Exercise 1： Softmax Regression

朱霞洋 2113301

Exercise 1： Softmax Regression
- 输入输出描述
- 实验原理
  - Softmax
  - 交叉熵损失函数
    - 1. 交叉熵损失的计算方法：
    - 2. 交叉熵损失的意义：
  - 梯度下降
- 程序设计
  - softmax_regression
  - cal_accuracy
- 优化方法
  - 用矩阵乘法和并行计算替代循环
  - 提前转换参数为float型大幅提升训练速度
  - 学习率α衰变
  - data_convert进行调优
- 最终结果

输入输出描述

为了方便后续的原理说明和方法介绍，先对模型的输入输出进行描述。

在Lab1中，训练集的输入train_images为矩阵X：

$$X=\begin{bmatrix} x_{10} & x_{11} & x_{12} & \cdots & x_{1n} \\ x_{20} & x_{21} & x_{22} & \cdots & x_{2n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_{m0} & x_{m1} & x_{m2} & \cdots & x_{mn} \end{bmatrix}$$

其中$m=60000, n=784$，这是初始的x的形状，后续训练时会需要转置；

而训练集的标签train_label在经过data_convert()函数进行独热编码后成为矩阵Y：

$$Y=\begin{bmatrix} y_{11} & y_{12} & \cdots & y_{1m} \\ y_{21} & y_{22} & \cdots & y_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ y_{k1} & y_{k2} & \cdots & y_{km} \end{bmatrix}$$

其中$k=10$,该矩阵的形状为$(10, 60000)$，Y的每一列只有一个值为1，其它全为0。
$y_{ij}=1(1\le i\le m, 1\le j\le k)$表示第j个样本对应的类别为第i类。

我们需要优化的参数矩阵为：

$$\theta=\begin{bmatrix} \omega_{01} & \omega_{02} & \cdots & \omega_{0n} \\ \omega_{11} & \omega_{12} & \cdots & \omega_{1n} \\ \omega_{21} & \omega_{22} & \cdots & \omega_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ \omega_{k1} & \omega_{k2} & \cdots & \omega_{kn} \end{bmatrix}$$

其形状为$(10, 784)$,由$\theta\cdot X.T$可以得到这一层网络的输出，记为矩阵Z，其形状$(10, 60000)$与Y相同：

$$Z = \begin{bmatrix} z_{11} & z_{12} & \cdots & z_{1m} \\ z_{21} & z_{22} & \cdots & z_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ z_{k1} & z_{k2} & \cdots & z_{km} \end{bmatrix}$$

而最终softmax后的结果为：

$$\hat{Y} = softmax(Z) = \begin{bmatrix} \hat{y}_{11} & \hat{y}_{12} & \cdots & \hat{y}_{1m} \\ \hat{y}_{21} & \hat{y}_{22} & \cdots & \hat{y}_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ \hat{y}_{k1} & \hat{y}_{k2} & \cdots & \hat{y}_{km} \end{bmatrix}$$

实验原理

Softmax

在多类别分类问题中，Softmax 函数通常用于神经网络的输出层，将网络的原始输出转化为各个类别的概率分布。在训练过程中，Softmax 函数可以最大化正确类别的概率，并最小化其他类别的概率，从而提高模型的分类性能。它的主要作用是将一组数值转换为概率分布，使得每个数值的范围在 0 到 1 之间，并且它们的总和为 1。

Softmax 函数的定义如下：

$$p_i = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}}$$

Softmax 函数的输出具有以下特性：

1. 输出的概率在 0 到 1 之间，可视为每个类别的预测概率。
2. 所有类别的概率总和为 1，因此可以解释为一个概率分布。

在多类别分类问题中，Softmax 函数通常用于神经网络的输出层，将网络的原始输出转化为各个类别的概率分布。在训练过程中，Softmax 函数帮助最大化正确类别的概率，并最小化其他类别的概率，从而提高模型的分类性能。

在输入与输出描述的矩阵$\hat{Y}$中，$\hat{y}_{ij} = \frac{e^{z_{ij}}}{\sum_{p=1}^{k} e^{z_{pj}}}$，表示模型输出中第j（0<j<=60000）个样本属于第i类的概率。

交叉熵损失函数

在 Softmax 中，可以用交叉熵损失函数来衡量模型输出的概率分布与实际标签之间的差异。交叉熵损失函数是多分类任务中常用的损失函数，尤其在神经网络训练中广泛使用；与 Softmax 结合起来，可以更好地引导模型学习正确的分类概率分布

1. 交叉熵损失的计算方法：

本次的手写数字识别任务是一个典型的多分类任务，在这样的情况下某一次识别的交叉熵损失函数计算公式为：

$$L(y, \hat{y}) = -\sum_{i=1}^{C} y_i \cdot \log(\hat{y}_i)$$

其中，C 是类别数，y_i 是实际标签的独热编码，$\hat{y}_i$ 是模型对类别 i 的预测概率。

又由于代码中训练时的实际标签会被设置为只有正确项$train_label[i]$为1，其余项为0的独热编码，因此对于某一次预测的交叉熵损失值可以简单由

$$L(y, \hat{y}) = -\log(\hat{y}_{train\_lable[i]})$$

计算得出，其中$train_label[i]$是这一次预测的真实值；

在本次实验中，将60000个样本的交叉熵取均值定义为模型的总损失函数：

$$F=\sum_{i=0}^{m}(-\sum_{j=1}^{k}y_{ij}\log\hat{y}_{ij})/m=\sum_{i=0}^{m}-\log(\hat{y}_{train\_lable[i]})/m$$

将代价函数F视为参数θ的函数J_θ,这就是我们要优化的目标。

2. 交叉熵损失的意义：

- **对数似然性质**： 交叉熵损失刻画了实际标签和模型预测之间的负对数似然，最小化交叉熵损失值，等价于最大化模型对实际标签的似然，也就是说最大化模型的准确率；
- **概率分布匹配**： 交叉熵损失的形式促使模型的输出概率分布与实际标签的分布尽可能接近，使得模型能更加专注于正确分类的类别，从而提升准确性。

梯度下降

梯度下降是一种迭代优化算法，用于最小化或最大化目标函数。在机器学习和深度学习中，梯度下降主要用于调整模型参数，使得模型能够更好地拟合训练数据或最小化损失函数。

由链式法则可知，

$$\frac{\partial J_\Omega}{\partial\omega_{pj}}=\sum_{i=1}^{m}\frac{\partial J_\Omega}{\partial z_{ij}}\frac{\partial z_{ij}}{\partial\omega_{pj}}$$

而

$$\frac{\partial J_\Omega}{\partial z_{ip}}=\frac{\partial\sum_{q=0}^{m}(-\sum_{j=1}^{k}\ln\hat{y}_{qj})}{\partial z_{ip}}=-\sum_{j=1}^{k}\frac{y_{ij}}{\hat{y}_{ij}}\frac{\partial\hat{y}_{ij}}{\partial z_{ip}}$$

由于

$$\frac{\partial\hat{y}_{ij}}{\partial z_{ip}}=\begin{cases}=\dfrac{e^{z_{ij}}}{\sum_{q=1}^{k}e^{z_{iq}}}-\dfrac{e^{z_{ij}}e^{z_{ip}}}{(\sum_{q=1}^{k}e^{z_{iq}})^2}=\hat{y}_{ip}-\hat{y}_{ip}^2 & j=p\\ =-\dfrac{e^{z_{ij}}e^{z_{ip}}}{(\sum_{q=1}^{k}e^{z_{iq}})^2}=-\hat{y}_{ij}\hat{y_i}p & j\neq p\end{cases}$$

由于$\sum_{j=1}^{k}y_{ij}=1$,有

$$\frac{\partial J_\Omega}{\partial z_{ip}}=\hat{y}_{ip}-y_{ip}$$

考虑$z_{ip}=\sum_{q=0}^{n}x_{iq}\omega_{qp}$，得到：

$$\frac{\partial J_\Omega}{\partial\omega_{pj}}=\sum_{i=1}^{m}\frac{\partial J_\Omega}{\partial z_{ij}}x_{ip}=\sum_{i=1}^{m}(\hat{y}_{ij}-y_{ij})x_{ip}$$

写回矩阵，有：

$$\nabla J_\Omega = \begin{bmatrix} \frac{J_\Omega}{\omega_{01}} & \frac{J_\Omega}{\omega_{02}} & \cdots & \frac{J_\Omega}{\omega_{0n}} \\ \frac{J_\Omega}{\omega_{11}} & \frac{J_\Omega}{\omega_{12}} & \cdots & \frac{J_\Omega}{\omega_{1n}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{J_\Omega}{\omega_{k1}} & \frac{J_\Omega}{\omega_{k2}} & \cdots & \frac{J_\Omega}{\omega_{kn}} \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^m x_{i0}(\hat{y}_{i1} - y_{i1}) & \sum_{i=1}^m x_{i0}(\hat{y}_{i2} - y_{i2}) & \cdots & \sum_{i=1}^m x_{i0}(\hat{y}_{in} - y_{in}) \\ \sum_{i=1}^m x_{i1}(\hat{y}_{i1} - y_{i1}) & \sum_{i=1}^m x_{i1}(\hat{y}_{i2} - y_{i2}) & \cdots & \sum_{i=1}^m x_{i1}(\hat{y}_{in} - y_{in}) \\ \vdots & \vdots & \ddots & \cdots \\ \sum_{i=1}^m x_{ik}(\hat{y}_{i1} - y_{i1}) & \sum_{i=1}^m x_{ik}(\hat{y}_{i2} - y_{i2}) & \cdots & \sum_{i=1}^m x_{ik}(\hat{y}_{in} - y_{in}) \end{bmatrix}$$

$$= (X.T \cdot (\hat{Y} - Y).T).T$$

因此在本次实验中，θ的梯度可以用$(X.T \cdot (\hat{Y} - Y).T).T$求出，下一步即按照指定学习率使用梯度下降方法：

$$\theta_{t+1} = \theta_t - \alpha \nabla J(\theta_t)$$

程序设计

本次实验主要实现**softmax_regression**和**cal_accuracy**两个函数

softmax_regression

```
def softmax_regression(theta, x, y, iters, alpha):
    # TODO: Do the softmax regression by computing the gradient and
    # the objective function value of every iteration and update the theta
    x = x.T  #如同上述，需要将X进行转置为形状()，方便后续运算
    Loss = [] #用于记录每次迭代后的损失值
    for i in tqdm(range(iters)):  #tqdm用于显示迭代进度与速度
        result = theta@x
        res_exp = np.exp(result)
        exp_sum = res_exp.sum(axis=0) #求每列之和，即每个训练样本，在0-9上的计算结果之和

        res_exp = res_exp/exp_sum[None,:] # exp/exp.sum(),每个z_ij除以每列之和，进行softmax
        #res_exp即是y_hat

        loss = np.log(res_exp) * y  #由于res_exp和y形状相同，可以通过二者的对位乘法，计算损失值
        f = -loss.sum()/x.shape[1]  #求取60000个样本的平均损失值，作为模型的损失值
        Loss.append(f)              #加入Loss列表

        grad = x@(res_exp-y).T      #如同原理部分所述，计算出梯度矩阵

        if i % 50 == 0 and i>0:     #学习率衰退，每50轮衰减0.95
            alpha = alpha *0.95

        theta = theta - alpha*1/x.shape[1]*grad.T  #更新theta矩阵


    return Loss,theta  #此处做了修改，为了方便分析损失值变化情况，会将Loss列表一同返回
```

cal_accuracy

```
def cal_accuracy(y_pred, y):
    # TODO: Compute the accuracy among the test set and store it in acc
    y=y.reshape(len(y))
    res = y_pred-y    #相减，相同的项相减后为0
    incorrect = np.count_nonzero(res)   #计算有多少个不为0的项，即y_pred和y不相同的项的个数
    acc = 1-incorrect/len(y)          #计算正确率

    return acc
```

优化方法

在最初只完成了简单的softmax_regression和cal_accuracy函数时，训练的速度异常缓慢，并且最终的正确率也只是在90%左右徘徊，经过多次尝试，最终找到了能大幅提升训练速度，和提升正确率的方法

用矩阵乘法和并行计算替代循环

在最初的程序设计中，计算softmax和损失值f时使用了循环的方法，导致效率低下，最终用了一些能够并行计算的代码和矩阵乘法等，使得python内部能通过并行计算的优化提升训练速度，例如：

```
exp_sum = res_exp.sum(axis=0) #求每列之和，即每个训练样本，在0-9上的计算结果之和
res_exp = res_exp/exp_sum[None,:] # exp/exp.sum(),每个z_ij除以每列之和，进行softmax
#res_exp即是y_hat

loss = np.log(res_exp) * y  #由于res_exp和y形状相同，可以通过二者的对位乘法，计算损失值
f = -loss.sum()/x.shape[1]  #求取60000个样本的平均损失值，作为模型的损失值
```

这样的方法可以避免使用循环，利用并行计算的优势提升速度。

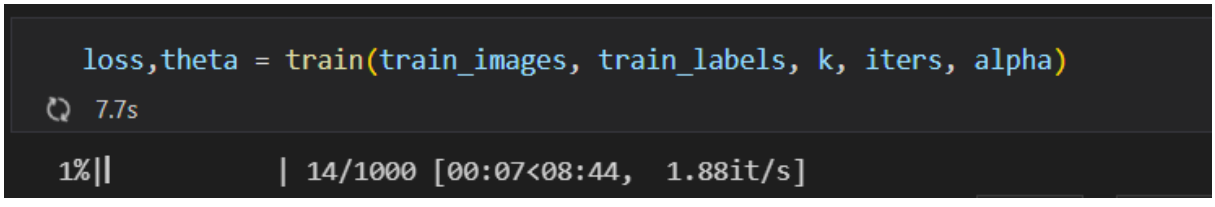
提前转换参数为float型大幅提升训练速度

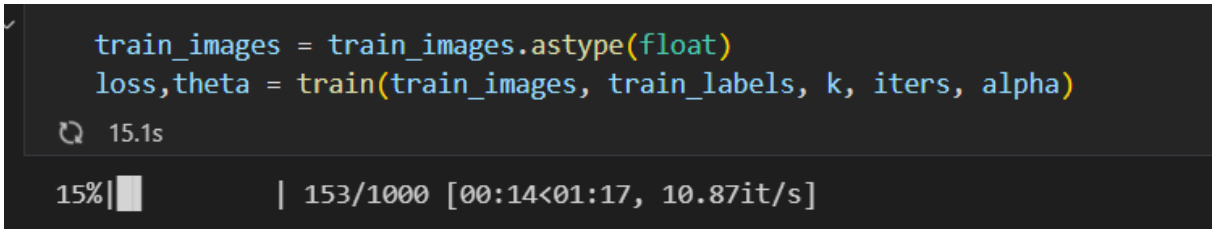
在最初的尝试中发现，将train_images矩阵进行除法操作后（例如train_images = train_images /1.0），训练速度会大幅提升，从原先的1.5it/s能提升至10it/s，经过向助教老师询问和查找相关资料后，发现这是因为：最初的输入矩阵X的元素都是int32类型，而在训练过程中，其会不停与θ的float类型的矩阵相乘，每次相乘都要经历一遍漫长的类型转换，导致训练速度降低，因此，在训练前可以通过简单的

```
train_images = train_images.astype(float)
```

做到事先将train_images转换为float类型的矩阵，从而大幅提升训练速度。

经实际检验，速度提升的效果如下：





即使是1000轮的迭代，也能在不到2min的时间完成训练

学习率α衰变

在训练过程中，学习率是一个十分重要的参数，如果学习率过低，会导致模型收敛过慢，可能使得最终结果不理想；学习率设置过高，可能导致模型过拟合，也会使得模型效果不理想，最终我采用的方案是，设置初始学习率为0.75，并且每50次迭代按照0.95的比例进行学习率的衰退：

```
if i % 50 == 0 and i>0:      #学习率衰退，每50轮衰减0.95
    alpha = alpha *0.95
```

经测试，这样能达到一个不错的效果

data_convert进行调优

在阅读原代码中，发现data_convert中有很重要的一步：

```
x[x<=40]=0
x[x>40]=1
```

这一步将原先的train_images按照各个像素的灰度进行了0-1映射，灰度大于40的映射为1，小于等于40的映射为0。

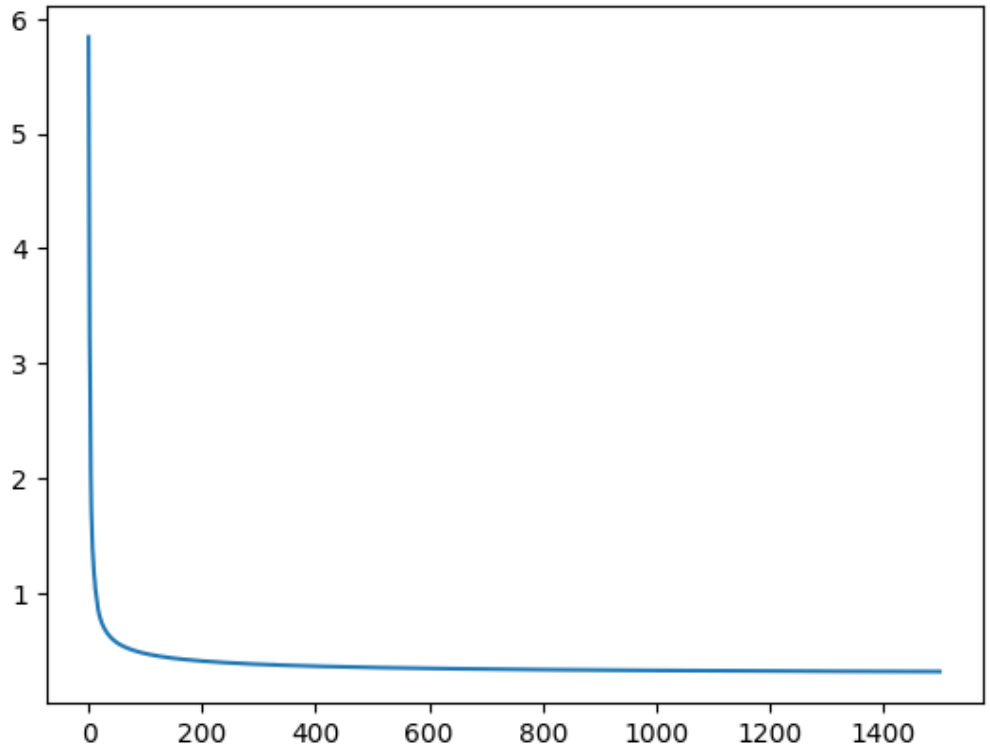
在实际训练中，我尝试更换了这个0-1映射的阈值，将其替换为100、120等值并观测效果，最终将其设置为100：

```
# in data_convert()
x[x<=100]=0
x[x>100]=1
```

最终结果

在1500次迭代，学习率初始设置为0.75，并且进行了类型转换、调整了0-1映射的阈值之后，得到的结果如下：

识别的正确率为**93.57%**，而模型的损失率如下图，不断下降最终在0.31左右：


6
5
4
3
2
1
0
200
400
600
800
1000
1200
1400