第四周深度学习作业报告

陆东伟 计算机应用技术

这周的报告有两部分组成，第一部分是上周的神经网络分层改写，另一部分是关于二维和三维数据的多次拟合实验。

Chapter 1 Rebuild Neural Network Code with Layers

1.1 Layers

上周的代码实现了神经网络的 BP 算法，以实现对数据的拟合。但是为了应对复杂大型神经网络的扩展和构建，上周的代码由于各部分功能相互交叉，功能划分拆解仍不够细分，应对未来复杂的神经网络层的出现，很难不重新修改原油代码，其次，应对不同规模大小和不同维度的样例，绘制的可视化的学习曲线或者是分类图无法很好的表述出来，因此在细分功能上还需要把接口进行独立和划分，即 MVC 的程序思想。神经网络出现于是按照学姐给出的分层思想的思路重新编写工程。以下介绍以下目前采用分层思想架构的神经网络工程的代码意图。

```
class BaseLayer(metaclass=abc.ABCMeta):

    last_layer = None
    next_layer = None
    value = None #n*m array for forward computation
    num = 0      #num of elements in column not num of samples in row
    delta = None #n*m array for backward computation
    def forward_compute(self):
        pass
    def backward_compute(self):
        pass
    def SetLast(self,lastLayer):
        self.last_layer = lastLayer
    def SetNext(self,nextLayer):
        self.next_layer = nextLayer
```

图 1.1.1 基层抽象类

这个基层类将所有层的行为都抽象出来，主要有四个方法，前向、反向计算，设置下一层、前一层，另外还有一些共有的属性，如这层的节点值 value，这层的节点数 num，这层反向计算过来的偏差 delta。

```
class InputLayer(BaseLayer):
    def __init__(self,num):
        self.num = num
        pass
    def setValue(self,value):
        self.value = value
```

图 1.1.2 输入层

这层继承了基层抽象类，特有的方法是设置输入值 value，即输入样本。

```
class InnerProductLayer(BaseLayer):
    theta = None
    grad_theta = None
    epsilon = 0.25
    rate = 0.1
    def __init__(self,num):
        self.num = num
        pass
    def SetLast(self,lastLayer):
        super().SetLast(lastLayer)
        self.theta = np.random.rand(self.num,lastLayer.num+1)*2*self.epsilon-self.epsilon
        pass
    def forward_compute(self):
        m = np.size(self.last_layer.value,1)
        tmp = np.vstack((np.ones([1,m]),self.last_layer.value))
        self.value = self.theta.dot(tmp)
    def backward_compute(self):
        m = np.size(self.last_layer.value,1)
        #self.delta = self.next_layer.theta.transpose().dot(self.next_layer.delta)
        tmp = np.vstack((np.ones([1,m]),self.last_layer.value))
        self.grad_theta = self.delta.dot(tmp.transpose())/(1.0*m)
        self.theta -= self.grad_theta*self.rate
        self.last_layer.delta = self.theta.transpose().dot(self.delta)
        self.last_layer.delta = self.last_layer.delta[1:,:]
```

图 1.1.3 全连层

这一层是对上一层进行线性变换计算的层，内含上一层到这一层的 theta 矩阵，初始化 epsilon 范围，学习速率等属性，在函数设置上，SetLast 函数对基层抽象类的方法进行覆盖，实现基本的连接上一层的操作的同时，取出上一层的节点数量再进行这一层的 theta 矩阵初始化。一个设计要点是，每一层的 value 都是和节点数量一一对应的，虽然计算的时候需要设置偏置项，但是在存储时偏置项统一不放入矩阵中，即随调随用。Backward 的时候也是将这层的偏差 delta 算好了传递给上一层，计算后把第一层偏置项给丢掉。

```
class ActivateLayer(BaseLayer):
    def SetLast(self,lastLayer):
        self.last_layer = lastLayer
        self.num = lastLayer.num
```

图 1.1.4 激活抽象层

设置上一层，覆盖了父类方法，因为要增加提取上一层节点数量的操作。

```
class SigmoidLayer(ActivateLayer):
    def __init__(self):
        pass
    def forward_compute(self):
        self.value = 1.0/(1+np.exp(-self.last_layer.value))
    def backward_compute(self):
        self.last_layer.delta = self.delta*     (self.value)*(1-self.value)
        pass

class TanhLayer(ActivateLayer):
    def __init__(self):
        pass
    def forward_compute(self):
        self.value = np.tanh(self.last_layer.value)
    def backward_compute(self):
        self.last_layer.delta = self.delta*     (1-self.value**2)

class ReLuLayer(ActivateLayer):
    alpha = 1
    def __init__(self):
        pass
    def forward_compute(self):
        self.value = np.maximum(0,self.alpha*self.last_layer.value)
    def backward_compute(self):
        tmp = self.value.copy()
        tmp[tmp<=0] = 0
        tmp[tmp>0] = self.alpha
        self.last_layer.delta = self.delta*tmp
```

图 1.1.5 激活层

三个不同的激活层类，继承了抽象激活层，分别是 Sigmoid，Tanh，ReLu，在正向和反向传播的方法都有所不同。

```
class OutputLayer(BaseLayer):
    h = None #hippothesis
    y = None #standard output
    costFunc = None
    def __init__(self):
        pass
    def LMS(self):
        res = np.sum((self.h-self.y)**2)/(2.0*np.size(self.y,1))
        return res
    def SoftMax(self):
        return -np.sum(self.y*np.log(self.h))/(1.0*np.size(self.y,1))

    def init(self,costFuncName='LMS'):
        if costFuncName is 'LMS':
            self.costFunc = self.LMS
        elif costFuncName is 'SoftMax':
            self.costFunc = self.SoftMax
    def setY(self,y):
        self.y = y
    def forward_compute(self):
        if self.costFunc == self.LMS:
            self.h = self.last_layer.value
        elif self.costFunc == self.SoftMax:
            fenmu = np.sum(np.exp(self.last_layer.value),axis=0)
            self.h = np.exp(self.last_layer.value)/fenmu
    def backward_compute(self):
        self.last_layer.delta = self.h-self.y
```

图 1.1.6 输出层

设置了输出层，提供了两个不同的计算 CostFunction 的方法，该层初始化需要传入 CostFunction 的函数名称 ,初始化成功才可进行网络的计算 ,forward 和 backward 都依赖 CostFunction 的选择。另外一个它唯一的函数是 setY，对输出样本进行设置。

1.2 Network Interface Design

```
def safety_check(self):
    #step 1
    res = True
    for e in self.initLayerList:
        if self.initLayerList[e] is False:
            print('%s has not been initialized.'%e)
            res = False
    if res is False:
        return res

    #step 2
    if self.initializeList['CONNECT_LAYERS']is False:
        print('LAYERS has not been connected.')
        print('Connect them now...')
        self.initializeList['CONNECT_LAYERS'] = self.ConnectLayers()
        print('LAYERS has been connected.')

    #step 3
    if self.gradiant_check() is False:
        return False
    print('Safety check pass!')
    return True
```

图 1.2.1 安全检查

这是一个即将运行在训练过程最开始需要运行的函数 ,他会检查一些网络开始学习所必须的组成部分是否都设置完毕。Step1，他会检查你的 initLayerList 中的层是否都初始化，Step2，会检查出了初始化层其他必须的操作，如连接各层的关系，组成类似双向链表的可前后关联的结构，Step3，进行梯度检查，梯度检查所需要的数据会在安全检查开始之前就设置好。

```
def SetInputLayer(self,inputNum):...

def SetOutputLayer(self,costFuncName):...

def SetInnerProductLayer(self,layerNodeList):...

def SetActivationLayer(self,activeName):...


def ConnectLayers(self):
    self.sequences = []
    self.sequences.append(self.input_layer)
    for layer in self.inner_product_layer:
        self.sequences[-1].SetNext(layer)
        layer.SetLast(self.sequences[-1])
        self.sequences.append(layer)

        if self.activeName=='Sigmoid':
            activeLayer = SigmoidLayer()
        elif self.activeName =='Tanh':
            activeLayer = TanhLayer()
        elif self.activeName =='ReLu':
            activeLayer = ReLuLayer()

        self.sequences[-1].SetNext(activeLayer)
        activeLayer.SetLast(self.sequences[-1])
        self.sequences.append(activeLayer)
    if self.activeName =='ReLu':
        self.sequences.pop()
        activeLayer = SigmoidLayer()
        self.sequences[-1].SetNext(activeLayer)
        activeLayer.SetLast(self.sequences[-1])
        self.sequences.append(activeLayer)

    if self.output_layer.costFunc == self.output_layer.SoftMax:
        self.sequences.pop()
    self.sequences[-1].SetNext(self.output_layer)
    self.output_layer.SetLast(self.sequences[-1])
    self.sequences.append(self.output_layer)
    return True
```

图 1.2.2 各必要初始化函数

ConnectLayers 是一个建立网络结构的过程方法，sequences 是我这个 Model 的层的序列的，建立网络。对于如今仅仅考虑全连层的神经网络，我需要先放置输入层，再放置全连层，每个全连层后面跟着一个激活层，建立前后的关系连接，如果最后输出层是使用 SoftMax 函数，需要取出最后那个激活层，直接进行全连层和输出层的连接。另外，如果使用 ReLu 的激活层，最后一层的激活层将会替换使用 Sigmoid 或者 Tanh 激活函数。

提供给神经网络搭建工程师的接口函数如下：

```
def forward_compute(self):
    layerNum = len(self.sequences)
    for i in range(1,layerNum):
        self.sequences[i].forward_compute()

def backward_compute(self):
    layerNum = len(self.sequences)
    for j in range(1,layerNum)[::-1]:
        self.sequences[j].backward_compute()

def costFunction(self):
    return self.output_layer.costFunc()

def minibatch_train(self,batch_size,steps):

    if self.safety_check() is False:
        return False

    total_size = np.size(self.train_x,1)
    if batch_size > total_size:
        print('batch size is too large.')
        return False
    elif batch_size<=1:
        print('batch size is too small.')
        return False
    draw_cost = []
    draw_steps = []
    for i in range(steps):
        choose = random.sample(range(total_size),batch_size)
        self.input_layer.setValue(self.train_x[:,choose])
        self.output_layer.setY(self.train_y[:,choose])
        self.forward_compute()
        self.backward_compute()
        draw_cost.append(self.output_layer.costFunc())
        draw_steps.append(i)
    plt.plot(draw_steps,draw_cost,marker='.')
    plt.title('Train with batch size %s' % batch_size)
    plt.show()
    return True
```

图 1.2.3 前后传递，训练网络

进行批量学习的最后会生成一个学习曲线。

1.3 Real Application

```
from NeuralNetwork.NeuralNetwork import NeuralNetwork
from NeuralNetwork.Layers.core import *
from initialize import *
if __name__ == '__main__':
    myann = NeuralNetwork('Dong Dong Network')
    myann.model.SetInputLayer(2)
    myann.model.SetOutputLayer('SoftMax')
    myann.model.SetInnerProductLayer([15,2])
    myann.model.SetActivationLayer('Tanh')



    Na = 500
    Nb = 500
    (x,y) = initialize(10,6,-4,Na,Nb)
    #y[y==0]=-1    #use tanh to classify
    yy = np.zeros([2,Na+Nb])
    for i in range(Na+Nb):
        if y[0,i]==1:
            yy[0,i] = 1
        else:
            yy[1,i] = 1
    myann.setSamples(x,yy)
    #myann.setSamples(x,y)
    myann.MiniBatch_Train(700,2000)
```

图 1.3.1 测试脚本

这个脚本显示了建立神经网络需要的步骤，设置输入节点量，设置输出层函数，设置全连层的各层个数，设置激活层，然后对整个网络放入样本，就可以进行样例的训练、交叉测试、最终测试的分割。

Chapter 2 Build Model for 2-D or 3-D Data Sets

对于拟合数据点的作业，我在思考进行高次项的模型建立时如何尽可能多的不漏过高次信息，于是编写了 2 维 3 维的组合程序，可以实现如 3 维 2 次 z = w0+w1*x+w2*y+w3*x^2+w4*y^2+w5*xy 的式子的建立，以及 2 维 3 维多次特征的建立。可是遇到了一个问题，就是关于隐函数的模型建立。下图分别是二维数据和三维数据的拟合随着次数增加对应的错误率下降信息的对比。结果发现，即使训练集在 10%，也可以完好的拟合数据，这可能是采用正规方程的优势，最小误差的方式得到近似解。

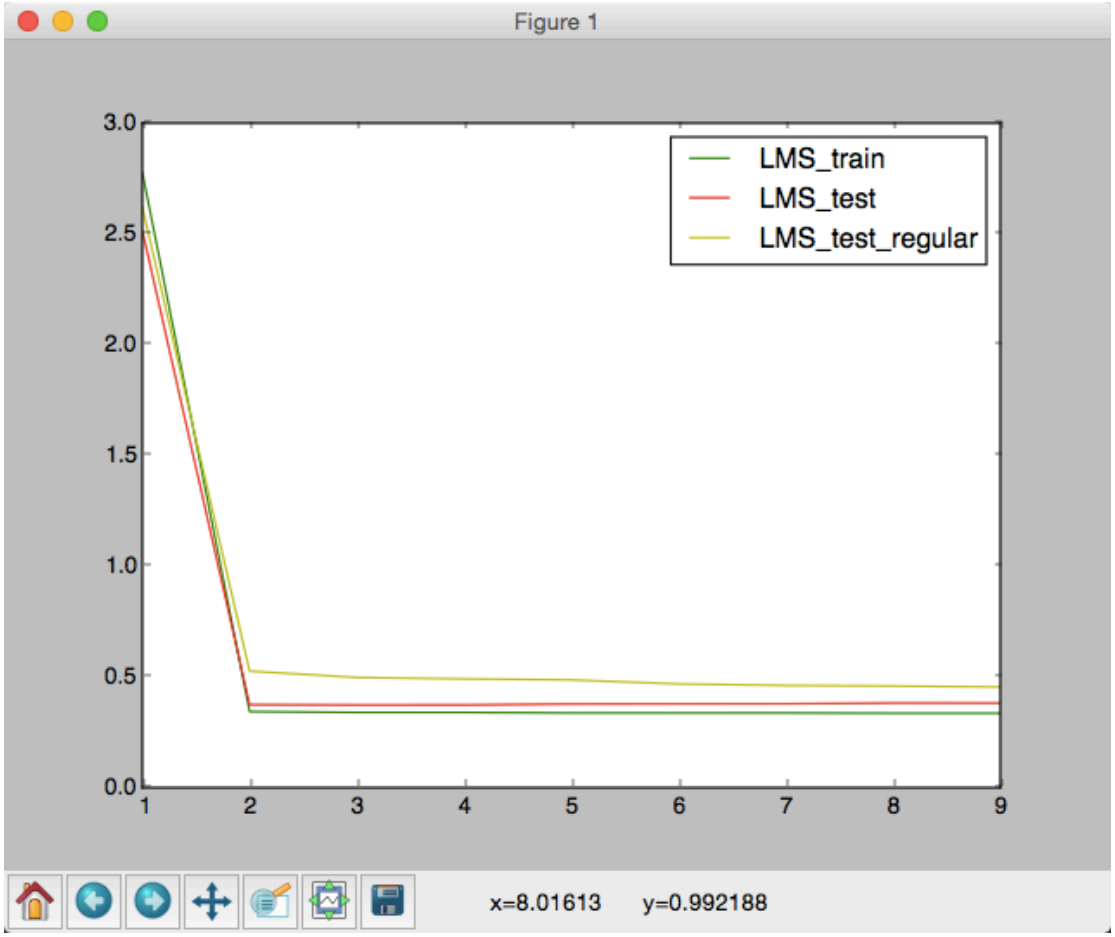

图 2.1 二维数据拟合

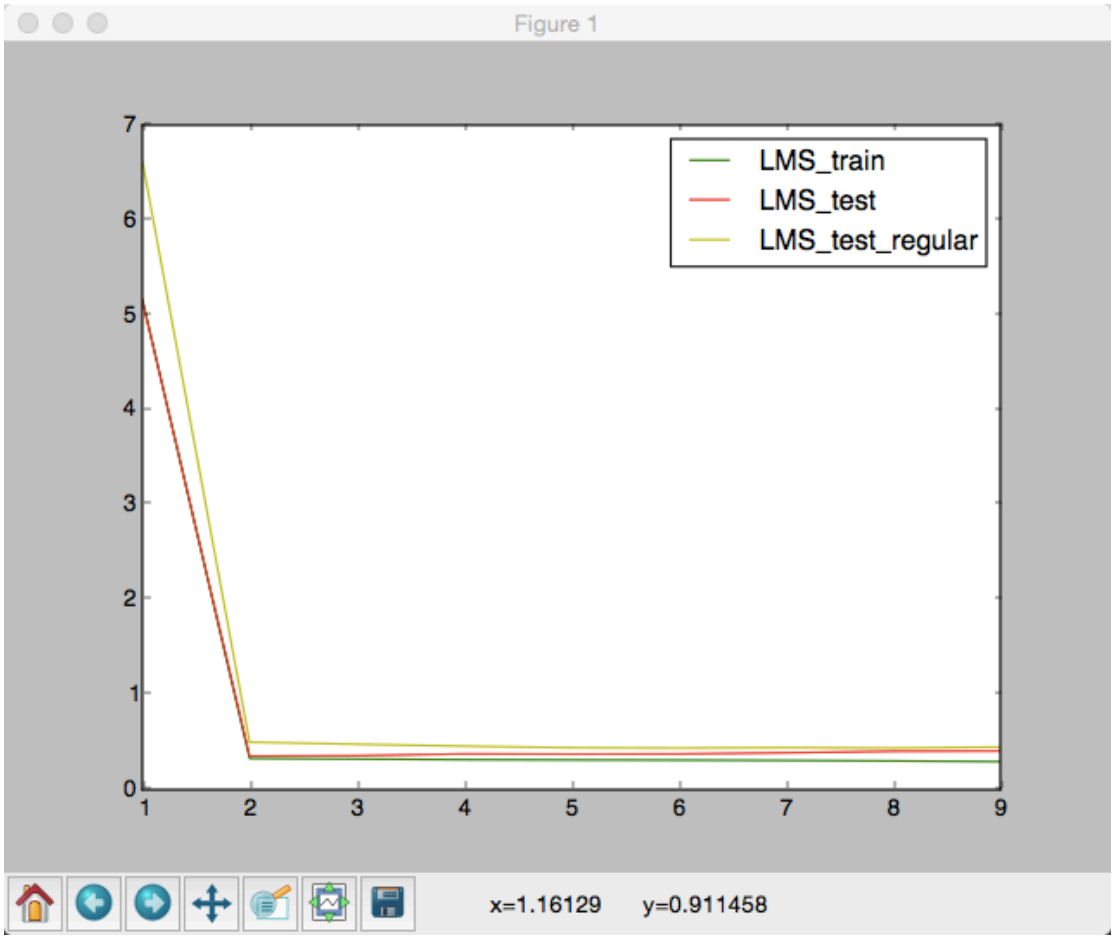

图 2.2 三维数据拟合

其中带有正则化的式子是采用 lambda 值为 500， 没有什么明显的变化。