy.dot()` 才是真正的矩阵相乘的结果

```
x = np.array([[1,2],[3,4]])
y = np.array([[1],[2]])
# y = np.array([[3,0],[0,6]])
out = np.dot(x,y)
print(out)
```

数学上将一维数组称为**向量**， 将二维数组称为**矩阵**。另外，可以将一般化之后的向量或矩阵等统 称为**张量（tensor）**

访问元素的其它方法

```
x = np.array([[1,2],[3,4]])
> x[0] #第0行
> x[0][1] #第0行第1列的元素
> for item in x #遍历元素
> x = x.flatten() #x转换为一维数组
> x[x>2] #输出所有>2的元素的值
```

实际上，如果是运算量大的处理对象，用 C/C++写程 序更好。为此，当 Python中追求性能时，人们会用 C/C++来实现 处理的内容。Python则承担“中间人”的角色，负责调用那些用 C/ C++写的程序。NumPy中，主要的处理也都是通过C或C++实现的。 因此，我们可以在不损失性能的情况下，使用Python便利的语法。

3. Matplotlib 数据可视化

显示函数图像

```
import numpy as np
import matplotlib.pyplot as plt
#生成数据
x  = np.arange(-6,6,0.1) #（0，6）以0.1为单位生成0到6的数据
y  = np.sin(x)
y1 = np.cos(x)
#绘制图像
plt.plot(x, y, label="sin")
plt.plot(x, y1, linestyle = "--", label="cos") # 用虚线绘制
plt.xlabel("x")
plt.ylabel("y")  #x轴和y轴标签
plt.title('sin & cos')  #标题
plt.legend() #加这个图列才能显示出来
plt.show()
```

显示图片

```
import matplotlib.pyplot as plt
from matplotlib.image import imread
img = imread('T:\Download\wallerpaper.jpg') # 读入图像（设定合适的路径！）
plt.imshow(img)
plt.show()
```

二、 感知机perceptron

感知机接收多个输入信号，输出一个信号。x1、x2是输入信号， y是输出信号，w1、w2是权重（w是weight的首字母）。图中的○称为“神 经元”或者“节点”。

$$y = \begin{cases} 0 & (w_1 x_1 + w_2 x_2 \leqslant \theta) \\ 1 & (w_1 x_1 + w_2 x_2 > \theta) \end{cases} \tag{2.1}$$

即达到一定阙值之后，神经元才会被激活

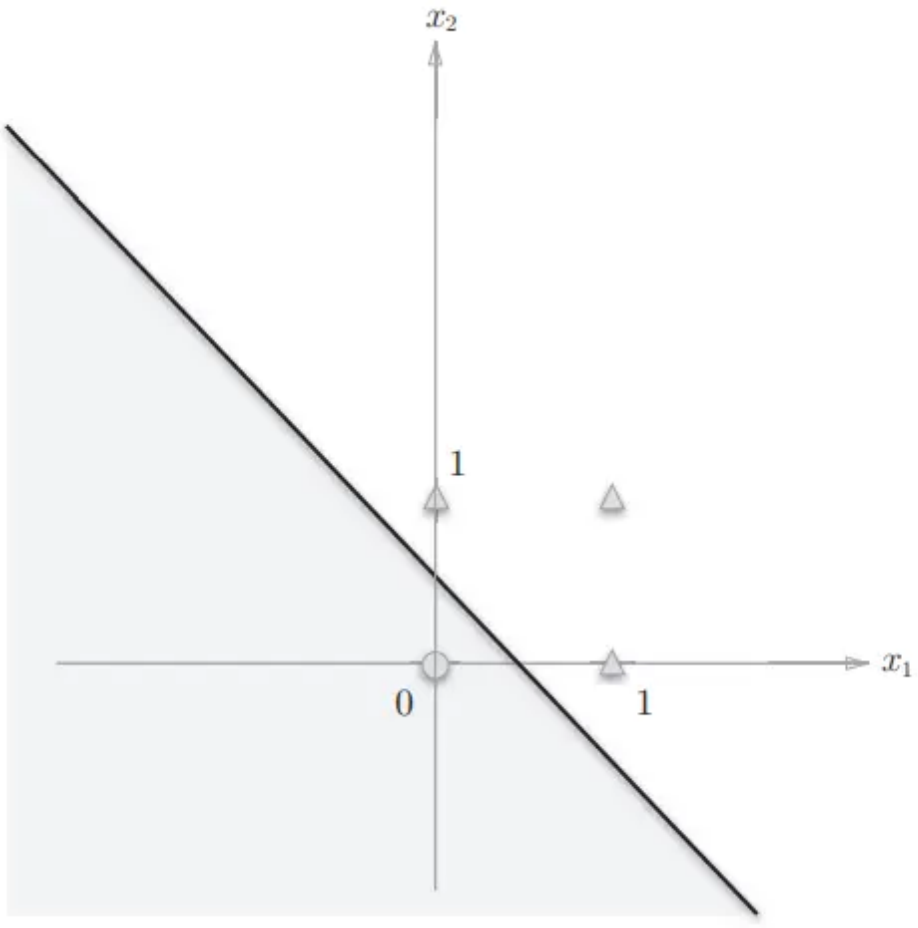
w1,w2表示参数的重要性，而偏置b表示的是整个神经元被激活的容易程度

为什么单层的感知机不能解决非线性问题

对于或门，一共四个点，我们可以在坐标轴当中用小三角表示取1的情况，小圆圈表示取0的情况

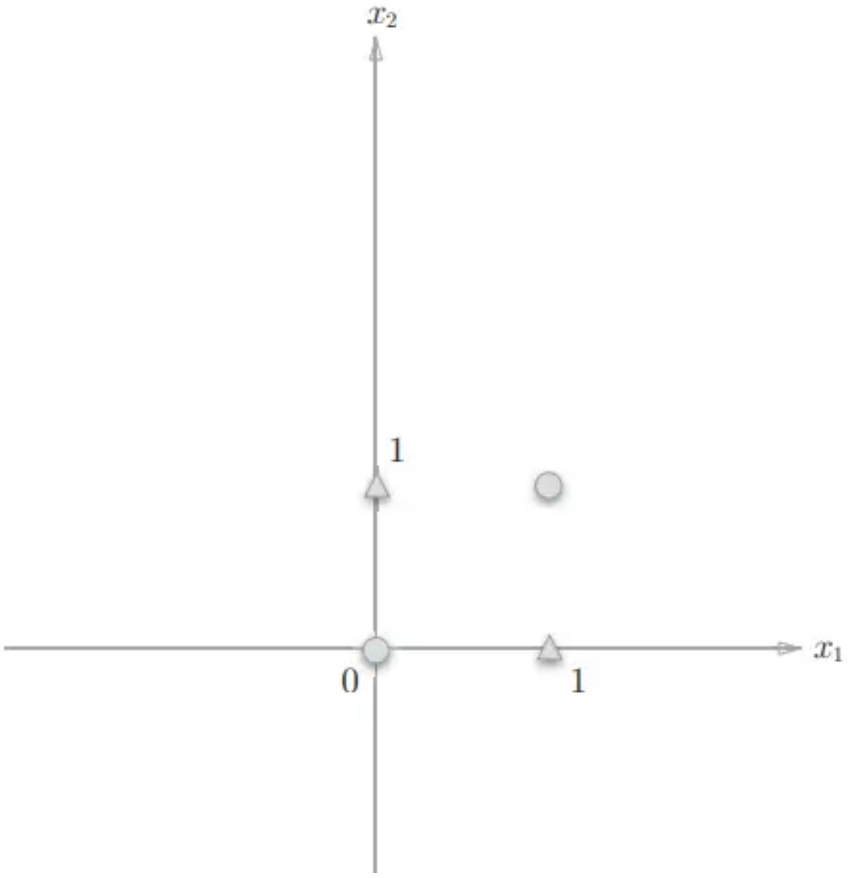
$$y = \begin{cases} 0 & (-0.5 + x_1 + x_2 \leqslant 0) \\ 1 & (-0.5 + x_1 + x_2 > 0) \end{cases} \tag{2.3}$$

可以在数轴下表示为如下的情况

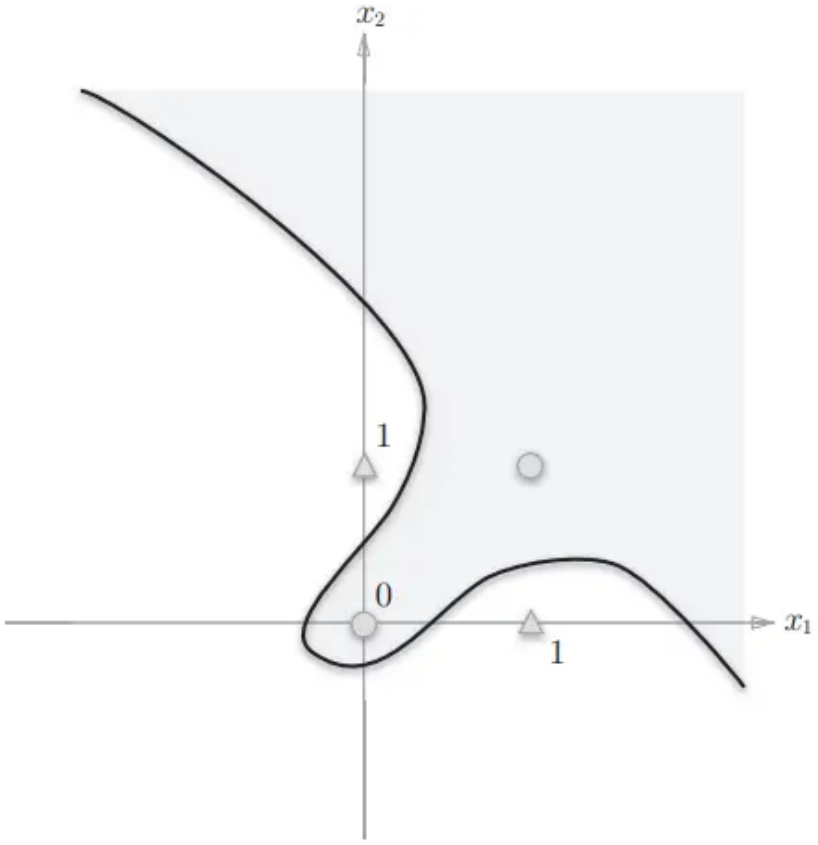


也就是可以用一条直线来划分区域，所以对于或门可以找到对应的参数w1,w2以及b满足条件

然而对于亦或，表示的点有如下情况



无法用一条直线来划分出两个区域，因此，这类问题**是线性不可分的问题，也就是非线性的**



感知机通过叠 加层能够进行非线性的表示，理论上还可以表示计算机进行的处理

三、神经网络

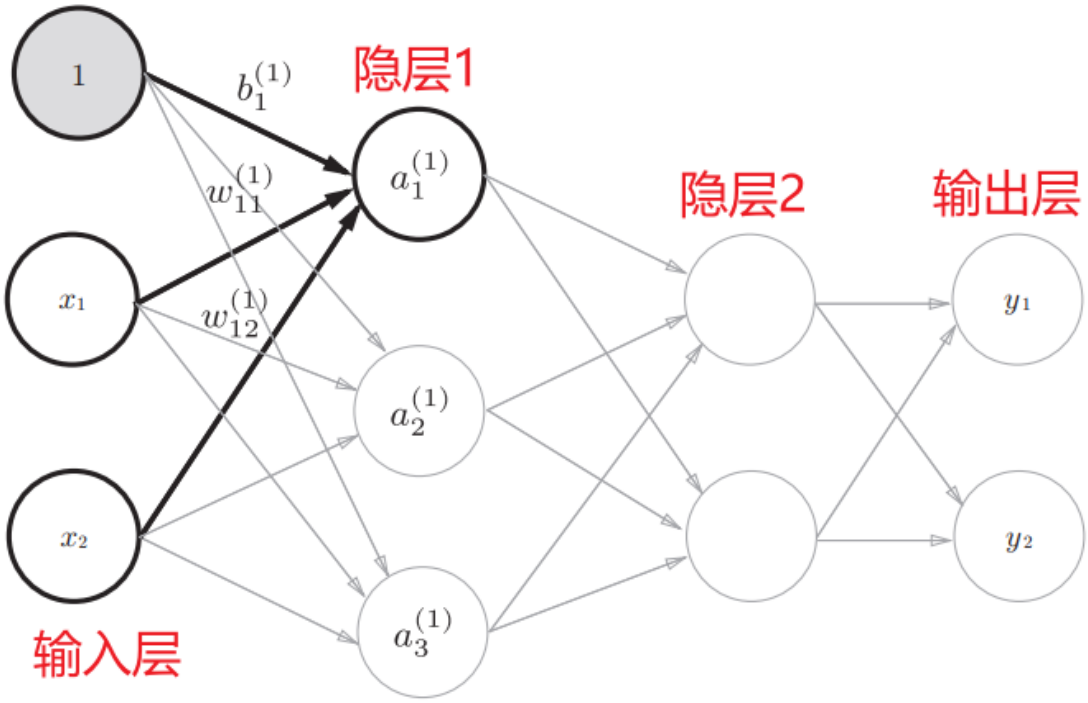
神经网络的一 个重要性质是它可以自动地从数据中学习到合适的权重参数。

神经网络的激活函数必须使用非线性函数,线性函数的问题在于，不管如何加深层数，总是存在与之等效的"无 隐藏层的神经网络"

把线性函数 h(x) = cx 作为激活 函数，把y(x) = h(h(h(x)))的运算对应3层神经网络A。这个运算会进行 y(x) = c × c × c × x的乘法运算，但是同样的处理可以由y(x) = ax（注意， a = c 3 ）这一次乘法运算（即没有隐藏层的神经网络）来表示

使用线性函数时，无法发挥多层网络带来的优势

神经网络的简单结构如下:



多维数组的运算

数组的维数可以用 `np.dim()` 函数来获得,数组的形状可以用 `np.shape()` 来获得每一维的情况

输出层所用的激活函数，要根据求解问题的性质决定。一般地，回归问题可以使用恒等函数，二元分类问题可以使用 sigmoid函数，多元分类问题可以使用 softmax函数。关于输出层的激活函数，我们将在下一节详细介绍。

softmax函数

机器学习的问题大致可以分为分类问题和回归问题。

恒等函数会把输入按原样输出

分类问题当中使用softmax()函数

分类问题中使用的softmax函数可以用下面的式(3.10)表示。

$$y_k = \frac{\exp(a_k)}{\sum_{i=1}^{n} \exp(a_i)} \tag{3.10}$$

$\exp(x)$是表示e^x的指数函数(e是纳皮尔常数2.7182…)。式(3.10)表示假设输出层共有n个神经元，计算第k个神经元的输出y_k。如式(3.10)所示，softmax函数的分子是输入信号a_k的指数函数，分母是所有输入信号的指数函数的和。

softmax函数会存在溢出的情况

softmax函数在进行运算时，加上或者减去某个函数并不会改变运算的结果。**一般使用的是输入信号当中的最大值**

softmax函数的实现可以像式(3.11)这样进行改进。

$$\begin{aligned} y_k = \frac{\exp(a_k)}{\sum_{i=1}^{n} \exp(a_i)} &= \frac{\mathrm{C}\exp(a_k)}{\mathrm{C}\sum_{i=1}^{n} \exp(a_i)} \\ &= \frac{\exp(a_k + \log \mathrm{C})}{\sum_{i=1}^{n} \exp(a_i + \log \mathrm{C})} \\ &= \frac{\exp(a_k + \mathrm{C}')}{\sum_{i=1}^{n} \exp(a_i + \mathrm{C}')} \end{aligned} \tag{3.11}$$

首先，式(3.11)在分子和分母上都乘上C这个任意的常数(因为同时对分母和分子乘以相同的常数，所以计算结果不变)。然后，把这个C移动到指数函数(exp)中，记为$\log \mathrm{C}$。最后，把$\log \mathrm{C}$替换为另一个符号C'。

softmax函数的输出是0.0到1.0之间的实数，并且函数的输出总和是1，因为这个性质，我们才可以softmax函的输出解释为"概率"

softmax函数的一些性质

- 输出的是 0.0 - 1.0之间的实数
- 所有的元素加起来之和为1，可以被当作概率来使用
- 元素之间的大小关系是相对的，不会改变对应的大小关系
- 神经网络把输出值最大的神经元作为识别结果

手写数字的识别

源代码及相关数据集见链接

MNIST是机器学习领域最有名的数据集之一

预处理： 提高识别性能和学习的效率

白化： 将数据整体的分布形状均匀化的方法

```
import sys, os
import pickle
sys.path.append(os.pardir)
import numpy as np
from dataset.mnist import load_mnist
from PIL import Image

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

def softmax(a):
    c = np.max(a)
    #上下同时减去一个常数 整个式子的值并不改变
    exp_a = np.exp(a - c)
    sum_exp_a = np.sum(exp_a)
    y = exp_a / sum_exp_a
    return y

def get_data():
    # 第一个参数将输入图像正规化为0.0-1.0的值
    # 第二个参数表示是否将参数展开为一维数组
    # 第三个参数表示是否将标签存为[0,0,1,000..]这种形式
    (x_train, t_train),(x_test,t_test) = load_mnist(normalize=True,
flatten=True, one_hot_label=False)
    return x_test,t_test

# 读取sample_weight.pkl当中学习到的权重参数，权重和偏置
def init_network():
    with open("sample_weight.pkl",'rb') as f:
        network  = pickle.load(f)

    return network

def predict(network,x):
    # 图像的格式为28*28 = 784
    # W1 (784,50) W2(50,100) W3(100,10)  b1(50,)  b2(100,) b3(10,)
    W1,W2,W3 = network['W1'],network['W2'],network['W3']
    b1,b2,b3 = network['b1'],network['b2'],network['b3']
    a1 = np.dot(x,W1) + b1
    z1 = sigmoid(a1)
    a2 = np.dot(z1,W2) + b2
    z2 = sigmoid(a2)
    a3 = np.dot(z2,W3) + b3
    y = softmax(a3)

    return y


```

```
# x是测试图像 t是图像对应的标签
x,t = get_data()
network = init_network()
accuracy_cnt = 0
for i in range(len(x)):
    # 返回的y是一个十个数字的元组
    y = predict(network,x[i])
    # p是最高概率的索引
    p = np.argmax(y)
    print(p)
    if p == t[i]:
        accuracy_cnt += 1

print("Accuracy:" + str(float(accuracy_cnt) / len(x) ))
```

批处理

大多数处理数值计算的库都进行了能够高效处理大型数组运算的最优化。并且，在神经网络的运算中，当数据传送成为瓶颈时，批处理可以减轻数据总线的负荷

代码与上述代码不同的部分

```
x, t = get_data()
network = init_network()
batch_size = 100 # 批数量
accuracy_cnt = 0
for i in range(0, len(x), batch_size):
    # i是一个起始位置
    # x_batch是取序列中的一段
    x_batch = x[i:i+batch_size]
    # y返回的是一个(batch_size,10)的二维元组
    y_batch = predict(network, x_batch)
    # axis=1 表示沿着第一维的方向进行寻找 找到最大的那个值的下标 p也是一个元组
    p = np.argmax(y_batch, axis=1)
    # 两个numpy的array比较 返回的是[true,false,true,false。。。]这种结果
    accuracy_cnt += np.sum(p == t[i:i+batch_size])
print("Accuracy:" + str(float(accuracy_cnt) / len(x)))
```

四、神经网络的学习

神经网络的学习，即利用数据决定参数值的方法，并用Python实现对MNIST手写数字数据集的学习

数据是机器学习的命根子。

4.1 一些概念

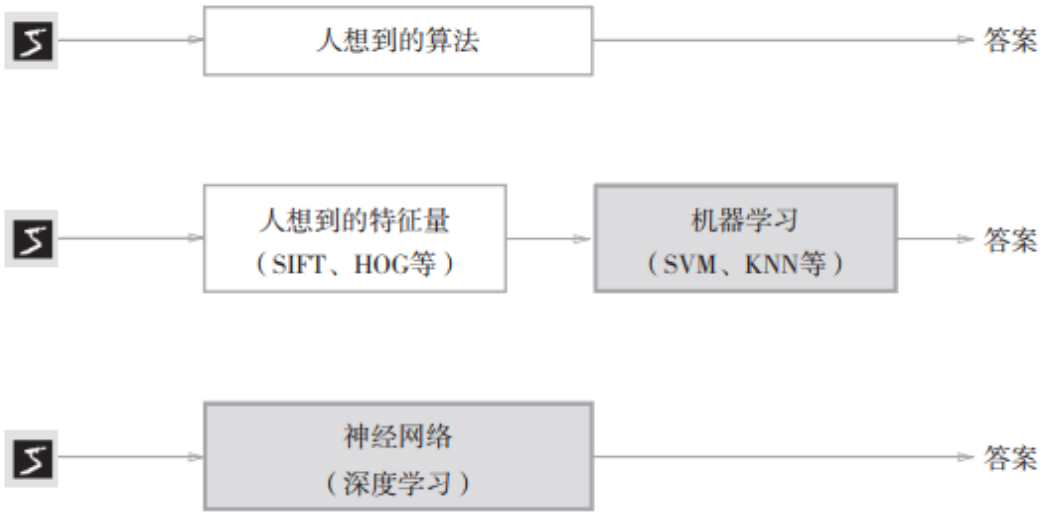
特征量：可以从输入数据（输入图像）中准确地提取本质数据（重要的数据）的转换器 常用的特征量包括SIFT、 SURF和HOG等

分类器：对输入进行分类，SVM，KNN等分类器进行学习

监督数据：用于训练的数据，即这个数据本来的值

one-hot表示： 把正确的解标签表示为1 其它标签表示为0

没有人介入的为深色矩形



mini-batch： 从全部数据中选出一部分，作为全部数据的“近似”。神经网络的学习也是从训练数据中选出一批数据（称为mini-batch,小批量），然后对每个mini-batch进行学习。比如，60000个训练数据中随机选择100笔，再用这100笔数据进行学习。这种学习方式称为mini-batch学习。

均方误差

$$E = \frac{1}{2}\sum_k (y_k - t_k)^2$$

对于手写数字的识别，tk是一个one-hot表示的元组，比如2的tk为[0,0,1,0,0,0,0,0,0,0],yk也是一个概率元组

交叉熵误差

$$E = -\sum_k t_k \log y_k$$

注意理解 对于手写数字的识别，tk是一个one-hot表示的元组，比如2的tk[0,0,1,0,0,0,0,0,0,0]，所以只是1乘以下标为2的yk处的数值

机器学习使用训练数据进行学习。使用训练数据进行学习，严格来说，就是针对训练数据计算损失函数的值，找出使该值尽可能小的参数。 因此，计算损失函数必须将所有的训练数据作为对象

如果导数的值为负，通过使该权重参数向正方向改变，可以减小损失函数的值；反过来，如果导数的值为正，则通过使该权重参数向负方向改变，可以减小损失函数的值

numerical differentiation数值微分

这里之所以取的是f(x + h) - f(x - h) 是为了减小数值微分的误差，这个也叫做中心差分， （f(x+h） 与 x 之间的差分为 向前差分）

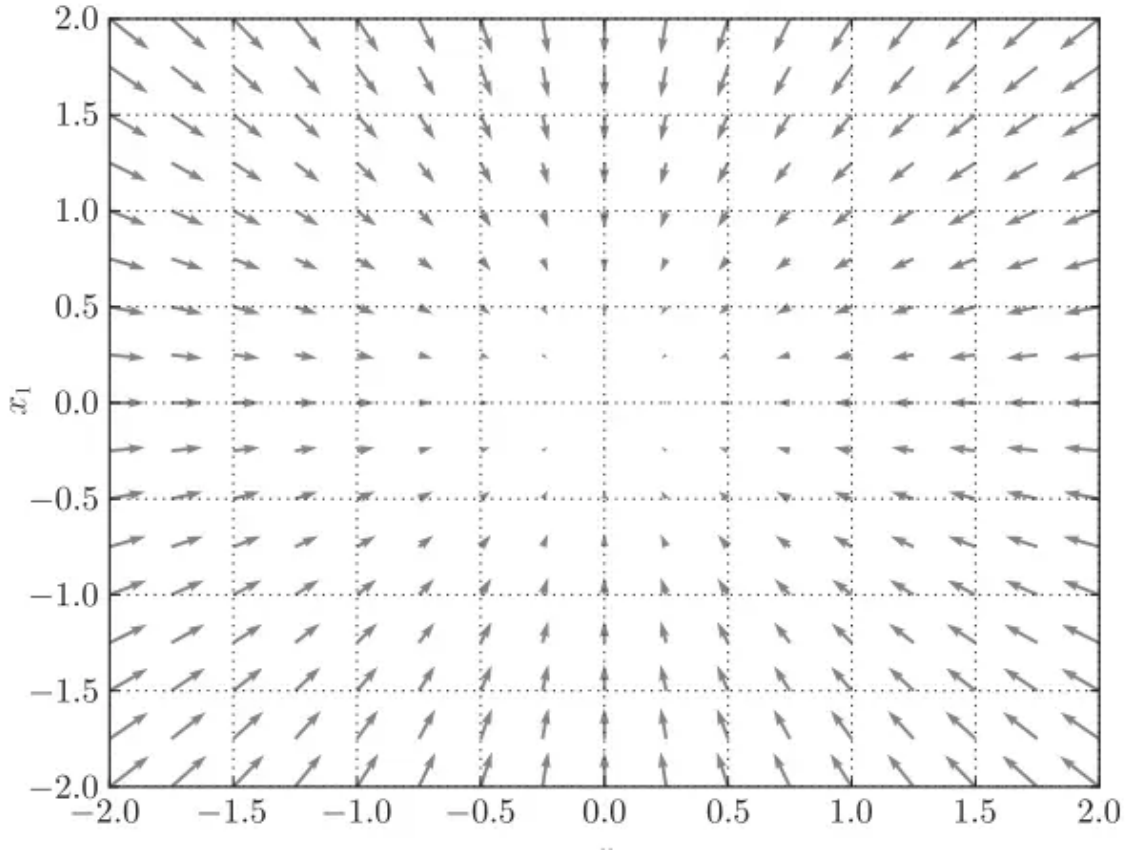
```python
def numerical_diff(f, x):
    h = 1e-4 # 0.0001
    return (f(x+h) - f(x-h)) / (2*h)
```

梯度 一个函数全部变量的偏导数汇总而成的向量为梯度（gradient）

梯度指示的方向是各点处的函数值减小最多的方向

$$f(x0, x1) = x0^2 + x1^2 \text{的梯度}$$

通过不断地沿梯度方向前进，逐渐减小函数值的过程就是梯度法（gradient method）




```python
# f为对应的函数 x为对应的变量的值的数组
def numerical_gradient(f, x):
    h = 1e-4 # 0.0001
    grad = np.zeros_like(x) # 生成和x形状相同的数组
    for idx in range(x.size):
        # 每一次求一个变量的偏导数的值
        tmp_val = x[idx]
        # f(x+h)的计算
        x[idx] = tmp_val + h
        fxh1 = f(x)
        # f(x-h)的计算
        x[idx] = tmp_val - h
        fxh2 = f(x)
        grad[idx] = (fxh1 - fxh2) / (2*h)
        x[idx] = tmp_val # 还原值
    return grad
```

神经网络的学习步骤

前提
神经网络存在合适的权重和偏置，调整权重和偏置以便拟合训练数据的过程称为“学习”。神经网络的学习分成下面4 个步骤。

步骤1（mini-batch）
从训练数据中随机选出一部分数据，这部分数据称为mini-batch。我们的目标是减小mini-batch 的损失函数的值。

步骤2（计算梯度）
为了减小mini-batch 的损失函数的值，需要求出各个权重参数的梯度。梯度表示损失函数的值减小最多的方向。

步骤3（更新参数）
将权重参数沿梯度方向进行微小更新。

交叉熵误差

其中t为数据的标签，y为实际的输出（都是数组形式）

$$E = -\frac{1}{N}\sum_{n}\sum_{k} t_{nk}\log y_{nk}$$

```python
# y是神经网络的输出 t是监督数据
def cross_entropy_error(y, t):
    # 维度
    if y.ndim == 1:
        t = t.reshape(1, t.size)
        y = y.reshape(1, y.size)

    # 监督数据是one-hot-vector的情况下，转换为正确解标签的索引
    if t.size == y.size:
        # 转换为如[[1],[0]...[9]]即第一维最大值的下标
        t = t.argmax(axis=1)

    batch_size = y.shape[0]
    return -np.sum(np.log(y[np.arange(batch_size), t] + 1e-7)) / batch_size
```

损失函数

对该权重参数（W）的损失函数（L）求导，如果导数的值为负，通过使该权重参数向正方向改变，可以减小损失函数的值；反过来，如果导数的值为正则通过使该权重参数向负方向改变，可以减小损失函数的值

数值微分

```python
def numerical_gradient(f, x):
    h = 1e-4 # 0.0001
    grad = np.zeros_like(x)
    # 迭代器 readwrite可读可写
    it = np.nditer(x, flags=['multi_index'], op_flags=['readwrite'])
    while not it.finished:
        idx = it.multi_index
        tmp_val = x[idx]
        x[idx] = float(tmp_val) + h
        fxh1 = f(x) # f(x+h)

        x[idx] = tmp_val - h
        fxh2 = f(x) # f(x-h)
        grad[idx] = (fxh1 - fxh2) / (2*h)

        x[idx] = tmp_val # 还原值
        it.iternext()

    return grad
```

两层神经网络

```python
# 两层神经网络
class TwoLayerNet:
    # 输入层神经元 隐藏层神经元 输出个数 学习步长
    def __init__(self, input_size, hidden_size,
output_size,weight_init_std=0.01):
        # 初始化权重
        self.params = {}
        # 权重参数需要服从高斯分布 np.radom.radn
        # 输入参数个数 * 隐层参数个数
        self.params['W1'] = weight_init_std * np.random.randn(input_size,
hidden_size)
        # 隐层参数个数
        self.params['b1'] = np.zeros(hidden_size)
        # 隐层参数个数 * 输出层参数个数
        self.params['W2'] = weight_init_std * np.random.randn(hidden_size,
output_size)
        # 输出的参数个数
        self.params['b2'] = np.zeros(output_size)

    # 进行推理，x是输入参数
    def predict(self, x):
        W1, W2 = self.params['W1'], self.params['W2']
        b1, b2 = self.params['b1'], self.params['b2']
        a1 = np.dot(x, W1) + b1
        z1 = sigmoid(a1)
        a2 = np.dot(z1, W2) + b2
        y = softmax(a2)
        return y

    # x:输入数据, t:监督数据
```

```
    def loss(self, x, t):
        y = self.predict(x)
        return cross_entropy_error(y, t)

    # 计算识别精度
    def accuracy(self, x, t):
        y = self.predict(x)
        y = np.argmax(y, axis=1)
        t = np.argmax(t, axis=1)
        accuracy = np.sum(y == t) / float(x.shape[0])
        return accuracy
    # x:输入数据, t:监督数据
    # x:输入数据, t:监督数据
    def numerical_gradient(self, x, t):
            loss_W = lambda W: self.loss(x, t)
            grads = {}
            grads['W1'] = numerical_gradient(loss_W, self.params['W1'])
            grads['b1'] = numerical_gradient(loss_W, self.params['b1'])
            grads['W2'] = numerical_gradient(loss_W, self.params['W2'])
            grads['b2'] = numerical_gradient(loss_W, self.params['b2'])
            return grads
```

测试代码

```
# (x_train, t_train), (x_test, t_test) =  load_mnist(normalize=True, one_hot_laobel = True)
(x_train, t_train),(x_test,t_test) = load_mnist(normalize=True, one_hot_label=False)
train_loss_list = []

# 超参数
iters_num = 100000
train_size = x_train.shape[0]
batch_size = 100
learning_rate = 0.1

network = TwoLayerNet(input_size=784, hidden_size=50, output_size=10)

for i in range(iters_num):
    # 获取mini-batch batch_mask是一个数组
    batch_mask = np.random.choice(train_size, batch_size)
    x_batch = x_train[batch_mask]
    t_batch = t_train[batch_mask]
    # 计算梯度
    grad = network.numerical_gradient(x_batch, t_batch)
    # grad = network.gradient(x_batch, t_batch) # 高速版!
    # 更新参数
    for key in ('W1', 'b1', 'W2', 'b2'):
        network.params[key] -= learning_rate * grad[key]
    # 记录学习过程
    loss = network.loss(x_batch, t_batch)
    train_loss_list.append(loss)
    print(loss)
```

完整数据集及实现代码参见:

https://github.com/SisyphusTang/AI-Computing-System/tree/master/%E5%AE%9E%E9%AA%8C/%E6%89%8B%E5%86%99%E6%95%B0%E5%AD%97%E7%9A%84%E8%AF%86%E5%88%AB