实践教程 | Pytorch中模型的保存与迁移

极市平台 2023-04-05 22:01:37 发表于广东 手机阅读

以下文章来源于月来客栈 ， 作者空字符

月来客栈

Hope is a good thing, maybe the best of things and no good thing ever dies.

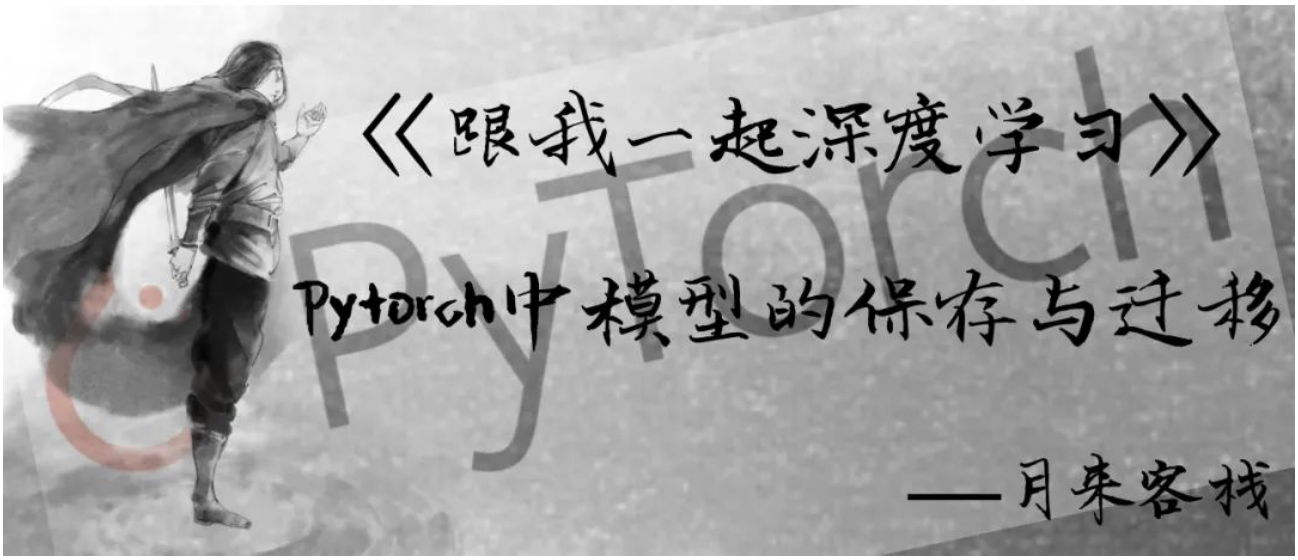
作者 | 空字符

来源 | 月来客栈

编辑 | 极市平台

极市导读

在本篇文章中，笔者首先介绍了模型复用的几种典型场景；然后介绍了如何查看Pytorch模型中的相关参数信息；接着介绍了如何载入模型、如何进行追加训练以及进行模型的迁移学习等。 >>加入极市CV技术交流群，走在计算机视觉的最前沿



1 引言

各位朋友大家好，今天要和大家介绍的内容是如何在Pytorch框架中对模型进行保存和载入、以及模型的迁移和再训练。

一般来说，**最常见的场景就是模型完成训练后的推断过程。**一个网络模型在完成训练后通常都需要对新样本进行预测，此时就只需要构建模型的前向传播过程，然后载入已训练好的参数初始化网络即可。

第2个场景就是模型的再训练过程。一个模型在一批数据上训练完成之后需要将其保存到本地，并且可能过了一段时间后又收集到了一批新的数据，因此这个时候就需要将之前的模型载入进行在新数据上进行增量训练（或者是在整个数据上进行全量训练）。

第3个应用场景就是模型的迁移学习。这个时候就是将别人已经训练好的预模型拿过来，作为你自己网络模型参数的一部分进行初始化。例如：你自己在Bert模型的基础上加了几个全连接层来做分类任务，那么你就需要将原始BERT模型中的参数载入并以此来初始化你的网络中的Bert部分的权重参数。

在接下来的这篇文章中，笔者就以上述3个场景为例来**介绍如何利用Pytorch框架来完成上述过程。**

2 模型的保存与复用

在Pytorch中，我们可以通过 `torch.save()` 和 `torch.load()` 来完成上述场景中的主要步骤。下面，笔者将以之前介绍的LeNet5网络模型为例来分别进行介绍。不过在这之前，我们先来看看Pytorch中模型参数的保存形式。

2.1 查看网络模型参数

（1）查看参数

首先定义好LeNet5的网络模型结构，如下代码所示：

```
class LeNet5(nn.Module):
    def __init__(self, ):
        super(LeNet5, self).__init__()
        self.conv = nn.Sequential(  # [n,1,28,28]
            nn.Conv2d(1, 6, 5, padding=2),  # in_channels, out_channels, kernel_size
            nn.ReLU(),  # [n,6,24,24]
            nn.MaxPool2d(2, 2),  # kernel_size, stride  [n,6,14,14]
            nn.Conv2d(6, 16, 5),  # [n,16,10,10]
            nn.ReLU(),
            nn.MaxPool2d(2, 2))  # [n,16,5,5]
        self.fc = nn.Sequential(
            nn.Flatten(),
            nn.Linear(16 * 5 * 5, 120),
            nn.ReLU(),
            nn.Linear(120, 84),
            nn.ReLU(),
            nn.Linear(84, 10))
    def forward(self, img):
        output = self.conv(img)
        output = self.fc(output)
        return output
```

在定义好LeNet5这个网络结构的类之后，只要我们完成了这个类的实例化操作，那么网络中对应的权重参数也都完成了初始化的工作，即有了一个初始值。同时，我们可以通过如下方式来访问：

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

其输出的结果为：

```
conv.0.weight   torch.Size([6, 1, 5, 5])
conv.0.bias   torch.Size([6])
conv.3.weight   torch.Size([16, 6, 5, 5])
....
....
```

可以发现，网络模型中的参数 model.state_dict() 其实是以字典的形式（实质上是 collections 模块中的 OrderedDict ）保存下来的：

```
print(model.state_dict().keys())
# odict_keys(['conv.0.weight', 'conv.0.bias', 'conv.3.weight',
'conv.3.bias', 'fc.1.weight', 'fc.1.bias', 'fc.3.weight', 'fc.3.bias',
'fc.5.weight', 'fc.5.bias'])
```

（2）自定义参数前缀

同时，这里值得注意的地方有两点：①参数名中的 fc 和 conv 前缀是根据你在上面定义 nn.Sequential() 时的名字所确定的；②参数名中的数字表示每个 Sequential() 中网络层所在的位置。例如将网络结构定义成如下形式：

```
class LeNet5(nn.Module):
    def __init__(self, ):
        super(LeNet5, self).__init__()
        self.moon = nn.Sequential(  # [n,1,28,28]
            nn.Conv2d(1, 6, 5, padding=2),  # in_channels, out_channels, kernel_size
            nn.ReLU(),  # [n,6,24,24]
            nn.MaxPool2d(2, 2),  # kernel_size, stride  [n,6,14,14]
            nn.Conv2d(6, 16, 5),  # [n,16,10,10]
            nn.ReLU(),
            nn.MaxPool2d(2, 2),
            nn.Flatten(),
            nn.Linear(16 * 5 * 5, 120),
            nn.ReLU(),
            nn.Linear(120, 84),
            nn.ReLU(),
            nn.Linear(84, 10))
```

那么其参数名则为：

```
print(model.state_dict().keys())
odict_keys(['moon.0.weight', 'moon.0.bias', 'moon.3.weight',
 'moon.3.bias', 'moon.7.weight', 'moon.7.bias', 'moon.9.weight',
'moon.9.bias', 'moon.11.weight', 'moon.11.bias'])
```

理解了这一点对于后续我们去解析和载入一些预训练模型很有帮助。

除此之外，对于中的优化器等，其同样有对应的 `state_dict()` 方法来获取对于的参数，例如：

```
optimizer = torch.optim.SGD(model.parameters(), lr=0.001, momentum=0.9)
print("Optimizer's state_dict:")
for var_name in optimizer.state_dict():
   print(var_name, "\t", optimizer.state_dict()[var_name])

#
Optimizer's state_dict:
state   {}
param_groups   [{'lr': 0.001, 'momentum': 0.9, 'dampening': 0,
'weight_decay': 0, 'nesterov': False,
'params': [140239245300504, 140239208339784, 140239245311360,
140239245310856, 140239266942480, 140239266942552, 140239266942624,
140239266942696, 140239266942912, 140239267041352]}]
```

在介绍完模型参数的查看方法后，就可以进入到模型复用阶段的内容介绍了。

2.2 载入模型进行推断

（1） 模型保存

在Pytorch中，对于模型的保存来说是非常简单的，通常来说通过如下两行代码便可以实现：

```
model_save_path = os.path.join(model_save_dir, 'model.pt')
torch.save(model.state_dict(), model_save_path)
```

在指定保存的模型名称时Pytorch官方建议的后缀为 `.pt` 或者 `.pth` （当然也不是强制的）。最后，只需要在合适的地方加入第2行代码即可完成模型的保存。

同时，如果想要在训练过程中保存某个条件下的最优模型，那么应该通过如下方式：

```
best_model_state = deepcopy(model.state_dict())
torch.save(best_model_state, model_save_path)
```

而不是：

```
best_model_state = model.state_dict()
torch.save(best_model_state, model_save_path)
```

因为后者 `best_model_state` 得到只是 `model.state_dict()` 的引用，它依旧会随着训练过程而发生改变。

（2）复用模型进行推断

在推断过程中，首先需要完成网络的初始化，然后再载入已有的模型参数来覆盖网络中的权重参数即可，示例代码如下：

```
def inference(data_iter, device, model_save_dir='./MODEL'):
    model = LeNet5()  # 初始化现有模型的权重参数
    model.to(device)
    model_save_path = os.path.join(model_save_dir, 'model.pt')
    if os.path.exists(model_save_path):
        loaded_paras = torch.load(model_save_path)
    model.load_state_dict(loaded_paras)  # 用本地已有模型来重新初始化网络权重参数
    model.eval() # 注意不要忘记
    with torch.no_grad():
        acc_sum, n = 0.0, 0
        for x, y in data_iter:
        x, y = x.to(device), y.to(device)
        logits = model(x)
```

```
        acc_sum += (logits.argmax(1) == y).float().sum().item()
        n += len(y)
        print("Accuracy in test data is :", acc_sum / n)
```

在上述代码中，4-7行便是用来载入本地模型参数，并用其覆盖网络模型中原有的参数。这样，便可以进行后续的推断工作：

```
Accuracy in test data is : 0.8851
```

2.3 载入模型进行训练

在介绍完模型的保存与复用之后，对于网络的追加训练就很简单了。最简便的一种方式就是在训练过程中只保存网络权重，然后在后续进行追加训练时只载入网络权重参数初始化网络进行训练即可，示例如下（完整代码参见[2]）：

```
def train(self):
      #......
      model_save_path = os.path.join(self.model_save_dir, 'model.pt')
      if os.path.exists(model_save_path):
          loaded_paras = torch.load(model_save_path)
          self.model.load_state_dict(loaded_paras)
          print("#### 成功载入已有模型，进行追加训练...")
      optimizer = torch.optim.Adam(self.model.parameters(), lr=self.learning_rate)  # 定
     #......
      for epoch in range(self.epochs):
          for i, (x, y) in enumerate(train_iter):
              x, y = x.to(device), y.to(device)
              logits = self.model(x)
              # ......
          print("Epochs[{}/{}]--acc on test {:.4}".format(epoch, self.epochs,
                                                 self.evaluate(test_iter, self.model, device)
          torch.save(self.model.state_dict(), model_save_path)
```

这样，便完成了模型的追加训练：

```
#### 成功载入已有模型，进行追加训练...
Epochs[0/5]---batch[938/0]---acc 0.9062---loss 0.2926
Epochs[0/5]---batch[938/100]---acc 0.9375---loss 0.1598
......
```

除此之外，你也可以在保存参数的时候，将优化器参数、损失值等一同保存下来，然后在恢复模型的时候连同其它参数一起恢复，示例如下：

```
model_save_path = os.path.join(model_save_dir, 'model.pt')
torch.save({
            'epoch': epoch,
            'model_state_dict': model.state_dict(),
            'optimizer_state_dict': optimizer.state_dict(),
            'loss': loss,
            ...
            }, model_save_path)
```

载入方式如下：

```
checkpoint = torch.load(model_save_path)
model.load_state_dict(checkpoint['model_state_dict'])
optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
epoch = checkpoint['epoch']
loss = checkpoint['loss']
```

2.4 载入模型进行迁移

（1）定义新模型

到目前为止，对于前面两种应用场景的介绍就算完成了，可以发现总体上并不复杂。但是对于第3中场景的应用来说就会略微复杂一点。

假设现在有一个LeNet6网络模型，它是在LeNet5的基础最后多加了一个全连接层，其定义如下：

```
class LeNet6(nn.Module):
    def __init__(self, ):
        super(LeNet6, self).__init__()
        self.conv = nn.Sequential(  # [n,1,28,28]
            nn.Conv2d(1, 6, 5, padding=2),  # in_channels, out_channels, kernel_size
            nn.ReLU(),  # [n,6,24,24]
            nn.MaxPool2d(2, 2),  # kernel_size, stride  [n,6,14,14]
            nn.Conv2d(6, 16, 5),  # [n,16,10,10]
            nn.ReLU(),
            nn.MaxPool2d(2, 2))  # [n,16,5,5]
        self.fc = nn.Sequential(
            nn.Flatten(),
            nn.Linear(16 * 5 * 5, 120),
            nn.ReLU(),
            nn.Linear(120, 84),
            nn.ReLU(),
            nn.Linear(84, 64),
            nn.ReLU(),
            nn.Linear(64, 10) ) # 新加入的全连接层
```

接下来，我们需要将在LeNet5上训练得到的权重参数迁移到LeNet6网络中去。从上面LeNet6的定义可以发现，此时尽管只是多加了一个全连接层，但是倒数第2层参数的维度也发生了变换。因此，对于LeNet6来说只能复用LeNet5网络前面4层的权重参数。

（2）查看模型参数

在拿到一个模型参数后，首先我们可以将其载入，然查看相关参数的信息：

```
model_save_path = os.path.join('./MODEL', 'model.pt')
loaded_paras = torch.load(model_save_path)
for param_tensor in loaded_paras:
    print(param_tensor, "\t", loaded_paras[param_tensor].size())

#---- 可复用部分
conv.0.weight   torch.Size([6, 1, 5, 5])
conv.0.bias   torch.Size([6])
conv.3.weight   torch.Size([16, 6, 5, 5])
conv.3.bias   torch.Size([16])
fc.1.weight   torch.Size([120, 400])
fc.1.bias   torch.Size([120])
```

```
fc.3.weight   torch.Size([84, 120])
fc.3.bias   torch.Size([84])
#----- 不可复用部分
fc.5.weight   torch.Size([10, 84])
fc.5.bias   torch.Size([10])
```

同时，对于LeNet6网络的参数信息为：

```
model = LeNet6()
for param_tensor in model.state_dict():
    print(param_tensor, "\t", model.state_dict()[param_tensor].size())
#
conv.0.weight   torch.Size([6, 1, 5, 5])
conv.0.bias   torch.Size([6])
conv.3.weight   torch.Size([16, 6, 5, 5])
conv.3.bias   torch.Size([16])
fc.1.weight   torch.Size([120, 400])
fc.1.bias   torch.Size([120])
fc.3.weight   torch.Size([84, 120])
fc.3.bias   torch.Size([84])
#------ 新加入部分
fc.5.weight   torch.Size([64, 84])
fc.5.bias   torch.Size([64])
fc.7.weight   torch.Size([10, 64])
fc.7.bias   torch.Size([10])
```

在理清楚了新旧模型的参数后，下面就可以将LeNet5中我们需要的参数给取出来，然后再换到LeNet6的网络中。

（3）模型迁移

虽然本地载入的模型参数（上面的 `loaded_paras` ）和模型初始化后的参数（上面的 `model.state_dict()` ）都是一个字典的形式，但是我们并不能够直接改变 `model.state_dict()` 中的权重参数。这里需要先构造一个 `state_dict` 然后通过 `model.load_state_dict()` 方法来重新初始化网络中的参数。

同时，在这个过程中我们需要筛选掉本地模型中不可复用的部分，具体代码如下：

```
def para_state_dict(model, model_save_dir):
    state_dict = deepcopy(model.state_dict())
    model_save_path = os.path.join(model_save_dir, 'model.pt')
    if os.path.exists(model_save_path):
        loaded_paras = torch.load(model_save_path)
        for key in state_dict:  # 在新的网络模型中遍历对应参数
            if key in loaded_paras and state_dict[key].size() == loaded_paras[key].size():
                print("成功初始化参数:", key)
                state_dict[key] = loaded_paras[key]
    return state_dict
```

在上述代码中，第2行的作用是先拷贝网络中（LeNet6）原有的参数；第6-9行则是用本地的模型参数（LeNet5）中可以复用的替换掉LeNet6中的对应部分，其中第7行就是判断可用的条件。同时需要注意的是在不同的情况下筛选的方式可能不一样，因此具体情况需要具体分析，但是整体逻辑是一样的。

最后，我们只需要在模型训练之前调用该函数，然后重新初始化LeNet6中的部分权重参数即可[2]：

```
state_dict = para_state_dict(self.model, self.model_save_dir)
self.model.load_state_dict(state_dict)
```

训练结果如下：

```
成功初始化参数: conv.0.weight
成功初始化参数: conv.0.bias
成功初始化参数: conv.3.weight
成功初始化参数: conv.3.bias
成功初始化参数: fc.1.weight
成功初始化参数: fc.1.bias
成功初始化参数: fc.3.weight
成功初始化参数: fc.3.bias
#### 成功载入已有模型，进行追加训练...
Epochs[0/5]---batch[938/0]---acc 0.1094---loss 2.512
Epochs[0/5]---batch[938/100]---acc 0.9375---loss 0.2141
Epochs[0/5]---batch[938/200]---acc 0.9219---loss 0.2729
Epochs[0/5]---batch[938/300]---acc 0.8906---loss 0.2958
......
```

```
Epochs[0/5]---batch[938/900]---acc 0.8906---loss 0.2828
Epochs[0/5]--acc on test 0.8808
```

可以发现，在大约100个batch之后，模型的准确率就提升上来了。

3 总结

在本篇文章中，笔者首先介绍了模型复用的几种典型场景；然后介绍了如何查看Pytorch模型中的相关参数信息；接着介绍了如何载入模型、如何进行追加训练以及进行模型的迁移学习等。

感谢您的阅读！

引用

[1] **SAVING AND LOADING MODELS** https://pytorch.org/tutorials/beginner/saving_loading_models.html

[2] **示例代码** https://github.com/moon-hotel/DeepLearningWithMe

CVPR2023：CVPR'23 最新 125 篇论文分方向整理 | 检测、分割、人脸、视频处理、医学影像、神经网络结构、小样本学习等方向

数据集：自动驾驶方向开源数据集资源汇总 | 医学影像方向开源数据集资源汇总 | 卫星图像公开数据集资源汇总

点击阅读原文进入CV社区

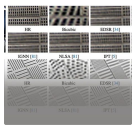
收获更多技术干货

阅读原文

喜欢此内容的人还喜欢

ICCV 2023 | 南开程明明团队提出适用于SR任务的新颖注意力机制（已开源）

极市平台



YOLOv5帮助母猪产仔？南京农业大学研发母猪产仔检测模型并部署到Jetson Nano开发板

极市平台

实践教程 | 使用 OpenCV 进行特征提取（颜色、形状和纹理）

极市平台