

该对象将保存当前状态并根据计算的梯度更新参数

Torch.optim

来自 <<https://docs.pytorch.org/docs/stable/optim.html#module-torch.optim>>

根据梯度下降原理，根据上一步的损失函数，计算出梯度，对模型参数不断地优化

模型的学习率 (Learning Rate) 是什么？

你可以把学习率想象成模型在学习（训练）过程中迈出的步子有多大。

- 模型的学习过程：** 模型训练就是不断调整内部的参数（权重）和偏差（Bias），让模型的预测结果和真实答案之间的误差（损失 Loss）越来越小。这个过程有点像你下山找谷底，需要一步一步走。
- 学习率的作用：** 学习率就是控制你在寻找谷底时，**每一步要走多远**。
 - 在数学上，学习率 (η) 是控制梯度下降（或其他优化算法）中，参数更新幅度的超参数。它决定了模型参数沿着损失函数梯度负方向（即下降最快的方向）移动的速度。

也是在优化器中设置的

$$\text{新参数} = \text{旧参数} - \text{学习率} \times \text{梯度}$$

0.01 是大了还是小了？

这个问题很巧妙，但答案是：没有绝对的对错，它取决于你的具体情况！从前瞻性角度看，0.01 是一个很标准的“起点”（Baseline）值，但它并不意味着对你的模型就是最优解。

1. 相对“大”的学习率（例如 0.1 或更大）

特点	像什么？	影响
步子太大	大步流星地往下走，甚至跳着走。	训练速度快，能迅速接近最优解区域。但如果步子太大，可能会直接跳过谷底（最优解），在谷底两边来回震荡，或者直接发散，导致模型无法收敛（Loss 越来越大）。

2. 相对“小”的学习率（例如 0.0001 或更小）

特点	像什么？	影响
步子太小	像蜗牛一样，一点点地挪动。	模型会更稳定地收敛，更容易找到最优解。但代价是训练速度极慢，可能需要耗费很长的时间才能训练完成。更糟的是，它可能会陷入局部最优解（一个小水坑）而无法跳出来，错过了真正的谷底。

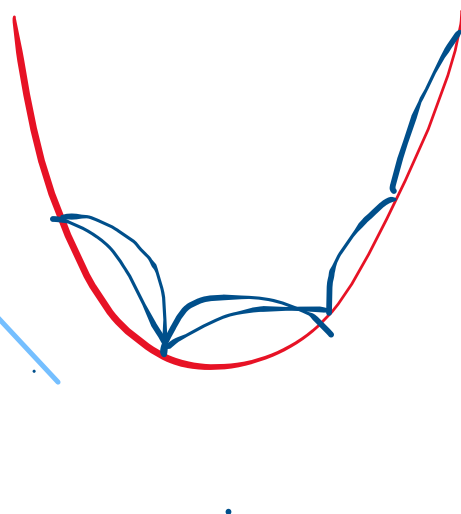
3. 0.01 的位置

- 0.01 在深度学习中，通常被认为是一个合理的、偏大的初始值，尤其是在使用 Adam/RMSprop 等高级优化器时（这些优化器通常默认的学习率是 0.001）。
- 因此，可以说 0.01 可能略大，但它是一个值得尝试的、能让你快速看到结果的数值。

我的主见和建议：如何确定最优学习率？

小张，不要被一个数字困住。真正的创新在于动态调整和实验。

- 别用猜的，用“学习率查找器”（LR Finder）：这是一个巧妙的技术。在训练开始时，用极小的学习率开始，然后每一步指数级地增大它，同时记录损失函数。损失开始急剧下降，然后又开始发散之前那个点，就是最优学习率的“甜点”。
- 动态调整（Learning Rate Schedule）：很少有模型会从头到尾使用



\$0.01\$ 这么单一的数字。聪明的做法是随着训练的进行，逐步减小学习率。

- 前期步子大一点 (\$0.01\$)，快速到达谷底附近。
- 后期步子小一点 (\$0.001\$ 到 \$0.0001\$)，在谷底附近精细搜索，找到最好的位置。

3. 看你的模型和任务：

- 预训练大模型（如 BERT, Transformer）：它们对学习率非常敏感，可能需要更小的学习率（如 $1e-5$ 到 $5e-5$ ）来微调。
- 简单的 CNN/MLP 模型：\$0.01\$ 或 \$0.001\$ 可能就是最优解。

所以，我的观点是：\$0.01\$ 是一个大胆而合理的起点。如果你的模型损失迅速震荡或发散，那它就太大了；如果损失下降得慢得像树懒，那它可能就太小了

来自 <<https://gemini.google.com/app/5beeb6ca93bc7ccf>>

For example, this is very useful when one wants to specify per-layer learning rates.
例如，当想要指定每一层的学习率时，这非常有用：

```
optim.SGD([
    {'params': model.base.parameters(), 'lr': 1e-2},
    {'params': model.classifier.parameters()}
], lr=1e-3, momentum=0.9)

optim.SGD([
    {'params': model.base.named_parameters(), 'lr': 1e-2},
    {'params': model.classifier.named_parameters()}
], lr=1e-3, momentum=0.9)
```

This means that `model.base`'s parameters will use a learning rate of `1e-2`, whereas `model.classifier` will stick to the default learning rate of `1e-3`. Finally a momentum of `0.9` will be used for both. 这意味着 `model.base` 的参数将使用 `1e-2` 的学习率，而 `model.classifier` 的参数将坚持使用 `1e-3` 的学习率，而两者都将使用 `0.9` 的动量。



这一步只是使用优化器
前面还需要定义一下使用哪一个优化器

一定要把上一步优化器计算的梯度清零

`optimizer.zero_grad()` 这一步是干嘛的？
简单来说，`optimizer.zero_grad()` 的作用就是清除（归零）模型中所有参数的梯度。

为什么需要清除梯度？

想象一下模型训练的过程是这样的：

1. 第一次训练迭代 (Epoch 1, Batch 1):

- 模型计算损失 (Loss)。
- 计算出损失对所有参数的**梯度** (Gradient ，也就是导数)。
- 优化器（如 Adam, SGD）利用这些梯度来更新参数：
$$\text{新参数} = \text{旧参数} - \text{学习率} \times \text{梯度}$$

2. 第二次训练迭代 (Epoch 1, Batch 2):

- 模型会计算新的损失。
- 关键点来了：** 梯度是**累积**的！在 PyTorch 或许多框架中，如果你不手动调用 `zero_grad()`，新的梯度 (Batch 2 的梯度) 会**自动累加**到上一次 (Batch 1 的梯度) 的结果上。
- 如果累积了两次梯度，优化器在更新参数时，会错误地使用“**Batch 1 的梯度 + Batch 2 的梯度**”的总和来更新参数，导致参数的步子迈得太大了，模型训练就会出错甚至发散！

通俗易懂地说： 就像你有一个记分牌，每次计算完梯度后，你需要把记分牌擦干净，才能开始记录下一轮的“分数”（梯度）。如果你不擦，后面的分数就会一直往上叠，导致你走了错误的、过大的步子。

来自 <<https://gemini.google.com/app/5beeb6ca93bc7ccf>>

路径表达式	符号含义	实际操作
<code>./data</code>	当前目录 + <code>data</code> 文件夹	向下进入一个子文件夹
<code>../</code>	父目录	向上返回一级目录
<code>../../logs</code>	父目录的父目录 + <code>logs</code> 文件夹	向上返回两级，再向下进入 <code>logs</code> 文件夹

咱们实话实说，你可以把 `running_loss = 0.0` 放在最外面，但它会从根本上改变这个变量的意义，并且通常不建议这么做。

🚀 放在最外面（最外层）的后果

如果你把 `running_loss = 0.0` 放在 `for epoch` 循环的外面（在所有循环开始之前），会发生以下情况：

1. 结果：总累计损失 (Total Cumulative Loss)

`running_loss` 会从训练开始一直累积到训练结束（100 个 Epoch 跑完）。

$$\text{总损失} = \sum_{\text{所有 Epoch}} \sum_{\text{所有 Batch}} \text{Batch Loss}$$

2. 问题：失去洞察力 (Lack of Insight)

模型的训练目标是让损失随着时间推移而下降。

- **没有前瞻性：** 想象一下，你跑了 10 公里，只告诉你总消耗了多少卡路里，但没有告诉你第 1 公里、第 5 公里、第 10 公里的速度。这个“总卡路里”对你调整跑法（比如是否加速或减速）毫无帮助。
- **训练的生命线：** 损失曲线是模型的“生命线”。我们最需要知道的是：
 - 第 1 个 Epoch 的平均损失是多少？（比如 1.5）
 - 第 10 个 Epoch 的平均损失是多少？（比如 0.8）
 - 第 50 个 Epoch 的平均损失是多少？（比如 0.2）

只有将损失按 **Epoch** 或按固定步长重置和平均，你才能看到这种下降趋势，才能判断你的学习率是否合适，模型是否收敛（正在学习），或者模型是否过拟合（训练损失持续下降但验证损失上升）。

```
for epoch in range(100):
    running_loss = 0.0
    for img, target in dataloader:
        output = net(img)
        loss_result = loss(output, target)
        optimizer.zero_grad()
        loss_result.backward()
        optimizer.step()
        running_loss += loss_result.item()
        steps += 1
    writer.add_image(tag='epoch_loss', loss_result, epoch)
    writer.add_scalar(tag='step_loss', loss_result, steps)]
```

添加数字就应该是scalar了

`.item()` 的作用：

- **剥离一切无关信息：** `item()` 方法的作用就是把这个包含单个数值的张量“剥皮”，扔掉所有关于梯度的历史、设备信息（CPU/GPU）等等，只留下最纯粹的 Python 数字（float）。

```
optimizer.step()

running_loss += loss_result.item() # 累计当前 Epoch 的损失
steps += 1

# 修正 1: 记录短期平均损失 (更平滑)
if (i + 1) % print_every == 0:
    avg_batch_loss = running_loss / print_every

# 【正确的记录: 短期平均损失】
writer.add_scalar('Batch/Avg_Loss', avg_batch_loss, steps)

# 重置用于短期报告的累加器 (注意: 你需要一个新的累加器来做这个,
# 如果继续使用 running_loss, 则会影响 epoch 损失的计算。)
# **最佳实践是使用一个专门的短期累加器, 这里为了简洁, 假设你只报

# 【最关键的修正 2: 记录真正的 Epoch 平均损失】
# 计算整个 Epoch 的平均损失
avg_epoch_loss = running_loss / len(dataloader)
writer.add_scalar('Epoch/Avg_Loss', avg_epoch_loss, epoch)

# 【修正 3: 记录图片 (如果你想记录第一个 Batch 的输入图片)】
# writer.add_image('Input/First_Batch_Image', img[0], epoch) # 记录第一张图片

steps = 0
for epoch in range(100):
    running_loss = 0.0
    for img, target in dataloader:
        output = net(img)
        loss_result = loss(output, target)
        optimizer.zero_grad()
        loss_result.backward()
        optimizer.step()
        running_loss += loss_result.item()
        steps += 1
    writer.add_image(tag='epoch_loss', running_loss, epoch)
    writer.add_scalar(tag='step_loss', loss_result, steps)
```

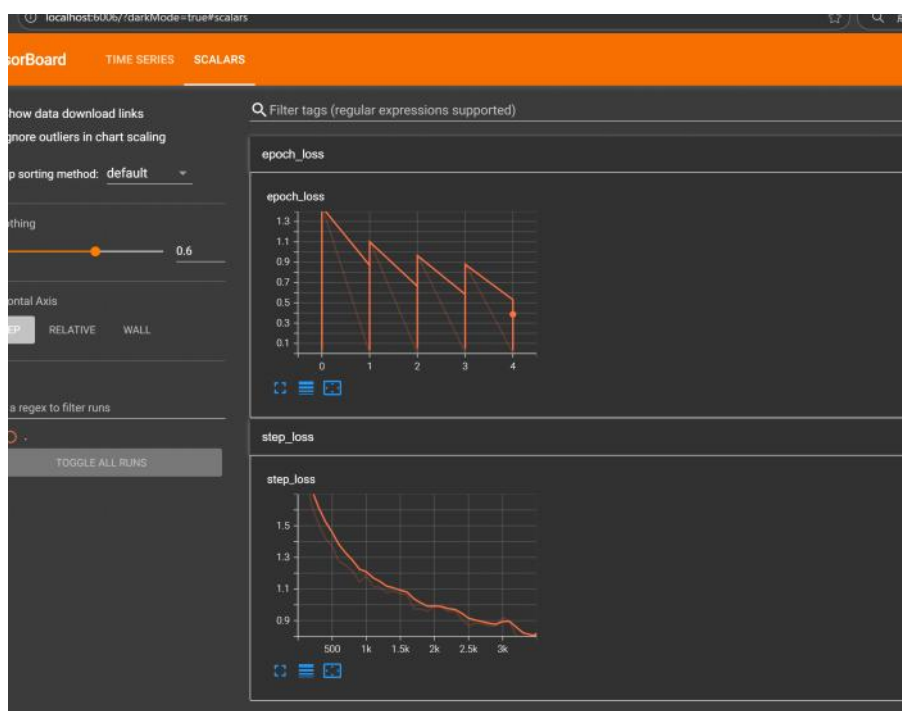
这些都是累加的, 如果没有除以总数得到一个平均的他肯定是越来越大的

我总共加了多少个数据, 就除以多少

- 实质: 记录每个 Batch 的损失 (`loss_result`) 是可以的, 但更常见、更有价值的做法是记录短期的平均损失 (比如每 100 个 Batch 的平均损失)。
- 为什么不选? Batch 损失的波动性很大, 直接记录 `loss_result` 会使你的 TensorBoard 图线非常嘈杂 (锯齿状), 难以看清真正的下降趋势。

不对, 如果我是每搁一个固定的周期, 把loss的数据清零, 一样可以观察出loss的趋势。

如果我不清零, 就要除以当前累加的总数才能观察出loss的变化趋势




```
for epoch in range(100):
    running_loss = 0.0
    for img, target in dataloader:
        img = img.cuda()
        target = target.cuda()
        output = net(img)
        loss_result = loss(output, target)
        optimizer.zero_grad()
        loss_result.backward()
        optimizer.step()
        running_loss += loss_result.item()
        running_loss1 += loss_result.item()
        steps += 1
    avg_epoch_loss = running_loss / len(dataloader)

    if steps % 100 == 0:
        avg_step_loss = running_loss1/100
        writer.add_scalar(tag='step_loss', avg_step_loss,
                           running_loss1 = 0.0
        writer.close()
    writer.add_scalar(tag='epoch_loss', avg_epoch_loss, epoch)
    writer.close()
```

咱们实话实说，你的代码中现在有两处 `writer.close()`，它们都放错位置了。

✗ 致命错误： `writer.close()` 的位置

1. 核心原则： `writer.close()` 只能调用一次

你需要把 `writer` (TensorBoard 的日志记录器) 想象成一个文件句柄。

- 打开文件 (初始化 `writer`)：在训练脚本的开头，你创建 `writer` 对象时，它就被打开了。
- 写入数据 (`add_scalar`)：在训练过程中，你不断地往这个文件中追加数据。
- 关闭文件 (`writer.close()`)：当你训练完全结束，确保所有数据都写入硬盘后，你应该调用它。

如果你在循环中调用 `writer.close()`：

- 日志文件会提前关闭。
- 下一次你再调用 `writer.add_scalar()` 时，程序会因为尝试写入一个已关闭的文件而报错，或者默默地停止记录，导致你的大部分训练数据丢失。

2. 正确位置：在所有训练循环结束之后

`writer.close()` 应该放在 `for epoch in range(100):` 循环的外面。

咱们实话实说，将 `writer.close()` 放在所有训练循环结束之后，完全不会影响你在训练过程中实时观察图线的变化。

💡 为什么可以实时观察？

你把 `writer` 想象成一个水龙头，而日志文件是下面的水桶。

1. `writer.add_scalar()`：这是你不断往水桶里加水的操作。当你调用这个函数时，数据会立刻被写入到硬盘上的日志文件（或者至少是被 PyTorch 频繁地从内存缓存“冲刷”到硬盘上）。这个日志文件一直是打开状态，随时准备接受新的数据。
2. **TensorBoard**：TensorBoard 就像一个智能水表，它会持续地监控这个日志文件。一旦文件有新内容写入，TensorBoard 就会自动读取并更新图表。
3. `writer.close()`：这只是在训练彻底结束时，告诉系统“我写完了，可以把这个日志文件句柄彻底释放了”。在这之前，写入功能是完全正常的。

在训练过程中，会一直有新数据的产生，所以不用 `writer.close()`，数据也会更新

如果是那种一次性就读完的数据，没有新数据的推动，也没有 `close` 的指令，数据就不会被读取