模式识别导论第三次作业

翁晨阳

简答与描述题

1. 请对反向传播算法的训练步骤进行总结：结合三层网络给出不超过三个有关权重更新的公式，并用文字描述所述公式的含义；指出哪些因素会对网络的性能产生影响。

训练步骤：

1. 首先通过前向传播得到输出，再计算输出和目标之间的误差
2. 通过误差首先修正输出层至最后一个隐含层的连接权重
3. 修正最后一个隐含层至倒数第二个隐含层的连接权重
4. 从后往前依次修正隐含层的连接权重
5. 修正第一隐含层至输入层的连接权重

公式：

1.

$$
\begin{aligned}
E(\mathbf{w})^k = J(\mathbf{w})^k &= \frac{1}{2}\sum_j (t_j^k - z_j^k)^2 \\
&= \frac{1}{2}\sum_j \left\{ t_j^k - f\left(\sum_h \mathbf{w}_{hj} f\left(\sum_i \mathbf{w}_{ih}\mathbf{x}_i^k \right) \right) \right\}^2
\end{aligned} \tag{1.1}
$$

式（1.1）是单个样本的误差函数，使用的误差函数是均方误差，第一行表示的是样本标签和输出值的误差平方和，第二行表示在三层网络的情况下，通过输入层计算的误差函数，其中$\mathbf{w}_{hj}, \mathbf{w}_{ih}$为连接权重，$f$为激励函数$\mathbf{x}_i^k$为第k个样本的输入。

2.

$$
\begin{aligned}
\Delta \mathbf{w}_{hj} = -\eta \frac{\partial E}{\partial \mathbf{w}_{hj}} &= -\eta \sum_k \frac{\partial E}{\partial \mathrm{net}_j^k} \frac{\partial \mathrm{net}_j^k}{\partial \mathbf{w}_{hj}} \\
&= \eta \sum_k (t_j^k - z_j^k) f'(\mathrm{net}_j^k) y_h^k \\
&= \eta \sum_k \delta_j^k y_h^k \\
\delta_j^k = \frac{-\partial E}{\partial \mathrm{net}_j^k} &= f'(\mathrm{net}_j^k)\Delta_j^k
\end{aligned} \tag{1.2}
$$

式（1.2）是批量情况下隐含层到输出层的连接权重调节量，其中η是学习率，调节量可以理解为输出层的导数和误差的积再乘上隐含层的值对样本求和。

$$
\begin{aligned}
\Delta \mathbf{w}_{ih} = -\eta \frac{\partial E}{\partial \mathbf{w}_{ih}} &= -\eta \sum_{k,j} \frac{\partial E}{\partial \mathrm{z}_j^k} \frac{\partial \mathrm{z}_j^k}{\partial \mathbf{w}_{ih}} \\
&= \eta \sum_{k,j} (t_j^k - z_j^k) \frac{\partial \mathrm{z}_j^k}{\partial \mathbf{w}_{ih}} \\
&= \eta \sum_{k,j} (t_j^k - z_j^k) \frac{\partial \mathrm{z}_j^k}{\partial \mathrm{net}_j^k} \frac{\partial \mathrm{net}_j^k}{\partial \mathbf{w}_{ih}} \\
&= \eta \sum_{k,j} (t_j^k - z_j^k) f'(\mathrm{net}_j^k) \frac{\partial \mathrm{net}_j^k}{\partial \mathrm{y}_h^k} \frac{\partial \mathrm{y}_h^k}{\partial \mathbf{w}_{ih}} \\
&= \eta \sum (t_j^k - z_j^k) f'(\mathrm{net}_j^k) \mathbf{w}_{hj} \frac{\partial \mathrm{y}_h^k}{\partial \mathbf{w}_{ih}}
\end{aligned}
$$

$$
\begin{aligned}
&\Delta \mathbf{w}_{ih} \\
&= \eta \sum_{k,j} (t_j^k - z_j^k) f'(\mathrm{net}_j^k) \mathbf{w}_{hj} \frac{\partial \mathrm{y}_h^k}{\partial \mathrm{net}_h^k} \frac{\partial \mathrm{net}_h^k}{\partial \mathbf{w}_{ih}} \\
&= \eta \sum_{k,j} (t_j^k - z_j^k) f'(\mathrm{net}_j^k) \mathbf{w}_{hj} f'(\mathrm{net}_h^k) \mathbf{x}_i^k \\
&= \eta \sum_{k,j} \delta_j^k \mathbf{w}_{hj} f'(\mathrm{net}_h^k) \mathbf{x}_i^k \\
&= \eta \sum_{k} \left(f'(\mathrm{net}_h^k) \sum_j \delta_j^k \mathbf{w}_{hj} \right) \mathbf{x}_i^k \\
&= \eta \sum_{k} \delta_h^k \mathbf{x}_i^k
\end{aligned} \tag{1.3}
$$

$$
\delta_j^k = f'(\mathrm{net}_j^k)(t_j^k - z_j^k) = f'(\mathrm{net}_j^k) \Delta_j^k
$$

$$
\delta_h^k = \frac{-\partial E}{\partial \mathrm{net}_h^k} = f'(\mathrm{net}_h^k) \sum_j \mathbf{w}_{hj} \delta_j^k = f'(\mathrm{net}_h^k) \Delta_h^k, \Delta_h^k = \sum_j \mathbf{w}_{hj} \delta_j^k
$$

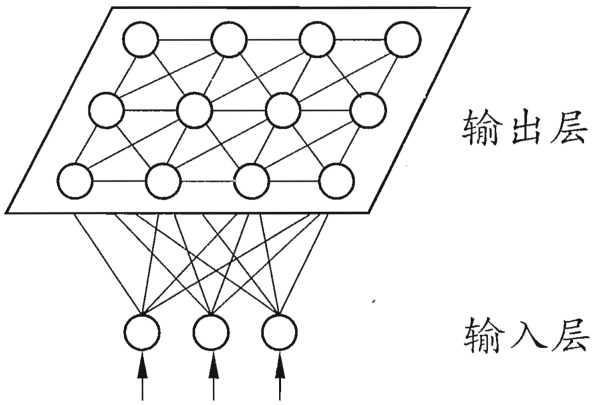
3.

式（1.3）是批量情况下输入层到隐含层的连接权重调节量，调节量可以理解为上一层激励函数的导数乘和上一层误差的积乘以这一层神经元的值对训练样本求和。

性能影响：隐含层的层数、各隐含层神经元的数量、学习率、激励函数的选择、损失函数的选择、训练样本的质量。

2. 请描述自组织映射网络的构造原理，给出自组织算法的计算步骤（即网络训练）

构造原理：



自组织映射(Self-organizing map, SOM)是一种**无监督**的人工神经网络。不同于一般神经网络基于损失函数的反向传递来训练，它运用**竞争学习**(competitive learning)策略,依靠神经元之间互相竞争逐步优化网络。且使用近邻关系函数(neighborhood function)来维持输入空间的拓扑结构。
SOM的网络结构有2层：输入层、输出层(也叫竞争层)。输入层神经元的数量是由输入向量的维度决定的，一个神经元对应一个特征。竞争层SOM神经元的数量决定了最终模型的粒度与规模；这对最终模型的准确性与泛化能力影响很大。
SOM神经元对其邻近神经元的影响是由近及远的，由兴奋逐渐转变为抑制。因此在学习算法中，不仅获胜神经元本身要调整权向量，它周围的神经元在其影响下也要不同程度地调整权重。邻域大小可随时间增长而减小。
获胜神经元为中心设定一个邻域半径，该半径圈定的范围称为优胜邻域。在SOM网学习算法中，优胜邻域内的所有神经元均按其距离获胜神经元的远近不同程度地调整权重。

计算步骤：

1. 随机初始化连接权重
2. 随机取出一个样本$\mathbf{x}_i$作为输入
3.

1. 遍历竞争层中每一个节点：计算Xi与节点之间的相似度

$$d_j = \sqrt{\sum_{i=1}^{d}(\mathbf{x}_i - \mathbf{w}_{ij})^2} \tag{2.1}$$

2. 选取距离最小的节点作为优胜节点

4. 根据邻域半径σ(sigma)确定**优胜邻域**将包含的节点；并通过邻域函数计算它们各自更新的幅度(基本思想是：越靠近优胜节点，更新幅度越大；越远离优胜节点，更新幅度越小)

5. 更新优胜邻域内节点的权重：

$$\begin{aligned} \Delta\mathbf{w}_{ij} &= \eta h(j, j^*)(\mathbf{x}_i - \mathbf{w}_{ij}) \\ \mathbf{w}_{ij}(t+1) &= \mathbf{w}_{ij}(t) + \Delta\mathbf{w}_{ij} \\ h(j, j^*) &= \exp\left(-\|j - j^*\|^2/\sigma^2\right) \end{aligned} \tag{2.2}$$

6. 完成一轮迭代(迭代次数+1)，返回第二步，直到满足设定的迭代条件

编程题

1.

code:

生成BP网络的类：

```
class BPNet:

    def __init__(self, input_size, hidden_size, output_size, weight_init_std = 0.01):
        # 初始化权重
        self.params = {}
        l = len(hidden_size)
        for i in range(l-1):
            no = i + 1
            w = 'W' + str(no)
            b = 'b' + str(no)
            self.params[w] = weight_init_std * np.random.randn(hidden_size[i],hidden_size[i+1])   #123
            self.params[b] = np.zeros(hidden_size[i+1])

        # 生成层
        self.layers = OrderedDict()
        for i in range(l-1):
            affine = 'Affine' + str(i + 1)
            #sgm = 'Sigmoid' + str(i + 1)
            th = 'Tanh' + str(i + 1)
            w = 'W' + str(i + 1)
            b = 'b' + str(i + 1)
            self.layers[affine] = Affine(self.params[w], self.params[b])

            if i != l-2:
                #self.layers[sgm] = Sigmoid()
                self.layers[th] = Tanh()

        self.lastLayer = Sigmoid()


```

```
    def predict(self, x):
        for layer in self.layers.values():
            x = layer.forward(x)

        return x

    def loss(self, x, t):
        y = self.predict(x)
        return self.lastLayer.forward(y, t)

    def accuracy(self, x, t):
        y = self.predict(x)
        y = np.argmax(y, axis=1)
        if t.ndim != 1 : t = np.argmax(t, axis=1)

        accuracy = np.sum(y == t) / float(x.shape[0])
        return accuracy

    def gradient(self, x, t):
        # forward
        self.loss(x, t)

        # backward
        dout = 1
        dout = self.lastLayer.backward(dout)

        layers = list(self.layers.values())
        layers.reverse()
        for layer in layers:
            dout = layer.backward(dout)

        # 设定
        grads = {}
        grads['W1'], grads['b1'] = self.layers['Affine1'].dW, self.layers['Affine1'].db
        grads['W2'], grads['b2'] = self.layers['Affine2'].dW, self.layers['Affine2'].db

        return grads
```

构造全连接层的类：

```
class Affine:
    def __init__(self, W, b):
        self.W =W
        self.b = b

        self.x = None
        self.original_x_shape = None
        # 权重和偏置参数的导数
        self.dW = None
        self.db = None

    def forward(self, x):
        # 对应张量
        self.original_x_shape = x.shape
        x = x.reshape(x.shape[0], -1)
```

```
        self.x = x

        out = np.dot(self.x, self.W) + self.b

        return out

    def backward(self, dout):
        dx = np.dot(dout, self.W.T)
        self.dW = np.dot(self.x.T, dout)
        self.db = np.sum(dout, axis=0)

        dx = dx.reshape(*self.original_x_shape)  # 还原输入数据的形状（对应张量）
        return dx
```

构造双曲正切的激励函数类：

```
class Tanh:

    def __init__(self):
        self.out = None

    def forward(self, x):
        out = tanh(x)
        self.out = out
        return out

    def backward(self, dout):
        dx = dout * (2.0 - self.out) * self.out

        return dx
```

构造sigmoid层的类：

```
class Sigmoid:

    def __init__(self):
        self.out = None
        self.loss = None
        self.y = None # 网络的输出
        self.t = None # 监督数据

    def forward(self, x, t):
        self.t = t
        self.y = sigmoid(x)
        out = sigmoid(x)
        self.out = out
        self.loss = cross_entropy_error(self.y, self.t)

        return self.loss
```

一些用到的函数：

```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def tanh(x):
```

```
    return 2 / (1 + np.exp(-2*x))

def cross_entropy_error(y, t):
    if y.ndim == 1:
        t = t.reshape(1, t.size)
        y = y.reshape(1, y.size)

    # 监督数据是one-hot-vector的情况下，转换为正确解标签的索引
    if t.size == y.size:
        batch_size = y.shape[0]
        return 0.5 * np.sum((y-t)**2)/batch_size
        #t = t.argmax(axis=1)

    batch_size = y.shape[0]
    return -np.sum(np.log(y[np.arange(batch_size), t] + 1e-7)) / batch_size
```

训练用到的代码：

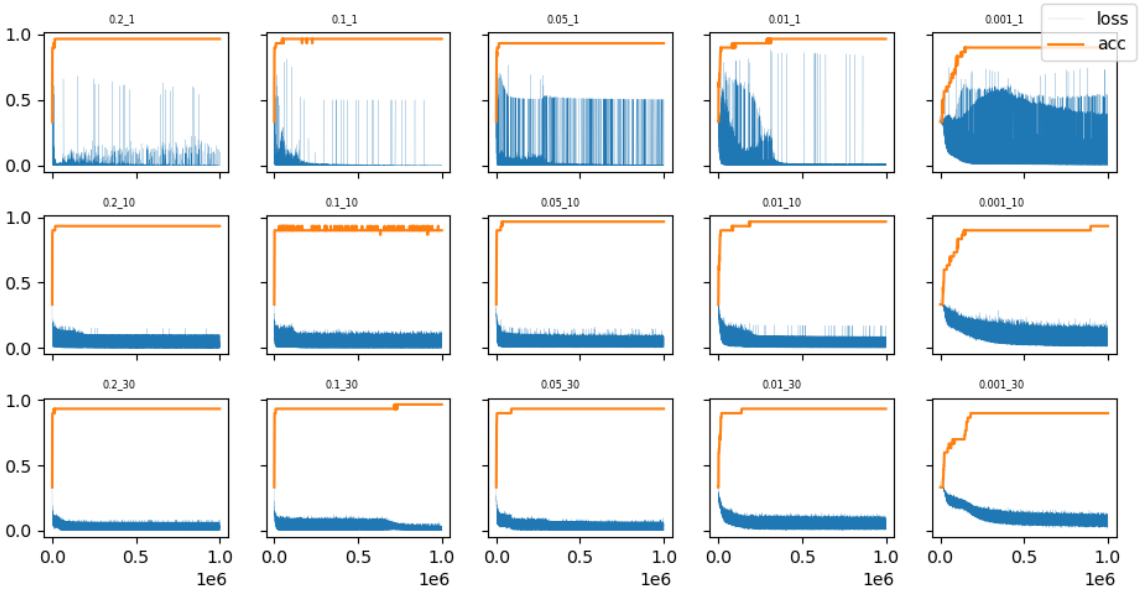
```
x_train = np.array([[1.58, 2.32, -5.8], [0.67, 1.58, -4.78], [1.04, 1.01, -3.63],
                    [-1.49, 2.18, -3.39], [-0.41, 1.21, -4.73], [1.39, 3.16, 2.87],
                    [1.20, 1.40, -1.89], [-0.92, 1.44, -3.22], [0.45, 1.33, -4.38],
                    [-0.76, 0.84, -1.96],
                    [0.21, 0.03, -2.21], [0.37, 0.28, -1.8], [0.18, 1.22, 0.16],
                    [-0.24, 0.93, -1.01], [-1.18, 0.39, -0.39], [0.74, 0.96, -1.16],
                    [-0.38, 1.94, -0.48], [0.02, 0.72, -0.17], [0.44, 1.31, -0.14],
                    [0.46, 1.49, 0.68],
                    [-1.54, 1.17, 0.64], [5.41, 3.45, -1.33], [1.55, 0.99, 2.69],
                    [1.86, 3.19, 1.51], [1.68, 1.79, -0.87], [3.51, -0.22, -1.39],
                    [1.40, -0.44, -0.92], [0.44, 0.83, 1.97], [0.25, 0.68, -0.99],
                    [0.66, -0.45, 0.08]])
t_train = np.array([[1,0,0],[1,0,0],[1,0,0],[1,0,0],[1,0,0],[1,0,0],[1,0,0],[1,0,0],[1,0,0],[1,0,0],
                     [0,1,0],[0,1,0],[0,1,0],[0,1,0],[0,1,0],[0,1,0],[0,1,0],[0,1,0],[0,1,0],[0,1,0],
                     [0,0,1],[0,0,1],[0,0,1],[0,0,1],[0,0,1],[0,0,1],[0,0,1],[0,0,1],[0,0,1],[0,0,1]])
print(x_train.shape,t_train.shape)

network = BPNet(input_size=3, hidden_size=[3,12,3], output_size=3)

iters_num = 1000000
train_size = x_train.shape[0]
batch_size = 10
learning_rate = 0.01

train_loss_list = []
train_acc_list = []
test_acc_list = []
```

```
28
29  iter_per_epoch = max(train_size / batch_size, 1)
30
31  for i in range(iters_num):
32      batch_mask = np.random.choice(train_size, batch_size)
33      x_batch = x_train[batch_mask]
34      t_batch = t_train[batch_mask]
35
36      # 梯度
37      grad = network.gradient(x_batch, t_batch)
38
39      # 更新
40      for key in ('W1', 'b1', 'W2', 'b2'):
41          network.params[key] -= learning_rate * grad[key]
42
43      loss = network.loss(x_batch, t_batch)
44      train_loss_list.append(loss)
45
46      if i % iter_per_epoch == 0:
47          train_acc = network.accuracy(x_train, t_train)
48          train_acc_list.append(train_acc)
49          print(i, train_acc, loss)
```

注：通过控制batch_size的大小可以用来控制是单样本训练还是批样本训练

通过BPNet的构造函数的hidden_size可以控制隐含层的层数和每一层的大小，learning_rate为学习步长。

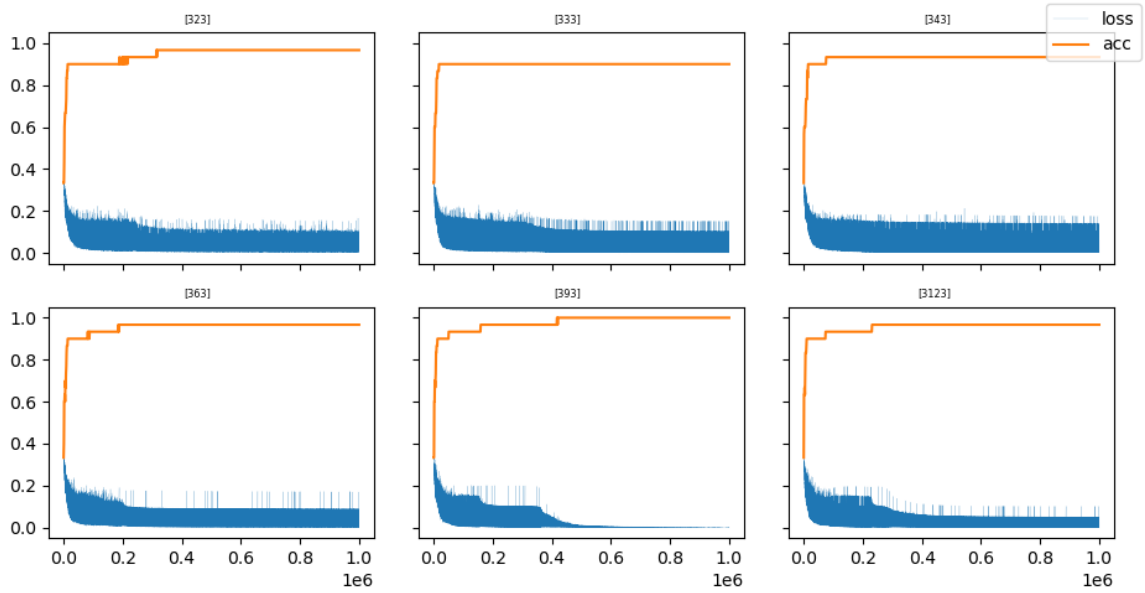
2.实验分析



上图是在3*6*3的网络情况下学习率和批大小对训练情况的影响，其中黄色线条表示准确率变化，蓝色线条表示loss值变化。图表水平方向的排列表示学习率的不同，从左到右依次表示学习率0.2、0.1、0.05、0.01、0.001；图表垂直方向的排列表示批大小的不同，从上到下的批大小依次是1、10、30。

分析图表可以得知更新步长越大loss下降的越快，但是最后收敛的时候会不稳定，更新步长小的时候loss下降较慢，可以明显发现，loss值有时候会被困在一个地方较长时间。因为更新步长大的时候每一次连接权重更新的幅度也大，所以在网络训练初期loss能够快速下降，但是到了训练后期，大步长也容易使loss冲出最优解范围，导致loss值震荡；同样的更新步长小的时候每一次连接权重的更新幅度也小，loss下降的速度也更慢，但后期出现震荡的可能也小。

观察batch_size的变化对训练的影响：发现在目前的情况下batch_size越小越好，当单样本的时候，训练极其不稳定，主要原因是训练集样本太小。当训练集样本足够的时候，batch_size大小就不能过大了，否则会影响迭代的时间。



上图是不同网络结构下的loss值和acc值随迭代次数的影响，网络结构从左到右从上到下依次是【3*2*3】、【3*3*3】、【3*4*3】、【3*6*3】、【3*9*3】、【3*12*3】，下图是【3*72*3】情况下的loss值和acc值变化曲线，以上模型均在学习率为0.01，批大小为10的情况下训练，可以大致看出随着网络结构的复杂化loss值将进一步减小，准确率也会变高，但是需要指出的是，本次训练用到的训练集的样本大小仅有30个，所以很明显在网络复杂的情况下很容易就会产生过拟合的现象。