A simple neural network

By Yubei Xiao

Problem 1

题意分析

第一题的题意其实就是让我们写一个指定参数的网络还有前向传播（forward）、测试逻辑（inference）。

输入输入空间中的数据点（x_1, x_2）预测得到它们的结果，根据结果的符号将输入空间（$-5 \leq x_1, x_2 \leq +5$）划分成两部分。

数学分析

由题可知：

第一题的网络是一个 2-2-1 的网络，且隐藏单元和输出单元的激活函数均为$f(net) = \frac{2a}{1+e^{-b \cdot net}} - a$，其中$a = 1.716, b = \frac{2}{3}$。

则从输入$\boldsymbol{x} =（x_1, x_2）$到隐藏神经元的神经网络方程为：

$$f(\begin{bmatrix} w_{0,1}^{[1]} & w_{1,1}^{[1]} & w_{2,1}^{[1]} \\ w_{0,2}^{[1]} & w_{1,2}^{[1]} & w_{2,2}^{[1]} \end{bmatrix} \begin{bmatrix} 1 \\ x_1 \\ x_2 \end{bmatrix}) = \begin{bmatrix} h_1 \\ h_2 \end{bmatrix}$$

从隐藏神经元的输出$\boldsymbol{h} = (h_1, h_2)$到输出神经元的神经网络方程为：

$$f(\begin{bmatrix} w_0^{[2]} & w_1^{[2]} & w_2^{[2]} \end{bmatrix} \begin{bmatrix} 1 \\ h_1 \\ h_2 \end{bmatrix}) = o$$

那么对于第(a)问，参数矩阵为：

$$\begin{bmatrix} w_{0,1}^{[1]} & w_{1,1}^{[1]} & w_{2,1}^{[1]} \\ w_{0,2}^{[1]} & w_{1,2}^{[1]} & w_{2,2}^{[1]} \end{bmatrix} = \begin{bmatrix} 0.5 & 0.3 & -0.1 \\ -0.5 & -0.4 & 1.0 \end{bmatrix}$$

$$\begin{bmatrix} w_0^{[2]} & w_1^{[2]} & w_2^{[2]} \end{bmatrix} = \begin{bmatrix} 1.0 & -2.0 & 0.5 \end{bmatrix}$$

对于第(b)问，参数矩阵为：

$$\begin{bmatrix} w_{0,1}^{[1]} & w_{1,1}^{[1]} & w_{2,1}^{[1]} \\ w_{0,2}^{[1]} & w_{1,2}^{[1]} & w_{2,2}^{[1]} \end{bmatrix} = \begin{bmatrix} -1.0 & -0.5 & 1.5 \\ 1.0 & 1.5 & -0.5 \end{bmatrix}$$

$$\begin{bmatrix} w_0^{[2]} & w_1^{[2]} & w_2^{[2]} \end{bmatrix} = \begin{bmatrix} 0.5 & -1.0 & 1.0 \end{bmatrix}$$

那么其实前向传播就是将输入从网络输入单元输入经过网络每层的计算以及向前传播之后得到最终输出的一个过程。其实我们就可以直接用矩阵运算从输入得到最后的结果：

$\boldsymbol{o} = f((w^{[2]})^T f((w^{[1]})^T \boldsymbol{x}))$，其中$w^{[1]}$ 为从$\boldsymbol{x}$到隐藏神经元的权重矩阵，$w^{[2]}$ 为从隐藏神经元到输出$\boldsymbol{o}$的权重矩阵，f是激活函数。

因此，当网络用数学分析完毕之后，代码就能够很简单的实现了。

代码实现

根据上面的题目分析和数学分析，整个网络中的参数我用矩阵方式存储:

```
hiddenWeights = [[0.5, 0.3, -0.1], [-0.5, -0.4, 1.0]]
outputWeights = [1.0, -2.0, 0.5]
```

前向传播其实就是矩阵运算，下面就是Network类中的前向传播函数forward：

```
    def forward(self, data):
        # set input
        self.setInput(data)
        # calculate output from input
        self.hiddenOutputBeforeActivationFunc = np.dot(self.hiddenWeights,
self.input)
        self.hiddenOutput =
self.sigmoid(self.hiddenOutputBeforeActivationFunc)
        self.hiddenOutput = np.insert(self.hiddenOutput, 0, 1)
        self.outputBeforeActivationFunc = np.dot(self.outputWeights,
self.hiddenOutput)
        self.output = self.sigmoid(self.outputBeforeActivationFunc)
        return self.output
```

传入的data就是输入数据$\boldsymbol{x} = （x_1, x_2）$，因为运算的时候需要添加第一维为1（可以参见数学表示中），因此通过self.setInput给$\boldsymbol{x}$加一维成$\boldsymbol{x} = （1, x_1, x_2）$：

```
    def setInput(self, data):
        self.input = np.insert(data, 0, 1)
        return
```

之后将输入self.input与隐藏层权重矩阵self.hiddenWeights进行矩阵乘法运算得到self.hiddenOutputBeforeActivationFunc，再输入到激活函数self.sigmoid中运算：

```
    def sigmoid(self, out):
        a = 1.716
        b = 2.0/3.0
        return np.array((2*a/(1+np.exp(-b*out)))-a)
```

隐藏层输出self.hiddenOutput也加上第一维1，再类似地与输出层权重矩阵self.outputWeights进行矩阵乘法运算得到self.outputBeforeActivationFunc，再输入到激活函数self.sigmoid中运算最后得到网络的输出 self.output。

那么测试逻辑（inference）代码如下：

```
    def inference(self, data):
        output = self.forward(data)
        return np.sign(output)
```

inference就是先输入测试数据data让网络前向传播得到结果output，然后再用np.sign取output的符号输出。

那么网络inference部分实现了之后，就是生成数据集，下面代码就是在输入空间（$-5 \leq x_1, x_2 \leq +5$）中均匀地生成10000个点：

```
def createDatasetEvenly():
    dataset = []
    total = 100
    interval = 10.0 / total
    for i in range(total):
        x1 = -5.0 + i * interval
        for j in range(total):
            x2 = -5.0 + j * interval
            point = [x1, x2]
            dataset.append(point)
    return dataset
```

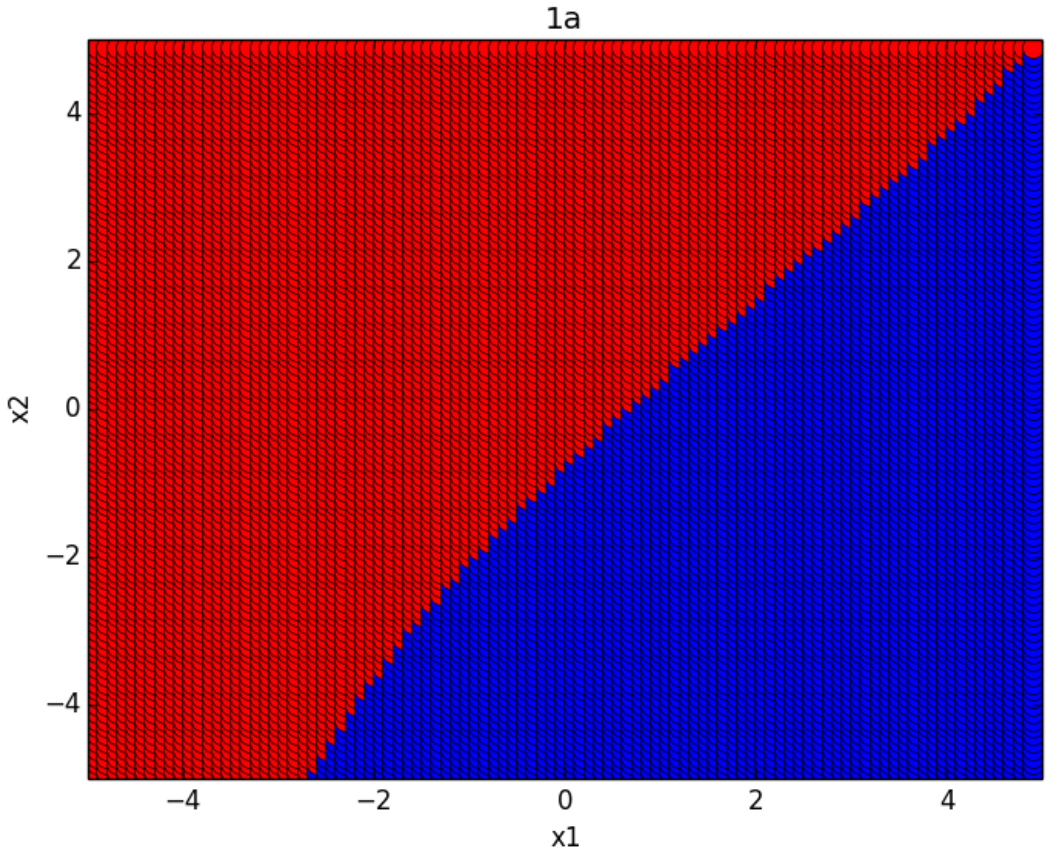
得到数据集之后，将数据输入到网络中inference：

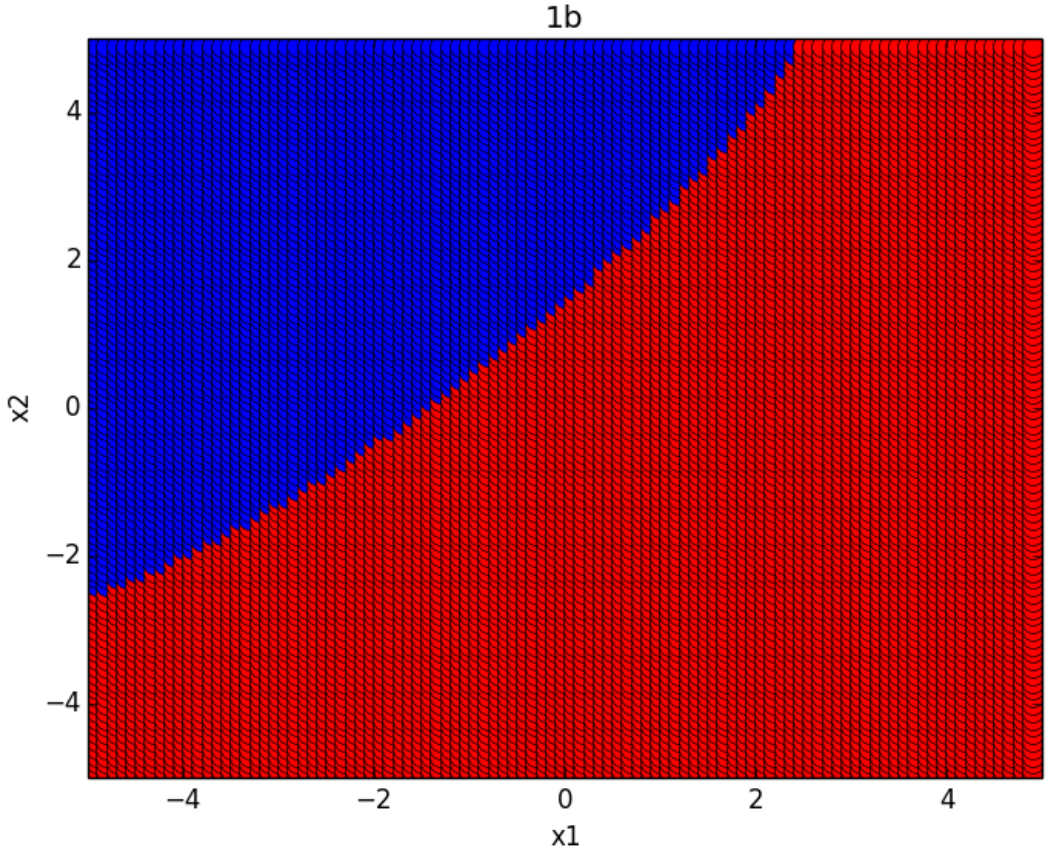
```
for data in dataset:
    result = net1a.inference(data)
    if result == -1:
        color = 'b'
    else:
        color = 'r'
    plt.plot(data[0], data[1], marker='o', color=color, markersize=10)
```

得到预测结果后再根据符号画图，从而就得到了最终结果。

实验结果

下面两张图分别是(a)问的结果和(b)问的结果，其中为了便于区分我用蓝红两色代替了题目中要求的黑白两色进行了划分，红色代表结果为正号（包括0），蓝色代表结果为负号。





Problem 2

题意分析

第二题的题意其实就是给定一个网络和初始化参数的方法，让我们用随机梯度下降的方式去实现网络的训练，并且把训练过程中每个epoch的训练损失打印出来，同时还需要分析比较一下两种不同初始化参数方法的不同点。

其实在第一题中已经实现了一个网络类、前向传播还有inference，都可以沿用到第二题，那么在第二题中其实主要就是要实现网络的训练，包括计算损失函数、反向传播（backpropagation）、随机梯度下降（stochastic gradient descent）还有初始化参数。

数学分析

由题可知：

第二题的网络是一个 3-1-1 的网络，且隐藏单元和输出单元的激活函数依旧是$f(net) = \frac{2a}{1+e^{-b \cdot net}} - a$，其中$a = 1.716, b = \frac{2}{3}$。

则从输入$\boldsymbol{x} = （ x_1, x_2, x_3 ）$到隐藏神经元的神经网络方程为：

$$f(\begin{bmatrix} w_0^{[1]} & w_1^{[1]} & w_2^{[1]} & w_3^{[1]} \end{bmatrix} \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix}) = h_1$$

从隐藏神经元的输出$\boldsymbol{h} = (h_1)$到输出神经元的神经网络方程为：

$$f(\begin{bmatrix} w_0^{[2]} & w_1^{[2]} \end{bmatrix} \begin{bmatrix} 1 \\ h_1 \end{bmatrix}) = o$$

那么在第(a)问中，是随机初始化参数矩阵：

$$\begin{bmatrix} w_0^{[1]} & w_1^{[1]} & w_2^{[1]} & w_3^{[1]} \end{bmatrix} = [random \quad random \quad random \quad random]$$

$$\begin{bmatrix} w_0^{[2]} & w_1^{[2]} \end{bmatrix} = [random \quad random]$$

对于第(b)问，由题可知参数矩阵为：

$$\begin{bmatrix} w_0^{[1]} & w_1^{[1]} & w_2^{[1]} & w_3^{[1]} \end{bmatrix} = [0.5 \quad 0.5 \quad 0.5 \quad 0.5]$$

$$\begin{bmatrix} w_0^{[2]} & w_1^{[2]} \end{bmatrix} = [-0.5 \quad -0.5]$$

那么对于前向传播，我们可以直接用矩阵运算出从输入到得到最后的结果：

$\boldsymbol{o} = f((w^{[2]})^T f((w^{[1]})^T \boldsymbol{x}))$，其中$w^{[1]}$ 为从$\boldsymbol{x}$到隐藏神经元的权重矩阵，$w^{[2]}$ 为从隐藏神经元到输出$\boldsymbol{o}$的权重矩阵，f是激活函数。

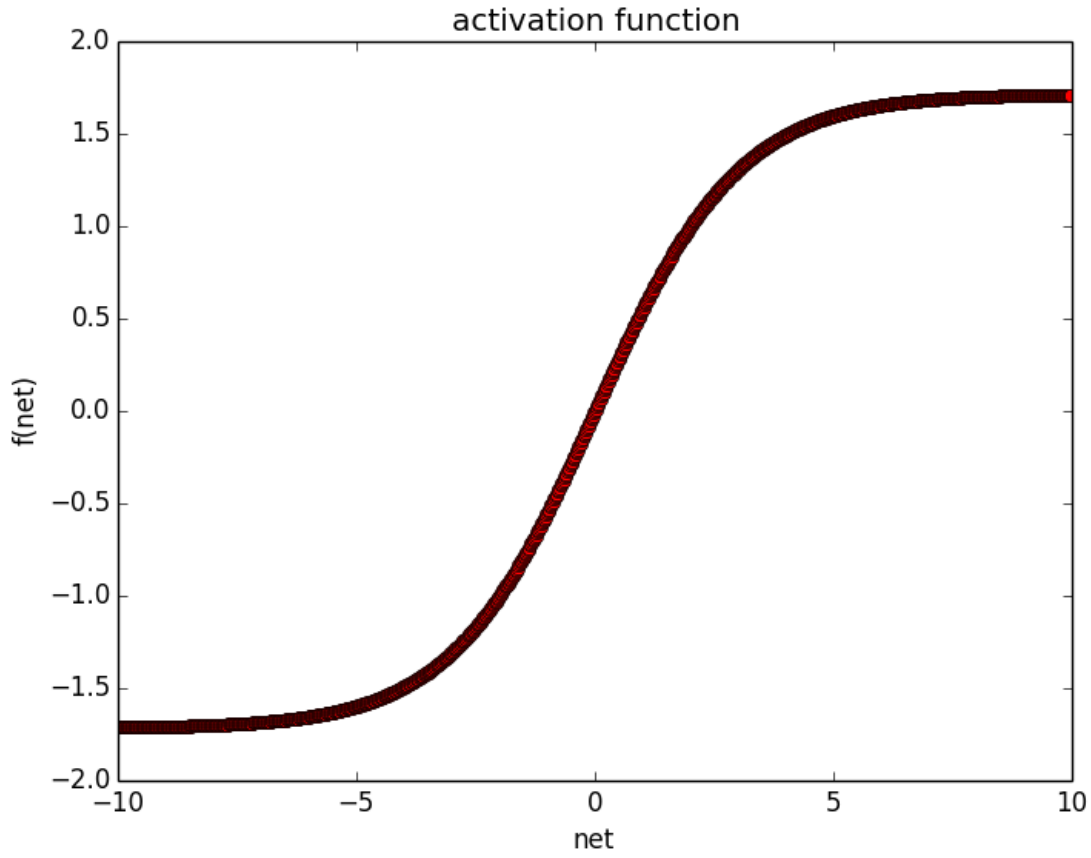
在得到前向传播的输出结果之后，就需要计算网络输出和label的损失函数值。我使用的就是均方误差函数(MSE)：

$l = \frac{1}{m} \sum_{i=1}^{m} (\boldsymbol{o}^{(i)} - \boldsymbol{y}^{(i)})^2$，其中上标$^{(i)}$表示的是第$i$个样本的对应的网络输出 $\boldsymbol{o}^{(i)}$ 或者真实标签(label) $\boldsymbol{y}^{(i)}$，m是样本总数。

不过在题中要求使用的是随机梯度下降方法，因此每次只需要随机选取一个样本计算损失、更新网络。因此损失函数可以简化为：

$l = (\boldsymbol{o} - \boldsymbol{y})^2$

对于其中的真实标签(label) $\boldsymbol{y}^{(i)}$ 我使用的是1.716去表示第一类(表格中的w_1类)，-1.716去表示第二类(表格中的w_2类)。因为题目中的激活函数$f(net) = \frac{2a}{1+e^{-b \cdot net}} - a$其实是sigmoid函数$f(net) = \frac{1}{1+e^{-net}}$放大2a倍再平移-a得到。而用sigmoid函数时一般取1和-1为两个类别的标签，因此对于题中激活函数，应该取2a*(-1,1)-a=(-1.716,1.716)为标签。同时我也用代码画了下激活函数的图，如下图，发现它上下限时趋近于(-1.716,1.716)的，因此真实标签用-1.716和1.716来表示是比较可行的。



那么接下来就是通过反向传播计算梯度：

$\because l = (\boldsymbol{o} - \boldsymbol{y})^2$

$\therefore$loss对网络输出$\boldsymbol{o}$的梯度为：

$\frac{\partial l}{\partial \boldsymbol{o}} = 2 * (\boldsymbol{o} - \boldsymbol{y})$

$\because \boldsymbol{o} = f((\boldsymbol{w}^{[2]})^T \boldsymbol{h})$

$\therefore$ $\boldsymbol{o}$对隐藏到输出的权重$\boldsymbol{w}^{[2]} = \begin{bmatrix} w_0^{[2]} & w_1^{[2]} \end{bmatrix}$的梯度为：

$\frac{\partial \boldsymbol{o}}{\partial \boldsymbol{w}^{[2]}} = f'((\boldsymbol{w}^{[2]})^T \boldsymbol{h}) * \boldsymbol{h}$

以及$\boldsymbol{o}$对隐藏层输出$\boldsymbol{h} = (h_1)$的梯度为：

$\frac{\partial \boldsymbol{o}}{\partial \boldsymbol{h}} = f'((\boldsymbol{w}^{[2]})^T \boldsymbol{h}) * \boldsymbol{w}_1^{[2]}$

$\because \boldsymbol{h} = f((w^{[1]})^T \boldsymbol{x})$

$\therefore$ $\boldsymbol{h}$对输入到隐藏层的权重$\boldsymbol{w}^{[1]} = \begin{bmatrix} w_0^{[1]} & w_1^{[1]} & w_2^{[1]} & w_3^{[1]} \end{bmatrix}$的梯度为：

$\frac{\partial \boldsymbol{h}}{\partial \boldsymbol{w}^{[1]}} = f'((\boldsymbol{w}^{[1]})^T \boldsymbol{x}) * \boldsymbol{x}$

在上式中的$f'(net)$即激活函数对内部net的求导，计算如下：

$\because f(net) = \frac{2a}{1+e^{-b \cdot net}} - a$

$\therefore f'(net) = \frac{2a}{1+e^{-x}}|_{dx} * b, where\ x = b \cdot net$

$\therefore f'(net) = 2ab * \frac{1}{1+e^{-x}} * (1 - \frac{1}{1+e^{-x}}), where\ x = b \cdot net$

令$k = \frac{1}{1+e^{-x}}$则：

$f'(net) = 2ab * k(1-k)$

那么通过上面的计算梯度已经全部求解出来，则每次训练就是随机一个样本进行梯度下降更新参数。

隐藏层到输出的权重矩阵梯度下降更新为：

$\boldsymbol{w}^{[2]} = \boldsymbol{w}^{[2]} - \eta \frac{\partial l}{\partial \boldsymbol{w}^{[2]}}$
$= \boldsymbol{w}^{[2]} - \eta \frac{\partial l}{\partial \boldsymbol{o}} * \frac{\partial \boldsymbol{o}}{\partial \boldsymbol{w}^{[2]}}$

输入到隐藏层的权重矩阵梯度下降更新为：

$\boldsymbol{w}^{[1]} = \boldsymbol{w}^{[1]} - \eta \frac{\partial l}{\partial \boldsymbol{w}^{[1]}}$
$= \boldsymbol{w}^{[1]} - \eta \frac{\partial l}{\partial \boldsymbol{o}} * \frac{\partial \boldsymbol{o}}{\partial \boldsymbol{h}} * \frac{\partial \boldsymbol{h}}{\partial \boldsymbol{w}^{[1]}}$

综上，第二题的整个过程已计算完毕，则代码实现就十分简单了。

代码实现

首先，两种初始化网络参数的方式，一个随机初始化，一个给定参数矩阵：

```
def intializeSameWeigths():
    hiddenWeights = [0.5, 0.5, 0.5, 0.5]
    outputWeights = [-0.5, -0.5]
    return hiddenWeights, outputWeights

def intializeRandomWeigths():
    hiddenWeights = [0, 0, 0, 0]
    outputWeights = [0, 0]
    for i in range(len(hiddenWeights)):
        hiddenWeights[i] = random.uniform(-1.0, 1.0)
    for i in range(len(outputWeights)):
        outputWeights[i] = random.uniform(-1.0, 1.0)
    return hiddenWeights, outputWeights
```

之后，直接获取w_1, w_2类别的数据集，真实标签(label) $\boldsymbol{y}^{(i)}$使用的是1.716去表示第一类(表格中的w_1类)，-1.716去表示第二类(表格中的w_2类)，原因在上面的分析已详细解释。

```
def getDataset():
    # label 1.716-> w1  -1.716-> w2
```

```
    dataset = []
    labels = []
    dataset = [[0.28, 1.31, -6.2],
        [0.07, 0.58, -0.78],
        [1.54, 2.01, -1.63],
        [-0.44, 1.18, -4.32],
        [-0.81, 0.21, 5.73],
        [1.52, 3.16, 2.77],
        [2.20, 2.42, -0.19],
        [0.91, 1.94, 6.21],
        [0.65, 1.93, 4.38],
        [-0.26, 0.82, -0.96],
        [0.011, 1.03, -0.21],
        [1.27, 1.28, 0.08],
        [0.13, 3.12, 0.16],
        [-0.21, 1.23, -0.11],
        [-2.18, 1.39, -0.19],
        [0.34, 1.96, -0.16],
        [-1.38, 0.94, 0.45],
        [-0.12, 0.82, 0.17],
        [-1.44, 2.31, 0.14],
        [0.26, 1.94, 0.08]]
    labels = [1.716,1.716,1.716,1.716,1.716,1.716,1.716,1.716,1.716,1.716,\
        -1.716,-1.716,-1.716,-1.716,-1.716,-1.716,-1.716,-1.716,-1.716,-1.716]
    indexes = [0,1,2,3,4,5,6,7,8,9,\
        10,11,12,13,14,15,16,17,18,19]
    return dataset, labels, indexes
```

之后使用Stochastic backpropagation进行训练，按照题目所给伪代码：

Algorithm 1 (Stochastic backpropagation)

1. **begin initialize** network topology(# hidden units),**w**, criterion θ, η, $m \leftarrow 0$
2. **do** $m \leftarrow m + 1$
3. $\mathbf{x}^m \leftarrow$ randomly chosen pattern
4. $w_{ij} \leftarrow w_{ij} + \eta\, \delta_j\, x_i$; $\quad w_{jk} \leftarrow w_{jk} + \eta\, \delta_k\, y_j$
5. **until** $\nabla J(\mathbf{w}) < \theta$
6. **return w**
7. **end**

因为题目需要给出每个epoch的训练误差，而**1个epoch表示过了1遍训练集中的所有样本**，因此我将伪代码中的m认为是epoch次数，而epoch里面分为很多个steps，所以第三行我将它解释为在每一个step时随机选取一个样本进行训练，然后第五行解释为一个epoch里面的平均梯度小于一定阈值之后则退出训练，其中阈值我设置为0.005。

因此实现代码如下：

```
while 1:
    trainingError = 0.0
    epochGradient = 0.0
    correctNum = 0.0
    # shuffle dataset
    random.shuffle(indexes)
    # stochastic gradient descend training
    for index in indexes:
        data = dataset[index]
        label = labels[index]
        predict = net2a.inference(data)
        labelSign = math.copysign(1, label)
        if (predict == labelSign):
            correctNum += 1
        stepTrainingError, stepGradient = net2a.train(data, label)
        trainingError += stepTrainingError
        epochGradient += stepGradient
    trainingError = trainingError / len(dataset)
    trainingErrors.append(trainingError)
    epochs.append(epoch)
    accuracy = correctNum / len(dataset)
    accuracyList.append(accuracy)
    epochGradient = epochGradient / len(dataset)
    if np.absolute(epochGradient).mean() < theta:
        break
    epoch += 1
```

每次epoch都会将整个数据集数据遍历到，先打乱一下数据集，然后就可以达到每个step能够随机选取样本的目的，每个step随机选取一个样本先inference一下得到predict用于计算准确率(accuracy)，之后再输入网络进行训练同时返回训练误差和梯度，每一个epoch都会计算出该epoch的平均训练误差、平均准确率以及平均梯度，当平均梯度小于一定阈值之后则退出训练。

网络训练函数如下：

```
def train(self, data, label):
  self.label = np.array(label)
  output = self.forward(data)
  loss = self.calculateLoss(output, label)
  gradient = self.backpropagation(loss)
  return loss, gradient
```

先forward一次得到网络输出，再通过self.calculateLoss计算损失：

```
def calculateLoss(self, predict, label):
  return ((predict - label)**2).mean()
```

然后进行反向传播：

```
def backpropagation(self, loss):
  learningRate = 0.1
  # partial loss/output
  partialLossDividedOutput = 2*(self.output - self.label)
  # partial output/outputWeights
  partialOutputDividedOutputWeights =
self.derivativeByActivation(self.outputBeforeActivationFunc)*self.hiddenOutput
  # partial output/hiddenOutput
  partialOutputDividedHiddenOutput =
self.derivativeByActivation(self.outputBeforeActivationFunc)*self.outputWeight
s[1]
  # partial hiddenOutput/hiddenWeights
  partialHiddenOutputDividedHiddenWeights =
self.derivativeByActivation(self.hiddenOutputBeforeActivationFunc)*self.input
  # partial loss/outputWeights
  partialLossDividedOutputWeights = partialLossDividedOutput *
partialOutputDividedOutputWeights
  # partial loss/hiddenWeights
  partialLossDividedHiddenWeights = partialLossDividedOutput *
partialOutputDividedHiddenOutput * partialHiddenOutputDividedHiddenWeights
  # gradient descend on outputWeights
  self.outputWeights = self.outputWeights - learningRate *
partialLossDividedOutputWeights
  # gradient descend on hiddenWeights
  self.hiddenWeights = self.hiddenWeights - learningRate *
partialLossDividedHiddenWeights
  return np.append(partialLossDividedOutputWeights,
partialLossDividedHiddenWeights)
```

反向传播中通过上面已经分析过的计算梯度公式来计算梯度，然后最后两行通过梯度下降公式来更新参数，最后返回当前计算得到的梯度。在代码中self.derivativeByActivation是计算激活函数对内部值的求导，在前面公式中也已经计算过，代码如下：

```
def derivativeByActivation(self, out):
  a = 1.716
  b = 2.0/3.0
  k = 1.0/(1+np.exp(-b*out))
  return 2*a*b*k*(1-k)
```

通过上述，整个训练过程的代码就已经介绍完毕，下面介绍实验结果以及两种初始化方法的结果分析：

实验结果与分析

在我的大量实验中发现，大部分的随机初始化的效果会比在每一层初始化为相同的值要好，而也有少部分时候随机初始化的效果与每一层初始化为相同的值差不多。

我通过random seed来控制不同的随机初始化，random seed不同，则会有一组不同的随机初始化参数。以下分别展示每一层初始化为相同的值的结果与三组不同random seed的结果比较。

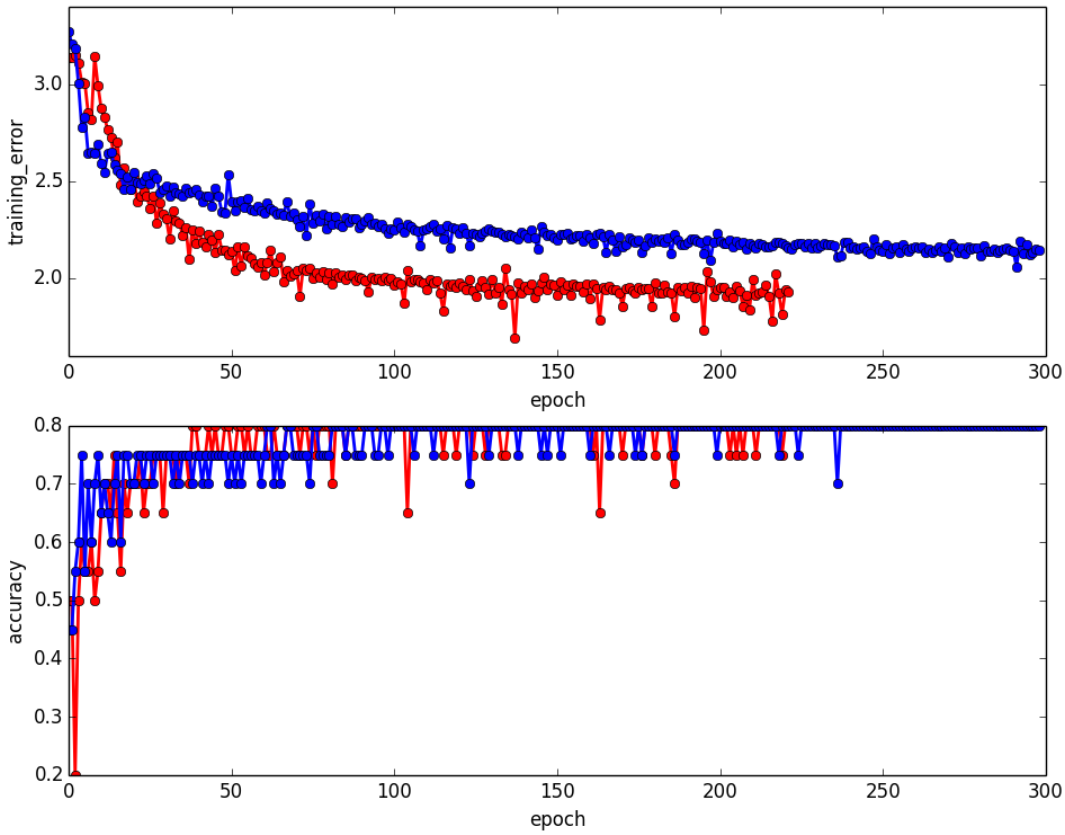
统一设置：每一组中我设置的退出训练阈值均为0.005，每一层初始化为相同值的方案即题目（b）问的方案：

$$\begin{bmatrix} w_0^{[1]} & w_1^{[1]} & w_2^{[1]} & w_3^{[1]} \end{bmatrix} = [0.5 \quad 0.5 \quad 0.5 \quad 0.5]$$

$$\begin{bmatrix} w_0^{[2]} & w_1^{[2]} \end{bmatrix} = [-0.5 \quad -0.5]$$

1.Random seed(6875673) vs Same throughout each level

训练误差曲线与准确率曲线比较，红色线为随机初始化，蓝色线为每一层初始化为相同值：

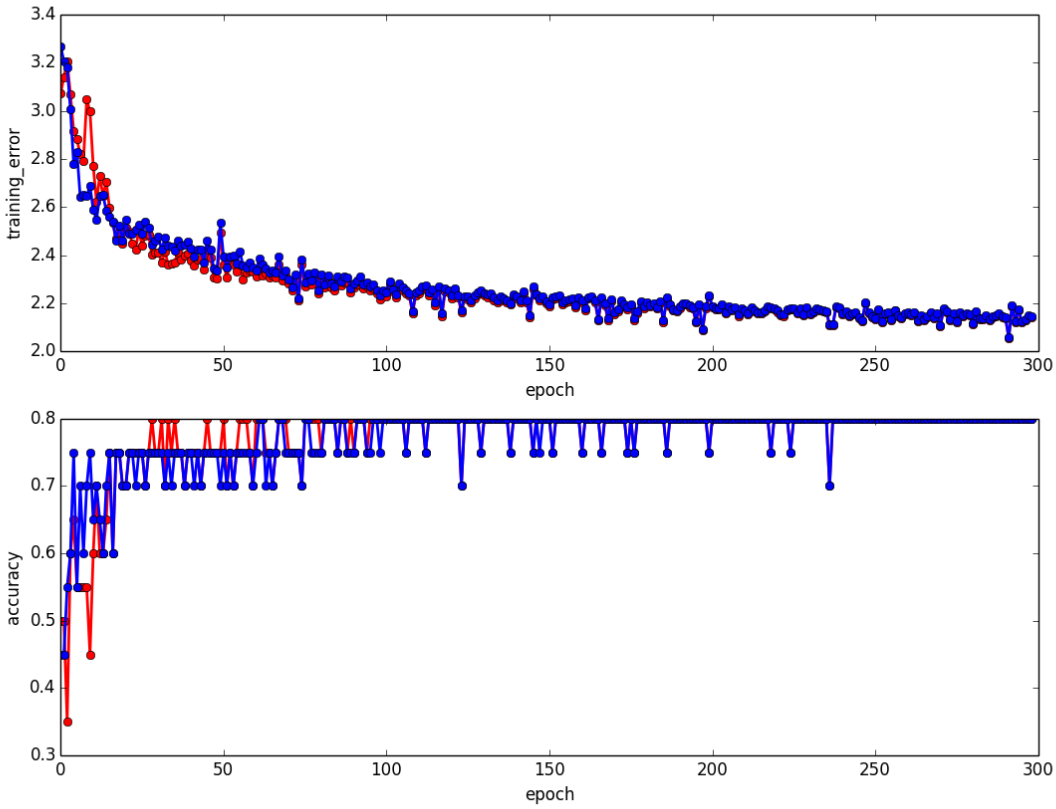


该random seed初始化出来的初始化参数 以及 两种初始化方法达到阈值时退出训练的epoch 如下：

```
('Random Hidden Weights:', [0.9982055994401462, 0.2126774398275122,
-0.8786620626084363, -0.4325124707588217])
('Random Output Weights:', [0.5186560335298704, -0.7963187458696399])
('Break Epoch in initialize all weights randomly:', 221)
('Break Epoch in weights initialized to be the same throughout each level:',
298)
```

2.Random seed(0) vs Same throughout each level

训练误差曲线与准确率曲线比较，红色线为随机初始化，蓝色线为每一层初始化为相同值：

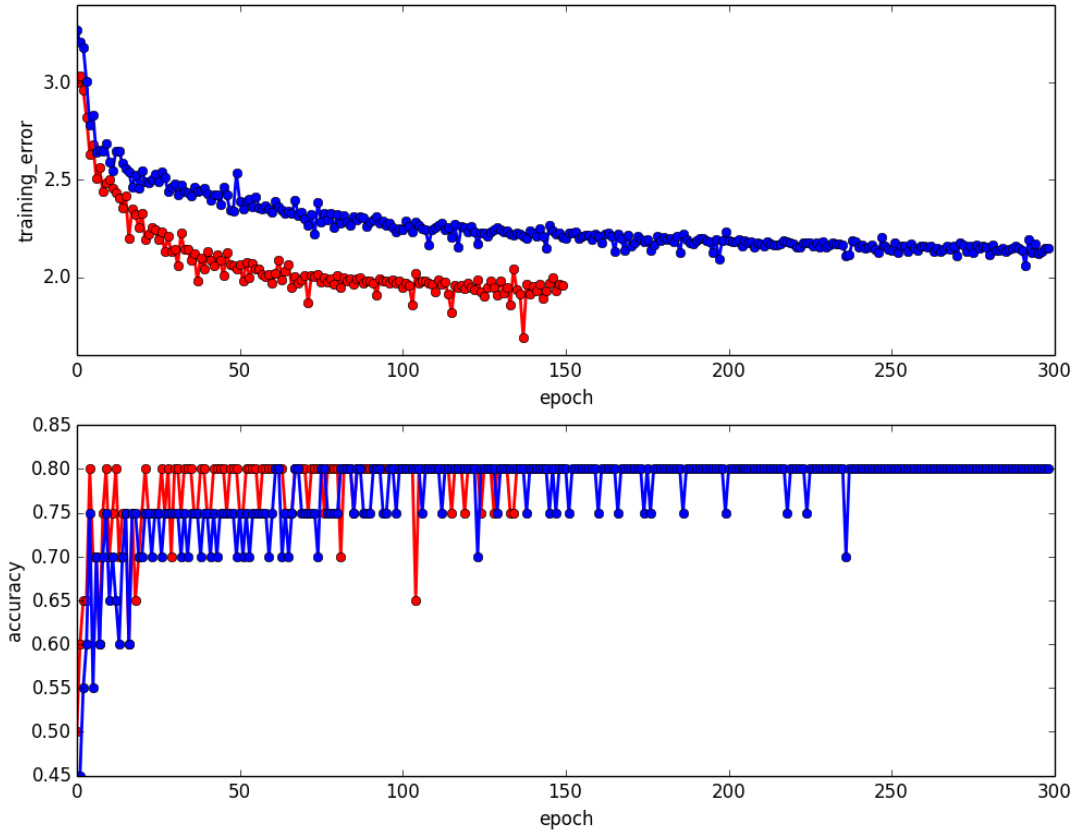


该random seed初始化出来的初始化参数 以及 两种初始化方法达到阈值时退出训练的epoch 如下：

```
('Random Hidden Weights:', [0.6888437030500962, 0.515908805880605,
-0.15885683833831, -0.4821664994140733])
('Random Output Weights:', [0.02254944273721704, -0.19013172509917142])
('Break Epoch in initialize all weights randomly:', 298)
('Break Epoch in weights initialized to be the same throughout each level:',
298)
```

3.Random seed(1500) vs Same throughout each level

训练误差曲线与准确率曲线比较，红色线为随机初始化，蓝色线为每一层初始化为相同值：



该random seed初始化出来的初始化参数 以及 两种初始化方法达到阈值时退出训练的epoch 如下：

```
('Random Hidden Weights:', [-0.3793387404607793, 0.6229944142121933,
-0.3224004941284364, -0.3494584654638493])
('Random Output Weights:', [0.5356593179104086, 0.8961368778300058])
('Break Epoch in initialize all weights randomly:', 149)
('Break Epoch in weights initialized to be the same throughout each level:',
298)
```

总结与原因分析

通过上面的实验结果发现有如下现象：

1. 根据我多次实验的结果发现，大部分情况下，随机初始化都能够比每一层初始化为相同值 提前退出训练（如1，3实验），也就是提前收敛。
2. 同时大部分情况下，随机初始化的training_error训练误差都会比每一层初始化为相同值 低（如1，3实验）。
3. 大部分情况下，随机初始化的accuracy准确率也都会比每一层初始化为相同值 提前达到80%（如1，2，3实验），其中80%是我大部分实验中的最高准确率。
4. 也有一部分情况下，随机初始化都能够比每一层初始化为相同值方法 结果类似，在相同epoch退出训练，训练误差和准确率也接近（如2实验）。

因此我分析原因如下：

其实随机初始化模型参数其中一个原因是为了“打破对称性”，这里的“打破对称性”指的是在进行梯度下降更新参数时隐藏层隐藏单元的梯度不能全都相同，否则就达不到训练模型的效果。如果权重初始化为同一个值，网络就会是对称的。

因为如果每一层初始化为相同值，那正向传播的时候，每个隐藏单元都会计算出相同的值（因为输入相同权重也相同），然后传到输出层。在反向传播中，计算出来的每个隐藏单元的参数梯度值也会相等，然后通过梯度下降算法更新参数后，每个隐藏单元的参数依旧相同。那每一次更新参数也依旧如此。所以在这种情况下，对于隐藏层，无论隐藏单元有多少，本质上只有1个隐藏单元在发挥作用，因此会导致网络必然对称，也会影响模型的训练效果。

不过这个问题中其实隐藏单元只有一个，它是一个3-1-1的网络，所以权重初始化为相同值可能对于模型训练的副作用没有那么严重。

另外，对于随机初始化，有的时候也可能会初始化到不太好的初始参数，所以其实也是有可能效果和权重初始化为相同值的方法差不多，其实也有可能会比较差，但是大部分情况下是比权重初始化为相同值的效果要好的。