Readme

Day02

Date:2021-07-25
Made By: 纸 石头 紫阳花

Pytorch 实现线性回归

模型+损失函数+优化器

构建数据集

```
import torch

x_data = torch.Tensor([[1.0], [2.0], [3.0]])
y_data = torch.Tensor([[2.0], [4.0], [6.0]])
```

设计训练模型

模型一般定义为类

构造Linear Unit 确定权重首先要知道输入输出的维度

需要的loss是一个标量（否则无法进行反向传播）故要进行求和运算

```
class LinearModule(torch.nn.Module):          # 继承Pytorch中的Module类初始化为线性模型
    def __init__(self):
        super(LinearModel, self).__init__()
        self.linear = torch.nn.Linear(1, 1) # 模型内部提供weight和bias，都是tensor类型
        #self.linear是一个可调用的对象（包含__call__方法）在模型内规定可以进行预测

    def forward(self, x):
        y_predict = self.linear(x)
        return y_predict
model = LinearModel()
```

nn.Module是所有神经网络的基类，我们构建的所有训练模型都应当将之继承。

backwards方法在module中已经自动实现，调用即可。

nn.Linear(in_feature, out_feature, bias = True):

in_feature 表示输入样本的维度，out_feature 表示输出样本的维度，依据此来确定权重。

权重最开始的随机值由nn.Linear提供，经训练后得到最终权重。

构造损失函数和优化器

```
criterion = torch.nn.MSELoss(size_average=Fasle)
optimizer = torch.optim.SGD(module.parameters(), lr = 0.01)
```

损失函数

MSELoss是继承自nn.Module的类，所有涉及计算图运算的类都需要继承Module类。

code torch.nn.MSELoss(size_average=True, reduce=True)

参数说明size_average表示求损失loss时是否要求均值

参数reduce表示确定是否最后要求和降维。

优化器

优化器`torch.optim.SGD`无需构建计算图，不继承Module，用于求出最优权重。

优化器第一个参数Module.parameters()表示项目权重，parameters方法会检查模型中的所有成员来找到相应的权重（待优化成员）。lr表示学习率。

常用优化器

```
torch.optim.Adagrad
torch.optim.Adam
torch.optim.Adamax
torch.optim.ASGD
torch.optim.LBFGS
torch.optim.RMSprop
torch.optim.Rprop
torch.optim.SGD
```

训练周期（前馈 反馈 更新）

```python
for epoch in range(100):
    y_pred = model(x_date)
    loss = criterion(y_predict, y_data)
    print(epoch, loss)

    optimizer.zero_grad()    # zero_grad方法每轮训练梯度重置
    loss.backward()
    optimizer.step()         # step方法根据反向传播得出的梯度队权重进行一轮优化
```

Logistic Regresssion

分类问题

逻辑回归问题需要将线性模型输出的值从实数空间映射到0-1。故需要Logistic Function来进行映射

LogisticFunction：　$\sigma(x)=\frac{1}{1+e^{-x}}$

通常在神经网络中我也称为激活函数，在层与层的连接中作为非线性因子存在。

二分类问题的损失函数

分布之间的差异根据概率论知识使用cross-entropy（交叉熵）来进行计算

$$Loss=-(y\log_2\hat{y}+(1-y)\log_2(1-\hat{y}))$$

二分类的损失函数称为BCELoss

构建逻辑回归模型

```
import torch.nn.funtional as F

class LogisticRegressionModel(torch.nn.Module):
    def __init__(self):
        super(LogisticRegressionModel, self).__init__()
        self.linear = torch.nn.Linear(1, 1)
        self.sigmoid = torch.nn.Sigmoid()

    def forward(self, x):
        y_pred = F.sigmoid(self.linear(x))
        return y_pred

module = LogisticRegressionModel()

criterion = torch.nn.BCELoss(size_average=False)
optimizer = torch.optim.SGD(module.parameters, lr=0.01)

for epoch in range(100):
    loss = criterion(module(x_data),y_data)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step
```

逻辑回归与线性回归模型在定义上唯一的差别就是在输出期望值前用Losgistic Function来进行映射从而使预测值处于0-1之间。

而损失函数的求法则改为使用BCELoss来求（配合Sigmoid函数）。

处理多维特征的输入

假如一个样本的维度变为了8为则原逻辑回归模型应改为
$\hat{y}^{(i)} = \sigma(\sum_{n=1}^{8} x_n^{(i)} \cdot \omega_n + b)$ 使个特征与权重之积的和作为节点的输出的值。

$$设z_i = \sigma([x_1^{(1)} \cdots x_1^{(8)}] \begin{bmatrix} \omega_1 \\ \omega_2 \\ \omega_8 \end{bmatrix} + b)且 \begin{bmatrix} \hat{y}^{(1)} \\ \hat{y}^{(2)} \\ \hat{y}^{(3)} \\ \cdots \\ \hat{y}^{(8)} \end{bmatrix} = \sigma(\begin{bmatrix} z^{(1)} \\ z^{(2)} \\ z^{(3)} \\ \cdots \\ z^{(8)} \end{bmatrix})$$

则可将模型函数写作如下形式

$$\begin{bmatrix}\hat{y}^{(1)}\\ \cdots \\ \hat{y}^{(N)}\end{bmatrix} = \sigma(\begin{bmatrix}z^{(1)}\\ \cdots \\ z^{(N)}\end{bmatrix}) = \sigma(\begin{bmatrix}x_1^{(1)} & \cdots & x_8^{(1)}\\ & \cdots & \\ x_1^{(N)} & \cdots & x_8^{(N)}\end{bmatrix} \cdot \begin{bmatrix}\omega_1\\ \cdots \\ \omega_8\end{bmatrix} + \begin{bmatrix}b\\ \cdots \\ b\end{bmatrix})$$

在代码中将模型的输入维度改为8维

```
self.linear = torch.nn.Linear(8,1)
```

构造多层神经网络

常用的空间变换里不一定是线性的映射，故可以通过多个线性变换层在找到最优权重后组合成非线性的空间变换函数。故神经网络的本质就是寻找一种非线性的变换函数。

Linear模型本身的函数是线性的（$X \cdot W$）而通过非线性的激活函数$\sigma(X \cdot W)$在每一层引入一层非线性因子从而去拟合真正需要的非线性模型。

导入数据集

```
import numpy as np
xy = np.loadtxt("diabetes.csv.gz", delimiter=", ", dtype=np.float32)
x_data = torch.from_numpy(xy[:, :-1])
y_data = torch.from_numpy(xy[:, [-1]])
```

设计模型

```
import torch

class Model(torch.nn.Module):
    def __init__(self):
        super(Model, self).__init__()
        self.linear1 = torch.nn.Linear(8, 6)
        self.linear2 = torch.nn.Linear(6, 4)
        self.linear3 = torch.nn.Linear(4, 1)
        self.sigmoid = torch.nn.Sigmoid()

    def forward(self, x):
        x = self.sigmoid(self.linear1(x))
        x = self.sigmoid(self.linear2(x))
        x = self.sigmoid(self.linear3(x))
        return x


model = Model()

# 构造损失函数和优化器
criterion = torch.nn.BCELoss(size_average = True)
optimizer = torch.optim.SGD(model.parameters(), lr=0.1)
```

训练模型

```
for epoch in range(100):
    # Forward
    loss = criterion(module(x_data),y_data)
    # Backward
    optimizer.zero_grad()
    loss.backward()
    # Upadate
    optimizer.step
```