**保存和加载模型**

当保存和加载模型时，需要熟悉三个核心功能：

**torch.save**：将序列化对象保存到磁盘。此函数使用Python的pickle模块进行序列化。使用此函数可以保存如模型、tensor、字典等各种对象。

**torch.load**：使用pickle的unpickling功能将pickle对象文件反序列化到内存。此功能还可以有助于设备加载数据。

**torch.nn.Module.load\_state\_dict**：使用反序列化函数 **state\_dict** 来加载模型的参数字典。

**1.什么是状态字典：state\_dict?**

在PyTorch中，torch.nn.Module模型的可学习参数（即权重和偏差）包含在模型的参数中，（使用model.parameters()可以进行访问）。

state\_dict是Python字典对象，它将每一层映射到其参数张量。注意，只有具有可学习参数的层（如卷积层，线性层等）的模型 才具有state\_dict这一项。目标优化torch.optim也有state\_dict属性，它包含有关优化器的状态信息，以及使用的超参数。

因为state\_dict的对象是Python字典，所以它们可以很容易的保存、更新、修改和恢复，为PyTorch模型和优化器添加了大量模块。

下面通过从简单模型训练一个[分类器](https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html#sphx-glr-beginner-blitz-cifar10-tutorial-py)中来了解一下state\_dict的使用。

# 定义模型

class TheModelClass(nn.Module):

def \_\_init\_\_(self):

super(TheModelClass, self).\_\_init\_\_()

self.conv1 = nn.Conv2d(3, 6, 5)

self.pool = nn.MaxPool2d(2, 2)

self.conv2 = nn.Conv2d(6, 16, 5)

self.fc1 = nn.Linear(16 \* 5 \* 5, 120)

self.fc2 = nn.Linear(120, 84)

self.fc3 = nn.Linear(84, 10)

def forward(self, x):

x = self.pool(F.relu(self.conv1(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

# 初始化模型

model = TheModelClass()

# 初始化优化器

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

# 打印模型的状态字典

print("Model's state\_dict:")

for param\_tensor in model.state\_dict():

print(param\_tensor, "\t", model.state\_dict()[param\_tensor].size())

# 打印优化器的状态字典

print("Optimizer's state\_dict:")

for var\_name in optimizer.state\_dict():

print(var\_name, "\t", optimizer.state\_dict()[var\_name])

输出

Model's state\_dict:

conv1.weight torch.Size([6, 3, 5, 5])

conv1.bias torch.Size([6])

conv2.weight torch.Size([16, 6, 5, 5])

conv2.bias torch.Size([16])

fc1.weight torch.Size([120, 400])

fc1.bias torch.Size([120])

fc2.weight torch.Size([84, 120])

fc2.bias torch.Size([84])

fc3.weight torch.Size([10, 84])

fc3.bias torch.Size([10])

Optimizer's state\_dict:

state {}

param\_groups [{'lr': 0.001, 'momentum': 0.9, 'dampening': 0, 'weight\_decay': 0, 'nesterov': False, 'params': [4675713712, 4675713784, 4675714000, 4675714072, 4675714216, 4675714288, 4675714432, 4675714504, 4675714648, 4675714720]}]

**2.保存和加载推理模型**

**2.1 保存/加载state\_dict（推荐使用）**

保存

torch.save(model.state\_dict(), PATH)

加载

model = TheModelClass(\*args, \*\*kwargs)

model.load\_state\_dict(torch.load(PATH))

model.eval()

当保存好模型用来推断的时候，只需要保存模型学习到的参数，使用torch.save()函数来保存模型state\_dict,它会给模型恢复提供 最大的灵活性，这就是为什么要推荐它来保存的原因。

在 PyTorch 中最常见的模型保存使**‘.pt’**或者是**‘.pth’**作为**模型文件扩展名**。

**请记住:**

在运行推理之前，务必调用**model.eval()**去设置 **dropout** 和 **batch normalization 层** 为评估模式。如果不这么做，可能导致 模型推断结果不一致。

注意

load\_state\_dict()函数只接受字典对象，而不是保存对象的路径。这就意味着在你传给load\_state\_dict()函数之前，你必须反序列化保存的state\_dict。

例如，你无法通过 model.load\_state\_dict(PATH)来加载模型。

**2.2 保存/加载完整模型**

保存

torch.save(model, PATH)

加载

# 模型类必须在此之前被定义

model = torch.load(PATH)

model.eval()

此部分 保存/加载过程使用 最直观的语法并涉及最少量的代码。以 Python pickle 模块的方式来保存模型。

这种方法的缺点是:

序列化数据受限于某种特殊的类而且需要确切的字典结构。这是因为pickle无法保存模型类本身。相反，它保存包含类的文件的路径，该文件在加载时使用。 因此，当在其他项目使用或者重构之后，您的代码可能会以各种方式中断。

在 PyTorch 中最常见的模型保存使用‘.pt’或者是‘.pth’作为模型文件扩展名。

**请记住:**

在运行推理之前，务必调用**model.eval()**去设置 **dropout** 和 **batch normalization 层** 为评估模式。如果不这么做，可能导致 模型推断结果不一致。

**3. 保存和加载 Checkpoint 用于推理/继续训练**

保存

torch.save({

'epoch': epoch,

'model\_state\_dict': model.state\_dict(),

'optimizer\_state\_dict': optimizer.state\_dict(),

'loss': loss,

...

}, PATH)

加载

model = TheModelClass(\*args, \*\*kwargs)

optimizer = TheOptimizerClass(\*args, \*\*kwargs)

checkpoint = torch.load(PATH)

model.load\_state\_dict(checkpoint['model\_state\_dict'])

optimizer.load\_state\_dict(checkpoint['optimizer\_state\_dict'])

epoch = checkpoint['epoch']

loss = checkpoint['loss']

model.eval()

# - or -

model.train()

当保存成 Checkpoint 的时候，可用于推理或者是继续训练，保存的不仅仅是模型的 state\_dict 。保存优化器的 state\_dict 也很重要, 因为它包含作为模型训练更新的缓冲区和参数。你也许想保存其他项目，比如最新记录的训练损失，外部的torch.nn.Embedding层等等。

**要保存多个组件**，请在字典中组织它们并使用torch.save()来序列化字典。PyTorch 中常见的保存checkpoint 是使用 .tar 文件扩展名。

**要加载项目**，首先需要初始化模型和优化器，然后使用torch.load()来加载本地字典。这里,你可以非常容易的通过简单查询字典来访问你所保存的项目。

**请记住:**

在运行推理之前，务必调用model.eval()去设置 dropout 和 batch normalization 为评估。如果不这样做，有可能得到不一致的推断结果。

**要恢复训练**，请调用model.train()以确保这些层处于训练模式。

**4. 在一个文件中保存多个模型**

保存

torch.save({

'modelA\_state\_dict': modelA.state\_dict(),

'modelB\_state\_dict': modelB.state\_dict(),

'optimizerA\_state\_dict': optimizerA.state\_dict(),

'optimizerB\_state\_dict': optimizerB.state\_dict(),

...

}, PATH)

加载

modelA = TheModelAClass(\*args, \*\*kwargs)

modelB = TheModelBClass(\*args, \*\*kwargs)

optimizerA = TheOptimizerAClass(\*args, \*\*kwargs)

optimizerB = TheOptimizerBClass(\*args, \*\*kwargs)

checkpoint = torch.load(PATH)

modelA.load\_state\_dict(checkpoint['modelA\_state\_dict'])

modelB.load\_state\_dict(checkpoint['modelB\_state\_dict'])

optimizerA.load\_state\_dict(checkpoint['optimizerA\_state\_dict'])

optimizerB.load\_state\_dict(checkpoint['optimizerB\_state\_dict'])

modelA.eval()

modelB.eval()

# - or -

modelA.train()

modelB.train()

当保存一个模型由多个torch.nn.Modules组成时，例如GAN(对抗生成网络)、sequence-to-sequence (序列到序列模型), 或者是多个模型融合, 可以**采用与保存常规检查点相同的方法**。换句话说，保存每个模型的 state\_dict 的字典和相对应的优化器。如前所述，可以通过简单地将它们附加到字典的方式来保存任何其他项目，这样有助于恢复训练。

PyTorch 中常见的保存 **checkpoint 是使用 .tar 文件扩展名**。

**要加载项目**，首先**需要初始化模型和优化**器，然后使用torch.load()来加载本地字典。这里，你可以非常容易的通过简单查询字典来访问你所保存的项目。

**请记住**:

在运行推理之前，务必调用model.eval()去设置 dropout 和 batch normalization 为评估。如果不这样做，有可能得到不一致的推断结果。

**要恢复训练**，请调用model.train()以确保这些层处于训练模式。

**5. 使用在不同模型参数下的热启动模式**

保存

torch.save(modelA.state\_dict(), PATH)

加载

modelB = TheModelBClass(\*args, \*\*kwargs)

modelB.load\_state\_dict(torch.load(PATH), strict=False)

在迁移学习或训练新的复杂模型时，部分加载模型或加载部分模型是常见的情况。利用训练好的参数，有助于热启动训练过程，并希望帮助你的模型比从头开始训练能够更快地收敛。

无论是从缺少某些键的 state\_dict 加载还是从键的数目多于加载模型的 state\_dict , 都可以通过在load\_state\_dict()函数中**将strict参数设置为 False 来忽略非匹配键的函数**。

如果要将参数从一个层加载到另一个层，但是某些键不匹配，主要修改正在加载的 state\_dict 中的参数键的名称以匹配要在加载到模型中的键即可。

**6. 通过设备保存/加载模型**

**6.1 保存到 CPU、加载到 CPU，即（C2C）**

保存

torch.save(model.state\_dict(), PATH)

加载

device = torch.device('cpu')

model = TheModelClass(\*args, \*\*kwargs)

model.load\_state\_dict(torch.load(PATH, map\_location=device))

当从CPU上加载模型在GPU上训练时, 将torch.device('cpu')传递给torch.load()函数中的map\_location参数。在这种情况下，使用 map\_location参数 将张量下的存储器动态的重新映射到CPU设备。

**6.2 保存到 GPU、加载到 GPU，即（G2G）**

保存

torch.save(model.state\_dict(), PATH)

加载

device = torch.device("cuda")

model = TheModelClass(\*args, \*\*kwargs)

model.load\_state\_dict(torch.load(PATH))

# 确保在你提供给模型的任何输入张量上调用input = input.to(device)

model.to(device)

当在GPU上训练并把模型保存在GPU，只需要使用model.to(torch.device('cuda'))，将初始化的 model 转换为 CUDA 优化模型。

另外，请务必在所有模型输入上使用.to(torch.device('cuda'))函数来为模型准备数据。

请注意：

调用my\_tensor.to(device)会在GPU上返回my\_tensor的副本。

因此，请记住手动覆盖张量：my\_tensor= my\_tensor.to(torch.device('cuda'))。

**6.3 保存到 CPU，加载到 GPU，即（C2G）**

保存

torch.save(model.state\_dict(), PATH)

加载

device = torch.device("cuda")

model = TheModelClass(\*args, \*\*kwargs)

# Choose whatever GPU device number you want

model.load\_state\_dict(torch.load(PATH, map\_location="cuda:0"))

# 确保在你提供给模型的任何输入张量上调用input = input.to(device)

model.to(device)

在CPU上训练好并保存的模型加载到GPU时，将torch.load()函数中的map\_location参数设置为cuda:device\_id。这会将模型加载到指定的GPU设备。

接下来，请务必调用model.to(torch.device('cuda'))将模型的参数张量转换为 CUDA 张量。

最后，确保在所有模型输入上使用 to(torch.device('cuda')) 函数来为CUDA优化模型。

请注意:

调用my\_tensor.to(device)会在GPU上返回my\_tensor的新副本。它不会覆盖my\_tensor。

因此， 请手动覆盖张量my\_tensor = my\_tensor.to(torch.device('cuda'))。

**6.4 保存 torch.nn.DataParallel 模型**

保存

torch.save(model.module.state\_dict(), PATH)

加载

# 加载任何你想要的设备

torch.nn.DataParallel是一个模型封装，支持并行GPU使用。

要普通保存 DataParallel 模型, 请保存model.module.state\_dict()。 这样，你就可以非常灵活地以任何方式加载模型到你想要的设备中。