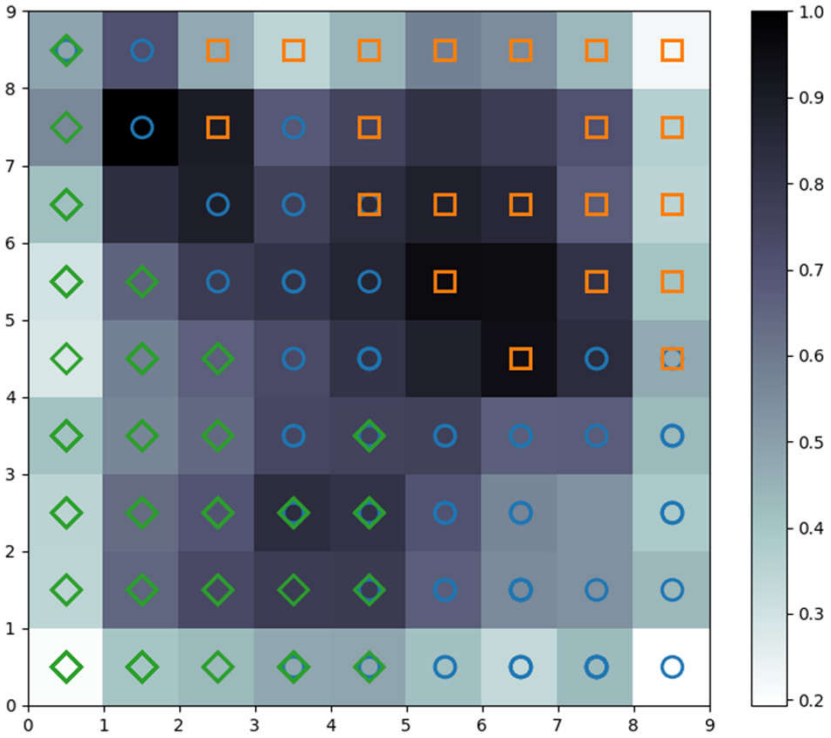
Python编程与人工智能实践

算法篇：SOM（Self Organizing Maps，自组织映射）

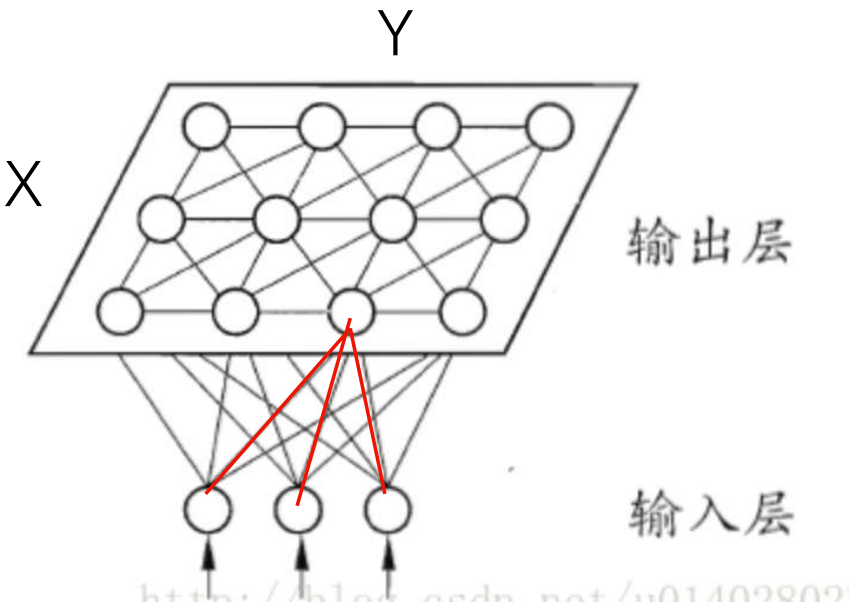


于泓
鲁东大学
信息与电气工程学院
2021.12.12

SOM 自组织（竞争型）神经网络

- SOM是神经网络的一种，它可以将相互关系复杂且非线性的高维数据，映射到具有简单几何结构及相互关系的低维空间中进行展示。（低维映射能够反映高维特征之间的拓扑结构）

- 可以实现数据可视化；聚类；分类；特征抽取等任务



输入层： D个节点， 与输入特征维度相同

输出层： X×Y个节点，排成矩阵的形状

（1）输出层的每个节点，通过D条权边与输入节点相连

$$\mathbf{W}_{ij} = [w_{ij0}, w_{ij1}, \ldots w_{ijD}]$$

换句话说：输出层的每个节点用一个D维矢量$\mathbf{W}_{ij}$来表征

（2）经过训练学习后

输出层的各个节点之间，按照距离远近具有一定的关联

（3）训练目的：学习一组权重 W, 可以将输入数据映射到输出层的节点上

高位空间距离较近的点，映射到输出层后距离也较近

比如 都映射到（i，j）上，或者映射到（i，j）和（i，j+1）上

模型训练过程:

(1) 准备训练数据 datas： N×D　　N为训练样本数量

通常需要进行正则化

$$datas=\frac{datas-mean(datas)}{std(datas)}$$

(2) 确定参数： X, Y　　$X=Y=\sqrt{5\sqrt{N}}$

(3) 权重初始化： **W：X×Y×D**

(4) 迭代训练

读取一个样本点 x： [D]

计算 x 和 X*Y个顶点的距离

(竞争)

找到距离最近的点 (i,j)作为**激活点**
设其权重为1

(自组织)

根据距离计算其他点的权重
得到：
g： X×Y　　(i,j)处值最大
以其为中心
越远越小

类似于寻找
聚类中心的
过程
令$\mathbf{W}_{ij}$ **接近**x

更新：

$$\mathbf{W}=\mathbf{W}+\eta\mathbf{g}\bullet(x-\mathbf{W})$$

具体细节：

权重初始化 W：[X,Y,D]

（1） 随机初始化 然后正则化 $\mathbf{W}=\frac{\mathbf{W}}{\|\mathbf{W}\|}$

（2）从训练数据中随机挑选 X*Y 个

（3）对训练数据进行PCA

取特征值最大的两个特征矢量 M： D×2

作为基向量进行映射

学习率的更新： $\eta=\frac{\eta_0}{1+\frac{t}{\max_{step}/2}}$

η随着迭代次数的增加，应当越来越小

获取激活点时的距离计算方法：

欧式距离 $dis=\|x-y\|$ → 二范数

计算输出层节点的权重：g

假设激活点为（c_x，c_y）

$g(i,j)=e^{-\frac{(c_x-i)^2}{2\delta^2}}e^{-\frac{(c_y-j)^2}{2\delta^2}}$ 高斯的方法

$g(i,j)=\begin{cases}1 & c_x-\delta\le i\le c_x+\delta \text{且} c_y-\delta\le j\le c_y+\delta\\ 0 & \text{其 他}\end{cases}$

硬阈值的方法

代码实现：

```
def train_SOM(X,
              Y,
              N_epoch,
              datas,
              init_lr = 0.5,
              sigma =0.5,
              dis_fun=euclidean_distance,
              neighborhood_fun=gaussion_neighborhood,
              init_weight_fun = None,
              seed =10):

    # 获取输入特征的维度
    N,D = np.shape(datas)

    # 训练的步数
    N_steps = N_epoch*N

    # 对权重进行初始化
    rng   =  np.random.RandomState(seed)
    if init_weight_fun is None:
        weights = rng.rand(X, Y, D)*2-1
        weights /= np.linalg.norm(weights, axis=-1, keepdims=True)
    else:
        weights = init_weight_fun(X,Y,datas)
```

PCA 初始化

```
def weights_PCA(X,Y,data):

    N,D = np.shape(data)
    weights = np.zeros([X,Y,D])

    pc_length, pc = np.linalg.eig(np.cov(np.transpose(data)))
    pc_order = np.argsort(-pc_length)
    for i, c1 in enumerate(np.linspace(-1, 1, X)):
        for j, c2 in enumerate(np.linspace(-1, 1, Y)):
            weights[i, j] = c1*pc[pc_order[0]] + c2*pc[pc_order[1]]
    return weights
```

```
for n_epoch in range(N_epoch):
    print("Epoch %d"%(n_epoch+1))
    # 打乱次序
    index = rng.permutation(np.arange(N))
    for n_step,_id in enumerate(index):

        # 取一个样本
        x = datas[_id]

        # 计算learning rate(eta)
        t = N*n_epoch+n_step
        eta = get_learning_rate(init_lr,t,N_steps)

        # 计算样本距离每个顶点的距离,并获得激活点的位置
        winner = get_winner_index(x,weights,dis_fun)

        # 根据激活点的位置计算临近点的权重
        new_sigma = get_learning_rate(sigma,t,N_steps)
        g = neighborhood_fun(X,Y,winner,new_sigma)
        g = g*eta

        # 进行权重的更新
        weights = weights + np.expand_dims(g,-1)*(x-weights)

    # 打印量化误差
    print("quantization_error= %.4f"%(get_quantization_error(datas,weights)))

return weights
```

```
# 计算学习率
def get_learning_rate(lr,t,max_steps):
    return lr / (1+t/(max_steps/2))
```

```
# 获取激活节点的位置
def get_winner_index(x,w,dis_fun=euclidean_distance):
    # 计算输入样本和各个节点的距离
    dis = dis_fun(x,w)

    # 找到距离最小的位置
    index = np.where(dis==np.min(dis))
    return (index[0][0],index[1][0])
```

```
# 计算欧式距离
def euclidean_distance(x, w):
    dis = np.expand_dims(x,axis=(0,1))-w
    return np.linalg.norm(dis, axis=-1)
```

```
for n_epoch in range(N_epoch):
    print("Epoch %d"%(n_epoch+1))
    # 打乱次序
    index = rng.permutation(np.arange(N))
    for n_step,_id in enumerate(index):

        # 取一个样本
        x = datas[_id]

        # 计算learning rate(eta)
        t = N*n_epoch+n_step
        eta = get_learning_rate(init_lr,t,N_steps)

        # 计算样本距离每个顶点的距离,并获得激活点的位置
        winner = get_winner_index(x,weights,dis_fun)

        # 根据激活点的位置计算临近点的权重
        new_sigma = get_learning_rate(sigma,t,N_steps)
        g = neighborhood_fun(X,Y,winner,new_sigma)
        g = g*eta

        # 进行权重的更新
        weights = weights + np.expand_dims(g,-1)*(x-weights)

    # 打印量化误差
    print("quantization_error= %.4f"%(get_quantization_error(datas,weights)))

return weights
```

```
# 利用高斯距离法计算临近点的权重
# X,Y 模板大小， c 中心点的位置， sigma 影响半径
def gaussion_neighborhood(X,Y,c,sigma):
    xx,yy = np.meshgrid(np.arange(X),np.arange(Y))
    d = 2*sigma*sigma
    ax = np.exp(-np.power(xx-xx.T[c], 2)/d)
    ay = np.exp(-np.power(yy-yy.T[c], 2)/d)
    return (ax * ay).T
```

$$\mathbf{W}=\mathbf{W}+\eta\mathbf{g}\bullet(x-\mathbf{W})$$

计算每个样本点和映射点之间的平均距离

```
# 计算量化误差
def get_quantization_error(datas,weights):
    w_x, w_y = zip(*[get_winner_index(d,weights) for d in datas])
    error = datas - weights[w_x,w_y]
    error = np.linalg.norm(error, axis=-1)
    return np.mean(error)
```

测试

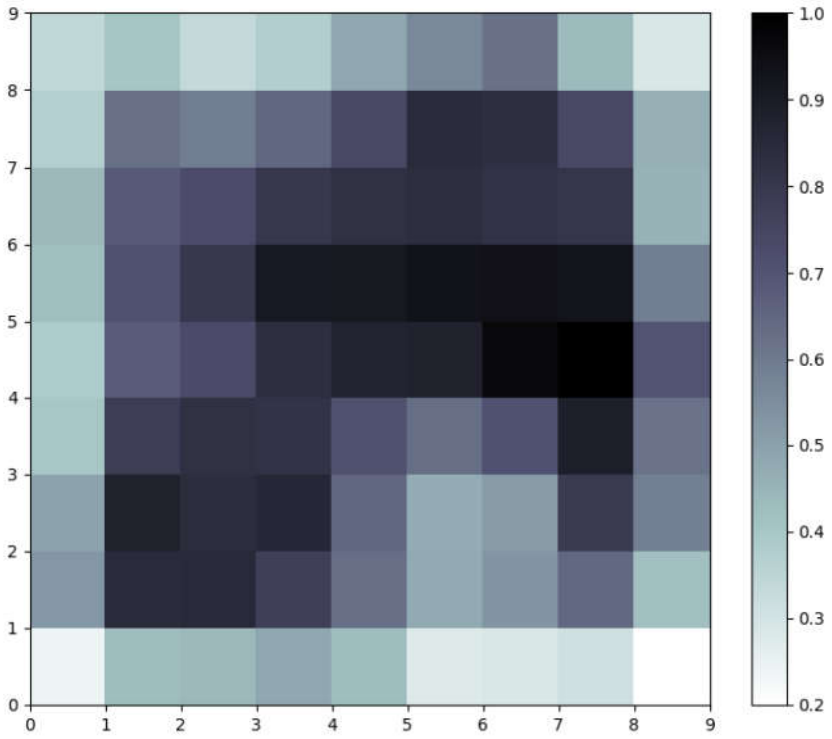
```
if __name__ == "__main__":
    # seed 数据展示
    columns=['area', 'perimeter', 'compactness', 'length_kernel', 'width_kernel',
                    'asymmetry_coefficient', 'length_kernel_groove', 'target']
    data = pd.read_csv('seeds_dataset.txt',
                       names=columns,
                      sep='\t+', engine='python')
    labs = data['target'].values
    label_names = {1:'Kama', 2:'Rosa', 3:'Canadian'}
    datas = data[data.columns[:-1]].values
    N,D = np.shape(datas)
    print(N,D)

    # 对训练数据进行正则化处理
    datas = feature_normalization(datas)

    # SOM的训练
    weights = train_SOM(X=9,Y=9,N_epoch=2,datas=datas,sigma=1.5,init_weight_fun=weights_PCA)

    # 获取UMAP
    UM = get_U_Matrix(weights)

    plt.figure(figsize=(9, 9))
    plt.pcolor(UM.T, cmap='bone_r')  # plotting the distance map as background
    plt.colorbar()
```

测试数据有7列

1	15.26	14.84	0.871	5.763	3.312	2.221	5.22	1
2	14.88	14.57	0.8811	5.554	3.333	1.018	4.956	1
3	14.29	14.09	0.905	5.291	3.337	2.699	4.825	1
4	13.84	13.94	0.8955	5.324	3.379	2.259	4.805	1
5	16.14	14.99	0.9034	5.658	3.562	1.355	5.175	1
6	14.38	14.21	0.8951	5.386	3.312	2.462	4.956	1
7	14.69	14.49	0.8799	5.563	3.259	3.586	5.219	1
8	14.11	14.1	0.8911	5.42	3.302	2.7	5	1

```
def get_U_Matrix(weights):

    X,Y,D = np.shape(weights)
    um = np.nan * np.zeros((X,Y,8)) # 8邻域

    ii = [0, -1, -1, -1, 0, 1, 1, 1]
    jj = [0, -1, -1, -1, 0, 1, 1, 1]

    for x in range(X):
        for y in range(Y):
            w_2 = weights[x, y]

            for k, (i, j) in enumerate(zip(ii, jj)):
                if (x+i >= 0 and x+i < X and y+j >= 0 and y+j < Y):
                    w_1 = weights[x+i, y+j]
                    um[x, y, k] = np.linalg.norm(w_1-w_2)


    um = np.nansum(um, axis=2)
    return um/um.max()
```

计算每个输出节点和周边节点之间的关系，用当前节点和周围8个临近点的欧式距离之和来评估



```
    markers = ['o', 's', 'D']
    colors = ['C0', 'C1', 'C2']

    for i in range(N):
        x = datas[i]
        w = get_winner_index(x,weights)
        i_lab = labs[i]-1

        plt.plot(w[0]+.5, w[1]+.5, markers[i_lab], markerfacecolor='None',
             markeredgecolor=colors[i_lab], markersize=12, markeredgewidth=2)

    plt.show()
```

```
PS D:\工作相关\我设计的课程\py
210 7
Epoch 1
quantization_error= 0.6511
Epoch 2
quantization_error= 0.5679
Epoch 3
quantization_error= 0.5153
```