

torch 笔记

徐世桐

1 import

```
from torch import nn, optim
import torch.nn.functional as F
from torchvision import models, transforms, utils.make_grid
from PIL import Image
from numpy.random import default_rng
import 自定义函数
```

python_file: 调用 import 的 python 文件

程序 path: python_file 的文件夹 path

运行 path: 执行 python python_file 时终端所在的 path

import path: 被 import 的程序 path (包含文件名)

1. 当程序 path 和运行 path 相同

```
from directory. ... .python_file import func
```

2. 当程序 path 和运行 path 不同时

```
import sys
```

```
sys.path.append(PATH1)
```

```
from PATH2 import func
```

保证 PATH1 + PATH2 = import path 即可

sys.path 中已包含程序 path, 但不可使用 import ..path 得到 不为程序 path 子文件夹内的程序

3. 在 jupyter-notebook 中运行时

方法 1: 2. 中 path 需包含至运行 path 最底文件夹位置

方法 2: 代码同 2. 在 import 中的 path 每一文件夹下创建空 __init__.py 文件, 重启 jupyter kernel 后运行

得到运行 path:

```
import os
```

```
path = os.path.abspath(os.path.join('.'))
```

得到现有 sys path:

```
import sys
```

```
path = sys.path
```

import path: 作用为在每一 sys.path 后连接 path

2 tensor 使用 GPU

```
if torch.cuda.is_available():
dev = "cuda:0"
else:
dev = "cpu"
device = torch.device(dev)
    得到 device 数据类型, 定义使用 CPU 或使用哪一 GPU
Tensor.get_device()
    查看张量存在的 CPU/GPU
```

3 tensor 数据类型

```
torch.arange
    torch.arange(a) 得到 Tensor[0, 1, ..., [a]]
    torch.arange(a, b) 得到 Tensor[a, a + 1, ..., a + n], n 为整数且 a + n < b
    torch.arange(a, b, c) 得到 Tensor[a, a + c, ..., a + nc], n 为整数且 a + nc < b
    torch.arange(..., requires_grad=True) 分配空间记录斜率, 同 mxnet 的 attach_grad()
    torch.arange(..., device=device 数据类型) 将张量分配进指定 CPU/GPU
torch.tensor([], REQUIRES_GRAD, DEVICE)
    通过 python 数组创建 Tensor
    REQUIRES_GRAD = True 分配空间记录斜率
    DEVICE = device 数据类型将张量分配进指定 CPU/GPU
torch.numpy() 得到 numpy array
torch.tolist() 得到 python 数组
Torch.item() 当张量中仅有一个元素, 得到此元素
torch.from_numpy(NDArray) 从 NDArray 创建 Tensor
torch.mm(Tensor, Tensor) 矩阵 向量或矩阵 矩阵乘法
torch.bmm(Tensor, Tensor) 张量最后两维进行矩阵乘法, 其余维长度必须相等
torch.matmul(Tensor, Tensor)
    任意张量间乘法
    张量间乘法: 两张量最后两维符合矩阵乘法, 高维使用广播机制广播矩阵
+ - * /
    同 NDArray 使用广播机制
    广播机制要求: 形状从最后一维开始, 每一维元素/张量个数相同 或 其中一个数为 1
torch.sum(Tensor, *DIM) 将某一纬度一下的值求和
torch.max(Tensor, *DIM)
    当没有指定维度 DIM, 求所有 Tensor 元素最大值
    当给定纬度 DIM, 返回 (Tensor 1, Tensor 2) 元组。第一张量为给定维度最大元素, 第二张量为
    给定维度最大元素 index
Tensor[:, ..., None]
```

'[]' 中前 n 位置为 ':', 则在第 n 至 第 $n+1$ 维间插入一维

则形状由 $(s_1, \dots, s_n, s_{n+1}, \dots)$ 转为 $(s_1, \dots, s_n, 1, s_{n+1}, \dots)$

可使用多个 None 加入多个 1 元素维

`Tensor.unsqueeze(dim*)`

在第 $\text{dim}-1$ 至 dim 维加入一维, 同 `Tensor[(: * dim), None]`

`Tensor.squeeze(dim)`

若第 dim 维为 1 元素维 (即 `Tensor.shape[dim] == 1`), 则删除此维。

若 dim 为 None, 则将所有 1 元素维删除

`Tensor.reshape()` 改变形状, 新形状元素数必须等于输入元素数

`Tensor.permute(SHAPE)` 等同 mxnet 的 `transpose`, 更改纬度顺序

`Tensor.transpose(dim0, dim1)` 交换两个纬度

`Tensor.masked_fill(MASK_Tensor, VALUE)`

根据 MASK_Tensor 中 True 值的位置将 Tensor 中对应位置设为 VALUE

MASK_Tensor 为 Boolean tensor, VALUE 为标量

`torch.vstack([Tensor, Tensor, ...])` 在第 0 维方向连接张量, (对矩阵即在竖直方向上连接)

`torch.hstack([Tensor, Tensor, ...])` 在第 1 维方向连接张量, (对矩阵即在水平方向上连接)

即 `vstack` 的形状第 0 位置值可不同, `hstack` 形状第 1 位置可不同

在循环中加入行得到张量, 初始化张量为 `torch.tensor([]).reshape(..., 0, ...)`

`Tensor.nonzero()` 得到一 list 的 index 向量 $\{v_i\}$, `Tensor[v_i]` 为非零值

`Tensor.to(device, NON-BLOCKING)`

将张量分配进指定 CPU/GPU

NON-BLOCKING 为 true 时此步不 `synchronize`, 新 device 得到部分数据即开始下一步计算。默认为 false

`Tensor = Tensor.type(torch.float64) Np.astype('float')` 转换类型

`F.one_hot(Tensor, NUM_CLASS)`

将 Tensor 中每一元素转为 NUM_CLASS 长度的 onehot 向量, NUM_CLASS 默认为 Tensor 中最大值

`torch.normal(MEAN, STD, SIZE*)`

`size=(x_1, x_2, \dots)` 限定输出张量形状

`mean=Tensor, std=Tensor/const` 当没有限定 size 时 mean 必为 float Tensor, 形状和输出形状相同。

`mean=Tensor/const, std=Tensor/const` 当限定 size 后 mean, std 可为 const 或单个值的 Tensor

`torch.rand(SIZE*)`

得到 SIZE 形状的随机数张量, 每一元素 $\in [0, 1)$ 。SIZE 无定义则得到 const 随机数

代替 `torch.uniform` 功能

`dataset = torch.utils.data.TensorDataset(样本 Tensor, 标签 Tensor)`

`dataiter = torch.utils.data.DataLoader(dataset, batch_size= 批量大小, shuffle=True)`

使用 torch 进行批量迭代

dataiter 输出的 feature, label 使用的 CPU/GPU 和样本 Tensor, 标签 Tensor 使用的 CPU/GPU 分别对应

`torch.save(Tensor, ' 文件名')` 文件中保存一张量

```
Tensor = torch.load(' 文件名') 读取文件中张量
torch.save("model": net, "loss": ..., ' 文件名') 保存一 checkpoint
checkpoint = torch.load(' 文件名') 读取一 checkpoint
model.load_state_dict(checkpoint['model'].state_dict()) 从 checkpoint 中读取参数
torch.cuda.synchronize()
    等待所有 GPU 异步计算结束, 打印结果同样等待异步计算
    若两计算操作处在不同 device 上, 且没有相互依靠关系/没有等待异步结果分隔, 则并行两 device
    计算
    当一计算 b 需要另一计算 a 结果, 则每当 a 得出结果部分, b 即开始处理, 无需等待 a 输出所
    有结果才开始 b
random_generator = default_rng()
shuffled_index = random_generator.permutation(列表大小)
    得到乱序 index 列表
```

4 torch 神经网络

```
net = nn.Sequential()
神经网络定义: net.add_module(' 层名', 层)
层定义:
    nn.Linear(输入节点数, 输出节点数) 定义全连接层
        当全连接层输入为 2 维以上的张量时, 全连接层仅对最低维操作。即最低维元素数同输入节点数
        可使用层直接进行前向计算, 训练函数中使用 [layer.weight, net.bias] 传入参数
        前向计算为 ( $|B|$ , 特征数) 和 权重 矩阵相乘
        使用 GPU 时层定义后需加.to(device), 并不可以使用 device= 赋 GPU
net.weight/bias.data.fill_(值) 对层中所有权重/偏差赋同一值
net.weight/bias = nn.Parameter(Tensor) 将参数初始化为指定张量, 参数参与反向传播, 作为 .parameters()
的输出之一
net.weight/bias = nn.ModuleList([nn.Module]) 参数初始化为一列表的 module, 参数参与反向传
播
nn.init.xavier_uniform_(net.weight/.bias) 对层中所有权重/偏差使用 xavier 初始化
nn.init.normal_(net.weight/.bias, MEAN, STD) 对层使用 normal 初始化
def init_func(layer):
    if isinstance(layer, nn.Linear):
        // 根据上一条笔记更新 layer 的参数
net.apply(init_func) // 对每一层参数初始化权重 偏差
rnn = nn.RNN(...)
    INPUT_SIZE: 一样本特征数
    HIDDEN_SIZE: RNN 隐藏层 neuron 数
    NUM_LAYERS: RNN 隐藏层数, 每层隐藏层都有 HIDDEN_SIZE neuron。default 1
    NONLINEARITY: 激活函数, 可 'relu' 或 'tanh' 字符创。default 为 'tanh'
    BIAS: hidden 计算中是否使用  $b_x$  和  $b_h$ 。default 为 true
```

BATCH_FIRST: `false` 时输入 `X` 为 (时间步数, 批量大小, 特征数), `true` 时 `X` 为 (批量大小, 时间步数, 特征数)。default 为 `false`

DROPOUT: 定义每一隐藏层后 `dropout` 层的几率, default 为 0。即不使用 `dropout`

BIDIRECTIONAL: 是否为双向神经网络, default 为 `false`

`rnn` 不可用 `.weight/.bias` 取参数

`rnn(X, H)`

计算 $H, H_n = \sigma(XW_{xh} + b_x + H_nW_{hh} + b_h)$

`X` 形状见 `RNN BATCH_SIZE` 变量说明

`H` 形状 (隐藏层数, 时间步数, 每层 `neuron` 数, 特征数)。双向 `RNN` 中第一维值 $\times 2$

H_n 为所有隐藏层的输出,

输出仅为隐藏状态, 不包含全连接层计算

H 为参与最后全连接层计算的输出, 即所有时间步的隐藏状态。

H_n 为参与下一次计算的隐藏状态张量, 即最后一时间步的隐藏状态, 与 H 中最后一张量相同

`loss`

= `nn.MSELoss(REDUCTION*)` 平方代价函数

`REDUCTION = 'none' | 'mean' | 'sum'` 得到每一样本代价值向量 | 得到平均代价 | 得到代价值和。默认为 `'mean'`

= `nn.CrossEntropyLoss()` `catagorical` 交叉熵损失函数, 已经包含 `softmax` 计算

= `nn.BCELoss()` 二元交叉熵损失函数

= `nn.BCEWithLogitsLoss()` 包含 `sigmoid` 的二元交叉熵损失函数, 数值稳定性更高

`trainer`

= `optim.SGD(net.parameters(), lr= 学习率)` `SGD` 迭代函数

= `optim.Adam(net.parameters(), lr= 学习率)` `Adam-SGD` 迭代

`trainer.step()` 进行迭代

每一迭代中 `trainer.grad_zero()` 清零斜率, 否则训练斜率为随机值, 代价值在某一高值波动

`net.parameters()` 得到权重

`list(net.parameters())` 得到 `param` 类型数组, 包含 [第一层权重, 第一层偏差, ..., 最后一层参数]

`param` 类型数组.`data` 得到参数张量

`param` 类型数组.`name` 得到所属层名, 可为空

`loss(y_hat, y).backward()` 得到代价函数值, 求导

不会调用 `.sum()` 或 `.mean()`, 求和方法在 `loss` 函数中定义

对同一网络的输出调用多次 `backward` 会将斜率叠加, 不会覆盖斜率

`Tensor.detach()`

当此 `Tensor` 作为另一神经网络输入时, `detach` 导致此张量不参与反向传播。即不对此张量和得到此张量的计算求导

`Tensor.require_grad=False`

前向计算中仍记录数值用于反向计算, 但调用 `step()` 不会更新参数

`class out_image(nn.Module):`

`def __init__(self):`

`super().__init__()`

```
def forward(self, x):
```

自定义神经网络

`net = models.NET_NAME(pretrained=True)` 得到预训练的神经网络

在多个 GPU 上训练神经网络

方法 1: `network partition`: 将网络按深度分配在多个 GPU 上

方法 2: `layer-wise partition`: 按通道分配在多 GPU, AlexNet 即此方法。

需要 `synchronize` 前一层结果, 比方法 1 传输更多数据。时间 `complexity` 更高

方法 3: `data partition`: 不同 GPU 训练不同 `batch` 数据, 分别反向传播。

得到斜率后求平均, 传入每一 GPU。每一 GPU 内模型迭代使用同一平均斜率

简单但无法训练更大模型

`torch.cuda.device_count()` 得到可用 `gpu` 数量 `i`, `cuda:0-cuda:(i-1)` 共 `i` 个 `gpu` 全部可

用

`net = nn.DataParallel(net, device_ids=DEVICES)` `DEVICES` 为一 `list` 的 `torch.device`

`net(X.to(DEVICE))` 计算时只需将数据放在同一类型 (`cpu/gpu`) `device` 上即可, `nn.DataParallel`

自动将数据分段

实现: `ring synchronization`

`gpu` 间连接为 `ring topology`

合并斜率时每一 `gpu` 同时向下一 `gpu` 传输自身斜率。进行 `n-1` 次传输后所有 `gpu` 得到其余

所有 `gpu` 参数

5 读取图像

```
image = Image.open(' 图像路径')
```

得到图片, 显示图片直接调取 `image.show()`, 显示结果不阻断 `python` 程序。

```
transform = transforms.Compose([trans1, trnas2, ...])
```

合并多个对图像的变换

`transforms.Resize(图片形状)` 缩放图片

`transforms.ToTensor()` 图片变为张量

`transforms.Normalize(MEAN, STD)` 对图片的张量输入, 求标准化。MEAN, STD 可为张量

内部实现: 对 RGB 3 通道上的像素分别使用 (3,) 形状的 MEAN, STD 值求标准化

`transform(image)` 使用 `transform`

显示图片

```
image = transforms.ToPILImage()(image_tensor)
```

```
image.show()
```

```
plt.figure((h, w)) 限制每一子图大小
```

或使用自定义包, 支持反标准化

```
import sys
```

```
sys.path.append('../machine_learning/')
```

```
from utils.functions import show_tensor_image, un_normalize_image
```

```
show_tensor_image(un_normalize_image(image.reshape((3, height, width)), image_mean,
```

```
image_std))
```

显示多行图片，图片大小相同

```
grids = make_grid([Tensor], nrow, padding)
plt.imshow(grids.permute((1, 2, 0)))
plt.show()
```

`nrow` 定义一行图片个数

`padding` 定义连接 2 图像时间隔像素数

`make_grid` 得到 (n, c, h, w) 形状张量，即 n 张图片，每一图使用 c 通道。返回将所有图片连接结果，形状为 $(3, h', w')$

并行显示图片，图片大小可不同

```
plt.subplot(abc)
plt.plot
```

共 a 行 b 列，在第 c 图片位置画图

或：

```
fig, axarr = plt.subplots(a, b, figsize=(h, w))
fig.suptitle(TITLE) 设置图片标题
axarr[i, j].imshow(Tensor) 填充图片信息
```

6 常见错误

调用 `trainer.zero_grad()`

否则参数代价值高，并迭代后不下降

使用网络层作为权重，不参与斜率计算时 调用 `layer.requires_grad_(False)`

否则调用 `loss.backward()` 时提示需要 `retain_graph=True`，由于反向传播在错误的试图更新网络权重

`transforms.ToPILImage()` 不保证像素值在 $[0,1]$ 区间，需调用 `image_tensor.clamp(min=0, max=1)`

`d2lzh` 自动对图像做 `clip`，保证值在 $[0,1]$ 区间

否则图像中包含突出像素点，如红 紫 蓝像素。

循环中更改 `Tensor` 值并将 `Tensor` 加入数组，使用 `Tensor.clone()` 复制斜率

否则下一迭代可能更改上一迭代已经加入数组的张量

前向计算中不能调用 `Tensor.detach()`

否则此项无法求斜率，无法进行迭代

`retain_graph=True`

当一次前向计算保存的记录需要被再次使用，设置 `retain_graph=True`

由于调用 `.backward()` 后前向传播记录被清空