PyTorch 图像分类模型（timm）：实用指南

本篇内容主要为如何使用timm库构建我们自己的训练脚本

写在前面，该内容主要翻译自该博客：https://towardsdatascience.com/getting-started-with-pytorch-image-models-timm-a-practitioners-guide-4e77b4bf9055#03bd，并且会加上一些细节上的东西，也算是自己的一些理解吧，诶，研一第一个学期过去了，我经常反思真的会用PyTorch了吗？知道该如何调参吗？用的东西是不是最新的？能自己搭建出网络模型和写出训练的脚本吗？很多时候我们总是在追一些论文，但是我们有花费功夫在论文总结和代码上吗？目前，我们的很多的工作都是基于前人的基础上进行的，用的很多模型都是SOTA，但是为什么放到我们自己的数据集上效果很差？诶，也希望自己可以把这个仓库维护下去，总希望可以给后来者一些便利。

简单介绍一下：timm

PyTorch Image Models（timm）是一个包含最先进图像分类的库，其涵盖了图像模型、优化器、调度器、增强等，它被评选为2021年Paper With Code上最热门的库。

本文重点内容

主要介绍如何使用timm库，而非分析原理和timm如何实现，这里给出两个官方的文档，里面会有详细的介绍：

https://huggingface.co/docs/timm/quickstart

Pytorch Image Models (timm) | timmdocs (fast.ai)

博客中作者使用的timm==0.5.4，而我们在这里将使用timm==0.6.12

本文目录：

- **Models**
  - Customizing models
  - Feature Extraction
  - Exporting to different formats
- **Data Augmentation**
  - RandAugment
  - CutMix and Mixup
- **Datasets**
  - Loading datasets from TorchVision
  - Loading datasets from TensorFlow Datasets
  - Loading datasets from local folders
  - The ImageDataset Class
- **Optimizers**
  - Usage Optimizer Example
  - Lookahead
- **Schedulers**
  - Usage Scheduler Example
  - Adjusting learning rate schedules
- **Exponential Moving Average Model**
- **Putting it all together!**

配置环境：

这是一个老生常谈的问题，也劝退许多人的一步。下面给出我们的最佳实践：

安装Anconda：https://www.anaconda.com/，傻瓜式安装一路到底。

安装PyTorch：https://pytorch.org/，官网首页提供有安装命令，形式如下：

```
conda install pytorch torchvision torchaudio pytorch-cuda=11.6 -c pytorch -c nvidia
```

PS：无需额外安装CUDA，安装时打开Anconda Powershell Prompt，也不要新开虚拟环境，直接在base环境下操作即可。

Model

这一章节将向您展示如何查看timm包含的模型以及如何使用timm创建网络模型。

```
import timm
import torch

# 查看timm库中模型的数量
print(len(timm.list_models('*')))

# 查看timm库中包含的预训练模型数量
print(len(timm.list_models(pretrained=True)))

# 查看指定模型下的预训练模型
print(timm.list_models('resnet*', pretrained=True))
```

也可以进入GItHub，查看所有支持的模型：https://github.com/rwightman/pytorch-image-models#models

使用timm创建模型是一件非常简单的事情：

```
# 创建网络模型，pretrain=True会下载预训练权重
model = timm.create_model('resnet18d', pretrained=True)

# 打印模型信息
print(model)

# 打印模型相关设置
print(model.default_cfg)
```

PS：ResNet D模型详详情请见这篇论文：https://arxiv.org/abs/1812.01187，model.default_cfg可以方便的显示当前网络模型的一些基本设置。

timm可以训练非3通道的特征图，详情见：https://fastai.github.io/timmdocs/models#So-how-is-timm-able-to-load-these-weights?

```
# 通过参数：in_chans，设置喂入模型的特征图通道数
m = timm.create_model('resnet18d', pretrained=True, in_chans=1)

x = torch.randn(1, 1, 224, 224)
print(m(x).shape)
```

尽管我们可以很方便的使用预训练模型，但是由于输入图片与模型训练时的图片存在较大差异，我们不应该期望一步到位，而是需要我们在新的数据集上进行微调。

Customizing Models

除了使用现有架构模型外，create_model还支持许多参数，使得我们可以根据当前任务来定制我们的模型。

所支持的参数是取决于底层模型架构的，其中一些参数例如：

- global_pool：确定在最终分类层之前要使用的全局池化类型。

需要注意的是并不是所有模型都有global_pool层，例如ViT是不存在global_pool的，因此修改模型时应熟悉当前所使用模型的结构。

当然也存在着几乎所有模型都有的参数，例如：

- **drop_rate:** 设置dropout的机率，默认为0。
- **num_classes:**分类类别数。

Changing the number of classes

我们可以直接调用**fc**来查看我们使用模型的分类头。

```
# 查看模型分类头
print(model.fc)
```

但是，这个名称可能会根据所使用的模型架构而改变。为了统一接口，timm的模型提供了get_classifier方法，我们可以使用它来检索分类头，而需要查找模块名称。

```
print(model.get_classifier())
```

由于这个模型是在 ImageNet 上预训练的，我们可以看到最后一层输出了 1000 个类。我们可以用 num_classes 参数改变它：

```
timm.create_model('resnet18d', pretrained=True, num_classes=10).get_classifier()
```

如果我们想避免完全创建最后一层，我们可以将类的数量设置为 0，这将创建一个具有恒等函数的模型作为最后一层；这对于检查倒数第二层的输出很有用。

```
timm.create_model('resnet18d', pretrained=True, num_classes=0).get_classifier()
```

Global pooling options

我们使用以下代码可以很轻松的查看到模型中全局池化的操作：

```
print(model.global_pool)
```

在这里，我们可以看到它返回了 SelectAdaptivePool2d 的一个实例，这是由 timm 提供的一个自定义层，它支持不同的池化和扁平化配置。在撰写本文时，支持的池选项如下：

- avg：平均池化
- max：最大池化
- avgmax：平均池化和最大池化总和，再缩放至0.5。
- catavgmax：沿特征维数的平均池化和最大池化的串联。该操作将导致特征维度加倍。
- ''：不使用池化操作，池化层将变为恒等映射。

我们可以通过下列代码进行观察：

```
pool_types = ['avg', 'max', 'avgmax', 'catavgmax', '']
for pool in pool_types:
    model = timm.create_model('resnet18d', pretrained=True, num_classes=0,
global_pool=pool)
    model.eval()
    feature_output = model(torch.randn(1, 3, 224, 224))
    print(feature_output.shape)
```

Modifying an existing model

我们还可以修改现有模型的分类器和池化层，使用reset_classifier方法：

```
m = timm.create_model('resnet18d', pretrained=True)

print(f'Original pooling: {m.global_pool}')
print(f'Original classifier: {m.get_classifier()}')
print('----------------------------------')

m.reset_classifier(10, 'max')

print(f'Original pooling: {m.global_pool}')
print(f'Original classifier: {m.get_classifier()}')
```

Creating a new classification head

虽然已经证明，使用单一的线性层作为我们的分类器足以取得良好的结果。当微调下游任务模型时，作者经常发现，使用稍微大一点的头可以导致性能的提高。让我们来探讨如何进一步修改ResNet模型。

首先，让我们像前面那样创建ResNet模型，指定我们需要的10个类。由于我们想要使用一个更大的分类头，因此，将采取catavgmax作为我们的池化层，以便可以为分类器提供更多信息。

```
model = timm.create_model('resnet18d', pretrained=True, num_classes=10,
global_pool='catavgmax')

num_in_features = model.get_classifier().in_features
print(num_in_features)
```

现在我们对分类头进行修改，代码如下：

```
from torch import nn

model.fc = nn.Sequential(
    nn.BatchNorm1d(num_in_features),
    nn.Linear(in_features=num_in_features, out_features=256, bias=False),
    nn.ReLU(),
    nn.BatchNorm1d(256),
    nn.Dropout(0.4),
    nn.Linear(in_features=256, out_features=10, bias=False)
)
```

修改好后，我们再来验证一下我们的模型是否可以正确输出：

```
model.eval()
o = model(torch.randn(1, 3, 224, 224))
print(o.shape)
```

经过上述操作，我们就得到了一个可以训练的模型了。

Feature Extraction

timm模型也有统一的机制来获取各种类型的中间特征，这对于使用timm模型作为下游任务的特征提取器是有用的，例如在目标检测中创建特征金字塔。

让我们通过牛津宠物数据集中的图像来可视化这个过程。

这里要说说明一下，请提前准备好数据集，这里不做限定。

```
# 先获取数据集中的图片路径
import os
pets_images_path = [os.path.join('./pets/images/', f) for f in
os.listdir('./pets/images/')]
```

```
# 导入需要的包
from PIL import Image
import numpy as np

# 显示数据集中的第一个图片
image = Image.open(pets_images_path[0])
image.show()
```

```
# 将图片转换为tensor
image = torch.as_tensor(np.array(image, dtype=np.float32)).transpose(2, 0)[None]
print(image.shape)

# 创建模型并检查网络配置
model = timm.create_model('resnet18d', pretrained=True)
print(model.default_cfg)
```

如果我们只对最终的特征图感兴趣，这个输出是在池化层和分类层之前的最后一个卷积层的输出，我们可以使用`forward_features`这个方法获得。

```
import matplotlib.pyplot as plt
# 获取特征图
feature_output = model.forward_features(image)

# 显示特征图的一个函数
def visualise_feature_output(f):
    print((f[0].transpose(0, 2).sum(-1).detach()).shape)
    plt.imshow(f[0].transpose(0, 2).sum(-1).detach())
    plt.show()

visualise_feature_output(feature_output)
```

Multiple feature outputs

尽管`forward_features`方法可以便捷的获取最终的特征图，timm也提供了方法，使得我们能够使用模型作为特征骨干，输出特征图可以为选定的水平。

我们可以在创建模型时指定参数`features_only=True`，大多数的模型会返回5步长的，一般从2开始（一些从1或4开始）。

同时，可以使用`out_index`和`output_stride`参数修改特征图的级别索引和步长，详情可以参考文档：https://rwightman.github.io/pytorch-image-models/feature_extraction/#multi-scale-feature-maps-feature-pyramid

让我们来看看如何在ResNet-D模型上操作。

```
model = timm.create_model('resnet18d', pretrained=True, features_only=True)
```

如下所示，我们可以获得关于返回特征图的信心，比如特定的模块名称、特征图的减少以及通道数量：

```
print(model.feature_info.module_name())

print(model.feature_info.reduction())

print(model.feature_info.channels())
```

现在，让我们通过我们的特征提取器传递图像并探索输出。

```
out = model(image)

len(out)
```

正如预计，返回了5个特征图。通过检查形状，我们可以看到通道的数量与我们预期的一致：

```
for o in out:
    print(o.shape)
```

将每个特征映射可视化，我们可以看到图像逐渐下采样，正如我们所预期的那样。

```
for o in out:
    plt.imshow(o.[0].transpose(0, 2).sum(-1).detach().numpy())
    plt.show()
```

Using FX

TorchVision实现了一个功能名为FX，这使得在 PyTorch 模块的正向传递期间访问输入的中间转换变得更加容易。这是通过特征性地跟踪转发方法来生成一个图，其中每个节点表示一个操作。由于节点的名称是人类可读的，所以很容易确切地指定我们想要访问哪些节点。FX 在文档和这篇博客文章中有更详细的描述。

https://pytorch.org/docs/stable/fx.html#module-torch.fx

https://pytorch.org/blog/FX-feature-extraction-torchvision/

注意：在编写本文时，当使用 FX 时，动态控制流还不能用静态图表示。

由于timm中的几乎所有模型都可以特征性地追踪，因此我们可以使用FX来操作这些模型。让我们来探索一下如何使用FX从timm模型中提取特征。

```
# 先导入我们所需要的包
from torchvision.models.feature_extraction import get_graph_node_names,
create_feature_extractor
```

现在，我们重新创建一个带有分类头的ResNet-D模型，并使用参数`exportable`确保我们的模型是可以被跟踪的。

```
model = timm.create_model('resnet18d', pretrained=True, exportable=True)
```

现在，我们可以使用 `get_graph_node_names` 方法按照执行顺序返回节点名称。模型会被跟踪两次，在train和eval模式下，两组节点都会返回。

```
nodes, _ = get_graph_node_names(model)
print(nodes)
```

使用FX，可以方便的访问任何节点的输出。让我们选择layer1中的第一个激活后的特征。

```
features = {'layer1.0.act1': 'out'}
```

使用 `create_feature_extractor`，我们可以轻松的"截断"我们的网络。

```
feature_extractor = create_feature_extractor(model, return_nodes=features)
print(feature_extractor)
```

现在，让我们可视化我们的特征图。

```
out = feature_extractor(image)
plt.imshow(out['out'][0].transpose(0, 2).sum(-1).detach().numpy())
```

Exporting to different formats

经过训练后，通常建议将模型导出到优化的格式下以便进行推理；PyTorch有多种选择来实现这一点。因为几乎所有的timm模型都可以编写脚本和可跟踪的，接下来让我们一起来探索。

Exporting to TorchScript

TorchScript是一种从PyTorch代码创建序列化和可优化模型的方法；任何的TorchScript能够从Python进程中保存，并且可以在没有Python依赖关系的进程中加载。

我们可以通过两种不同的方式将模型转化为TorchScript：

- Tracing：运行代码，记录发生的操作，并构造一个包含这些操作的ScriptModule。控制流或者像if/else语句这样的动态行为会被擦除。
- Scripting：使用脚本编译器对 Python 源代码进行直接分析，将其转换为 TorchScript。这保留了动态控制流，并适用于不同大小的输入。

更多信息请见：

https://pytorch.org/docs/stable/jit.html

https://pytorch.org/tutorials/beginner/Intro_to_TorchScript_tutorial.html

由于大多数timm模型都是可以编写脚本的，因此让我们使用脚本导出ResNet-D模型。我们可以设置层配置，以便在创建模型时使用参数 `scriptable` 生成**jit scriptable**。

```
model = timm.create_model('resnet18d', pretrained=True, scriptable=True)
model.eval()
```

在导出模型之前调用 `model. eval()`，将模型置于推理模式中，这一点很重要，因为诸如dropout和批量规范之类的操作符根据模式的不同而有不同的行为。

现在，我们可以验证是否能够编写脚本并使用模型。

```
scripted_model = torch.jit.script(model)
print(scripted_model)
print(scripted_model(torch.rand(8, 3, 224, 224)).shape)
```

Exporting to ONNX

Open Neural Network eXchange(ONNX)是一种用于表示机器学习模型的开放标准格式。

我们可以使用`torch.ONNX`模块将timm模型导出到ONNX；使其可以被支持ONNX的使用。如果用不是ScriptModule的Module调用`torch.onnx.export()`，它首先执行相当于`torch.jit.trace()`的操作，后者使用给定的参数执行模型一次，并记录执行期间发生的所有操作。这意味这如果这个模型是动态的，例如，根据输入数据改变行为，导出的模型将不会捕获这种动态行为。类似的，跟踪可能只对特定的输入大小有效。

更多有关ONNX的细节可以在文档中找到：https://pytorch.org/docs/master/onnx.html

为了能够导出ONNX格式的timm模型，我们可以在创建模型时使用参数`exportable`，这将确保模型是可跟踪的。

```
model = timm.create_model('resnet50d', pretrained=True, exportable=True)
model.eval()
```

我们现在可以跟踪和导出我们的模型：

```
x = torch.randn(2, 3, 224, 224, requires_grad=True)
torch_out = model(x)

torch.onnx.export(
    model,
    x,
    "resnet18d.onnx",
    export_params=True,
    opset_version=10,
    do_constant_folding=True,
    input_names=['input'],
    output_names=['output'],
    dynamic_axes={
        'input': {0: 'batch_size'},
        'output': {0: 'batch_size'}
    }
)
```

我再这里给出可以参考的博客：

模型部署入门教程（一）：模型部署简介 - 知乎 (zhihu.com)

模型部署入门教程（二）：解决模型部署中的难题 - 知乎 (zhihu.com)

模型部署入门教程（三）：PyTorch 转 ONNX 详解 - 知乎 (zhihu.com)

现在，我们可以使用`check_model`函数验证我们的模型是否有效。

```
# 可能你需要先执行：pip install onnx
import onnx

onnx_model = onnx.load("resnet18d.onnx")
onnx.checker.check_model(onnx_model)
```

因为我们指定模型应该是可跟踪的，所以我们也可以手动执行跟踪，如下所示：

```
traced_model = torch.jit.trace(model, torch.rand(8, 3, 224, 224))
print(type(traced_model))
print(traced_model(torch.rand(8, 3, 224, 224)).shape)
```

Data Augmentation

timm包括许多数据增强转换，这些转换可以连接在一起形成数据增强pipelines; 与TorchVision类似，这些pipelines期望将PIL图像作为输入。

最简单的入门方法是使用`create_transform`工厂函数，让我们在下面探索如何使用它。

```
from timm.data.transforms_factory import create_transform

print(create_transform(224,))
print(create_transform(224, is_training=True))
```

在这里，我们可以看到，这已经创建了一些基本的数据增强pipelines，包括调整大小，标准化和转换图像到张量。正如我们所预期的那样，我们可以看到在设置`is _ training = True`时包含了额外的转换，比如水平翻转和颜色抖动。这些增强的大小可以通过参数控制，比如`hflip`、`vflip`和`color _ jitter`。

我们可以看到一些数据增强的方法取决于我们模型是否处于训练阶段，标准的`Resize`和`CenterCrop`被应用于训练和验证阶段，在训练阶段使用`RandomResizedCropAndInterpolation`将被使用。让我们看看它在下面做了什么。这种变换的实现也使我们能够设置不同的图像插值方法，这里我们选择的插值是随机选择的。

```
import matplotlib.pyplot as plt
from timm.data.transforms import RandomResizedCropAndInterpolation

image = Image.open(pets_images_path[0])
tfm = RandomResizedCropAndInterpolation(size=350, interpolation='random')
fig, ax = plt.subplots(2, 4, figsize=(10, 5))

for idx, im in enumerate([tfm(image) for i in range(4)]):
    ax[0, idx].imshow(im)

for idx, im in enumerate([tfm(image) for i in range(4)]):
    ax[1, idx].imshow(im)

fig.tight_layout()
plt.show()
```

运行多次转换，我们可以观察到不同的作物已经采取了图像。虽然这在训练中是有益的，但是在评估中这可能会使任务更加困难。根据图像类型的不同，这种类型的变换可能导致图像的主题被裁剪出图像；如果我们看第一行的第二张图像，我们可以看到这种情况的一个例子！虽然这不应该是一个巨大的问题，如果想它不经常发生，我们可以通过调整比例参数来避免这一点。

```
tfm = RandomResizedCropAndInterpolation(size=224, scale=(0.8, 1))
tfm(image)
```

RandAugment

当开始一个新任务时，可能很难知道要使用哪些数据增强方法，以及按照什么顺序使用；随着现在可用数据增强方法的增加，可以组合的数量是巨大的。

通常，一个好的起点是使用在其他任务上表现出良好性能的数据增强pipelines。其中一个策略是RandAugment，它是一种自动化的数据增强方法，从一组增强中统一采样操作，例如均衡、旋转、日光、色彩扰动、色调调整、对比度变化、亮度变化、剪切和平移并按照顺序应用这些方法。你可以参考这篇论文：[RandAugment: Practical automated data augmentation with a reduced search space](#) 。

然而，在timm中提供的实现有几个关键的不同之处，timm的创建者Ross Wightman在论文[ResNet strikes back: An improved training procedure in timm](#)的附录中进行了详尽的描述，我在下面对此进行了解释：

> 最初的RandAugment规范有两个超参数，M和N，其中M是失真大小，N是每幅图像均匀采样和应用的失真数量。RandAugment的目标是使M和N都可以被人类解释。
>
> 然而，最终M（在最初的实现中）并非如此。一些增强的尺度是倒退的，或者在这个范围内不是单调增加的，因此增加M并不增加所有增强的强度（在最初的实现中，虽然一些增强措施随着M的增加而增强，但其他增强措施却在减少或被完全剔除，这样，每个M基本代表了它自己的策略）。
>
> timm的实现试图通过增加一个“增加”模式来改善这种情况（默认启用）。在这个模式下，所有的增强强度都会随着幅度的增加而增加（这使得M更加直观，因为所有的增强现在应该随着M的相应减少/增加而减少/增加强度）。
>
> 此外，timm增加了一个MSTD参数，它在每个失真应用的M值中增加了具有指定标准差的高斯噪声。如果MSTD被设置为"-inf"，则每次失真时，M被从0-M中均匀的采样。
>
> timm中的RandAugment很注意减少对图像平均值的影响，归一化参数可以作为一个参数传递，这样所有可能引入边界像素的增强都可以使用指定的平均值，而不是像其他的实现那样默认为0或一个硬编码的元组。
>
> 最后，默认情况下不包括Cutout，以利于单独使用timm的随机擦除实现（详情见：https://fastai.github.io/timmdocs/RandomErase），这对平均数和标准差的影响较小。对增强的图像的平均值和标准差的影响较小。

现在我们了解了RandAugment是什么，让我们看看如何在数据增强pipelines使用。

在timm中，我们通过使用配置字符串来定义我们的RandAugment策略的参数；它由多个部分组成，以破折号(-)分隔。

第一部分定义了随机增强的具体变体（目前只支持rand）。其余部分可以按照任何顺序排列，它们是：

- **m**(*integer*)：随机数据增强的幅度。
- **n**(*integer*)：每幅图像所选择的转换操作数，这是可选的，默认为2。
- **mstd**(*float*)：应用的量级噪声的标准偏差。
- **mmax**(*integer*)：设置幅度的上限，而不是默认的10。
- **w**(*integer*)：概率权重指数（影响操作选择的一组权重指数）。
- **inc**(*bool-{0, 1}*)：使用随幅度增大而增大严重性的增量，这是可选的，默认为0。

举个例子：

- *'rand-m9-n3-mstd0.5'*：使用RandAugment的幅度为9，每张图片使用3种增强方法，mstd的值设定为0.5。
- *'rand-mstd1-w0'*：mstd的值设定为1.0，权重为0，其余默认设置m为10，每张图片应用的数据增强方法为2。

向`create_transform`传递一个配置字符串，我们可以看到这是由`RandAugment`对象处理，我们可以看到所有可用的操作名称。

```
t = create_transform(150, is_training=True, auto_augment='rand-m9-mstd0.5')
print(t)
```

我们还可以通过使用`rand_augment_transform`函数创建这个对象，以便在自定义pipelines中使用，如下所示：

```
from timm.data.auto_augment import rand_augment_transform

tfm = rand_augment_transform(
    config_str='rand-m9-mstd0.5',
    hparams={'img_mean': (124, 116, 104)}
)

print(tfm)
```

让我们将这个策略应用于一个图像，做一些可视化的展示。

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots(2, 4, figsize=(10, 5))

for idx, im in enumerate([tfm(image) for i in range(4)]):
    ax[0, idx].imshow(im)

for idx, im in enumerate([tfm(image) for i in range(4)]):
    ax[1, idx].imshow(im)

fig.tight_layout()
plt.show()
```

由此，我们可以看到，使用随机增强已经给了我们很多的变化的图像！

CutMix and Mixup

timm使用`Mixup`类为CutMix和Mixup增强提供了一个灵活的实现；它处理这两种增强方法，并提供在它们之间切换的选项。

使用Mixup，我们可以从各种不同的混合策略中进行选择：

- `batch`：CutMix与Mixup的选择，lambda和CutMix区域采样是按批次进行的。
- `pair`：在一个批次中对采样对进行混合、lambda和区域采样。
- `elem`：混合、lambda和区域采样是在批次内对每张图像进行的。
- `half`：与elementwise相同，但每个混合对中的一个被丢弃，这样每个样本在每个训练轮次中被看到一次。

让我们思考一下这是怎么回事。为此，我们需要创建一个DataLoader，对其进行迭代，并将增强功能应用到批次中。在这里，我们将使用宠物数据集里的图像。

```
from timm.data import ImageDataset
from torch.utils.data import DataLoader

def create_dataloader_iterator():
    dataset = ImageDataset('pets/images', transform=create_transform(224))
    dl = iter(DataLoader(dataset, batch_size=2))
    return dl

dataloader = create_dataloader_iterator()
inputs, classes = next(dataloader)
```

为了能看到图像，我们定义一个函数来显示张量。在这里，实现来自timmdocs。

```
import numpy as np
import matplotlib.pyplot as plt

def imshow(inp, title=None):
    """Imshow for Tensor."""
    inp = inp.cpu().numpy().transpose((1, 2, 0))
    mean = np.array([0.485, 0.456, 0.406])
    std = np.array([0.229, 0.224, 0.225])
    inp = std * inp + mean
    inp = np.clip(inp, 0, 1)
    plt.imshow(inp)
    if title is not None:
```

```
        plt.title(title)
    plt.pause(0.001)
```

使用torchvision中的辅助函数，我们可以直观的看到批处理的图像，而不需要任何增强：

```
import torchvision

out = torchvision.utils.make_grid(inputs)
imshow(out, title=[x.item() for x in classes])
```

现在，让我们来创建我们的Mixup数据增强。

`Mixup` 支持下列参数：

- `mixup_alpha` (*float*)：mixup alpha值，如果大于0，则激活mixup（默认为1）。
- `cutmix_alpha` (*float*)：cutmix alpha值，如果大于0，则激活cutmix（默认为0）。
- `cutmix_minmax` (*List[float]*)：cutmix最小/最大图像比率，如果cutmix_alpha值大于0，则该参数被激活使用。
- `prob` (*float*)：每批和每个元素应用mixup或cutmix的概率（默认为1）。
- `switch_prob` (*float*)：当两者处于激活状态时，从mixup切换到cutmix的概率（默认为0.5）。
- `mode` (*str*)：如何应用mixup/cutmix参数（默认为 `batch` ）。
- `label_smoothing` (*float*)：标签平滑量应用于混合目标张量（默认为0.1）。
- `num_classes` (*int*)：目标变量的类别数。

让我们定义一组参数，这样我们可以对一批图像应用mixup或cutmix，并能够交替使用它们，让我们来创建 `Mixup` 数据增强：

```
from timm.data.mixup import Mixup

mixup_args = {
    'mixup_alpha': 1.,
    'cutmix_alpha': 1.,
    'prob': 1,
    'switch_prob': 0.5,
    'mode': 'batch',
    'label_smoothing': 0.1,
    'num_classes': 2
}

mixup_fn = Mixup(**mixup_args)
```

由于mixup和cutmix是在一批图像上进行的，我们可以在应用增强之前将这批图像放在GPU上，以加快进度，在这里我们可以看到mixup已经被应用于这批图像。

```
mixed_inputs, mixed_classes = mixup_fn(inputs.to(torch.device('cuda:0')),
classes.to(torch.device('cuda:0')))
out = torchvision.utils.make_grid(mixed_inputs)
imshow(out, title=mixed_classes)
```

多次执行可以看到mixup和cutmix的结果。

从上面打印的标签，我们可以观察到，我们也可以在Mixup上使用标签平滑。

Datasets

timm为处理不同类型的数据集提供了许多有用的实用工具。最简单的入门方法是使用 `create_dataset` 函数，它将为我们创建一个合适的数据集。

`create_dataset`需要两个参数：

- `name`：想要加载的数据集名称。
- `root`：本地文件系统上数据集的根文件夹。

但是有额外的关键字参数，可以用于指定选项，如果是否需要加载训练集或验证集。

我们还可以使用`create_data`从几个不同的地方加载数据：

- datasets available in Datasets — Torchvision main documentation (pytorch.org)
- datasets available in TensorFlow datasets
- datasets stored in local folders

让我们来探讨其中一些选项。

```
from timm.data import create_dataset
```

Loading datasets from TorchVision

要加载TorchVision包含的数据集，我们只需要在我们希望加载的数据集的名称前指定前缀`torch/`。如果数据集不存在于系统中，我们可以通过设置`download=True`来下载这些数据集/此外这里我们还可以指定参数`split`分割数据集。

```
ds = create_dataset('torch/cifar10', 'cifar10', download=True, split='train')
```

```
ds, type(ds)
```

通过类型检查，我们可以看到这是一个TorchVision数据集。我们可以像往常一样通过一个索引来访问它：

```
print(ds[0])
```

Loading datasets from TensorFlow Datasets

我们在当前工作中从未涉足过TensorFlow，因此我们便不翻译这部分文档，详情请见原博客。

Loading data from local folders

我们还可以从本地文件夹加载数据，在这种情况下，我们只需要使用空字符串(`''`)作为数据集名称。

除了能够从ImageNet风格的文件目录中加载外，`create_dataset`还能让我们从一个或多个tar文件中获取；我们可以用它来避免解压tar文件。作为一个例子，我们可以在Imagenette数据集上试试这个方法。

此外，到目前为止我们已经加载了原始图像，所以让我们也使用`transform`参数来应用一些转换；这里可以使用我们之前看到的`create_change`函数来快速创建一些合适的转换！

```
from timm.data.transforms_factory import create_transform

ds = create_dataset(name='',
                    root='imagenette/imagenette2-320.tar',
                    transform=create_transform(224))

image, label = ds[0]

print(image.shape)
```

通过检查图像的形状，我们可以看到我们的变换已经被应用了。

The ImageDataset Class

正如我们所见到的，`create_dataset`函数为处理不同类型的数据集提供了很多选项。timm之所以能够提供这种灵活性，是因为它在可能的情况下使用了**torchvision**提供的现有数据集，并提供了一些额外的实现——`ImageDataset`和`IterableImageDataset`，它们可以在很多场景中使用。

本质上，`create_dataset`通过选择适当的类为我们简化了这个过程，但是有时候我们可能希望直接使用底层组件。

我使用最多的`ImageDataset`，它类似于`torchvision.dataset.ImageFolder`，但是有一些额外的功能。让我们来看看如何使用它来加载解压缩的图像数据集。

```
from pathlib import Path
from timm.data import ImageDataset

imagenette_ds = ImageDataset('imagenette/imagenette2-320/train')
len(imagenette_ds)
```

`ImageDataset`灵活性的关键在于它的索引和加载样本的方式被抽象为一个`Parser`对象。

timm包含几个解析器，包括从文件夹、tar文件和tensorflow数据集读取图像的解析器。解析器可以作为一个参数传递给数据集，我们可以直接访问解析器。

```
print(imagenette_ds.parser)
```

在这里，我们可以看到默认的解析器是`ParserImageFolder`的一个实例。解析器还包含有用的信息，比如类查找，我们可以访问这些信息，如下所示。

```
print(imagenette_ds.parser.class_to_idx)
```

我们可以看到这个解析器已经将原始标签转换成了整数，可以将整数反馈给我们的模型。

Selecting a parser manually - tar example

因此，除了选择一个合适的类之外，`create_dataset`还负责选择正确的解析器。 再次考虑到压缩的Imagenette数据集，我们可以通过手动选择`ParserImageInTar`解析器并覆盖`ImageDataset`的默认解析器来达到同样的效果。

```
from timm.data.parsers.parser_image_in_tar import ParserImageInTar

data_path = 'imagenette'

ds = ImageDataset(data_path, parser=ParserImageInTar(data_path))
```

检查第一个样本，我们可以验证它是否已经正确加载。

```
print(ds[0])
```

Writing a custom Parser - Pets example

不幸的是，数据集的结构并不总像ImageNet那样，也就是说，具有以下结构：

```
root/class_1/xx1.jpg
root/class_1/xx2.jpg
root/class_2/xx1.jpg
root/class_2/xx2.jpg
```

对于这些数据集，`ImageDataset`不能够开箱即用。虽然我们总是可以实现一个自定义的数据集集来处理这个问题，但这会是一个编程上的难题，取决于数据的存储方式。另一个选择是编写一个自定义解析器，与`ImageDataset`一起使用。

作为一个例子，让我们考虑一下之前下载的宠物数据集，其中所有图像都位于一个文件夹中，而图像所属类别的名称则包含在文件名中。

在这种情况下，由于我们任然从本地文件系统中加载图片，所以只是在`ParserImageFolder`上做了一些调整。让我们来看看是如何实现的，以获得灵感。

```
timm.data.parsers.parser_image_folder.ParserImageFolder
```

```
class ParserImageFolder(Parser):
        def __init__(self, root, class_map=""):
            super().__init__()

            self.root = root
            class_to_idx = None
            if class_map:
                class_to_idx = load_class_map(class_map, root)
            self.samples, self.class_to_idx = find_images_and_targets(
                root, class_to_idx=class_to_idx
            )
            if len(self.samples) == 0:
                raise RuntimeError(
                    f'Found 0 images in subfolders of {root}. Supported image
extensions are {", ".join(IMG_EXTENSIONS)}'
                )

        def __getitem__(self, index):
            path, target = self.samples[index]
            return open(path, "rb"), target

        def __len__(self):
            return len(self.samples)

        def _filename(self, index, basename=False, absolute=False):
            filename = self.samples[index][0]
            if basename:
                filename = os.path.basename(filename)
            elif not absolute:
                filename = os.path.relpath(filename, self.root)
            return filename
```

从这里，我们可以看到`ParserImageFolder`做了以下几件事情：

- 创建类的映射。
- 实现了`__len__`以返回samples大小。
- 实现`__filename__`以返回sample的文件名，并提供用于确定它是绝对路径还是相对路径的选项。
- 实现`__getitem__`以返回sample和标记。

现在我们了解了必须实现的方法，我们可以基于此来创建自己的实现。在这里，我使用了标准库中的`pathlib`来提取类名并处理我们的路径，因为我发现这个比`os`更容易使用。

```
from pathlib import Path

from timm.data.parsers.parser import Parser

class ParserImageName(Parser):
    def __init__(self, root, class_to_idx=None):
        super().__init__()

        self.root = Path(root)
        self.samples = list(self.root.glob("*.jpg"))

        if class_to_idx:
            self.class_to_idx = class_to_idx
        else:
            classes = sorted(
                set([self.__extract_label_from_path(p) for p in self.samples]),
                key=lambda s: s.lower(),
            )
            self.class_to_idx = {c: idx for idx, c in enumerate(classes)}

    def __extract_label_from_path(self, path):
        return "_".join(path.parts[-1].split("_")[0:-1])

    def __getitem__(self, index):
        path = self.samples[index]
        target = self.class_to_idx[self.__extract_label_from_path(path)]
        return open(path, "rb"), target

    def __len__(self):
        return len(self.samples)

    def _filename(self, index, basename=False, absolute=False):
        filename = self.samples[index][0]
        if basename:
            filename = filename.parts[-1]
        elif not absolute:
            filename = filename.absolute()
        return filename
```

现在，我们可以将解析器实例传递给`ImageDataset`，这应该能够使它正确加载宠物数据集。

```
data_path = Path('pets/images')
ds = ImageDataset(str(data_path), parser=ParserImageName(data_path))
```

让我们通过检查第一个示例来验证解析器是否工作。

```
print(ds[0])
```

从这一点来看，我们的解析器似乎已经工作了！此外，与默认解析器一样，我们可以检查已经执行的类别映射。

```
print(da.parser.class_to_idx)
```

在这个简单的示例中，创建自定义数据集实现只需要稍微多一点的工作。但是，希望这有助于编写自定义解析器并使其与`ImageDataset`一起工作是多么容易！

Optimizers

timm具有大量的优化器，其中有一部分是PyTorch所不具备的。

除了可以方便的使用SGD，Adam和AdamW这些熟悉的优化器之外，一些值得注意的内容包括：

- `AdamP`：https://arxiv.org/abs/2006.08217
- `RMSPropTF`：基于原始TensorFlow实现的RMSProp实现，这里进行了一些小的调整https://github.com/pytorch/pytorch/issues/23796。根据我的经验，这通常比PyTorch版本训练时更加稳定。
- `LAMB`：Apex的FusedLAMB优化器的纯PyTorch变体，在使用PyTorch XLA时与TPU兼容https://nvidia.github.io/apex/optimizers.html#apex.optimizers.FusedLAMB。
- `AdaBelief`：https://arxiv.org/abs/2010.07468。设置超参数的指南可以在这里找到https://github.com/juntang-zhuang/Adabelief-Optimizer#quick-guide。
- `MADGRAD`：https://arxiv.org/abs/2101.11075。
- `AdaHessian`：本文介绍了一种自适应二阶优化器https://arxiv.org/abs/2006.00719。

timm中的优化器支持与`torch.optim`中的优化器相同接口，并且在大多数情况下可以简单的放到训练脚本中，而不需要做任何的修改。

要查看timm实现的所有的优化器，我们可以检查`timm.optim`模块。

```
import inspect

import timm.optim

print([cls_name for cls_name, cls_obj in inspect.getmembers(timm.optim) if
inspect.isclass(cls_obj) if cls_name !='Lookahead'])
```

创建优化器最简单的方法是使用`create_optimiezr_v2`工厂函数，该函数需要以下参数：

- 一个模型或一组参数。
- 优化器的名称。
- 传递给优化器的其他参数。

我们可以使用这个函数创建timm提供的任何优化器实现，以及`timm.optim`提供的流行优化器和Apex提供的融合优化器（如果已经安装Apex）。

让我们来看一些例子。

```
import torch

model = torch.nn.Sequential(
    torch.nn.Linear(2, 1),
    torch.nn.Flatten(0, 1)
)

optimizer = timm.optim.create_optimizer_v2(model, opt='sgd', lr=0.01, momentum=0.8)

print(optimizer)
print(type(optimizer))
```

在这里，我们可以看到，由于timm不包含SGD的实现，它使用`torch.optim`的实现创建了我们的优化器。

让我们尝试创建一个在timm中实现的优化器。

```
optimizer = timm.optim.create_optimizer_v2(model,
                                           opt='lamb',
                                           lr=0.01,
                                           weight_decay=0.01)

print(optimizer)
print(type(optimizer))
```

我们可以验证timm的`Lamb`实现是正确的，并且我们的权重衰减已经被应用到参数组1。

Creating optimizers manually

当然，如果我们不喜欢用`create_optimizer_v2`工厂函数来创建优化器，所有这些优化器都可以按照通常的方式创建。

```
optimizer = timm.optim.RMSpropTF(model.parameters(), lr=0.01)
```

Usage Optimizer Example

现在，我们可以使用这些优化器的大部分，如下图所示：

```
# replace
# optimizer = torch.optim.Adam(model.parameters(), lr=0.1)

# with
optimizer = timm.optim.AdamP(model.parameters(), lr=0.01)

for epoch in num_epochs:
    for batch in training_dataloader:
        inputs, targets = batch
        outputs = model(inputs)
        loss = loss_function(outputs, targets)

        loss.backward()
        optimizer.step()
        optimizer.zero_grad()
```

在写这篇文章时，唯一的例外是二阶`Adahessian`优化器，它在执行`backward`步骤时需要一个小的调整；类似的调整可能需要额外的二阶优化器，它可能在未来添加。

```
optimizer = timm.optim.Adahessian(model.parameters(), lr=0.01)

is_second_order = hasattr(optimizer, 'is_second_order') and optimizer.is_second_order

for epoch in num_epochs:
    for batch in training_dataloader:
        inputs, targets = batch
        outputs = model(inputs)
        loss = loss_function(outputs, targets)

        loss.backward(create_graph=second_order)
        optimizer.step()
        optimizer.zero_grad()
```

Lookahead

timm还使我们能够将前瞻算法应用于优化器；这里给出了介绍：https://arxiv.org/abs/1907.08610；具体详细的解释见：https://www.youtube.com/watch?v=TxGxiDK0Ccc。前瞻可以提高学习的稳定性，降低其内部优化器的方差，计算量和内存消耗可以忽略不计。

我们可以通过在优化器名称前面加上`lookahead_`来将前瞻算法应用于优化器。

```
optimizer = timm.optim.create_optimizer_v2(model.patameters(), opt='lookahead_adam',
lr=0.01)
```

或者通过timm的`Lookahead`类中的优化器实例来进行包装。

```
timm.optim.Lookahead(optimizer, alpha=0.5, k=6)
```

当使用Lookahead时，我们需要更新我们的训练脚本，包括以下一行，以更新慢速权重。

```
optimizer.sync_lookahead()
```

下面是一个如何使用它的一个示例：

```
optimizer = timm.optim.AdamP(model.paraeters(), lr=0.01)
optimizer = timm.optim.Lookahead(optimizer)

for epoch in num_epochs:
    for batch in training_dataloader:
        inputs, targets = batch
        outputs = model(inputs)
        loss = loss_function(outputs, targets)

        loss.backward()
        optimizer.step()
        optimizer.zero_grad()

    optimizer.sync_lookahead()
```

Schedulers

在编写本文时，timm包含以下调度程序：

- `StepLRScheduler`：学习率每n步衰减；类似于`torch.optim.lr_scheduler.StepLR`。
- `MultiStepLRScheduler`：支持多个里程碑以降低学习率的步骤调度器；类似于`torch.optim.lr_scheduler.MultiStepLR`。
- `PlateauLRScheduler`：每当指定的指标趋于平稳时，将学习率降低一个指定的系数；类似于`torch.optim.lr_scheduler.ReduceLROnPlateau`。
- `CosineLRScheduler`：余弦衰减调度器，详细描述见：https://arxiv.org/abs/1608.03983；类似于`torch.optim.lr_scheduler.CosineAnnealingWarmRestarts`。
- `TanhLRScheduler`：hyberbolic-tangent decay schedule with restarts，详细描述见：https://arxiv.org/abs/1806.01593。
- `PolyLRScheduler`：多项式递减调度器；详细描述见：https://arxiv.org/abs/2004.05909。

虽然许多在timm中实现的调度器在PyTorch中也有对应的调度器，但timm版本通常有不同的默认超参数，并提供额外的选项和灵活性；所有timm调度器都支持warmup epochs，以及可以在调度中添加随机噪声。此外，`CosineLRScheduler`和`PolyLRScheduler`支持这里介绍的被称为*k-decay*的衰变选项，详情请见：https://arxiv.org/abs/2004.05909。

首先让我们探讨一下如何在自定义训练脚本中使用来自timm的调度器，然后再研究这些调度器提供的一些选项。

Usage Scheduler Example

与PyTorch中包含的调度器不同的是，良好的做法是在每个epoch中更新timm调度器两次。

- `.step_update` 方法应该在每次优化器更新后被调用，并提供下一次更新的索引；这就是我们为PyTorch调度器调用 `.step` 的地方。
- `.step` 方法应该在每个epoch结束时被调用，并标明下一个纪元的索引。

通过明确提供更新次数和epochs，这使得timm调度器能够消除PyTorch调度器中观察到的混乱的 `last_epoch` 和 `-1` 行为。

下面是一个如何使用timm调度器的例子：

```
training_epochs = 300
cooldown_epochs = 10
num_epochs = training_epochs + cooldown_epochs

optimizer = timm.optim.AdamP(my_model.parameters(), lr=0.01)
scheduler = timm.schduler.CosineLRScheduler(optimizer, t_initial=training_epochs)

for epoch in range(num_epochs):
    num_steps_per_epoch = len(train_dataloader)
    num_updates = epoch * num_steps_per_epoch

    for batch in training_dataloader:
        inputs, targets = batch
        outputs = model(inputs)
        loss = loss_function(outputs, targets)
        loss.backward()
        optimizer.step()
        scheduler.step_update(num_updates=num_updates)
        optimizer.zero_grad()

    scheduler.step(epoch + 1)
```

Adjusting Learning rate schedules

为了展示timm提供的一些选项，我们来探讨一些可用的超参数，以及修改这些参数对学习率调度的影响。

在这里，我们将重点讨论CosineLRScheduler，因为这是timm的训练脚本中默认使用的调度器。如上所述，添加预热和噪声等功能存在于上述所有的调度器中。

为了让我们能够直观的看到学习率的变化，让我们定义一个函数来创建一个模型和优化器，以便与我们的调度器一起使用。注意，由于我们只是更新调度器，模型实际上并没有被优化，但是我们需要一个优化器实例与我们的调度器一起工作，而优化器需要一个模型。

```
def create_model_and_optimizer():
    model = torch.nn.Linear(2, 1)
    optimizer = torch.optim.SGD(model.parameters(), lr=0.05)
    return model, optimizer
```

Using the `CosineAnnealingWarmRestarts` scheduler from PyTorch

为了说明timm的余弦调度器与PyTorch中的调度器不同，让我们先看看我们将如何使用`CosineAnnealingWarmRestarts`的Torch实现。

此类别支持以下参数：

- `T_0` (*int*)：第一次重新启动的迭代次数。
- `T_mult` (*int*)：重启后增加`T_{i}`的系数。（默认：`1`）
- `eta_min` (*float*)：最小学习率。（默认：`0.`）
- `last_epoch` (*int*)：最后一个epoch的索引。（默认：`-1`）

为了设置我们的调度器，我们需要定义以下内容：epochs的数量，每个epoch发生的更新数量，以及——如果我们想要启用重启——学习率应该返回到其初始值的步骤数量。因为我们在这里没有使用任何数据，所以我们可以任意的设置这些。

```
num_epochs=300
num_epoch_repeat = num_epochs//2
num_steps_per_epoch = 10
```

注意：在这里，我们指定学习率在训练运行一半时"重新开始"。这主要是出于可视化的目的——这样我们就可以了解这个调度器的重启情况——而不是在实际训练过程中推荐使用这个方法。

现在，让我们来创建我们的学习率调度器。由于`T_0`要求用迭代次数来指定第一次重启的时间——每一次迭代都是一个批次——我们通过将我们希望重启发生的epoch与每个epoch中的迭代次数相乘来计算。这里，我们还指定了学习率不应低于`1e-6`。

```
model, optimizer = create_model_and_optimizer()
scheduler = torch.optim.lr_scheduler.CosineAnnealingWarmRestarts(optimizer,

 T_0=num_epoch_repeat*num_steps_per_epoch,
                                                                  T_mult=1,
                                                                  eta_min=1e-6,
                                                                  last_epoch=-1)
```

现在，我们可以在训练循环中使用这个调度器进行模拟。由于我们使用的是PyTorch的实现，我们只需要在每次优化器更新后调用`step`，即每批一次。在这里我们记录了每个`step`后的学习率，这样我们就可以直观的看到学习率在整个训练过程中是如何变化的。

```
import matplotlib.pyplot as plt

lrs = []

for epoch in range(num_epochs):

    for i in range(num_steps_per_epoch):
        scheduler.step()

    lrs.append(
        optimizer.param_groups[0]["lr"]
    )

plt.plot(lrs)
```

从这张图中，我们可以看到学习率衰减到第150个epoch时，学习率被重置到它的初始值，然后再次衰减，这正是我们预期的那样。

Using the `CosineLRScheduler` scheduler from timm

```
import timm.scheduler
```

现在我们已经了解了如何使用PyTorch中的余弦调度器，让我们来探讨一下它与timm中的有何不同，以及在timm中额外的选项。我们先用timm中的余弦调度器来实现先前的`CosineLRScheduler`学习率调度。

我们需要参数的一部分与先前相同：

- `t_initial` (*int*)：第一次重新启动的迭代次数，这与`T_0`相同。
- `lr_min` (*float*)：最小学习率，这与`eta_min`相同。（默认：`0.`）
- `cycle_mul` (*float*)：重启后增加`T_{i}`的因子，这与`T_mult`相同。（默认：`1`）

为了复现PyTorch中调度器的行为，我们还需要设置以下参数：

- `cycle_limit` (*int*)：限制一个循环中重启启动的次数。（默认：`1`）
- `t_in_epochs` (*bool*)：迭代次数是否以epochs为单位，而不是以批量更新的次数为单位。（默认：`True`）

首先，让我们定义与前面相同的调度。

```
num_epochs=300
num_epoch_repeat = num_epochs/2
num_steps_per_epoch = 10
```

现在，我们可以创建我们的调度器实例。在这里，我们用更新步骤的数量来表示迭代的数量，并将周期限制增加到超过我们所需的重启次数；这样，参数就与我们之前在Torch的使用的相同。

```
model, optimizer = create_model_and_optimizer()

scheduler = timm.scheduler.CosineLRScheduler(optimizer,

 t_initial=num_epoch_repeat*num_steps_per_epoch,
                                              lr_min=1e-6,
                                              cycle_limit=num_epoch_repeat+1,
                                              t_in_epochs=False)
```

现在，让我们定义一个新函数来模拟在训练运行中使用timm调度器，并记录对学习率的更新。

```
def plot_lrs_for_timm_scheduler(scheduler):
    lrs = []

    for epoch in range(num_epochs):
        num_updates = epoch * num_steps_per_epoch

        for i in range(num_steps_per_epoch):
            num_updates += 1
            scheduler.step_update(num_updates=num_updates)

        scheduler.step(epoch + 1)

        lrs.append(optimizer.param_groups[0]["lr"])

    plt.plot(lrs)
```

现在，让我们展示学习率的变化情况。

```
plot_lrs_for_timm_scheduler(scheduler)
```

正如预期的那样，我们的图表看起来和我们之前看到的一模一样。

现在我们已经复现了我们在torch中的行为，让我们更加详细的看看timm提供的一些额外功能。

到目前为止，我们用优化器更新来表示迭代次数；这需要我们用`num_epoch_repeat * num_steps_per_epoch`来计算第一次迭代的次数。然而，通过以epochs为单位指定我们的迭代——这是timm的默认设置——我们可以避免做这种计算。使用默认设置，我们简单的传递我们想让第一次重启发生的epoch索引，如下图所示：

Adding Warm up and Noise

所有的timm优化器的另外一个特点是，它们支持在学习率调度器中添加预热和噪声。我们可以通过`warmup_t`和`warmup_lr_init`参数指定预热的epochs，以及预热时使用的初始学习率。让我们看看，如果我们指定20个预热epochs，我们的学习率将如何变化。

```
model, optimizer = create_model_and_optimizer()

scheduler = timm.scheduler.CosineLRScheduler(optimizer,
                                             t_initial=num_epoch_repeat,
                                             lr_min=1e-5,
                                             warmup_lr_init=0.01,
                                             warmup_t=20,
                                             cycle_limit=num_epoch_repeat+1
                                            )
plot_lrs_for_timm_scheduler(scheduler)
```

这里，我们已经可以看到学习率的变化已经与先前的有所不同。

我们还可以使用`noise_range_t`和`noise_pct`参数，将噪声添加到某个范围的epochs中，让我们在前150个epochs中添加少量的噪声。

```
model, optimizer = create_model_and_optimizer()

scheduler = timm.scheduler.CosineLRScheduler(optimizer,
                                             t_initial=num_epoch_repeat,
                                             lr_min=1e-5,
                                             noise_range_t=(0, 150),
                                             noise_pct=0.1,
                                             cycle_limit=num_epoch_repeat+1
                                            )

plot_lrs_for_timm_scheduler(scheduler)
```

我们可以看到，直到第150个epoch，添加的噪声影响了我们的调度器，所以学习率不会以平滑曲线下降。我们可以通过增加`noise_pct`来使其更加极端。

```
model, optimizer = create_model_and_optimizer()

scheduler = timm.scheduler.CosineLRScheduler(optimizer,
                                             t_initial=num_epoch_repeat,
                                             lr_min=1e-5,
                                             noise_range_t=(0, 150),
                                             noise_pct=0.8,
                                             cycle_limit=num_epoch_repeat+1
                                            )

plot_lrs_for_timm_scheduler(scheduler)
```

Additional options for `CosineLRScheduler`

虽然预热和噪音可以用于任何调度器，但是有一些额外的功能使专门针对`CosineLRScheduler`。让我们来探讨一下这些是如何影响我们的学习率。

我们可以使用`cycle_mul`，来增加下一次重启的时间，如下所示：

```
model, optimizer = create_model_and_optimizer()

scheduler = timm.scheduler.CosineLRScheduler(optimizer,

 t_initial=num_epoch_repeat*num_steps_per_epoch,
                                             cycle_limit=num_epoch_repeat+1,
                                             cycle_mul=2.,
                                             t_in_epochs=False)
plot_lrs_for_timm_scheduler(scheduler)
```

此外，timm提供了一个选项，用`cycle_limit`来限制重新启动的次数。默认情况下，它被设置为`1`，如下所示：

```
model, optimizer = create_model_and_optimizer()

scheduler = timm.scheduler.CosineLRScheduler(optimizer,
                                             t_initial=num_epoch_repeat,
                                             lr_min=1e-5,
                                             cycle_limit=1)



plot_lrs_for_timm_scheduler(scheduler)
```

`CosineLRScheduler`还支持不同类型的衰减。我们可以使用`cycle_decay`来减少（或增加）每次连续重启时将设置的新的学习率。

```
model, optimizer = create_model_and_optimizer()

scheduler = timm.scheduler.CosineLRScheduler(optimizer,
                                             t_initial=50,
                                             lr_min=1e-5,
                                             cycle_decay=0.8,
                                             cycle_limit=num_epoch_repeat+1)
plot_lrs_for_timm_scheduler(scheduler)
```

注意：这里我们增加了重启次数的频率，以更好的说明衰减。

为了控制曲线本身，我们可以使用`k_decay`参数，对于这个参数，学习率的变化率由它的k阶导数来改变，详情见本文：https://arxiv.org/abs/2004.05909

```
model, optimizer = create_model_and_optimizer()

scheduler = timm.scheduler.CosineLRScheduler(optimizer,
                                             t_initial=num_epoch_repeat,
                                             lr_min=1e-5,
                                             k_decay=0.5,
                                             cycle_limit=num_epoch_repeat+1)
plot_lrs_for_timm_scheduler(scheduler)
```

```
model, optimizer = create_model_and_optimizer()

scheduler = timm.scheduler.CosineLRScheduler(optimizer,
                                             t_initial=num_epoch_repeat,
                                             lr_min=1e-5,
                                             k_decay=2,
                                             cycle_limit=num_epoch_repeat+1)
plot_lrs_for_timm_scheduler(scheduler)
```

这个选项提供了对这个调度器所执行的退火以更多的控制。

Default setting in timm's training script

如果我们使用timm训练脚本的默认设置来设置这个调度器，我们会观察到以下的调度器计划。

注意：在训练脚本中，训练继续进行了10个epochs，没有进一步修改修改学习率以此做为"冷却"。

```
model, optimizer = create_model_and_optimizer()

training_epochs = 300
cooldown_epochs = 10
num_epochs = training_epochs + cooldown_epochs

lr_sched = timm.scheduler.CosineLRScheduler(optimizer,
                                            t_initial=training_epochs,
                                            cycle_decay=0.5,
                                            lr_min=1e-6,
                                            t_in_epochs=True,
                                            warmup_t=3,
                                            warmup_lr_init=1e-4,
                                            cycle_limit=1)
plot_lrs_for_timm_scheduler(lr_sched)
```

正如我们所看到的，在默认设置下根本没有重启!

Other Learning rate schedulers

虽然我最喜欢的调度器是`CosineLRScheduler`，但是其他一些调度器可能会有帮助，因为PyTorch中没有对应的调度器。这两个调度器与余弦调度器类似，都是在指定的epochs后重置学习率——假设没有设置周期限制——但是退火的方式略有不同。

对于`TanhLRScheduler`，退火是使用双曲正切函数进行的，如下图所示。

```
model, optimizer = create_model_and_optimizer()

scheduler = timm.scheduler.TanhLRScheduler(optimizer,
                                           t_initial=num_epoch_repeat,
                                           lr_min=1e-5,
                                           cycle_limit=num_epoch_repeat+1)

plot_lrs_for_timm_scheduler(scheduler)
```

timm还提供了`PolyLRScheduler`，它使用多项式衰减。

```
model, optimizer = create_model_and_optimizer()

lr_sched = timm.scheduler.PolyLRScheduler(optimizer,
                                          t_initial=num_epoch_repeat,
                                          lr_min=1e-5,
                                          cycle_limit=num_epoch_repeat+1)

plot_lrs_for_timm_scheduler(lr_sched)
```

与`CosineLRScheduler`相似，`PolyLRScheduler`也支持`k_decay`参数，如下所示。

```
model, optimizer = create_model_and_optimizer()

lr_sched = timm.scheduler.PolyLRScheduler(optimizer,
                                          t_initial=num_epoch_repeat,
                                          lr_min=1e-5,
                                          cycle_limit=num_epoch_repeat+1,
                                          k_decay=0.5)

plot_lrs_for_timm_scheduler(lr_sched)
```

```
model, optimizer = create_model_and_optimizer()

lr_sched = timm.scheduler.PolyLRScheduler(optimizer,
                                          t_initial=num_epoch_repeat,
                                          lr_min=1e-5,
                                          cycle_limit=num_epoch_repeat+1,
                                          k_decay=2)

plot_lrs_for_timm_scheduler(lr_sched)
```

Exponential Moving Average Model

训练模型时，通过获取在整个训练运行中观察到的参数的移动平均值来设置模型权重的值可能是有益的，而不是使用上次增量更新后获得的参数。在实践中，这通常是通过维护 EMA 模型来完成的，该模型是我们正在训练的模型的副本。但是，我们不是在每个更新步骤后更新此模型的所有参数，而是使用现有参数值和更新值的线性组合来设置这些参数。这是使用以下公式完成的：

$$updated\_EMA\_model\_weights = decay \times EMA\_model\_weights + (1. - decay) \times updated\_model\_weights$$

其中衰减是我们设定的一个参数。例如，如果我们设置`decay=0.99`，我们有：

$$updated\_EMA\_model\_weights = 0.99 \times EMA\_model\_weights + 0.01 \times updated\_model\_weights$$

我们可以看到，它保留了99%的现有状态，而只有1%的新状态！

为了理解为什么这可能是有益的，让我们考虑这样的情况：我们的模型，在训练的早期阶段，某一批数据上表现的特别差。这可能会导致对我们的参数进行大量更新，过度补偿所获得的高损失，这对接下来的批次是不利的。通过只纳入最新参数的一小部分，剧烈的更新将被"平滑"，并对模型的权重产生较小的整体影响。

在评估过程中，这些平均参数有时会产生明显更好的结果，这种技术已经被应用于一些流行模型的训练方案，如训练MNASNet、MobileNet-V3和EfficientNet，使用TensorFlow实现。使用timm中实现的`ModelEmaV2`模块，我们可以复制这种行为，并将同样的做法应用于我们自己的训练脚本。

`ModelEmaV2`需要以下参数：

- `model`：是我们正在训练的`nn.Module`的子类。这就是我们将在训练过程中正常更新的模型。
- `decay` (*float*)：使用的衰减量，这决定了之前的状态将被保持多少。TensorFlow文档表明，合理的衰减值接近于1.0，通常在多个9的范围内，0.999、0.9999等。（默认：`0.9999`）
- `device`：应该用于评估EMA模型的设备。如果没有设置，EMA模型将在用于该模型的同一设备上创建。

让我们来探讨一下如何将这种方法融入到训练循环中。

Usage EMA Example

```
model = create_model().to(gpu_device)
ema_model = ModelEmaV2(model, decay=0.9998)

for epoch in num_epochs:
    for batch in training_dataloader:
        inputs, targets = batch
        outputs = model(inputs)
        loss = loss_function(outputs, targets)

        loss.backward()
        optimizer.step()
        optimizer.zero_grad()

        model_ema.update(model)

    for batch in validation_dataloader:
        inputs, targets = batch
        outputs = model(inputs)
        validation_loss = loss_function(outputs, targets)

        ema_model_outputs = model_ema.model(inputs)
        ema_model_validation_loss = loss_function(ema_model_outputs, targets)
```

我们可以看到，为了更新EMA模型的参数，我们需要在每次参数更新后调用`.update`。由于EMA模型与被训练的模型有不同的参数，我们必须单独评估。

值得注意的是，这个类对它的初始化位置很敏感。在分布式训练中，它应该在转化为`SyncBatchNorm`之前和使用`DistributedDataParallel`包装器之前被应用。

此外，在保存EMA模型时，`state_dict`里面的键将与正在训练的模型的键相同，所以应该使用不同的检查点！

Putting it all together!

虽然本文中的伪代码片段说明了每个组件如何在训练中单独使用，但让我们探讨一个同时使用许多不同组件的例子。

这里，我们将探讨如何在Imagenette上训练一个模型。请注意，由于Imagenette是Imagenet的一个子集，如果我们使用一个预训练的模型，我们就会略微作弊，因为只有新的分类头会用随机权重进行初始化；因此，在这个例子中，我们将重头训练。

注意：这个例子将如何将timm的多个组件一起使用。因此，所选择的特征和所使用的超参数在某种程度上是任意选择的；因此，通过一些仔细的调整，其性能可能会得到提高。

为了去除我们通常在PyTorch训练中看到的模板，比如在`DataLoaders`中迭代和在设备间移动数据，我们将使用`PyTorch-accelerated`来处理我们的训练；这使我们能够只关注使用timm组件时的差异。

如果你不熟悉`PyTorch-accelerated`，并想在深入学习本文之前了解它，请查看博客或文档；另外，它非常简单，缺乏这方面的知识不影响你对这里探讨的内容的理解。

在`PyTorch-accelerated`中，训练是由`Trainer`类处理的；我们可与覆盖特定的方法来改变某些步骤的行为。在伪代码中，`PyTorch_accelerated`的`Trainer`内部的训练行为可以被描述为：

```
train_dl = create_train_dataloader()
eval_dl = create_eval_dataloader()
scheduler = create_scheduler()

training_run_start()
on_training_run_start()

for epoch in num_epochs:
    train_epoch_start()
    on_train_epoch_start()

    for batch in train_dl:
        on_train_step_start()
        batch_output = calulate_train_batch_loss(batch)
        on_train_step_end(batch, batch_output)
        backward_step(batch_output["loss"])
        optimizer_step()
        scheduler_step()
        optimizer_zero_grad()

    train_epoch_end()
    on_train_epoch_end()

    eval_epoch_start()
    on_eval_epoch_start()
    for batch in eval_dl:
        on_eval_step_start()
        batch_output = calculate_eval_batch_loss(batch)
        on_eval_step_end(batch, batch_output)
    eval_epoch_end()
    on_eval_epoch_end()

    training_run_epoch_end()
    on_training_run_epoch_end()

training_run_end()
on_training_run_end()
```

关于训练器是如何工作的更多细节可以在文档中找到。我们可以对默认的训练器进行子类化，并在训练脚本中使用它，如下所示：

```
import argparse
from pathlib import Path

import timm
import timm.data
```

```
import timm.loss
import timm.optim
import timm.utils
import torch
import torchmetrics
from timm.scheduler import CosineLRScheduler

from pytorch_accelerated.callbacks import SaveBestModelCallback
from pytorch_accelerated.trainer import Trainer, DEFAULT_CALLBACKS

def create_datasets(image_size, data_mean, data_std, train_path, val_path):
    train_transforms = timm.data.create_transform(
        input_size=image_size,
        is_training=True,
        mean=data_mean,
        std=data_std,
        auto_augment="rand-m7-mstd0.5-inc1"
    )

    eval_transforms = timm.data.create_transform(
        input_size=image_size,
        mean=data_mean,
        std=data_std
    )

    train_dataset = timm.data.dataset.ImageDataset(
        train_path,
        transform=train_transforms
    )

    eval_dataset = timm.data.dataset.ImageDataset(
        val_path,
        transform=eval_transforms
    )

    return train_dataset, eval_dataset

class TimmMixupTrainer(Trainer):
    def __init__(self, eval_loss_fn, mixup_args, num_class, *args, **kwargs):
        super(TimmMixupTrainer, self).__init__(*args, **kwargs)
        self.eval_loss_fn = eval_loss_fn
        self.num_updates = None
        self.mixup_fn = timm.data.Mixup(**mixup_args)

        self.accuracy = torchmetrics.Accuracy(num_classes=num_class)
        self.ema_accuracy = torchmetrics.Accuracy(num_classes=num_class)
        self.ema_model = None

    def create_scheduler(self):
        return timm.scheduler.CosineLRScheduler(
            self.optimizer,
            t_initial=self.run_config.num_epochs,
            cycle_decay=0.5,
            lr_min=1e-6,
            t_in_epochs=True,
            warmup_t=3,
            warmup_lr_init=1e-4,
            cycle_limit=1
        )
```

```
    def training_run_start(self):
        # Model EMA requires the model without a DDP wrapper and before sync batchnorm
conversion
        self.ema_model = timm.utils.ModelEmaV2(
            self._accelerator.unwrap_model(self.model),
            decay=0.9
        )
        if self.run_config.is_distributed:
            self.model = torch.nn.SyncBatchNorm.convert_sync_batchnorm(self.model)

    def train_epoch_start(self):
        super(TimmMixupTrainer, self).train_epoch_start()
        self.num_updates = self.run_history.current_epoch * len(self._train_dataloader)


    def calculate_train_batch_loss(self, batch) -> dict:
        xb, yb = batch
        mixup_xb, mixup_yb = self.mixup_fn(xb, yb)
        return super(TimmMixupTrainer, self).calculate_train_batch_loss((mixup_xb,
mixup_yb))

    def train_epoch_end(self):
        self.ema_model.update(self.model)
        self.ema_model.eval()

        if hasattr(self.optimizer, "sync_lookahead"):
            self.optimizer.sync_lookahead()

    def scheduler_step(self):
        self.num_updates += 1
        if self.scheduler is not None:
            self.scheduler.step_update(num_updates=self.num_updates)

    def calculate_eval_batch_loss(self, batch) -> dict:
        with torch.no_grad():
            xb, yb = batch
            outputs = self.model(xb)
            val_loss = self.eval_loss_fn(outputs, yb)
            self.accuracy.update(outputs.argmax(-1), yb)

            ema_model_preds = self.ema_model.module(xb).argmax(-1)
            self.ema_accuracy.update(ema_model_preds, yb)

        return {"loss": val_loss, "model_outputs": outputs,"batch_size": xb.size(0)}

    def eval_epoch_end(self):
        super(TimmMixupTrainer, self).eval_epoch_end()

        if self.scheduler is not None:
            self.scheduler.step(self.run_history.current_epoch + 1)

        self.run_history.update_metric("accuracy", self.accuracy.compute().cpu())
        self.run_history.update_metric(
            "ema_model_accuracy", self.ema_accuracy.compute().cpu()
        )
        self.accuracy.reset()
        self.ema_accuracy.reset()
```

```
def main(data_path):
    # Set training argument
    image_size = (224, 224)
    lr = 5e-3
    smoothing = 0.1
    mixup = 0.2
    cutmix = 1.0
    batch_size = 32
    bce_target_thresh = 0.2
    num_epochs = 40

    data_path = Path(data_path)
    train_path = data_path / "train"
    val_path = data_path / "val"
    num_classes = len(list(train_path.iterdir()))

    mixup_args = dict(
        mixup_alpha=mixup,
        cutmix_alpha=cutmix,
        label_smoothing=smoothing,
        num_classes=num_classes
    )

    model = timm.create_model(
        "resnet50d",
        pretrained=False,
        num_classes=num_classes,
        drop_path_rate=0.05
    )

    data_config = timm.data.resolve_data_config({}, model=model, verbose=True)
    data_mean = data_config["mean"]
    data_std = data_config["std"]

    train_dataset, eval_dataset = create_datasets(
        train_path=train_path,
        val_path=val_path,
        image_size=image_size,
        data_mean=data_mean,
        data_std=data_std
    )

    optimizer = timm.optim.create_optimizer_v2(
        model,
        opt="lookahead_AdamW",
        lr=lr,
        weight_decay=0.01
    )

    train_loss_fn = timm.loss.BinaryCrossEntropy(
        target_threshold=bce_target_thresh,
        smoothing=smoothing
    )
    validate_loss_fn = torch.nn.CrossEntropyLoss()

    trainer = TimmMixupTrainer(
        model=model,
        optimizer=optimizer,
```

```
        loss_func=train_loss_fn,
        eval_loss_fn=validate_loss_fn,
        mixup_args=mixup_args,
        num_class=num_classes,
        callbacks=[
            *DEFAULT_CALLBACKS,
            SaveBestModelCallback(watch_metric="accuracy", greater_is_better=True)
        ]
    )

    trainer.train(
        per_device_batch_size=batch_size,
        train_dataset=train_dataset,
        eval_dataset=eval_dataset,
        num_epochs=num_epochs,
        create_scheduler_fn=trainer.create_scheduler
    )


if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="Simple example of training script
using timm.")
    parser.add_argument("--data_dir", required=True, help="The data folder on disk.")
    args = parser.parse_args()
    main(args.data_dir)
```

现在，让我们为我们的训练创建一个配置文件。

```
compute_environment: LOCAL_MACHINE
deepspeed_config: {}
distributed_type: MULTI_GPU
fp16: true
machine_rank: 0
main_process_ip: null
main_process_port: null
main_training_function: main
num_machines: 1
num_processes: 1
```

我们现在用以下命令启动我们的训练：

```
accelerate launch --config_file accelerate_config.yaml train.py --data_dir
imagenette/imagenette2-320
```

在Imagenette上使用这个训练脚本，使用2个GPU，按照这里说明，我获得以下指标：

- accuracy: 0.89
- ema_model_accuracy: 0.85

在34个epochs后，考虑到超参数并没有仔细调整，这还算不错！

Conclusion

希望这能在一定程度上全面介绍了timm中包含的功能，以及如何将这些功能应用于自定义的训练脚本。

最后，我想用一点时间来感谢timm的创建者Ross Wightman为创建这个很棒的库所付出的巨大努力。Ross致力于提供最先进的计算机视觉模型的实现，使整个数据科学界都能够轻松使用，这是首屈一指的。如果你还没有使用过，就去Github看看吧！