Neural Network

1. 数据集介绍

本次选用的数据集为Iris数据集，数据集地址:https://archive.ics.uci.edu/dataset/53/iris

该数据集是分类数据集，共有120条数据，每行数据有4个维度的特征，且都为连续变量

目标类别有三种，分别为Iris Setosa, Iris Versicolour和Iris Virginica

2. 程序模块介绍

2.1 Network

为了设计反向传播算法，首先我们需要定义反向传播中需要的网络层，本次我们处理的是一维变量，且为分类任务，因此我们需要设计全连接层，激活函数层，损失函数和用于分类的softmax层，下面我们将介绍这四个层的设计

2.1.1 basicNet

首先我们需要设计一个基类basicNet，后面的具体层实现需要继承这个类，为了实现反向传播，我们需要实现3个函数

forward(self):该函数用于前向传播，需要实现每个函数的具体定义

back_pro(self,pre_dif):该函数用于反向传播，原理是之前积累的梯度乘自身的梯度，用于实现梯度下降，在这里因为并未实现计算图来完成自动求导，因此我们需要对每个层进行具体求导，从而手动实现梯度下降的过程

update_weight(self):根据梯度下降算法，更新每层网络的参数，本次任务里实际上只用更新全连接层的参数即可

2.1.2 fullConnectLayer

在该类里我们将实现全连接层的代码:

1. 首先我们需要初始化全连接层的权重和偏差，在这里我们进根据输入的(in_dim,out_dim)随机初始化对应矩阵

```
self.weight=np.random.normal(loc=0.0,scale=std,size=(self.in_dim,self.out_dim))
self.bias=np.zeros([1,self.out_dim])
```

2. 之后我们实现前向传递的forward函数，使用numpy的矩阵乘法运算即可

```
return np.matmul(input,self.weight)+self.bias
```

3. 之后实现反向传播函数，对y=AX+B求导我们可以得到反向传播的具体实现为，注意np.sum的维度

```
self.weight_dif=np.dot(self.input.T,pre_dif)
self.bias_dif=np.sum(pre_dif,axis=0)
return np.dot(pre_dif,self.weight.T)
```

4. 最后需要实现梯度的更新，根据输入的步长lr，根据梯度下降算法进行更新

```
self.weight=self.weight-self.lr*self.weight_dif
self.bias=self.bias-self.lr*self.bias_dif
```

2.1.3 reluLayer

在激活函数上我们选择relu作为激活函数，因为它本身定义和求导都比较简单，而且也能避免梯度消失的问题，

需要注意对于relu函数求导，对于x<0的部分，将其导数设置为0即可，>0的部分导数保持不变即可，其对应代码如下:

```
output_dif=pre_diff
output_dif[self.input<0]=0
return output_dif
```

2.1.4 SoftmaxLayer

作为分类问题，在得到模型输出后，需要进行softmax操作将其转变为概率值，用于之后计算loss，在计算中发现使用原始softmax可能会得到nan值，查询后得知需要先从所有x中减去max(x),之后再进行softmax，因此得到其forward函数为:

```
self.input_max=np.max(input,axis=1,keepdims=True) #注意axis=1
self.output_mid=np.exp(input-self.input_max)
self.output=self.output_mid/np.sum(self.output_mid,axis=1,keepdims=True)
```

2.1.5 crossLossLayer

最后我们需要实现损失函数，对于分类问题，我们选择交叉熵作为损失函数，在这里需要对数据进行预处理操作，因为输入的y_true为(n,1)的标签，我们需要对其进行one_hot编码，用于之后和预测的概率值进行log计算，其代码如下:

```
self.batch_size=input.shape[0]
label=np.eye(num_classes)[label]
label = np.squeeze(label, axis=1)
loss=-np.sum(np.log(input)*label)/self.batch_size
```

2.2 Model

该部分将在上述网络模块搭建的基础上，来设计对应的网络架构，并完成训练和验证的部分

2.2.1 网络架构

这里我们选择5层全连接层(其实用在Iris上有点多，主要是为了验证可行性)和4层激活函数，最后通过softmax层后计算损失函数，其网络代码如下:

```
self.num_classes=num_classes
self.fc1=fullConnectLayer(input_size,hidden_sizes[0],lr=_lr)
self.relu1=reluLayer()
self.fc2=fullConnectLayer(hidden_sizes[0],hidden_sizes[1],lr=_lr)
self.relu2=reluLayer()
self.fc3=fullConnectLayer(hidden_sizes[1],hidden_sizes[2],lr=_lr)
self.relu3=reluLayer()
self.fc4=fullConnectLayer(hidden_sizes[2],hidden_sizes[3],lr=_lr)
self.relu4=reluLayer()
self.fc5=fullConnectLayer(hidden_sizes[3],hidden_sizes[4],lr=_lr)
self.classify=SoftmaxLayer()
self.loss_cross=crossLossLayer()
```

2.2.2 训练和验证

我们将数据集的最后30行数据作为验证集，剩余的数据作为训练集，选取batch_size=5,lr=0.01,epoch=100,bacth_size=10,其训练流程如下:

1. 每个epoch中选取1个batch_size的数据，调用forward()函数通过前向传递过程计算模型输出，再计算和true_label的损失
2. 之后再调用back_pro()函数反向传播计算每个网络层的导数，再调用update_weight()函数更新全连接层的参数，之后清空当前loss进行下一batch训练

验证时输入test数据，跟test_trueLabel比较即可

2.3 其他模块

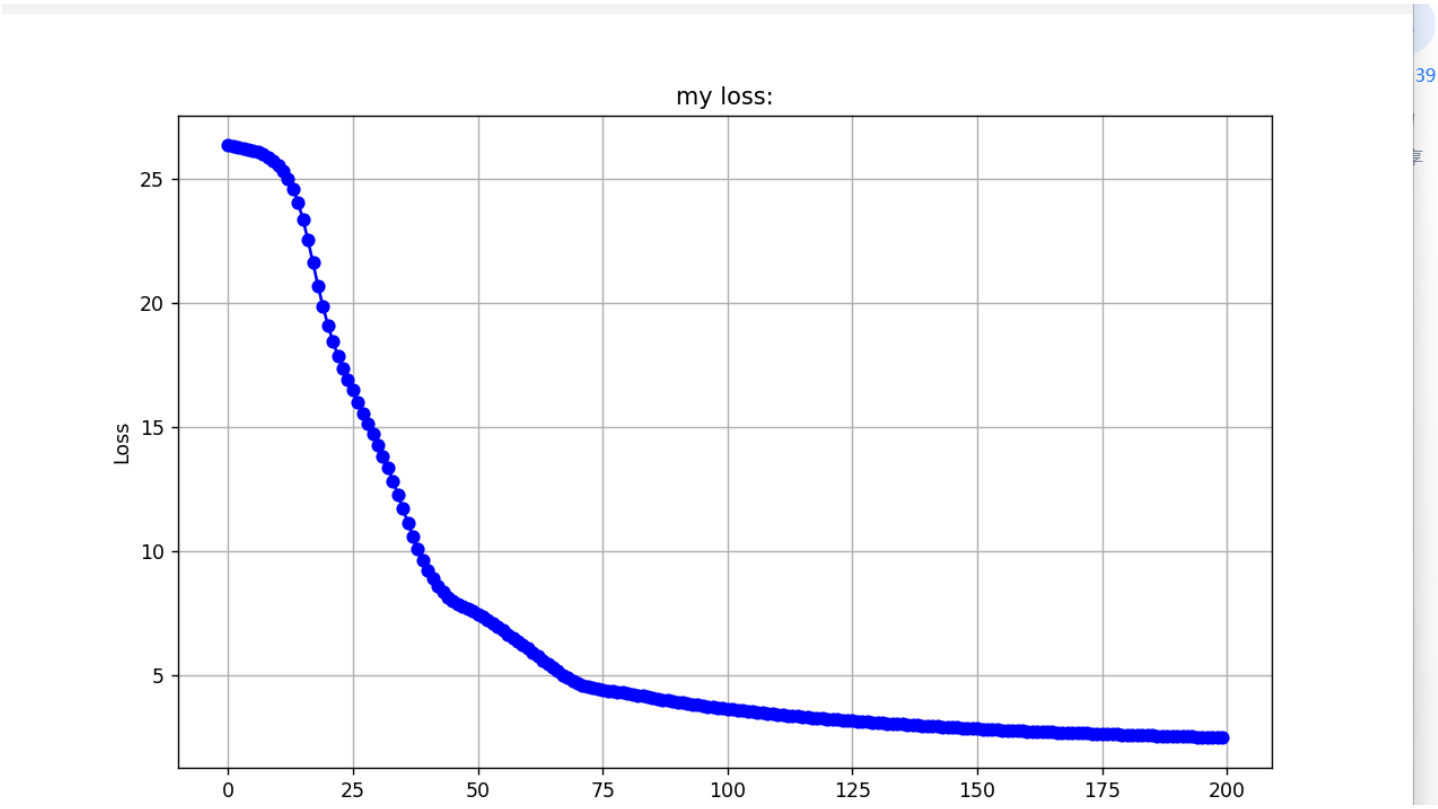
此外还有数据读取，画图和pytorch对比三个模块，主要用来辅助上面的模型训练和验证，详见源码

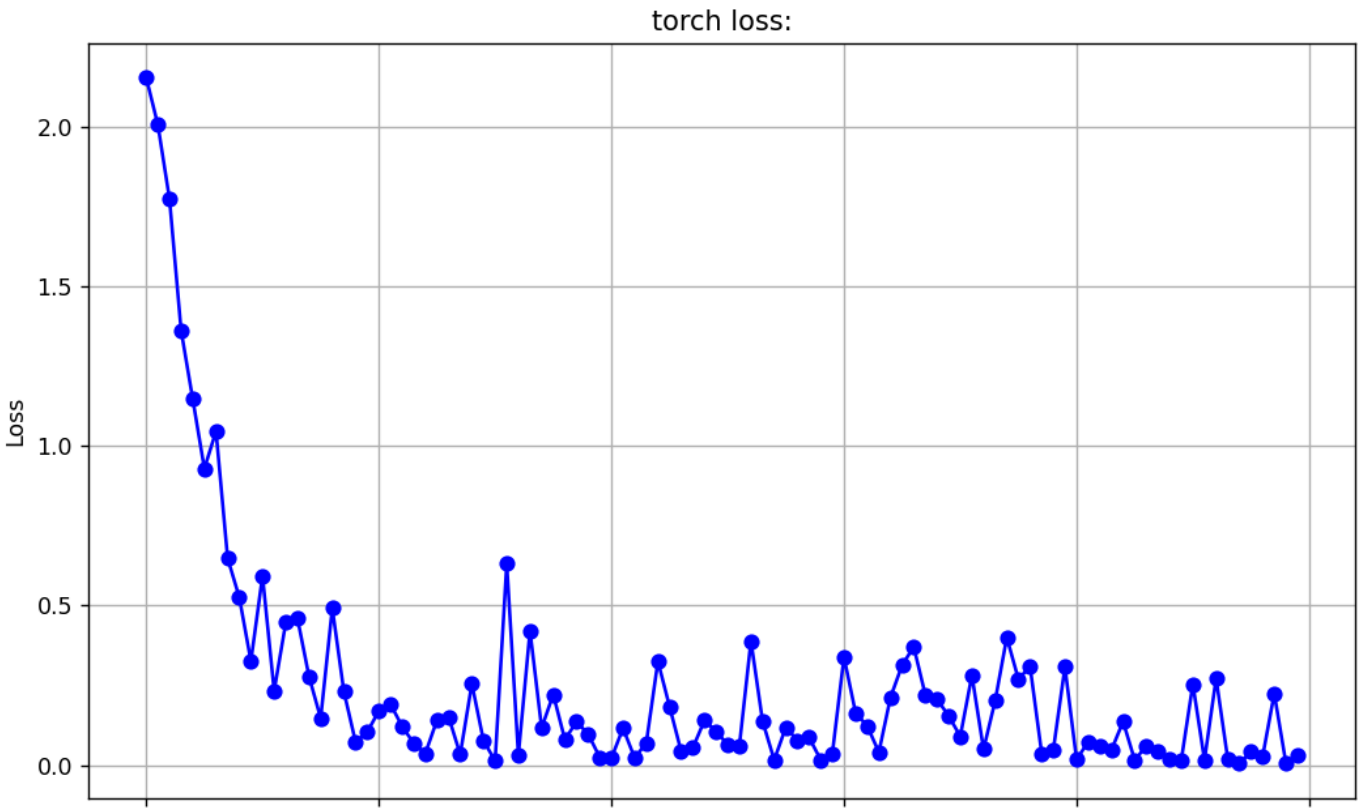
3. 结果展示和验证

3.1 loss对比

在这里我们展示自己模型的loss和pytorch搭建同样模型的loss，如下图所示:

可以发现两者loss都降到很小，但pytorch的loss波荡的比较大，猜测是因为在这里我选择了(128,64,32,16,8)的全连接层，实际复杂了该问题，而我的梯度下降策略比较简单，反而loss比较稳定





3.2 Precision

两者在测试集上的正确率差不多，这里没固定随机数种子，可能每次正确率会出入一些

```
测试集上pytorch模型正确率:0.9655172413793104
测试集上我的模型正确率:0.9310344827586207
```

4. 不足和改进

1. 该反向传播框架因为没有实现计算图，所以实际上不能实现自动微分的功能，在反向传播的过程中，还需要在代码中定义一遍反向传播的路径，对应新增加的网络还需要定义具体导数，之后有时间准备参考CMU 10-414/714: Deep Learning Systems - CS自学指南实现真正的机器学习框架
2. 网络层只实现了很少的一部分，激活函数也只实现了relu，也没有加预测的损失函数，优化器则是选择了最简单的梯度下降，此外发现有些数据集中的feature既有离散也有连续的，用正常神经网络的效果感觉不太好，之后可以继续改进