# Pytorch使用说明



训练阶段有后向计算，测试则不需要。测试数据不需要增强操作

## 数据增强

transforms.Compose([

transforms.CenterCrop(10),

transforms.ToTensor(),

### Resize

图像尺寸变化

torchvision.transforms.Resize(size, interpolation=2)

### 标准化

对数据按通道进行标准化，即先减均值，再除以标准差，注意是 hwc

torchvision.transforms.Normalize(mean, std)

### 转为 Tensor

将 PIL Image 或者 ndarray 转换为 tensor，并且归一化至[0-1]

torchvision.transforms.ToTensor

### 中心裁剪 CenterCrop

依据给定的 size 从中心裁剪

torchvision.transforms.CenterCrop(size)

### 随机裁剪 RandomCrop

依据给定的 size 随机裁剪

torchvision.transforms.RandomCrop(size,padding=0,pad\_if\_needed=False)

### 随机长宽比裁剪 RandomResizedCrop

随机大小，随机长宽比裁剪原始图片，最后将图片 resize 到设定好的 size

torchvision.transforms.RandomResizedCrop(size, scale=(0.08, 1.0), ratio=(0.75, 1.33), interpolation=2)

### 上下左右中心裁剪 FiveCrop

对图片进行上下左右以及中心裁剪，获得 5 张图片，返回一个 4D-tensor

torchvision.transforms.FiveCrop(size)

### 上下左右中心裁剪后翻转

对图片进行上下左右以及中心裁剪，然后全部翻转（水平或者垂直），获得 10 张图片，返回一个 4D-tensor。

torchvision.transforms.TenCrop(size, vertical\_flip=False)

### 3-1 随机水平翻转 RandomHorizontalFlip

依据概率 p 对 PIL 图片进行水平翻转

torchvision.transforms.RandomHorizontalFlip(p=0.5)

### 随机垂直翻转 RandomVerticalFlip

依据概率 p 对 PIL 图片进行垂直翻转

torchvision.transforms.RandomVerticalFlip(p=0.5)

### 随机旋转 RandomRotation

依 degrees 随机旋转一定角度

torchvision.transforms.RandomRotation(degrees, resample=False, expand=False, center=None)

### 亮度对比度饱和度变换

修改亮度、对比度和饱和度

torchvision.transforms.ColorJitter(brightness=0, contrast=0, saturation=0, hue=0)

### 转灰度图

将图片转换为灰度图

torchvision.transforms.Grayscale(num\_output\_channels=1)

### 随机转灰度图

依概率 p 将图片转换为灰度图

torchvision.transforms.RandomGrayscale(p=0.1)

### 线性变换

对矩阵做线性变化，可用于白化处理

torchvision.transforms.LinearTransformation(transformation\_matrix)

### 仿射变换

torchvision.transforms.RandomAffine(degrees, translate=None, scale=None, shear=None, resample=False, fillcolor=0)

### 填充

torchvision.transforms.Pad(padding, fill=0, padding\_mode='constant')

### 概率transforms

给一个 transform 加上概率，依概率进行操作

torchvision.transforms.RandomApply(transforms, p=0.5)

### 随机transforms

从给定的一系列 transforms 中选一个进行操作

torchvision.transforms.RandomChoice(transforms)

### 乱序transforms

将 transforms 中的操作随机打乱

torchvision.transforms.RandomOrder(transforms)

## 数据导入与加载

### 数据导入

1. 已定义读取函数

datasets.ImageFolder(root="root folder path", [transform, target\_transform])

root为待导入文件，root文件下每个子文件代表着一类，自动加注标签。

1. 自定义读取函数

3.数据加载器

DataLoader(dataset, batch\_size=1, shuffle=False, sampler=None, num\_workers=0, collate\_fn=<function default\_collate>, pin\_memory=False, drop\_last=False)

Dataset为定义的读取函数，

## 模型搭建

需求包：import torch

import torch.nn.functional as F

from collections import OrderedDict

### 第一种方法

# Method 1 -----------------------------------------

class Net1(torch.nn.Module):

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

super(Net1, self).\_\_init\_\_()

self.conv1 = torch.nn.Conv2d(3, 32, 3, 1, 1)

self.dense1 = torch.nn.Linear(32 \* 3 \* 3, 128)

self.dense2 = torch.nn.Linear(128, 10)

def forward(self, x):

x = F.max\_pool2d(F.relu(self.conv(x)), 2)

x = x.view(x.size(0), -1)

x = F.relu(self.dense1(x))

x = self.dense2(x)

return x

print("Method 1:")

model1 = Net1()

### 第二种

# Method 2 ------------------------------------------

class Net2(torch.nn.Module):

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

super(Net2, self).\_\_init\_\_()

self.conv = torch.nn.Sequential(

torch.nn.Conv2d(3, 32, 3, 1, 1),

torch.nn.ReLU(),

torch.nn.MaxPool2d(2))

self.dense = torch.nn.Sequential(

torch.nn.Linear(32 \* 3 \* 3, 128),

torch.nn.ReLU(),

torch.nn.Linear(128, 10)

)

def forward(self, x):

conv\_out = self.conv1(x)

res = conv\_out.view(conv\_out.size(0), -1)

out = self.dense(res)

return out

print("Method 2:")

model2 = Net2()

torch.nn.Sequential（）容器进行快速搭建，模型的各层被顺序添加到容器中。缺点是每层的编号是默认的阿拉伯数字，不易区分。

### 第三种

# Method 3 -------------------------------

class Net3(torch.nn.Module):

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

super(Net3, self).\_\_init\_\_()

self.conv=torch.nn.Sequential()

self.conv.add\_module("conv1",torch.nn.Conv2d(3, 32, 3, 1, 1))

self.conv.add\_module("relu1",torch.nn.ReLU())

self.conv.add\_module("pool1",torch.nn.MaxPool2d(2))

self.dense = torch.nn.Sequential()

self.dense.add\_module("dense1",torch.nn.Linear(32 \* 3 \* 3, 128))

self.dense.add\_module("relu2",torch.nn.ReLU())

self.dense.add\_module("dense2",torch.nn.Linear(128, 10))

def forward(self, x):

conv\_out = self.conv1(x)

res = conv\_out.view(conv\_out.size(0), -1)

out = self.dense(res)

return out

print("Method 3:")

model3 = Net3()

print(model3)

这种方法是对第二种方法的改进：通过add\_module()添加每一层，并且为每一层增加了一个单独的名字。

### 第四种

# Method 4 ------------------------------------------

class Net4(torch.nn.Module):

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

super(Net4, self).\_\_init\_\_()

self.conv = torch.nn.Sequential(

OrderedDict(

[

("conv1", torch.nn.Conv2d(3, 32, 3, 1, 1)),

("relu1", torch.nn.ReLU()),

("pool", torch.nn.MaxPool2d(2))

]

))

self.dense = torch.nn.Sequential(

OrderedDict([

("dense1", torch.nn.Linear(32 \* 3 \* 3, 128)),

("relu2", torch.nn.ReLU()),

("dense2", torch.nn.Linear(128, 10))

])

)

def forward(self, x):

conv\_out = self.conv1(x)

res = conv\_out.view(conv\_out.size(0), -1)

out = self.dense(res)

return out

print("Method 4:")

model4 = Net4()

print(model4)

是第三种方法的另外一种写法，通过字典的形式添加每一层，并且设置单独的层名称

## 权重初始化方法（pytorch）

以下tensor均为n维的torch.Tensor或autograd.Variable，fan\_in、fan\_out为输入输出维度

调用函数torch.nn.init

### 1.常数初始化

constant\_(tensor, val)

val为填充的常数

### 2.均匀初始化

uniform\_(tensor, a=0, b=1)

从a到b均匀填充

### 3.正态分布初始化

normal\_(tensor, mean=0, std=1)

mean，std为均值和标准差

### 4.Xavier均匀初始化

xavier\_uniform\_(tensor, gain=1)

值均匀填充自U(-a, a)，其中a= gain \* sqrt( 2/(fan\_in + fan\_out))\* sqrt(3)

### 5.Xavier均匀初始化

xavier\_normal\_(tensor, gain=1)

均值为0，标准差为gain \* sqrt(2/(fan\_in + fan\_out))的正态分布

### 6.kaiming均匀分布

kaiming\_uniform\_(tensor, a=0, mode='fan\_in', nonlinearity='leaky\_relu')



### 7.单位矩阵初始化

eye\_(tensor)

单位矩阵来填充2维输入张量或变量。在线性层尽可能多的保存输入特性

### 8.正交初始化

orthogonal\_(tensor, gain=1)

（半）正交矩阵填充

### 9.稀疏初始化

sparse\_(tensor, sparsity, std=0.01)

sparsity每列设置为零的比例，非零元素填充均值为0，标准差为std

### 10.狄拉克&函数初始化

dirac\_(tensor)

## 损失函数

均通过torch.nn实现

### 自定义函数

criterion = LossCriterion() #构造函数有自己的参数

loss = criterion(output, target) #调用标准时也有参数

### L1loss范数损失

L1Loss(size\_average=None, reduce=None, reduction='mean')

Output与target之间绝对值

### 均方差损失

MSELoss(size\_average=None, reduce=None, reduction='mean')

两者之间方差

### 交叉熵损失

CrossEntropyLoss(weight=None, size\_average=None, ignore\_index=-100, reduce=None, reduction='mean')

### CTC损失

CTCLoss(blank=0, reduction='mean')

### 负对数似然损失NLLLoss

NLLLoss(weight=None, size\_average=None, ignore\_index=-100, reduce=None, reduction='mean')

训练 C 个类别的分类问题.

### NLLLOSS2d

每个像素的负对数似然损失

### 泊松分布负数似然损失

PoissonNLLLoss(log\_input=True, full=False, size\_average=None, eps=1e-08, reduce=None, reduction='mean')

### KL散度损失

计算 input 和 target 之间的 KL 散度。KL 散度可用于衡量不同的连续分布之间的距离, 在连续的输出分布的空间上(离散采样)上进行直接回归时 很有效

KLDivLoss(size\_average=None, reduce=None, reduction='mean')

### 二进制交叉熵损失

二分类任务时的交叉熵计算函数。用于测量重构的误差, 例如自动编码机. 注意目标的值 t[i] 的范围为0到1之间.

BCELoss(weight=None, size\_average=None, reduce=None, reduction='mean')

### 二进制损失BCEWithLogitsLoss

BCEWithLogitsLoss(weight=None, size\_average=None, reduce=None, reduction='mean', pos\_weight=None)

BCEWithLogitsLoss损失函数把 Sigmoid 层集成到了 BCELoss 类中. 该版比用一个简单的 Sigmoid 层和 BCELoss 在数值上更稳定, 因为把这两个操作合并为一个层之后, 可以利用 log-sum-exp 的 技巧来实现数值稳定.

### MarginRankingLoss

MarginRankingLoss(margin=0.0, size\_average=None, reduce=None, reduction='mean')

### HingeEmbeddingLoss

HingeEmbeddingLoss(margin=1.0, size\_average=None, reduce=None, reduction='mean')

### 多标签分类损失MultiLabelMarginLoss

MultiLabelMarginLoss(size\_average=None, reduce=None, reduction='mean')

### SmoothL1Loss

SmoothL1Loss(size\_average=None, reduce=None, reduction='mean')

### 二分类损失SoftMarginLoss

SoftMarginLoss(size\_average=None, reduce=None, reduction='mean')

### 多标签 one-versus-all 损失 MultiLabelSoftMarginLoss

MultiLabelSoftMarginLoss(weight=None, size\_average=None, reduce=None, reduction='mean')

### cosine 损失 CosineEmbeddingLoss

CosineEmbeddingLoss(margin=0.0, size\_average=None, reduce=None, reduction='mean')

### 多类别分类的hinge损失 MultiMarginLoss

MultiMarginLoss(p=1, margin=1.0, weight=None, size\_average=None, reduce=None, reduction='mean')

### 三元组损失 TripletMarginLoss

TripletMarginLoss(margin=1.0, p=2.0, eps=1e-06, swap=False, size\_average=None, reduce=None, reduction='mean')

明日工作：**混淆矩阵**

## 学习率衰减六大策略(pytorch)

以下学习率lr=0.1，学习率调整倍数为 [gamma](https://www.baidu.com/s?wd=gamma&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd) 倍，均调用torch.optim. lr\_scheduler库函数

### 1. 等间隔调整学习率 StepLR

调用函数： StepLR(optimizer, step\_size, gamma=0.1, last\_epoch=-

方法：调整间隔为 step\_size，list形式。间隔单位是epoch，若step\_size=30，则每30个epoch下降学习率\*0.1

### **2.按需调整学习率**

调用函数：MultiStepLR(optimizer, milestones, gamma=0.1, last\_epoch=-1)

milestones(list)- 一个 list，每一个元素代表何时调整学习率。若meilestones=[10,30,50]，则在第10,30,50个epoch时，学习率\*0.1，0.01,0.001,0.0001.

### 3. 指数衰减调整学习率

调用函数：ExponentialLR(optimizer, gamma, last\_epoch=-1)

方法：lr=lr∗gamma∗∗epoch ，逐epoch下降

### 4.余弦退火调整学习率

调用函数： CosineAnnealingLR(optimizer, T\_max, eta\_min=0, last\_epoch=-1)

方法：以余弦函数为周期，并在每个周期最大值时重新设置学习率。以初始学习率为最大学习率，以 2∗Tmax 为周期，在一个周期内先下降，后上升。eta\_min为周期内最小学习率

### 5.自适应调整学习率

调用函数： ReduceLROnPlateau(optimizer, mode='min', factor=0.1, patience=10, verbose=False, threshold=0.0001, threshold\_mode='rel', cooldown=0, min\_lr=0, eps=1e-08)

方法：在patience个epoch内mode模式下，loss不下降或者正确率不上升，开始降低学习率，eps衰减最小值

### 6.自定义调解学习率

调用函数：LambdaLR(optimizer, lr\_lambda, last\_epoch=-1)

lr=base\_lr∗lmbda(self.last\_epoch)

调整方法：lr\_lambda(function or list)- 一个计算学习率调整倍数的函数，输入通常为 step，当有多个参数组时，设为 list。

## 模型保存与加载

### 模型保存

torch.save(net1, 'net.pkl') # 保存整个神经网络的结构和模型参数

torch.save(net1.state\_dict(), 'net\_params.pkl') # 只保存神经网络的模型参数

### 模型加载

# 仅保存和加载模型参数

torch.save(model\_object.state\_dict(), 'params.pth')

model\_object.load\_state\_dict(torch.load('params.pth'))

模型太深，第一种方法耗费时间过长

## 预训练模型

### 局部微调

有时候加载训练模型后，只想调节最后的几层，其他层不训练。 其实**不训练也就意味着不进行梯度计算**，PyTorch 中提供稍微**requires\_grad** 使得对训练的控制变得非常简单.

在 PyTorch 中，每个 **Variable数据** 含有两个flag（requires\_grad 和 volatile）用于指示是否计算此Variable的梯度. 设置 requires\_grad = False，或者设置 volatile=True，即可指示不计算此Variable的梯度.

model = torchvision.models.resnet18(pretrained=True)

for param in model.parameters():

param.requires\_grad = False # 提取 fc 层固定的参数

fc\_features = model.fc.in\_features # 替换最后的全连接层， 改为训练100类 # 新构造的模块的参数默认requires\_grad为True

model.fc = nn.Linear(fc\_features, 100)

### 修改模型内部网络层

局部微调网络的输出层，仅适用于简单的修改，如果需要对网络的内部结构进行改动，则需要采用参数覆盖的方法 - 即，先定义类似网络结构，再提取预训练模型的权重参数，覆盖到自定义网络结构中。

### 全局微调

有时候需要对全局都进行 finetune，只不过**希望改换过的层和其他的学习速率不一样**，这时候可以把其他层和新层在 optimizer 中单独赋予不同的学习速率.

## GPU使用方法

### 使用指定gpu

import os

os.environ["CUDA\_VISIBLE\_DEVICES"] = "1，2"

net.cuda()

net=nn.DataParallel(net)

## 打印模型

summary(model, (3, 572, 572))

## 网络构造

模型主要考虑精度、速度和内存消耗这三个性能指标。

1. 模型结构复杂：对于CNN而言，有一些常用的方法是增加通道数以及深度来增加精度，但是会牺牲仿真运行速度和内存. 然而，需要注意的是，层数增加对精度的提升的效果是递减的，即添加的层越多，后续添加的层对精度的提升效果越小，甚至会出现过拟合现象. 层数量或尺寸的增加，可以使得分类器包含更多不必要的参数. 如，将 数据的噪声数据也进行了加权. 其会导致过拟合问题和精度降低. 而且，会导致分类器花费更多的训练和预测时间.
2. 激活函数：对于神经网络模型而言，激活函数是必不可少的. 传统的激活函数，比如 Softmax、Tanh 等函数已不适用于 CNN 模型，有相关的研究者提出了一些新的激活函数，比如 Hinton 提出的 ReLU 激活函数，使用 ReLU 激活函数通常会得到一些好的结果，而不需要像使用 ELU、PReLU 或 LeakyReLU 函数那样进行繁琐的参数调整. 一旦确定使用ReLU能够获得比较好的结果，那么可以优化网络的其它部分并调整参数以期待更好的精度.
3. 卷积核大小：普遍认为使用较大的卷积核（比如5x5、7x7）总是会产生最高的精度，然而，并不总是这样. 研究人员发现，使用较大的卷积核使得网络难以分离，最好的使用像3x3这样更小的内核，ResNet和VGGNet已经很好地证明了这一点. 此外，也可以使用1x1这样的卷积核来减少特征图（Feature map）的数量.

4.优化器选择：当对网络训练过程优化时，有几种优化算法可供选择. 常用的算法是随机梯度下降算法（SGD），但该算法需要调整学习率等参数，这一过程略显乏味；另外使用自适应学习率梯度下降算法，比如Adam、Adagrad或Adadelta算法，是比较容易实现的，但是可能无法获得最佳的梯度下降算法精度.最好的办法是遵循和激活函数类似的处理方式，先用简单的训练方法来看看设计的模型是否工作得很好，然后用更复杂的方式进行调整和优化. 个人推荐从Adam开始，该方法使用起来非常容易：只需要设定一个不太高的学习率，通常默认设置为0.01，这样一般会得到非常好的效果，但是论文中一般使用SGD算法进行微调.

5. 为了避免引入偏差，在训练分类器时，需要将图像的顺序进行随机化.

6. 学习率调整：如果学习率过大，可能不能得到损失函数最小值，难以收敛.如果学习率过小，分类器的训练会非常慢.一般从0.1，0.01,0.001作为初始学习率，然后搭配学习率优化策略

7.参数初始化：它能够更快得到最优梯度下降方向，减少计算量。自己搭建模型是运用权重初始化方法，另一种是运用预训练模型权重