# Q&A

## Q1：训练时有哪些可以调节的超参数？

①BATCH\_SIZE;

②Learning\_Rate;

③EPOCH\_SIZE;

④优化器；

## Q2：Pytorch中有哪些优化器，可用于调节的参数有哪些？

每次反向传播都会给模型各个可学习参数params计算出一个偏导数，用于更新对应的参数。通常偏导数不会直接作用到对应的可学习参数上，而是通过优化器做一下处理，得到一个新的值，处理过程用函数F表示（不同的优化器对应的*F*的内容不同），即，然后和学习率*lr*一起用于更新可学习参数，即。

### (1)SGD算法

torch.optim.SGD(params,

lr=<required parameter>,

momentum=0,

dampening=0,

weight\_decay=0,

nesterov=False,

maximize=False,

foreach=None,

differentiable=False

)

### 参数解释

params：模型里需要被更新的可学习的参数。

lr：学习率。

momentum：动量，通过引入一个新的变量*b*去积累之前的梯度，加速学习过程。

上一次得到的和当前得到的偏导数做(1)式的运算修改偏导数。

 (1)

然后通过(2)式更新参数：

 (2)

dempening：乘到偏导数上的一个数，即(1)式中的。该参数在优化器第一次更新时不起作用。

weight decay：用当前可学习参数*p*的值修改偏导数，即：

 (3)

再通过(1)式计算得到

### (2)RMSprop算法

假设当前的损失函数为：

 (4)

通过求导可得*x*和*y*方向的导数：

 (5)

 (6)

显然，当*x*=0，*y*=0时(4)式最小。假设当前初始值*x*=40，*y*=20，则，，可见在*x*方向的移动距离比*y*方向的移动距离小，但是实际*x*离最小点更远。

**RMSprop算法有效解决了这个问题。通过累计各个变量的梯度的平方*v*，然后用每个变量的梯度除以*v*，即可有效缓解变量间的梯度差异。**

torch.optim.RMSprop(params,

lr=0.01,

alpha=0.99,

eps=1e-08,

weight\_decay=0,

momentum=0,

centered=False,

foreach=None,

maximize=False,

differentiable=False

)

### 参数解释

params：模型里需要被更新的可学习的参数。

lr：学习率。

alpha：平滑常数。将梯度的平方进行平滑处理：



eps：加在分母上防止除零

weight\_decay：用当前可学习参数*p*的值修改偏导数。参看式(3)。

momentum：动量。

centered：

当为False时，分母为；

当为True时，分母的计算过程变为：

 (7)

 (8)

### (3)Adagrad算法

在一般的优化算法中，目标函数自变量的每一个元素在相同迭代次数中都使用同一个学习率来自我迭代。当每个元素的梯度值有较大差别时，需要选择足够小的学习率使得自变量在梯度值较大的维度上不发散，但这样会导致自变量在梯度值较小的维度上迭代过慢。

Adagrad算法可以自适应的给所有的参数分配学习率，根据自变量在每个维度的梯度值的大小来调整各个维度上的学习率，从而避免统一的学习率难以适应所有维度的问题。

torch.optim.Adagrad(params,

lr=0.01,

lr\_decay=0,

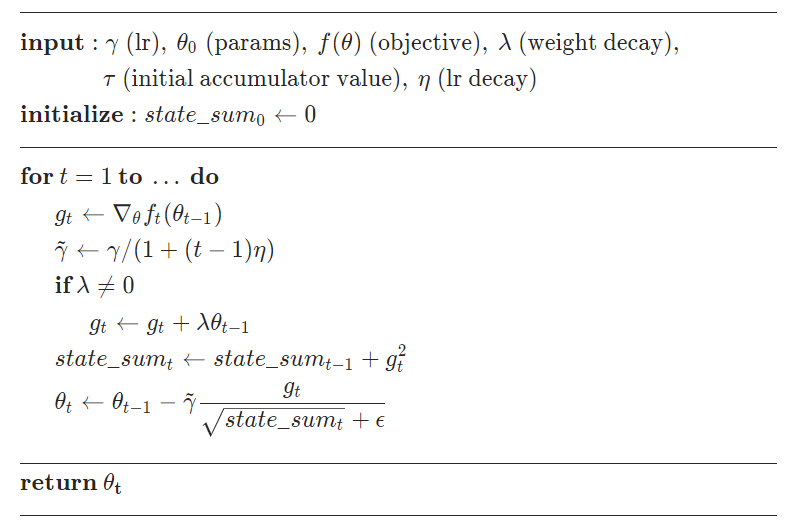
weight\_decay=0,

initial\_accumulator\_value=0,

eps=1e-10,

foreach=None,

maximize=False

)

### (4)Adam算法

在RMSProp的基础上，做两个改进：梯度滑动平均和偏差纠正。

torch.optim.Adam(params,

lr=0.001,

betas=(0.9, 0.999),

eps=1e-08,

weight\_decay=0,

amsgrad=False,

foreach=None,

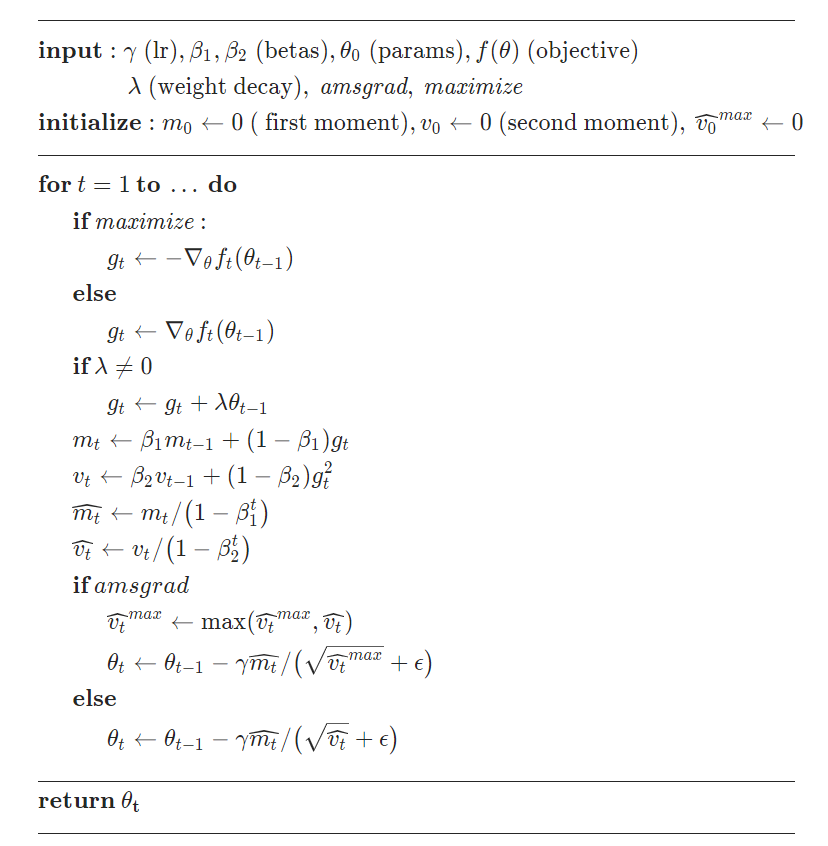
maximize=False,

capturable=False,

differentiable=False,

fused=False

)



先计算所有可学习参数的梯度，将梯度进行累计得到：

 (9)

然后累计梯度的平方：

 (10)

计算偏差纠正*m*和*v*：

 (11)

 (12)

最后通过更新参数

## Q3：预训练模型的作用？

加速模型的收敛速度；使模型的效果更好。

## Q4：训练时，影响显存占用的因素有哪些？可以从哪些方面解决？

GPU的内存占用率主要是模型的大小，包括网络的宽度，深度，参数量，中间每一层的缓存，都会在内存中开辟空间来进行保存，所以模型本身会占用很大一部分内存。

其次是batch\_size的大小。模型的加载很难控制，一般调节batch\_size来控制显存的占用。

## Q5：有哪几种权重初始化的方式？



参数传递过程示意图

其中，为激活函数，为神经元的输入。

损失函数关于第一个神经元的梯度为：

 (13)

通过式(13)可以看出梯度和权重矩阵和激活函数相关，为了防止出现梯度消失和梯度爆炸现象，要对权重的方差加以控制。

有Xavier初始化（适合适用于tanh和sigmoid激活函数）、He初始化（适用于ReLU激活函数）、RandomNormal和RandomUniform初始化

## Q6：数据读取Dataset的操作流程是怎样的？

P.S.以num\_woker=0时为例

DataLoader进入迭代器，启用\_SigleProcessDataLoaderIter一次迭代一个batch\_size大小的数据。其过程是：

通过BathSampler()的sampler随机产生索引值然后再迭代，当迭代够一个batch\_size大小后返回一次索引值，然后DataSet的\_getitem\_获取返回的索引值，得到图片数据和类别标签(data, label)，再通过\_MapDataset Fetcher得到batch\_size哥图像信息（其中每个信息为tuple类型，batch\_size个tuple组合为一个list类型的数据），一个batch的数据通过collate\_fn进行数据打包为Tensor类型(batch, channel, height, width)

## Q7：使用Pytorch搭建时卷积层、池化层、全连接层等算子有什么参数可以调节？

①卷积层

kernel\_size（卷积核大小）、stride（卷积步长）、padding（填充）、dilation（膨胀系数）、group（将原始输入通道划分成的组数）

②池化层

kernel\_size（卷积核大小）、stride（卷积步长）、padding（填充）、dilation（膨胀系数）

③全连接层

in\_feature（输入样本的大小）、out\_feature（输出样本的大小）、bias（如果设置为False，则图层不会学习附加偏差）

## Q8：模型在训练和测试时有哪些差异？

### BatchNormal在训练和测试时有不同的计算流程：

|  |  |  |
| --- | --- | --- |
| training | track\_running\_stats | 状态 |
| True | Ture | running\_mean 和 running\_var 会跟踪整个训练过程中 batch 的统计特性，而每组输入数据用当前 batch 的 mean 和 var 统计特性做归一化，然后再更新 running\_mean 和 running\_var。 |
| Ture | False | 每组输入数据会根据当前 batch 的统计特性做归一化，但不会有 running\_mean 和 running\_var 参数了 |
| False | Ture | 使用 running\_mean 和 running\_var 做归一化，并且不会对其进行更新 |
| False | False | 效果同第二点，只不过处于推理状态，不会学习 weight 和 bias 两个参数。一般不采用该状态 |

训练时

running\_mean = (1 - momentum) \* mean\_old + momentum \* mean\_new

running\_var = (1 - momentum) \* var\_old + momentum \* var\_new

测试时

running\_mean = mean\_old

running\_var = var\_old

### Dropout在训练和测试时有不同：

在model.train()时随机取一部分网络连接来训练更新参数。

在model.eval()时利用到了所有网络连接，不进行随机舍弃神经元。

## Q9：如果一个分类模型只能进行二分类，怎么修改让它进行十分类？

将最后一层全连接层nn.Linear的out\_features改为所需的类别数

## Q10：在Pytorch中，哪些常用的操作、方法、成员？

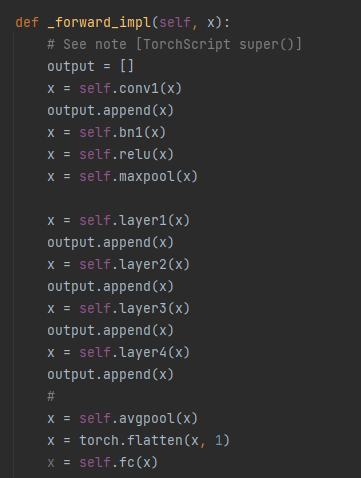
resize、permute（维度调换）、cat（维度拼接）、unsqueeze（维度扩充）

# 疑问

（1）if rank in[-1, 0]这个判断起到什么作用呢？



（2）resnet中



这个\_forward\_impl什么时候会调用这一块？

（3）TripletLoss这一块看不懂，在后面的学习过程中会涉及到嘛？如果会的话可以先不用帮我解释。

# 记录：

## 参数解析

①先将所有节点初始化，通过merge\_from\_file()来update



如果不将所有的参数都初始化一遍的话会报以下形式的错误：



②

通过opt = yaml.load(file.read, Loader=yaml.FullLoader)来读取yaml文件，访问的方式可以通过字典查询的方式，e.g.:opt[‘一级’][‘二级’]

③

通过argparse.Namespace(\*\*)来读取

④

更适合参数少，且每次改动不多的情况

Argparse.ArgumentParse()

.add\_argument()

.parse\_args()

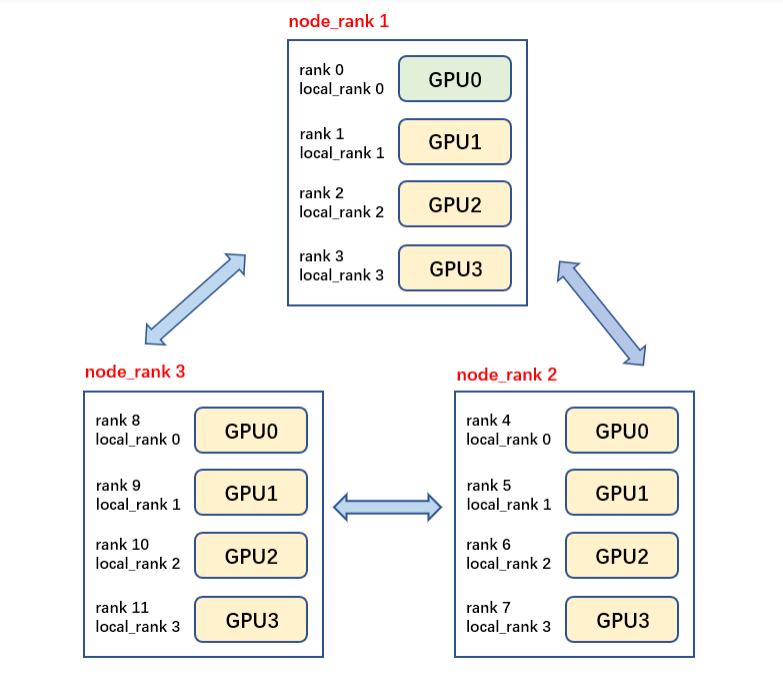
## 分布式训练

RANK：进程号，在多进程上下文中，我们通常假定rank 0是第一个进程或者主进程，其它进程分别具有1，2，3不同rank号，这样总共具有4个进程

NODE：物理节点，可以是一个容器也可以是一台机器，节点内部可以有多个GPU；nnodes指物理节点数量，nproc\_per\_node指每个物理节点上面进程的数量

LOCAL\_RANK：指在一个node上进程的相对序号，local\_rank在node之间相互独立

WORLD\_SIZE：全局进程总个数，即在一个分布式任务中rank的数量



## 学习率设置

lr\_scheduler.LambdaLR(optimizer, lr\_lambda)

更新学习率的方式是：lr = lr \* lr\_lambda

不同的层使用不同的学习率调整方法，分别分为权重层weight，偏执层bais，和BN层，单独调整不同层的学习率可以使得模型训练的更好

torch.optim.lr\_scheduler包用于动态修改lr，使用的时候需要显式地调用optimizer.step()和scheduler.step()来更新lr，学习率的更新应该放在优化器更新之后

model = [Parameter(torch.randn(2, 2, requires\_grad=True))]

optimizer = SGD(model, 0.1)

scheduler = ExponentialLR(optimizer, gamma=0.9)

for epoch in range(20):

for input, target in dataset:

optimizer.zero\_grad()

output = model(input)

loss = loss\_fn(output, target)

loss.backward()

optimizer.step()

scheduler.step()