L3 homework

yfpeng

September 2022

1 数据集生成

本次作业中，我学会了使用 np.random.multivariate normal 函数生成高斯分布的数据集，而不是像上周那样一维一维地采样。

```python
#生成训练数据的函数
import numpy as np
def get_data(num,expect,variation,seed):
    m=np.array(expect)
    s=np.array(variation)
    np.random.seed(seed)
    raw=np.random.multivariate_normal(m,s,num)
    return raw
def get_dataset(num,expect1,variation1,expect2,variation2,seed1,seed2,seed3)
    raw=get_data(num,expect1,variation1,seed1)
    ones=np.ones((num,1))
    aug_data=np.concatenate([ones,raw],axis=1)
    labels=np.ones((num,1))
    final_data=np.concatenate([aug_data,labels],axis=1)
    raw_1=get_data(num,expect2,variation2,seed2)
    neg_labels=-np.ones((num,1))
    aug_data_1=np.concatenate([ones,raw_1],axis=1)
    final_data_1=np.concatenate([aug_data_1,neg_labels],axis=1)
    dataset=np.concatenate([final_data,final_data_1],axis=0)
    #打乱
    np.random.seed(seed3)
    np.random.shuffle(dataset)
    return dataset
source_data=get_dataset(200,[-5,0],[[1,0],[0,1]],[0,5],[[1,0],[0,1]],4,8,12)
print('show a few data:',source_data[0:4])
print('shape of dataset:',source_data.shape)
```

图 1: 数据集生成的算法

2 算法实现

下面是广义逆算法

```
#算法1，广义逆算法
def generilized_inverse(training_set):
    X=np.matrix(training_set[:,0:3])
    Y=np.matrix(training_set[:,3]).T
    w=(X.T*X).I*X.T*Y
    return w
```

图 2: 广义逆算法

下面是梯度下降法的实现，由于最原始的梯度下降法是对整个数据集，因此，以下的算法是没有 mini-batch 的版本：

```
#算法2，梯度下降
def gradient_descent(train_set,learning_rate,rand_init=True,error=1e-5,max_iter=20):
    X=np.matrix(train_set[:,0:3])
    Y=np.matrix(train_set[:,3]).T
    N=X.shape[0]
    loss_epoch=[]
    if rand_init==True:
        w=np.random.normal(0,1,3)
    else:
        w=np.array([0,0,0])
    w=np.matrix(w).T
    for i in range(max_iter):
        loss=(np.linalg.norm(X*w-Y)**2)/N
        loss_epoch.append(loss)
        gradient=(X.T*X*w-X.T*Y)*2/N
        if np.linalg.norm(gradient)<=error:
            break
        w=w-learning_rate*gradient
    return w,loss_epoch
```

图 3: 梯度下降算法算法

下面是使用了 mini-batch 的版本：

```python
#改变batch_size,原来相当于是以一个训练集为batch，现在改变batch大小
#算法2改进，mini-batch梯度下降
def gradient_descent_mini_batch(train_set,learning_rate,rand_init=True,error=1e-5,max_iter=20,batch_size=10):
    B=batch_size
    loss_epoch=[]
    if rand_init==True:
        w=np.random.normal(0,1,3)
    else:
        w=np.array([0,0,0])
    w=np.matrix(w).T
    for i in range(max_iter):
        loss=0
        np.random.shuffle(train_set)
        for j in range(train_set.shape[0]//B):
            X=np.matrix(train_set[B*j:B*(j+1),0:3])
            Y=np.matrix(train_set[B*j:B*(j+1),3]).T
            loss+=(np.linalg.norm(X*w-Y)**2)/B
            gradient=(X.T*X*w-X.T*Y)*2/B
            if np.linalg.norm(gradient)<=error:
                return w,loss_epoch
            w=w-learning_rate*gradient
        loss_epoch.append(loss)
    return w,loss_epoch
```

图 4: mini-batch 实现版本

3 实验结果

3.1 广义逆

training-set 和 test-set 准确率均为 100%, 以下是可视化:

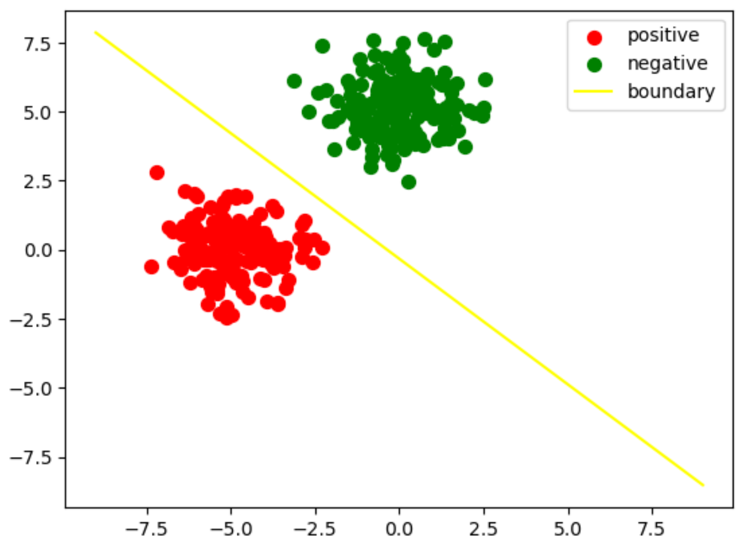

图 5: 训练集可视化

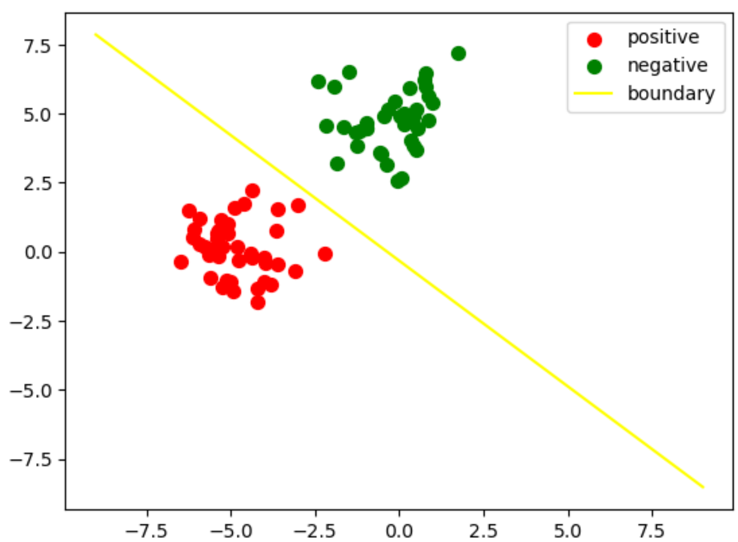

图 6: 测试集可视化

3.2 梯度下降

training-set 和 test-set 准确率均为 100%, 以下是可视化:

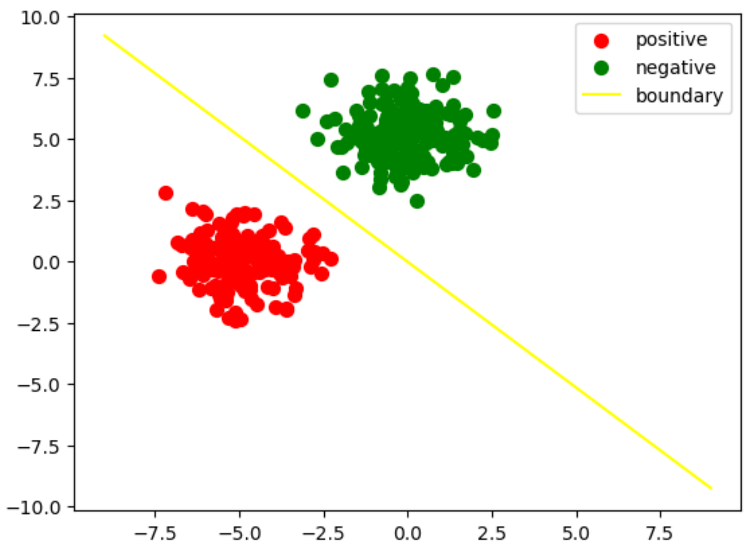

图 7: 训练集可视化

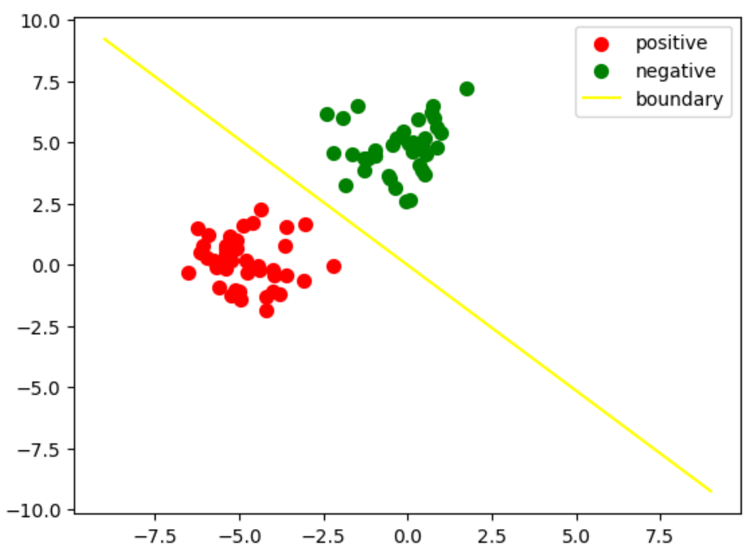

图 8: 测试集可视化

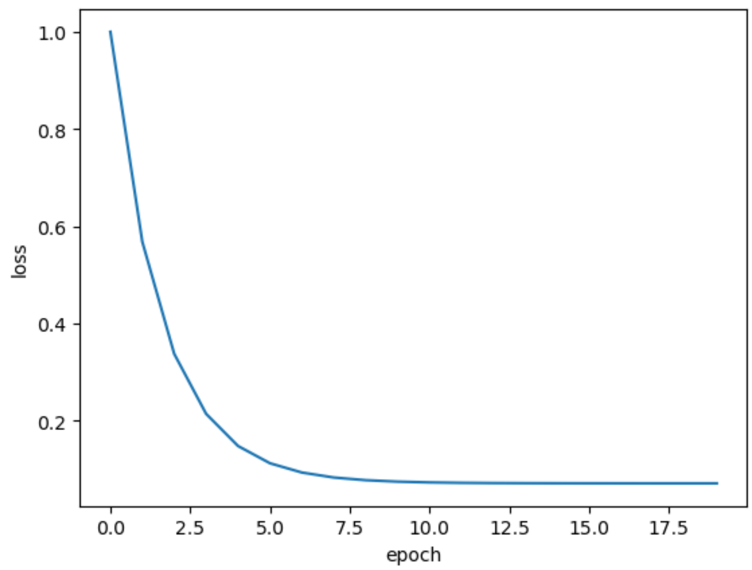

图 9: loss 随 epoch 变化

4 改变样本分布

改变样本分布后，由于数据集已经不能线性可分，准确率大幅度下降，以下展示 loss 函数迭代过程中的变化：

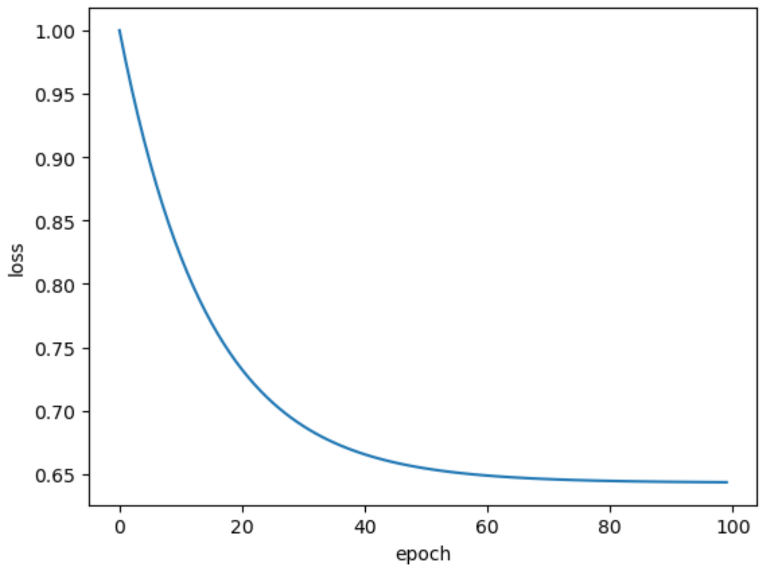

图 10: loss 随 epoch 变化

可见，此时的 loss 函数以无法像之前那样下降为 0

5 改变学习率

我把学习率由原来的 0.01 改为 0.001，得下图：

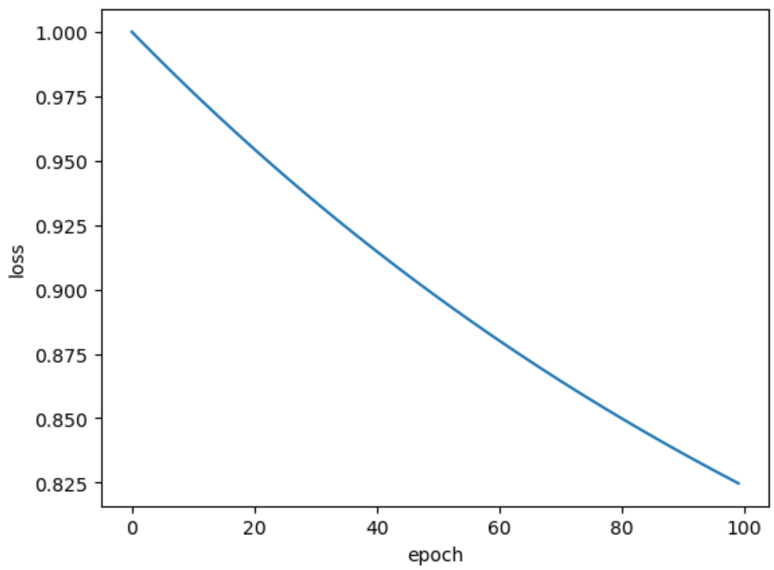

图 11: loss 随 epoch 变化

显然，学习率降低后，收敛速度明显变慢

6 改变 batch-size

以下，是分别采用 10 和 20 作为 batch-size 的结果

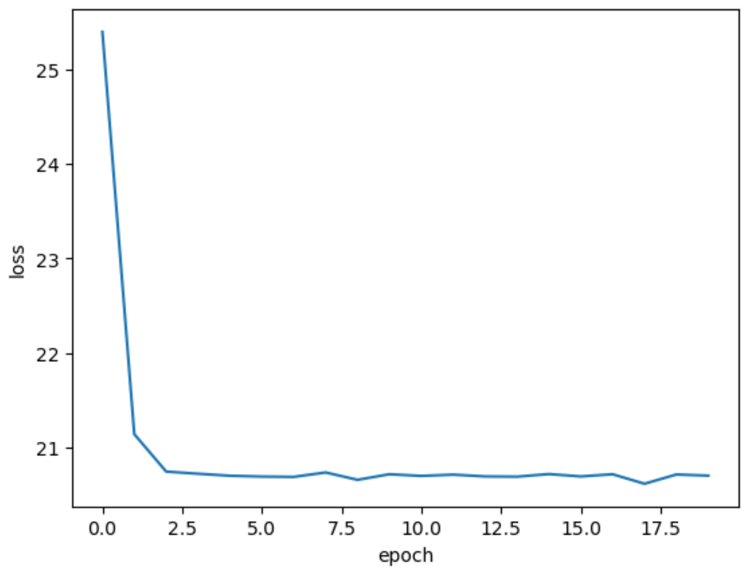

图 12: batch-size=10

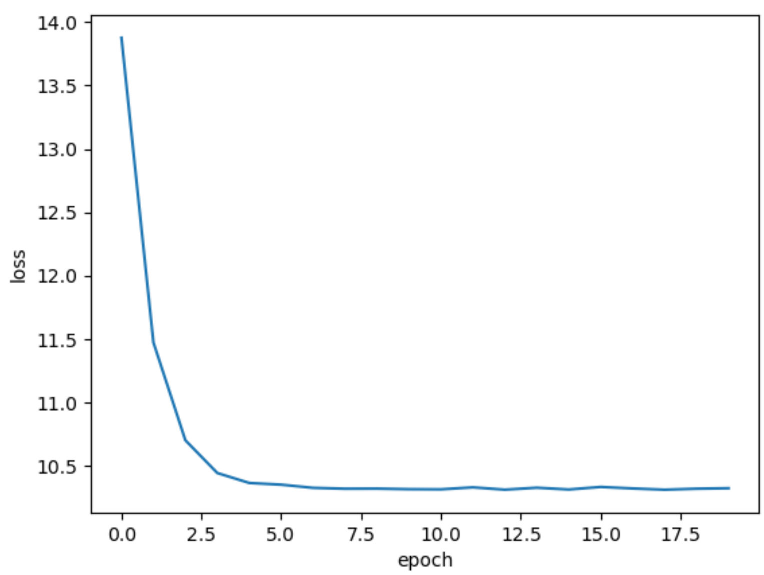

图 13: batch-size=20

可见，batch-size 适当变小有助于提升收敛速度，但是，同时噪音会稍

微大一点，稳定性要稍微差一点

7 epoch 变化

为了讨论 epoch 的影响，我们提升训练难度，将两个分布拉近，得到的结果是：

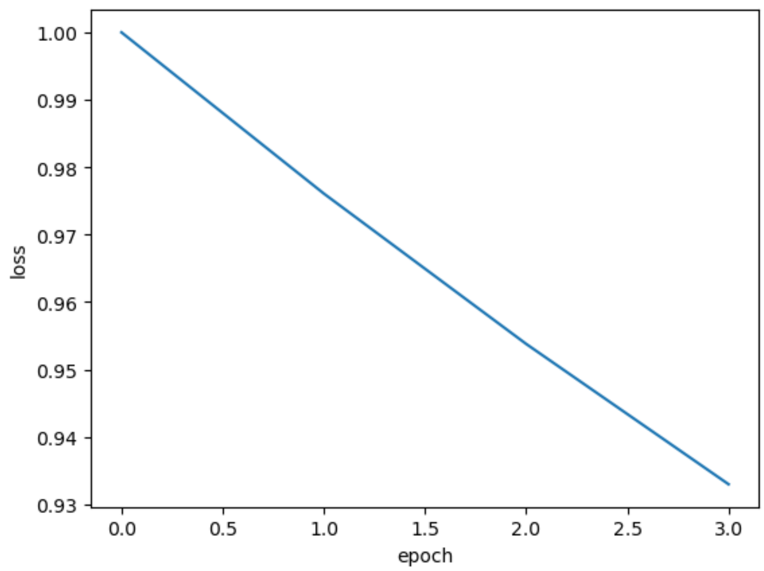

图 14: epoch=20

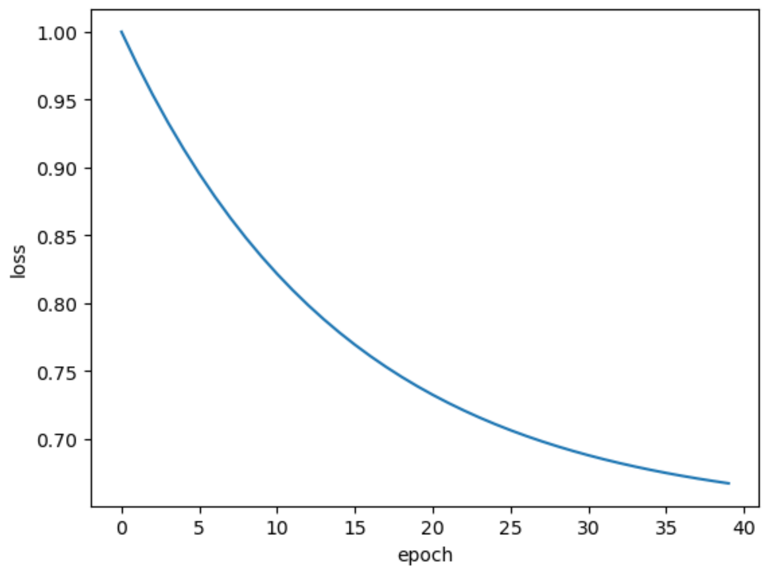

图 15: epoch=40

可见，在收敛之前，epoch 越大，表现越好，但收敛之后，epoch 对表现几乎没有影响

8 单变量函数

$$f(x) = x * cos(0.25\pi * x) \tag{1}$$

求导：

$$\nabla f(x) = cos(0.25\pi * x) - 0.25\pi x * sin(0.25\pi * x) \tag{2}$$

依据以上两个公式，我们进行后面的求解

8.1 梯度下降

```python
#梯度下降求解
def GD(initial=-4,max_iter=10,lr=0.4):
    x=initial
    f=value(x)
    x_all=[x]
    f_all=[f]
    for i in range(max_iter):
        grad=gradient(x)
        x=x-lr*grad
        f=value(x)
        x_all.append(x)
        f_all.append(f)
    epochs=[i for i in range(len(x_all))]
    print(epochs)
    plt.plot(epochs,x_all,color='r',label='x')
    plt.plot(epochs,f_all,color='g',label='f')
    plt.xlabel('epoch')
    plt.ylabel('x and f')
    plt.legend(loc=0)
GD(-4,10,0.4)
```

图 16: 梯度下降算法实现

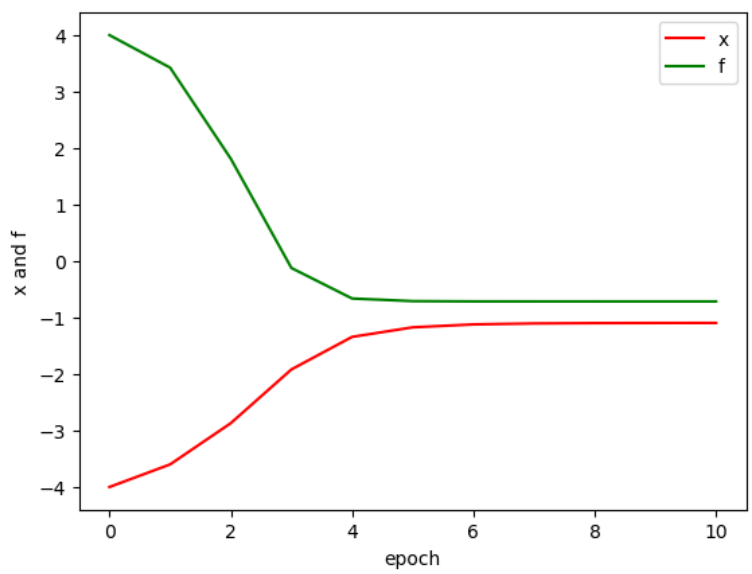

图 17: 梯度下降结果

8.2 随机梯度下降

```python
#随机梯度下降
def SGD(initial=-4,max_iter=10,lr=0.4):
    x=initial
    f=value(x)
    x_all=[x]
    f_all=[f]
    for i in range(max_iter):
        grad=gradient(x,True)
        x=x-lr*grad
        f=value(x)
        x_all.append(x)
        f_all.append(f)
    epochs=[i for i in range(len(x_all))]
    print(epochs)
    plt.plot(epochs,x_all,color='r',label='x')
    plt.plot(epochs,f_all,color='g',label='f')
    plt.xlabel('epoch')
    plt.ylabel('x and f')
    plt.legend(loc=0)
SGD(-4,10,0.4)
```

图 18: 随机梯度下降算法实现

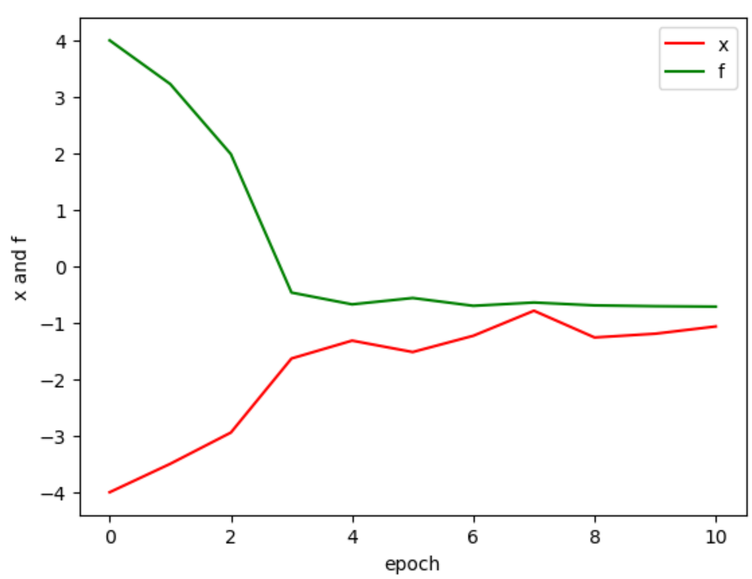

图 19: 随机梯度下降结果

通过比较我们发现，SGD 相对于 GD，收敛速度可能会加快（有一定随机性），但同时，会引入额外的噪音，函数值的抖动更大，梯度没有那么稳定

8.3 Adagrad

```
def Adagrad(initial=-4,max_iter=10,lr=0.4,epsilon=1e-6):
    x=initial
    f=value(x)
    x_all=[x]
    f_all=[f]
    grad_all=[]
    for i in range(max_iter):
        grad=gradient(x)
        grad_all.append(grad)
        sigma=float(np.linalg.norm(np.array(grad_all))/np.sqrt(len(grad_all)))+epsilon
        x=x-lr*grad/sigma
        f=value(x)
        x_all.append(x)
        f_all.append(f)
    epochs=[i for i in range(len(x_all))]
    print(epochs)
    plt.plot(epochs,x_all,color='r',label='x')
    plt.plot(epochs,f_all,color='g',label='f')
    plt.xlabel('epoch')
    plt.ylabel('x and f')
    plt.legend(loc=0)
Adagrad(-4,10,0.4,1e-6)
```

图 20: Adagrad 算法实现

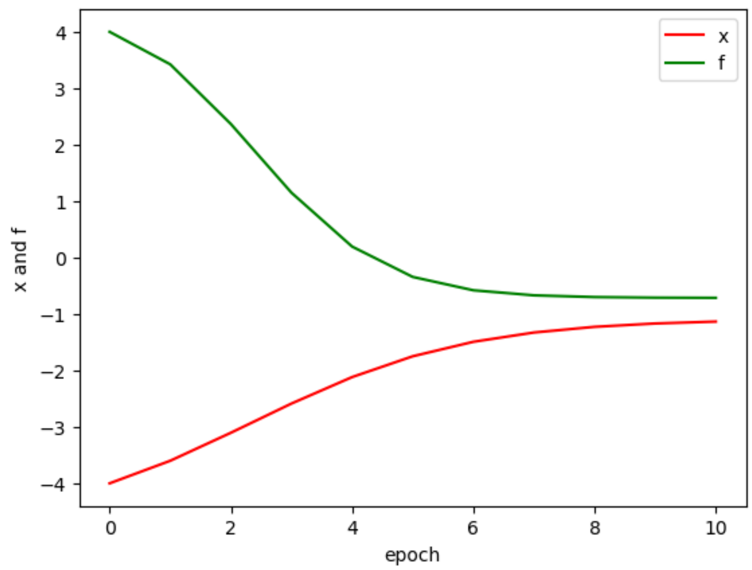

图 21: Adagrad 结果

从上图看出，Adagrad 的函数值变化更平滑，但同时，它的收敛速度也更慢

8.4 RMSProp

```python
def RMS_prop(initial=-4,max_iter=10,lr=0.4,alpha=0.9):
    x=initial
    f=value(x)
    x_all=[x]
    f_all=[f]
    for i in  (function) gradient: (x: Any, SGD: bool = False) -> Any
        grad=gradient(x)
        if i==0:
            sigma=abs(grad)
        else:
            sigma=math.sqrt(alpha*(sigma**2)+(1-alpha)*(grad**2))
        x=x-lr*grad/sigma
        f=value(x)
        x_all.append(x)
        f_all.append(f)
    epochs=[i for i in range(len(x_all))]
    print(epochs)
    plt.plot(epochs,x_all,color='r',label='x')
    plt.plot(epochs,f_all,color='g',label='f')
    plt.xlabel('epoch')
    plt.ylabel('x and f')
    plt.legend(loc=0)
RMS_prop(-4,10,0.4,0.9)
```

图 22: RMSProp 算法实现

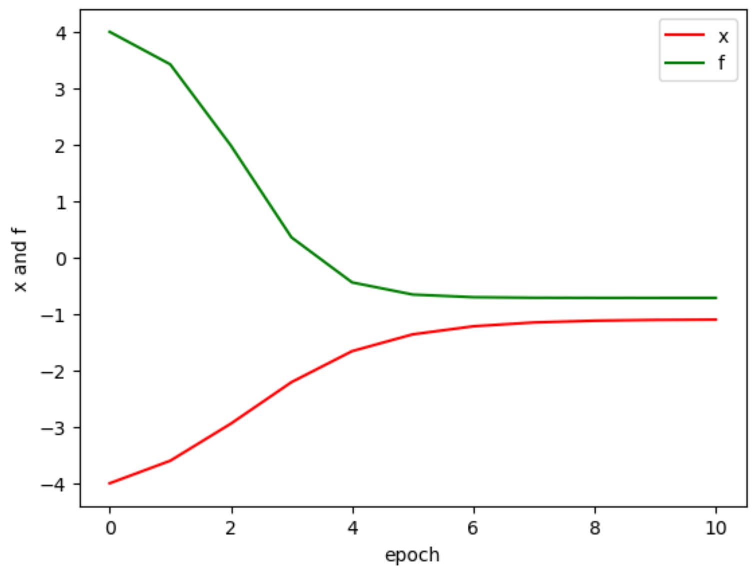

图 23: RMSProp 结果

RMSProp 相对于 Adagrad 效果有一定提升，主要是收敛速度更快，这也许是因为他更多的考虑最近时刻的梯度，实时性更好

8.5 momentum

```python
def momentum(initial=-4,max_iter=10,lr=0.4,lamd=0.9):
    x=initial
    f=value(x)
    x_all=[x]
    f_all=[f]
    m=0.0
    for i in range(max_iter):
        grad=gradient(x)
        m=lamd*m-lr*grad
        x=x+m
        f=value(x)
        x_all.append(x)
        f_all.append(f)
    epochs=[i for i in range(len(x_all))]
    print(epochs)
    plt.plot(epochs,x_all,color='r',label='x')
    plt.plot(epochs,f_all,color='g',label='f')
    plt.xlabel('epoch')
    plt.ylabel('x and f')
    plt.legend(loc=0)
momentum(-4,10,0.4,0.9)
```

图 24: momentum 算法实现

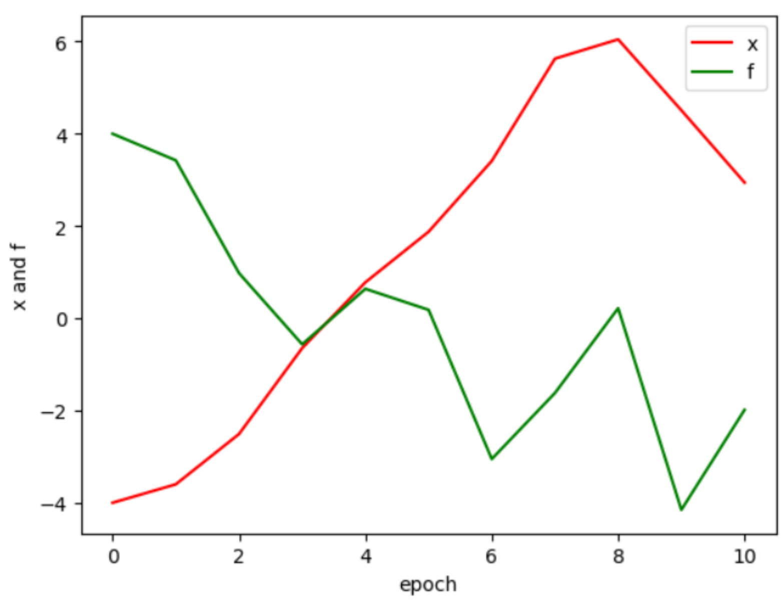

图 25: momentum 结果

momentum 成功的让 point 偏离局部极值点，因此它的函数值相对于前面的算法更有，但是也许是迭代次数不够多，它没有在 10 个 epoch 内收敛。

8.6 Adam

```python
def Adam(initial=-4,max_iter=10,lr=0.4,beta1=0.9,beta2=0.999,epsilon=1e-6):
    x=initial
    f=value(x)
    x_all=[x]
    f_all=[f]
    m=0.0
    v=0.0
    for i in range(max_iter):
        grad=gradient(x)
        m=beta1*m+(1-beta1)*grad
        v=beta2*v+(1-beta2)*(grad**2)
        m_hat=m/(1-beta1**(i+1))
        v_hat=v/(1-beta2**(i+1))
        x=x-lr*m_hat/(math.sqrt(v_hat)+epsilon)
        f=value(x)
        x_all.append(x)
        f_all.append(f)
    epochs=[i for i in range(len(x_all))]
    print(epochs)
    plt.plot(epochs,x_all,color='r',label='x')
    plt.plot(epochs,f_all,color='g',label='f')
    plt.xlabel('epoch')
    plt.ylabel('x and f')
    plt.legend(loc=0)
Adam(-4,10,0.4,0.9,0.999,1e-6)
```

图 26: Adam 算法实现

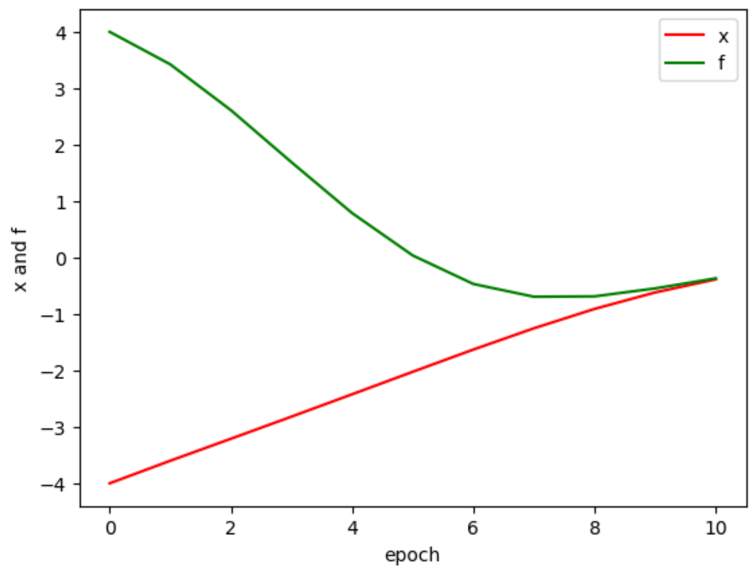

图 27: Adam 结果

奇怪的是，adam 并没能帮助 point 跳出局部极值点，这可能是超参设定不合适造成的

8.7 修正后的 Adam

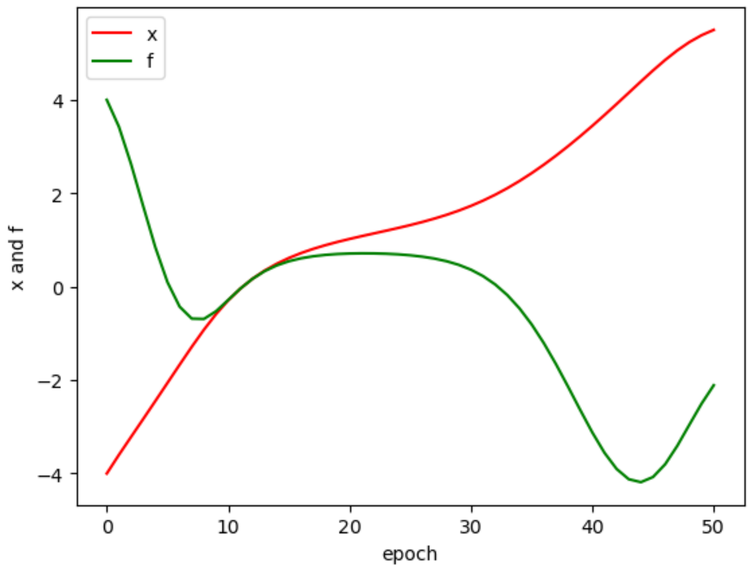

图 28: 修正后 Adam 结果

这时，成功跳出局部极值，可见 $\beta1$ 越大，动量越大，越有可能跳出局部极值点。

8.8 改变 GD、SGD、Adagrad、RMSProp 的参数

鉴于这几个算法都只能改变迭代次数，因此我们一起比较：

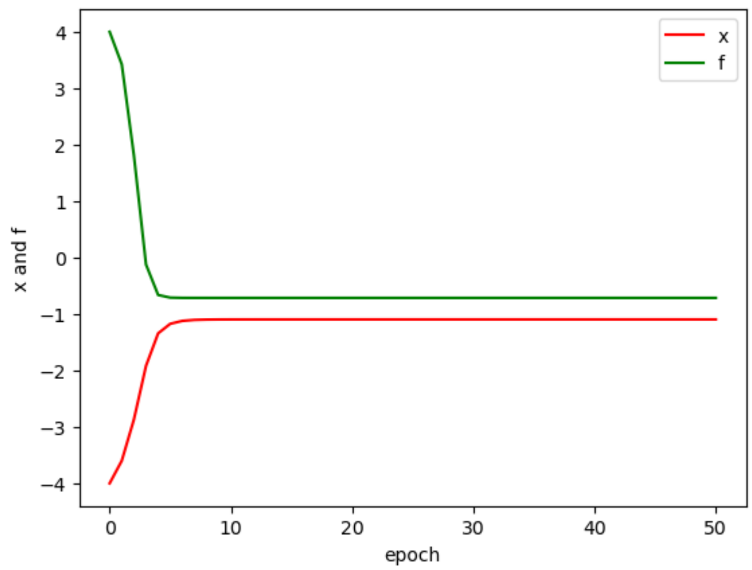

图 29: 改变迭代次数后 GD 结果

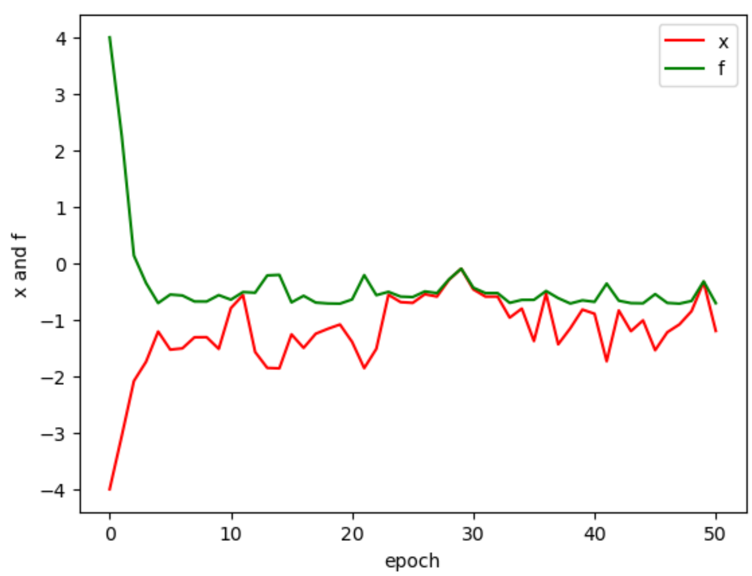

图 30: 改变迭代次数后 SGD 结果

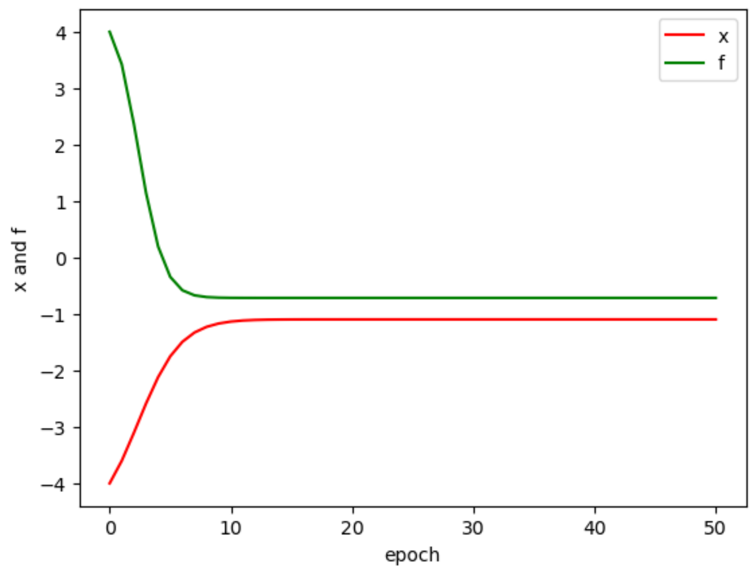

图 31: 改变迭代次数后 Adagrad 结果

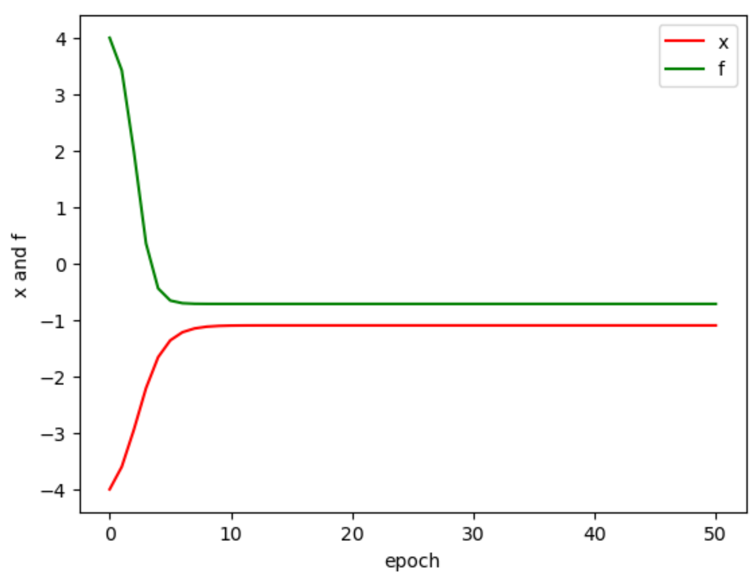

图 32: 改变迭代次数后 RMSProp 结果

可以预料到，这几个算法没有脱离局部极值的能力，因此，都无一例外

的陷入局部极值，无法继续优化

8.9 调整 momentum

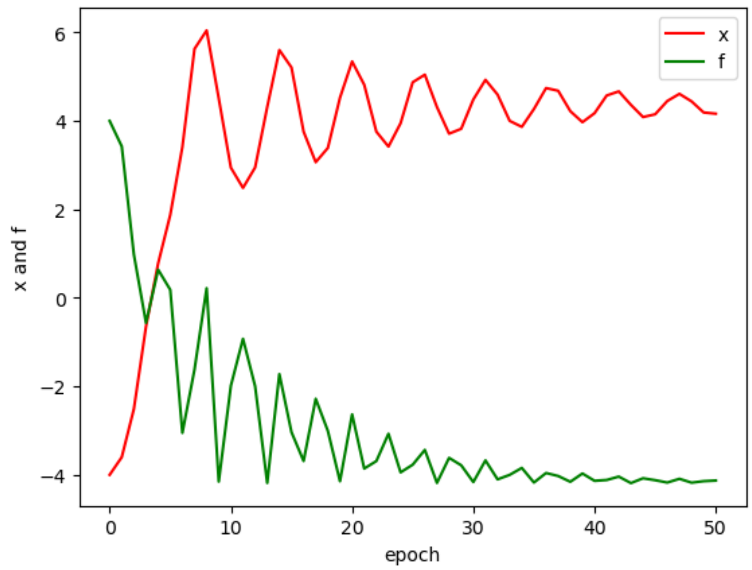

图 33: 改变迭代次数后 momentum 结果

可以看见，我们增加迭代次数后，momentum 在脱离了局部极值后，可以收敛到全局最优。这里就不去改变动量了，因为可以想到动量减小之后它无法脱离局部极值

8.10 调整 Adam

在经过大量调参之后，得到如下收敛的结果：

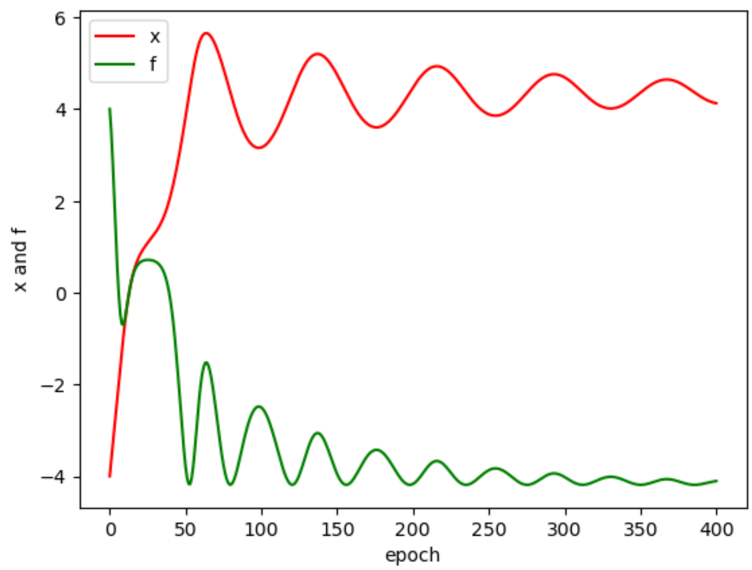

图 34: 修正后 Adam 结果

我发现，适当降低学习率，同时提升动量，增加迭代次数，可以让它收敛于全局最优。说实话，Adam 的调优相对于 momentum 的调优难度要大很多，还是要凭经验。