最优化概论期末大作业

19335019 陈泷

问题1

问题描述

一、考虑一个 20 节点的分布式系统。节点 i 有线性测量 $b_i=A_ix+e_i$，其中 b_i 为 10 维的测量值，A_i 为 10×300 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：

$$\min\ (1/2)||A_1x-b_1||_2^2+\cdots+(1/2)||A_{20}x-b_{20}||_2^2+p||x||_1$$

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。

首先根据题目要求生成对应数据，便于后面求解。

```python
"""
定义相关常量
"""
node_num = 20 # 节点总数
A_dim = (10, 300)  # A: 10*300
A_all_dim = (node_num, 10, 300)
x_dim = 300  # x: 300*1
x_sparsity = 5  # 稀疏度为5
e_dim = 10  # noise
b_all_dim = (node_num, 10)

"""
生成数据
"""
# 生成x的真值
x_nonzero_index = random.sample(range(x_dim), x_sparsity)  # 非0元素的下标
x_nonzero_element = np.random.normal(0, 1, x_sparsity)
x_real = np.zeros(x_dim)
x_real[x_nonzero_index] = x_nonzero_element
# print(f'x_real:{x_real}')

A = np.zeros(A_all_dim)
e = np.zeros(b_all_dim)
b = np.zeros(b_all_dim)
for i in range(node_num):
    # 生成测量矩阵A
```

```
    A[i] = np.random.normal(0, 1, A_dim)
    # 生成测量噪声e
    e[i] = np.random.normal(0, 0.2, e_dim)
    # 计算带噪声的b
    b[i] = A[i] @ x_real + e[i]
```

邻近点梯度法

1 算法设计

优化问题：$min\ f_0(x) = \frac{1}{2}\|A_1x - b_1\|_2^2 + \ldots + \frac{1}{2}\|A_{20}x - b_{20}\|_2^2 + p\|x\|_1$

在本题中，$s(x) = \frac{1}{2}\|A_1x - b_1\|_2^2 + \ldots + \frac{1}{2}\|A_{20}x - b_{20}\|_2^2$，$r(x) = p\|x\|_1$，

算法如下：

$$x^{k+\frac{1}{2}} = x^k - \alpha \times \sum_{i=1}^{20} A_i^T(A_ix^k - b_i)$$
$$x^{k+1} = \arg\min_x\{p\|x\|_1 + \frac{1}{2\alpha}\left\|x - x^{k+\frac{1}{2}}\right\|^2\}$$

由软门限可得，

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

在20个节点中，不妨设节点1为master，剩余节点为worker。工作流程如下：
（1）节点1发送 x^k 给其他节点
（2）所有节点计算各自的$A_i^T(A_ix^k - b_i)$，并发送给节点1
（3）节点1根据如下公式，计算出新的 x^k
（4）重复（1）~（3）直到算法收敛或到达最大迭代次数

$$x^{k+\frac{1}{2}} = x^k - \alpha \times \sum_{i=1}^{20} A_i^T(A_ix^k - b_i)$$
$$x^{k+1} = \arg\min_x\{p\|x\|_1 + \frac{1}{2\alpha}\left\|x - x^{k+\frac{1}{2}}\right\|^2\}$$

```
while k < max_iteration_times:
    '''Step1: master节点发送 x ^ k 给其他节点
            在此不进行模拟'''

    '''Step2:所有worker节点计算各自值，并传送给master节点
            这里在模拟时计算后直接加入x ^ {k + 1 / 2}中'''
    x_k_temp = x_k.copy()  # master节点数据：记录 x^{k+1/2}
    for i in range(node_num):
        x_k_temp -= alpha * A[i].T @ (A[i] @ x_k - b[i])

    '''Step3:master节点根据公式，计算出新的 x^k
            模拟时以下工作都由master节点实现'''
    # 临近点投影:使用软门限进行求值
    for i in range(x_dim):
        if x_k_temp[i] < -alpha * p:
            x_k[i] = x_k_temp[i] + alpha * p
        elif x_k_temp[i] > alpha * p:
            x_k[i] = x_k_temp[i] - alpha * p
        else:
```

```
            x_k[i] = 0
    iteration_result.append(x_k.copy())  # 记录每步计算结果

    '''Step4:重复（1）~（3）直到算法收敛或到达最大迭代次数'''
    if np.linalg.norm(x_k - x_k_pre) < epsilon:  # 如果误差小于精度，则视为收敛，退出循环
        break
    else:  # 如果误差大于精度，则视为未收敛，开始新一轮迭代
        x_k_pre = x_k.copy()  # 深拷贝
        k += 1
```

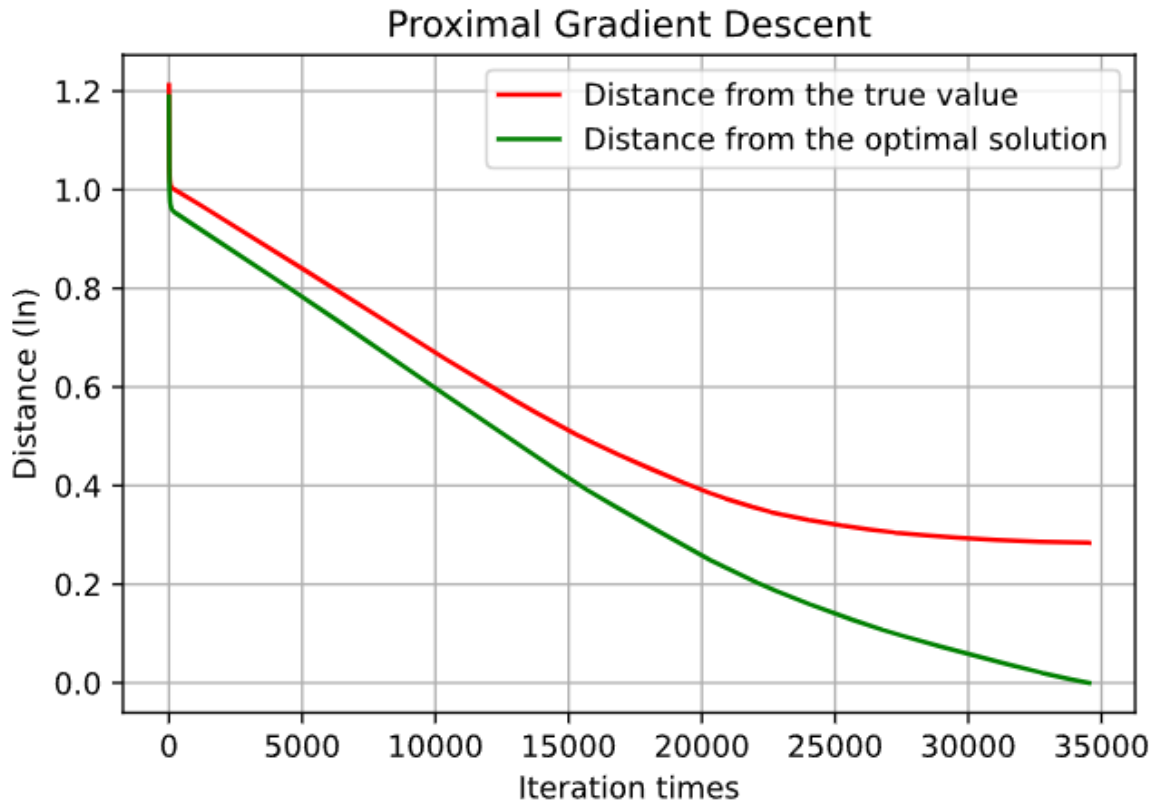
关键步骤代码如上所示。

数据处理：计算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。需要注意的是，对结果取 ln 值：取log是为了便于观察线性收敛；+1是为了防止取log后最终结果接近负无穷导致溢出。代码如下：

```
for x_i in iteration_result:
    # 取 ln 值：取log是为了便于观察线性收敛；+1是为了防止取log后最终结果接近负无穷导致溢出
    dist_real.append(np.log(np.absolute(np.linalg.norm(x_i - x_real) + 1)))
    dist_opt.append(np.log(np.absolute(np.linalg.norm(x_i - x_opt) + 1)))
```

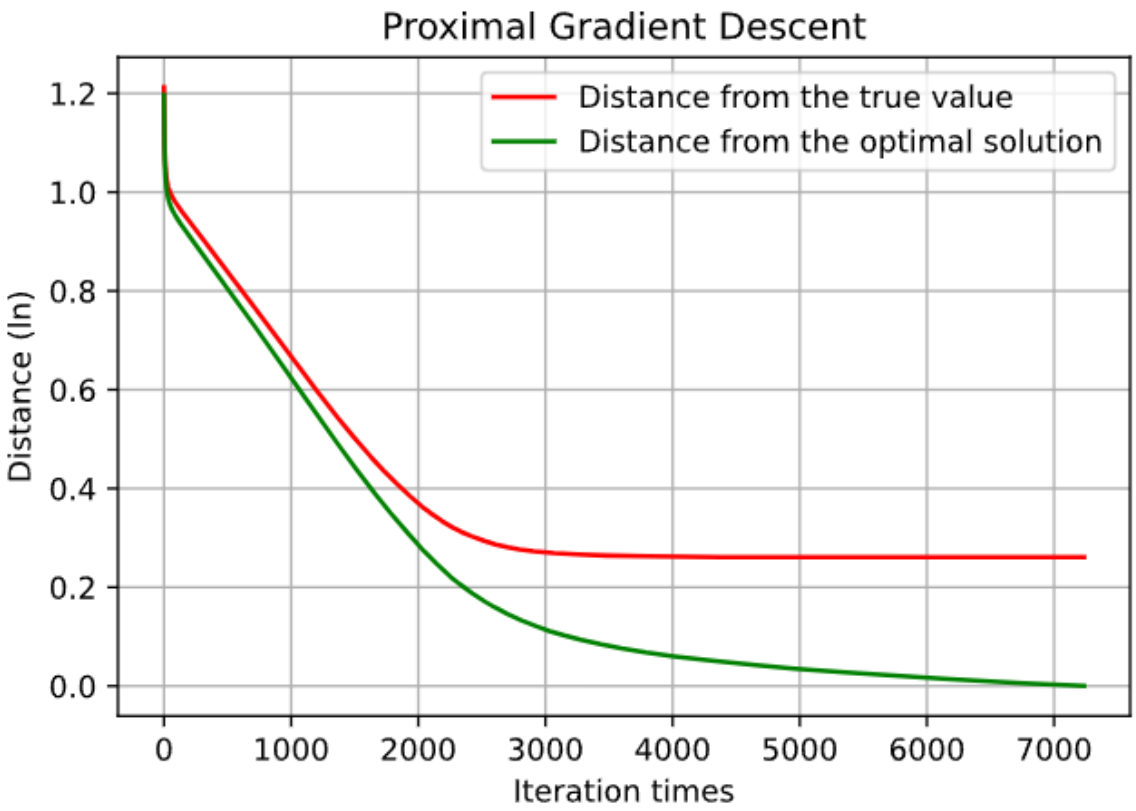
2 数值实验

2.1 p=0.01



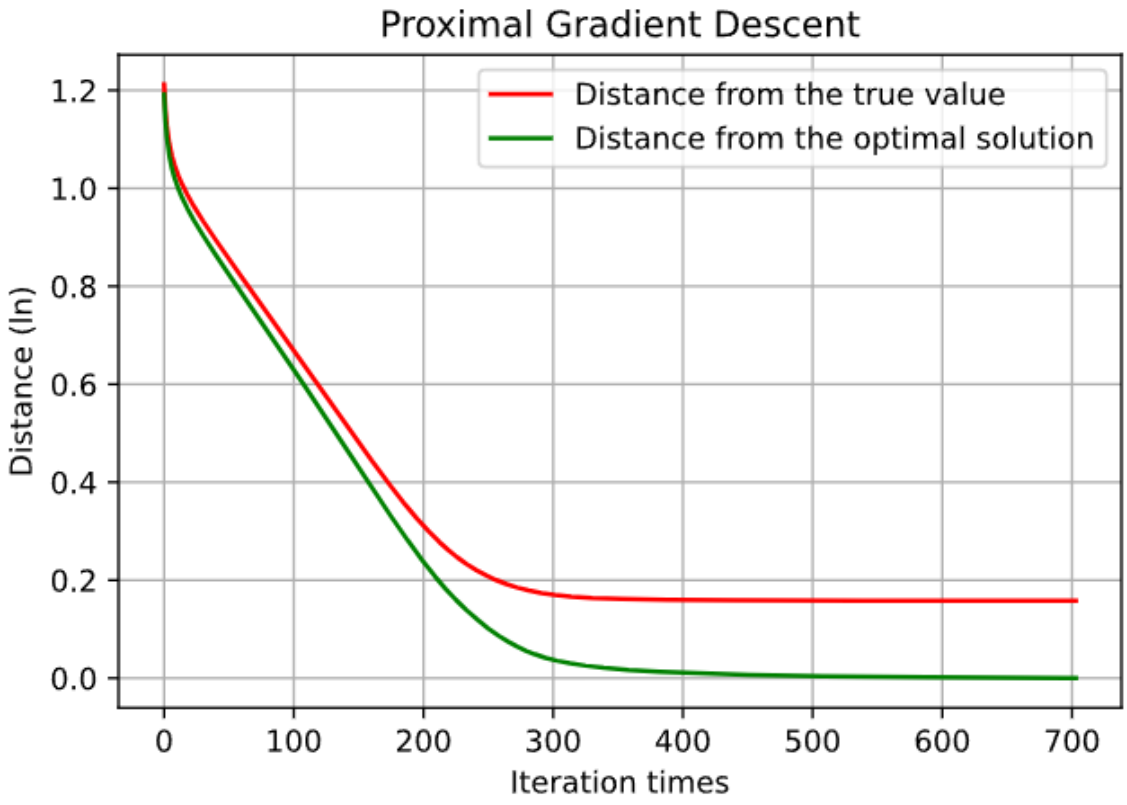
最优解的稀疏度：224

2.2 p=0.1



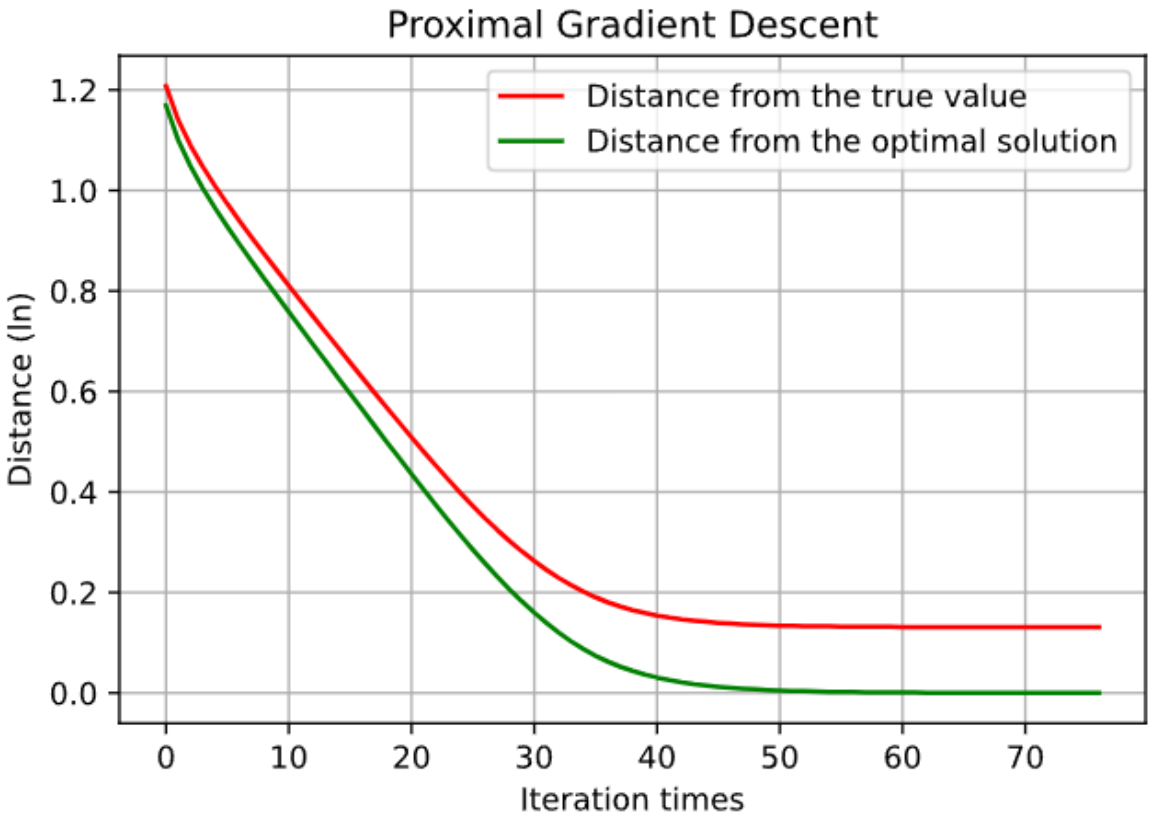
最优解的稀疏度：197

2.3 p=1



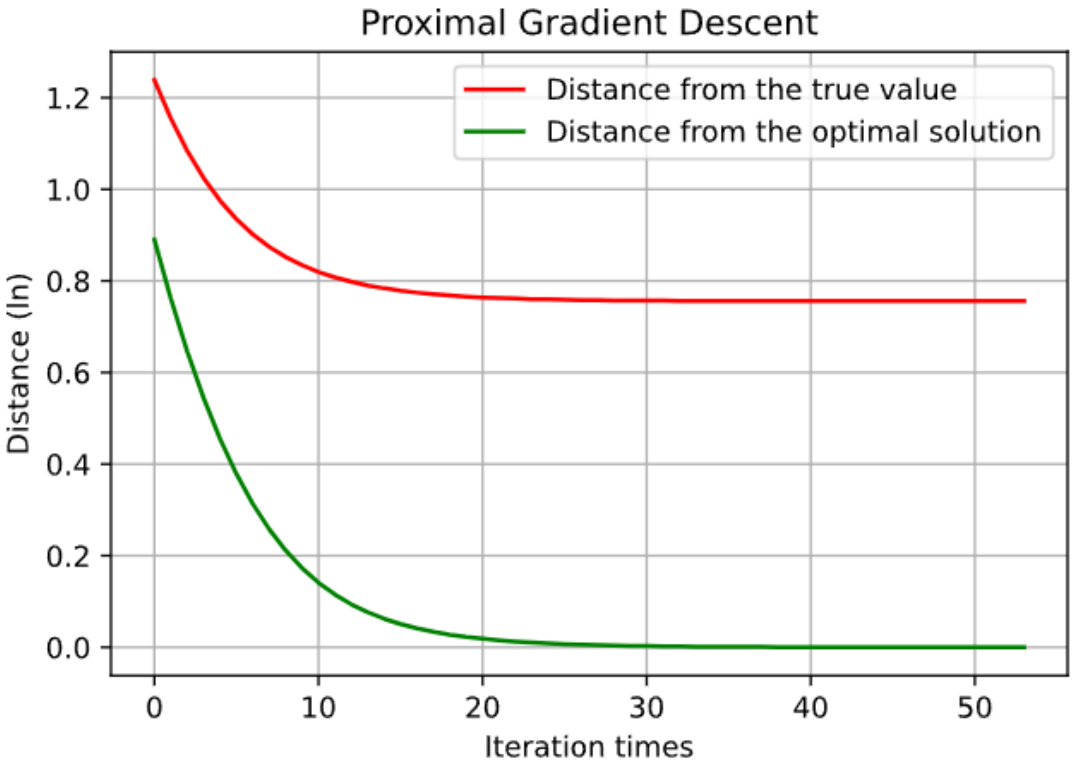
最优解的稀疏度：142

2.4 p=10



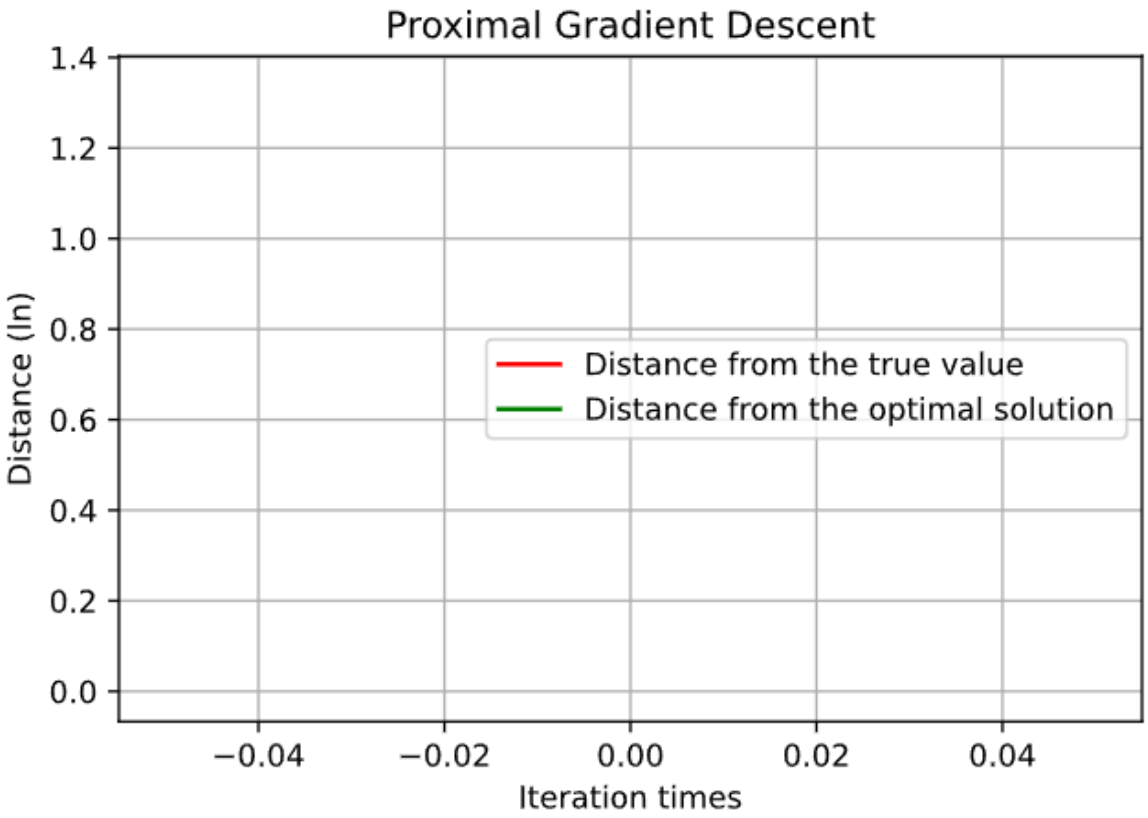
最优解的稀疏度：5

2.5 p=100



最优解的稀疏度：4

2.6 p=1000



最优解的稀疏度：0

3 结果分析

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如图像所示。

由图像可知，p 越大，最优解越稀疏，收敛速度越快，迭代次数越少。在一定限度内，p 越大，最优解越接近真实解，超过该限度后，最优解越来越稀疏，因此越来越接近0，因此此时最优解和真实解的距离不断增大。

交替方向乘子法

1 算法设计

优化问题：$min\ f_0(x) = \frac{1}{2}\|A_1x - b_1\|_2^2 + \ldots + \frac{1}{2}\|A_{20}x - b_{20}\|_2^2 + p\|x\|_1$

为了应用交替方向乘子法，将问题修改为：

$$\begin{aligned} &\min\ \frac{1}{2}\|A_1x - b_1\|_2^2 + \ldots + \frac{1}{2}\|A_{20}x - b_{20}\|_2^2 + p\|y\|_1 \\ &\text{s.t. } x - y = 0 \end{aligned}$$

在本题中，算法迭代公式如下：

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

在上面的公式中，

x^{k+1} 的求解公式 $\frac{1}{2}\sum_{i=1}^{20}\|A_ix-b_i\|_2^2+<\lambda^k,x>+\frac{c}{2}\left\|x-y^k\right\|_2^2$ 可微，因此令其导数为0可以进行求解。

y^{k+1} 中有1范数不可微，因此通过软门限进行求解。

λ^{k+1} 直接套用公式即可求解。

在20个节点中，不妨设节点1为master，剩余节点为worker。工作流程如下：

（1）节点1发送 x^k 给其他节点

（2）所有节点计算各自的$\|A_ix-b_i\|_2^2$，并发送给节点1

（3）节点1根据迭代公式，计算出新值

（4）重复（1）~（3）直到算法收敛或到达最大迭代次数

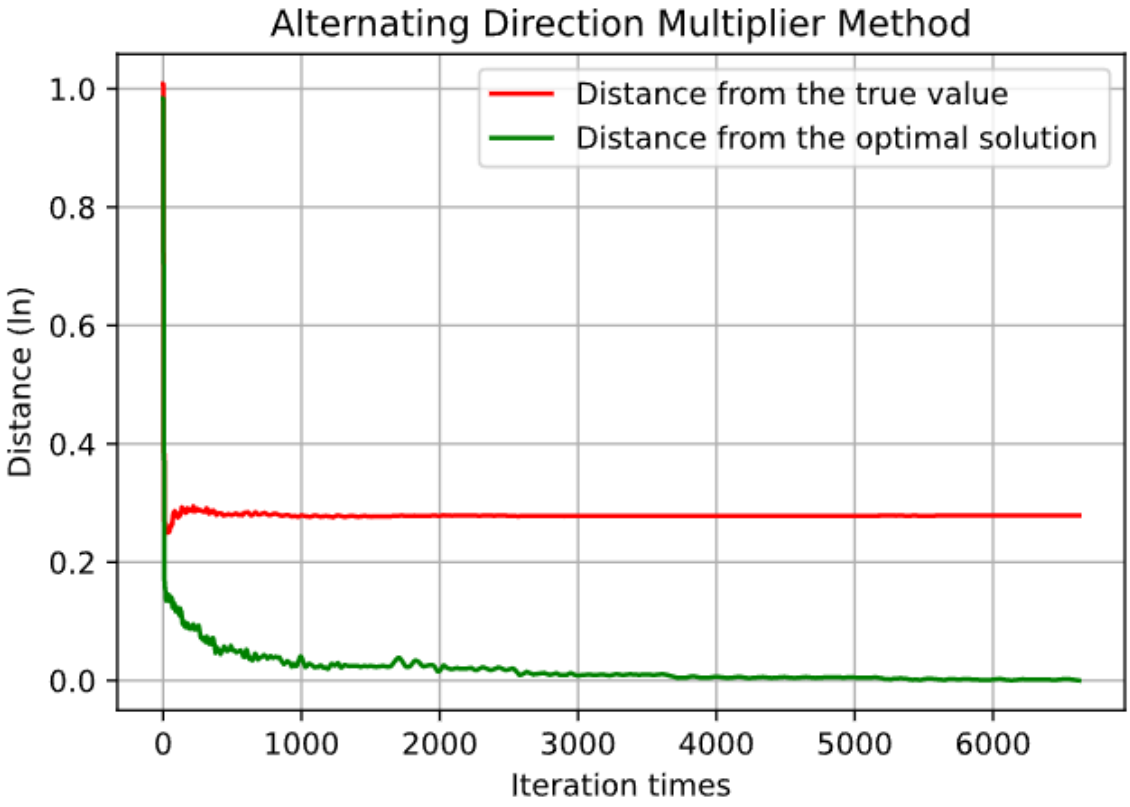
其他内容基本于邻近点梯度发一致，不再赘述。

关键步骤对应代码如下：

```
while k < max_iteration_times:
    '''Step1: master节点发送 x ^ k 给其他节点
                在此不进行模拟'''
    '''Step2:所有worker节点计算各自值，并传送给master节点
                这里在模拟时计算后直接加入temp_numerator和temp_denominator中'''
    temp_numerator = np.zeros(x_dim)
    temp_denominator = np.zeros((x_dim, x_dim))
    for i in range(node_num):
        temp_numerator += A[i].T @ b[i]
        temp_denominator += A[i].T @ A[i]

    '''Step3:master节点根据公式，计算出新的 x y v
                模拟时以下工作都由master节点实现'''
    # 更新x：令导数为0推导公式，从而进行求解
    x_k = np.linalg.inv(temp_denominator + c * np.eye(x_dim, x_dim)) @
(temp_numerator + c * y_k - v_k)
    # 更新y：通过软门限求解
    for i in range(x_dim):
        if x_k[i] + v_k[i] / c < -p / c:
            y_k[i] = x_k[i] + v_k[i] / c + p / c
        elif x_k[i] + v_k[i] / c > p / c:
            y_k[i] = x_k[i] + v_k[i] / c - p / c
        else:
            y_k[i] = 0
    # 更新v：直接代入公式求解
    v_k = v_k + c * (x_k - y_k)

    iteration_result.append(x_k.copy())  # 记录每步计算结果

    '''Step4:重复（1）~（3）直到算法收敛或到达最大迭代次数'''
    if np.linalg.norm(x_k - x_k_pre) < epsilon:
        break
    else:
        x_k_pre = x_k.copy()  # 深拷贝
        k += 1
```

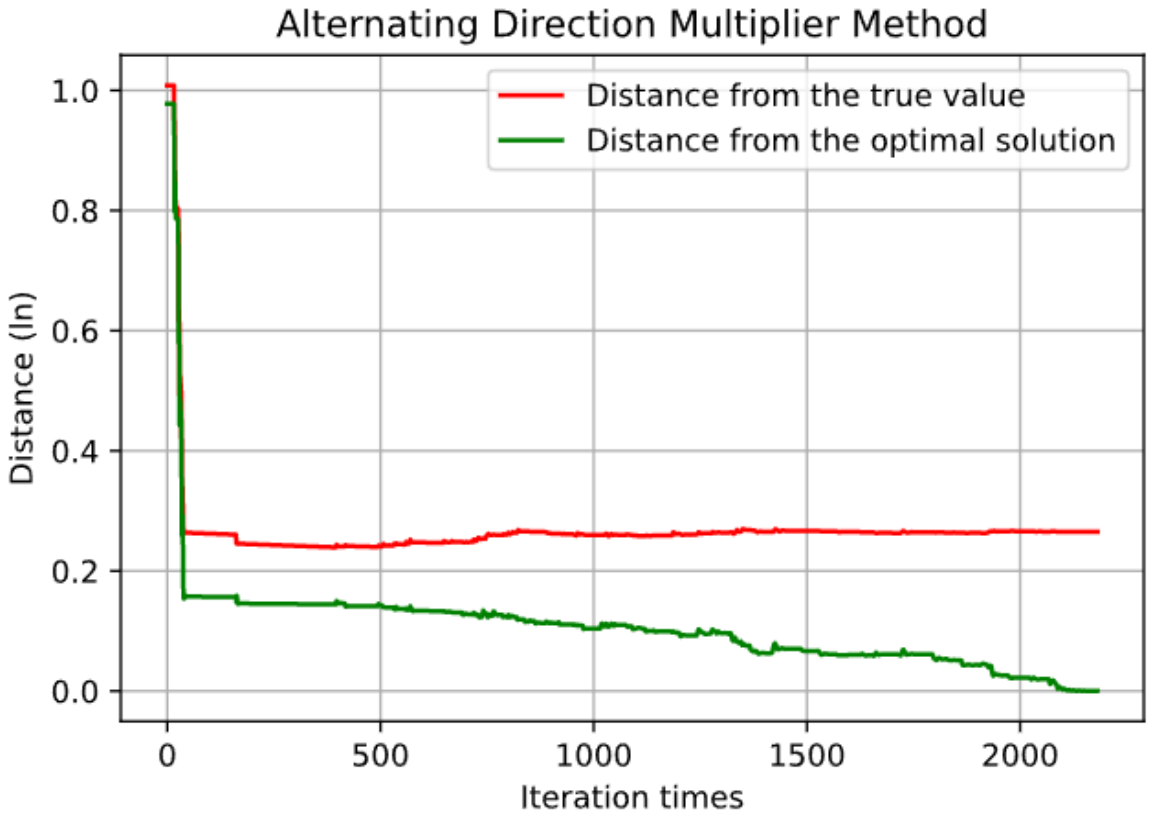
2 数值实验

2.1 p=0.01



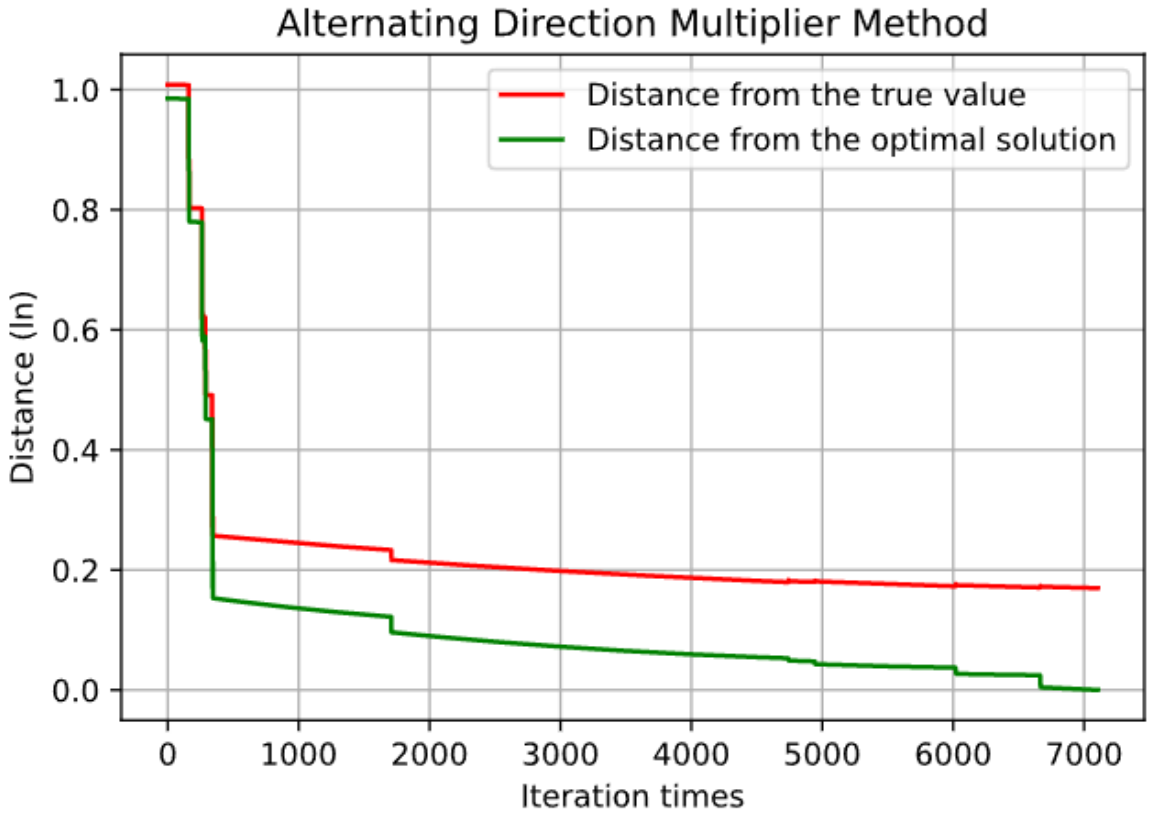
最优解的稀疏度：300

2.2 p=0.1



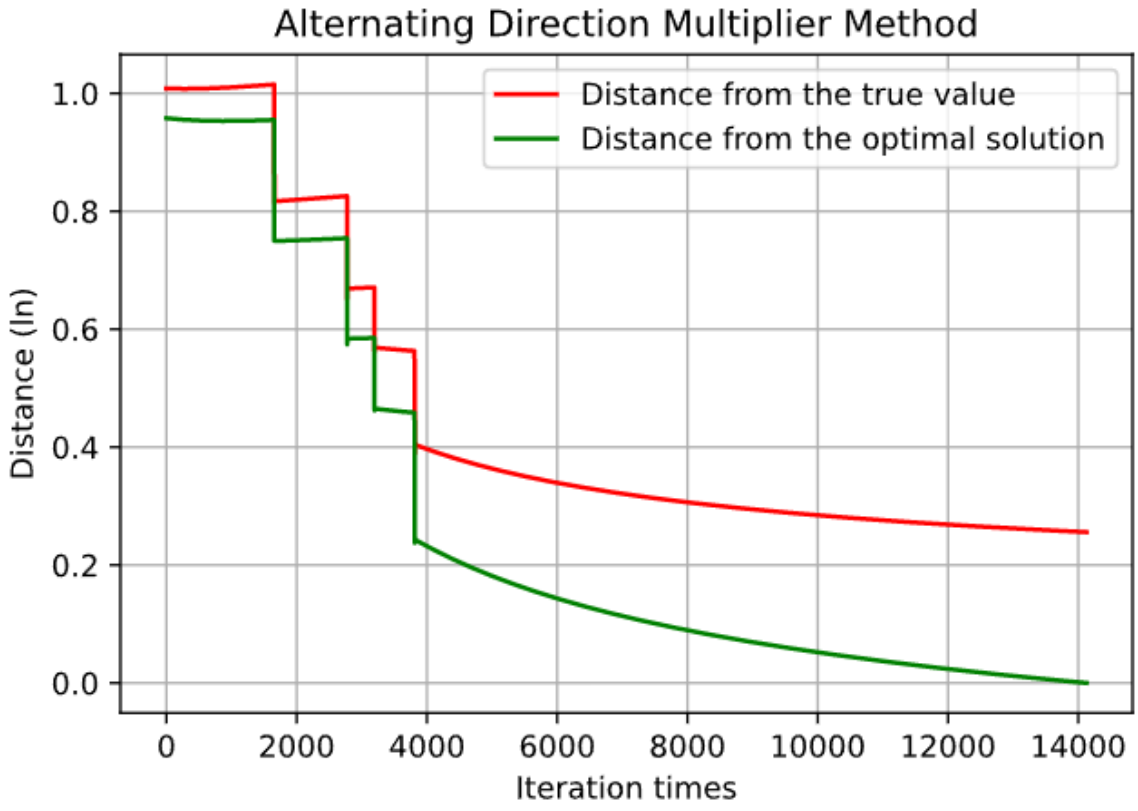
最优解的稀疏度：300

2.3 p=1



最优解的稀疏度：300

2.4 p=10



最优解的稀疏度：300

3 结果分析

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如图像所示。

由图像可知，在一定限度内，p 越大，迭代次数越少，收敛速度越快，超过该限度后，迭代次数增大，收敛速度变慢。

次梯度法

1 算法设计

优化问题：$min\ f_0(x) = \frac{1}{2}\|A_1x - b_1\|_2^2 + \ldots + \frac{1}{2}\|A_{20}x - b_{20}\|_2^2 + p\|x\|_1$

令 $g_0(x) \in \partial f_0(x)$, 则迭代公式为 $x^{k+1} = x^k - \alpha \times g_0(x^k)$

由题目可知，$g_0(x) = \sum_{i=1}^{20} A_i^T(A_ix + b_i) + p\frac{\partial \| x \|_1}{\partial x}$

在$f_0(x)$中，$\| x \|_1$不可微，其次梯度取值如下：

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

另外，在本题中，步长a设置为固定步长0.001。

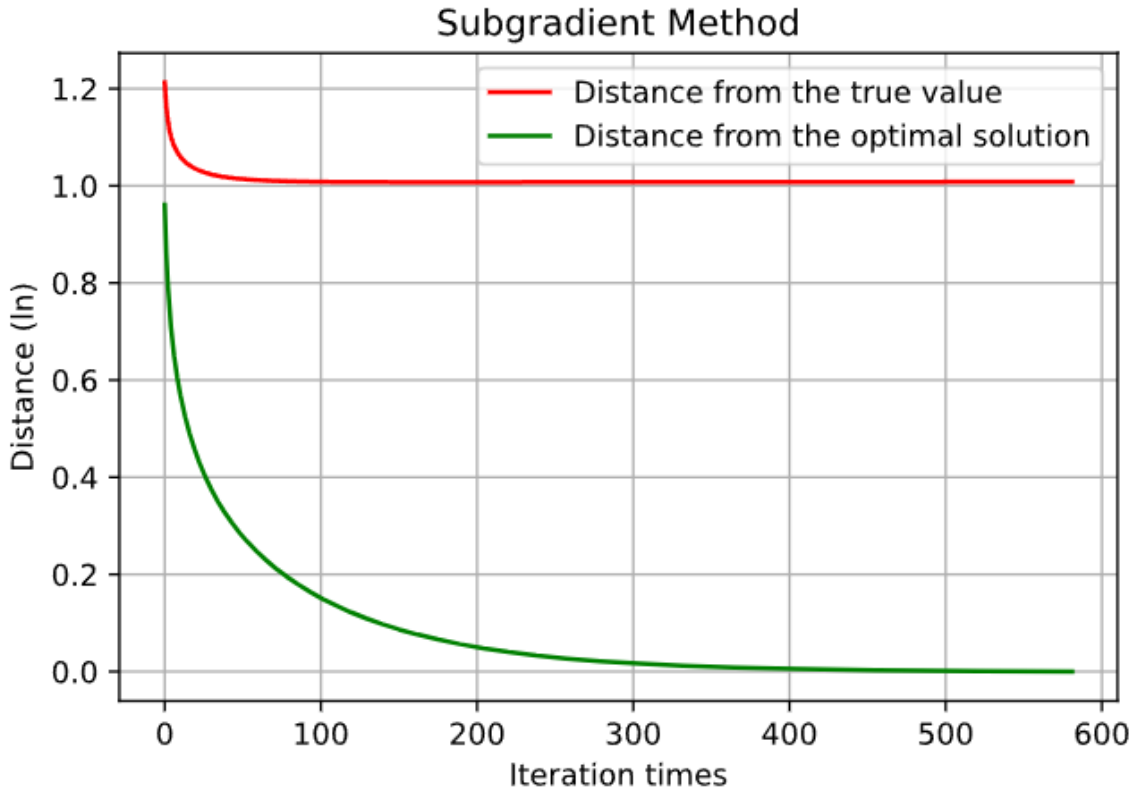
在20个节点中，不妨设节点1为master，剩余节点为worker。工作流程如下：
（1）节点1发送 x^k 给其他节点
（2）所有节点计算各自的$A_i^T(A_ix + b_i)$，并发送给节点1
（3）节点1根据迭代公式，计算出新值
（4）重复（1）~（3）直到算法收敛或到达最大迭代次数

其他内容基本于邻近点梯度发一致，不再赘述。

关键步骤对应代码如下：

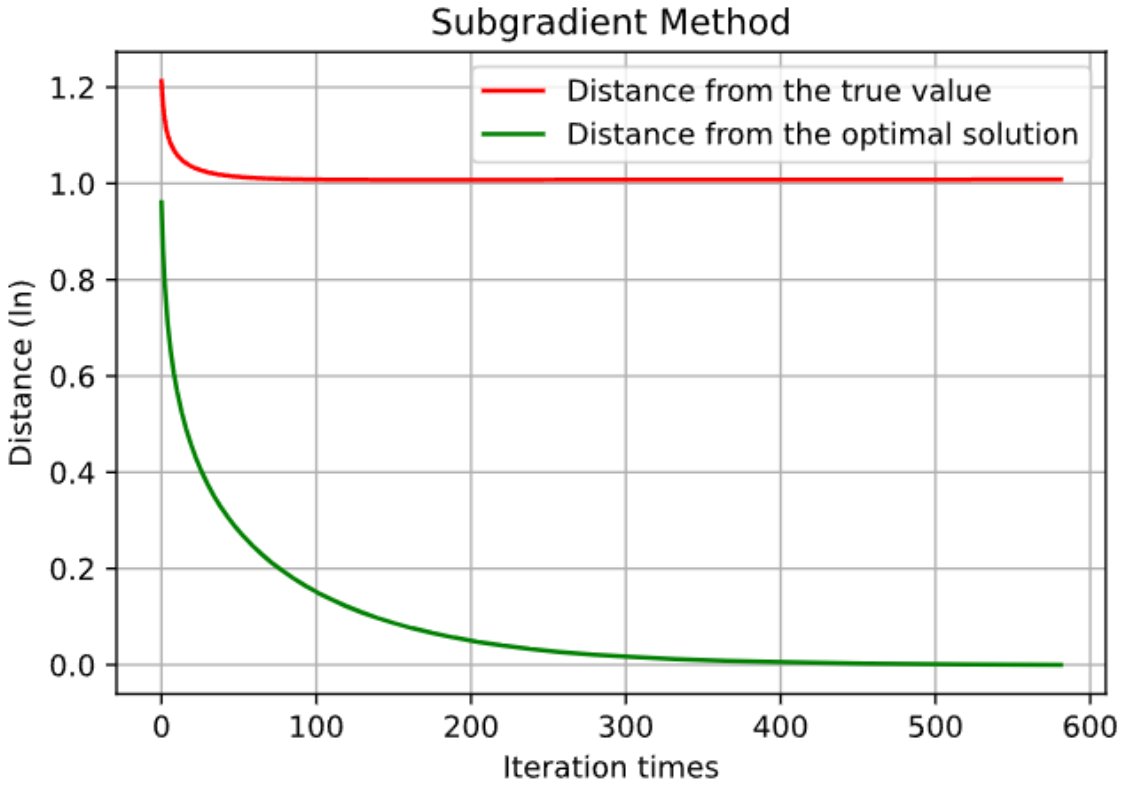
2 数值实验

2.1 p=0.01



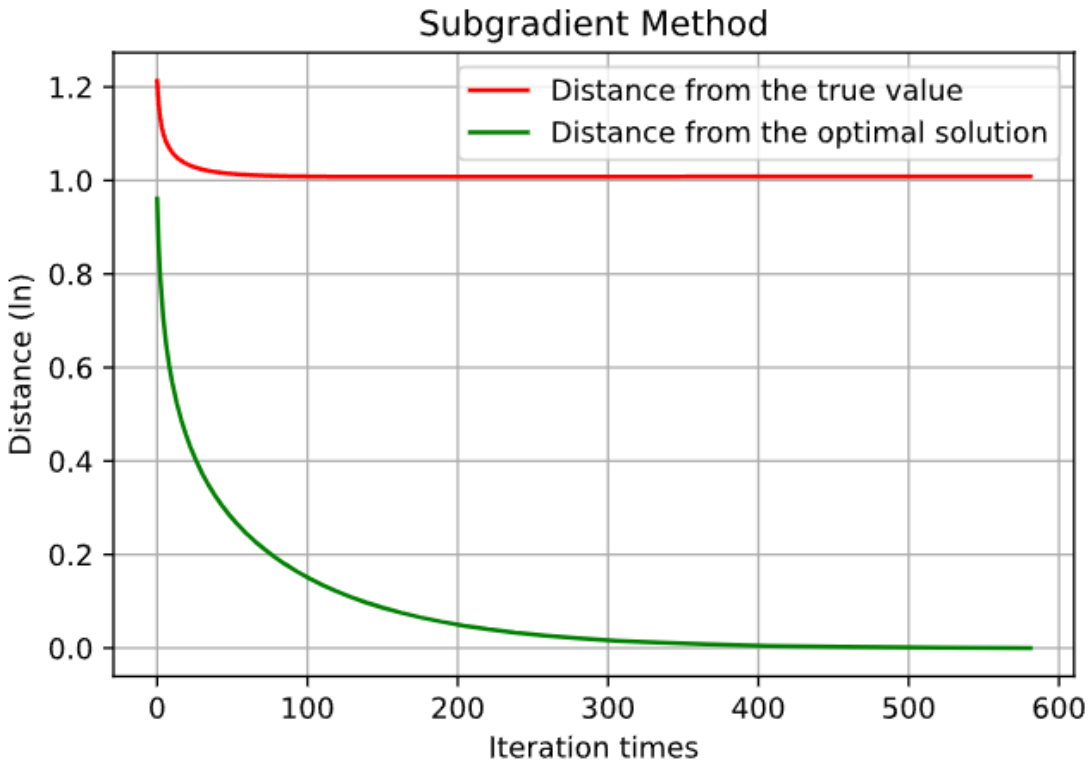
最优解的稀疏度：300

2.2 p=0.1



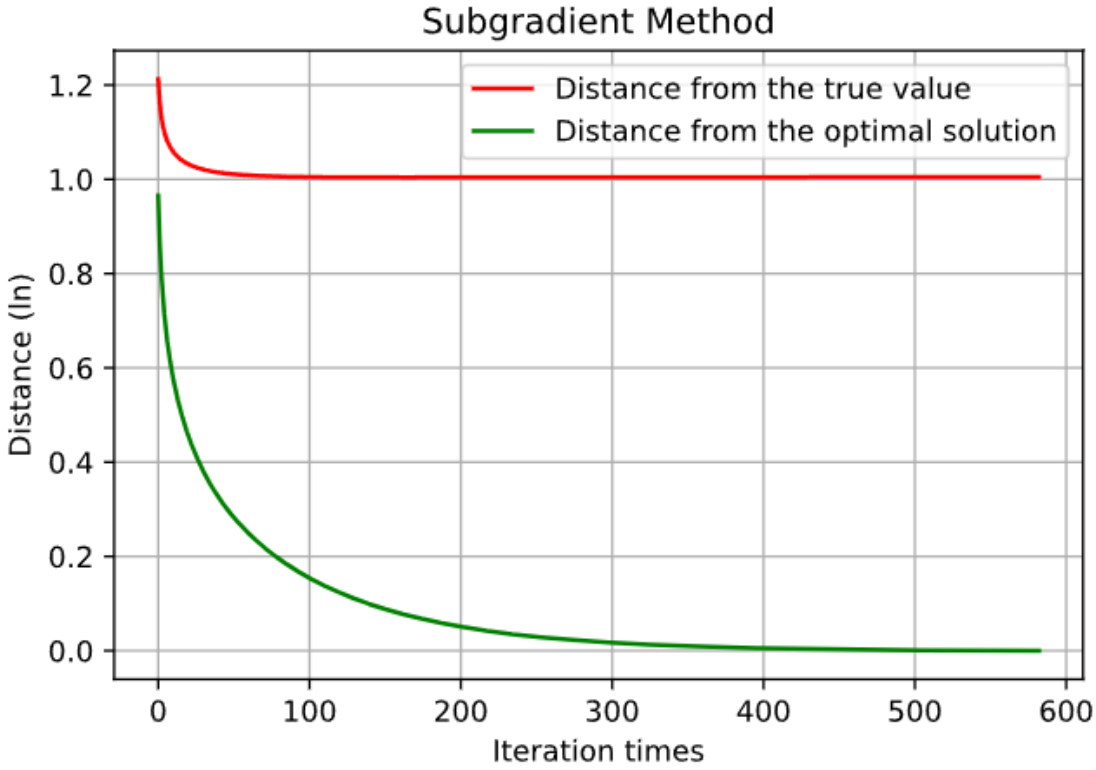
最优解的稀疏度：300

2.3 p=1



最优解的稀疏度：300

2.4 p=10



最优解的稀疏度：300

3 结果分析

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如图像所示。

由图像可知，p的变化对于计算结果没有明显的影响。

问题2

问题描述

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。

首先调用pytorch下载数据集并划分好batch，关键代码如下，重要指令均含有注释，因此不再赘述。

```
"""下载训练集与测试集数据"""
train_loader = torch.utils.data.DataLoader(
        datasets.MNIST('data', # 数据集下载到本地后的根目录，包括 training.pt 和 test.pt 文件
                       train=True, # 设置为True，从training.pt创建数据集，否则会从test.pt创建
                       download=True, # 设置为True，从互联网下载数据并放到root文件夹下
                       transform=transforms.Compose([
                           transforms.ToTensor(),
                           transforms.Normalize((0.1307,), (0.3081,)) # 一种函数或变换，输入PIL图片，返回变换之后的数据
                       ])),
        batch_size=BATCH_SIZE, shuffle=True)# 设置每个批次的大小，并设置进行随机打乱顺序的操作
test_loader = torch.utils.data.DataLoader(
        datasets.MNIST('data', # 数据集下载到本地后的根目录，包括 training.pt 和 test.pt 文件
                       train=False, # 设置为False，从test.pt创建数据集
                       transform=transforms.Compose([
                           transforms.ToTensor(),
                           transforms.Normalize((0.1307,), (0.3081,)) # 一种函数或变换，输入PIL图片，返回变换之后的数据
                       ])),
        batch_size=BATCH_SIZE, shuffle=True)# 设置每个批次的大小，并设置进行随机打乱顺序的操作
```

算法设计

采用逻辑回归模型对mnist数据集进行分类，非题目重点的其他设计源代码均有相应的注释，不再进行赘述。题目考察的算法内容设计如下：

1. 梯度下降法

迭代公式：

$$x^{k+1} = x^k - \alpha^k \frac{1}{N} \sum_{i=1}^{N} \nabla f_i(x^k)$$

由公式可知，在更新参数的过程中，需要计算每一个输入数据对应的梯度，再求和算平均值。对应的关键代码如下：

```
        for x_batch,y_batch in train_dataset:
            batch_index += 1
            print(f"Training Batch: {batch_index}/{len(train_dataset)}")
            # 梯度清零
            W_gradients = np.zeros(W_dim)      # (10,784)
            b_gradients = np.zeros(b_dim)      # (10,1)

            batch_size = len(x_batch)
            for j in range(batch_size): # 每个batch中的每个样本都计算梯度值
                if j % 50 == 0:
                    print(f'index:{j}/{batch_size}...')
                W_g, b_g = L_Gra(W, b, x_batch[j].reshape(x_dim),
y_batch[j])   # 计算梯度值
                W_gradients += W_g
                b_gradients += b_g
            W_gradients /= batch_size   # 求W梯度的平均值
            b_gradients /= batch_size   # 求b梯度的平均值
            # 根据梯度值采用梯度下降法更新参数
            W -= lr * W_gradients
            b -= lr * b_gradients
```

2. 随机梯度法

迭代公式：

$$x^{k+1} = x^k - \alpha^k \nabla f_{i_k}(x^k)$$

由公式可知，每次更新参数时，随机选择batch中某一个输入数据对应的梯度作为当前批次的梯度，从而对参数进行更新。对应的关键代码如下：

```
        for x_batch,y_batch in train_dataset:
            batch_index += 1
            print(f"Training Batch: {batch_index}/{len(train_dataset)}")
            # init grads
            W_gradients = np.zeros(W_dim)      # (10,784)
            b_gradients = np.zeros(b_dim)      # (10,1)

            # j为随机选择的样本
            batch_size = len(x_batch)
            j = np.random.randint(batch_size)  # 随机选择batch中某一个输入数据对
应的梯度作为当前批次的梯度，从而对参数进行更新
            W_g, b_g = L_Gra(W, b, x_batch[j].reshape(x_dim), y_batch[j])
# 计算该样本的梯度值
            W_gradients += W_g
            b_gradients += b_g

```

```
            # 采用随机梯度法更新参数:
            W -= lr * W_gradients
            b -= lr * b_gradients
```

数值实验

1. 梯度下降法

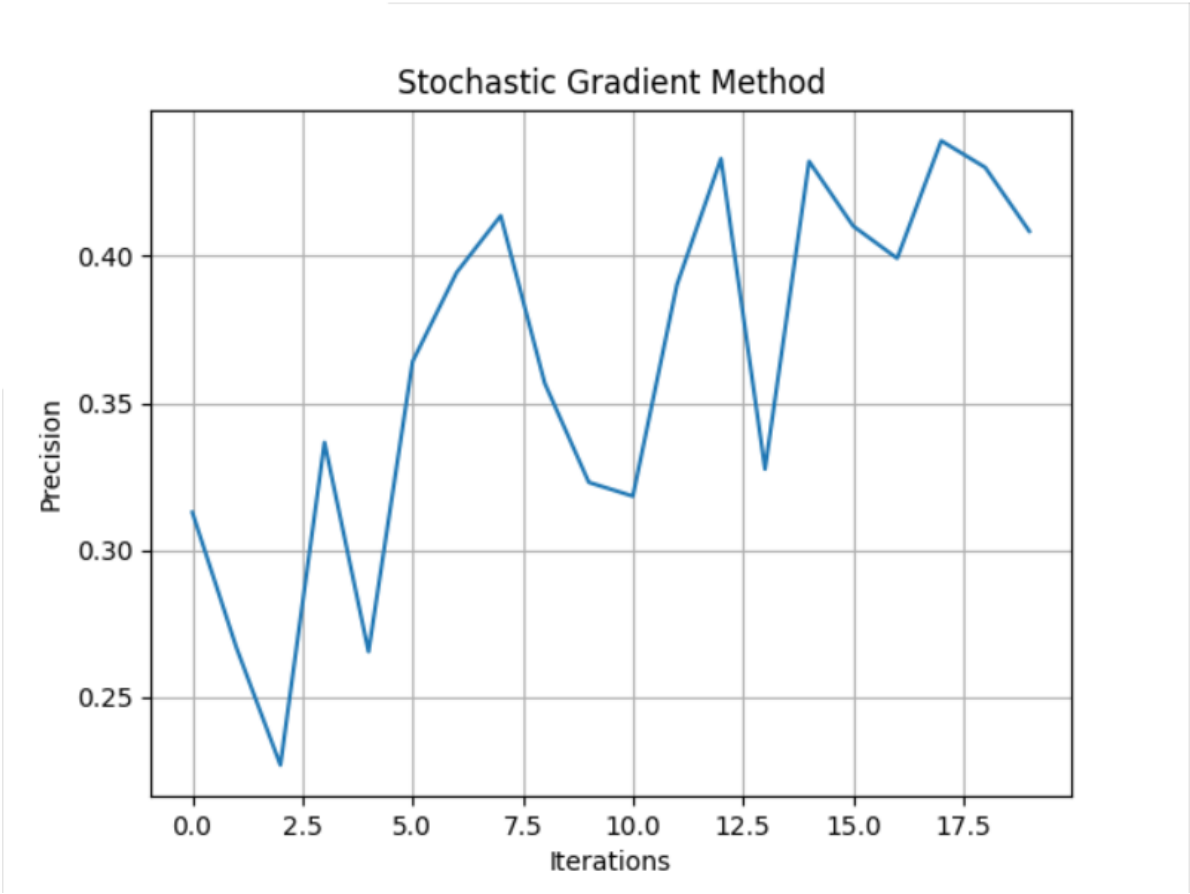
由于梯度下降法的训练时间过长，因此只训练了一个epoch，训练的正确率大概是75%左右。测试集各个batch的预测正确的数据个数，总数据个数以及算得的正确率如下所示(1+3k行是数据个数，2+3k行是总数据个数，3k行是算得的正确率)

```
293
400
0.7325
283
400
0.7075
286
400
0.715
301
400
0.7525
300
400
0.75
300
400
0.75
299
400
0.7475
299
400
0.7475
299
400
0.7475
306
400
0.765
286
400
0.715
292
400
0.73
306
400
0.765
302
400
0.755
299
400
```

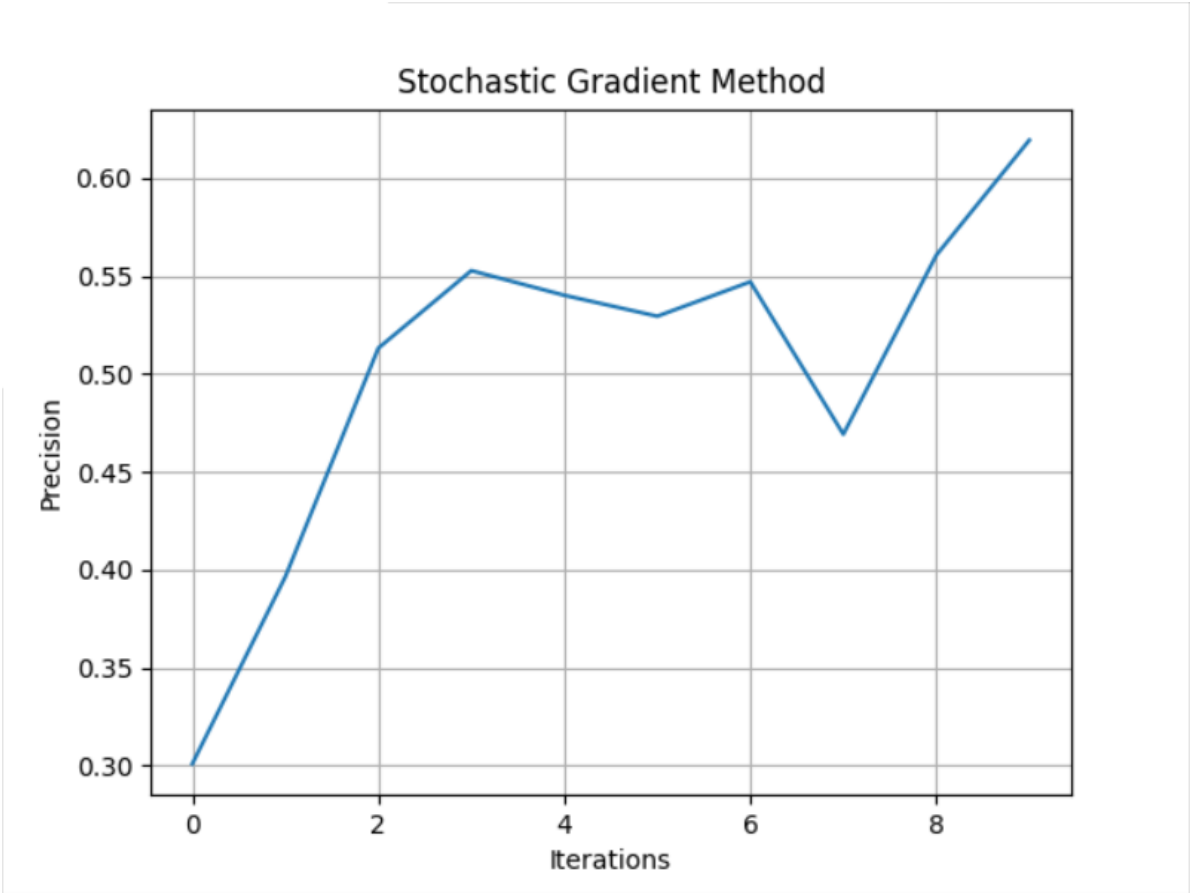
```
0.7475
277
400
0.6925
297
400
0.7425
313
400
0.7825
301
400
0.7525
297
400
0.7425
303
400
0.7575
303
400
0.7575
293
400
0.7325
271
400
0.6775
297
400
0.7425
```

2. 随机梯度法

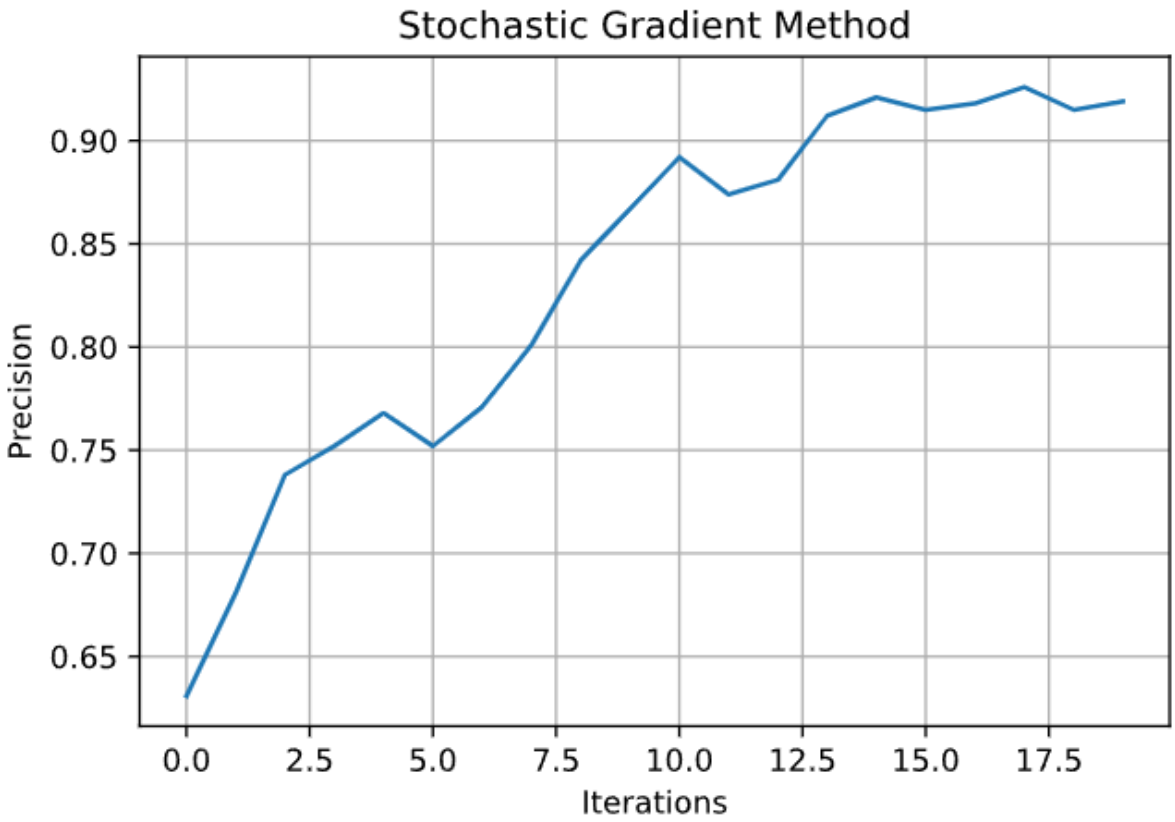
2.1 batch size: 10000，epoch： 20



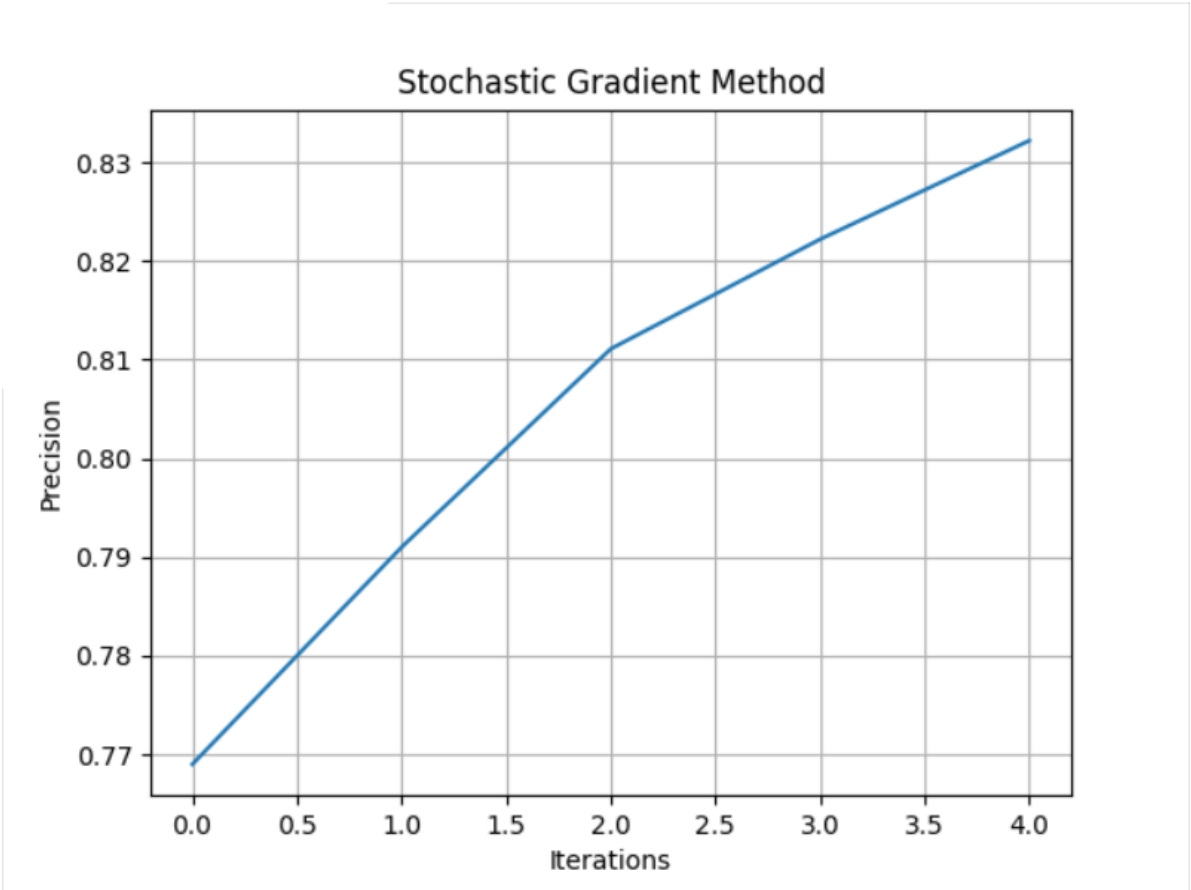
2.2 batch size: 1000，epoch: 10



2.3 batch size: 100, epoch: 20



2.4 batch size: 10, epoch: 5



2.5 结论

在batch size: 100, epoch: 20的情况下，正确率可以到达91.58%。

结果分析

1. 梯度下降法：
   梯度下降法算得得梯度更为精确，但耗费的时间极长，不太具有可行性，同时对最终的结果提升程度也较为有限。也因为耗时过长，所以最终只训练了一个epoch，正确率大概是75%。
2. 随机梯度法：
   相比梯度下降法，随机梯度法的耗时更短，虽然算得的梯度不是十分精确，但依旧有助于模型的训练和收敛。在batch size: 100, epoch: 20的情况下，正确率可以到达91.58%。
   对于随机梯度法，mini-batch 大小对于结果有一定的影响：如数值实验的图像所示，mini-batch越大，每个epoch训练所需的时间越少，但是训练时模型预测正确率的波动程度越大，同时训练相同epoch的情况下，正确率也越低。