pytorch中的register_buffer方法详解

2 分钟前

武辰
数学话题下的优秀答主
关注

前置知识:state_dict

state_dict的基本概念

在PyTorch中，state_dict是一个Python字典对象，保存了一个模型的参数。具体来说，它存储了模型中所有层的权重和偏置等参数。如果模型中包含了优化器，那么优化器的state_dict也会包含关于优化器状态以及所使用的超参数信息。

一个模型的state_dict包含的内容通常如下：

1. **模型参数**：对于每一层（比如全连接层、卷积层、归一化层等）， `state_dict` 包含了它们的权重和偏置。

- 权重（weights）通常命名为 `layer_name.weight`
- 偏置（biases）通常命名为 `layer_name.bias`

举例来说：

```
import torch.nn as nn

class MyModel(nn.Module):
    def __init__(self):
        super(MyModel, self).__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.fc1 = nn.Linear(20, 10)

model = MyModel()
state_dict = model.state_dict()
for param_tensor in state_dict:
    print(param_tensor, "\t", state_dict[para
```

在这个例子中， `state_dict` 将包含如下结构（格式可能略有不同）：

```
conv1.weight     torch.Size([20, 1, 5, 5])
conv1.bias       torch.Size([20])
fc1.weight       torch.Size([10, 20])
fc1.bias         torch.Size([10])
```

2.优化器参数：如果使用了优化器，比如Adam或SGD，它的 `state_dict` 将会包括关于当前的动量（如果有）、学习率等状态信息。

优化器的 `state_dict` 通常包含如下信息：

- 学习率
- 动量
- 权重衰减
- 优化器的内部状态信息，这在像Adam这样的基于动量算法的优化器中尤其重要。

例如：

```
import torch.optim as optim

optimizer = optim.Adam(model.parameters(), lr
optimizer_state_dict = optimizer.state_dict()
for var_name in optimizer_state_dict:
    print(var_name, "\t", optimizer_state_dic
```

通过保存和加载 `state_dict` ，可以很方便地使模型接着之前的训练状态继续训练，或者用于模型的推理。这是因为 `state_dict` 是与模型的结构相独立的，只要目标模型的结构和待加载 `state_dict` 中的参数结构匹配，就可以加载这个 `state_dict` 。

在PyTorch中，模型的 `state_dict` 和优化器的 `state_dict` 是两个不同的概念，它们分别包含了不同类型的信息。

1. **模型的** `state_dict` ：它主要包含了模型各层的参数，如权重（weights）和偏置（biases）。这些参数是模型训练中学习得到的并且用于模型的前向传播过程。模型的 `state_dict` 通常用于保存模型的参数，以便于之后可以重新加载这些参数继续训练或者用于模型的评估。
2. **优化器的** `state_dict` ：优化器的 `state_dict` 则存储了所有优化器的状态信息以及所使用的超参数。例如，对于使用动量的优化器，它的 `state_dict` 会包含动量缓冲区等状态信息；对于使用自适应学习率算法的优化器（如Adam），它的 `state_dict` 会包含特定于算法的状态信息（如梯度的一阶矩和二阶矩估计）。这部分信息对于继续模型的训练是必要的，尤其是当训练需要被暂停然后重新启动的时候。

总结来说，模型的 `state_dict` 和优化器的 `state_dict` 是针对不同目的而设计的，它们结构上不相同，包含的信息也不一致，但它们都是为了方便模型训练状态的保存和恢复。通常，当需要保存一个完整的训练状态以便后续可以完全恢复时，你会同时保存模型的 `state_dict` 和优化器的 `state_dict` 。

一个具体的例子

接下来，我训练一个简单的神经网络，来讲解state_dict保存的内容。

```
import torch
import torch.nn as nn
import torch.optim as optim

# 定义一个简单的神经网络模型
class SimpleModel(nn.Module):
    def __init__(self):
        super(SimpleModel, self).__init__()
        self.fc = nn.Linear(3, 2)

    def forward(self, x):
        return self.fc(x)

# 实例化模型
model = SimpleModel()

# 定义一个简单的优化器，使用SGD
optimizer = optim.Adam(model.parameters(), lr

# 定义损失函数
criterion = nn.MSELoss()

# 模拟一个小批次训练数据
batch_size = 32
inputs = torch.randn(batch_size, 3)
targets = torch.randn(batch_size, 2)

# 简单训练循环
for epoch in range(1):  # 假设我们只训练一个epoc
    for i in range(batch_size):
        # 获取批次数据
        input_sample = inputs[i].unsqueeze(0)
        target_sample = targets[i].unsqueeze(

        # 前向传播
        outputs = model(input_sample)
        loss = criterion(outputs, target_samp

        # 反向传播和优化
        optimizer.zero_grad()  # 清零梯度
        loss.backward()  # 反向传播
        optimizer.step()  # 更新模型参数

# 保存模型和优化器的state_dict
torch.save({
    'model_state_dict': model.state_dict(),
    'optimizer_state_dict': optimizer.state_d
}, 'model_and_optimizer.pth')

# 加载模型和优化器的state_dict
checkpoint = torch.load('model_and_optimizer.
model.load_state_dict(checkpoint['model_state
optimizer.load_state_dict(checkpoint['optimiz

# 进入评估模式
model.eval()

# ... 在这里进行评估/测试

# 如果要继续训练，切换回训练模式
# model.train()

# ... 继续执行训练过程
```

注：adam的梯度更新公式如下：

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$
$$w = w - learning\_rate \cdot \frac{\frac{m_t}{1-\beta_1^t}}{\sqrt{\frac{v_t}{1-\beta_2^t}}}$$

打印model.state_dict()：

```
OrderedDict([('fc.weight',
              tensor([[-0.4955,  0.0081,  0.2
                      [-0.3830, -0.3976, -0.1
             ('fc.bias', tensor([-0.0065, -0.
```

打印optimizer.state_dict()：

```
{'state': {0: {'step': tensor(32.),
   'exp_avg': tensor([[-0.3188,  0.0248,  0.6
           [-0.1113, -0.6170, -0.8932]]),
   'exp_avg_sq': tensor([[0.0793, 0.0193, 0.0
           [0.0335, 0.0321, 0.0961]])},
  1: {'step': tensor(32.),
   'exp_avg': tensor([ 0.4341, -0.3487]),
   'exp_avg_sq': tensor([0.0473, 0.0430])}},
 'param_groups': [{'lr': 0.01,
   'betas': (0.9, 0.999),
   'eps': 1e-08,
   'weight_decay': 0,
   'amsgrad': False,
   'maximize': False,
   'foreach': None,
   'capturable': False,
   'params': [0, 1]}]}
```

可以看到它包括state和param_groups两个部分的内容。

`state` ：这个字典保存了每个参数的优化器状态。每个参数（由其Tensor ID标识）具有以下状态：

- `step` ：参数更新的次数。在这里，参数的更新步数是32，表示优化器执行了32次参数更新。
- `exp_avg` ：梯度的指数移动平均（即动量），用于计算参数更新。这个移动平均是基于first moment（即一阶矩）的。
- `exp_avg_sq` ：梯度平方的指数移动平均，用于自适应学习率的计算。这个移动平均是基于second moment（即二阶矩）的。

`param_groups` ：包含了优化器参数组的列表。每个参数组字典包含了以下字段：

- `lr` ：学习率，这里是0.01。
- `betas` ：用于计算 `exp_avg` 和 `exp_avg_sq` 的系数，对应Adam优化器的 `(beta1, beta2)` ，这里是(0.9, 0.999)。
- `eps` ：为了提高数值稳定性而加到分母里的小常数ε，这里是1e-8。
- `weight_decay` ：权重衰减（L2正则化），这里是0。
- `amsgrad` ：是否使用AMSGrad变种，这里是False。
- `maximize` ：优化目标是最大化还是最小化，这里是False（一般用于最小化）。
- `foreach` ：指定是否使用循环优化每个参数，默认是None。
- `capturable` ：与进行模型捕获相关的标志，默认是False。
- `params` ：包含参数Tensor ID的列表，这里有两个参数（因为模型是一个具有两个参数的小型网络，包括一个权重和一个偏置）。

这个状态信息对于优化器恢复是非常重要的，尤其在暂停和恢复训练时，因为使用Adam算法时，梯度的历史信息（动量和平方动量）对于决定未来的参数更新至关重要。通过加载这个状态信息，您可以精确地恢复训练过程，继续之前的训练而不会丢失历史梯度信息。

requires_grad = False的参数是否会被优化器跟踪?

打印optimizer.state_dict()的结果中，可以看到state有两个部分的内容："0"和"1"。0代表weight参数，1代表bias参数。

如果把bias的参数设置为不需要更新梯度：

```
for name,param in model.named_parameters():
    if 'bias' in name:
        param.requires_grad = False
```

我们会发现optimizer.state_dict()不会跟踪bias的信息，即state只有"0"。

在PyTorch中，当我们使用model.parameters()来检索模型的参数传递给优化器时，实际上所有的参数都会被传递，不管它们的requires_grad属性是什么值。然而，requires_grad属性确实决定了是否在这些参数上计算梯度。如果参数设置为requires_grad=False，即使这个参数被传递给了优化器，也不会有梯度计算，因此也不会有更新步骤发生。

register_buffer

register_buffer是什么?

在PyTorch中，register_buffer是Module类中的一个方法，它用于注册一个不需要梯度的缓冲区。这在当你有模型中需要持续跟踪但不需要计算梯度的参数时非常有用。例如，批量归一化（BatchNorm）层会跟踪运行时均值和方差的统计数据，这些统计数据在训练过程中更新但是在反向传播中不需要计算梯度。

使用register_buffer，你可以确保这些张量在模型的state_dict中正确存储和加载，且会自动地移动到设备（CPU或GPU），如同其他参数一样。这对于模型保存和加载、迁移到不同的设备等场景非常重要。

注册一个缓冲区的基本语法如下：

```
self.register_buffer('buffer_name', torch.ten
```

这里，self是继承自nn.Module的类的一个实例，buffer_name是你想要给缓冲区的名称，它将会成为模块属性的一部分，并且torch.tensor(...)是初始的缓冲区数据。

一个具体的例子

下面是一个简单的使用register_buffer的例子，其中包含了一个简单的模型与一轮的训练过程，并显示了state_dict中的值。

这个例子中，我们将定义一个简单的nn.Module网络，其中包含一个running_mean缓冲区。这个缓冲区不会在反向传播中更新梯度，但我们会在每次前向传递时更新它。

```
import torch
import torch.nn as nn
import torch.optim as optim

# 简单模型 - 只有一个全连接层
class SimpleModel(nn.Module):
    def __init__(self, input_size, output_siz
        super(SimpleModel, self).__init__()
        self.fc = nn.Linear(input_size, outpu
        # 注册一个没有梯度的缓冲区
        self.register_buffer('running_mean',

    def forward(self, x):
        # 模拟更新运行平均值
        self.running_mean = 0.9 * self.runnin
        return self.fc(x)

# 输入和输出大小
input_size = 3
output_size = 2

# 创建模型实例
model = SimpleModel(input_size, output_size)

# 定义损失函数
criterion = nn.MSELoss()

# 定义优化器
optimizer = optim.Adam(model.parameters(), lr

# 生成一些随机数据作为输入和目标输出
x = torch.randn(32, input_size)  # 随机输入
y_target = torch.randn(32, output_size)  # 随

# 前向传播
y_pred = model(x)

# 计算损失
loss = criterion(y_pred, y_target)

# 清除之前的梯度
optimizer.zero_grad()

# 反向传播
loss.backward()

# 更新模型参数
optimizer.step()

# 打印 running_mean 缓冲区的值
print("\n=== Updated running_mean\n:", model.

# 打印 state_dict
print("\n=== model's state_dict()\n:",model.s
print("\n=== optimizer.state_dict()\n",optimi
```

这份代码的输出是：

```
=== Updated running_mean
: tensor([ 0.0003, -0.0031, -0.0046])

=== model's state_dict()
: OrderedDict([('running_mean', tensor([ 0.00
        [-0.3007,  0.3960,  0.5304]])), ('fc.

=== optimizer.state_dict()
 {'state': {0: {'step': tensor(1.), 'exp_avg'
        [-0.0328,  0.0489,  0.0520]]), 'exp_a
        [1.0748e-04, 2.3911e-04, 2.7044e-04]]
```

可以看到running_mean被model的state_dict追踪。state_dict在保存和加载模型时非常重要。比如说，对于模型推理（inference）阶段，特别是使用Batch Normalization层的模型，需要包含batch normalization层的运行均值（running mean）和方差（running variance）。这些统计数据在训练阶段被计算并通过register_buffer注册以便作为模型的一部分保存下来。如果在推理阶段没有这些统计数据，Batch Normalization层将无法正常工作，因为它不知道应该使用什么均值和方差来归一化输入数据。结果，模型的预测性能很可能会受到严重影响。

但是register_buffer注册的数据不会被优化器的state_dict追踪。register_buffer注册的缓冲区专门用于模型内部的中间数据，它们是模型的一部分，但不是模型参数（不像nn.Parameter），因此不需要梯度更新。

总结：register_buffer的作用是用于注册模型中那些不需要梯度更新的参数（比如batch normalization层中的运行均值和方差），同时确保这些参数被包括在模型的state_dict中。

编辑于 2024-06-14 22:29・IP 属地广东