PyTorch 中模型的使用


赵剑行
手可摘星辰

神经网络训练后我们需要将模型进行保存，要用的时候将保存的模型进行加载，PyTorch 中保存和加载模型主要分为两类：保存加载整个模型和只保存加载模型参数。

知乎 首发于 PyTorch 的那些事

一、保存加载模型基本用法

1、保存加载整个模型

保存整个网络模型（网络结构+权重参数）。

```
torch.save(model, 'net.pkl')
```

直接加载整个网络模型（可能比较耗时）。

```
model = torch.load('net.pkl')
```

2、只保存加载模型参数

只保存模型的权重参数（速度快，占内存少）。

```
torch.save(model.state_dict(), 'net_params.pkl')
```

因为我们只保存了模型的参数，所以需要先定义一个网络对象，然后再加载模型参数。

```
# 构建一个网络结构
model = ClassNet()
# 将模型参数加载到新模型中
state_dict = torch.load('net_params.pkl')
model.load_state_dict(state_dict)
```

保存模型进行推理测试时，只需保存训练好的模型的权重参数，即推荐第二种方法。

主要用法就是上面这些，接下来讲一下PyTorch中保存加载模型内部的一些原理，以及我们可能会遇到的一些特殊的需求。

二、保存加载自定义模型

上面保存加载的 `net.pkl` 其实一个字典，通常包含如下内容：

1. **网络结构**：输入尺寸、输出尺寸以及隐藏层信息，以便能够在加载时重建模型。
2. **模型的权重参数**：包含各网络层训练后的可学习参数，可以在模型实例上调用 `state_dict()` 方法来获取，比如前面介绍只保存模型权重参数时用到的 `model.state_dict()` 。
3. **优化器参数**：有时保存模型的参数需要稍后接着训练，那么就必须保存优化器的状态和所其使用的超参数，也是在优化器实例上调用 `state_dict()` 方法来获取这些参数。
4. 其他信息：有时我们需要保存一些其他的信息，比如 `epoch` ， `batch_size` 等超参数。

知道了这些，那么我们就可以自定义需要保存的内容，比如：

```
# saving a checkpoint assuming the network class named ClassNet
checkpoint = {'model': ClassNet(),
              'model_state_dict': model.state_dict(),
              'optimizer_state_dict': optimizer.state_dict(),
              'epoch': epoch}

torch.save(checkpoint, 'checkpoint.pkl')
```

上面的 checkpoint 是个字典，里面有4个键值对，分别表示网络模型的不同信息。

然后我们要加载上面保存的自定义的模型：

```
def load_checkpoint(filepath):
    checkpoint = torch.load(filepath)
    model = checkpoint['model']  # 提取网络结构
    model.load_state_dict(checkpoint['model_state_dict'])  # 加载网络权重参数
```

```
    optimizer = TheOptimizerClass()
    optimizer.load_state_dict(checkpoint['optimizer_state_dict'])  # 加载优化器参数

    for parameter in model.parameters():
        parameter.requires_grad = False
    model.eval()

    return model

model = load_checkpoint('checkpoint.pkl')
```

如果加载模型只是为了进行推理测试，则将每一层的 `requires_grad` 置为 `False`，即固定这些权重参数；还需要调用 `model.eval()` 将模型置为测试模式，主要是将 `dropout` 和 `batch normalization` 层进行固定，否则模型的预测结果每次都会不同。

如果希望继续训练，则调用 `model.train()`，以确保网络模型处于训练模式。

`state_dict()` 也是一个Python字典对象， `model.state_dict()` 将每一层的可学习参数映射为参数矩阵，其中只包含具有可学习参数的层(卷积层、全连接层等)。

比如下面这个例子：

```
# Define model
class TheModelClass(nn.Module):
    def __init__(self):
        super(TheModelClass, self).__init__()
        self.conv1 = nn.Conv2d(3, 8, 5)
        self.bn = nn.BatchNorm2d(8)
        self.conv2 = nn.Conv2d(8, 16, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.bn(x)
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

    # Initialize model
    model = TheModelClass()

    # Initialize optimizer
    optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9)
```

```
    print("Model's state_dict:")
    for param_tensor in model.state_dict():
        print(param_tensor, "\t", model.state_dict()[param_tensor].size())

    print("Optimizer's state_dict:")
    for var_name in optimizer.state_dict():
        print(var_name, "\t", optimizer.state_dict()[var_name])
```

输出为：

```
Model's state_dict:
conv1.weight            torch.Size([8, 3, 5, 5])
conv1.bias              torch.Size([8])
bn.weight               torch.Size([8])
bn.bias                 torch.Size([8])
bn.running_mean         torch.Size([8])
bn.running_var          torch.Size([8])
bn.num_batches_tracked  torch.Size([])
conv2.weight            torch.Size([16, 8, 5, 5])
conv2.bias              torch.Size([16])
fc1.weight              torch.Size([120, 400])
fc1.bias                torch.Size([120])
fc2.weight              torch.Size([10, 120])
fc2.bias                torch.Size([10])
Optimizer's state_dict:
state           {}
param_groups    [{'lr': 0.001, 'momentum': 0.9, 'dampening': 0, 'weight_decay':
```

可以看到 `model.state_dict()` 保存了卷积层，BatchNorm层和最大池化层的信息；而 `optimizer.state_dict()` 则保存的优化器的状态和相关的超参数。

三、跨设备保存加载模型

1、在 CPU 上加载在 GPU 上训练并保存的模型（Save on GPU, Load on CPU）：

```
device = torch.device('cpu')
model = TheModelClass()
# Load all tensors onto the CPU device
model.load_state_dict(torch.load('net_params.pkl', map_location=device))
```

`map_location`： a function, torch.device, string or a dict specifying how to remap storage locations

令 `torch.load()` 函数的 `map_location` 参数等于 `torch.device('cpu')` 即可。 这里令 `map_location` 参数等于 `'cpu'` 也同样可以。

2、在 GPU 上加载在 GPU 上训练并保存的模型（Save on GPU, Load on GPU）：

```
device = torch.device("cuda")
model = TheModelClass()
model.load_state_dict(torch.load('net_params.pkl'))
model.to(device)
```

在这里使用 `map_location` 参数不起作用，要使用 `model.to(torch.device("cuda"))` 将模型转换为CUDA优化的模型。

还需要对将要输入模型的数据调用 `data = data.to(device)`，即将数据从CPU转移到GPU。请注意，调用 `my_tensor.to(device)` 会返回一个 `my_tensor` 在 GPU 上的副本，它不会覆盖 `my_tensor`。因此需要手动覆盖张量：`my_tensor = my_tensor.to(device)`。

3、在 GPU 上加载在 GPU 上训练并保存的模型（Save on CPU, Load on GPU）

```
device = torch.device("cuda")
model = TheModelClass()
model.load_state_dict(torch.load('net_params.pkl', map_location="cuda:0"))
model.to(device)
```

当加载包含GPU tensors的模型时，这些tensors 会被默认加载到GPU上，不过是同一个GPU设备。

当有多个GPU设备时，可以通过将 `map_location` 设定为 *`cuda:device_id`* 来指定使用哪一个GPU设备，上面例子是指定编号为0的GPU设备。

其实也可以将 `torch.device("cuda")` 改为 `torch.device("cuda:0")` 来指定编号为0的GPU设备。

最后调用 `model.to(torch.device('cuda'))` 来将模型的tensors转换为 CUDA tensors。

下面是PyTorch官方文档上的用法，可以进行参考：

```
>>> torch.load('tensors.pt')
# Load all tensors onto the CPU
>>> torch.load('tensors.pt', map_location=torch.device('cpu'))
# Load all tensors onto the CPU, using a function
>>> torch.load('tensors.pt', map_location=lambda storage, loc: storage)
# Load all tensors onto GPU 1
>>> torch.load('tensors.pt', map_location=lambda storage, loc: storage.cuda(1))
# Map tensors from GPU 1 to GPU 0
>>> torch.load('tensors.pt', map_location={'cuda:1':'cuda:0'})
```

四、CUDA 的用法

在PyTorch中和GPU相关的几个函数：

```
import torch

# 判断cuda是否可用;
print(torch.cuda.is_available())

# 获取gpu数量;
print(torch.cuda.device_count())

# 获取gpu名字;
print(torch.cuda.get_device_name(0))

# 返回当前gpu设备索引，默认从0开始;
print(torch.cuda.current_device())

# 查看tensor或者model在哪块GPU上
print(torch.tensor([0]).get_device())
```

我的电脑输出为：

```
True
1
GeForce RTX 2080 Ti
0
```

有时我们需要把数据和模型从cpu移到gpu中，有以下两种方法：

```
use_cuda = torch.cuda.is_available()

# 方法一：
if use_cuda:
    data = data.cuda()
    model.cuda()

# 方法二：
device = torch.device("cuda" if use_cuda else "cpu")
data = data.to(device)
model.to(device)
```

个人比较习惯第二种方法，可以少一个 if 语句。而且该方法还可以通过设备号指定使用哪个GPU

设备，比如使用0号设备：

```
device = torch.device("cuda:0" if use_cuda else "cpu")
```

参考

pytorch.org/tutorials/b...

Saving and loading a model in Pytorch?

如果觉得有用，点个赞吧(ง •̀_•́)ง。

编辑于 2019-12-23

PyTorch 深度学习（Deep Learning） CUDA

文章被以下专栏收录

PyTorch 的那些事 关注专栏

推荐阅读

PyTorch 使用 Horovod 进行分布式训练

一、什么是分布式1、模型并行把复杂的神经网络进行拆分，分布在GPU里面进行训练，让每个GPU同步进行计算。这个方法通常用在模型比较复杂的情况下，但效率会有折扣。 2、数据并行即让每个机...

Towar... 发表于pytor...

Pytorch 训练优化

在单机多卡下，使用数据并行，可以通过调整GPU的参与数量，合理设置 batch size。单机多卡训练它会把整个模型加载到每张卡上，每张卡上都是完整的模型，然后把batch数据分到不同卡上，比如4...

落木潇潇

Pytorch的那些坑

birdl

2 条评论 切换为时间排序

写下你的评论...

幽睢 2019-09-07

不错！

赞

缄默的温度 2020-05-07

在optimizer.zero_grad()报错是为什么呢

赞