人工神经网络模型与算法

2025-3-11

姓名： 李丰克

专业： 自动化（控制）

年级： 大二

学号： 3230105182

Chapter 1 基于numpy手工实现多层感知机模型

摘要：本章简述了MLP的原理以及反向传播算法的模型，同时基于python的numpy库实现了该算法，并在FashionMNIST分类数据集上进行了实验，尝试了不同的网络参数、激活函数、优化算法来提高算法的性能以及准确率。

1.1 问题简介

简述反向传播算法的计算图表示方法。

手动搭建ANN并做一做一些其他有趣的分类集合,比如：FashionMNIST，MedMNIST。

尝试不同的网络参数：全连接网络[784, 28, 10] 或更多，尝试用不同的激活函数，尝试用不同的优化算法；改善程序性能，给出你的最优方案。

1.2 模型与算法介绍

1.2.1 多层感知机模型（MLP）

如图1所示，多层感知机的基本思想为：前一层的输出作为下一层的输入。在每一层中，输出的每个神经元等于所有输入的神经元乘以对应的权重再加上对应的偏置项得到的结果再经过激活函数得到的值，每层的神经元数量可各不相同，选用的激活函数也各不相同，因此会出现各种形状的权重矩阵和偏置项，调整这些权重矩阵以及偏置项参数以提高模型的准确率是我们要做的工作。

假设共有n层网络，用n维向量存储每层的神经元个数$D = [D_0\ D_1\ ...\ D_n]$，我们令第K层的输出为$X^{(K)}$，该输出经过激活函数前的值为$P^{(K)}$，该层的权重矩阵为$W^{(K)}$，偏置向量为$b^{(K)}$。可得：$P^{(K)} = W^{(K)}X^{(K-1)} + b^{(K)}, P^{(K)} = f(X^{(K)})$，其中$W^{(K)}$为$D_k \times D_{k-1}$的矩阵，$b^{(K)}$为$D_k \times 1$的向量。

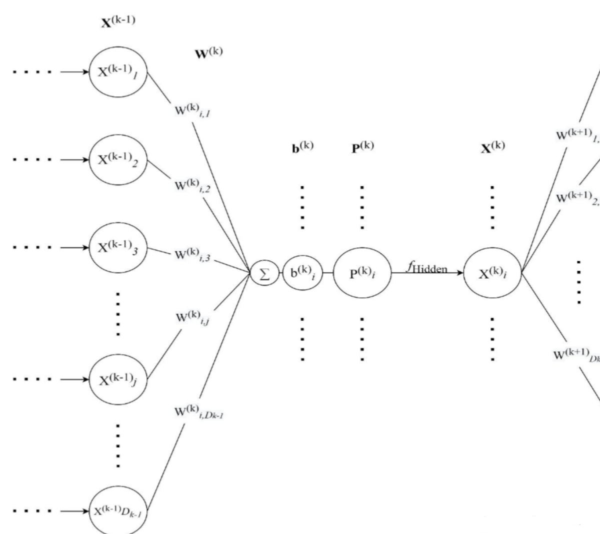


图 1: 多层感知机(第k层第i个神经元)

中间层统称为隐藏层，其输出层还会再接入一个损失函数Loss(y)，损失函数可以看作模型准确率的指标，为了使损失函数最小化，选择用梯度下降法来调整各参数。

1.2.2 反向传播算法

梯度下降法的基本思想为通过计算损失函数相对于每个参数的梯度，并沿着梯度的反方向更新参数值。MLP本质上可看作输出为Loss(y)，输入为$X^{(0)}$的多元函数，每层每个权重、偏置都是其参数，所以我们可以求其对每个参数的偏导再逐步更新

参数值。以下是简要推导过程[1, 2]：

$$\frac{\partial\mathcal{L}}{\partial\mathbf{W}^k}=\frac{\partial\mathcal{L}}{\partial\mathbf{P}^k}\cdot\frac{\partial\mathbf{P}^k}{\partial\mathbf{W}^k}\quad\frac{\partial\mathcal{L}}{\partial\mathbf{b}^k}=\frac{\partial\mathcal{L}}{\partial\mathbf{P}^k}\cdot\frac{\partial\mathbf{P}^k}{\partial\mathbf{b}^k}$$

先求$\partial\mathcal{L}/\partial\mathbf{P}^k$，令其等于$\boldsymbol{\delta}^k$。

$$\begin{aligned}
\delta_i^k &= \frac{\partial\mathcal{L}}{\partial P_i^k}\\
&= \frac{\partial\mathcal{L}}{\partial X_i^k}\cdot\frac{\partial X_i^k}{\partial P_i^k}\\
&= (\frac{\partial\mathcal{L}}{\partial P_1^{k+1}}\cdot\frac{\partial P_1^{k+1}}{\partial X_i^k}+\frac{\partial\mathcal{L}}{\partial P_2^{k+1}}\cdot\frac{\partial P_2^{k+1}}{\partial X_i^k}+\cdots+\frac{\partial\mathcal{L}}{\partial P_{D_{k+1}}^{k+1}}\cdot\frac{\partial P_{D_{k+1}}^{k+1}}{\partial X_i^k})(f'(P_i^k))\\
&= (\sum_{j=1}^{D_{k+1}}\frac{\partial\mathcal{L}}{\partial P_j^{k+1}}\cdot\frac{\partial P_j^{k+1}}{\partial X_i^k})(f'(P_i^k))\\
&= (\sum_{j=1}^{D_{k+1}}\frac{\partial\mathcal{L}}{\partial P_j^{k+1}}\cdot\frac{\partial(\sum_{m=1}^{D_k}W_{j,m}^{k+1}\cdot X_m^k+b_j^{k+1})}{\partial X_i^k})(f'(P_i^k))\\
&= (\sum_{j=1}^{D_{k+1}}\frac{\partial\mathcal{L}}{\partial P_j^{k+1}}\cdot\frac{\partial(W_{j,1}^{k+1}\cdot X_1^k+W_{j,2}^{k+1}\cdot X_2^k+\cdots+W_{j,i}^{k+1}\cdot X_i^k+\cdots+W_{j,D_k}^{k+1}\cdot}{\partial X_i^k}\\
&= (\sum_{j=1}^{D_{k+1}}\delta_j^{k+1}\cdot W_{j,i}^{k+1})(f'(P_i^k))
\end{aligned}$$

可得δ_i^k是行向量$\boldsymbol{\delta}^{k+1}$和列向量 $\mathbf{W}_{:,i}^{k+1}$的点乘再乘激活函数对应的偏导，因此有$\boldsymbol{\delta}^k=(\boldsymbol{\delta}^{k+1}\cdot\mathbf{W}^{k+1})\odot f'(\mathbf{P}^k)$。发现有自后向前的迭代关系。然后再代回偏导公式：

$$\begin{aligned}
\frac{\partial\mathcal{L}}{\partial W_{i,j}^k} &= \frac{\partial\mathcal{L}}{\partial P_i^k}\cdot\frac{\partial P_i^k}{\partial W_{i,j}^k}\\
&= \delta_i^k\\
&\cdot\frac{\partial(W_{i,1}^k\cdot X_1^{k-1}+W_{i,2}^k\cdot X_2^{k-1}+\cdots+W_{i,j}^k\cdot X_i^{k-1}+\cdots+W_{i,D_{k-1}}^k\cdot X_{D_{k-1}}^{k-1}+b_i^k)}{\partial W_{i,j}^k}\\
&= \delta_i^k\cdot X_i^{k-1}
\end{aligned}$$

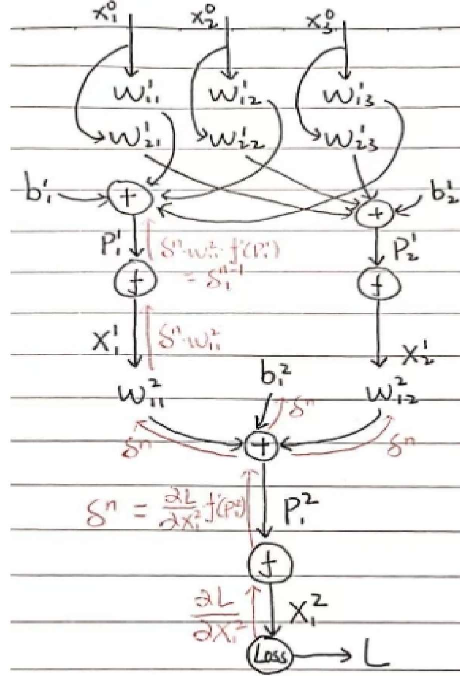
$$\begin{aligned}
\frac{\partial\mathcal{L}}{\partial b_i^k} &= \frac{\partial\mathcal{L}}{\partial P_i^k}\cdot\frac{\partial P_i^k}{\partial b_i^k}\\
&= \delta_i^k\\
&\cdot\frac{\partial(W_{i,1}^k\cdot X_1^{k-1}+W_{i,2}^k\cdot X_2^{k-1}+\cdots+W_{i,j}^k\cdot X_i^{k-1}+\cdots+W_{i,D_{k-1}}^k\cdot X_{D_{k-1}}^{k-1}+b_i^k)}{\partial b_i^k}\\
&= \delta_i^k
\end{aligned}$$

最终得到

$$\frac{\partial\mathcal{L}}{\partial\mathbf{W}^k}=(\boldsymbol{\delta}^k)^T\cdot(\mathbf{X}^{k-1})^T$$
$$\frac{\partial\mathcal{L}}{\partial\mathbf{b}^k}=(\boldsymbol{\delta}^k)^T$$

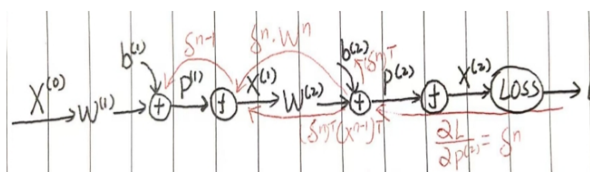
故要先解得损失函数对于输出层输出的偏导，然后再以此递推。

1.2.3 反向传播的计算图简述



如图是手绘的一个三层神经网络的计算图，其神经元个数分别为3，2，1。该计算图中共有三种节点：加法器，乘法器和函数，前向传播时对其进行对应的运算，根据链式法则，反向传播时，每经过一个节点乘上对应的偏导运算。经过加法器时偏导为1；经过乘法器时偏导为放大增益；经过函数时，偏导为函数对运算值的偏导数。

从图中可以看出$\boldsymbol{\delta}^n$可直接求，从P_1^2经过加法器的过程中，可以直接得到b_1^2，这符合前面对$\frac{\partial\mathcal{L}}{\partial\mathbf{b}^k}=(\boldsymbol{\delta}^k)^T$的推导。在经过乘法器时，将对应的X看作增益，即可得到对W的偏导，符合$\frac{\partial\mathcal{L}}{\partial\mathbf{W}^k}=(\boldsymbol{\delta}^k)^T\cdot(\mathbf{X}^{k-1})^T$。将W看为增益，再经过函数，便可得到下一个$\boldsymbol{\delta}^{n-1}$，符合$\boldsymbol{\delta}^k=(\boldsymbol{\delta}^{k+1}\cdot\mathbf{W}^{k+1})\odot f'(\mathbf{P}^k)$的推导。



如图用矩阵表示计算图的各环节，其中各层的偏导数的迭代关系则一目了然，再按照矩阵的运算规律适当变形，便可得出相同的推导规律。

1.3 算法实现

为了实现MLP，我们需要事先准备：神经网络层数及各层神经元个数，前向传播函数，反向传播函数，各种激活函数及其导数，各种优化算法，损失函数（这里默认为均方误差）。

定义了tanh(x)，sigmoid(x)，RELU(x)，RELU3(x)四种激活函数及其导数。权重矩阵与偏置矩阵的初始化如下：

```
#权重与偏置初始化
def WandBInit(layers):
    weights=[]
    thetas=[]
    for i in range(len(layers)-1):
        weights.append(2*(np.random.rand(layers[i+1],layers[i]))-1)
        thetas.append(2*(np.random.random(layers[i+1]))-1)
        thetas[i]=np.asmatrix(thetas[i]).T
    return weights,thetas
```

反向传播函数如下：

```
#定义反向传播函数
def backpropagation(weights,thetas,activation,activation_deriv,x,y):
    #error是损失函数对最后一次输出的直接偏导数，这个与损失函数的选择有关，这里选择用
    #均方误差函数，其对输出求偏导结果为y_pred-y_true，即输出值-真实值
    error=0
    y_true=propagation(weights,thetas,activation,x,1)
    for i in range(len(y)):
        error+=y_true[i][0]-y[i]
    n_w=len(weights)
    z=[]#每一层的不经激活函数的输出，即P
    K=[]#每一层经过激活函数的输出，从输入层开始
    dweights=[]#损失函数L对每个权重的偏导W的矩阵
    dthetas=[]#损失函数L对每个偏置b的偏导的矩阵|
    delta=[]#损失函数L对每一层输出Pi的偏导
    for i in range(n_w):#前向传播得到每层输出
        if i==0:
            z.append(np.dot(weights[i], x) +thetas[i])
            K.append(x)
            delta.append(x)
        else:
            z.append(np.dot(weights[i],activation(z[i-1]))+thetas[i])
            K.append(activation(z[i-1]))
            delta.append(activation(z[i-1]))
    for i in range(n_w-1, -1, -1):#得到全部delta
        if i==n_w-1:
            delta[i] = np.multiply(error,activation_deriv(z[i]))#得到delta^n
        else:
            delta[i] = np.multiply(np.dot(weights[i+1].T, delta[i+1]),activation_deriv(z[i]))
            #依次迭代得到每个delta
    for i in range(n_w):#得到全部偏导
        dweights.append(np.dot(delta[i], K[i].T))
        dthetas.append(delta[i])
    return dweights, dthetas
```

梯度下降优化函数如下：

```
#定义梯度下降优化算法
def GD(weights,thetas,layers,X,Y,k,learning_rate,activation,activation_deriv):
    perf=0#perf是总损失
    ddweights=[]#更新后的权重
    ddthetas=[]#更新后的偏置
    for i in range(len(layers)-1):#形状初始化
        ddweights.append(np.zeros((layers[i+1], layers[i])))
        ddthetas.append(np.zeros(layers[i+1]))
        ddthetas[i] = np.asmatrix(ddthetas[i]).T
    for j in range(int(X.shape[1])):
        input = np.asmatrix(X[:,j]).T
        output = propagation(weights,thetas,activation,input,1)
        y = np.asmatrix(Y[:,j]).T
        error = y - output
        perf += 1.0/2 * np.sum(np.multiply(error, error))#均方误差损失函数
        dweights, dthetas = backpropagation(weights,thetas,activation,activation_deriv,x,y)
        for i in range(len(weights)):#更新参数
            ddweights[i] += k * 1.0/len(X)*learning_rate * dweights[i]
            ddthetas[i] += k * 1.0/len(X)*learning_rate * dthetas[i]
    for i in range(len(weights)):#赋值
        weights[i] += ddweights[i]
        thetas[i] += ddthetas[i]
    return perf,weights,thetas
```

前向传播以及其他优化算法均见代码附录。

1.4 数据集及算例参数介绍

1.4.1 数据集

选用的数据集为FashionMNIST，是一个分类数据集，其包含60000个训练样本和10000个测试样本，每个样本都是28×28的灰度图，其标签从0到9分别对应了10种时尚产品。

从本地加载数据，由于数据包为gz压缩，调用gzip库解压缩并将训练图像和测试图像reshape为28×28的灰度图矩阵，可以抽取前若干张用matplotlib库输出图像并看是否与标签对应，保证数据正确引用。

引用后得到的train-images是一个$60000\times28\times28$的三维数据，将每个28×28数据reshape为784×1的列向量依次作为输入，train-labels作为标签reshape为60000×1的向量，测试集同理。

由于标签涉及0-9，激活函数值域不够，最后输出时需要相应的放大倍数k，比如对sigmoid函数来说k取10。

1.4.2 算例参数以及各种优化策略选择

共有三个变量：优化算法，网络参数，激活函数。受限于电脑性能，每次迭代时间太长，因此在决定使用哪种策略更好的时候并不需要把迭代次数设置的太多以使得loss最小，准确率最高，可以通过控制变量的方法来确定，暂且假设三个因素是相互独立的。同时权重与偏置只初始化一次，确保所有实验是在同一个基础上开始的。

选择优化算法：对于GD(批量梯度下降)算法来说，本数据集数据量太大了，无论是从CPU占用，还是迭代时间上来说，本电脑都无法承受，因此选择取数据集的前1%来进行训练;对于SGD(随机梯度下降)，由于其原理是随机抽取小批量样本进行随机训练，虽然训练过程不稳定，多少效率高，可以对全部数据进行训练;对于ADAM，其过程中也抽取了一部分样本进行随机训练，速度略快于GD。因此定为:

网络参数	[784,256,128,1]
激活函数	sigmoid
放大倍数	k=10
学习率	0.1
更新缩放因子	m=1
迭代次数	1000

其中对于GD选取前1%的数据，SGD与ADAM选择全部数据。

选择网络参数：由于SGD与ADAM具有随机性，选用GD作为优化算法，网络参数分为两个极端：一种是层数少，相邻层间跨度大;另一种是层数多，相邻层间平滑过渡。因此可以选择三种网络层数[784,256,1],[784,256,128,1],[784,392,196,49,1]。其他参数如下：

优化算法	GD
激活函数	sigmoid
放大倍数	k=10
学习率	0.1
更新缩放因子	m=1
迭代次数	1000

选择激活函数：激活函数共有四个，分别为tanh，sigmoid，RELU，RELU3。对于tanh与sigmoid函数放大倍数要选k=10，其余两个k=1即可。其他参数如下：

优化算法	GD
网络参数	[784,256,128,1]
学习率	0.1
更新缩放因子	m=1
迭代次数	1000

由此可以组合出性能较佳的参数，但是不一定是最佳的，因为算法中存在的随机性，或者各条件直接并不是绝对的独立关系，或者与数据集本身特点有关等等。

1.5 结果汇总

事实证明，仅仅一千次迭代结果并不能得到较高的准确率，但是足以反映各参数之间的区别。

以优化算法为变量：

```
Epoch 994: Loss = 210.0200038145937
Epoch 995: Loss = 209.81944492602048
Epoch 996: Loss = 209.61920815033838
Epoch 997: Loss = 209.41929290312945
Epoch 998: Loss = 209.21969859901085
Epoch 999: Loss = 209.02042465163439
```

```
Epoch 950: Loss = 919.8444236971885
Epoch 960: Loss = 787.0678279017072
Epoch 970: Loss = 803.1193932931988
Epoch 980: Loss = 884.1280940442069
Epoch 990: Loss = 910.0213626553469
```

```
Epoch 996: Loss = 8657.5
Epoch 997: Loss = 8289.5
Epoch 998: Loss = 8239.0
Epoch 999: Loss = 8086.5
```

自上而下依次为GD，SGD，ADAM三种优化算法的迭代情况，由于对损失函数的取值不同。ADAM取数据集的百分之十而SGD取百分之一，故刚好差1个数量级。SGD与ADAM的收敛过程为振荡收敛，而GD为单调收敛。收敛速度SGD是最快的，而GD是最慢的(如果以全部数据来训练)。

三者在测试集上的准确率依次为0.2447, 0.2485, 0.2515， GD的训练集太小，泛化能力不够，如果让其在整个数据集上训练，训练时间太长，耗费太大，但是一定能够收敛到一个较合适的值。SAG迭代速度快，适合处理大数据，且准确率也尚可。ADAM的准确率最高，说明其收敛性最好，得益于其自动调整学习率的算法原理。

以激活函数为变量：

```
Epoch 994: Loss = 210.0200038145937
Epoch 995: Loss = 209.81944492602048
Epoch 996: Loss = 209.61920815033838
Epoch 997: Loss = 209.41929290312945
Epoch 998: Loss = 209.21969859901085
Epoch 999: Loss = 209.02042465163439
```

```
Epoch 995: Loss = 7957.346738689841
Epoch 996: Loss = 7523.692390601167
Epoch 997: Loss = 11524.001073726467
Epoch 998: Loss = 11340.591088827683
Epoch 999: Loss = 10704.54210360599
```

```
Epoch 995: Loss = 6009.0
Epoch 996: Loss = 6009.0
Epoch 997: Loss = 6009.0
Epoch 998: Loss = 6009.0
Epoch 999: Loss = 6009.0
```

```
Epoch 995: Loss = 6009.0
Epoch 996: Loss = 6009.0
Epoch 997: Loss = 6009.0
Epoch 998: Loss = 6009.0
Epoch 999: Loss = 6009.0
```

依次为sigmoid,tanh,RELU,RELU3为激活函数对应的迭代过程。奇怪的是，RELU与RELU3作为激活函数时，总会陷入死循环，使神经元失效，可能与数据集自身有关，也可能在算法某些过程上出现了问题，目前仍未排查。

对tanh函数，其迭代过程始终在振荡，损失在7000 12000的区间内振荡，始终不能再下降，推测应该是学习率设置的太高的问题，将学习率设置为0.01，再跑一次。

```
Epoch 995: Loss = 2970.667548326755
Epoch 996: Loss = 2592.9146275539783
Epoch 997: Loss = 2969.613239908594
Epoch 998: Loss = 2592.140719720677
Epoch 999: Loss = 2968.5524827996396
```

发现整体的损失减小了，但是仍然存在振荡不收敛的现象，说明学习率仍然高，只要再适当减小学习率，理论上可以收敛。这也说明了tanh函数受学习率影响较大，适用于较小学习率，这也使得其收敛速度变慢。目前来说，还是sigmoid性能最好。

以网络参数为变量：

```
Epoch 995: Loss = 828.970839980209
Epoch 996: Loss = 750.9533274225571
Epoch 997: Loss = 828.3051932309572
Epoch 998: Loss = 750.281835245485
Epoch 999: Loss = 827.6397561278371
```

```
Epoch 994: Loss = 210.0200038145937
Epoch 995: Loss = 209.81944492602048
Epoch 996: Loss = 209.61920815033838
Epoch 997: Loss = 209.41929290312945
Epoch 998: Loss = 209.21969859901085
Epoch 999: Loss = 209.02042465163439
```

```
Epoch 995: Loss = 273.0550110958664
Epoch 996: Loss = 272.7778542055008
Epoch 997: Loss = 272.5010582501921
Epoch 998: Loss = 272.22462291103255
Epoch 999: Loss = 271.94854786671107
```

自上而下依次为[784,256,1]，[784,256,128,1]，[784,392,196,49,1]对应的迭代过程。由此可以发现，神经网络层数越多，层间过渡越平稳，参数收敛越快，性能越好，但是相应的每次迭代所花费的时间也越多

```
749.6116537645371
0.1489

208.82147047368906
0.2447

271.6728327935459
0.3215
```

以上为三种参数在测试集上的损失以及准确率，迭代次数有限情况下，准确率与收敛速度挂钩，与前面的规律也相吻合。如果不考虑时间花费的话，一定是[784,392,196,49,1]性能最优。如果要综合考虑数据集大小、时间花费的话，还是选择折中的[784,256,128,1]较好。

1.6 结论

综上，在三个变量相互独立的假设下，选择[784,256,128,1]，sigmoid激活函数，ADAM优化算法的优化策略，使模型无论从收敛速度，准确率，时间花费上都最佳。

其他的参数为：放大倍数k=10，缩放因子m=1，学习率0.1，迭代次数10000（尽量大）。

1.7 代码附录

```
import numpy as np
import struct

def tanh(x):
return np.tanh(x)
def sigmoid(x):
return 1.0 / (1 + np.exp(-1.0*x))
def RELU(x):
return np.maximum(0,x)
def RELU3(x):
return np.multiply(np.multiply(RELU(x),RELU(x)),RELU(x))

def tanh_deriv(x):
return 1.0 - np.multiply(tanh(x),tanh(x))
def sigmoid_deriv(x):
return np.multiply(1.0 - sigmoid(x), sigmoid(x))
def RELU_deriv(x):
return x>=0
def RELU3_deriv(x):
return 3.0 * np.multiply(RELU(x),RELU(x))

def WandBInit(layers):
weights=[]
thetas=[]
for i in range(len(layers)-1):
weights.append(2*(np.random.rand(layers[i+1],layers[i]))-1)
thetas.append(2*(np.random.random(layers[i+1]))-1)
thetas[i]=np.asmatrix(thetas[i]).T
return weights,thetas

def propagation(weights,thetas,activation,x,k):
temp = x
for i in range(len(weights)):
temp=activation(np.dot(weights[i],temp)+thetas[i])
z = k*temp
return z

def backpropagation(weights,thetas,activation,activation_deriv,x,y,k):

error=0
y_true=propagation(weights,thetas,activation,x,k)
for i in range(len(y)):
error+=y_true[i][0]-y[i]
n_w=len(weights)
z=[]
K=[]
dweights=[]
dthetas=[]
delta=[]
for i in range(n_w):
if i==0:
z.append(np.dot(weights[i], x) + thetas[i])
K.append(x)
delta.append(x)
else:
z.append(np.dot(weights[i],activation(z[i-1]))+thetas[i])
K.append(activation(z[i-1]))
delta.append(activation(z[i-1]))
for i in range(n_w-1, -1, -1):
if i==n_w-1:
```

```
            delta[i] = np.multiply(error, activation_deriv(z[i]))
        else:
            delta[i] = np.multiply(np.dot(weights[i+1].T, delta[i+1]), activation_deriv(z[i]))
    for i in range(n_w):
        dweights.append(np.dot(delta[i], K[i].T))
        dthetas.append(delta[i])
    return dweights, dthetas

def GD(weights,thetas,layers,X,Y,k,learning_rate,activation,activation_deriv,m):
    perf=0
    ddweights=[]
    ddthetas=[]
    for i in range(len(layers)-1):
        ddweights.append(np.zeros((layers[i+1], layers[i])))
        ddthetas.append(np.zeros(layers[i+1]))
        ddthetas[i] = np.asmatrix(ddthetas[i]).T
    for j in range(int(X.shape[1])):
        input = np.asmatrix(X[:,j]).T
        output = propagation(weights,thetas,activation,input,k)
        y = np.asmatrix(Y[:,j]).T
        error = y - output
        perf += 1.0/2 * np.sum(np.multiply(error, error))
        dweights, dthetas = backpropagation(weights,thetas,activation,activation_deriv,input,y,k)
        for i in range(len(weights)):
            ddweights[i] += m * 1.0/len(X)*learning_rate * dweights[i]
            ddthetas[i] += m * 1.0/len(X)*learning_rate * dthetas[i]
    for i in range(len(weights)):
        weights[i] -= ddweights[i]
        thetas[i] -= ddthetas[i]
    return perf,weights,thetas

def SGD(weights, thetas, layers, X, Y, k, learning_rate, activation, activation_deriv, m):
    perf = 0
    ddweights = []
    ddthetas = []
    for i in range(len(layers) - 1):
        ddweights.append(np.zeros((layers[i + 1], layers[i])))
        ddthetas.append(np.zeros(layers[i + 1]))
        ddthetas[i] = np.asmatrix(ddthetas[i]).T
    rand_X = np.arange(X.shape[1])
    np.random.shuffle(rand_X)
    n = int(X.shape[1] / 100)

    for j in range(n):
        input = np.asmatrix(X[:, rand_X[j]]).T
        output = propagation(weights, thetas, activation, input, k)
        y = np.asmatrix(Y[:, rand_X[j]]).T
        error = y - output
        perf += 1.0 / 2 * np.sum(np.multiply(error, error))
        dweights, dthetas = backpropagation(weights, thetas, activation, activation_deriv, input, y, k)
        for i in range(len(weights)):
            ddweights[i] -= m * 1.0 / n * learning_rate * dweights[i]
            ddthetas[i] -= m * 1.0 / n * learning_rate * dthetas[i]
    for i in range(len(weights)):
        weights[i] += ddweights[i]
        thetas[i] += ddthetas[i]
    return perf, weights, thetas

def ADAM(weights,thetas,layers,X,Y,k,learning_rate,activation,activation_deriv,m,epochs,mw,mtheta,vw,vtheta):
    perf = 0
    beta2 = 0.999
    beta1 = 0.9
    epsilon = 0.00000001
    ddweights = []
    ddthetas = []
    for i in range(len(layers)-1):
        ddweights.append(np.zeros((layers[i+1], layers[i])))
        ddthetas.append(np.zeros(layers[i+1]))
        ddthetas[i] = np.asmatrix(ddthetas[i]).T
    rand_X = np.arange(X.shape[1])
    np.random.shuffle(rand_X)
    n = int(X.shape[1]/10)
    for j in range(n):
        input = np.asmatrix(X[:,rand_X[j]]).T
        output = propagation(weights,thetas,activation,input,k)
        y = np.asmatrix(Y[:,rand_X[j]]).T
        error = y - output
        perf += 1.0/2 * np.sum(np.multiply(error, error))

        dweights, dthetas = backpropagation(weights,thetas,activation,activation_deriv,input,y,k)
        for i in range(len(weights)):
            ddweights[i] += m * 1.0/n * dweights[i]
            ddthetas[i] += m * 1.0/n * dthetas[i]
    for j in range(len(weights)):
        mw[j] = beta1*mw[j] + (1-beta1)*ddweights[j]
```

```
146 vw[j] = beta2*vw[j] + (1-beta2)*np
        .multiply(ddweights[j],
        ddweights[j])
147 mtheta[j] = beta1*mtheta[j] + (1-
        beta1)*ddthetas[j]
148 vtheta[j] = beta2*vtheta[j] + (1-
        beta2)*np.multiply(ddthetas[j],
        ddthetas[j])
149 mwHat = mw[j]/(1 - beta1**epochs+
        epsilon)
150 vwHat = vw[j]/(1 - beta2**epochs+
        epsilon)
151 mtHat = mtheta[j] / (1-beta1**
        epochs+ epsilon)
152 vtHat = vtheta[j] / (1-beta2**
        epochs+ epsilon)
153 weights[j]+= learning_rate * np.
        multiply(mwHat, 1.0/(np.sqrt(
        vwHat) + epsilon))
154 thetas[j] += learning_rate * np.
        multiply(mtHat, 1.0/(np.sqrt(
        vtHat) + epsilon))
155 return perf,weights,thetas,mw,vw,
        mtheta,vtheta
156
157 import matplotlib.pyplot as plt
158 import gzip
159
160 def get_data():
161 train_image = r"E:\pycharm\
        pythonProject1\data\fashion\
        train-images-idx3-ubyte.gz"
162 test_image = r"E:\pycharm\
        pythonProject1\data\fashion\
        t10k-images-idx3-ubyte.gz"
163 train_label = r"E:\pycharm\
        pythonProject1\data\fashion\
        train-labels-idx1-ubyte.gz"
164 test_label = r"E:\pycharm\
        pythonProject1\data\fashion\
        t10k-labels-idx1-ubyte.gz"
165 paths = [train_label, train_image,
         test_label,test_image]
166 with gzip.open(paths[0], 'rb') as
        lbpath:
167 y_train = np.frombuffer(lbpath.
        read(), np.uint8, offset=8)
168 with gzip.open(paths[1], 'rb') as
        imgpath:
169 x_train = np.frombuffer(
170 imgpath.read(), np.uint8, offset
        =16).reshape(len(y_train), 28,
        28)
171 with gzip.open(paths[2], 'rb') as
        lbpath:
172 y_test = np.frombuffer(lbpath.read
        (), np.uint8, offset=8)
173 with gzip.open(paths[3], 'rb') as
        imgpath:
174 x_test = np.frombuffer(
175 imgpath.read(), np.uint8, offset
        =16).reshape(len(y_test), 28,
        28)
176 return x_train, y_train, x_test,
        y_test
177
178 class_names = ['T-shirt/top', '
        Trouser', 'Pullover', 'Dress',
        'Coat',
179 'Sandal', 'Shirt', 'Sneaker', 'Bag
        ', 'Ankle boot']
180 train_images,train_labels,
        test_images,test_labels=
        get_data()
181 plt.figure(figsize=(10,10))
182 for i in range(25):
183 plt.subplot(5,5,i+1)
184 plt.xticks([])
185 plt.yticks([])
186 plt.grid(False)
187 plt.imshow(train_images[i], cmap=
        plt.cm.binary)
188 plt.xlabel(class_names[
        train_labels[i]])
189 plt.show()
190
191 train_images = train_images.
        reshape(60000, 784).T
192 train_labels = train_labels.
        reshape(1, 60000)
193 test_images = test_images.reshape
        (10000, 784).T
194 test_labels = test_labels.reshape
        (1, 10000)
195 train_images = train_images /
        255.0
196 test_images = test_images / 255.0
197
198 layers=[784,392,198,49,1]
199 weights,thetas=WandBInit(layers)
200 weights_copy=weights
201 theta_copy=thetas
202 print(train_images[1].shape)
203 m=np.asmatrix(train_labels[0])
204 print(m)
205
206 train_images_GD=train_images
        [:,:600]
207 train_labels_GD=train_labels
        [:,:600]
208 test_images_GD=test_images[:,:100]
209 test_labels_GD=test_labels[:,:100]
210
211 mw=[]
212 mtheta=[]
213 vw=[]
214 vtheta=[]
215 for i in range(len(layers)-1):
216 mw.append(np.zeros((layers[i+1],
        layers[i])))
217 mtheta.append(np.zeros(layers[i
        +1]))
218 mtheta[i] = np.asmatrix(mtheta[i])
        .T
219 for i in range(len(layers)-1):
220 vw.append(np.zeros((layers[i+1],
        layers[i])))
221 vtheta.append(np.zeros(layers[i
        +1]))
```

```
vtheta[i] = np.asmatrix(vtheta[i])
    .T

epochs=1000
for epoch in range(epochs):
perf, a, b=GD(weights,thetas,
    layers,train_images_GD,
    train_labels_GD,10,0.1,sigmoid,
    sigmoid_deriv,1)
weights=a
thetas=b
print(f"Epoch {epoch}: Loss = {
    perf}")

pred=[]
perf=0
n = int(test_images.shape[1]/100)
rand_X = np.arange(test_images.
    shape[1])
for j in range(n):
input = np.asmatrix(test_images[:,
    rand_X[j]]).T
output = propagation(weights,
    thetas,sigmoid,input,10)
y = np.asmatrix(test_labels[:,
    rand_X[j]]).T
error = y - output
perf += 1.0/2 * np.sum(np.multiply
    (error, error))
print(perf)
for j in range(int(test_images.
    shape[1])):
input = np.asmatrix(test_images[:,
    j]).T
output = propagation(weights,
    thetas,sigmoid,input,10)
pred=np.append(pred,round(float(
    output[0,0])))
count=0

for i in range(int(test_images.
    shape[1])):
if pred[i]==test_labels[0][i]:
count+=1
print(count/test_images.shape[1])
```

参考文献

[1] Nielsen M. Neural Networks and Deep Learning[M]. San Francisco: Determination Press, 2015.

[2] Johnson J. CS231n: Convolutional Neural Networks for Visual Recognition[EB/OL]. Stanford: Stanford University, 2019. [2025-03-16]. https://cs231n.github.io/.