最优化大作业

一

> *一、考虑一个20 节点的分布式系统。节点i 有线性测量$b_i = A_i x + e_i$ ，其中b_i 为10 维的测量值，A_i 为10x300 维的测量矩阵，x 为300 维的未知稀疏向量且稀疏度为5，e_i 为10 维的测量噪声。从所有b_i 与A_i 中恢复x 的一范数规范化最小二乘模型如下：*
>
> $$min(1/2)||A_1 x - b_1||_2^2 + \ldots + (1/2)||A_{20} x - b_{20}||_2^2 + p||x||_1$$
>
> *其中p 为非负的正则化参数。请设计下述分布式算法求解该问题：*
> *1、邻近点梯度法；*
> *2、交替方向乘子法；*
> *3、次梯度法；*
> *在实验中，设x 的真值中的非零元素服从均值为0 方差为1 的高斯分布，*
>
> *A_i 中的元素服从均值为0 方差为1 的高斯分布，e_i 中的元素服从均值为0 方差为0.2 的高斯分布。*
>
> *对于每种算法，请给出每步计算结果与真值的距离以及每步计算结果与最优解的距离。*
>
> *此外，请讨论正则化参数p 对计算结果的影响。*

1、邻近点梯度法

算法解释

邻近点梯度算法：

- 首先对函数的光滑部分s(x)进行梯度下降：

  $$x^{k+\frac{1}{2}} = x^k - \alpha \bigtriangledown s(x^k)$$

  在本题中是：

  $$x^{k+\frac{1}{2}} = x^k - \alpha * \sum_{i=1}^{20} A_i^T (A_i x^k - b_i)$$

  - 其中对$\frac{1}{2}||A_i x - b_i||_2^2$进行求导：

    令

    $$f(x) = \frac{1}{2}||Ax - b||_2^2$$

    $$\begin{aligned} f(x) &= \frac{1}{2}||Ax - b||_2^2 \\ &= \frac{1}{2}(Ax - b)^T (Ax - b) \\ &= \frac{1}{2}(x^T A^T - b^T)(Ax - b) \\ &= \frac{1}{2}(x^T A^T Ax - x^T A^T b - b^T Ax + b^T b) \end{aligned}$$

    利用如下性质：

$$\frac{\partial x^T D x}{\partial x} = (D + D^T)x$$
$$\frac{\partial D^T x}{\partial x} = D$$
$$\frac{\partial x^T D}{\partial x} = D$$

则有

$$\begin{aligned}\frac{\partial f(x)}{\partial x} &= \frac{1}{2}\{[A^T A + (A^T A)^T]x - A^T b - A^T b\} \\ &= \frac{1}{2}\{2A^T Ax - 2A^T b\} \\ &= A^T Ax - 2A^T b \\ &= A^T(Ax - b)\end{aligned}$$

因此

$$\frac{\partial f(x)}{\partial x} = A^T(Ax - b)$$

- 然后更新x^k:

$$x^k = arg_{min} p||x||_1 + \frac{1}{2\alpha}||x - x^{k+\frac{1}{2}}||^2$$

直观理解：先求解一个距离$x^{k+\frac{1}{2}}$（根据光滑项梯度下降得到的x）不太远的点x，并使得$p||x||_1$也相对较小。用这个值来更新得到x^k

使用软阈值法进行计算，结果：

(下面的大小比较指的是x的每一项进行比较)

$$x^k = \begin{cases} x^{k+\frac{1}{2}} + \alpha * p & , x^{k+\frac{1}{2}} < -\alpha * p \\ 0 & , |x^{k+\frac{1}{2}}| < \alpha * p \\ x^{k+\frac{1}{2}} - \alpha * p & , x^{k+\frac{1}{2}} > \alpha * p \end{cases}$$

  - 对软阈值法的解释：

    要求解：

$$x^k = arg_{min} p||x||_1 + \frac{1}{2\alpha}||x - x^{k+\frac{1}{2}}||^2 \quad , x \in R^n, x^{k+\frac{1}{2}} \in R^n$$

    根据范数的定义可以将上面的目标函数拆成：

$$F(x) = (\frac{1}{2\alpha}(x_1 - x_1^{k+\frac{1}{2}})^2 + p|x_1|) + (\frac{1}{2\alpha}(x_2 - x_2^{k+\frac{1}{2}})^2 + p|x_2|) + \ldots + (\frac{1}{2\alpha}(x_n - x_n^{k+\frac{1}{2}})^2 + p|x_n|)$$

    注：$x_j^{k+\frac{1}{2}}$表示$x^{k+\frac{1}{2}}$中的第j个元素，是已计算得到的值（常量），x是变量,根据题目，这里的n=300

    也就是说可以通过求解N个独立的形如下面函数的优化问题，来求解上面的x^k(此时下面的$x, x^{k+\frac{1}{2}}$都表示矩阵中的一个元素):

$$f(x)=\frac{1}{2\alpha}(x-x^{k+\frac{1}{2}})^2+p|x|$$

f(x)可导：

$$\frac{\partial f(x)}{\partial x}=\frac{1}{\alpha}(x-x^{k+\frac{1}{2}})+p*sgn(x)$$

$$sgn(x)=\begin{cases}1 & ,x>0\\0 & ,x=0\\-1, & x<0\end{cases}$$

要求极值，令导数为0，得：

$$x=x^{k+\frac{1}{2}}-\alpha*p*sgn(x)$$

这个结果左右两端都有x，分情况讨论：

- $x^{k+\frac{1}{2}}>\alpha*p$时
  - 假设x<0,则sgn(x)=-1,则$x=x^{k+\frac{1}{2}}+\alpha*p>0,(\text{因为}x^{k+\frac{1}{2}}>\alpha*p>0)$，与x<0矛盾
  - 假设x>0,则sgn(x)=1,则$x=x^{k+\frac{1}{2}}-\alpha*p>0$，成立

    因此在$x=x^{k+\frac{1}{2}}-\alpha*p>0$处取得极值，

    假设$x=0$，则$f(x)>0$,让$x=\infty$,则$f(x)>0$，因此在$x=x^{k+\frac{1}{2}}-\alpha*p>0$处取得的是极小值
- $x^{k+\frac{1}{2}}<-\alpha*p$时
  - 假设x<0,则sgn(x)=-1,则$x=x^{k+\frac{1}{2}}+\alpha*p<0$，成立
  - 假设x>0,则sgn(x)=1,则$x=x^{k+\frac{1}{2}}-\alpha*p>0$，$(x^{k+\frac{1}{2}}<-\alpha*p<0)$，与$x>0$矛盾

    因此在$x=x^{k+\frac{1}{2}}+\alpha*p<0$处取得极值，

    假设$x=0$，则$f(x)>0$,让$x=-\infty$,则$f(x)>0$，因此在$x=x^{k+\frac{1}{2}}+\alpha*p<0$处取得的是极小值
- $-\alpha*p<x^{k+\frac{1}{2}}<\alpha*p$时,(即$|x^{k+\frac{1}{2}}|<\alpha*p$):
  - 假设x<0,则sgn(x)=-1,则$x=x^{k+\frac{1}{2}}+\alpha*p>0$，与$x<0$矛盾
  - 假设x>0,则sgn(x)=1,则$x=x^{k+\frac{1}{2}}-\alpha*p<0$，与$x>0$矛盾

综合上面三种情况，f(x)的最小值在以下几个位置取得：

$$arg_{min}f(x)=\begin{cases}x^{k+\frac{1}{2}}+\alpha*p & ,x^{k+\frac{1}{2}}<-\alpha*p\\0 & ,|x^{k+\frac{1}{2}}|<\alpha*p\\x^{k+\frac{1}{2}}-\alpha*p & ,x^{k+\frac{1}{2}}>\alpha*p\end{cases}$$

因此可以得到$arg_{min}F(x)$优化问题的解为：

注：此时的$x^{k+\frac{1}{2}}$表示的是矩阵形式：

$$arg_{min}F(x)=soft(\sum_{i=1}^{20}x_i^{k+\frac{1}{2}},\alpha*p)=\begin{cases}x^{k+\frac{1}{2}}+\alpha*p & ,x^{k+\frac{1}{2}}<-\alpha*p\\0 & ,|x^{k+\frac{1}{2}}|<\alpha*p\\x^{k+\frac{1}{2}}-\alpha*p & ,x^{k+\frac{1}{2}}>\alpha*p\end{cases}$$

代码

需要用到的一些库：

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

全局变量，参数和数据

```
#data
alpha = 0.001
p = 10        # 正则化参数 10的时候最好然后11 然后1
epsilon = 1e-5  # 最大允许误差

A = {}
x = np.zeros((300,1))
e = {}
b = {}
```

用于生成随机数的函数，并保存，只调用一次，之后的算法运行时也不用调用，直接读取数据：

```
# 生成数据
def Generate_data():
    # x 为 300 维的未知稀疏向量且稀疏度为 5
    # 设 x 的真值中的非零元素服从均值为 0 方差为 1 的高斯分布
    data_x = np.random.normal(0, 1, 5)
    col = np.random.randint(0, 300, 5)
    for i in range(5):
        x[col[i]] = data_x[i]
    np.save("./data/x", x)
    for i in range(20):
        # A_i 为 10 x 300 维的测量矩阵
        # A_i 中的元素服从均值为 0 方差为 1 的高斯分布
        A[i] = np.random.normal(0, 1, (10,300))
        # e_i为 10 维的测量噪声
        # e_i 中的元素服从均值为 0 方差为 0.2 的高斯分布
        e[i] = np.random.normal(0, math.sqrt(0.2), (10,1))
        # b_i 为 10 维的测量值
        b[i] = np.dot(A[i], x)+e[i]
        name_A = "./data/A/A" + str(i)
        name_e = "./data/e/e" + str(i)
        name_b = "./data/b/b" + str(i)
        np.save(name_A, A[i])
        np.save(name_e, e[i])
        np.save(name_b, b[i])
```

用于读取数据的函数：

```python
# 获取数据
def get_data():
    for i in range(20):
        name_A = "./data/A/A" + str(i) + ".npy"
        name_e = "./data/e/e" + str(i) + ".npy"
        name_b = "./data/b/b" + str(i) + ".npy"
        A[i] = np.load(name_A)
        e[i] = np.load(name_e)
        b[i] = np.load(name_b)
    # x = np.load("./data/x.npy")
```

模拟分布式，将x^k由主节点传入每个节点，计算$\alpha * A_i^T(A_i x^k - b_i)$，再将结果传回主节点：

```python
def A_node(xk,i):# get_xk_half_temp
    xk_half_temp = alpha*(A[i].T.dot(A[i].dot(xk)-b[i]))
    return xk_half_temp
```

主节点上:

- 设置变量
- 开始迭代：
  - 调用各个节点计算每一部分值$\alpha * A_i^T(A_i x^k - b_i)$,然后求得$x^{k+\frac{1}{2}}$
  - 根据软阈值法计算x^{k+1}
  - 记录每一步的迭代结果
  - 判断结果是否收敛，是否要停止迭代
- 记录最优值
- 返回每一步的迭代结果和最优值

```python
def master():
    # Generate_data()
    get_data()
    xk_half_temp_sum = np.zeros((300,1))     # x^{k+1/2}_temp
    xk_half = np.zeros((300,1))              # x^{k+1/2}
    xk = np.zeros((300,1))                   # xk
    xk_new = np.zeros((300,1))               # 每一步更新后的xk
    k = 0 # 记录迭代次数
    xk_step = []     # 记录每一步的计算结果

    while 1:
        xk_half_temp_sum = np.zeros((300,1))  # 重新更新时，要记得将和清零，否则会不断叠加

        # 从master将xk分发到20个节点上计算x^{k+1/2}, 计数是0，……，19，仍是20个节点
        for i in range(0,20):
            # 将计算得到的部分x^{k+1/2}传回master，更新x^k,先进行求和
            xk_half_temp_sum = xk_half_temp_sum + A_node(xk,i)

        xk_half = xk - xk_half_temp_sum
        # 软阈值法
        for i in range(300):
            if xk_half[i] < -alpha*p:
                xk_new[i] = xk_half[i]+alpha*p
            elif xk_half[i] > alpha*p:
                xk_new[i] = xk_half[i]-alpha*p
```

```
            else:
                xk_new[i] = 0

        xk_step.append(xk_new.copy())  # 记录每一步的迭代结果,这里一定要进行深拷贝

        # 因为在计算中无法准确找到次梯度为0的点，我们近似地两步之间的二范数10^{-5},看作结果收敛
        if np.linalg.norm(xk_new-xk) < epsilon:
            break
        else:
            # 进行更新xk，注意一定要进行深拷贝，否则当xk_new改变时，xk也会改变
            xk = xk_new.copy()
            k = k+1

    x_optm = xk_new.copy()  # 跳出循环得到最优解
    return xk_step, x_optm
```

根据结果绘图，绘制每步计算结果与真值的距离以及每步计算结果与最优解的距离：

```
def draw(xk_step, x_optm):
    x = np.load("./data/x.npy")
    dis_xk_real = []    # xk与真实值的距离
    dis_xk_optm = []    # xk与最优解的距离

    for xk in xk_step:
        dis_xk_real.append(np.linalg.norm(xk - x))        # 记录每步计算结果与真值的距离
        dis_xk_optm.append(np.linalg.norm(xk - x_optm))   # 记录每步计算结果与最优解的距离

    # 绘图
    plt.title('Proximal Gradient Method')
    plt.xlabel('k:Iteration_times')
    plt.ylabel('distance')

    plt.plot(dis_xk_real, 'r', label='distance from truth value to xk')
    plt.plot(dis_xk_optm, 'b', label='distance from the optimal solution to xk')

    plt.grid()
    plt.legend()
    plt.show()
```

最后调用主函数和绘图函数

```
xk_step, x_optm = master()
draw(xk_step, x_optm)
```

计算结果

收敛速度较快，计算结果也很好，和真值之间的差距较小。对于这个结果需要根据参数p以及α进行调试。

2、交替方向乘子法

算法解释

- 交替方向乘子的基本形式是

  对于优化问题：

$$min f_i(x) + f_2(y)$$
$$s.t.\ Ax + By = 0$$

它的拉格朗日函数为：

$$L(x, y, v) = f_1(x) + f_2(y) + < v, Ax - b >$$

它的增广拉格朗日函数为：

$$L_c(x, y, v) = L(x, y, v) + \frac{c}{2}||Ax - b||_2^2$$

迭代算法如下：

$$\begin{cases} x^{k+1} = arg\,min_x L_c(x, y^k, v^k) \\ y^{k+1} = arg\,min_y L_c(x^{k+1}, y, v^k) \\ v^{k+1} = v^k + c(Ax^{k+1} + By^{k+1}) \end{cases}$$

其中$L_c(x, y, v)$为原目标函数的增广拉格朗日函数，c 为惩罚项系数。

- 对于本题：
  - 首先，为了让原本的无约束优化问题变为有约束优化问题，引入一个新变量y = x，所以一致性约束为x - y = 0:

$$min \frac{1}{2} \sum_{i=1}^{20} ||A_i x_i - b_i||_2^2 + p||y||_1$$
$$s.t.\ \ \forall i \quad x_i - y = 0$$

  - 它的拉格朗日函数形式为：

$$L(\{x_i\}, y, \{\lambda_i\}) = \frac{1}{2} \sum_{i=1}^{20} ||A_i x_i - b_i||_2^2 + p||y||_1 + \sum_{i=1}^{20} < \lambda_i, x_i - y >$$

  - 它的增广拉格朗日函数为(加入惩罚项)：

$$L(\{x_i\}, y, \{\lambda_i\}) = \frac{1}{2} \sum_{i=1}^{20} ||A_i x_i - b_i||_2^2 + p||y||_1 + \sum_{i=1}^{20} < \lambda_i, x_i - y > + \frac{c}{2} \sum_{i=1}^{20} ||x_i - y||_2^2$$

  - 然后根据算法，可以获得迭代方程组(删去无关项，如第x_i项的数据更新和第x_{i-1}等都无关)：

$$\begin{cases} x_i^{k+1} = arg\,min_x \frac{1}{2}||A_i x_i - b_i||_2^2 + < \lambda_i^k, x_i - y^k > + \frac{c}{2}||x_i - y^k||_2^2 \\ y^{k+1} = arg\,min_y \ \ p||y||_1 + \sum_{i=1}^{20} < \lambda^k, x_i^{k+1} - y > + \frac{c}{2} \sum_{i=1}^{20} ||x_i^{k+1} - y||_2^2 \\ \lambda_i^{k+1} = \lambda_i^k + c(x_i^{k+1} - y^{k+1}) \end{cases}$$

  - 然后对上面的三步迭代fe分别求解：
    - 对于x_i^{k+1}

      x_i^{k+1}部分的目标函数是光滑的，可导，可以直接求得它的梯度：

$$f(x_i) = \frac{1}{2}||A_i x_i - b_i||_2^2 + < \lambda_i^k, x_i - y^k > + \frac{c}{2}||x_i - y^k||_2^2$$
$$\frac{\partial f(x)}{\partial x} = A_i^T(A_i x_i - b_i) + \lambda_i^k + c * I(x_i - y^k)$$

令梯度为0，取极值：

$$A_i^T(A_ix_i-b_i)+\lambda_i^k+c*I(x_i-y^k)=0$$

因此，可得

$$x_i^{k+1}=(A_i^TA_i+c*I)^{-1}(A_i^Tb_i+c*y^k-\lambda_i^k)\quad ,I是单位矩阵$$

- 对于y^{k+1}

  虽然y^{k+1}不是光滑的，但可以利用软门限法，求它的邻近点投影。

  在求它的邻近点投影前，先对y^{k+1}做一次变换，得到

$$y^{k+1}=arg\,min_y\ \ p||y||_1+\frac{c}{2}\sum_{i=1}^{20}||y-(x_i^{k+1}+\frac{\lambda_i^k}{c})||_2^2$$

  注：这样转换的依据是（下面的x_{i_j},指的是x_i的第j个元素，λ_{i_j}同理）：

$$\begin{aligned}y^{k+1}&=arg\,min_y\ \ p||y||_1+\sum_{i=1}^{20}[\sum_j^n(\lambda_{i_j}^k*(x_{i_j}^{k+1}-y_{i_j}))+\frac{c}{2}\sum_j^n(x_{i_j}^{k+1}-y_j)^2]\\&=arg\,min_y\ \ p||y||_1+\sum_{i=1}^{20}[\sum_j^n(\lambda_{i_j}^k*(x_{i_j}^{k+1}-y_j))+\frac{c}{2}\sum_j^n(x_{i_j}^{k+1}-y_j)^2+\sum_j^n\frac{(\lambda_{i_j}^k)^2}{2}]\end{aligned}$$

  注：这里求使得整体最大的y，与y无关的$\sum_{i=1}^{20}\sum_j^n\frac{(\lambda_{i_j}^k)^2}{2}$加在式子后面对求得的结果$y$并无影响

$$\begin{aligned}&=arg\,min_y\ \ p||y||_1+\sum_{i=1}^{20}[\frac{c}{2}\sum_j^n((x_{i_j}^{k+1}-y_j)^2+\frac{2\lambda_{i_j}{}^k}{c}*(x_{i_j}^{k+1}-y_j)+\frac{(\lambda_{i_j}^k)^2}{c})]\\&=arg\,min_y\ \ p||y||_1+\sum_{i=1}^{20}[\frac{c}{2}\sum_j^n((x_{i_j}^{k+1}-y_j+\frac{\lambda_{i_j}^k}{c})^2)]\\&=arg\,min_y\ \ p||y||_1+\sum_{i=1}^{20}(\frac{c}{2}||x_i^{k+1}-y+\frac{\lambda_i^k}{c}||_2^2)\\&=arg\,min_y\ \ p||y||_1+\frac{c}{2}\sum_{i=1}^{20}||y-(x_i^{k+1}+\frac{\lambda_i^k}{c})||_2^2\end{aligned}$$

  然后这个形式就可以对y^{k+1}使用软门限法求得最小值，之前在邻近点算法中已经详细解释过软门限法了，这里就不再赘述，直接写出结果：

$$y^{k+1}=\begin{cases}\frac{1}{20}(\sum_{i=1}^{20}(x_i^{k+1}+\frac{\lambda^k}{c})+\frac{p}{c}) & ,\sum_{i=1}^{20}(x_i^{k+1}+\frac{\lambda^k}{c})<-\frac{p}{c}\\0 & ,|\sum_{i=1}^{20}(x_i^{k+1}+\frac{\lambda^k}{c})|<\frac{p}{c}\\\frac{1}{20}(\sum_{i=1}^{20}(x_i^{k+1}+\frac{\lambda^k}{c})-\frac{p}{c}) & ,\sum_{i=1}^{20}(x_i^{k+1}+\frac{\lambda^k}{c})>\frac{p}{c}\end{cases}$$

- 对于λ^{k+1}

  就直接利用下面的公式进行更新：

$$\lambda_i^{k+1}=\lambda_i^k+c(x_i^{k+1}-y^{k+1})$$

代码

读取数据部分和绘图的函数这里就不再赘述。

参数以及全局变量：

```
# data
p = 0.01
c = 0.005
epsilon = 7*1e-4  # 最大允许误差

A = {}
x = np.zeros((300,1))
e = {}
b = {}

ATb = []
ATA_add_c_inv = [] # 这两个值在进行计算的时候无论迭代多少次都不会变,计算一次后直接调用
```

- 每个节点要做的事：
  - 计算$A_i^TA_i$和$A_i^Tb_i$

    ```
    def A_node_ATmul(i, c):
        if c=='A':
            result = A[i].T.dot(A[i])
        elif c=='b':
            result = A[i].T.dot(b[i])
        return result
    ```

  - 计算$x_i^{k+1}=(A_i^TA_i+c*I)^{-1}(A_i^Tb_i+c*y^k-\lambda_i^k)$ ，I是单位矩阵

    ```
    def A_node_xk_1(yk, lk_i, i):
        xk_1_i = ATA_add_c_inv[i].dot(ATb[i] + c*yk - lk_i)
        return xk_1_i
    ```

  - 计算$\lambda_i^{k+1}=\lambda_i^k+c(x_i^{k+1}-y^{k+1})$

    ```
    def A_node_lk_1(lk_i, xk_1_i, yk_1):
        lk_1_i = lk_i + c*(xk_1_i - yk_1)
        return lk_1_i
    ```

- 主节点
  - 初始化参数
  - 调用每个节点计算$A_i^TA_i$和$A_i^Tb_i$，以及$(A_i^TA_i+c*I)^{-1}$（都只计算一次）
  - 开始迭代
    - 调用每个节点，更新x_i^{k+1}
    - 在主节点上计算y^{k+1}
    - 调用每个节点，更新λ_i^{k+1}
    - 将迭代得到的x_0^{k+1}保存
    - 判断三个变量是否都已收敛，如果收敛就跳出循环，否则更新后继续迭代
  - 记录跳出循环得到最优解，返回得到的最优解和每一步的迭代结果

```
def master():
```

```
    get_data()
    xk = []
    yk = np.zeros((300,1))
    lk = []
    for i in range(20):
        xk.append(np.zeros((300,1)))
        lk.append(np.zeros((300,1)))
    yk_1 = np.zeros((300,1))
    xk_step_0 = []
    xk_1 = []
    lk_1 = []

    ATA = []
    count = 0

    for i in range(20):# 这个值是固定的，在开始时计算一次，不需要重复计算
        ATA.append(A_node_ATmul(i,'A'))
        ATb.append(A_node_ATmul(i,'b'))
    I = np.eye(300,300)
    for i in range(20):
        ATA_add_c_inv.append( np.linalg.inv(ATA[i]+c*I) )

    while 1:    # 开始迭代
        count = count + 1
        for i in range(20):
            xk_1.append( A_node_xk_1(yk, lk[i], i) )

        # 计算y的更新值
        for i in range(300):
            temp = 0
            for j in range(20):
                temp = temp + (xk_1[j][i]+lk[j][i]/c)     # 这个值要多次用到，防止重复计算

            if temp < -p/c:
                yk_1[i] = (temp + p/c)/20
            elif temp > p/c:
                yk_1[i] = (temp - p/c)/20
            else:
                yk_1[i] = 0

        # 计算\lambda的更新值
        for i in range(20): # 将值传入每个节点并将结果返回
            lk_1.append( A_node_lk_1(lk[i], xk_1[i], yk_1) )

        xk_step_0.append(xk_1[0])

        # 需要三个值都收敛
        if np.linalg.norm(xk_1[0]-xk[0]) < epsilon and np.linalg.norm(lk_1[0]-lk[0])
<epsilon and np.linalg.norm(yk_1-yk) < epsilon:
            print(np.linalg.norm(xk_1[0]-xk[0]))
            print(count, np.linalg.norm(xk_1[0] - xk[0]), np.linalg.norm(lk_1[0] -
lk[0]), np.linalg.norm(yk_1 - yk))
            break
        else:
            # 更新值
            if count%500 == 0:
                print(count, np.linalg.norm(xk_1[0] - xk[0]), np.linalg.norm(lk_1[0] -
lk[0]), np.linalg.norm(yk_1 - yk))
```

```
            xk = xk_1.copy()
            yk = yk_1.copy()
            lk = lk_1.copy()
            xk_1 = []
            lk_1 = []


    x_optm = xk_1[0].copy()  # 跳出循环得到最优解
    return xk_step_0, x_optm
```

最后调用主函数和绘图函数

```
xk_step, x_optm = master()
draw(xk_step, x_optm)
```

计算结果

交替方向乘子算法的收敛相较于其他两个算法非常慢，可以让收敛限制大一点，就可以更快得到结果：

交替乘子算法收敛的过程相较于邻近点梯度波动比较大，但是收敛之后的效果非常好，和真值的差距在0.1左右。可以调整收敛限制、参数p、参数c来对使得结果能有更好的效果。

3、次梯度法

算法解释

- 一般的次梯度法：

  用 $g_0(x) \in \partial f_0(x)$ 表示 $f_0(x)$ 在 x 处的次梯度。

  次梯度法，就是在不可微点处用次梯度代替梯度，然后使用梯度下降法。当函数可微时，与梯度下降法无异。

  $$x^{k+1} = x^k - \alpha * g_0(x)$$

  次梯度法的步长选取：

  - 固定步长：$\alpha_k = \alpha$
  - 不可加，但平方可加：$\alpha_k = \frac{1}{k}$
  - 不可加、平方不可加，但能收敛至0，$\alpha_k = \frac{1}{\sqrt{k}}$

- 对于本题：

  对原问题的目标函数

  $$f(x) = \frac{1}{2}\sum_{i=1}^{20}||A_i x - b_i||_2^2 + p||x||_1$$

  进行求导得到(二范数的部分可导)

  $$\frac{\partial f(x)}{\partial x} = \sum_{i=1}^{20} A_i^T(A_i x - b_i) + p\frac{\partial ||x||_1}{\partial x}$$

  但正则化项在其元素等于0时不可微，它的次梯度为[-1,1]之间的任意值，即

$$\begin{cases}(\frac{\partial||x||_1}{\partial x})_i = -1, x_i < 0 \\ (|\frac{\partial||x||_1}{\partial x})_i| <= 1, x_i = 0 \\ (\frac{\partial||x||_1}{\partial x})_i = 1, x_i > 0 \end{cases}$$

选用递减步长：$\alpha^k = 0.001 \times \frac{1}{\sqrt{k}}$

然后每一步迭代：

$$x^{k+1} = x^k - \alpha * g_0(x)$$

代码

参数以及全局变量：

```
# data
p = 0.01
alpha = 0.001
epsilon = 1e-5  # 最大允许误差

A = {}
x = np.zeros((300,1))
e = {}
b = {}
```

每个节点，计算$A_i^T(A_ix - b_i)$,即梯度里的一部分：

```
def A_node(xk, i):
    gradient_i = A[i].T.dot(A[i].dot(xk)-b[i])
    return gradient_i
```

主节点上：

- 更新步长
- 设置变量
- 进入迭代
    - 利用每个节点传回的值进行求和，得到光滑部分的梯度
    - 判断x^k中的每一个值，计算他们的次梯度
    - 最后求和，得到梯度
    - 进行梯度下降得到x^{k+1}，并记录
    - 判断是否已收敛，如果是就跳出迭代
- 记录最优解，并作为返回值返回

```
def master():
    get_data()
    xk = np.zeros((300,1))
    k = 0
    a = alpha
    xk_step = []    # 纪录每一步的结果

```

```
    while 1:
        k = k + 1
        a = a / math.sqrt(k)
        partial_gradient = 0    # 光滑部分梯度
        for i in range(20):
            partial_gradient = partial_gradient + A_node(xk, i)

        L1_gradient = np.zeros((300,1))
        for i in range(300):
            if xk[i] > 0:
                L1_gradient[i] = 1
            elif xk[i] < 0:
                L1_gradient[i] = -1
            else:
                L1_gradient[i] = np.random.uniform(-1,1)

        gradient = partial_gradient + L1_gradient
        xk_1 = xk - a * gradient

        xk_step.append(xk_1)

        if np.linalg.norm(xk_1-xk) < epsilon:
        # if k > 6000:
            break
        else:
            if k%500 == 0:
                print(k, np.linalg.norm(xk_1 - xk))
            xk = xk_1.copy()

    x_optm = xk_1.copy()    # 跳出循环得到最优解
    return xk_step, x_optm
```

最后调用主函数和绘图函数

```
xk_step, x_optm = master()
draw(xk_step, x_optm)
```

计算结果

可以比较看出，次梯度法的回归效果并不是很理想，尽管最优解能够收敛，然而最优解与真值的差距较大。

正则化参数 p 对计算结果的影响

L1 正则化可以起到使参数更加系稀疏的作用，即得到的参数是一个稀疏矩阵，可以用于特征选择。即得到的参数是一个稀疏矩阵，可以用于特征选择。通常机器学习中特征数量很多，在预测或分类时，那么多特征显然难以选择，但是如果代入这些特征得到的模型是一个稀疏模型，很多参数是0，表示只有少数特征对这个模型有贡献，绝大部分特征是没有贡献的，即使去掉对模型也没有什么影响，此时我们就可以只关注系数是非零值的特征。这相当于对模型进行了一次特征选择，只留下一些比较重要的特征，提高模型的泛化能力，降低过拟合的可能。因此P对系数矩阵更有效，本题中的x就是一个300*1,稀疏度为5的稀疏矩阵。

邻近点梯度法

p=0.01　　p=1

p=10 p=100

可以看到p过大或过小时，结果都不太好，而当p=10左右的时候迭代结果最优。

交替方向乘子法

p=0.01 p=0.1

p=1 p=10

当p较小时结果并没有p较大时稳定，且p较大时结果更优。可以从p=10的迭代50000次看到，它收敛真的很慢。

次梯度法

p=0.01 p=0.1

p=1 p=10

可以看出，p在次梯度法上对结果并无太大影响。

二

> *二、请设计下述算法，求解MNIST 数据集上的分类问题：*
> *1、梯度下降法；*
> *2、随机梯度法；*
> *3、随机平均梯度法SAG（选做）。*
> *对于每种算法，请给出每步计算结果在测试集上所对应的分类精度。对于随机梯度法，请讨论mini-batch 大小的影响。可采用Logistic Regression 模型或神经网络模型。*

代码

环境的搭建

由于要运行神经网络模型，因此需要搭建pytorch深度学习框架的环境

- 下载并安装Anaconda3和vscode
  由于之前的课程需要，已经安装过了Anaconda3和vscode

- 搭建环境

- 之前安装过一个pytorch的虚拟环境，python版本为3.8.8，并且也已经激活了这个虚拟环境：

```
PS D:\> conda info -e
# conda environments:
#
base                  *  D:\Download_program\Anaconda3\anaconda3
hw                       D:\Download_program\Anaconda3\anaconda3\envs\hw
pytorch                  D:\Download_program\Anaconda3\anaconda3\envs\pytorch
```

- 可以使用指令

```
$ conda activate pytorch
```

进入这个虚拟环境：

```
PS D:\>  conda activate pytorch
(pytorch) PS D:\>
```

- 查看安装的torch以及torchvision版本：

```
(pytorch) PS D:\> python
Python 3.8.8 (default, Apr 13 2021, 15:08:03) [MSC v.1916 64 bit (AMD64)] :: Anaconda, Inc. on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import torch
>>> torch.__version__
'1.11.0'
>>>
>>> import torchvision
>>> torchvision.__version__
'0.12.0'
```

- 这里我遇到的一些问题:
  - 在之前安装Anaconda3后发现缺失大量文件，如scripts、Library等，最后重装了几遍也没发现问题，更改了安装的Anaconda3版本的安装包后才正常安装。
  - 在powershell中激活虚拟环境的时候，使用指令“conda init powershell”，会出现“CommandNotFoundError: No command 'conda init'的报错，最后发现是因为conda的版本过老，将其更新至4.6就不再出现上面的报错，能够正常进入虚拟环境了。

```
(pytorch) PS D:\> conda -V
conda 4.6.7
(pytorch) PS D:\>
```

  - 还有需要注意的是安装torch和torchvision时，他们的版本要对应，否则会报错：

torch	torchvision	python
main / nightly	main / nightly	>=3.7 , <=3.10
1.11.0	0.12.0	>=3.7 , <=3.10
1.10.2	0.11.3	>=3.6 , <=3.9
1.10.1	0.11.2	>=3.6 , <=3.9
1.10.0	0.11.1	>=3.6 , <=3.9
1.9.1	0.10.1	>=3.6 , <=3.9
1.9.0	0.10.0	>=3.6 , <=3.9
1.8.2	0.9.2	>=3.6 , <=3.9
1.8.1	0.9.1	>=3.6 , <=3.9
1.8.0	0.9.0	>=3.6 , <=3.9

  - 还有在安装torchvision的时候会有报错：“Could not find a version that satisfies the requirement torchvision (from versions: none)”，最后发现可能是源出了问题，可以更换一个镜像源，在“pip install XXX”的命令后加上：（以清华源为例）

```
--default-timeout=100 -i https://pypi.tuna.tsinghua.edu.cn/simple
```

- 在vscode中搭建pytorch相关环境：
  - 首先安装相关python的一些插件，这些插件之前已经安装过
  - 配置环境

    在“File->Preferences->Settings->Extensions->Python->setting.json”中添加：

    ```
    "python.defaultInterpreterPath":"D\\Anaconda3\\anaconda3\\envs\\pytorch\\python.exe"
    ```

    后面这不服路径就是我的虚拟环境中的python执行文件的路径。

    但是有一点特别坑的是，很多教程上写的添加python路径的代码是：

    ```
    “python.pythonPath” :“D:\\anaconda3\\envs\\xiaolvshijie\\python.exe”
    ```

    但是这条指令已经过时了并没有用。
  - 之后在vscode打开终端就能够自动进入我的pytorch虚拟环境了：

    ```
    尝试新的跨平台 PowerShell https://aka.ms/pscore6

    加载个人及系统配置文件用了 1061 毫秒。
    (base) PS D:\code\pytorch> D:/Download_program/Anaconda3/anaconda3/Scripts/activate
    (base) PS D:\code\pytorch> conda activate pytorch
    (pytorch) PS D:\code\pytorch> python
    Python 3.8.8 (default, Apr 13 2021, 15:08:03) [MSC v.1916 64 bit (AMD64)] :: Anaconda, Inc. on w
    in32
    Type "help", "copyright", "credits" or "license" for more information.
    >>> import torch
    >>> torch.__version__
    '1.11.0'
    >>> import torchvision
    >>> torchvision.__version__
    '0.12.0'
    >>>
    ```

代码解释

- 首先导入一些需要的库：

```
import torch
import torch.nn as nn             # 建立神经网络模块
import torch.utils.data as Data   # 用来包装批处理的数据
import torchvision                # 关于图像操作的一些方便工具库
import cv2                        # 显示图片
```

- 设置一些超参数：

```
torch.manual_seed(1)   # 使用随机化种子使神经网络的初始化每次都相同

# 设置超参数
EPOCH = 8               # 训练整批数据的次数
BATCH_SIZE = 50         # 每个batch加载多少个样本
LR = 0.0001             # 学习率
DOWNLOAD_MNIST = False  # True
# 决定是否下载数据集，如果写的是True表示还没有下载数据集，如果数据集下载好了就写False
```

- 对MINIST图像集的加载和处理
  - 加载训练集和测试集，并根据超参数的设置决定是否要下载MINIST图像集：

```
# 训练数据
```

```
train_data = torchvision.datasets.MNIST(
    root='./data/',  # 保存或提取的位置  会放在当前文件夹中
    train=True,      # true说明是用于训练的数据，false说明是用于测试的数据
    transform=torchvision.transforms.ToTensor(),
    # 转换PIL.Image or numpy.ndarray

    download=DOWNLOAD_MNIST,  # 已经下载了就不需要下载了
)

# 测试数据
test_data = torchvision.datasets.MNIST(
    root='./data/',# 提取的位置
    train=False     # 表明不是训练集是测试集
)
```

- 将原来的数据打包成很多批次，每个批次传入设计的神经网络进行训练，BATCH_SIZE就是我们设置的每个批次里包含的样本数量：

```
# 批训练 50个samples， 1  channel，28x28 (50,1,28,28)
# Torch中的DataLoader是用来包装数据的工具，它能帮我们有效迭代数据，这样就可以进行批训练
train_loader = Data.DataLoader(
    dataset=train_data,
    batch_size=BATCH_SIZE,
    shuffle=True            # 是否打乱数据，一般都打乱
)
```

- 然后再从测试样本中取出2000个再接下来进行测试的时候使用，只取2000个是为了节约时间：

```
# 测试数据
test_x = torch.unsqueeze(test_data.data, dim=1).type(torch.FloatTensor)[:2000]/255
# torch.unsqueeze(a) 是用来对数据维度进行扩充
# 这样shape就从(2000,28,28)->(2000,1,28,28)
# 图像的pixel本来是0到255之间，除以255对图像进行归一化使取值范围在(0,1)
test_y = test_data.targets[:2000]
```

- 然后用class类来建立CNN模型

  先建立它的网络结构，设计每个模块的结构，然后将整个流程串接起来。

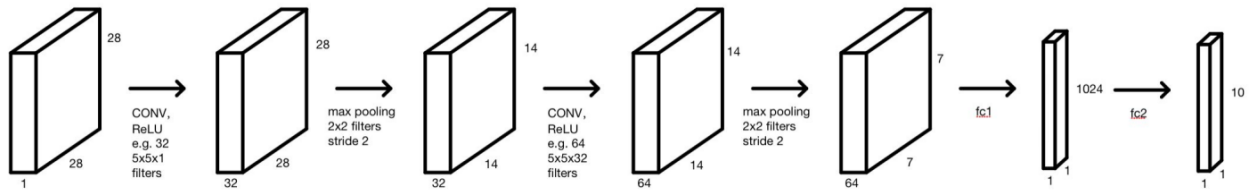
  这里设计的**CNN模型**的流程是：

  输入数据(1,28,28) ->

  卷积(Conv2d)(16,28,28) -> 激励函数(ReLU)(16,28,28) -> 池化(MaxPooling)(16,14,14) ->

  卷积(Conv2d)(32,14,14) -> 激励函数(ReLU)(32,14,14) -> 池化(MaxPooling)(32,7,7)->

  展开多维的卷积成的特征图(1,32*7*7)->接入全连接层(Linear)(10)->输出(输入的图像属于0-9的概率)




```
# 用class类来建立CNN模型

class CNN(nn.Module):  # 我们建立的CNN继承nn.Module这个模块
    def __init__(self):
```

```
        super(CNN, self).__init__()
        # 建立第一个卷积(Conv2d)-> 激励函数(ReLU)->池化(MaxPooling)
        # 输入(1,28,28)的图像，经过16个5*5的卷积核卷积，得到16个28*28的特征图
        self.conv1 = nn.Sequential(
            # 第一个卷积con2d
            nn.Conv2d(              # 输入图像大小(1,28,28)
                in_channels=1,      # 输入图片的高度，
                                    # minist数据集是灰度图像因此只有一个通道
                out_channels=16,    # n_filters 卷积核的高度
                kernel_size=5,      # filter size 卷积核的大小 也就是长x宽=5x5
                stride=1,           # 步长
                padding=2,          # 想要con2d输出的图片长宽不变，就进行补零操作
                                    # padding = (kernel_size-1)/2
            ),                      # 输出图像大小(16,28,28)
            # 激活函数
            nn.ReLU(), # nn.Sigmoid(),
            # 池化，下采样
            nn.MaxPool2d(kernel_size=2),  # 在2x2空间下采样，使用最大值池化
            # 输出图像大小(16,14,14)
        )

        # 建立第二个卷积(Conv2d)-> 激励函数(ReLU)->池化(MaxPooling)
        self.conv2 = nn.Sequential(
            # 输入图像大小(16,14,14)
            nn.Conv2d(          # 也可以直接简化写成nn.Conv2d(16,32,5,1,2)
                in_channels=16,
                out_channels=32,
                kernel_size=5,
                stride=1,
                padding=2
            ),
            # 输出图像大小 (32,14,14)
            nn.ReLU(),        # nn.Sigmoid(), 激活函数
            nn.MaxPool2d(2), # 池化
            # 输出图像大小(32,7,7)
        )

        # 建立全卷积连接层
        self.out = nn.Linear(32 * 7 * 7, 10)  # 输出是10个类

    # 下面定义x的传播路线
    def forward(self, x):
        x = self.conv1(x)  # x先通过conv1
        x = self.conv2(x)  # 再通过conv2
        # 把每一个批次的每一个输入都拉成一个维度，即(batch_size,32*7*7)
        # 因为pytorch里特征的形式是[bs,channel,h,w]，所以x.size(0)就是batchsize
        x = x.view(x.size(0), -1)  # view就是把x弄成batchsize行个tensor
        output = self.out(x)
        return output


cnn = CNN()
print(cnn)
```

可以输出这个CNN结构，可以看到和我们设计的结构吻合：

```
(pytorch) PS D:\code\pytorch> python test.py
CNN(
  (conv1): Sequential(
    (0): Conv2d(1, 16, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (conv2): Sequential(
    (0): Conv2d(16, 32, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (out): Linear(in_features=1568, out_features=10, bias=True)
)
```

- 然后设置训练这个模型的优化器以及损失函数，都是利用库中的函数

（之后有对梯度下降和随机梯度下降方法的介绍）

Adam结合了AdaGrad和RMSProp两种优化算法的优点。对梯度的一阶矩估计（First Moment Estimation，即梯度的均值）和二阶矩估计（Second Moment Estimation，即梯度的未中心化的方差）进行综合考虑，计算出更新步长，即可以自适应地调整学习率。综合Adam在很多情况下算作默认工作性能比较优秀的优化器。

损失函数使用交叉熵来进行计算。

```
# 优化器选择Adam
optimizer = torch.optim.Adam(cnn.parameters(), lr=LR)
# 损失函数，交叉熵
loss_func = nn.CrossEntropyLoss()  # 目标标签是one-hotted
```

- 然后开始训练这个模型：

整个训练的过程就是先进行前向传播，计算输出结果，然后计算与真实值之间的误差，再进行反向传播，每传入一层，就将这一层的参数根据loss进行更新。然后将所有的训练样本按BATCH_SIZE个为一组进行训练，所有的训练样本都训练过一次后就是一个EPOCH，将此时的训练结果进行输出，主要输出的是训练样本的loss（由于全部样本过多，因此只输出每一次EPOCH的最后一个批次的loss情况），以及测试样本的准确率。

训练结束后保存CNN模型，以便之后可以直接加载这个模型进行使用，不需要再进行训练了。

```
# 开始训练
for epoch in range(EPOCH):
    for step, (b_x, b_y) in enumerate(train_loader):  # 分配batch data
        output = cnn(b_x)       # 先将数据放到cnn中计算output
        loss = loss_func(output, b_y)  # 输出和真实标签的loss，二者位置不可颠倒
        optimizer.zero_grad()  # 清除之前学到的梯度的参数
        loss.backward()         # 反向传播，计算梯度
        optimizer.step()        # 应用梯度

    test_output = cnn(test_x)
    pred_y = torch.max(test_output, 1)[1].data.numpy()
    # 指按照行来找，找到最大值（即每个测试样本中概率最大的）。
    # [1]指的是输出最大值的下标，即我们识别到的数字
    accuracy = float((pred_y == test_y.data.numpy()).astype(int).sum()) /
float(test_y.size(0))
    print('Epoch: ', epoch, '| train loss: %.6f' % loss.data.numpy(), '| test accuracy:
%.6f' % accuracy)

torch.save(cnn.state_dict(), 'cnn.pkl')#保存模型
```

- 选择图片进行识别测试：

```
# 加载模型，调用时不需要再将原本的模型重新进行训练
```

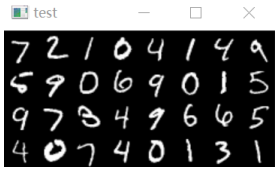
```
cnn.load_state_dict(torch.load('cnn.pkl'))
cnn.eval()

# print 10 predictions from test data
inputs = test_x[:32]        # 测试32个数据
test_output = cnn(inputs)   # 传入CNN进行计算
pred_y = torch.max(test_output, 1)[1].data.numpy() # 将每一行的最大值的下标输出
print('prediction number:')
print(pred_y)               # 打印识别后的数字

img = torchvision.utils.make_grid(inputs) # 将32个测试图像形成一张图
img = img.numpy().transpose(1, 2, 0)
# Pytorch中为[Channels, H, W]，而plt.imshow()中则是[H, W, Channels]
# 因此，要先转置一下。

# opencv显示需要识别的数据图片
cv2.imshow('test', img)
key_pressed = cv2.waitKey(0)
```

可以看到最后测试的32个图像：



输出结果：

```
(pytorch) PS D:\code\pytorch> python test.py
prediction number:
[7 2 1 0 4 1 4 9 5 9 0 6 9 0 1 5 9 7 3 4 9 6 6 5 4 0 7 4 0 1 3 1]
(pytorch) PS D:\code\pytorch>
```

结果都正确。

还可以输出训练过程中每个轮次的loss和准确率（注：这是使用优化器Adam来计算得到的结果，之后有对梯度下降和随机梯度下降训练的结果）：

```
(pytorch) PS D:\code\pytorch> python test.py
Epoch:  0 | train loss: 0.106451 | test accuracy: 0.972000
Epoch:  1 | train loss: 0.010249 | test accuracy: 0.978000
Epoch:  2 | train loss: 0.004339 | test accuracy: 0.984000
Epoch:  3 | train loss: 0.021179 | test accuracy: 0.982500
Epoch:  4 | train loss: 0.006234 | test accuracy: 0.982500
Epoch:  5 | train loss: 0.009494 | test accuracy: 0.988500
Epoch:  6 | train loss: 0.015029 | test accuracy: 0.991000
Epoch:  7 | train loss: 0.026549 | test accuracy: 0.990000
```

训练结果

使用不同的激活函数进行训练

sigmoid 函数

$$sigmoid(x) = \frac{1}{1+e^{-x}}$$

会将 (-∞,+∞) 范围内的输入映射到 (-1,1)。sigmoid 函数有较好的解释性，但是也有一些明显的缺点：当神经元的激活在接近 0 或 1 处时会饱和，Sigmoid 函数饱和使梯度消失，在这些区域，梯度几乎为 0。并且使用这个函数在深度神经网络时候相比其他激活函数相比较慢。

Relu 函数

$$Relu(x) = max(0, x)$$

相比于上面提到的饱和激活函数 sigmoid ，非饱和激活函数 2 可以解决“梯度消失”的问题，加快收敛。修正非线性单元（Rectified Linear Units (ReLUs)）就是一种非饱和激活函数。就梯度下降法的训练时间而言，使用 ReLU 做为激活函数的大型卷积神经网络就比起使用 tanh 单元作为激活函数的训练起来要快。

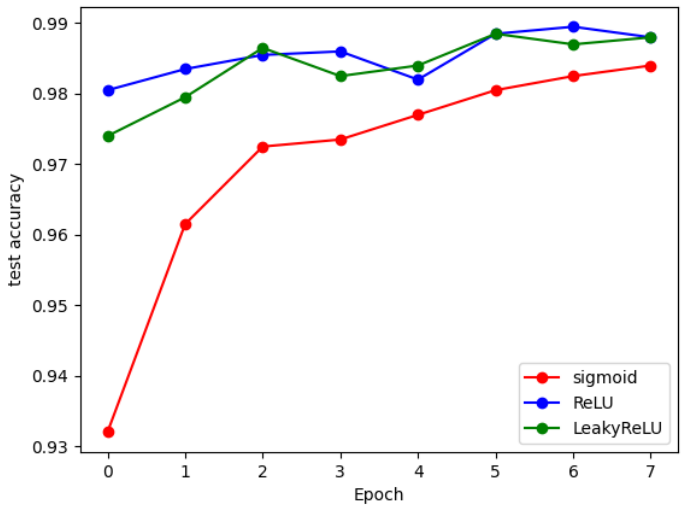
但这个激活函数也有一些明显的不足：当 x 是小于 0 的时候，那么从此所以流过这个神经元的梯度将都变成 0；这个时候这个 ReLU 单元在训练中将死亡（也就是参数无法更新），也就导致了数据多样化的丢失（因为数据一旦使得梯度为 0，也就说明这些数据已不起作用）。

LeakyReLU 函数

Relu函数通过裁剪负值，消除了梯度问题，但同时带来了Dead ReLU问题，即随着训练的进行，网络中的某些输出会变为0并且再也不会恢复。为了消除这一现象，有人有提出了Leaky ReLU或 Parametric ReLU(PReLU)等Relu函数的变体，这两种变体可以统一用以下 计算公式 来表示：

$$LeakyReLU(x) = max(\alpha x, x)$$

将使用这三个不同激活函数来训练CNN的结果（测试样本的准确度）进行绘图：



很明显，可以看出LeakyReLU 和 ReLU 的训练效果要比sigmoid 函数的效果更好。

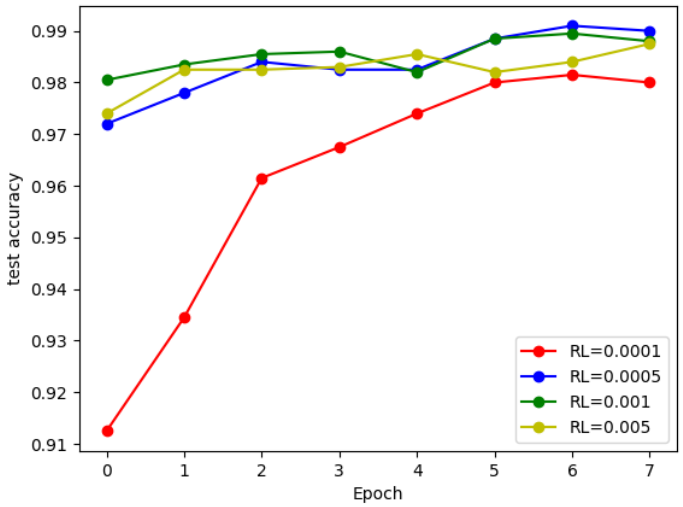
下面是三种激活函数输出的数据（包括每个轮次里训练样本的最后一个批次的loss值）：

	sigmoid		**ReLU**		**LeakyReLU**	
Epoch	train loss	test accuracy	train loss	test accuracy	train loss	test accuracy
0	0.306262	0.932000	0.097391	0.980500	0.120878	0.974000
1	0.022345	0.961500	0.017261	0.983500	0.010664	0.979500
2	0.017852	0.972500	0.001939	0.985500	0.003178	0.986500
3	0.070685	0.973500	0.006908	0.986000	0.033206	0.982500
4	0.013153	0.977000	0.003283	0.982000	0.004099	0.984000

	sigmoid		ReLU		LeakyReLU	
5	0.042432	0.980500	0.002672	0.988500	0.004242	0.988500
6	0.017355	0.982500	0.007232	0.989500	0.008922	0.987000
7	0.015131	0.984000	0.023573	0.988000	0.030906	0.988000

通过调整学习率增加准确率

- 虽然Adam会自适应调整学习率，但是学习率会有一个初始值，这个值是一个超参数，也会影响训练结果：
  这里设置RL分别为0.001、0.005、0.0005以及0.0001来进行训练：



可以看出在最后的训练轮次中RL=0.0005的效果是最好的，测试样本的准确率达到了0.991000
他们的train loss以及test accuracy如下：

	RL=0.0001		RL=0.0005		RL=0.001		RL=0.005	
Epoch	train loss	test accuracy	train loss	test accuracy	train loss	test accuracy	train loss	test accuracy
0	0.265836	0.912500	0.106451	0.972000	0.097391	0.980500	0.048787	0.974000
1	0.051537	0.934500	0.010249	0.978000	0.017261	0.983500	0.011305	0.982500
2	0.026979	0.961500	0.004339	0.984000	0.001939	0.985500	0.000504	0.982500
3	0.077339	0.967500	0.021179	0.982500	0.006908	0.986000	0.008578	0.983000
4	0.031563	0.974000	0.006234	0.982500	0.003283	0.982000	0.005992	0.985500
5	0.085501	0.980000	0.009494	0.988500	0.002672	0.988500	0.004629	0.982000
6	0.019264	0.981500	0.015029	0.991000	0.007232	0.989500	0.000264	0.984000
7	0.055292	0.980000	0.026549	0.990000	0.023573	0.988000	0.010241	0.987500

1、梯度下降法

首先写出一个损失函数(交叉熵)，然后使用梯度下降法来进行最小化。

```
# 损失函数，交叉熵
loss_func = nn.CrossEntropyLoss()  # 目标标签是one-hotted

b_x = torch.unsqueeze(train_data.data, dim=1).type(torch.FloatTensor)[:55000]/255
b_y = train_data.targets[:55000]
# 梯度下降法，使用全部训练样本
for epoch in range(EPOCH):
    output = cnn(b_x)  # 先将数据放到cnn中计算output
    loss = loss_func(output, b_y)  # 输出和真实标签的loss，二者位置不可颠倒
    optimizer.zero_grad()  # 清除之前学到的梯度的参数
    loss.backward()  # 反向传播，计算梯度
    optimizer.step()  # 应用梯度


    test_output = cnn(test_x)
    pred_y = torch.max(test_output, 1)[1].data.numpy() # 指按照行来找，找到最大值（即每个测试样本中概率最大的）。[1]指的是输出最大值的下标，即我们识别到的数字
    accuracy = float((pred_y == test_y.data.numpy()).astype(int).sum()) / float(test_y.size(0))
    print('Epoch: ', epoch, '| train loss: %.6f' % loss.data.numpy(), '| test accuracy: %.6f' % accuracy)
```

由于梯度下降法是要将全部数据计算梯度进行平均后再进行更新参数，时间非常慢，因此这里只迭代了20次，RL=0.01，准确率达到了0.8，如果再进行训练，准确度应该会更高：

2、随机梯度法

```
# 优化器选择SGD
optimizer = torch.optim.SGD(cnn.parameters(), lr=LR)
# 损失函数，交叉熵
loss_func = nn.CrossEntropyLoss()  # 目标标签是one-hotted

# 开始训练
for epoch in range(EPOCH):
    for step, (b_x, b_y) in enumerate(train_loader):  # 分配batch data
        output = cnn(b_x)  # 先将数据放到cnn中计算output
        loss = loss_func(output, b_y)  # 输出和真实标签的loss，二者位置不可颠倒
        optimizer.zero_grad()  # 清除之前学到的梯度的参数
        loss.backward()  # 反向传播，计算梯度
        optimizer.step()  # 应用梯度


    test_output = cnn(test_x)
    pred_y = torch.max(test_output, 1)[1].data.numpy() # 指按照行来找，找到最大值（即每个测试样本中概率最大的）。[1]指的是输出最大值的下标，即我们识别到的数字
    accuracy = float((pred_y == test_y.data.numpy()).astype(int).sum()) / float(test_y.size(0))
    print('Epoch: ', epoch, '| train loss: %.6f' % loss.data.numpy(), '| test accuracy: %.6f' % accuracy)
```

讨论 **mini-batch** 大小的影响:

不同mini-batch分别他们的准确度对应的结果：

从他们单个的结果来看，整体都是越训练，效果越好，但是看不出他们之间的关系，可以将它们放入同一张图：

这样就可以比较出来，mini-batch越小，每个epoch里训练的轮次也就越多，准确率也就越高。mini-batch越大，每次迭代需要计算的数据量也就越大，要达到相同的准确率，需要的epoch也就越多，速度也就越慢。

总结

本次大作业让我对优化算法中的几个梯度下降算法有了更深刻的认识，尤其是第一题的三个算法。他们的算法部分更重要，代码比较好写，虽然因为矩阵的问题还是改了很多bug。二我刚开始往往是本来按照自己的理解写出来算法觉得是对的，但是第二天看以前的笔记以及一些基本概念，发现自己理解又有问题，又推翻重写，尤其是交替方向乘子算法，由于我对一些i的理解问题，导致重写了几遍算法。并且这次作业让我对投影、软阈值法都能更理解了。

第二题相较于算法，环境的搭建以及代码要更难写得多。主要是要能理解整个代码的流程，以及每一个板块是在做什么，输入了什么内容，在最后输出的又是什么结果，其实整个过程非常清楚，就是要具体理解一些库的使用。由于之前写过CNN相关的综述，因此对整个过程以及结构有大概得认识。环境的搭建也出现了很多问题，比如之前安装Anaconda3时候因为一些版本问题导致反复重装，而且在配置环境过程中遇到的种种报错。我的心得就是多看博客多尝试，可能这个博客说的问题和我遇到的报错不完全相同，但还是可以取其部分内容进行尝试，而不要说一定找到一个和我遇到的问题完全相同的博客才能解决我的问题。

总之完成本次大作业后，收获良多。

reference

- 领近点梯度下降法、交替方向乘子法、次梯度法使用实例（Python实现）*肥宅*Sean的博客-CSDN博客_临近点梯度下降法
- 【Python】用邻近点梯度法、交替方向乘子法、次梯度法求解 LASSO 回归模型 - Jed伪极客 (jeddd.com)
- 凸优化：邻近点梯度法、交替方向乘子法、次梯度法 matlab实现*无聊的小把戏的博客-CSDN博客*matlab交替方向法
- 中科大凸优化51-52：交替方向乘子法 - 知乎 (zhihu.com)
- 优化算法-1|拉格朗日函数和对偶性 - 知乎 (zhihu.com)
- 优化算法-3|增广拉格朗日函数(ALM)及其优化方法-ADMM算法、AMA算法 - 知乎 (zhihu.com)
- Python中的赋值(复制)、浅拷贝与深拷贝 - 知乎 (zhihu.com)
- 最优化方法 18：近似点算子 Proximal Mapping_Bonennult的博客-CSDN博客_近似点算子
- 18-lect-prox_map.pdf (pku.edu.cn)（北大ppt）
- 软阈值(Soft Thresholding)函数解读*jbb0523的博客-CSDN博客*软阈值
-
- 临近梯度下降算法（Proximal Gradient Method）的推导以及优势 - kkzhang - 博客园 (cnblogs.com)
- 凸优化ADMM(Alternating Direction Method of Multipliers)交替方向乘子算法 - 胖虎的博客 | Kimmy's Blog (jiaqianlee.com)
- 交替方向乘子法（Alternating Direction Multiplier Method，ADMM） - kkzhang - 博客园 (cnblogs.com)

- 凸优化学习-（二十九）有约束优化算法——增广拉格朗日法、交替方向乘子法（ADMM）*明远湖边的秃头的博客-CSDN博客*交替方向乘子法和增广拉格朗日方法的区别和联系?