最优化homework3

姓名	学号
崔璨明	20337025

1、实验要求

按Topic 6 第41页里面的要求，使用Proximal Gradient Descent的算法来实现图像分类任务。补全附件里面的代码。注意一点的是，1）SVM loss是有平方的，2）还有正则项 $lambda||W||^2$

2、实验原理

- Linear Model: $S = f(x;W) = Wx, s_j = w_j x, W \in \mathbb{R}^{C\times D}, S \in \mathbb{R}^C$
- Score Function: $L_i = \sum_{j\neq y_i} max(0, s_j - s_{yi} + 1)^{\mathbf{2}}$
- Regularization: $R(W) = \sum_{d=1}^{D}\sum_{j=1}^{C} W_{j,d}^2$
- Loss function: $L(W) = \frac{1}{N}\sum_{i=1}^{N} L_i + \lambda R(W)$

Loss函数对W进行求导，分为两部分，后面正则项的求导为$\mathbf{2} * \lambda * W$
前面部分的求导基于$(X_i * W_j - X_i * W_{y_i} + 1) > 0$。当这个条件不满足时，对应的$dW$为0。

依次对W的每一列求导。得到下式：

$$\frac{\partial L}{\partial W_j} = \frac{1}{N}\sum_{i=1}^{N}\frac{\partial L_i}{\partial W_j}$$

$$L_i = \sum_{j\neq y_i}^{C}(X_i * W_j - X_i * W_{y_j} + 1)$$

当$j \neq y_i$时：

$$\frac{\partial L_i}{\partial W_j} = X_i$$

当$j = y_i$时：

$$\frac{\partial L_i}{\partial W_j} = -\sum_{j \neq y_i}^{C} X_i$$

该式的求和不是单纯地将C-1个X_i相加。因为对于某个j，很有可能计算得到的$(X_i * W_j - X_i * W_{y_i} + 1) \leq 0$，这种情况下得到的值是0。所以需要统计满足$(X_i * W_j - X_i * W_{y_i} + 1) > 0$对应的j的个数，用这个系数乘$X_i$.

3、补全代码

将代码补全，实现使用Proximal Gradient Descent的算法来实现图像分类任务，补全部分如下，具体解析见注释：

```
for step in range(epochs):
  pred = train_X.dot(W)
  # 补充完整的代码
  N = train_Y_len
  dw = np.zeros(W.shape)
  loss = 0.0

  #计算不同X对不同类别的得分，对应X*W
  score = train_X.dot(W)
  #scores[N,y]取出score中 第i行，第yi列的数据，即每个X对于其正确类别的得分
  score_y = score[range(N),train_Y]
  score_y=score_y.reshape(-1,1)

  #计算差值
  diff = score - score_y + 1
  # j!=yi,if j=yi loss=0
  diff[range(N),train_Y] =0

  #和0比较，取最大值
  tmp=np.zeros(diff.shape)
  loss = np.sum(np.maximum(diff,tmp)*np.maximum(diff,tmp)) /
N+lamda*np.sum(W*W)
  # 求导的一部分
  d_loss=2*np.sum(np.maximum(diff,tmp))/N

  s = diff>0   #if score>0,true
  num_of_score = np.sum(s,axis=1)  # 每个数据得分大于0的个数
  #将M矩阵初始化为全零矩阵，对于diff大于零的部分，在M矩阵的对应位置赋值为1
  M = np.zeros(s.shape)
```

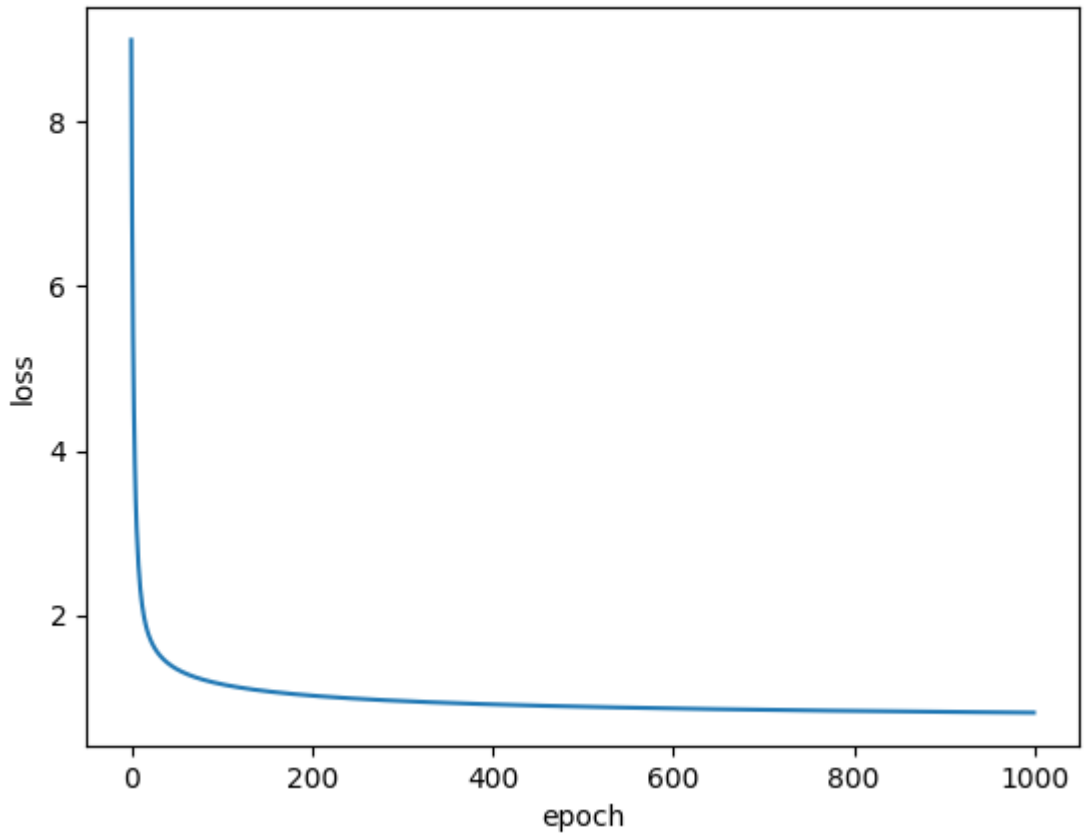
```
    #j=yi，乘上大于0的个数
    M[range(N),train_Y] = -num_of_score
    M = M + s
    #计算对W的导数
    dw = d_loss * train_X.T.dot(M)/N + 2*lamda*W
    #更新W
    W=W-lr*dw
    #------------------------------
    pred = test_X.dot(W)
    pred = np.argmax(pred, 1)
    acc = np.equal(test_Y, pred).sum() / len(test_X)
    print("[%d/%d]LOSS:%.3f, ACC:%.3f" %(step+1, epochs, loss, acc))
```

4、实验结果

在测试集上的测试结果，准确率为88.5%：

```
[998/1000]LOSS:0.810, ACC:0.885
[999/1000]LOSS:0.810, ACC:0.885
[1000/1000]LOSS:0.810, ACC:0.885
```

loss函数的变化曲线：



5、实现Nesterov's accelerated gradient descent

根据老师课上的要求，尝试用Nesterov's accelerated gradient descent来加速PGD，只需修改W更新的代码即可：

```
W = np.zeros((784, num_classes))
W_y = np.zeros((784, num_classes))
W_z = np.zeros((784, num_classes))
for step in range(epochs):
...
        dw = d_loss * train_X.T.dot(M)/N + 2*lamda*W
        W_y=W-lr*dw
        W_z=W_z-(float(epochs+1)*lr/2)*dw
        W=(float(epochs+1)/float(epochs+3))*W_y+(2/float(epochs+3))*W_z
...
```

运行程序，以下是PGD在12次迭代中的结果，可以看到LOSS函数下降到了2.213

以下是使用Nesterov’s accelerated gradient descent加速的PGD在12次迭代中的结果，可以看到LOSS函数比未加速的PGD收敛要快。