# 7.PyTorch训练技巧

## 7.1.模型保存与加载

保存：

torch.save  
 主要参数：  
 • obj：对象  
 • f：输出路径

保存分为两种模式：

1: 保存整个Module，即保存了整个模型的框架和参数  
 torch.save(net, path)  
 2: 保存模型参数，即只保存模型的参数，下次使用时需要自己重新构建框架  
 state\_dict = net.state\_dict()  
 torch.save(state\_dict , path)

加载：

2. torch.load  
 主要参数  
 • f：文件路径  
 • map\_location：指定存放位置, cpu or gpu

加载也有两种模式，与保存的两种方式对应，即加载模型和加载参数

代码实现：

模型的保存

*# -\*- coding: utf-8 -\*-***import** torch  
**import** torch.nn **as** nn  
  
*# 定义模型***class** LeNet2(nn.Module):  
 **def** \_\_init\_\_(self, classes):  
 super(LeNet2, self).\_\_init\_\_()  
 self.features = nn.Sequential(  
 nn.Conv2d(3, 6, 5),  
 nn.ReLU(),  
 nn.MaxPool2d(2, 2),  
 nn.Conv2d(6, 16, 5),  
 nn.ReLU(),  
 nn.MaxPool2d(2, 2)  
 )  
 self.classifier = nn.Sequential(  
 nn.Linear(16\*5\*5, 120),  
 nn.ReLU(),  
 nn.Linear(120, 84),  
 nn.ReLU(),  
 nn.Linear(84, classes)  
 )  
  
 **def** forward(self, x):  
 x = self.features(x)  
 x = x.view(x.size()[0], -1)  
 x = self.classifier(x)  
 **return** x  
  
 **def** initialize(self):  
 **for** p **in** self.parameters():  
 p.data.fill\_(2020)  
  
*# 创建模型*net = LeNet2(classes=10)  
  
*# "训练"*print(**"训练前: "**, net.features[0].weight[0, ...])  
net.initialize()  
print(**"训练后: "**, net.features[0].weight[0, ...])  
  
*# 设置保存路径，分别保存模型与参数*path\_model = **"./model.pkl"**path\_state\_dict = **"./model\_state\_dict.pkl"***# 保存整个模型*torch.save(net, path\_model)  
  
*# 保存模型参数*net\_state\_dict = net.state\_dict()  
torch.save(net\_state\_dict, path\_state\_dict)

|  |
| --- |
| 训练前: tensor([[[ 0.0791, -0.0743, -0.0078, -0.0547, 0.0729],  [ 0.0526, 0.1094, 0.1121, 0.0600, 0.1051],  ...  [-0.0740, 0.0235, 0.1109, 0.0119, 0.0783]]],  grad\_fn=<SelectBackward>)  训练后: tensor([[[2020., 2020., 2020., 2020., 2020.],  [2020., 2020., 2020., 2020., 2020.],  ...  [2020., 2020., 2020., 2020., 2020.]]], grad\_fn=<SelectBackward>) |

保存后，可以在保存的目录查看到保存的模型文件

模型的加载：

1.加载整个模型

*# -\*- coding: utf-8 -\*-***import** torch  
**import** torch.nn **as** nn  
  
flag = 1  
*# flag = 0***if** flag:  
 path\_model = **"./model.pkl"** net\_load = torch.load(path\_model)  
  
 print(net\_load)  
 print(net\_load.features[0].weight[0, ...])

|  |
| --- |
| LeNet2(  (features): Sequential(  (0): Conv2d(3, 6, kernel\_size=(5, 5), stride=(1, 1))  (1): ReLU()  (2): MaxPool2d(kernel\_size=2, stride=2, padding=0, dilation=1, ceil\_mode=False)  (3): Conv2d(6, 16, kernel\_size=(5, 5), stride=(1, 1))  (4): ReLU()  (5): MaxPool2d(kernel\_size=2, stride=2, padding=0, dilation=1, ceil\_mode=False)  )  (classifier): Sequential(  (0): Linear(in\_features=400, out\_features=120, bias=True)  (1): ReLU()  (2): Linear(in\_features=120, out\_features=84, bias=True)  (3): ReLU()  (4): Linear(in\_features=84, out\_features=10, bias=True)  )  )  tensor([[[2020., 2020., 2020., 2020., 2020.],  [2020., 2020., 2020., 2020., 2020.],  ...  [2020., 2020., 2020., 2020., 2020.]]], grad\_fn=<SelectBackward>) |

2.加载模型参数，并自己创建模型

**class** LeNet2(nn.Module):  
 **def** \_\_init\_\_(self, classes):  
 super(LeNet2, self).\_\_init\_\_()  
 self.features = nn.Sequential(  
 nn.Conv2d(3, 6, 5),  
 nn.ReLU(),  
 nn.MaxPool2d(2, 2),  
 nn.Conv2d(6, 16, 5),  
 nn.ReLU(),  
 nn.MaxPool2d(2, 2)  
 )  
 self.classifier = nn.Sequential(  
 nn.Linear(16\*5\*5, 120),  
 nn.ReLU(),  
 nn.Linear(120, 84),  
 nn.ReLU(),  
 nn.Linear(84, classes)  
 )  
  
 **def** forward(self, x):  
 x = self.features(x)  
 x = x.view(x.size()[0], -1)  
 x = self.classifier(x)  
 **return** x  
  
 **def** initialize(self):  
 **for** p **in** self.parameters():  
 p.data.fill\_(2020)

flag = 1  
*# flag = 0***if** flag:  
 path\_state\_dict = **"./model\_state\_dict.pkl"** state\_dict\_load = torch.load(path\_state\_dict)  
  
 net\_new = LeNet2(classes=10)  
  
 print(**"加载前: "**, net\_new.features[0].weight[0, ...])  
 net\_new.load\_state\_dict(state\_dict\_load)  
 print(**"加载后: "**, net\_new.features[0].weight[0, ...])

|  |
| --- |
| 加载前: tensor([[[-0.1134, 0.0326, -0.0086, 0.0342, -0.0673],  [-0.0222, -0.1068, 0.0853, 0.0753, -0.0508],  ...  [ 0.0497, -0.0221, 0.0892, -0.0781, -0.0866]]],  grad\_fn=<SelectBackward>)  加载后: tensor([[[2020., 2020., 2020., 2020., 2020.],  [2020., 2020., 2020., 2020., 2020.],  ...  [2020., 2020., 2020., 2020., 2020.]]], grad\_fn=<SelectBackward>) |

断点续训练

在迭代过程中，如果训练时间很长，如果中间意外中断，那么又需要从头开始训练，我们可以设置检查点，例如每5个epoch设置一次检查点，这样如果中断，从上次的检查点获取之前的模型参数即可。

设置检查点：通常需要将模型参数、优化器参数、训练轮数保存

checkpoint = {  
 "model\_state\_dict": net.state\_dict(),  
 "optimizer\_state\_dict": optimizer.state\_dict(),  
 "epoch": epoch  
 }

设置保存点代码：

checkpoint = {**"model\_state\_dict"**: net.state\_dict(),  
 **"optimizer\_state\_dict"**: optimizer.state\_dict(),  
 **"epoch"**: epoch}  
path\_checkpoint = **"./checkpoint\_{}\_epoch.pkl"**.format(epoch)  
torch.save(checkpoint, path\_checkpoint)

断点恢复代码：

path\_checkpoint = **"./checkpoint\_4\_epoch.pkl"**checkpoint = torch.load(path\_checkpoint)  
  
net.load\_state\_dict(checkpoint[**'model\_state\_dict'**])  
optimizer.load\_state\_dict(checkpoint[**'optimizer\_state\_dict'**])  
start\_epoch = checkpoint[**'epoch'**]  
  
*# 设置学习率也是从新的epoch开始的*scheduler.last\_epoch = start\_epoch

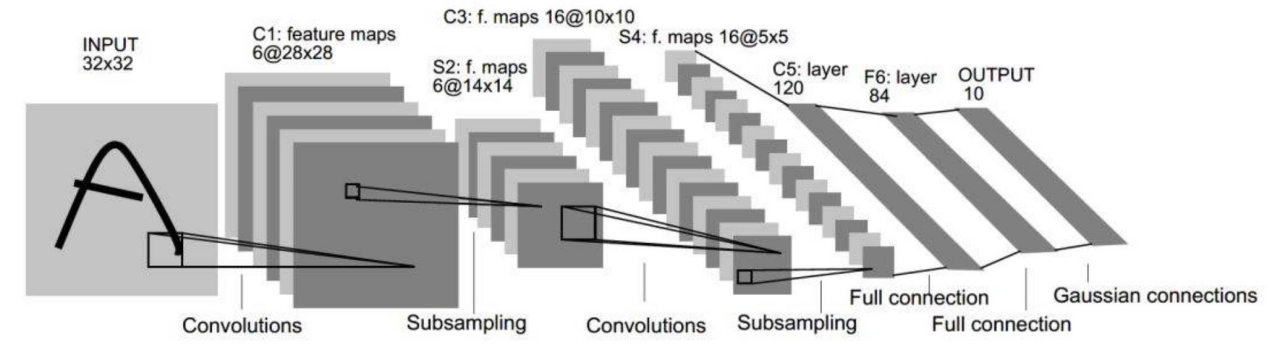
## 7.2.模型微调

模型微调 - Model Finetune：模型的迁移学习

当我们训练一个模型时，如果从头训练可能比较困难，那么可以基于别人已经训练好的模型，在其基础上再结合自己的数据进行训练，这样训练的模型往往会达到更好的效果。

模型微调，也称为迁移学习，即将已经训练好的模型迁移到我们将要训练的模型中

例如对于下面的模型：



可以将前面卷积层部分看做特征提取，将后面的全连接层看做是分类，那么前面的卷积层的模型和参数我们就可以拿来直接使用，而针对我们自己的数据自定义后面的全连接层。

在训练时，对于前面的卷积层，我们不需要修改其参数或者微小的改动其参数，因此，我们在训练时可以将前面的卷积层和后面的全连接层分为两个组，前面组可以设置其梯度为0，或者设置学习率为0，也可以为它们设置很小的数，这样相当于主要训练的就是后面的全连接层的参数。

因此，模型微调过程总结为：

模型微调步骤：  
 1. 获取预训练模型参数  
 2. 加载模型（ load\_state\_dict）  
 3. 修改输出层

模型微调训练方法：  
 1. 固定预训练的参数(requires\_grad =False； lr=0)  
 2. Features Extractor较小学习率（ params\_group）

案例：

以二分类为例，我们做蚂蚁和蜜蜂的二分类

数据：

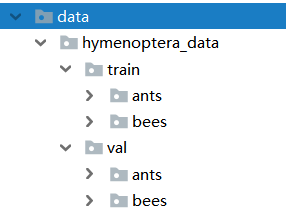
训练集：各120张 验证集：各70张



由于数据量较少，所以我们可以使用已经训练好的模型Finetune Resnet-18，在其基础上训练我们的二分类模型。

代码实现：

数据准备：



同样，将resnet18模型的参数放入data目录下



首先，准备一个DataSet

*# -\*- coding: utf-8 -\*-***import** os  
**from** PIL **import** Image  
**from** torch.utils.data **import** Dataset  
  
**class** AntsDataset(Dataset):  
 **def** \_\_init\_\_(self, data\_dir, transform=**None**):  
 self.label\_name = {**"ants"**: 0, **"bees"**: 1}  
 *# 调用get\_img\_info获取图片路径及标签* self.data\_info = self.get\_img\_info(data\_dir)  
 self.transform = transform  
  
 *# 返回数据以及对应的标签* **def** \_\_getitem\_\_(self, index):  
 path\_img, label = self.data\_info[index]  
 img = Image.open(path\_img).convert(**'RGB'**)  
  
 **if** self.transform **is not None**:  
 img = self.transform(img)  
  
 **return** img, label  
  
 *# 返回样本数量* **def** \_\_len\_\_(self):  
 **return** len(self.data\_info)  
  
 **def** get\_img\_info(self, data\_dir):  
 data\_info = list()  
 **for** root, dirs, \_ **in** os.walk(data\_dir):  
 *# 遍历类别* **for** sub\_dir **in** dirs:  
 img\_names = os.listdir(os.path.join(root, sub\_dir))  
 img\_names = list(filter(**lambda** x: x.endswith(**'.jpg'**), img\_names))  
  
 *# 遍历图片* **for** i **in** range(len(img\_names)):  
 img\_name = img\_names[i]  
 path\_img = os.path.join(root, sub\_dir, img\_name)  
 label = self.label\_name[sub\_dir]  
 *# 将图片路径及label添加到data\_info中* data\_info.append((path\_img, int(label)))  
  
 **if** len(data\_info) == 0:  
 *# 如果data\_info中没有图片，通过raise显示地引发异常信息* **raise** Exception(**"\ndata\_dir:{} is a empty dir! Please checkout your path to images!"**.format(data\_dir))  
  
 **return** data\_info

**训练代码实现**

*# -\*- coding: utf-8 -\*-***import** os  
**import** numpy **as** np  
**import** torch  
**import** random  
**import** torch.nn **as** nn  
**from** torch.utils.data **import** DataLoader  
**import** torchvision.transforms **as** transforms  
**import** torch.optim **as** optim  
**from** matplotlib **import** pyplot **as** plt  
**from** tools.my\_dataset **import** AntsDataset  
**import** torchvision.models **as** models  
BASEDIR = os.path.dirname(os.path.abspath(\_\_file\_\_))  
*# 设置device，用来设置是否使用gpu训练*device = torch.device(**"cuda" if** torch.cuda.is\_available() **else "cpu"**)  
print(**"use device :{}"**.format(device))  
  
**def** set\_seed(seed=1):  
 random.seed(seed)  
 np.random.seed(seed)  
 torch.manual\_seed(seed)  
 torch.cuda.manual\_seed(seed)  
  
set\_seed(1) *# 设置随机种子*label\_name = {**"ants"**: 0, **"bees"**: 1}  
  
*# 参数设置*MAX\_EPOCH = 25  
BATCH\_SIZE = 16  
LR = 0.001  
*# 每几批数据打印一次结果,10\*16=160<240*log\_interval = 10  
*# 每几轮做一次验证*val\_interval = 1  
classes = 2  
start\_epoch = -1  
*# 学习率下降间隔数*lr\_decay\_step = 7  
  
*# ============================ step 1/5 数据 ============================*data\_dir = os.path.join(BASEDIR, **".."**, **".."**, **"data/hymenoptera\_data"**)  
train\_dir = os.path.join(data\_dir, **"train"**)  
valid\_dir = os.path.join(data\_dir, **"val"**)  
  
norm\_mean = [0.485, 0.456, 0.406]  
norm\_std = [0.229, 0.224, 0.225]  
  
train\_transform = transforms.Compose([  
 transforms.RandomResizedCrop(224),  
 *# 概率水平翻转* transforms.RandomHorizontalFlip(),  
 transforms.ToTensor(),  
 transforms.Normalize(norm\_mean, norm\_std),  
])  
  
valid\_transform = transforms.Compose([  
 transforms.Resize(256),  
 *# 按照中心裁剪* transforms.CenterCrop(224),  
 transforms.ToTensor(),  
 transforms.Normalize(norm\_mean, norm\_std),  
])  
  
*# 构建MyDataset实例*train\_data = AntsDataset(data\_dir=train\_dir, transform=train\_transform)  
valid\_data = AntsDataset(data\_dir=valid\_dir, transform=valid\_transform)  
  
*# 构建DataLoder*train\_loader = DataLoader(dataset=train\_data, batch\_size=BATCH\_SIZE, shuffle=**True**)  
valid\_loader = DataLoader(dataset=valid\_data, batch\_size=BATCH\_SIZE)  
  
*# ============================ step 2/5 模型 ============================  
  
# 1/3 构建模型*resnet18\_ft = models.resnet18()  
  
*# 2/3 加载参数*flag = 0  
*# flag = 1***if** flag:  
 *# 加载resnet18模型参数* path\_pretrained\_model = os.path.join(BASEDIR, **".."**, **".."**, **"data/resnet18-5c106cde.pth"**)  
 state\_dict\_load = torch.load(path\_pretrained\_model)  
 resnet18\_ft.load\_state\_dict(state\_dict\_load)  
  
*# 法1 : 冻结卷积层*flag\_m1 = 0  
*# flag\_m1 = 1***if** flag\_m1:  
 **for** param **in** resnet18\_ft.parameters():  
 param.requires\_grad = **False** print(**"conv1.weights[0, 0, ...]:\n {}"**.format(resnet18\_ft.conv1.weight[0, 0, ...]))  
  
  
*# 3/3 替换全连接fc层  
# 获取原resnet18模型的全连接层的输入特征数量*num\_ftrs = resnet18\_ft.fc.in\_features  
*# 自定义线性层并赋值给resnet18\_ft模型，线性层输入为原resnet18模型的全连接层输入，输入为分类数*resnet18\_ft.fc = nn.Linear(num\_ftrs, classes)  
  
*# 将模型放在gpu上运行*resnet18\_ft.to(device)

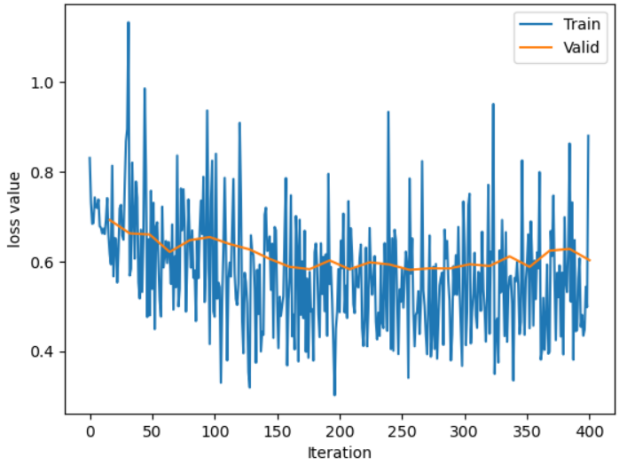
*# ============================ step 3/5 损失函数 ============================*criterion = nn.CrossEntropyLoss() *# 选择损失函数  
  
# ============================ step 4/5 优化器 ============================  
# 法2 : conv 小学习率*flag = 0  
*# flag = 1***if** flag:  
 *# 为卷积层部分设置一个参数组，为全连接层部分设置一个参数组  
 # 获取全连接层的参数地址，并存储为list* fc\_params\_id = list(map(id, resnet18\_ft.fc.parameters()))  
 *# 获取resnet18\_ft模型的所有参数，并过滤掉所有全连接层的参数，即为非全连接层参数* base\_params = filter(**lambda** p: id(p) **not in** fc\_params\_id, resnet18\_ft.parameters())  
 *# 为非全连接层参数设置学习率为0.1，全连接层设置正常学习率，共用momentum* optimizer = optim.SGD([  
 {**'params'**: base\_params, **'lr'**: LR\*0.1}, *# 0* {**'params'**: resnet18\_ft.fc.parameters(), **'lr'**: LR}], momentum=0.9)  
  
**else**:  
 *# 所有参数都使用同一个学习率* optimizer = optim.SGD(resnet18\_ft.parameters(), lr=LR, momentum=0.9) *# 选择优化器  
  
# 设置学习率下降策略*scheduler = torch.optim.lr\_scheduler.StepLR(optimizer, step\_size=lr\_decay\_step, gamma=0.1)  
  
*# ============================ step 5/5 训练 ============================  
# 创建两个list，存放训练集和测试集每批的loss*train\_curve = list()  
valid\_curve = list()  
  
**for** epoch **in** range(start\_epoch + 1, MAX\_EPOCH):  
 *# 分别定义平均loss、预测正确总数、总样本数* loss\_mean = 0.  
 correct = 0.  
 total = 0.  
  
 *# 设置模型为训练模式* resnet18\_ft.train()  
 **for** i, data **in** enumerate(train\_loader):  
  
 *# forward* inputs, labels = data  
 *# 数据与标签也要放到gpu上运行* inputs, labels = inputs.to(device), labels.to(device)  
 outputs = resnet18\_ft(inputs)  
  
 *# backward* optimizer.zero\_grad()  
 loss = criterion(outputs, labels)  
 loss.backward()  
  
 *# update weights* optimizer.step()  
  
 *# 统计分类情况  
 # outputs.data为16×2，即样本数×分类数  
 # torch.max(outputs.data, 1)从第一维度上求最大，即每个样本的最大值和对应的下标  
 # predicted记录了输出的两个分类较大数的下标* \_, predicted = torch.max(outputs.data, 1)  
 *# 计算总样本数* total += labels.size(0)  
 *# 计算总预测正确数，.cpu()是将张量移到cpu上，因为张量的某些操作不能再cuda上执行* correct += (predicted == labels).squeeze().cpu().sum().numpy()  
  
 *# 打印训练信息  
 # 计算总loss* loss\_mean += loss.item()  
 *# 将loss添加到train\_curve中* train\_curve.append(loss.item())  
 **if** (i+1) % log\_interval == 0:  
 *# 计算log\_interval批的平均loss* loss\_mean = loss\_mean / log\_interval  
 *# 打印当前的epoch数/总epoch数、当前批次/总批次、平均loss、准确率* print(**"Training:Epoch[{:0>3}/{:0>3}] Iteration[{:0>3}/{:0>3}] Loss: {:.4f} Acc:{:.2%}"**.format(  
 epoch, MAX\_EPOCH, i+1, len(train\_loader), loss\_mean, correct / total))  
  
 *# 对平均loss清零* loss\_mean = 0.  
  
 *# 打印卷积层权重* **if** flag\_m1:  
 print(**"epoch:{} conv1.weights[0, 0, ...] :\n {}"**.format(epoch, resnet18\_ft.conv1.weight[0, 0, ...]))  
  
 scheduler.step() *# 更新学习率*

*# 设置检查点，用来保存模型训练中的参数信息，每12轮保存一次***if** epoch % 12 == 0:  
 checkpoint = {**"model\_state\_dict"**: resnet18\_ft.state\_dict(),  
 **"optimizer\_state\_dict"**: optimizer.state\_dict(),  
 **"epoch"**: epoch}  
 path\_checkpoint = **"./checkpoint\_{}\_epoch.pkl"**.format(epoch)  
 torch.save(checkpoint, path\_checkpoint)

*# 进行测试* **if** (epoch+1) % val\_interval == 0:  
  
 correct\_val = 0.  
 total\_val = 0.  
 loss\_val = 0.  
 *# 设置模型为测试模式* resnet18\_ft.eval()  
 *# with torch.no\_grad():被该语句包裹起来的部分将不会计算梯度，因为预测阶段不更新梯度* **with** torch.no\_grad():  
 **for** j, data **in** enumerate(valid\_loader):  
 inputs, labels = data  
 inputs, labels = inputs.to(device), labels.to(device)  
  
 outputs = resnet18\_ft(inputs)  
 loss = criterion(outputs, labels)  
  
 \_, predicted = torch.max(outputs.data, 1)  
 total\_val += labels.size(0)  
 correct\_val += (predicted == labels).squeeze().cpu().sum().numpy()  
  
 loss\_val += loss.item()  
  
 loss\_val\_mean = loss\_val/len(valid\_loader)  
 valid\_curve.append(loss\_val\_mean)  
 print(**"Valid:\t Epoch[{:0>3}/{:0>3}] Iteration[{:0>3}/{:0>3}] Loss: {:.4f} Acc:{:.2%}"**.format(  
 epoch, MAX\_EPOCH, j+1, len(valid\_loader), loss\_val\_mean, correct\_val / total\_val))  
  
*# 绘图，绘制训练集的loss图*train\_x = range(len(train\_curve))  
train\_y = train\_curve  
  
train\_iters = len(train\_loader)  
*# 由于valid中记录的是epochloss，需要对记录点进行转换到iterations*valid\_x = np.arange(1, len(valid\_curve)+1) \* train\_iters\*val\_interval  
valid\_y = valid\_curve  
  
plt.plot(train\_x, train\_y, label=**'Train'**)  
plt.plot(valid\_x, valid\_y, label=**'Valid'**)  
  
plt.legend(loc=**'upper right'**)  
plt.ylabel(**'loss value'**)  
plt.xlabel(**'Iteration'**)  
plt.show()

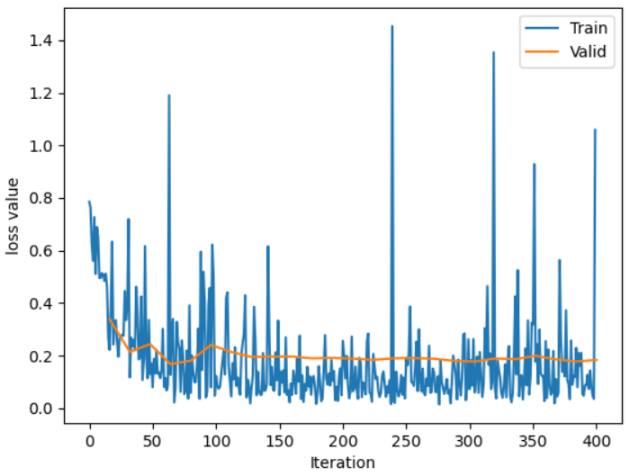
当不使用resnet18的参数，只使用模型结构自己训练时的结构如下：

|  |
| --- |
| use device :cuda  Training:Epoch[000/025] Iteration[010/016] Loss: 0.7210 Acc:48.75%  Valid: Epoch[000/025] Iteration[010/010] Loss: 0.6928 Acc:48.37%  Training:Epoch[001/025] Iteration[010/016] Loss: 0.6516 Acc:58.75%  Valid: Epoch[001/025] Iteration[010/010] Loss: 0.6630 Acc:59.48%  ...  Training:Epoch[024/025] Iteration[010/016] Loss: 0.5653 Acc:68.75%  Valid: Epoch[024/025] Iteration[010/010] Loss: 0.6032 Acc:67.97% |



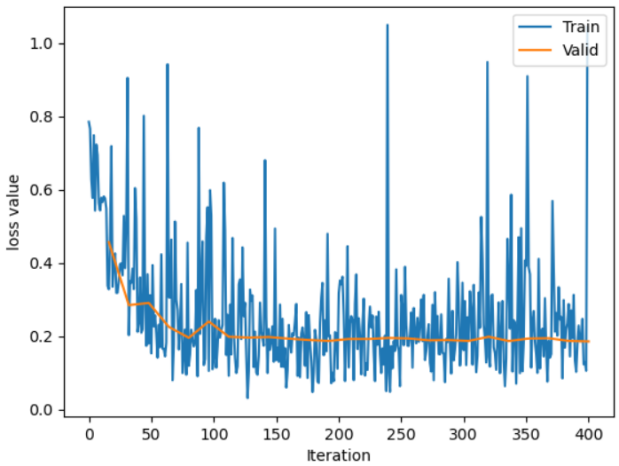
当使用resnet18的参数，但是设置统一的学习率时结果如下：

|  |
| --- |
| use device :cuda  Training:Epoch[000/025] Iteration[010/016] Loss: 0.6300 Acc:65.00%  Valid: Epoch[000/025] Iteration[010/010] Loss: 0.3385 Acc:90.20%  Training:Epoch[001/025] Iteration[010/016] Loss: 0.3122 Acc:90.00%  Valid: Epoch[001/025] Iteration[010/010] Loss: 0.2142 Acc:93.46%  ...  Training:Epoch[024/025] Iteration[010/016] Loss: 0.1293 Acc:96.25%  Valid: Epoch[024/025] Iteration[010/010] Loss: 0.1832 Acc:96.08% |



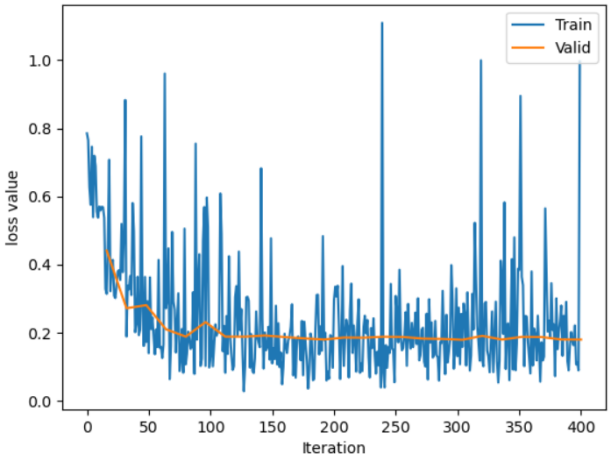
当使用resnet18的参数，并设置卷积层学习率为0时结果如下：

|  |
| --- |
| use device :cuda  Training:Epoch[000/025] Iteration[010/016] Loss: 0.6572 Acc:60.62%  Valid: Epoch[000/025] Iteration[010/010] Loss: 0.4565 Acc:84.97%  Training:Epoch[001/025] Iteration[010/016] Loss: 0.4074 Acc:85.00%  Valid: Epoch[001/025] Iteration[010/010] Loss: 0.2846 Acc:93.46%  Training:Epoch[024/025] Iteration[010/016] Loss: 0.2039 Acc:93.12%  Valid: Epoch[024/025] Iteration[010/010] Loss: 0.1854 Acc:96.73% |



当使用resnet18的参数，并设置卷积层学习率为较小值时结果如下：

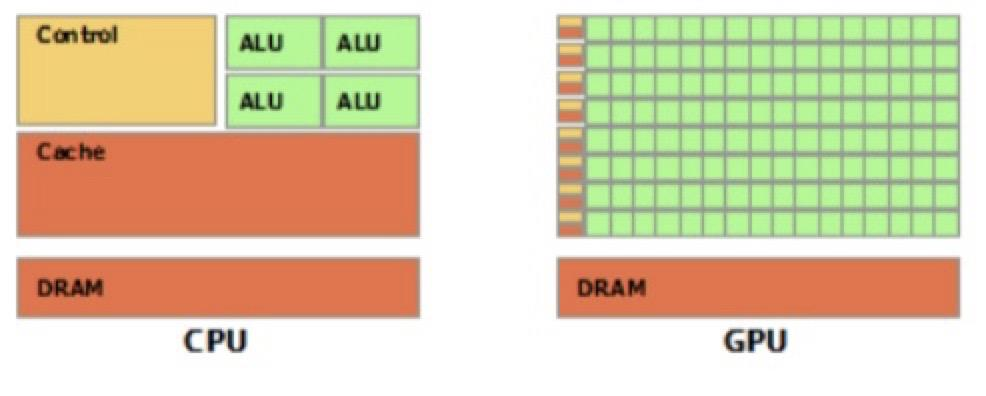
|  |
| --- |
| use device :cuda  Training:Epoch[000/025] Iteration[010/016] Loss: 0.6543 Acc:61.25%  Valid: Epoch[000/025] Iteration[010/010] Loss: 0.4412 Acc:85.62%  Training:Epoch[001/025] Iteration[010/016] Loss: 0.3948 Acc:85.62%  Valid: Epoch[001/025] Iteration[010/010] Loss: 0.2721 Acc:93.46%  ...  Training:Epoch[024/025] Iteration[010/016] Loss: 0.1888 Acc:93.12%  Valid: Epoch[024/025] Iteration[010/010] Loss: 0.1801 Acc:96.73% |



## 7.3.GPU的使用

CPU (Central Processing Unit, 中央处理器)：主要包括控制器和运算器  
 GPU (Graphics Processing Unit, 图形处理器)：处理统一的大规模数据运算

cpu与gpu示意图如下：其中绿色为计算单元



在PyTorch中提供了数据或者模型在CPU和GPU之间迁移的方法：

将数据或者模型迁移到GPU：

data.to(“cuda”)

将数据或者模型迁移到CPU：

data.to(“CPU”)

其中，data表示Tensor或者Module

关于to函数：转换数据类型/设备

张量转换数据类型：

x = torch.ones((3, 3))  
 x = x.to(torch.float64)

张量转换设备：

x = torch.ones((3, 3))  
 x = x.to("cuda")

模型转换数据类型：即模型的所有参数都会转换数据类型

linear = nn.Linear(2, 2)  
 linear.to(torch.double)

模型转换设备：需要先创建device

gpu1 = torch.device("cuda")  
 linear.to(gpu1)

注意：

张量的to函数不执行inplace，因此需要重新赋值

模型的to函数执行inplace，因此无需重新赋值

代码实现：

Tonsor转换GPU设备

*# -\*- coding: utf-8 -\*-***import** torch  
**import** torch.nn **as** nn  
*# 判断cuda是否可用，如果可用设置为cuda:0，否则设置为cpu*device = torch.device(**"cuda:0" if** torch.cuda.is\_available() **else "cpu"**)  
  
*# flag = 0*flag = 1  
**if** flag:  
 x\_cpu = torch.ones((3, 3))  
 *# 打印数据的设备信息，是否使用的cuda，以及内存地址* print(**"x\_cpu:\ndevice: {} is\_cuda: {} id: {}"**.format(x\_cpu.device, x\_cpu.is\_cuda, id(x\_cpu)))  
  
 *# 将数据迁移到cuda* x\_gpu = x\_cpu.to(device)  
 print(**"x\_gpu:\ndevice: {} is\_cuda: {} id: {}"**.format(x\_gpu.device, x\_gpu.is\_cuda, id(x\_gpu)))

|  |
| --- |
| x\_cpu:  device: cpu is\_cuda: False id: 1736317533304  x\_gpu:  device: cuda:0 is\_cuda: True id: 1736753674504 |

两次id不同，说明不是inplace操作

model转换GPU设备

*# flag = 0*flag = 1  
**if** flag:  
 net = nn.Sequential(nn.Linear(3, 3))  
 *# 打印模型的地址以及参数是否在cuda上* print(**"\nid:{} is\_cuda: {}"**.format(id(net), next(net.parameters()).is\_cuda))  
  
 net.to(device)  
 print(**"\nid:{} is\_cuda: {}"**.format(id(net), next(net.parameters()).is\_cuda))

|  |
| --- |
| id:2716271949896 is\_cuda: False  id:2716271949896 is\_cuda: True |

两次id相同，说明是inplace操作

查看gpu上的模型与数据的运行结果

*# flag = 0*flag = 1  
**if** flag:  
 output = net(x\_gpu)  
 print(**"output is\_cuda: {}"**.format(output.is\_cuda))  
  
 *# 如果模型在gpu而数据在cpu，则无法运行  
 # output = net(x\_cpu)*

|  |
| --- |
| output is\_cuda: True |

说明如果模型与数据都在gpu上，则输出也在gpu上

PyTorch中提供了一些cuda相关的常用方法：

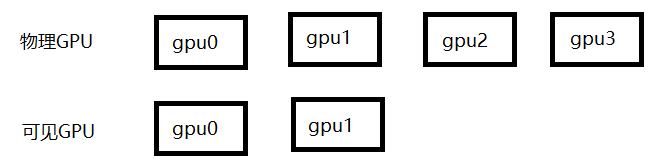
1. torch.cuda.device\_count()：计算当前可见可用gpu数  
 2. torch.cuda.get\_device\_name()：获取gpu名称  
 3. torch.cuda.manual\_seed()：为当前gpu设置随机种子  
 4. torch.cuda.manual\_seed\_all()：为所有可见可用gpu设置随机种子

5. os.environ.setdefault("CUDA\_VISIBLE\_DEVICES", "2, 3")：设置当前可见的GPU

物理gpu：电脑自带的gpu硬件

可见gpu：PyTorch可用的gpu

例如：当设置可见gpu为2、3时，则可见gpu有两个，可见gpu0对应物理gpu2，可见gpu1对应物理gpu3



多GPU运行时，有一个主GPU，默认为gpu0，负责分发数据到其他GPU以及从其他GPU收集处理后的结果。

多gpu运算的分发并行机制：

torch.nn.DataParallel  
 功能：包装模型，实现分发并行机制  
 • module: 需要包装分发的模型  
 • device\_ids: 可分发的gpu，默认分发到所有可见可用gpu  
 • output\_device: 结果输出设备

torch.nn.DataParallel(module, device\_ids=None,  
 output\_device=None, dim=0)

代码实现：

*# -\*- coding: utf-8 -\*-***import** os  
**import** numpy **as** np  
**import** torch  
**import** torch.nn **as** nn  
  
*# 配置可见的gpu的list，如果此电脑只有一个gpu，则只能配置0*gpu\_list = [0] *# [2, 3]  
# 将gpu的格式变为 2,3 格式*gpu\_list\_str = **','**.join(map(str, gpu\_list))  
*# 配置物理gpu为可见gpu*os.environ.setdefault(**"CUDA\_VISIBLE\_DEVICES"**, gpu\_list\_str)  
device = torch.device(**"cuda" if** torch.cuda.is\_available() **else "cpu"**)  
  
**class** FooNet(nn.Module):  
 **def** \_\_init\_\_(self, neural\_num, layers=3):  
 super(FooNet, self).\_\_init\_\_()  
 self.linears = nn.ModuleList([nn.Linear(neural\_num, neural\_num, bias=**False**) **for** i **in** range(layers)])  
  
 **def** forward(self, x):  
 *# 这里打印的是每个gpu上batch\_size的数量，即除以gpu数量的结果* print(**"\nbatch size in forward: {}"**.format(x.size()[0]))  
  
 **for** (i, linear) **in** enumerate(self.linears):  
 x = linear(x)  
 x = torch.relu(x)

**return** x  
  
**if** \_\_name\_\_ == **"\_\_main\_\_"**:  
  
 batch\_size = 16  
  
 *# data* inputs = torch.randn(batch\_size, 3)  
 labels = torch.randn(batch\_size, 3)  
  
 inputs, labels = inputs.to(device), labels.to(device)  
  
 *# model* net = FooNet(neural\_num=3, layers=3)  
 *# 将模型用DataParallel包装* net = nn.DataParallel(net)  
 *# 将模型交给gpu，这样就会将一个batch\_size的数据分发到所有可见的gpu上* net.to(device)  
  
 *# 训练过程* **for** epoch **in** range(1):  
 outputs = net(inputs)  
  
 print(**"model outputs.size: {}"**.format(outputs.size()))  
  
 print(**"CUDA\_VISIBLE\_DEVICES :{}"**.format(os.environ[**"CUDA\_VISIBLE\_DEVICES"**]))  
 print(**"device\_count :{}"**.format(torch.cuda.device\_count()))

|  |
| --- |
| batch size in forward: 16  model outputs.size: torch.Size([16, 3])  CUDA\_VISIBLE\_DEVICES :0  device\_count :1 |

gpu模型加载问题：

如果数据在gpu上时，当保存数据后，再次加载保存数据的文件时，默认会加载到gpu上，但是如果加载模型的机器没有gpu，会导致运行时报错，因此在加载模型时可以指定加载到cpu中。

如：torch.load(path\_state \_dict, map\_location="cpu")

代码实现：

*# -\*- coding: utf-8 -\*-***import** os  
**import** numpy **as** np  
**import** torch  
**import** torch.nn **as** nn  
  
**class** FooNet(nn.Module):  
 **def** \_\_init\_\_(self, neural\_num, layers=3):  
 super(FooNet, self).\_\_init\_\_()  
 self.linears = nn.ModuleList([nn.Linear(neural\_num, neural\_num, bias=**False**) **for** i **in** range(layers)])  
  
 **def** forward(self, x):  
 print(**"\nbatch size in forward: {}"**.format(x.size()[0]))  
  
 **for** (i, linear) **in** enumerate(self.linears):  
 x = linear(x)  
 x = torch.relu(x)

**return** x  
  
*# 将数据加载至cpu*gpu\_list = [0]  
gpu\_list\_str = **','**.join(map(str, gpu\_list))  
os.environ.setdefault(**"CUDA\_VISIBLE\_DEVICES"**, gpu\_list\_str)  
device = torch.device(**"cuda" if** torch.cuda.is\_available() **else "cpu"**)  
  
net = FooNet(neural\_num=3, layers=3)  
net.to(device)  
  
*# save*net\_state\_dict = net.state\_dict()  
path\_state\_dict = **"./model\_in\_gpu\_0.pkl"**torch.save(net\_state\_dict, path\_state\_dict)  
  
*# load  
# 默认加载时，数据是以device="cuda"的形式加载的，但是如果加载的机器没有cuda，则会报错*state\_dict\_load = torch.load(path\_state\_dict)  
  
*# 为了解决上面的问题，加载时，可以将参数加载到cpu上，这样就不会报错  
# state\_dict\_load = torch.load(path\_state\_dict, map\_location="cpu")*print(**"state\_dict\_load:\n{}"**.format(state\_dict\_load))

|  |
| --- |
| state\_dict\_load:  OrderedDict([('linears.0.weight', tensor([[-0.4292, -0.5397, -0.0086],  [ 0.1546, 0.5700, 0.5638],  [ 0.1969, 0.5569, -0.1533]], device='cuda:0')), ('linears.1.weight', tensor([[ 0.5638, 0.1799, -0.1404],  [ 0.2120, 0.2129, 0.0515],  [-0.3619, -0.1249, 0.0383]], device='cuda:0')), ('linears.2.weight', tensor([[-0.2883, 0.3434, 0.0235],  [ 0.2079, -0.3871, 0.4005],  [ 0.5305, 0.5581, 0.3371]], device='cuda:0'))]) |

可以看到默认数据是加载到gpu上的

## 7.4.PyTorch常见报错

1.报错： ValueError: num\_samples should be a positive integer value, but got num\_samples=0

可能的原因：传入的Dataset中的len(self.data\_info)==0，即传入该dataloader的dataset里没有数据

解决方法：

1. 检查dataset中的路径，路径不对，读取不到数据。

2. 检查Dataset的\_\_len\_\_()函数为何输出为零

2报错：TypeError: pic should be PIL Image or ndarray. Got <class 'torch.Tensor'>

可能的原因：当前操作需要PIL Image或ndarray数据类型，但传入了Tensor

解决方法：

1. 检查transform中是否存在两次ToTensor()方法

2. 检查transform中每一个操作的数据类型变化

3报错：RuntimeError: invalid argument 0: Sizes of tensors must match except in dimension 0. Got 93 and 89 in dimension 1 at /Users/soumith/code/builder/wheel/pytorch-src/aten/src/TH/generic/THTensorMath.cpp:3616

可能的原因：dataloader的\_\_getitem\_\_函数中，返回的图片形状不一致，导致无法stack

解决方法：

检查\_\_getitem\_\_函数中的操作

4报错：conv: RuntimeError: Given groups=1, weight of size 6 1 5 5, expected input[16, 3, 32, 32] to have 1 channels, but got 3 channels instead

linear: RuntimeError: size mismatch, m1: [16 x 576], m2: [400 x 120] at ../aten/src/TH/generic/THTensorMath.cpp:752

可能的原因：网络层输入数据与网络的参数不匹配

解决方法：

1. 检查对应网络层前后定义是否有误

2. 检查输入数据shape

5报错：AttributeError: 'DataParallel' object has no attribute 'linear'

可能的原因：并行运算时，模型被dataparallel包装，所有module都增加一个属性 module. 因此需要通过 net.module.linear调用

解决方法：

网络层前加入module.

6报错: RuntimeError: Attempting to deserialize object on a CUDA device but torch.cuda.is\_available() is False. If you are running on a CPU-only machine, please use torch.load with map\_location=torch.device('cpu') to map your storages to the CPU.

可能的原因：gpu训练的模型保存后，在无gpu设备上无法直接加载

解决方法：

需要设置map\_location="cpu"

7报错：AttributeError: Can't get attribute 'FooNet2' on <module '\_\_main\_\_' from '

可能的原因：保存的网络模型在当前python脚本中没有定义

解决方法：

提前定义该类

8报错：RuntimeError: Assertion `cur\_target >= 0 && cur\_target < n\_classes' failed. at ../aten/src/THNN/generic/ClassNLLCriterion.c:94

可能的原因：

标签数大于等于类别数量，即不满足 cur\_target < n\_classes，通常是因为标签从1开始而不是从0开始

解决方法：

修改label，从0开始，例如：10分类的标签取值应该是0-9

9报错：RuntimeError: expected device cuda:0 and dtype Long but got device cpu and dtype Long

Expected object of backend CPU but got backend CUDA for argument #2 'weight'

可能的原因：需计算的两个数据不在同一个设备上

解决方法：

采用to函数将数据迁移到同一个设备上

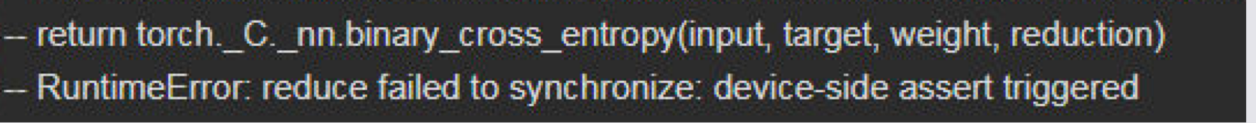
10报错：RuntimeError: DataLoader worker (pid 27) is killed by signal: Killed. Details are lost due to multiprocessing. Rerunning with num\_workers=0 may give better error trace.

可能原因：内存不够（不是gpu显存，是内存）

解决方法：

申请更大内存

11报错：



RuntimeError: reduce failed to synchronize: device-side assert triggered

可能的原因：采用BCE损失函数的时候，input必须是0-1之间，由于模型最后没有加sigmoid激活函数，导致的。

解决方法：

让模型输出的值域在[0, 1]

12报错：RuntimeError: unexpected EOF. The file might be corrupted.

可能的原因：torch.load加载模型过程报错，因为模型传输过程中有问题，重新传一遍模型即可

13 报错：UnicodeDecodeError: 'utf-8' codec can't decode byte 0xff in position 1: invalid start byte

可能的原因：python2保存，python3加载，会报错

解决方法：

把encoding改为encoding='iso-8859-1'

check\_p = torch.load(path, map\_location="cpu", encoding='iso-8859-1')

14报错：RuntimeError: Input type (torch.cuda.FloatTensor) and weight type (torch.FloatTensor) should be the same

问题原因：数据张量已经转换到GPU上，但模型参数还在cpu上，造成计算不匹配问题。

解决方法：

通过添加model.cuda()将模型转移到GPU上以解决这个问题。或者通过添加model.to(cuda)解决问题

15报错：RuntimeError: cuDNN error: CUDNN\_STATUS\_INTERNAL\_ERROR

问题原因：jupyter notebook中调用了cuda，但没有释放

解决方法：

把对应的ipynb文件shutdown就可以了

16报错: RuntimeError: CUDA out of memory. Tried to allocate 46.00 MiB (GPU 0; 2.00 GiB total capacity; 54.79 MiB already allocated; 39.30 MiB free; 74.00 MiB reserved in total by PyTorch)

原因: 可以看出在GPU充足的情况下无法使用,本机有两个GPU,其中一个GPU的内存不可用?

解决办法:

在model文件(只有model中使用了cuda)添加下面两句:

import os

os.environ['CUDA\_VISIBLE\_DEVICES']='2, 3'