常见损失函数的导数及解读

8 分钟前

武辰
数学话题下的优秀答主
关注

前置知识

对于 $f(x)=sigmoid(x)=\frac{1}{1+e^{-x}}$

有 $f'(x)=-\frac{1}{(1+e^{-x})^2}e^{-x}=f(x)(1-f(x))$

MSE

记 $loss=(y-ypred)^2$

则

$$\frac{\partial loss}{\partial w}=\frac{\partial loss}{\partial ypred}\cdot\frac{\partial ypred}{\partial w} \\ =2(y-ypred)\frac{\partial ypred}{\partial w}$$

如果最后一层是sigmoid层

即 $ypred=\frac{1}{1+e^{-wx}}$

则

$$\frac{\partial loss}{\partial w}=2(y-ypred)\frac{\partial ypred}{\partial w} \\ =2(y-ypred)ypred(1-ypred)(x)$$

这里的x是最后一层（sigmoid层）的输入

如何解读这个结果?

对该式子稍加观察，如果ypred接近1或者接近0，那么导数值 $\frac{\partial loss}{\partial w}$ 会接近0，导致梯度更新很慢，很难有效学习。因此，MSE不适合作为logistic回归问题的损失函数。（比如ytrue=1，但是ypred=0，这个时候梯度很小）

如果最后一层是全连接层

不考虑relu的情况下，

$$ypred=wx$$

那么

\frac{\partial loss}{\partial w} =2(y - ypred)\frac{\partial ypred}{\partial w}\\ = 2(y - ypred)x

这里的x是最后一层（全连接层）的输入

如何解读这个结果？好像平平无奇，找不出可以解读的角度。唯一可能可解读的点是当y和ypred很接近的时候，梯度接近0，参数将会更新得慢一些。这是符合直觉的。

Cross Entropy

loss=-(ylog(ypred)+(1-y)log(1-ypred))

则

\frac{\partial loss}{\partial w} = \frac{\partial loss}{\partial ypred} \cdot\frac{\partial ypred}{\partial w} \\= (\frac{-y}{ypred}+\frac{1-y}{1-ypred})\cdot\frac{\partial ypred}{\partial w}

如果最后一层是sigmoid层

即 ypred=\frac{1}{1+e^{-wx}}

则 \frac{\partial loss}{\partial w}=(\frac{-y}{ypred}+\frac{1-y}{1-ypred})\cdot\frac{\partial ypred}{\partial w} \\=(\frac{-y}{ypred}+\frac{1-y}{1-ypred})ypred(1-ypred)(x)\\=(ypred-y)x

这个结果可以怎么解读？它和【最后一层是MLP层且损失函数是MSE】的情况（比如对连续值的线性回归）很类似，但是可能仅仅是巧合而已。当y和ypred很接近的时候，梯度接近0，参数将会更新得慢一些。这是符合直觉的。

由这个结果可以引出一个思考：

考虑如果将神经网络全0初始化，网络真的学不到东西吗？假如神经网络只有一个sigmoid层（该网络等价于logistic回归），即使所有权重全0初始化，网络也能学到东西。因为梯度值是 (ypred-y)x ，它未必是0.首先，由于全部w是0，那么wx=0，经过sigmoid后，ypred=0.5。而y非0即1，因此(ypred-y)不可能是0.由于输入输出有很多个，因此x不可能总是0.综合来看，logistic回归可以有效学习。

假如该神经网络有很多个全连接层，每一层的激活函数都是sigmoid层，模型权重全0初始化，仍然可以学到东西吗?

答案是可以，但学的不多。首先，由刚刚的分析可得，最后一层的参数是可以正常更新的。

然后接下来考虑倒数第二层参数的梯度，既然倒数第一层的参数可以正常更新，建立在这个前提下，把倒数第一层拿掉，倒数第二层就相当于倒数第一层了。回到了刚刚说的情况。这样递推下去，所有的参数都可以更新。

但是，回到梯度的式子： \frac{\partial loss}{\partial w}=(ypred-y)x

如果 $x=[x_1,x_2,...,x_n]$ 中各个分量相等，那么w的各个分量更新的梯度也相等。由于w的初始值都相等，因此更新后的w的各个分量也都相等。

如果神经网络只有一层sigmoid，由于输入数据一般有多样性，不太可能每个分量都相等，因此w的各个分量的更新梯度不相等，整个神经网络可以正常更新梯度值。

但是如果神经网络有多层sigmoid，由于参数都初始化为0，这会导致中间每层的激活值的各个分量都相等。举个例子，考虑第二层的输出 $h_2=sigmoid(w_1h_1)$ ，这里的 h_2,w_1,h_1 都是向量， h_1 是第一层的输出。由于 w_1 是全0组成的向量，因此 h_2 是全部为0.5组成的向量。以此类推，每一层的激活值都相等，这会导致每一层的w的更新梯度都相等，最终导致对于同一层的 $w=[w_1,w_2,...,w_n]$,有 $w_1=w_2=...=w_n$ ，表面上有n参数，实际上只有1个参数。所以神经网络可以进行梯度更新，但是效果很差，相当于每一个神经元只有一个参数。

扩展：如果所有参数不是全0初始化，而是都初始化成某个固定的常数，也是同样的结果。因为这包含了“每一层的w的各个分量都相等”的情况。

接下来是代码验证，大家可以改改参数，试试不同的情况。代码中数据集的分类标签是根据一条直线来划分的。这是可以被logistic回归学习到的。

```
import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt

# 定义一个简单的二分类模型
class SimpleBinaryClassifier(nn.Module):
    def __init__(self):
        super(SimpleBinaryClassifier, self)._
        self.linear = nn.Linear(num_features,
        self.linear2 = nn.Linear(num_features
        nn.init.constant_(self.linear.weight,
        nn.init.constant_(self.linear2.weight

    def forward(self, y):
        y = self.linear(y)
        y = torch.sigmoid(y)
        y = self.linear2(y)
        return torch.sigmoid(y)

# 创建简单的训练数据集
num_samples = 10000
num_features = 2
input_features = torch.randn(num_samples, num
labels = (input_features[:, 1] > (2*input_fea

# 创建简单的测试数据集
test_num_samples = 20
test_input_features = torch.randn(test_num_sa
test_labels = (test_input_features[:, 1] > (2

# 初始化模型、损失函数和优化器
model = SimpleBinaryClassifier()
criterion = nn.BCELoss()
optimizer = optim.SGD(model.parameters(), lr=

# 训练模型
num_epochs = 1000

for epoch in range(num_epochs):
    outputs = model(input_features)
    loss = criterion(outputs, labels)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
    # 打印损失值
    print(f"Epoch [{epoch+1}/{num_epochs}], L

for name,param in model.named_parameters():
        print(name,param)

# 测试模型
model.eval()
test_outputs = model(test_input_features)
test_predictions = (test_outputs > 0.5).float
accuracy = (test_predictions == test_labels).
print(f"Test Accuracy: {accuracy:.2f}")

import numpy as np

# 可视化测试数据集
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=
ticks = np.arange(-3, 3, 0.5)
xtrue = np.linspace(-2.5, 2.5, num=200)
ytrue = 2 * xtrue

ax1.scatter(test_input_features[:, 0], test_i
            marker='o', c=test_labels.flatten
ax1.set_title("True Labels")
ax1.grid(True)    # 添加网格线
ax1.plot(xtrue, ytrue)
ax1.xticks=ticks
ax1.set_aspect("equal")  # 设置坐标轴的长宽比为

ax2.scatter(test_input_features[:, 0], test_i
ax2.set_title("Predicted Labels")
ax2.grid(True)    # 添加网格线
ax2.plot(xtrue, ytrue)
ax2.xticks=ticks
ax2.set_aspect("equal")  # 设置坐标轴的长宽比为

plt.show()
```

如果最后一层是全连接层

不考虑relu的情况下，

$$ypred=wx$$

那么

$$\frac{\partial loss}{\partial w}=(\frac{-y}{ypred}+\frac{1-y}{1-ypred})\cdot\frac{\partial ypred}{\partial w} \\ =(\frac{-y}{ypred}+\frac{1-y}{1-ypred})x \\ =\frac{ypred-y}{yrepd(1-ypred)}x$$

这个式子怎么解读？当y和ypred很接近的时候，梯度接近0，参数将会更新得慢一些。这是正常的。但是看分母，当ypred接近0或者接近1的时候，梯度值会变大。这不利于学习。

换一个角度来解释：最后一层是全连接层而不是sigmoid，相当于计算交叉熵loss的时候，没有做softmax操作（二分类中，softmax和sigmoid等价）。没有做softmax操作的交叉熵loss，表现自然是不如加了softmax操作的交叉熵loss。因为交叉熵loss有一个假设，就是参与计算是y和ypred是一个概率分布。加了softmax，可以保证ypred是一个概率分布。但是如果没有加softmax，ypred可能无法表示一个概率分布。前提假设不满足，交叉熵自然不能发挥出应有的功效。

编辑于 2024-06-14 23:10 · IP 属地广东