全连接神经网络02

一、使用全连接神经网络实现回归

Python

```python
import matplotlib.pyplot as plt
import numpy as np
import torch
import tqdm
from torch import nn


class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(1, 512)
        self.relu1 = nn.ReLU()
        self.fc2 = nn.Linear(512, 256)
        self.relu2 = nn.ReLU()
        self.fc3 = nn.Linear(256, 1)

    def forward(self, x):
        x = self.relu1(self.fc1(x))
        x = self.relu2(self.fc2(x))
        out = self.fc3(x)
        return out


if __name__ == '__main__':
    # 种子是一个整数，用于初始化随机数生成器。相同的种子将导致生成相同的随机数序列。
    # 通过设置种子，你可以确保每次运行代码时都能得到相同的随机数，
    # 这有助于实验的可重复性
    # 准备数据
    np.random.seed(42)
    X = np.random.rand(100, 1) * 10
    # 100, 1
    Y = np.sin(X) + 0.1 * np.random.randn(100, 1)
    x_tensor = torch.tensor(X, dtype=torch.float32)
    y_tensor = torch.tensor(Y, dtype=torch.float32)
    # 100, 1
    X_test = torch.FloatTensor(np.linspace(0, 10, 100).reshape(-1, 1))
    # 训练
    # 模型
    # 损失函数
    # 学习率
    epochs = 3000
    learning_rate = 0.001
    net = Net()
    loss_fn = nn.MSELoss()
    for epoch in tqdm.tqdm(range(epochs), total=epochs):
```

```
        out = net(x_tensor)
        loss = loss_fn(out, y_tensor)
        # 梯度清零
        net.zero_grad()
        # 反向传播
        loss.backward()
        # 更新参数
        for param in net.parameters():
            param.data -= learning_rate * param.grad

    with torch.no_grad():
        pred_out = net(X_test)

    # 绘制效果图
    plt.plot(X, Y, ".")
    plt.plot(X_test, pred_out)
    plt.show()
```

`tqdm` 是一个快速、可扩展的进度条，用于 Python 长循环和其他迭代器，可以帮助开发者在终端中显示操作的进度。

> conda install tqdm

二、参数和超参数

在机器学习和深度学习中，模型参数（parameters）和超参数（hyperparameters）是两个不同的概念，它们各自在模型训练过程中扮演着不同的角色。

2.1 模型参数（Parameters）

模型参数是指模型内部的可学习变量，这些变量是在训练过程中通过优化算法（如梯度下降）自动调整的。这些参数定义了模型的具体形式，并决定了模型如何映射输入到输出。例如，在神经网络中，权重矩阵（weights）和偏置项（biases）就是模型参数。

示例：

- 在线性回归中，模型参数通常是斜率 mm 和截距 bb。
- 在神经网络中，每层的权重矩阵 WW 和偏置向量 bb 都是模型参数。

2.2 超参数（Hyperparameters）

超参数是指在训练模型之前需要手动设定的参数，它们控制了模型的学习过程，但不是模型的一部分。超参数的选择会影响模型的性能，但它们不会通过训练过程被改变。选择合适的超参数对于获得好的模型性能至关重要。

示例：

- 学习率（Learning Rate）：控制每次更新参数时步长的大小。
- 批量大小（Batch Size）：每次迭代时使用的样本数量。
- 迭代次数（Number of Epochs）：整个训练集被遍历的次数。
- 正则化参数（Regularization Parameter）：用于控制正则化强度的参数，如 L1 或 L2 正则化中的 lambda。
- 神经网络层数（Number of Layers）和每层的节点数（Number of Nodes per Layer）。

2.3 区别

- **可学习性**：模型参数是在训练过程中自动学习得到的；而超参数是由用户根据经验和实验手动设置的。
- **影响范围**：模型参数直接影响模型的预测能力；超参数影响训练过程本身，间接影响模型的性能。
- **调整方式**：模型参数通过反向传播和优化算法调整；超参数通常通过网格搜索（Grid Search）、随机搜索（Random Search）或贝叶斯优化（Bayesian Optimization）等方法进行调优。

2.4 举例说明

假设你正在训练一个简单的多层感知器（MLP）：

模型参数

- 权重矩阵 W 和偏置向量 b。

超参数

- 学习率：0.01
- 批量大小：32
- 迭代次数：100
- 隐藏层节点数：64
- 正则化参数：0.001

在训练过程中，你会通过反向传播调整权重矩阵和偏置向量，以最小化损失函数。而在开始训练之前，你需要选择上述超参数的值，这些值会影响训练过程和最终模型的性能。

正确选择超参数是一项挑战性的任务，通常需要通过多次试验来确定最佳设置。现代的机器学习框架提供了多种工具和技术来帮助自动调整超参数，从而提高模型性能。

三、参数个数

3.1 模型参数的计算

以一个隐藏层为例：该隐藏层有3个神经元， 接收3个特征数据，每个神经元的参数为：4个（w1,w2,w3,b1），所以一共用3x4=12个参数。

计算公式如下：

> 输入神经元个数 * 输出神经元个数 + 输出神经元个数(偏置)

3.2 查看参数

`torchsummary` 是一个非常有用的库，可以让你快速地总结 PyTorch 模型的结构，包括每层的输出形状、参数数量等信息。这可以帮助你更好地理解模型架构并检查是否有任何维度上的错误。

Python

```
pip install torchsummary -i https://mirrors.aliyun.com/pypi/simple/

from torchsummary import summary

summary(模型, input_size=(输入特征数,), batch_size=批次大小,
            device="cpu")
```

四、损失函数

在深度学习中,损失函数是用来衡量模型参数的质量的函数,衡量的方式是比较网络输出和真实输出的差异,
损失函数在不同的文献中名称是不一样的,主要有以下几种命名方式

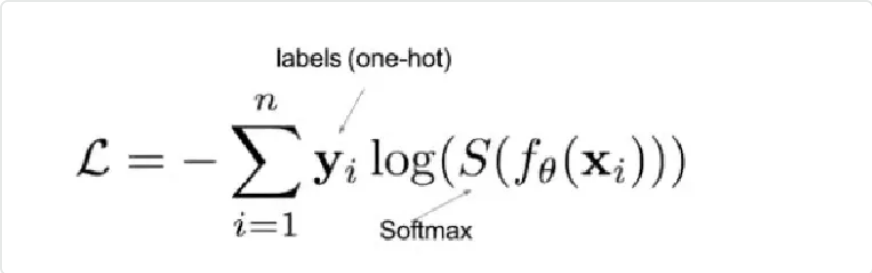
4.1 损失函数分类

根据不同的训练任务，需要使用不同的损失函数

- 多分类损失函数
- 二分类损失函数
- 回归任务损失函数

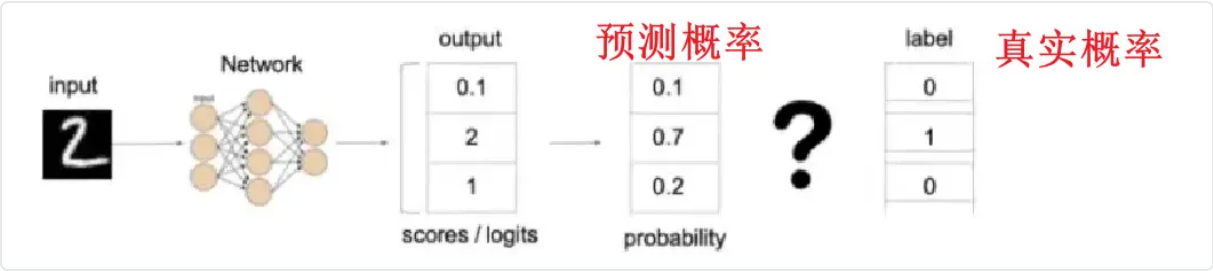
4.2 多分类交叉熵损失函数

在多分类任务通常使用softmax将logits转换为概率的形式，所以多分类的交叉熵损失也叫做softmax损失，它的计算方法是：



其中，y是样本x属于某一个类别的真实概率，而f(x)是样本属于某一类别的预测分数，S是softmax函数，L用来衡量p,q之间差异性的损失结果。

比如：



上面例子中的交叉熵损失为：

$-(0\log(0.10) + 1\log(0.7) + 0\log(0.2)) = -\log 0.7$

在PyTorch中，`CrossEntropyLoss` 和 `NLLLoss`（负对数似然损失）是两个常用的损失函数，它们都用于分类任务，特别是多类分类。不过，它们之间存在一些重要的区别：

1. CrossEntropyLoss

- `CrossEntropyLoss` 实际上是 `log_softmax` 和 `NLLLoss` 的组合。它接受未归一化的输出（通常称为“分数”或“原生得分”），然后内部应用 `log_softmax`，再计算 `NLLLoss`。
- 输入的张量形状也是 `(N, C)`，但是这里的值不是对数概率，而是每个类别的原始分数。`CrossEntropyLoss` 内部会将这些分数转换为对数概率，然后进行损失计算。
- 标签的格式与 `NLLLoss` 相同，是一个包含真实类别索引的 `(N)` 张量。

2. NLLLoss (Negative Log Likelihood Loss)

- `NLLLoss` 接受的是对数概率作为输入，并且它假设这些对数概率已经通过 `log_softmax` 函数进行了处理。这意味着输入应该是对数空间中的值，即每个样本的每个类别都有一个对数值表示其概率。
- 它的输入是一个形状为 `(N, C)` 的张量，其中 `N` 是批量大小，`C` 是类别数量。每个元素都是相应类别的对数概率。
- 标签是一个形状为 `(N)` 的张量，包含每个样本的真实类别索引。

3. 使用场景

- 当你的模型最后一层使用了 `log_softmax` 并且输出是对数概率时，你应该使用 `NLLLoss`。
- 如果你的模型的最后一层没有激活函数，或者更常见的情况是直接从全连接层输出（线性变换），那么你应该使用 `CrossEntropyLoss`。

4. 代码演示

```python
import torch
import torch.nn as nn
import torch.nn.functional as F

# 假设我们有3个样本，每个样本有5个可能的类别
# logits = torch.randn(3, 5)
logits = torch.tensor([[-1.4925, -1.4033, 0.8390, -1.6440, 1.5562],
                       [2.0672, -0.3932, -1.2632, 0.6108, 1.3722],
                       [-1.5822, -0.0761, 0.3674, -1.7704, 0.1582]])
print(logits)
# 标签编码
targets = torch.tensor([0, 3, 4])  # 每个样本的目标类别

# 使用CrossEntropyLoss
criterion = nn.CrossEntropyLoss()
loss = criterion(logits, targets)
print("CrossEntropyLoss loss:", loss.item())

# 使用NLLLoss
log_probs = F.log_softmax(logits, dim=1)  # 应用log_softmax
nll_criterion = nn.NLLLoss()
nll_loss = nll_criterion(log_probs, targets)
print("NLLLoss loss:", nll_loss.item())

# 手动验证
a = torch.softmax(logits, dim=1)
print(a)
b = -((1 * F.math.log(0.0291) + 0 * F.math.log(0.0318) + 0 * F.math.log(0.2998) + 0 * F.math.log(
    0.0250) + 0 * F.math.log(0.6142)) +
      (0 * F.math.log(0.5396) + 0 * F.math.log(0.0461) + 0 * F.math.log(0.0193) + 1 * F.math.log(
          0.1258) + 0 * F.math.log(0.2693)) +
      (0 * F.math.log(0.0525) + 0 * F.math.log(0.2365) + 0 * F.math.log(0.3686) + 0 * F.math.log(
          0.0435) + 1 * F.math.log(0.2990)))

print(b, b / 3)
```

5. 总结：

根据以上的代码，我们总结如下：

- 真实值targets， 会自动的进行one-hot编码
- 预测值logits，会自动的使用softmax归一化
- 多分类交叉熵损失CrossEntropyLoss函数中自动调用了softmax
- 这里的损失函数默认计算的是n个样本的平均损失

```
reduction: str = 'mean'
```

4.3 二分类交叉熵损失函数

在处理二分类任务时，我们不在使用softmax激活函数，而是使用sigmoid激活函数，那损失函数也相应的进行调整，使用二分类的交叉熵损失函数：

$$L = -y \log \hat{y} - (1 - y) \log(1 - \hat{y})$$

其中，y是样本x属于某一个类别的真实概率，而y^是样本属于某一类别的预测概率，L用来衡量真实值与预测值之间差异性的损失结果。

在pytorch中实现时使用nn.BCELoss()或者nn.BCEWithLogitsLoss()。

> 二分类交叉熵损失函数就是多分类交叉熵损失函数的一种特例。

1. BCELoss

```python
import torch
import torch.nn as nn

# 三个样本的类别
targets = torch.tensor([0, 1, 0], dtype=torch.float32)
# 二分类中每个样本的预测值只有一个值
logits = torch.tensor([1.70, -0.38, 2.14])

# 实例化二分类交叉熵损失函数
loss = nn.BCELoss()

# 将logits结果交给sigmoid函数， 将输出结果变成0-1之间的数据
y_sigma_pred = torch.sigmoid(logits)
print(y_sigma_pred)

# 计算损失
loss_value = loss(y_sigma_pred, targets)
print(loss_value)
```

2. BCEWithLogitsLoss

BCEWithLogitsLoss不需要手动调用sigmoid函数了

```python
import torch
import torch.nn as nn

# 三个样本的类别
targets = torch.tensor([0, 1, 0], dtype=torch.float32)
# 二分类中每个样本的预测值只有一个值
logits = torch.tensor([1.70, -0.38, 2.14])

# 实例化二分类交叉熵损失函数
# 不需要手动调用sigmoid函数了
loss = nn.BCEWithLogitsLoss()
# 计算损失
loss_value = loss(logits, targets)
print(loss_value)

```

4.4 回归任务的损失函数

1. MAE损失函数

Mean absolute loss(MAE)也被称为L1 Loss，是以绝对误差作为距离：

$$\mathcal{L} = \frac{1}{n}\sum_{i=1}^{n}|y_i - f_\theta(x_i)|$$

Python

```python
import torch
import torch.nn as nn

y_true = torch.tensor([2.0, 3.2, 1.9])
y_pred = torch.tensor([2.1, 3.1, 1.8])

# 实例化MAE
loss = nn.L1Loss()

# 计算损失
loss_value = loss(y_pred, y_true)
print(loss_value)
```

2. MSE损失函数

Mean Squared Loss/ Quadratic Loss(MSE loss)也被称为L2 loss。

$$\mathcal{L} = \frac{1}{n}\sum_{i=1}^{n}(y_i - f_\theta(x_i))^2$$

> 当预测值与目标值相差很大时,梯度容易爆炸。

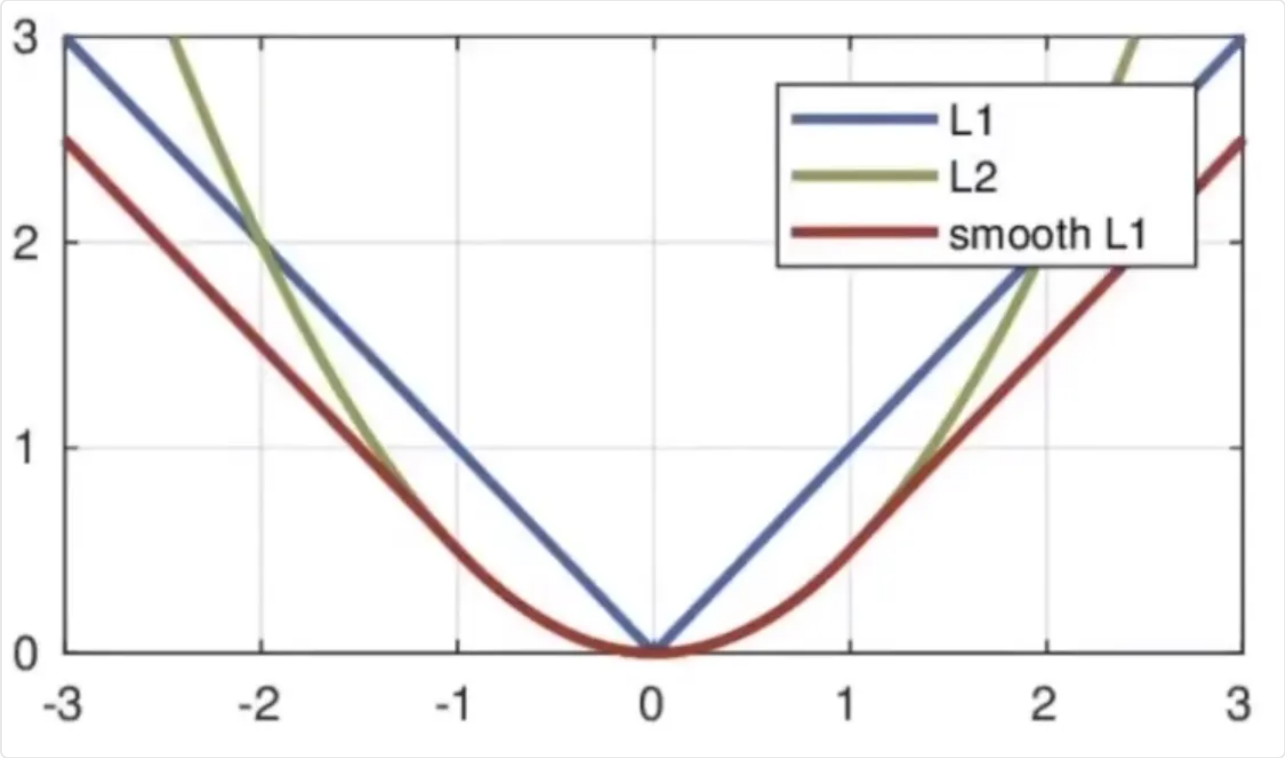
```python
import torch
import torch.nn as nn

y_true = torch.tensor([2.0, 3.2, 1.9])
y_pred = torch.tensor([2.1, 3.1, 1.8])

# 实例化MSE
loss = nn.MSELoss()

# 计算损失
loss_value = loss(y_pred, y_true)
print(loss_value)
```

3. smooth L1损失函数

`SmoothL1Loss` 是一种损失函数，通常用于机器学习中的回归任务。它结合了均方误差（MSE, Mean Squared Error）和平均绝对误差（MAE, Mean Absolute Error）的优点，试图在两者之间取得平衡。具体来说，当预测值与真实值之间的差异较小时，SmoothL1Loss 表现得像 MSE；而当差异较大时，则表现得更像 MAE。

$$\text{smooth}_{L_1}(x) = \begin{cases} 0.5x^2 & \text{if } |x| < 1 \\ |x| - 0.5 & \text{otherwise} \end{cases}$$

其中：x=f(x)−y 为真实值和预测值的差值。



从上图中可以看出，该函数实际上就是一个分段函数，在[–1,1]之间实际上就是L2损失，这样解决了L1的不光滑问题，在[–1,1]区间外，实际上就是L1损失，这样就解决了离群点梯度爆炸的问题。通常在目标检测中使用该损失函数。

- **缺点**：虽然 SmoothL1Loss 具有良好的平衡特性，但它引入了一个额外的超参数（β），这需要根据具体应用进行调整。

Python

```python
import torch
import torch.nn as nn

y_true = torch.tensor([1, 0])
y_pred = torch.tensor([0.4, 0.6])

# 实例化SmoothL1
loss = nn.SmoothL1Loss()

# 计算损失
loss_value = loss(y_pred, y_true)
print(loss_value)
```

五、手写数字识别

5.1 mnist数据集介绍

官网地址：http://yann.lecun.com/exdb/mnist/

5.2 pytorch中使用mnist数据集

Python

```python
from torchvision import datasets

if __name__ == '__main__':
    mnist = datasets.MNIST("./data", train=True, download=True)
    print(mnist)
    # mnist[0]是一个元组，第一个值是PIL图像， 第二个值是 标签
    print(mnist[0])
    print(mnist.classes)
    print(mnist.class_to_idx)
    print(mnist.targets)
    # train_labels过时了("train_labels has been renamed targets")
    print(mnist.train_labels)
```

5.3 手写数字识别代码

Python

```python
"""
使用pytorch搭建神经网络
加载数据    创建模型    训练    测试
"""
import torch
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
from torch import nn


def get_loader(is_train=True):
    """
      根据刚才的调式我们发现：
        1. MNIST中的数据是 HWC的数据，Pytorch中需要处理的图像是CHW
        2. MNIST中数据值是0-255，像数值差异比较大，可能导致梯度波动较大，
        影响训练的稳定性及收敛速度，所以我们需要做归一化处理成0-1之间的数据
        归一化后可以：
        1. 加速训练过程
        2. 数据归一化后可以帮助模型更好的泛化未见过的数据

        pytorch中可以使用 transform来对数据进行预处理:
        - HWC --> CHW
        - 0-255  --->  [0.0, 1.0]
    """
    mnist_set = datasets.MNIST(root='./data', train=is_train, download=True, transform=transforms.ToTensor())
    data_loader = DataLoader(dataset=mnist_set, batch_size=100, shuffle=is_train)
    return data_loader


class MLP(nn.Module):
    def __init__(self):
        super().__init__()
        self.layers = nn.Sequential(
            # 1 * 28 * 28
            # 第一层:  784 * 512
            nn.Linear(in_features=28 * 28, out_features=512), nn.ReLU(),
            nn.Linear(in_features=512, out_features=256), nn.ReLU(),
            nn.Linear(in_features=256, out_features=128), nn.ReLU(),
            nn.Linear(in_features=128, out_features=64), nn.ReLU(),
            nn.Linear(in_features=64, out_features=10), nn.ReLU(),
            nn.Softmax(dim=-1)
        )

```

```
    def forward(self, x):
        out_date = self.layers(x)
        return out_date


def train(epochs):
    # 学习率
    learn_rate = 0.01
    device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu'
)
    # 创建网络模型
    net = MLP().to(device)
    # 获取数据加载器
    train_data_loader = get_loader()
    # 定义损失函数
    loss_fn = nn.MSELoss()
    # 训练轮次
    for epoch in range(epochs):
        # 遍历数据加载器获取每个批次数据集的数据
        for idx, (images, labels) in enumerate(train_data_loader):
            images, labels = images.to(device), labels.to(device)
            # images的形状是： [100, 1, 28, 28]
            # labels的形状是： [100]
            # print(images.shape, labels.shape)
            # print(len(images), len(labels))
            # 全连接神经网络要求接受数据的形状是NV结构，也就是[每个批次的数量，特征向
量]
            images = images.reshape(-1, 1 * 28 * 28)
            # 将数据交给网络模型训练，得到每一种类别的概率值
            # out的形状: [100, 10]
            out = net(images)
            # 真实标签labels的形状: [100]
            # 使用预测的结果和真实的标签计算损失
            # 但是：out的形状和labels的形状不相同，而且out的结果是概率值，lables的
结果是真实标签
            # 所以需要将lables进行one-hot编码
            # loss = loss_fn(out, labels)
            labels = nn.functional.one_hot(labels, 10).float()
            loss = loss_fn(out, labels)
            net.zero_grad()
            loss.backward()
            for param in net.parameters():
                param.data -= learn_rate * param.grad
            if (idx + 1) % 100 == 0:
                print(f"train---> epoch:{epoch + 1}/{epochs}, loss:{loss.i
tem()}")
if __name__ == '__main__':
    # t = torch.Tensor(5, 28*28)
```

```
88      # net = MLP()
89      # out = net(t)
90      # print(out)
91      # loader = get_loader()
92      # print(len(loader))
93      train(500)
94
95
```