learning rate scheduler

torch.optim.lr\_scheduler.StepLR（等间隔调整学习率）

torch.optim.lr\_scheduler.StepLR(optimizer,step\_size,gamma=0.1,last\_epoch=-1,verbose=False)

optimizer:模型优化器

step\_size:学习率更新步长(注意这里是每step\_size个epoch更新一次)

gamma:衰减率(默认0.1 即每step\_size个epoch后lr变为lr\*gamma

last\_epoch:last\_epoch之后恢复lr为initial\_lr(如果是训练了很多个epoch后中断了继续训练,这个值就等于加载的模型当前的epoch。默认为-1表示从头开始训练，即从epoch=1开始。

Verbose:是否每次改变都输出一次lr的值(单词的意思就是“冗长的” 默认是False)

其公式如下：

torch.optim.lr\_scheduler.MultiStepLR（按需调整学习率）

torch.optim.lr\_scheduler.MultiStepLR(optimizer,milestones,gamma=0.1,last\_epoch=-1,verbose=False)

milestones 数据类型是递增的list 按设定的间隔调整学习率，其余参数与StepLR 相同。

两者不同点在于StepLR 是每隔step\_size个epoch更新一次lr，而MultiStepLR是自定义的每隔多少个epoch更新一次lr。自定义由milestones的列表决定。

这个方法适合后期调试使用观察loss曲线，为每个实验定制学习率调整时机。其公式如下：

torch.optim.lr\_scheduler.ExponentialLR（指数衰减调整学习率）

torch.optim.lr\_scheduler.ExponentialLR(optimizer,gamma,last\_epoch=-1,verbose=False)

该方法为以上两个方法的简化版。其公式如下：

torch.optim.lr\_scheduler.CosineAnnealingLR（余弦退火调整学习率）

torch.optim.lr\_scheduler.CosineAnnealingLR(optimizer,T\_max,eta\_min=0,last\_epoch=-1,verbose=False)

T\_max代表1/2个cos周期所对应的epoch值 学习率下降到最小值时的epoch数 即当epoch=T\_max时 学习率下降到余弦函数最小值，当epoch>T\_max时，学习率将增大。

eta\_min最小学习率(默认0)

其公式如下：

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，以为周期，在一个周期内先下降，后上升。

torch.optim.lr\_scheduler.ReduceLROnPlateau（自适应调整学习率）

torch.optim.lr\_scheduler.ReduceLROnPlateau(optimizer, mode='min', factor=0.1, patience=10, threshold=0.0001, threshold\_mode='rel', cooldown=0, min\_lr=0, eps=1e-08, verbose=False)

mode(str)模式选择，有min和max两种模式。min表示当指标不再降低(如监测loss)； max表示当指标不再升高(如监测accuracy)。

factor(float)学习率调整倍数(等同于其它方法的gamma) 即学习率更新为 lr = lr \* factor

patience(int)忍受该指标多少个step不变化，当忍无可忍时，调整学习率

threshold(float)配合threshold\_mode使用 默认值1e-4 作用是用来控制当前指标与best指标的差异。

threshold\_mode(str)选择判断指标是否达最优的模式 有两种模式 rel和abs。

当threshold\_mode = rel并且mode = max时dynamic\_threshold = best \* ( 1 + threshold )

当threshold\_mode = rel并且mode = min时dynamic\_threshold = best \* ( 1 - threshold )

当threshold\_mode = abs并且mode = max时dynamic\_threshold = best + threshold

当threshold\_mode = abs并且mode = min时，dynamic\_threshold = best - threshold

cooldown(int)“冷却时间“当调整学习率之后，让学习率调整策略冷静一下，让模型再训练一段时间再重启监测模式。

min\_lr(float or list)学习率下限。可为float或者list 当有多个参数组时，可用list进行设置。

eps(float) 学习率衰减的最小值，当学习率变化小于eps时，则不调整学习率。

当某指标不再变化（下降或升高）时，调整学习率。这是非常实用的学习率调整策略。

torch.optim.lr\_scheduler.LambdaLR（自定义调整学习率）

torch.optim.lr\_scheduler.LambdaLR(optimizer, lr\_lambda, last\_epoch=-1)

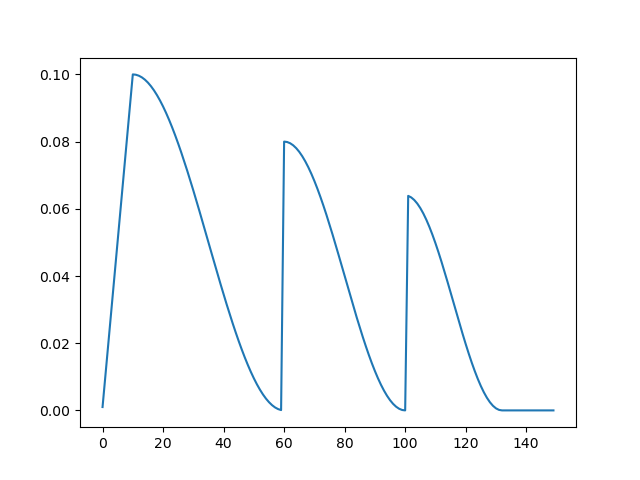
λ：通过参数lr\_lambda和epoch得到的（python中的lambda函数）或者是一个list的这样的function（lambda函数）。分别计算各个parameter groups的学习率更新用到的λ。

该方法在fine-tune 中十分有用，我们不仅可为不同的层设定不同的学习率，还可以为其设定不同的学习率调整策略。

关于optimizer.step()和scheduler.step()，optimizer.step()是在每一个batch中进行，而scheduler.step()是在每一个epoch中进行。

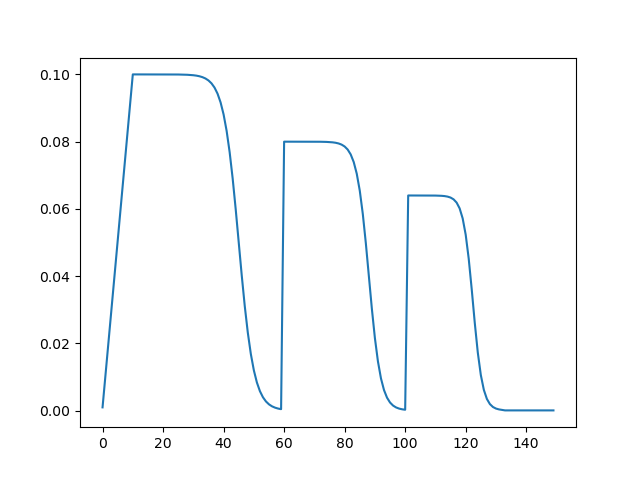
timm.scheduler. CosineLRScheduler

CosineLRScheduler(optimizer, t\_initial, lr\_min=0.0, cycle\_mul=1.0, cycle\_decay= 1.0, cycle\_limit=1, warmup\_t=0, warmup\_lr\_init=0, warmup\_prefix=False, t\_in\_epochs=True, noise\_range\_t=None, noise\_pct=0.67, noise\_std=1.0, noise\_seed=42, k\_decay=1.0, initialize=True)。该函数实现了Warm up、余弦衰减、重启功能。其中常用参数t\_initial：指初始迭代epoch数、lr\_min：最小学习率数值、cycle\_mul：重启后周期变为原来的cycle\_mul倍、cycle\_decay：重启后学习率变为原来的cycle\_decay倍、cycle\_limit：最大重启次数、warmup\_t：Warm up周期数、warmup\_lr\_init：Warm up初始学习率、warmup\_prefix：是否修正因为Warm up而减少的epoch数。



timm.scheduler. TanhLRScheduler

TanhLRScheduler(optimizer, t\_initial, lb=-7.0, ub =3.0, lr\_min =0.0, cycle\_mul =1.0, cycle\_decay =1.0, cycle\_limit=1, warmup\_t=0, warmup\_lr\_init=0, warmup\_prefix=False, t\_in\_epochs=True, noise\_range\_t=None, noise\_pct=0.67, noise\_std=1.0, noise\_seed=42, initialize=True)。该函数实现了Warm up、正切衰减、重启功能。其中参数lb, ub表示区间的上下界。



optimization

torch.optim.SGD（stochastic gradient descent）

torch.optim.SGD (params, lr, momentum=0, dampening=0, weight\_decay=0, nesterov=False, \*, maximize=False, foreach=None, differentiable=False)。该函数实现了随机梯度下降法，常用的参数有momentum：动量，使用以前时刻的梯度与当前时刻的梯度共同更新参数、dampening：抑制动量，用于调整momentum以前时刻与当前时刻的比例。weight\_decay：L2正则化，用于抑制网络过拟合、nesterov：牛顿法，使其先按原参数更新，达到新的位置后计算新的位置的梯度，然后再进行参数更新，类似用了梯度二阶项（但是对收敛率的作用却不是很大）、maximize：最大化，指优化目标为最大，默认为最小化。实现步骤如下：

|  |  |
| --- | --- |
| 1. 求梯度 |  |
| 2. 权重衰减 |  |
| 如果是L1正则化 |  |
| 不使用正则化 |  |
| 3. 动量计算及抑制 |  |
| 不使用动量 |  |
| 4. 使用nesterov法 |  |
| 不使用nesterov法 |  |
| 5. 更新参数 |  |

torch.optim.Adagrad（Adaptive Gradient Algorithm）

torch.optim.Adagrad(params, lr=0.01, lr\_decay=0, weight\_decay=0, initial\_accumulator\_value=0, eps=1e-10, foreach=None, \*, maximize=False)。该函数实现了Adagrad算法，是在SGD算法的基础上改进而来，其学习率会由根据历史梯度来进行调整，即全局学习率逐参数的除以历史梯度平方和的平方根，使得每个参数的学习率不同。在参数空间更为平缓的方向，会取得更大的进步（因为平缓，所以历史梯度平方和较小，对应学习下降的幅度较小），并且能够使得陡峭的方向变得平缓，从而加快训练速度。同时其使得其适合稀疏数据或者分布不平衡的数据集，但随着迭代次数增多，学习率会越来越小，最终会趋近于0。实现步骤如下：

|  |  |
| --- | --- |
| 1. 求梯度 |  |
| 2. 学习率衰减 |  |
| 3. 权重衰减 |  |
| 不使用正则化 |  |
| 4. 计算历史梯度 |  |
| 5. 计算结果 |  |

torch.optim.Rprop(Resilient Back Propagation)

torch.optim.Rprop(params, lr=0.01, etas=(0.5, 1.2), step\_sizes=(1e-06, 50), foreach=None, maximize=False)。该函数实现了Rprop算法。该算法首先为各权重变化赋一个初始值，设定权重变化加速因子与减速因子。在网络迭代中当连续误差梯度符号不变时，采用加速策略，加快训练速度；当连续误差梯度符号变化时，采用减速策略，以期稳定收敛。网络结合当前误差梯度符号与变化步长实现误差反向传播，同时，为了避免网络学习发生振荡或下溢，算法要求设定权重变化的上下限。优化方法适用于full-batch，不适用于mini-batch。实现步骤如下：

|  |  |
| --- | --- |
| 1. 求梯度 |  |
| 2. 计算更新步长 |  |
|  |  |
|  |  |
| 3. 计算结果 |  |

torch.optim.RMSprop（Root Mean Square prop）

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)。该函数实现了RMSprop算法，是在Rprop算法和Adagrad算法的基础上改进而来。其中参数centered指是否中心化。该算法使梯度震动较大的项，在下降时减小其下降速度；对于震动幅度小的项，在下降时加速其下降速度（类似于Adagrad算法）。RMSprop算法采用均方根作为分母，可缓解Adagrad学习率下降较快的问题，对于RNN有很好的效果。实现步骤如下：

|  |  |
| --- | --- |
| 1. 求梯度 |  |
| 2. 权重衰减 |  |
| 不使用正则化 |  |
| 3. 求滑动平均 |  |
| 4. 如果中心化 |  |
| 5. 如果使用动量 |  |
| 不使用动量 |  |

torch.optim.Adam (Adaptive Moment Estimation)

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)。该函数实现了Adam算法，是一种自适应学习率的算法，其中的参数amsgrad为在Adam上做了一点更改。实现步骤如下：

|  |  |
| --- | --- |
| 1. 求梯度 |  |
| 2. 权重衰减 |  |
| 不使用正则化 |  |
| 3. 求滑动平均 |  |
| 4. 使用amsgrad |  |
| 5. 不使用amsgrad |  |

迭代更新之前每次计算梯度的一阶moment和二阶moment，并计算滑动平均（EMA）值，来更新当前的参数。这种思想结合了Adagrad算法的处理稀疏型数据，又结合了RMSProp算法的可以处理非稳态的数据。用动量来累积梯度，又使得收敛速度更快同时，波动的幅度更小，并进行了偏差修正。

torch.optim.AdamW（Adam with decoupled weight decay）

torch.optim.AdamW(params, lr=0.001, betas=(0.9, 0.999), eps=1e-08, weight\_decay=0.01, amsgrad=False, maximize=False, foreach=None, capturable=False)。该函数实现了AdamW算法，该算法的权重衰减直接对参数进行，将weight decay与梯度更新解耦。而Adam算法的参数衰减参与到后面的运算，将最后计算得到的结果进行权重衰减，这会导致权重衰减时无论权重大小，其衰减是差不多的。AdamW算法的实现步骤如下：

|  |  |
| --- | --- |
| 1. 求梯度 |  |
| 2. 权重衰减 |  |
| 3. 求滑动平均 |  |
| 4. 使用amsgrad |  |
| 5. 不使用amsgrad |  |

torch.optim.Adamax

torch.optim.Adamax(params, lr=0.002, betas=(0.9, 0.999), eps=1e-08, weight\_decay=0, foreach=None, maximize=False)。该函数实现了Adamax 算法，对Adam中的计算公式替换为无穷范数，对Adam增加了一个学习率上限的概念。实现步骤如下：

|  |  |
| --- | --- |
| 1. 求梯度 |  |
| 2. 权重衰减 |  |
| 不使用正则化 |  |
| 3. 求滑动平均 |  |
| 4. 计算结果 |  |

torch.optim.RAdam (Rectified Adaptive Moment Estimation)

torch.optim.RAdam(params, lr=0.001, betas=(0.9, 0.999), eps=1e-08, weight\_decay=0, foreach=None)。该函数实现了RAdam算法，该算法构建了一个整流器项，这允许自适应动量作为一个潜在的方差的函数缓慢但稳定地得到充分表达，避免在训练的早期阶段，基于有限的训练数据进行过多的跳跃，从而会陷入较差的局部优化。实现步骤如下：

|  |  |
| --- | --- |
| 1. 求梯度 |  |
| 2. 权重衰减 |  |
| 不使用正则化 |  |
| 3. 求滑动平均 |  |
| 4. 计算结果 |  |
|  |  |

torch.optim.NAdam（Nesterov-accelerated Adaptive Moment Estimation）

torch.optim.NAdam(params, lr=0.002, betas=(0.9, 0.999), eps=1e-08, weight\_decay=0, momentum\_decay=0.004, foreach=None)。该函数实现了NAdam算法，为Nesterov加速的自适应矩估计，将Adam和NAG结合起来，Nadam对学习率有了更强的约束，同时对梯度的更新也有更直接的影响。一般而言，在想使用带动量的RMSprop，或者Adam的地方，大多可以使用NAdam取得更好的效果。实现步骤如下：

|  |  |
| --- | --- |
| 1. 求梯度 |  |
| 2. 权重衰减 |  |
| 不使用正则化 |  |
| 3. 求滑动平均 |  |
| 4. 计算结果 |  |

torch.optim.Adadelta

torch.optim.Adadelta(params, lr=1.0, rho=0.9, eps=1e-06, weight\_decay=0, foreach=None, maximize=False)。该函数实现了Adadelta算法，是在Adagrad算法的基础上改进而来，对学习率进行自适应约束，避免在训练后期，学习率过小；初期和中期，加速效果不错，进行了计算上的简化，训练速度快。但是初始梯度很大的话，会导致整个训练过程的学习率一直很小，在模型训练的后期，模型会反复地在局部最小值附近抖动，从而导致学习时间变长。实现步骤如下：

|  |  |
| --- | --- |
| 1. 求梯度 |  |
| 2. 权重衰减 |  |
| 不使用正则化 |  |
| 3. 求滑动平均 |  |
| 4. 更新系数 |  |
| 5. 计算结果 |  |

torch.optim.ASGD（Averaged Stochastic Gradient Descent）

torch.optim.ASGD(params, lr=0.01, lambd=0.0001, alpha=0.75, t0=1000000.0, weight\_decay=0, foreach=None, maximize=False)。

torch.optim.SparseAdam

torch.optim.SparseAdam(params, lr=0.001, betas=(0.9, 0.999), eps=1e-08, maximize=False)。实现了适用于稀疏张量的Adam算法，相当于Adam的稀疏张量专用版本。

一些解释

Weight Decay VS L2 Regularization

一、L2正则和Weight Decay并不等价。两者的目的都是想是使得模型权重接近于0。L2正则是在损失函数的基础上增加L2 norm。而权重衰减则是在梯度更新时直接增加一项。在标准SGD的情况下，通过对衰减系数做变换，可以将L2正则和Weight Decay视为一样。但是在Adam这种自适应学习率算法中两者并不等价。

二、使用Adam优化带L2正则的损失并不有效。如果引入L2正则项，在计算梯度的时候会加上对正则项求梯度的结果。那么如果本身比较大的一些权重对应的梯度也会比较大，由于Adam计算步骤中减去项会有除以梯度平方的累积，使得减去项偏小。按常理说，越大的权重应该惩罚越大，但是在Adam并不是这样。而权重衰减对所有的权重都是采用相同的系数进行更新，越大的权重显然惩罚越大。而AdamW优化器就是先进行Weight Decay，之后按Adam的更新方式更新。

Warm up方法

主要针对较大、较深的网络，很大的数据集。刚开始训练时，模型的权重(weights)是随机初始化的，对数据的“分布”理解为零，因此训练时模型的loss很大，并且基于有限的训练数据（相对于整体的数据集）会使得自适应学习率优化器（Adam等）的方差太大，进行过多的跳跃，从而会陷入较差的局部优化，甚至之后都无法拉回到较优点，同时大的学习率会造成模型在初始阶段对mini-batch的提前过拟合。当训练了几个epoch后，模型已经对当前的mini-batch有了一些正确的先验，较大的学习率就不那么容易会使模型学偏，所以可以适当调大学习率。这个过程就可以看做是Warm up。当模型训到一定阶段后，模型的分布就已经比较固定了，接近loss的local optimal了。如果还沿用较大的学习率，就会破坏这种稳定性， 因此需要进行学习率衰减。