完整的模型训练套路

2025年12月2日 21:44

Python

```
# 你的代码（在第 25 行）：
super().__init__(self) #super(Net,self).__init__()
```

诊断：为什么会多出一个参数？

1. **Python 类的调用机制：** 当你执行 `model = Net()` 时：
   - Python 会自动调用 Net 类的 __init__ 方法。
   - 它会将**新创建的实例对象**作为**第一个参数**传递给 __init__，这个参数就是 `self`。
   - 所以，Net.__init__ **接收了**参数 `self`。
2. `super()` **的正确用法：**
   - `super().__init__()` **会自动**将当前实例（即 $self$）传递给父类（$torch.nn.Module$）的 __init__ 方法。
   - **你不需要再手动将** $self$ **传一次。**

当你写成 `super().__init__(self)` 时，实际上你是在告诉 Python：请将 `self` 这个参数作为第二个位置参数传给父类的构造函数（因为第一个参数 $self$ 已经隐式传递了）。

- $torch.nn.Module$'s __init__ **不接受任何参数**（除了隐式的 $self$）。
- 你传入了 `self`，导致它收到了两个参数（隐式的 $self$ 和你显式传入的 $self$），所以它报错说："我只需要 1 个参数（隐式的 $self$），但你给了我 2 个！"

```
    def forward(self, x):
        return self.model(x)

if __name__ == '__main__':#-----用于测试，且被调用时不会执行
    model = Net()
    input = torch.ones(64,3,32,32)
    output = model(input)
    print(output)
```

点这里就能运行了

```
#准备数据集 需要使用到官方提供的数据集，先引入torch visio
import torch
import torchvision
from torch import nn
from torch.utils.data import DataLoader
```

有时候 pycharm自动引入的是错误的，报错时，要注意排查

complete training process.py × model.py

```
#准备数据集 需要使用到官方提供的数据集，先引入torch vi
import torchvision

from torch.utils.data import DataLoader

from model import *
#训练数据
train_dataset = torchvision.datasets.CIFAR10(
#测试数据
```

在完整的训练过程中
最好把模型文件和训练文件分开单独放

最后再在训练文件中把模型引入

```
#优化器
learning_rate = 0.01 #-----1e-2
optimer = torch.optim.SGD(model.parameters(), lr=learning_rate)#---随

#设置训练网络的参数
# 记录训练的步数
total_train_step = 0
total_test_step = 0
#训练的轮数
epoch = 100

for i in range(epoch):
    print('---------第{}轮训练开始----------'.format(i+1))
```

把需要用到数字的地方，都先定义好

1. 主体：你搭建的 Net 网络框架

你通过 `self.model = nn.Sequential(...)` 定义了一系列具体的层（$nn.Conv2d$，$nn.Linear$ 等）。

- 当你在 Net 类的 __init__ 方法中实例化这些层时，例如 `nn.Conv2d(...)`，这些层内部就立刻创建了它们自己的可训练参数（权重 W 和偏置 b）。
- 这些参数（W, b）是以 $torch.nn.Parameter$ 对象的形式存在的。

所以，从**物理上看**，权重是存在于你定义的 Net 实例的各个子模块（$self.model$ 里的每个 $Conv2d$ 和 $Linear$）内部的。

2. 桥梁：继承的 $nn.Module$ 父类

$nn.Module$ 类是 PyTorch 中所有神经网络层的**基类（Base Class）**，它提供了管理参数和计算的功能。

$nn.Module$ 的核心作用在于：

- **自动注册（Registration）：** 当你在 __init__ 中将一个 $nn.Module$ 子类（如 $nn.Conv2d$ 或 $nn.Linear$）赋值给 $self$ 的一个属性时（例如 `self.model`），$nn.Module$ 的底层机制就会自动"发现"这个子模块及其包含的**所有参数**。
- **参数收集：** 正是由于 $nn.Module$ 提供的这个机制，当你调用 `model.parameters()` 时，它能递归地遍历你整个 Net 模型（包括 $self.model$ 里面的所有层），然后把**所有**可训练的权重和偏置都收集起来，返回给优化器。

权重什么的是不是都在继承的module类里，还是我搭建的网络框架

```
    output = model(img)
    loss = loss_fn(output,target)#损失函数，输出的结果 与 预期的标签
    #优化前梯度清零
    optimer.zero_grad()
    loss.backward()
    optimer.step()#----优化----
    total_train_step += 1 #记录步数
    if total_train_step % 100 == 0:#----不需要每次都打印
        print('训练次数: {},loss: {}'.format( *args: total_train_step,loss.item()))
        writer.add_scalar( tag: 'train_loss',loss.item(),total_train_step)
```

在循环内打印，就在循环内加

```
#进行测试\
total_test_loss = 0
with torch.no_grad():#-----测试不需要梯度
    for img ,target in test_loader:
        output = model(img)
        loss = loss_fn(output,target)
        total_test_loss += loss.item()
    print('整体测试集的loss: {}'.format(total_test_loss))#---每一轮 total_test_loss都会清零，不需要再除总数
    writer.add_scalar( tag: 'test_loss',total_test_loss,total_test_step)#---tota_test_step要加一
    total_test_loss += 1
```

在循环外打印，就在循环外加

虽然能看到loss是下降的
但不能很好地判断模型训练效果到底怎么样

如何反映正确率

2 x input

Model(2分类)

Outputs =
[0.1, 0.2]
[0.3, 0.4]

对于每个输入的图片，每个标签对应的概率

0. 1

Argmax 找出最大值对应的编号

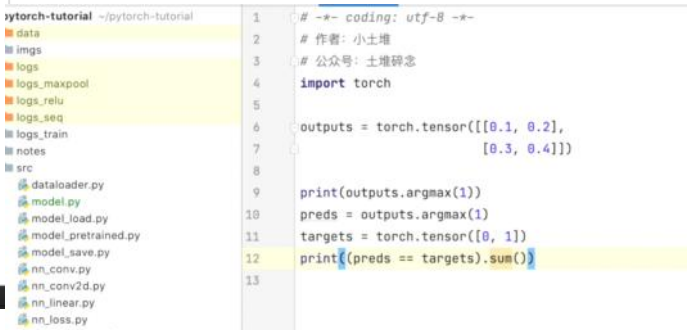
0 是沿着垂直方向（列）
1是水平方向（行）

Preds = [1]
[1]

Inputs target = [0][1]

Preds == inputs target 和预期比较一下，相等是true，不等是false

[false, true].sum() = 1 求和，然后再除以总输入个数，即可得到概率

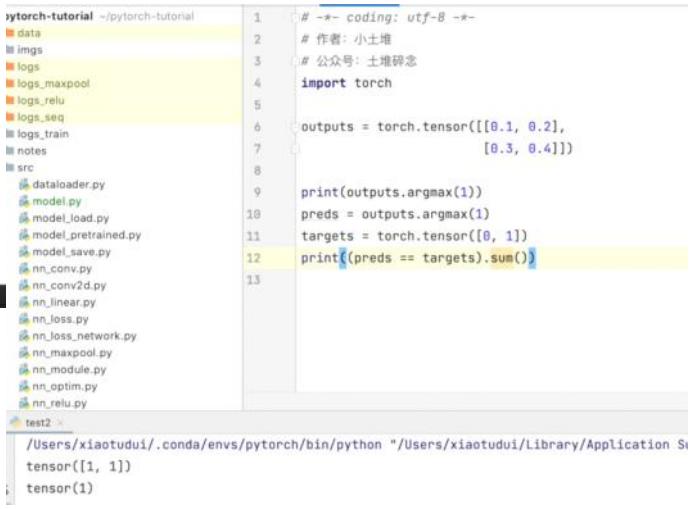


Preds = [1]
[1]

Inputs target = [0][1]

Preds == inputs target　　和预期比较一下，相等是true，不等是false

[false, true].sum() = 1　　求和，然后再除以总输入个数，即可得到概率




```
#训练开始
model.train()
for img ,target in train_loader:
    output = model(img)
    loss = loss_fn(output,target)#损失函数，输出的结果 与 预期的标签
    #优化前梯度清零
    optimer.zero_grad()
    loss.backward()
    optimer.step()#-----优化----
    total_train_step += 1 #记录步数
    if total_train_step % 100 == 0:#----不需要每次都打印
        print('训练次数: {},loss: {}'.format( *args: total_train_ste
        writer.add_scalar( tag: 'train_loss',loss.item(),total_tra


#进行测试
model.eval()
total_test_loss = 0
total_accuracy = 0
```

对特定的层有作用

使用GPU 训练

```
#创建网络模型
model = Net()
model = model.cuda()
```

记得要返回

```
#损失函数
loss_fn = nn.CrossEntropyLoss()#-----交叉
loss_fn = loss_fn.cuda()
```

网络模型

数据（输入，标注）

损失函数

.cuda()

这些类型可以使用CUDA

```
#训练开始
model.train()
for img ,target in train_loader:
    img = img.cuda()
    target = target.cuda()
    output = model(img)
```

优化

```
if torch.cuda.is_available():
    img = img.cuda()
    target = target.cuda()
```

```
#进行测试
model.eval()
total_test_loss = 0
total_accuracy = 0
with torch.no_grad():#-----测试不需要梯度
    for img ,target in test_loader:
        img = img.cuda()
        target = target.cuda()
        output = model(img)
        loss = loss_fn(output,target)
```