

Lesson 12 Notes: Advanced training techniques; ULMFiT from scratch

概述

本课实现了一些今天非常重要的训练技术，全部使用回调：

- **MixUp**: 一种数据增强技术，可显着改善结果，尤其是当您的数据较少或可以训练更长时间时。
- **标签平滑**: 它与 MixUp 配合得特别好，当你有嘈杂的标签时，可以显着改善结果
- **混合精度训练**: 在许多情况下训练模型的速度提高 3 倍左右。
- **它还实现了 XResNet**: 这是经典 resnet 架构的调整版本，提供了实质性的改进。而且，更重要的是，它的开发提供了对什么使架构运行良好的深刻见解。
- **最后**，本课展示了如何从头开始实现 ULMFiT，包括构建 LSTM RNN，并查看处理自然语言数据以将其传递到神经网络所需的各个步骤。

链接和参考

- [第12课视频讲座](#)
- [标签平滑 [paperswithcode/methods](#)]
- [混合精度训练 [NVIDIA 博客](#)]
- [浮动工作原理的一个很好的解释\[YouTube\]](#)。
- 该视频通过[在位级别添加两个浮点数 [YouTube](#)]
- [Backpropagation Through Time 博客文章 [机器学习精通](#)]
- [由 Stefano Giomo 撰写的FastAI 论坛帖子](#)揭开了[RNN](#)中序列长度、BPTT 和批次大小之间的差异的神秘面纱。
- RNN 复习：[可视化RNN](#) [[Josh Varty](#)]
- [了解 LSTM 网络 [Chris Olah](#)]
- [了解 ULMFiT 的构建块 [Kerem Turgutlu; 中等的](#)]
- paper:
 - 快速图像增强：[Mixup 论文 [2017](#)]、[CutOut 论文 [2017](#)]、[CutMix [2019](#)]
 - [标签平滑何时有帮助？ [2019](#)]
 - [原始 ResNet 论文 [2015](#)]
 - [Bag of Tricks 纸 [2019](#)]
 - AWD-LSTM -[正则化和优化 LSTM 语言模型，Merity 等人。 [2017](#)]

Today we're going to start to move from a minimal training loop to something that is SoTA on Imagenet*



- 要完成transforms, mixup, label smoothing, Resnet架构等。

在09b_learner.ipynb中的Runner做了一些修改，改掉了learner，因为它只存了四个值。然后有add_cbs, add_cb函数来添加回调函数。

- 理解代码的唯一方式，就是run。
- 之前在init的时候构建opt，现在是在fit的时候构建Opt。可以在fit的时候改变学习率、逐步解冻等事情，可以在fit的时候改变。虽然只有一行代码，在从概念上，它非常重要。所以这是对9b的一些小改动

```
1 #export
2 from exp.nb_10 import *
3 path = datasets untar_data(datasets.URLS.IMGNETTE_160)
4 # rgb图像 resize, 转为tensor和float
5 tfms = [make_rgb, ResizeFixed(128), to_byte_tensor, to_float_tensor]
6 bs = 64
7
8 il = ImageList.from_files(path, tfms=tfms)
9
10 sd = SplitData.split_by_func(il, partial(grandparent_splitter,
11 valid_name='val'))
11 ll = label_by_func(sd, parent_labeler, proc_y=categoryProcessor())
12 data = ll.to_databunch(bs, c_in=3, c_out=10, num_workers=4)
```

更好的图像训练：混合/标签平滑

- 上一课结束时对用GPU做数据增强很兴奋。GPU加速的数据增强，仅使用简单的pytorch操作，这是巨大的胜利。但很可能我们不再需要这种数据增强，因为对我们对这种称为混合的数据增强的实验中，我们发现我们可以删除大多数其他数据增强并获得惊人的好结果

混合

(笔记本：[10b_mixup_label_smoothing.ipynb](#))

- 我们很可能不再需要对图像进行大量数据增强。FastAI 对称为**Mixup**的数据增强进行了实验，他们发现他们可以删除大多数其他数据增强并获得惊人的好结果。这真的很简单，您还可以使用MixUp 进行很长时间的训练并获得非常好的结果。

- MixUp 来自论文: [mixup: Beyond Empirical Risk Minimization [2017](#)]。这是一篇非常容易阅读的论文。
- MixUp 在论文中被证明是一种非常有效的训练技术: [Bag of Tricks for Image Classification with Convolutional Neural Networks [2019](#)]。 (本课将大量引用这篇论文, 因此也值得一读。)

以下是该论文中尝试的不同训练技巧的结果表:

Refinements	ResNet-50-D		Inception-V3		MobileNet	
	Top-1	Top-5	Top-1	Top-5	Top-1	Top-5
Efficient	77.16	93.52	77.50	93.60	71.90	90.53
+ cosine decay	77.91	93.81	78.19	94.06	72.83	91.00
+ label smoothing	78.31	94.09	78.40	94.13	72.93	91.14
+ distill w/o mixup	78.67	94.36	78.26	94.01	71.97	90.89
+ mixup w/o distill	79.15	94.58	78.77	94.39	73.28	91.30
+ distill w/ mixup	79.29	94.63	78.34	94.16	72.51	91.02

- (注意 MixUp 他们跑了更多的epoch)
- 当他们打开mixup时, 他们也开始训练 200 个 epoch, 而不是 90 个, 而不是 120 个, 因此当你解释他们的纸质表格时要小心一点, 当它从 label smoothing 94.1 到与蒸馏混合时 94.58。他们也几乎翻了一番他们所做的epoch数, 但你可以感觉到你可以大大减少错误。
- **什么是mixup?** 我们将拍摄两个不同的图像, 然后将它们组合起来。如何? 通过简单地将两者进行凸组合。所以你做了一张图片的 30% 和另一张图片的 70%:
 - 但我们用多少比例的图像mixup, 我们将随机使用权重。但是不是均匀分布, 而是 β 分布!
 - why beta 分布?
 - 大部分时间随机数接近0或者1, 偶尔会解决0.5. 大部分还是单个图像, 只是偶尔预测一些0.5的mixup。
- 顾名思义, [mixup 文章](#)的作者, 正如名称所暗示的那样, mixup文章的作者提出在训练集的混合图片上训练模型。例如, 假设我们在CIFAR10上, 然后不给模型提供原始图像, 我们取两个(可能是同一类, 也可能不是), 并对它们进行线性组合:从张量的角度



- `new_image = t * image1 + (1-t) * image2`
- 这里t是一个介于0和1之间的浮点数。那么我们分配给图像的目标就是原始目标的相同组合:您还必须对标签进行 MixUp。因此, 您的标签将变成如下所示: `y = [0.3 (gas pump), 0.7 (dog)]`
- `new_target = t * target1 + (1-t) * target2`

假设你的目标是一个热点编码(这不是pytorch通常的情况)。就是这么简单。

```

1 | img1 = PIL.Image.open(l1.train.x.items[0])
2 | img1
3 | img2 = PIL.Image.open(l1.train.x.items[4000])
4 | img2
5 | mixed_up = l1.train.x[0] * 0.3 + l1.train.x[4000] * 0.7
6 | plt.imshow(mixed_up.permute(1,2,0));

```

Implementation

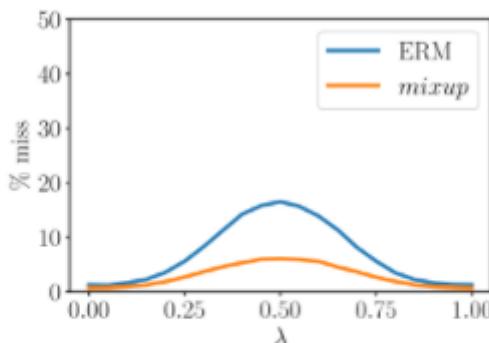
- 这个实现依赖于一个叫做beta分布的东西，这个分布反过来又使用了一个叫伽马函数的东西，杰里米仍然觉得有点可怕。为了克服他的恐惧，杰里米提醒自己，只是一个阶乘函数，(有点)也可以很好地平滑地插值到非整数。它究竟是如何做到这一点并不重要.....
- pytorch没有 γ 函数，但有一个 loggamma 函数。

```

1 | # PyTorch has a log-gamma but not a gamma, so we'll create one
2 | Γ = Lambda x: x.loggamma().exp()

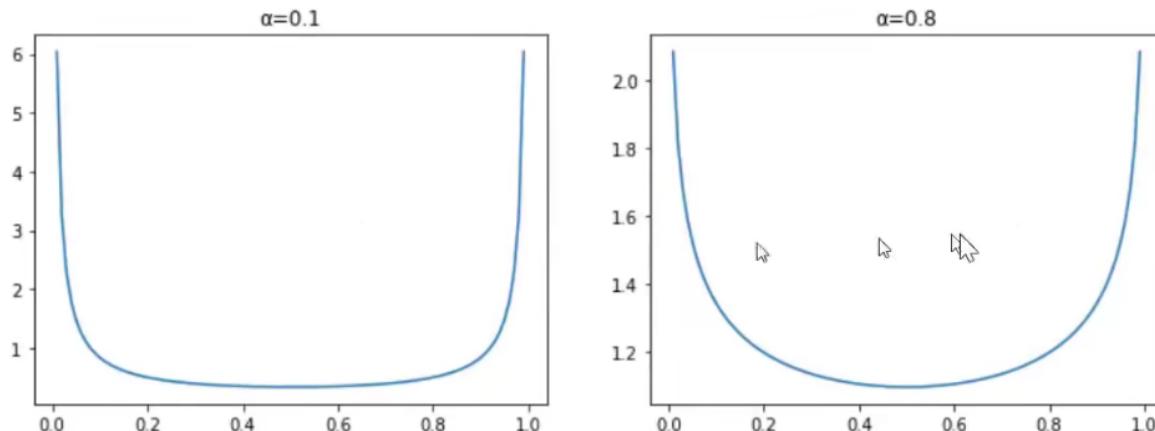
```

- 当我们生成这些混合训练示例时，我们需要选择我们将使用的每个图像的多少。我们定义一个混合比例 λ ，因此我们的混合图像将是 $\lambda x_i + (1 - \lambda) x_j$ 。我们每次随机选取 λ ，但是我们不想简单地从均匀分布中生成它。在MixUp的文章中，**他们探索了混合参数如何影响性能**，并得出如下图(越高越差)：



(a) Prediction errors in-between training data. Evaluated at $x = \lambda x_i + (1 - \lambda) x_j$, a prediction is counted as a “miss” if it does not belong to $\{y_i, y_j\}$. The model trained with mixup has fewer misses.

- 为了得到好的值，我们需要从一个更有可能选择接近0或1的数字的分布中取样。这样的分布就是gamma分布。这是一种奇怪的分布，它不是很直观从它的公式，但它的形状看起来是这样的两个不同的参数值 α :

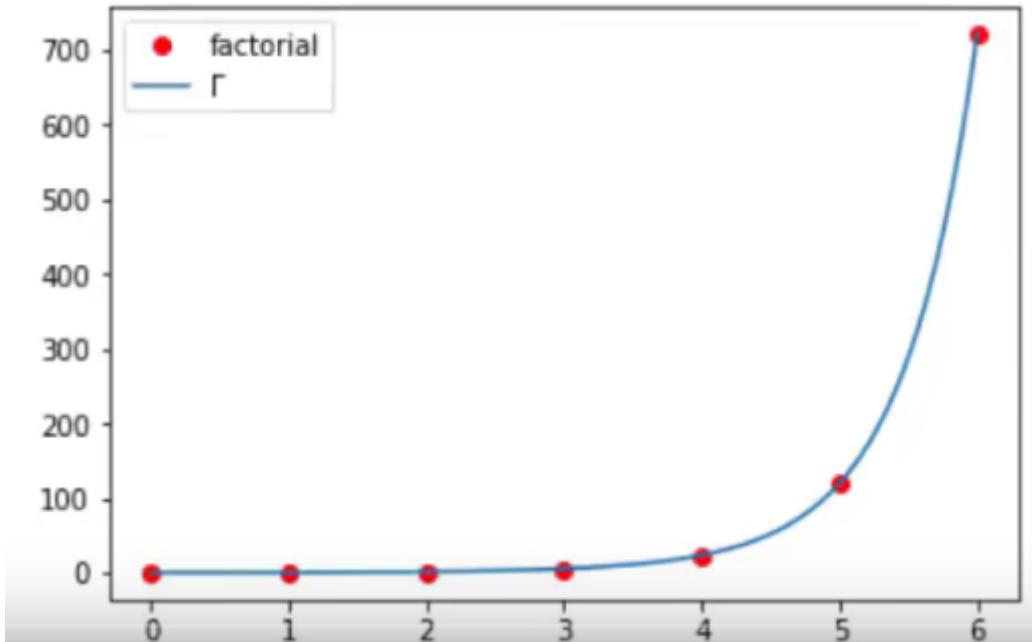


- α 太大，我们大概率会得到相等的mixup。 α 太低，则概率是0和1.因为数据增强我们需要能够调整一个杠杆，说明我正在做多少正则化，我在做多少方向，所以你可以上下移动你的

alpha。

- 注意：如果你看到你不知道的数学符号，你可以像这样在谷歌上搜索它们：[函数](#)。
- 如果你不习惯打字的unicode符号，在Mac类型ctrl- cmd-space弹出一个搜索框表情符号。在Linux上，您可以使用[compose键](#)。在Windows上，您也可以使用组合键，但您首先需要安装[WinCompose](#)。默认情况下，compose键是右手Alt键。
- 您可以在WinCompose中搜索符号名称。希腊字母通常是compose- *- letter（例如，字母是希腊字母 α alpha）。

```
1 facts = [math.factorial(i) for i in range(7)]
2 plt.plot(range(7), facts, 'ro')
3 plt.plot(torch.linspace(0,6), gamma(torch.linspace(0,6)+1))
4 plt.legend(['factorial', 'Γ']);
5 torch.linspace(0,0.9,10)
```

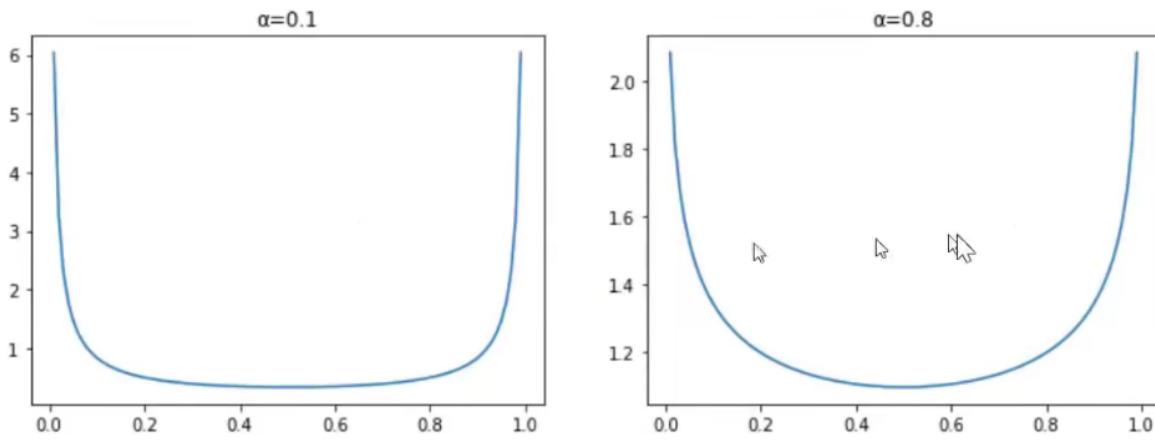


在原始文章中，作者提出了三件事：

1. 创建两个单独的数据加载器，并在每次迭代时从每个加载器中抽取一批以混合它们
2. 在具有参数 α 的beta分布之后绘制值（他们的文章中建议为0.4）
3. 将具有相同值 t 的两个批次混合在一起。
4. 使用one-hot编码目标

为什么 β 分布具有相同的参数 α ？它看起来像这样：

```
1 _,axs = plt.subplots(1,2, figsize=(12,4))
2 x = torch.linspace(0,1, 100)
3 for alpha,ax in zip([0.1,0.8], axs):
4     alpha = tensor(alpha)
5     y = (x.pow(alpha-1) * (1-x).pow(alpha-1)) / (gamma_func(alpha ** 2) /
6         gamma_func(alpha))
6     y = (x**(alpha-1) * (1-x)**(alpha-1)) / (gamma_func(alpha ** 2) / gamma_func(2*alpha))
7     ax.plot(x,y)
8     ax.set_title(f"α={alpha:.1f}")
```



- Beta 分布倾向于在边缘生成数字。如果将其与上面的预测误差图进行比较，您会发现它们互为倒数。（另外：[这篇关于 Beta 发行版背后直觉的 SO 帖子](#)非常有趣）。
- 我们来看看 MixUp 的实现.....

原始 MixUp 算法：在原始文章中，作者提出了三件事：

1. 创建两个单独的数据加载器，并在每次迭代时从每个加载器中抽取一批以混合它们
2. 根据带有参数 α 的beta分布绘制 λ 值（他们的文章中建议使用 0.4）
3. 将两个具有相同 λ 值的批次混合。
4. 使用 one-hot 编码目标

虽然上面的方法非常有效，但并不是最快的方法。减慢这一过程的主要原因是每次迭代都需要两个不同的批次（这意味着加载两倍数量的图像，并对它们应用其他数据增强功能）。为了避免这种速度变慢，我们可以更聪明一点，将批处理与自己的打乱版本混合（这样混合后的图像仍然是不同的）。这是 MixUp 论文中提出的一个技巧。

- **FastAI MixUp Algorithm:** FastAI采用了一些技巧来改进它：
 1. 创建一个数据加载器，绘制一个批处理， X ，带有标签 y ，我们可以通过变换这个批处理来创建混合图像。
 2. 对于批处理中的每个项目，选择一个生成 λ 值的向量($\alpha=0.4$ 的Beta分布)。为了避免潜在的重复混淆，修复 λ 值: $t = \max(t, 1-t)$
 3. 创建一批 X' 和标签 y' 的随机排列。
 4. 返回原批次和随机排列的线性组合: $\lambda X + (1-\lambda)X'$ 。同样地， $\lambda y + (1-\lambda)y'$ 。
- 一个技巧是为批处理中的每个图像选择不同的 λ ，因为fastai发现这样做可以使网络**更快地收敛**。
- 第二个技巧是使用单个批并将其用于MixUp，而不是加载两个批。但是，此策略可能会创建重复项。假设这个批处理有两张图片，我们先将Image0和Image1混合， $\lambda=0.1$ ，然后将Image1和Image0混合， $\lambda=0.9$:

```
1 | image0 * 0.1 + shuffle0 * (1-0.1) = image0 * 0.1 + image1 * 0.9
2 | image1 * 0.9 + shuffle1 * (1-0.9) = image1 * 0.9 + image0 * 0.1
```

- 对于低 α ，我们以高概率选择接近 0. 和 1. 的值，中间的值都具有相同的概率。越大 α ，0. 和 1. 得到的概率越低。
- 然后 pytorch 非常小心地避免在可能的情况下使用 one-hot 编码目标，因此撤消此操作似乎有点麻烦。对我们来说，幸运的是，如果损失是典型的交叉熵我们有

```
1 | loss(output, new_target) = t * loss(output, target1) + (1-t) * loss(output, target2)
```

- 所以我们不会对任何东西进行one-hot编码，只计算这两个损失，然后进行线性组合。

- 对整批使用相同的参数 t 似乎也有点低效。在我们的实验中，我们注意到如果我们为批次中的每张图像绘制不同的 t ，模型可以训练得更快（两个选项在准确性方面得到相同的结果，只是一个到达那里的速度更慢）。
- 我们必须应用的最后一个技巧是，这种策略可能会有一些重复：我们假设或者shuffle假设混合image0和image1，然后image1和image0，我们绘制 $t=0.1$ ，第二个 $t=0.9$ 。

```
1 | image0 * 0.1 + shuffle0 * (1-0.1) = image0 * 0.1 + image1 * 0.9
2 | image1 * 0.9 + shuffle1 * (1-0.9) = image1 * 0.9 + image0 * 0.1
```

将是相同的。当然，我们不得不有点不走运，但在实践中，我们看到使用这个而不去除那些近似重复的数据会降低准确性。为了避免它们，技巧是替换我们绘制的参数向量

```
1 | t = max(t, 1-t)
```

两个参数相等的 beta 分布在任何情况下都是对称的，这样我们可以确保最大的系数总是在第一张图像(未打乱的批处理)附近。

在Mixup中，我们有处理损失的函数，它具有属性约化(如nn.CrossEntropy())。为了处理各种类型的损失函数的reduction=None，而不修改范围外的实际损失函数，我们需要在不减少的情况下执行这些操作，我们创建一个上下文管理器：

```
1 | #export
2 | class NoneReduce():
3 |     def __init__(self, loss_func):
4 |         self.loss_func, self.old_red = loss_func, None
5 |
6 |     def __enter__(self):
7 |         if hasattr(self.loss_func, 'reduction'):
8 |             self.old_red = getattr(self.loss_func, 'reduction')
9 |             setattr(self.loss_func, 'reduction', 'none')
10 |             return self.loss_func
11 |         else: return partial(self.loss_func, reduction='none')
12 |
13 |     def __exit__(self, type, value, traceback):
14 |         if self.old_red is not None: setattr(self.loss_func, 'reduction',
| self.old_red)
```

这是callback MixUp的代码。该begin_batch方法实现了上述算法：

```
1 | #export
2 | from torch.distributions.beta import Beta
3 |
4 | def unsqueeze(input, dims):
5 |     for dim in listify(dims): input = torch.unsqueeze(input, dim)
6 |     return input
7 |
8 | def reduce_loss(loss, reduction='mean'):
9 |     return loss.mean() if reduction=='mean' else loss.sum() if
| reduction=='sum' else loss
```

- mixup需要我们改变我们的loss函数。mixup的loss_func
- 一组只是常规组，第二组是我们将随机抓取所有其他图像置换一个并随机选择一个共享**
- reduction的使用，有很多细节。

- 在计算两种loss的时候，要关闭reduction
- 在batch计算完了之后，有mean和sum

```

1
2 #export
3 class Mixup(Callback):
4     _order = 90 #Runs after normalization and cuda
5     def __init__(self, α:float=0.4): self.distrib = Beta(tensor([α]),
6      tensor([α]))
7
8     def begin_fit(self):
9         self.old_loss_func, self.run.loss_func =
10        self.run.loss_func, self.loss_func
11
12     def begin_batch(self):
13         # 只有训练时做mixup, valid不做mixup
14         # 就在自己的batch里面做一个mixup, 自己跟自己相加, λ加数据, λ加标签
15         if not self.in_train: return #Only mixup things during training
16         λ =
17         self.distrib.sample((self.yb.size(0),)).squeeze().to(self.xb.device)
18         λ = torch.stack([λ, 1-λ], 1)
19         self.λ = unsqueeze(λ.max(1)[0], (1,2,3))
20         shuffle = torch.randperm(self.yb.size(0)).to(self.xb.device)
21         xb1, self.yb1 = self.xb[shuffle], self.yb[shuffle]
22         self.run.xb = lin_comb(self.xb, xb1, self.λ)
23
24     def after_fit(self): self.run.loss_func = self.old_loss_func
25
26     def loss_func(self, pred, yb):
27         if not self.in_train: return self.old_loss_func(pred, yb)
28         with NoneReduce(self.old_loss_func) as loss_func:
29             # 训练时, 计算两种不同的图像损失, 一个normal; 对损失也要这样做
30             loss1 = loss_func(pred, yb)
31             loss2 = loss_func(pred, self.yb1)
32             loss = lin_comb(loss1, loss2, self.λ)
33             # 返回batch的loss的均值或者和, 或者本身。
34             return reduce_loss(loss, getattr(self.old_loss_func, 'reduction',
35 'mean'))
36
37 nfs = [32,64,128,256,512]
38
39 def get_learner(nfs, data, lr, layer, loss_func=F.cross_entropy,
40                  cb_funcs=None, opt_func=optim.SGD, **kwargs):
41     model = get_cnn_model(data, nfs, layer, **kwargs)
42     init_cnn(model)
43     return Learner(model, data, loss_func, lr=lr, cb_funcs=cb_funcs,
44 opt_func=opt_func)
45
46 cbfs = [partial(AvgStatsCallback,accuracy),
47          CudaCallback,
48          ProgressCallback,
49          partial(BatchTransformCallback, norm_imagenette),
50          Mixup]
51
52 learn = get_learner(nfs, data, 0.4, conv_layer, cb_funcs=cbfs)
53 learn.fit(1)

```

问题:softmax如何与所有这些互动? 我们应该直接 from mixup to inference?吗?

- 因为softmax是One-hot coding, 标签是确定的0和1, 这不是很好。也不能跟我们的label smoothing和mixup兼容。我们有标签噪声, 肯定不能100%对标签肯定, 所以不希望one-hot coding, 希望是概率性的标签。这样就可以使用mixup了。这就是下面要将的label smoothing。
- softmax希望一个类别高, 其他类别低这种。所以要用好Mixup, 只要数据的标签含有标签噪声, 就不会100%正确。所以不要one-hot编码, 而是90%正确的标签。这个称为标签平滑。
- **我们如何修改损失函数?** 看 `loss_func` 上面的方法。就像我们对交叉熵损失进行编码时一样, 我们不需要将目标扩展为完整的分类分布, 而是可以为 MixUp 编写一个专门的交叉熵版本:

```
1 | loss(output, new_target) = t * _loss(output, target1) + (1-t) * _loss(output, target2)
```

- PyTorch 损失函数 `nn.CrossEntropy` 有一个 `reduction` 属性来指定如何从单个损失计算整批的损失, 例如取平均值。
- 我们希望在计算出单个损失的线性组合后对批次进行这种reduction。
- 所以对于线性组合需要关闭reduction, 然后再开启。
- 这些计算在GPU上进行, 所以速度很快。超级强大的增强系统这不会给我们的代码增加任何开销, 需要注意的一件事是我们实际上正在替换损失函数, 损失函数还有reduction的东西。
- mixup它真的很有趣, 因为你可以将它用于输入层以外的层, 你可以在第一层使用它, 也许与嵌入一起使用, 所以你可以这样做例如, 在 NLP 中混合增强, 就像人们还没有真正深入研究过的东西, 但这似乎是一个机会, 可以在许多我们目前没有真正看到它的地方添加增强, 这意味着我们可以训练更好的模型和更少的数据
 - mixup的概念很有趣, 不止可以在第一层使用, 也可以在后面的层使用。人们研究的比较少, 这里可能是一个机会。

Label smoothing

- **这是处理数据中的噪声标签的一种非常简单但非常有效的方法。** 例如, 在医疗问题中, 诊断标签并不完美。结果是, 如果你使用标签平滑, 噪音标签通常不是那么大的问题。有趣的是, 人们故意排列他们的标签, 所以他们有50%的错误, 他们仍然得到良好的结果, 标签平滑。这也可能让你在花费大量时间清理数据之前, 更快地接受训练, 检查某些东西是否有效。
 - 所以不要听信其他人说, 你数据没清理完, 你就不能开始model, 或者应该停止model。看看结果是否正确, 如果结果正确, 那么也许您可以跳过所有清理工作或同时进行这些工作
 -
- 另一种经常用于分类的正则化技术是标签平滑, 它故意为标签引入噪声。它的设计是通过改变目标标签, 使模型的决策不那么确定:
- 我们不再硬性预测正确类别为一, 其他类别为零, 而是将目标改为正确类别为 $1-\epsilon$, 其他类别为 $\epsilon/(k-1)$, 其中 ϵ 是个小正数, k 代表类别的数量。
- 我们可以将损失更新为:

$$loss = (1 - \epsilon) \text{ce}(i) + \epsilon \sum_j \text{ce}(j)/(k - 1)$$

- where $\text{ce}(x)$ is the cross-entropy of x (i.e. $-\log(px)$), and i is the correct class. Typical value: $\epsilon=0.1$
- 有噪声的标签并没有你想的那么严重。

```

1 #export
2 class LabelSmoothingCrossEntropy(nn.Module):
3     def __init__(self, ε:float=0.1, reduction='mean'):
4         super().__init__()
5         self.ε, self.reduction = ε, reduction
6
7     def forward(self, output, target):
8         c = output.size()[-1]
9         log_preds = F.log_softmax(output, dim=-1)
10        loss = reduce_loss(-log_preds.sum(dim=-1), self.reduction)
11        nll = F.nll_loss(log_preds, target, reduction=self.reduction)
12        return lin_comb(loss/c, nll, self.ε)
13
14 cbfs = [partial(AvgStatsCallback, accuracy),
15          CudaCallback,
16          ProgressCallback,
17          partial(BatchTransformXCallback, norm_imagenette)]
18 learn = get_learner(nfs, data, 0.4, conv_layer, cb_funcs=cbfs,
19 loss_func=LabelSmoothingCrossEntropy())
20 learn.fit(1)
21 assert learn.loss_func.reduction == 'mean'

```

epoch	train_loss	train_accuracy	valid_loss	valid_accuracy	time
0	1.922940	0.387002	1.921750	0.404000	00:07

- 我们可以把它作为损失函数放进去，代替通常的交叉熵。
- Additional reading:
 - [Label Smoothing [paperswithcode/methods](#)]
 - [When Does Label Smoothing Help? [2019](#)]

Training in Mixed Precision

10c_fp16.ipynb

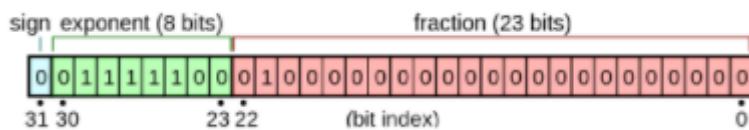
- 所以这是一种超级强大的技术，它已经存在了几年这两种技术，但不是几乎尽可能广泛地使用，然后如果您使用的是 volta tensor core 2080 任何种类的几乎任何当前一代的 nvidia 显卡，您都可以在理论上使用半精度浮点进行训练，理论快10倍。
- 如果您使用的是现代加速器，则可以使用半精度浮点数进行训练。这些只是 16 位浮点数 (FP16)，而不是通常的单精度 32 位浮点数 (FP32)。从理论上讲，这应该可以将速度提高 10 倍，但在实践中，您可以在深度学习中获得 2-3 倍的加速。
- 使用 FP16 将内存使用量减少一半，因此您可以将模型的大小加倍，并将批量大小加倍。现代加速器中的专用硬件单元，例如张量核心，也可以更快地在 FP16 上执行操作。在 Volta 一代 NVIDIA 卡上，这些张量核心理论上提供了 8 倍的加速（遗憾的是，只是理论上）。
- 因此，半精度训练更适合您的内存使用，如果您有 Volta GPU，速度会更快（如果没有，仍然会快一点，因为计算最简单）。我们如何使用它？在 PyTorch 中，您只需要加载 .half() 所有张量即可。问题是，你通常不会在最后看到相同的精度，因为半精度并不是很精确，很有趣。
- 但时nvidia指出，又不能在所有地方都使用半精度，因为总是半精度训练不准确，是颠簸的，因为半精度经常四舍五入。所以只在一些地方使用半精度。如前向传播和反向传播中使用半精度，也就是Hard work都在fp16上进行，其他地方使用fp32。

- 在 $w=w-lr*w.grad$, 由于grad的精度与w差几个数量级, 所以step更新的这部分必须在FP32上进行。

继续这里关于 fastai_v1 开发的文档是关于混合精度训练的简短文章。对它的一个非常好的和清晰的介绍是[来自 NVIDIA 的这个视频](#)。

Aside: Some Floating Point Revision

- 浮点数可能看起来很神秘或有点像黑暗艺术, 但它们确实非常优雅且易于理解。首先必须了解它们基本上类似于科学记数法, 除了以 2 为基数而不是以 10 为基数:
- $x = 0.1101101 \times 2^4, x = (-1)^s \times M \times 2^E$
- 在IEEE浮点数中, 标准浮点数用上式表示, 其中:
 - 符号s决定是负数(s=0)还是正数(s=1)
 - M是介于[1, 2-epsilon]的二进制小数, 或者是[0, 1-epsilon]的小数
 - 指数E用2的a次幂(可能是负的)加权值。
- 这三个部分中的每一部分都占用一定数量的位。这里是32位浮点数的表示:([Source](#))

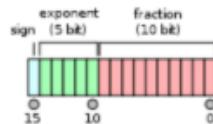


- 以下是进一步介绍材料的一些链接:
 - 关于浮动如何工作的一个很好的解释: [YouTube](#)。
 - 该视频通过在位级别添加两个浮点数来工作: [YouTube](#)

半精度问题

在高精度（FP32、FP64）下，您有足够的喘息空间，大多数时候您无需担心近似值崩溃并且一切都变得糟糕的情况。在 FP16，您必须不断考虑边缘情况。

让我们从位层面来看看 FP16 是什么样子的：



(来源)

- 指数有 5 位，给它一个范围 [-14, 15]。
- 分数有 10 位。
- FP16 范围： 2^{-14} 到 2^{15} 大约
- FP32 范围： 2^{-126} 到 2^{127}
- 数字之间的“空格”在 FP16 中增加。1 到 2 之间的浮点数是有限的，并且 $1 + 0.0001 = 1$ 在 FP16 中。这会在训练过程中出现问题。
- 当时 $update/param < 2^{-11}$ ，更新将无效。

您不能在任何地方都使用半精度，因为您几乎总是会遇到上述问题之一。相反，我们进行所谓的混合精度训练。这是您在训练的某些部分下降到 FP16 并恢复到 FP32 以保持其他部分的精度的地方。

我们在 FP16 中进行前向传递和后向传递，并且几乎在其他任何地方我们都使用 FP32。例如，当我们在权重更新中应用梯度时，我们使用全精度。在 FP32 中累积并存储在 FP16 中。

如果我们这样做，仍然存在一些问题：

1. 权重更新不精确。 $1 + 0.0001 = 1 \Rightarrow$ 消失的梯度。
2. 梯度可以下溢。数字变得太低，被 0 \Rightarrow 消失的梯度取代。
3. 激活、损失或减少可能会溢出 \Rightarrow 使 NaN，训练发散。

以下小节显示了如何解决这些问题。

https://www.cnblogs.com/h5l0/p/lib_hl_fixed-point.html

<https://blog.csdn.net/dreamer2020/article/details/24158303>

由于是规格化的浮点数，所以小数部分都要加上1，可以知道，单精度浮点数的小数部分最小是1.00000011920928955078125，其次是1.0000002384185791015625，注意到这两个小数之间是有间隔的，如果要表示1.0000001和1.0000002之间的数，则单精度浮点数无能为力，1.0000001已经是23位小数部分描述的最小值了。通过这样的分析可以发现，23位只能描述到小数点后第7位，即1.0000001, 1.0000002, 1.0000004, 1.0000009对应了二进制的小数值，其他要通过上面几个的组合来表示。

事实上，如果考虑第八位的舍入，1.0000004, 1.0000009本身的表示也是不准确的。为了验证这一点，通过程序将1.0000001到1.0000009的结果均打印出来，得如下：

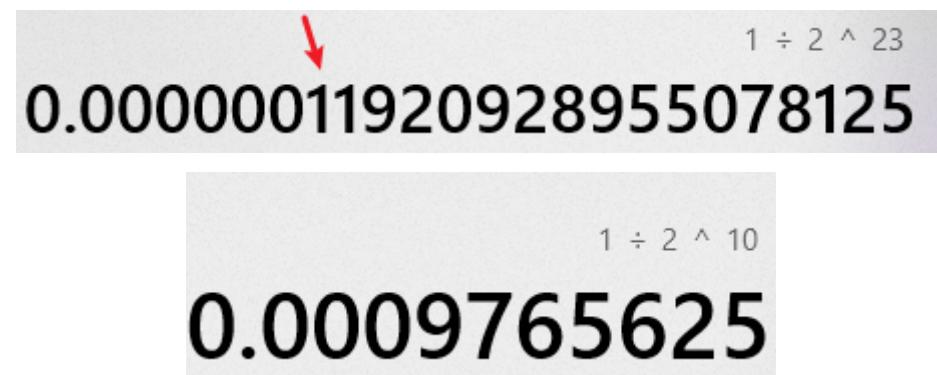
1.0000001	1.0000001192092895507812500
1.0000002	1.0000002384185791015625000
1.0000004	1.0000003576278686523437500
1.0000004	1.0000003576278686523437500 第7位是不准确的，四舍五入来的。只有第6位是准确的
1.0000005	1.0000004768371582031250000
1.0000006	1.0000005960464477539062500
1.0000007	1.0000007152557373046875000
1.0000008	1.0000008344650268554687500
1.0000010	1.0000009536743164062500000

程序中直接将1.0000001到1.0000009分别赋给float型变量，左边一列是输出时小数点后保留7位的结果，右边是保留了25位的结果。

通过上面的结果可以发现，小数点后第7位是部分准确的。例如，1.0000004就是1.00000035通过得到的，其实际保存和1.0000003相同。1.0000006也是通过舍入得到的。再往前第6位及以后均可以通过小数准确表示出来。通常说float数据的有效位是6~7位，也是这个原因。

类似的分析，双精度浮点数小数部分有52位，和上面类似，最低6位($2^{-52}, 2^{-51}, \dots$)表示的规格化小数如下所示。从图中可以看出，双精度浮点数能准确表示到小数点后第15位，第16位部分准确。

1.000000000000000222044604925031
1.000000000000000444089209850063
1.000000000000000888178419700125
1.000000000000001776356839400250
1.000000000000002552713678200501



- 所以FP32是6~7位有效小数位，而FP16只有3-4位有效小数位！

fp32为了运算精度，给小数部分分配了23位（可以说是非常重视精度），这样小数的分度值是 $1/2^{23}$ ，到小数点后6位的精度，而整数只有12位，除去符号位，可表示 $2^{11}=2048$ ，范围就是-2048~2047。

fp16的位长有效，给小数分配10位，也只有 $1/2^{10}=1/1024$ 也就是0.001的精度，而整数只剩可怜的5位，范围是-32~31。

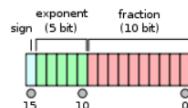
• 什么是半精度？

- 在神经网络中，所有的计算通常都是在单一精度下完成的，这意味着所有数组中代表输入、激活、权重的所有浮点数.....都是32位浮点数（本文其余部分为FP32）。一个减少内存使用（并避免那些烦人的cuda错误）的想法是尝试以半精度做同样的事情，这意味着使用16位浮点数（或本文其余部分中的FP16）。根据定义，它们占用RAM中一半的空间，理论上可以让模型的大小加倍，并将批量大小加倍。
- 另一个非常好的特性是NVIDIA开发了其最新的GPU（Volta系列）以充分利用半精度张量。基本上，如果你给它们半精度张量，它们就会堆叠起来，这样每个核就可以同时做更多的操作，并在理论上提供8倍的加速（遗憾的是，只是在理论上）。

因此，半精度训练更适合您的内存使用，如果您有 Volta GPU，速度会更快（如果没有，仍然会快一点，因为计算最简单）。我们该怎么做呢？在 pytorch 中非常容易，我们只需将 .half() 放在任何地方：在我们模型的输入和所有参数上。问题是您通常最终不会看到相同的准确度（因此有时会发生这种情况），因为半精度是……好吧……没有那么精确）。

半精度问题：

为了理解半精度的问题，让我们简要地看一下 FP16 的样子（更多信息[在这里](#)）。



符号位给我们 +1 或 -1，然后我们有 5 位来编码 -14 和 15 之间的指数，而小数部分有剩余的 10 位。与 FP32 相比，我们的可能值范围更小（大致为 2e-14 到 2e15，与 FP32 的 2e-126 到 2e127 相比），但偏移量也更小。

例如，在 1 和 2 之间，FP16 格式仅表示数字 1、1+2e-10、1+2^2e-10……这意味着 $1 + 0.0001 = 1$ 的半精度。这就是会导致一定数量问题的原因，特别是可能发生的三个问题并扰乱您的训练。

1. 权重更新不精确：在优化器中，您基本上对网络的每个权重执行 $w = w - lr \cdot w.grad$ 。以半精度执行此操作的问题在于，很多时候， $w.grad$ 比 w 低几个数量级，并且学习率也很小。因此， $lr=1$ 和 $lr \cdot w.grad$ 为 0.0001（或更低）的情况非常常见，但在这些情况下更新不会做任何事情。
2. 您的渐变可能会下溢。在 FP16 中，您的梯度很容易被 0 替换，因为它们太低了。
3. 您的激活或丢失可能会溢出。与梯度相反的问题：在 FP16 精度中更容易达到 nan（或无穷大），并且您的训练可能更容易发散。

解决方案：混合精度训练

为了解决这三个问题，我们没有完全训练 FP16 精度。正如名称混合训练所暗示的那样，一些操作将在 FP16 中完成，其他操作将在 FP32 中完成。这主要是为了解决上面列出的第一个问题，对于接下来的两个，还有额外的技巧。

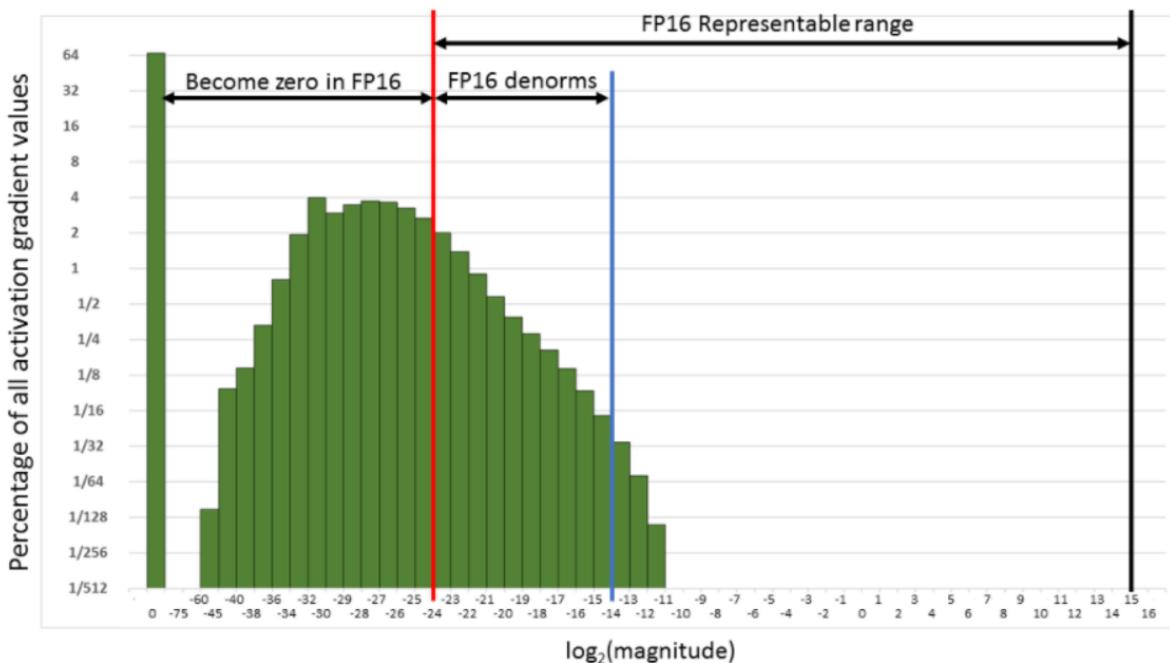
主要思想是我们希望以半精度（以加快速度）进行前向传递和梯度计算，但以单精度（更精确）进行更新。如果 w 和 $grad$ 都是半浮点数也没关系，但是当我们进行 $w = w - lr * grad$ 操作时，我们需要在 FP32 中计算它。这样我们的 $1 + 0.0001$ 就是 1.0001。

这就是我们在 FP32（称为主模型）中保留权重副本的原因。然后，我们的训练循环将如下所示：

1. 用 FP16 模型计算输出，然后损失
2. 以半精度反向传播梯度。
3. 以 FP32 精度复制梯度
4. 对主模型进行更新（以 FP32 精度）
5. 复制 FP16 模型中的主模型。

请注意，我们在第 5 步中失去了精度，并且其中一个权重中的 1.0001 将回到 1。但是如果下次更新对应再次添加 0.0001，由于优化器步骤是在主模型上完成的，1.0001 将变为 1.0002，如果我们最终像这样上升到 1.0005，FP16 模型将能够分辨出差异。

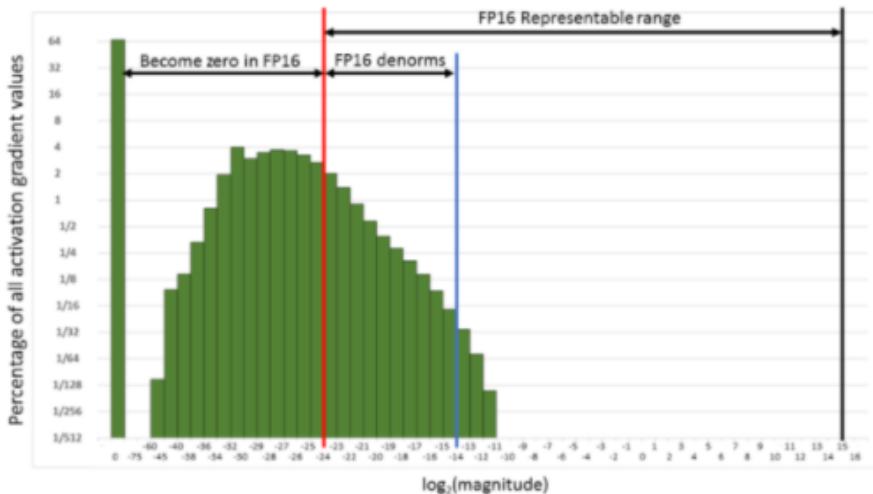
这解决了问题 1。对于第二个问题，我们使用称为梯度缩放的东西：为了避免梯度被 FP16 精度归零，我们将损失乘以一个比例因子（例如 scale=512）。这样我们就可以将下图中的梯度向右推，并使它们不为零。



损失缩放

接下来我们需要解决第二个问题——在 FP16 中进行反向传播时梯度可能会下溢。为了避免梯度被 FP16 精度归零，我们将损失乘以比例因子。通常，该系数类似于 512 或 128。

我们想要这样做是因为激活梯度值通常非常小，因此落在 FP16 的可表示范围之外。这是激活梯度大小的直方图：



我们要做的是将该分布向右推入 FP16 的可表示范围。我们可以通过将损失乘以 512 或 1024 来实现。

我们不希望这些 512 缩放的梯度出现在权重更新中，因此在将它们转换为 FP32 之后，我们需要通过除以缩放因子来“去缩放”。

训练循环变为：

1. 用 FP16 模型计算输出，然后是损失。
2. 按比例乘以损失，然后以半精度反向传播梯度。
3. 以 FP32 精度复制渐变，然后按比例划分。
4. 对主模型进行更新（以 FP32 精度）。
5. 复制 FP16 模型中的主模型。

当然我们不希望那些 512-scaled 的梯度在权重更新中，所以在将它们转换成 FP32 之后，我们可以将它们除以这个比例因子（一旦它们没有变成 0 的风险）。这将循环更改为：

1. 用 FP16 模型计算输出，然后是损失。
2. 将损失乘以比例，然后以半精度反向传播梯度。
3. 以 FP32 精度复制梯度，然后按比例划分。
4. 对主模型进行更新（以 FP32 精度）。
5. 复制 FP16 模型中的主模型。

对于最后一个问题，NVIDIA 提供的技巧是将 batchnorm 层保留为单精度（它们没有很多权重，因此这不是一个很大的内存挑战）并计算单精度的损失（这意味着转换在将模型传递给损失之前，以单精度形式进行模型）。

累积到 FP32

最后一个问题是——激活、损失或减少可能溢出——需要在几个地方处理。

首先，损失可能会溢出，所以让我们进行减少计算，给出 FP32 中的损失：

```
y_pred = model(x) # y_pred: fp16
loss = F.mse_loss(y_pred.float(), y.float()) # Loss is now FP32
scaled_loss = scale_factor * loss
```

另一个溢出风险是 Batchnorm，它也应该减少 FP32。您可以递归地遍历模型并将所有 Batchnorm 层更改回 FP32 使用此函数：

```
bn_types = (nn.BatchNorm1d, nn.BatchNorm2d, nn.BatchNorm3d)

def bn_to_float(model):
    if isinstance(model, bn_types): model.float()
    for child in model.children(): bn_to_float(child)
    return model
```

然后，您可以使用此函数将 Pytorch 模型转换为半精度：

```
def model_to_half(model):
    model = model.half()
    return bn_to_float(model)
```

动态损失缩放

损失缩放的问题在于它有一个神奇的数字，`scale_factor`您必须对其进行调整。随着模型训练，可能需要不同的值。动态损失缩放是一种`scale_factor`在运行时自适应地将设置为正确值的技术。这个值将完美地适合我们的模型，并且可以随着训练的进行继续动态调整，如果它仍然太高，每次我们溢出时只需将其减半。不过过了一段时间，训练会收敛，梯度会开始变小，所以我们还需要一种机制，在安全的情况下，让这个动态损失规模变大。

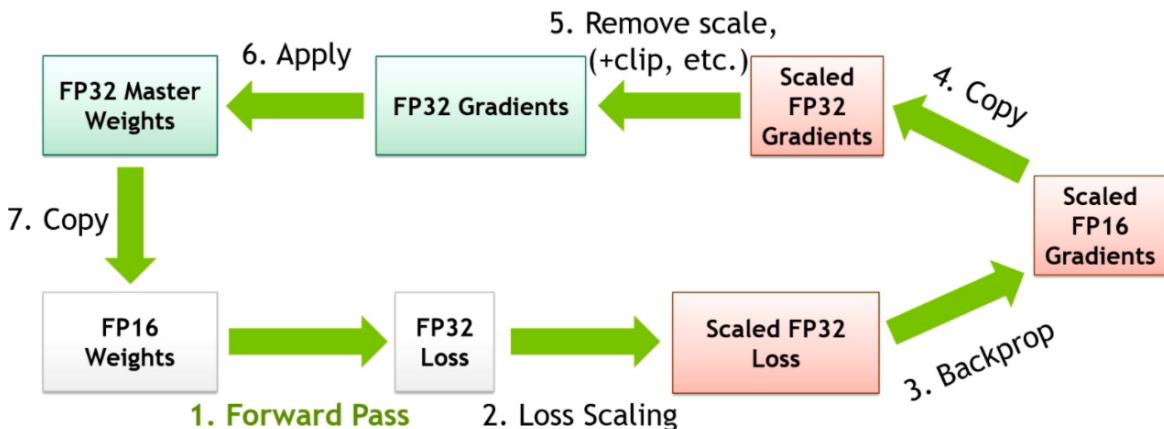
算法：

1. 首先`scale_factor`用一个非常高的值初始化，例如 512。
2. 做一个向前和向后的传球。
3. 检查是否有任何梯度溢出。
4. 如果任何梯度溢出，则将减半并将`scale_factor`梯度归零（从而跳过优化步骤）。
5. 如果循环进行 500 步而没有溢出，则将`scale_factor`。

我们如何测试溢出？NaN 的一个有用属性是它们可以传播 - 将任何内容添加到 NaN，如果是 NaN，则结果。因此，如果我们对包含 NaN 的张量求和，结果将是 NaN。要检查它是否为 NaN，我们可以使用违反直觉的属性，`NaN != NaN`只需检查总和的结果是否等于自身。这是代码：

```
def test_overflow(x):
    s = float(x.float().sum())
    return (s == float('inf') or s == float('-inf') or s != s)
```

MIXED PRECISION TRAINING



(来源: [NVIDIA - 使用Tensor Cores 进行深度学习的混合精度训练技术](#))

总之，以下是使用 FP16 转换模型引起的 3 个问题以及如何缓解它们：

1. 权重更新不精确 => FP32 中的“主”权重
2. 梯度可能下溢 => 损失 (梯度) 缩放
3. 激活或丢失可能会溢出 => 在 FP32 中累积

有时半精度训练会得到更好的结果。更多的随机性，一些正则化。通常结果与使用 FP32 获得的结果相似，但速度更快。

Util functions

在进入主回调之前，我们需要一些辅助函数。我们将使用APEX库 [APEX library](#) util函数进行重构。如果你不能安装CUDA/ c++，那么python版本就足够了。

我们需要一个函数将模型的所有层转换为FP16精度，除了类似batchnorm的层(因为这些需要在FP32精度中完成以保持稳定)。我们分两步完成:首先将模型转换为FP16，然后循环所有层，如果它们是BatchNorm层，则将它们放回FP32。

在Apex中，为我们完成这一任务的函数是convert_network。我们可以使用它将模型放入FP16或返回FP32。

```
1 # export
2 import apex.fp16_utils as fp16
3
4 bn_types = (nn.BatchNorm1d, nn.BatchNorm2d, nn.BatchNorm3d)
5 def bn_to_float(model):
6     if isinstance(model, bn_types): model.float()
7     for child in model.children(): bn_to_float(child)
8     return model
9 def model_to_half(model):
10    model = model.half()
11    return bn_to_float(model)
12
13 model = nn.Sequential(nn.Linear(10, 30), nn.BatchNorm1d(30),
14                      nn.Linear(30, 2)).cuda()
15 model = model_to_half(model)
16 def check_weights(model):
17     for i,t in enumerate([torch.float16, torch.float32, torch.float16]):
18         assert model[i].weight.dtype == t
19         assert model[i].bias.dtype == t
```

```

19 check_weights(model)
20
21 model = nn.Sequential(nn.Linear(10, 30), nn.BatchNorm1d(30),
22 nn.Linear(30, 2)).cuda()
23 model = fp16.convert_network(model, torch.float16)
check_weights(model)

```

Creating the master copy of the parameters

为了解决上面列出的第一个问题——权重更新不精确——我们可以在 FP32 中存储权重的“主”副本。这是传递给优化器的：

```
opt = torch.optim.SGD(master_params, lr=1e-3)
```

在优化器步骤之后，您将主权重复制回 FP16 中的模型权重：

```
master.grad.data.copy_(model.grad.data)
```

然后，我们的训练循环将如下所示：

1. 用FP16模型计算输出，然后损失
2. 以半精度反向传播梯度。
3. 以 FP32 精度复制grad
4. 对主模型进行更新（以 FP32 精度）
5. 复制 FP16 模型中的主模型。

- 从我们的模型参数(主要在FP16中)，我们将希望在FP32中创建一个副本(主参数)，我们将用于优化器中的步骤。可选地，我们连接所有的参数来做一个平坦的大张量，这可以让这个step快一点。

```

1 from torch.nn.utils import parameters_to_vector
2
3 def get_master(model, flat_master=False):
4     model_params = [param for param in model.parameters() if
5         param.requires_grad]
6     if flat_master:
7         master_param = parameters_to_vector([param.data.float() for param in
8             model_params])
9         master_param = torch.nn.Parameter(master_param, requires_grad=True)
10        if master_param.grad is None: master_param.grad =
11            master_param.new(*master_param.size())
12        return model_params, [master_param]
13    else:
14        master_params = [param.clone().float().detach() for param in
15            model_params]
16        for param in master_params: param.requires_grad_(True)
17        return model_params, master_params

```

The util function from Apex to do this is `prep_param_lists`.

```

1 model_p, master_p = get_master(model)
2 model_p1, master_p1 = fp16.prep_param_lists(model)
3
4 def same_lists(ps1, ps2):
5     assert len(ps1) == len(ps2)
6     for (p1,p2) in zip(ps1,ps2):
7         assert p1.requires_grad == p2.requires_grad
8         assert torch.allclose(p1.data.float(), p2.data.float())
9
10 same_lists(model_p,model_p1)
11 same_lists(model_p, master_p)
12 same_lists(master_p, master_p1)
13 same_lists(model_p1, master_p1)

```

We can't use flat_master when there is a mix of FP32 and FP16 parameters (like batchnorm here).

```

1 model1 = nn.Sequential(nn.Linear(10,30), nn.Linear(30,2)).cuda()
2 model1 = fp16.convert_network(model1, torch.float16)
3 model_p, master_p = get_master(model1, flat_master=True)
4 model_p1, master_p1 = fp16.prep_param_lists(model1, flat_master=True)
5 same_lists(model_p, model_p1)
6 same_lists(master_p, master_p1)
7 assert len(master_p[0]) == 10*30 + 30 + 30*2 + 2
8 assert len(master_p1[0]) == 10*30 + 30 + 30*2 + 2

```

```

1 def get_master(opt, flat_master=False):
2     model_params = [[param for param in pg if param.requires_grad] for pg in
3 opt.param_groups]
4     if flat_master:
5         master_params = []
6         for pg in model_params:
7             mp = parameters_to_vector([param.data.float() for param in pg])
8             mp = torch.nn.Parameter(mp, requires_grad=True)
9             if mp.grad is None: mp.grad = mp.new(*mp.size())
10            master_params.append(mp)
11    else:
12        master_params = [[param.clone().float().detach() for param in pg]
13 for pg in model_params]
14        for pg in master_params:
15            for param in pg: param.requires_grad_(True)
16    return model_params, master_params

```

Copy the gradients from model params to master params

在反向传播之后，必须将所有梯度复制到master params中，然后才能在FP32中执行优化器步骤optimizer step。

```

1 def to_master_grads(model_params, master_params, flat_master=False)-
>None:
2     if flat_master:
3         if master_params[0].grad is None: master_params[0].grad =
master_params[0].data.new(*master_params[0].data.size())
4
5         master_params[0].grad.data.copy_(parameters_to_vector([p.grad.data.float()
for p in model_params]))
6     else:
7         for model, master in zip(model_params, master_params):
8             if model.grad is not None:
9                 if master.grad is None: master.grad =
master.data.new(*master.data.size())
10                master.grad.data.copy_(model.grad.data)
11            else: master.grad = None

```

The corresponding function in the Apex utils is `model_grads_to_master_grads`.

```

1 x = torch.randn(20,10).half().cuda()
2 z = model(x)
3 loss = F.cross_entropy(z, torch.randint(0, 2, (20,)).cuda())
4 loss.backward()
5 to_master_grads(model_p, master_p)
6
7 def check_grads(m1, m2):
8     for p1,p2 in zip(m1,m2):
9         if p1.grad is None: assert p2.grad is None
10        else: assert torch.allclose(p1.grad.data, p2.grad.data)
11
12 check_grads(model_p, master_p)
13
14 fp16.model_grads_to_master_grads(model_p, master_p)
15
16 check_grads(model_p, master_p)

```

Copy the master params to the model params

After the step, we need to copy back the master parameters to the model parameters for the next update.

```

1 from torch._utils import _unflatten_dense_tensors
2
3 def to_model_params(model_params, master_params, flat_master=False)-
>None:
4     if flat_master:
5         for model, master in zip(model_params,
6             _unflatten_dense_tensors(master_params[0].data, model_params)):
7                 model.data.copy_(master)
8             else:
9                 for model, master in zip(model_params, master_params):
10                     model.data.copy_(master.data)

```

The corresponding function in Apex is `master_params_to_model_params`.

But we need to handle param groups 我们要处理参数组

问题是，我们并不总是希望模型中的所有参数都在同一个参数组中，因为我们可能：

- 要做迁移学习，冻结几层
- 使用有差异的学习率 discriminative learning rates
- 不要将weight decay应用于某些层(如BatchNorm)或偏差bias term

因此，我们实际上需要一个函数，根据正确的参数组来分割优化器(而不是模型)的参数。下面的函数需要处理参数列表的列表(在model_pgs和master_pgs中每个参数组的列表)

```
1 # export
2 def get_master(opt, flat_master=False):
3     model_pgs = [[param for param in pg if param.requires_grad] for pg in
4 opt.param_groups]
5     if flat_master:
6         master_pgs = []
7         for pg in model_pgs:
8             mp = parameters_to_vector([param.data.float() for param in pg])
9             mp = torch.nn.Parameter(mp, requires_grad=True)
10            if mp.grad is None: mp.grad = mp.new(*mp.size())
11            master_pgs.append([mp])
12    else:
13        master_pgs = [[param.clone().float().detach() for param in pg] for
14 pg in model_pgs]
15        for pg in master_pgs:
16            for param in pg: param.requires_grad_(True)
17    return model_pgs, master_pgs
18 # export
19 def to_master_grads(model_pgs, master_pgs, flat_master:bool=False)->None:
20     for (model_params,master_params) in zip(model_pgs,master_pgs):
21         fp16.model_grads_to_master_grads(model_params, master_params,
22 flat_master=flat_master)
23 # export
24 def to_model_params(model_pgs, master_pgs, flat_master:bool=False)->None:
25     for (model_params,master_params) in zip(model_pgs,master_pgs):
26         fp16.master_params_to_model_params(model_params, master_params,
27 flat_master=flat_master)
```

The main Callback

```
1 class MixedPrecision(callback):
2     _order = 99
3     def __init__(self, loss_scale=512, flat_master=False):
4         assert torch.backends.cudnn.enabled, "Mixed precision training
5 requires cudnn."
6         self.loss_scale, self.flat_master = loss_scale, flat_master
7
8     def begin_fit(self):
9         self.run.model = fp16.convert_network(self.model,
10 dtype=torch.float16)
11         self.model_pgs, self.master_pgs = get_master(self.opt,
12 self.flat_master)
13         #Changes the optimizer so that the optimization step is done in
14 FP32.
```

```

11         self.run.opt.param_groups = self.master_pgs #Put those param groups
12         inside our runner.
13
14     def after_fit(self): self.model.float()
15
16     def begin_batch(self): self.run.xb = self.run.xb.half() #Put the inputs
17         to half precision
18     def after_pred(self): self.run.pred = self.run.pred.float() #Compute
19         the loss in FP32
20     def after_loss(self): self.run.loss *= self.loss_scale #Loss scaling to
21         avoid gradient underflow
22
23     def after_backward(self):
24         #Copy the gradients to master and unscale
25         to_master_grads(self.model_pgs, self.master_pgs, self.flat_master)
26         for master_params in self.master_pgs:
27             for param in master_params:
28                 if param.grad is not None: param.grad.div_(self.loss_scale)
29
30     def after_step(self):
31         #Zero the gradients of the model since the optimizer is
32         disconnected.
33         self.model.zero_grad()
34         #Update the params from master to model.
35         to_model_params(self.model_pgs, self.master_pgs, self.flat_master)

```

Now let's test this on Imagenette

```

1 path = datasets untar_data(datasets.URLS.IMAGENETTE_160)
2 tfms = [make_rgb, ResizeFixed(128), to_byte_tensor, to_float_tensor]
3 bs = 64
4
5 il = ImageList.from_files(path, tfms=tfms)
6 sd = SplitData.split_by_func(il, partial(grandparent_splitter,
7     valid_name='val'))
8 ll = label_by_func(sd, parent_labeler, proc_y=categoryProcessor())
9 data = ll.to_databunch(bs, c_in=3, c_out=10, num_workers=4)
10
11 nfs = [32,64,128,256,512]
12 def get_learner(nfs, data, lr, layer, loss_func=F.cross_entropy,
13     cb_funcs=None, opt_func=adam_opt(), **kwargs):
14     model = get_cnn_model(data, nfs, layer, **kwargs)
15     init_cnn(model)
16     return Learner(model, data, loss_func, lr=lr, cb_funcs=cb_funcs,
17     opt_func=opt_func)

```

Training without mixed precision

```

1 cbfs = [partial(AvgStatsCallback,accuracy),
2         ProgressCallback,
3         CudaCallback,
4         partial(BatchTransformxCallback, norm_imagenette)]
5 learn = get_learner(nfs, data, 1e-2, conv_layer, cb_funcs=cbfs)
6 learn.fit(1)

```

Training with mixed precision

```

1 cbfs = [partial(AvgStatsCallback, accuracy),
2         CudaCallback,
3         ProgressCallback,
4         partial(BatchTransformxCallback, norm_imagenette),
5         MixedPrecision]
6 learn = get_learner(nfs, data, 1e-2, conv_layer, cb_funcs=cbfs)
7 learn.fit(1)
8 test_eq(next(learn.model.parameters()).type(), 'torch.cuda.FloatTensor')

```

Dynamic loss scaling

- 512对训练会产生很大的影响。所以用动态loss 缩放，动态损失缩放，实际上尝试了几个不同的损失缩放值，以找出它在什么时候变为无穷大inf，因此它动态地计算出我们可以达到的最高丢失错误。
- 有时候半精度比FP32训练，能得到更好的结果。我不知道，也许更多的随机性也许它有点正则化但通常它超级超级相似只是更快。
- Question: mixup，我认为混合真正好的一件事是它不需要任何特定领域的思考就像我们水平翻转一样，你也可以判断我们可以旋转多少它不会产生任何损失，就像在角落里没有反射填充或黑色填充一样，所以它有点像非常漂亮和干净。
- 它也几乎是无限的就它可以创建的不同图像的数量而言，因此您可以将每个图像与其他已经很大的图像进行这种排列，然后再进行不同的混合，所以它只是你可以用它做很多增强，还有其他类似的东西，所以还有一种叫做 cutout 的东西，你只需删除正方形并用黑色替换，它还有另一个你删除正方形并用随机像素替换它的東西，我还没有看到但我真的很想看到人们做的是删除一个正方形并用不同的图像替换它。所以我希望有人尝试混合而不是平均线性组合而不是选择一个 alpha .
- 损失缩放的问题在于它有一个神奇的数字，scale_factor 您必须对其进行调整。随着模型训练，可能需要不同的值。动态损失缩放是一种 scale_factor 在运行时自适应地将设置为正确值的技术。这个值将完美地适合我们的模型，并且可以随着训练的进行继续动态调整，如果它仍然太高，每次我们溢出时只需将其减半。不过过了一段时间，训练会收敛，梯度会开始变小，所以我们还需要一种机制，在安全的情况下，让这个动态损失规模变大。

在之前的混合精度训练的实现中，唯一令人恼火的是，它引入了一个新的超参数来进行调优，即损失缩放值 loss scaling。幸运的是，我们有办法解决这个问题。我们希望损失比例尽可能高，以便我们的梯度可以使用整个范围的表示，所以让我们首先尝试一个真正高的值。很有可能，这将导致我们的梯度或损失溢出，我们将再次尝试这个大值的一半，直到我们达到最大的损失规模，使梯度不溢出。

算法：

1. 首先 scale_factor 用一个非常高的值初始化，例如 512。
2. 做一个向前和向后的传球。
3. 检查是否有任何梯度溢出。
4. 如果任何梯度溢出，则将 减半并将 scale_factor 梯度归零（从而跳过优化步骤）。
5. 如果循环进行了500步没有溢出，则将scale_factor加倍。.

我们如何测试溢出？NaN 的一个有用属性是它们可以传播 - 向NaN添加任何东西，如果NaN会得到结果。因此，如果我们对包含 Nan 的张量求和，结果将是 NaN。要检查它是否为 NaN，我们可以使用违反直觉的属性，NaN!=NaN 只需检查总和的结果是否等于自身。这是代码：

```

1 def test_overflow(x):
2     s = float(x.float().sum())
3     return (s == float('inf') or s == float('-inf') or s != s)

```

这个值将完美地适合我们的模型，并且可以随着训练的进行继续进行动态调整，如果它仍然太高，每次我们溢出时只需将它减半。一段时间后，训练会收敛，梯度会开始变小，所以我们还需要一种机制，让这个动态损失规模变大，如果这样做是安全的。Apex库中使用的策略是，每次我们在没有溢出的情况下进行给定数量的迭代时，将损失规模乘以2。

为了检查梯度是否溢出，我们检查它们的总和(在FP32中计算)。如果一项是nan，那么和就是nan。有趣的是，在GPU上，它比检查torch.isnan快：

```
1 # export
2 def test_overflow(x):
3     s = float(x.float().sum())
4     return (s == float('inf') or s == float('-inf') or s != s)
5
6 x = torch.randn(512,1024).cuda()
7 test_overflow(x)
8 x[123,145] = float('inf')
9 test_overflow(x)
10 %timeit test_overflow(x)
11 %timeit torch.isnan(x).any().item()
```

So we can use it in the following function that checks for gradient overflow:

```
1 # export
2 def grad_overflow(param_groups):
3     for group in param_groups:
4         for p in group:
5             if p.grad is not None:
6                 s = float(p.grad.data.float().sum())
7                 if s == float('inf') or s == float('-inf') or s != s: return
8 True
9 return False
```

And now we can write a new version of the `callback` that handles dynamic loss scaling.

```
1 # export
2 class MixedPrecision(callback):
3     _order = 99
4     def __init__(self, loss_scale=512, flat_master=False, dynamic=True,
5                  max_loss_scale=2.**24, div_factor=2.,
6                  scale_wait=500):
7         assert torch.backends.cudnn.enabled, "Mixed precision training
8 requires cudnn."
9         self.flat_master, self.dynamic, self.max_loss_scale =
10        flat_master, dynamic, max_loss_scale
11         self.div_factor, self.scale_wait = div_factor, scale_wait
12         self.loss_scale = max_loss_scale if dynamic else loss_scale
13
14     def begin_fit(self):
15         self.run.model = fp16.convert_network(self.model,
16                                             dtype=torch.float16)
17         self.model_pgs, self.master_pgs = get_master(self.opt,
18                                           self.flat_master)
19         #Changes the optimizer so that the optimization step is done in
20         #FP32.
21         self.run.opt.param_groups = self.master_pgs #Put those param groups
22         inside our runner.
```

```

16     if self.dynamic: self.count = 0
17
18     def begin_batch(self): self.run.xb = self.run.xb.half() #Put the inputs
19         to half precision
20         def after_pred(self): self.run.pred = self.run.pred.float() #Compute
21             the loss in FP32
22         def after_loss(self):
23             if self.in_train: self.run.loss *= self.loss_scale #Loss scaling to
24                 avoid gradient underflow
25
26         def after_backward(self):
27             #First, check for an overflow
28             if self.dynamic and grad_overflow(self.model_pgs):
29                 #Divide the loss scale by div_factor, zero the grad (after_step
30                 will be skipped)
31                 self.loss_scale /= self.div_factor
32                 self.model.zero_grad()
33                 return True #skip step and zero_grad
34             #Copy the gradients to master and unscale
35             to_master_grads(self.model_pgs, self.master_pgs, self.flat_master)
36             for master_params in self.master_pgs:
37                 for param in master_params:
38                     if param.grad is not None: param.grad.div_(self.loss_scale)
39             #Check if it's been long enough without overflow
40             if self.dynamic:
41                 self.count += 1
42                 if self.count == self.scale_wait:
43                     self.count = 0
44                     self.loss_scale *= self.div_factor
45
46         def after_step(self):
47             #zero the gradients of the model since the optimizer is
48             disconnected.
49             self.model.zero_grad()
50             #Update the params from master to model.
51             to_model_params(self.model_pgs, self.master_pgs, self.flat_master)

```

```

1 cbfs = [partial(AvgStatsCallback,accuracy),
2         Cudacallback,
3         ProgressCallback,
4         partial(BatchTransformxCallback, norm_imagenette),
5         MixedPrecision]

```

```

1 learn = get_learner(nfs, data, 1e-2, conv_layer, cb_funcs=cbfs)
2 learn.fit(1)

```

The loss scale used is way higher than our previous number:

```

1 | learn.cbs[-1].loss_scale

```

Imagenet(te) training

`11_train_imagenette.ipynb`

```

1 #export
2 from exp.nb_10c import *

1 path = datasets untar_data(datasets.URLS.IMAGENETTE_160)
2 size = 128
3 # 随机调整裁剪, 最小比例的0.35, 最大为1.发现效果很好。
4 tfms = [make_rgb, RandomResizedCrop(size, scale=(0.35,1)), np_to_float,
PILRandomFlip()]
5
6 bs = 64
7
8 il = ImageList.from_files(path, tfms=tfms)
9 sd = splitData.split_by_func(il, partial(grandparent_splitter,
valid_name='val'))
10 ll = label_by_func(sd, parent_labeler, proc_y=categoryProcessor())
11 ll.valid.x.tfms = [make_rgb, CenterCrop(size), np_to_float]
12
13 data = ll.to_databunch(bs, c_in=3, c_out=10, num_workers=8)

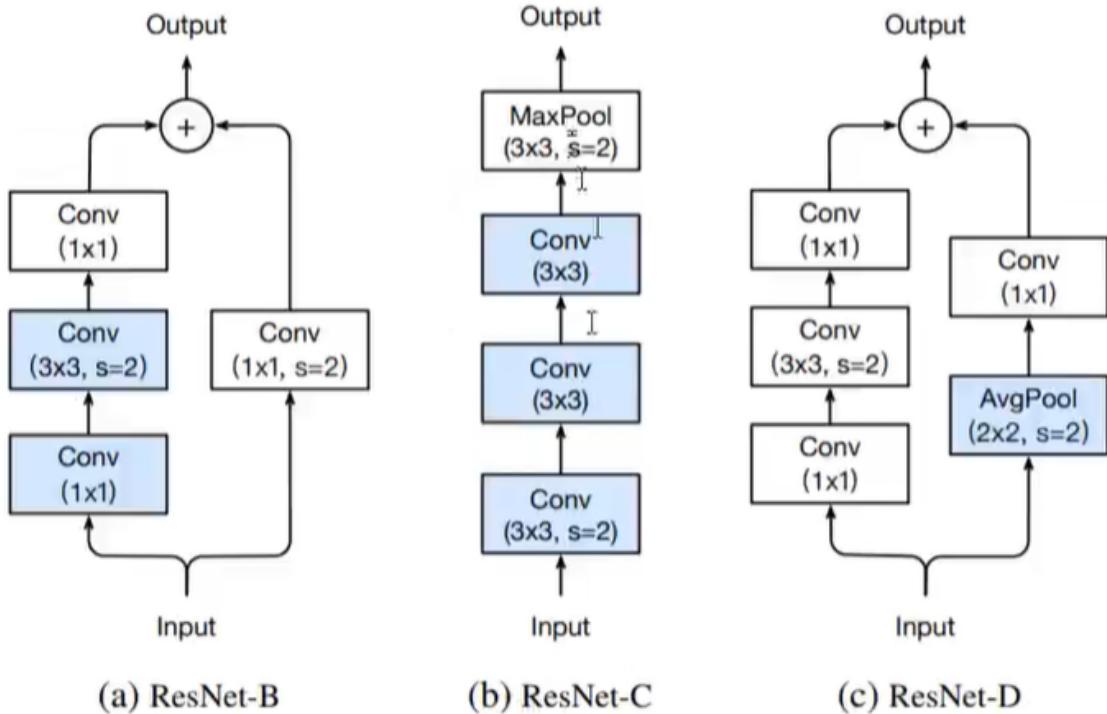
```

XResNet

- 到目前为止, 我们使用的所有图像模型都是无聊的卷积模型。我们真正想要使用的是**ResNet 模型**。我们将实现**XResNet**, 它是 ResNet 的突变/扩展版本。这是从[技巧袋论文中](#)提取的经过调整的 ResNet。

让我们来看看 XResNet 的修改.....

- 第一个修改: **ResNetC** - 一开始不要做大的 7×7 卷积, 因为它效率低下并且只是一个单一的线性模型。而是连续进行三个 3×3 转换。感受野仍然是大约 7×7 , 但它可以学习的东西要丰富得多, 因为它有 3 层而不是 1 层。我们称这些第一层为**stem**。 (这在[第 11 课](#)中也有介绍)
 - Conv 层接收多个输入通道 `c_in` 并输出多个输出通道 `c_out`。
 - 默认情况下第一层有, `c_in=3` 因为通常我们有 RGB 图像。
 - 我们将输出数量设置为 `c_out=(c_in+1)*8`。这为第二层提供了 32 个通道的输入, 这是技巧包论文所推荐的。
 - 因子 8 还有助于更有效地使用 GPU 架构。这会随着输入通道的数量而自行增长/缩小, 因此如果您有更多的输入, 那么它就会有更多的激活。如果不是 8 的倍数, nvidia 显卡可能会很慢。



前几层称为 the `stem`，它看起来像：下面的stem

```

1 nfs = [c_in, (c_out+1)*8, 64, 64] # c_in/c_outs for the 3 conv layers
2 stem = [conv_layer(nfs[i], nfs[i+1], stride=2 if i==0 else 1) for i in
range(3)]

```

`conv_layer` 是一个 `nn.Sequential` 对象：

- 一个卷积
- 后跟一个 `BatchNorm`
- 和可选的激活（默认 `ReLU`）

```

1 act_fn = nn.ReLU(inplace=True)
2
3 def conv(ni, nf, ks=3, stride=1, bias=False):
4     return nn.Conv2d(ni, nf, kernel_size=ks, stride=stride, padding=ks//2,
bias=bias)
5
6 def conv_layer(ni, nf, ks=3, stride=1, zero_bn=False, act=True):
7     bn = nn.BatchNorm2d(nf)
8     nn.init.constant_(bn.weight, 0. if zero_bn else 1.) # init batchnorm
trick
9     layers = [conv(ni, nf, ks, stride=stride), bn]
10    if act: layers.append(act_fn)
11    return nn.Sequential(*layers)

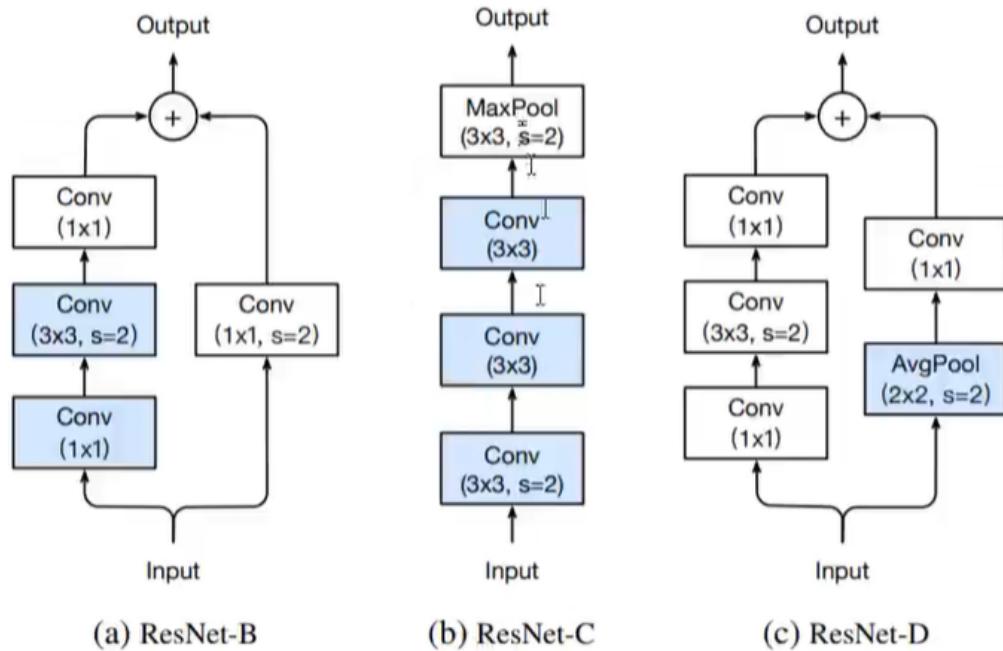
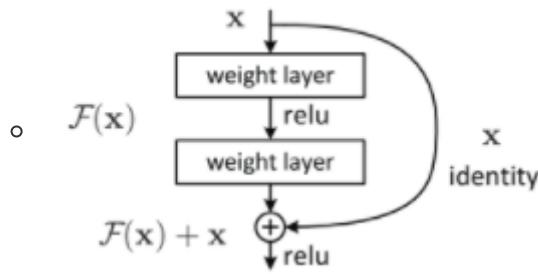
```

Zero BatchNorm Trick

- 在 `stem` 之后，ResNet 主体的其余部分是任意数量的 `resblock`。在 `ResBlock` 中，`BatchNorm` 初始化还有一个额外的技巧。我们有时将 `BatchNorm` 权重初始化为 0，有时将它初始化为 1。

why? 要知道为什么这是有用的，回想一下标准 `ResBlock` 的图表：

- `resnet-D` 是一个标准的 `resnet-block` 残差块。

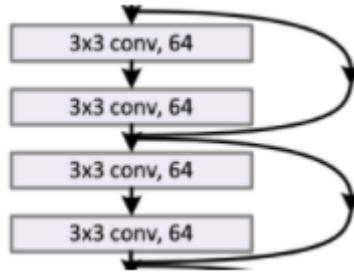


- 上面的每个“权重层”都是一个Conv/BatchNorm。如果ResBlock的输入是 x , 那么它的输出是 $x+block(x)$ 。如果我们将block中的最后一个BatchNorm层初始化为0, 那么这就相当于将输入乘以0, 因此 $block(x)=0$ 。因此, 在训练开始时, 所有的resblock只是返回它们的输入, 这模拟了一个更少层的网络, **在初始阶段更容易训练**。
 - 如果你有一个1000层的深的模型, 任何层的稍微的变化都会导致梯度以这种螺旋式下降到零阶无穷大。我们将三层的最后一个bn的权重设置为0, 让我们可以在高学习率下训练非常深的模型。
- 像google、Facebook这样的大公司喜欢炫耀他们的巨型数据中心, 如果有1000个CPU, 可以创建很大的batch size。但对普通人来说, 用了resnet我们也可以使用很大的学习率, 也可以让我们创建大的batch size。
- 使用更高的学习率, 可以训练更快, 泛化能力更好。

Resblock

- 在stem之后, ResNet主体的其余部分是任意数量的resblock。ResBlock代码如下:
- 在一个ResNet中有几种不同类型的ResBlock, 这些都包含在上面完整的ResBlock代码中, 参数的扩展和stride都可以改变。

在ResNet18/30中, 有一个标准的ResBlock, 它的扩展值expansion为1,stride值为1, 它们堆叠在一起, 如图所示:



- 除了这个标准之外，还有另外两个ResBlock——扩展(也就是瓶颈)ResBlock和降采样ResBlock。the *Expansion* (AKA Bottleneck) ResBlock, and the *Downsampling* ResBlock. 让我们来看看它们是如何通过《戏法之袋》的论文进行调整的。

```

1 #export
2 def noop(x): return x
3
4 class Flatten(nn.Module):
5     def forward(self, x): return x.view(x.size(0), -1)
6
7 def conv(ni, nf, ks=3, stride=1, bias=False):
8     return nn.Conv2d(ni, nf, kernel_size=ks, stride=stride, padding=ks//2,
9                     bias=bias)
10 #export
11 act_fn = nn.ReLU(inplace=True)
12
13 def init_cnn(m):
14     if getattr(m, 'bias', None) is not None: nn.init.constant_(m.bias, 0)
15     if isinstance(m, (nn.Conv2d, nn.Linear)):
16         nn.init.kaiming_normal_(m.weight)
17         for l in m.children(): init_cnn(l)
18
19 def conv_layer(ni, nf, ks=3, stride=1, zero_bn=False, act=True):
20     bn = nn.BatchNorm2d(nf)
21     # 有时初始化bn的权重为1, 有时初始化为1
22     nn.init.constant_(bn.weight, 0. if zero_bn else 1.)
23     layers = [conv(ni, nf, ks, stride=stride), bn]
24     if act: layers.append(act_fn)
25     return nn.Sequential(*layers)
26
27 #export
28 class ResBlock(nn.Module):
29     def __init__(self, expansion, ni, nh, stride=1):
30         super().__init__()
31         nf, ni = nh*expansion, ni*expansion
32         layers = [conv_layer(ni, nh, 3, stride=stride),
33                   conv_layer(nh, nf, 3, zero_bn=True, act=False)]
34         if expansion == 1 else [
35             conv_layer(ni, nh, 1),
36             conv_layer(nh, nh, 3, stride=stride),
37             conv_layer(nh, nf, 1, zero_bn=True, act=False)]
38         self.convs = nn.Sequential(*layers)
39         self.idconv = noop if ni==nf else conv_layer(ni, nf, 1, act=False)
40         self.pool = noop if stride==1 else nn.AvgPool2d(2, ceil_mode=True)
41
42     def forward(self, x): return act_fn(self.convs(x)) +
43         self.idconv(self.pool(x)))

```

Expansion/BottleNeck ResBlock

- 对于ResNet18/34, ResBlock看起来像下面左边的图表——一个张量带有shape[*, *, 64], 并经历两个3x3 conv_layers。然而, 对于更深层次的ResNets(例如50+), 做所有这些3x3的conv_layers是昂贵的和消耗内存的。
- 相反, 我们使用一个 *BottleNeck* 具有1x1卷积的 *BottleNeck*, 将通道数量压缩4个, 然后我们进行一个3x3卷积, 然后再执行另一个1x1, 将其投影回原来的形状。由于我们在3x3的conv_layer中将通道数压缩到原来的4倍, 因此我们将模型中的正常通道数扩展到原来的4倍, 以得到相当于基本块大小的卷积。如下图所示:

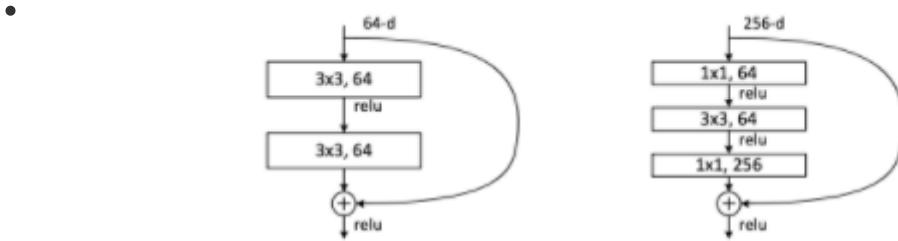


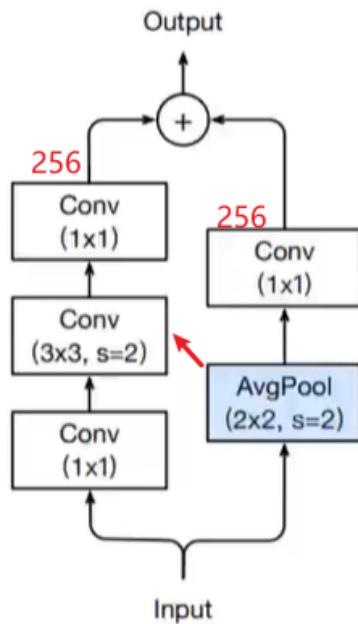
Figure 5. A deeper residual function \mathcal{F} for ImageNet. Left: a building block (on 56×56 feature maps) as in Fig. 3 for ResNet-34. Right: a “bottleneck” building block for ResNet-50/101/152.

- (图表摘自原ResNet论文)

在ResBlock代码中, 这个BottleNeck 层是通过expansion参数实现的。expansion是1或4。我们将输入和输出通道的数量乘以这个因子: `nf, ni = nh*expansion, ni*expansion.`。对于ResNet50+, 这个因子是4。

Downsampling ResBlock

- 在一组新的 ResBlocks 开始时, 我们通常将空间维度减半, stride=2 的卷积, 同时通道数量加倍。维度现在已经改变, 那么identity connection连接会发生什么? 在最初的论文中, 他们使用投影矩阵来减少维数, 我看到的其他实现使用1x1 conv_layer和stride 2。
 - 由于resnet block中, 残差块中有一个 3×3 的Conv, stride=2, 下采样了。所以 identity的加法时featuremap是不一样的, 所以需要一个AvgPool进行特征图减半处理。才能做加法。
 - 同时通道数也变化了, 3层resblock的最后一个将通道数变为4倍了。identity连接也会出问题, 所以在avgpool之后再加入了一个 1×1 卷积层, 改变通道数。
- 他们在the bag of tricks paper 的做法是, 将 AveragePooling 步长 2 设为网格大小的一半, 然后使用 1×1 conv_layer (步长 1) 来增加通道数。这是下采样 ResBlock 的示意图:



(c) ResNet-D

- 进一步的调整，如上所示，是将步幅2放到 3×3 的conv_layer中。在此之前，人们在第一个 1×1 conv_layer中大步2，这是一件可怕的事情，因为你只是丢弃了 $3/4$ 的数据。
- 有趣的是人们花了数年时间才意识到他们实际上丢弃了四分之三的数据**
- 所以我提到这些细节的原因是，为了你可以阅读那篇论文并花时间思考每个ResNet调整。你是否理解他们为什么这样做是正确的？一些神经架构搜索尝试一切无脑使用我们所有的计算机方法，它是单独的让我们坐下来思考**我们如何实际使用我们拥有的所有输入**，以及我们如何实际利用我们正在做的所有计算。所以它是我的意思是之前存在的大部分调整和内容，他们已经引用了所有这些，但是如果你把它们放在一起，那就太好了，就像这里是如何思考架构设计一样
- 你如何能把resblock架构对？如果些微改变一些东西，又会怎样不同？**因此对于研究和生产，您希望针对您的架构像这样重构您的代码，以便您可以查看它并说明究竟出了什么问题，我该怎么做，对于有效的从业者来说，能够编写漂亮简洁的架构非常重要，这样您就可以更改它们并理解它们，这就是我们的X ResNet**

Putting it Together

```

1 #export
2 class XResNet(nn.Sequential):
3     @classmethod
4     def create(cls, expansion, layers, c_in=3, c_out=1000):
5         # 3,32,64,64 前三层的网络
6         nfs = [c_in, (c_in+1)*8, 64, 64]
7         stem = [conv_layer(nfs[i], nfs[i+1], stride=2 if i==0 else 1)
8                 for i in range(3)]
9
10        nfs = [64//expansion, 64, 128, 256, 512]
11        res_layers = [cls._make_layer(expansion, nfs[i], nfs[i+1],
12                                         n_blocks=1, stride=1 if i==0 else 2)
13                         for i,l in enumerate(layers)]
14        res = cls(
15             *stem,
16             nn.MaxPool2d(kernel_size=3, stride=2, padding=1),
17             *res_layers,
18             nn.AdaptiveAvgPool2d(1), Flatten(),
19             nn.Linear(nfs[-1]*expansion, c_out),

```

```

20         )
21     init_cnn(res)
22     return res
23
24     @staticmethod
25     def _make_layer(expansion, ni, nf, n_blocks, stride):
26         return nn.Sequential(
27             *[ResBlock(expansion, ni if i==0 else nf, nf, stride if i==0
28 else 1)
29             for i in range(n_blocks)])
30     #export
31     def xresnet18 (**kwargs): return XResNet.create(1, [2, 2, 2, 2], **kwargs)
32     def xresnet34 (**kwargs): return XResNet.create(1, [3, 4, 6, 3], **kwargs)
33     def xresnet50 (**kwargs): return XResNet.create(4, [3, 4, 6, 3], **kwargs)
34     def xresnet101(**kwargs): return XResNet.create(4, [3, 4, 23, 3], **kwargs)
35     def xresnet152(**kwargs): return XResNet.create(4, [3, 8, 36, 3], **kwargs)

```

与创建任何ResNet模型所需的ResBlock代码相结合。：）

现在我们可以通过列出每一层有多少块和扩展因子(50+ 4)来创建所有的ResNets:

Train

- 使用loss function为交叉熵。有label smoothing标签平滑。resnet18.

```

1 cbfs = [partial(AvgstatsCallback, accuracy), ProgressCallback, CudaCallback,
2         partial(BatchTransformxCallback, norm_imagenette),
3         #         partial(Mixup, alpha=0.2)
4         ]
5 loss_func = LabelSmoothingCrossEntropy()
6 arch = partial(xresnet18, c_out=10)
7 opt_func = adam_opt(mom=0.9, mom_sqr=0.99, eps=1e-6, wd=1e-2)
8 #export
9 def get_batch(dl, learn):
10     learn.xb, learn.yb = next(iter(dl))
11     learn.do_begin_fit(0)
12     learn('begin_batch')
13     learn('after_fit')
14     return learn.xb, learn.yb

```

We need to replace the old `model_summary` since it used to take a `Runner`. 打印模型摘要，看看模型怎么组成的。

```

1 # export
2 def model_summary(model, data, find_all=False, print_mod=False):
3     xb,yb = get_batch(data.valid_dl, learn)
4     mods = find_modules(model, is_lin_layer) if find_all else
5     model.children()
6     f = lambda hook,mod,inp,out: print(f"====\n{mod}\n" if print_mod else "", 
7     out.shape)
8     with Hooks(mods, f) as hooks: learn.model(xb)
9     learn = Learner(arch(), data, loss_func, lr=1, cb_funcs=cbfs,
10     opt_func=opt_func)
11     learn.model = learn.model.cuda()
12     model_summary(learn.model, data, print_mod=False) # 改为true, 可以打印出整个块这将
13     非常有用, 可以帮助您了解模型中发生的事情

```

epoch	train_loss	train_accuracy	valid_loss	valid_accuracy	time
-------	------------	----------------	------------	----------------	------

```

1 torch.Size([128, 32, 64, 64])
2 torch.Size([128, 64, 64, 64])
3 torch.Size([128, 64, 64, 64])
4 torch.Size([128, 64, 32, 32])
5 torch.Size([128, 64, 32, 32])
6 torch.Size([128, 128, 16, 16])
7 torch.Size([128, 256, 8, 8])
8 torch.Size([128, 512, 4, 4])
9 torch.Size([128, 512, 1, 1])
10 torch.Size([128, 512])
11 torch.Size([128, 10])

```

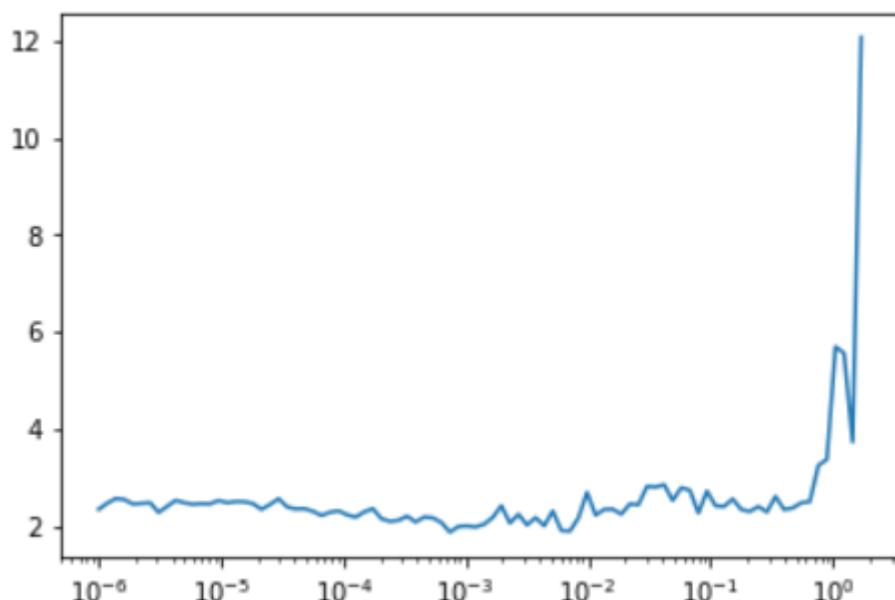
```

1 # 这里是我们的架构
2 arch = partial(xresnet34, c_out=10)
3 learn = Learner(arch(), data, loss_func, lr=1, cb_funcs=cbfs,
4 opt_func=opt_func)
5 learn.fit(1, cbs=[LR_Finder(), Recorder()])

```

epoch	train_loss	train_accuracy	valid_loss	valid_accuracy	time
-------	------------	----------------	------------	----------------	------

```
1 | learn.recorder.plot(3)
```



```

1 #export
2 def create_phases(phases):
3     phases = listify(phases)
4     return phases + [1-sum(phases)]
5 print(create_phases(0.3))
6 print(create_phases([0.3,0.2]))
7 [0.3, 0.7]
8 [0.3, 0.2, 0.5]
9 lr = 1e-2
10 pct_start = 0.5
11 phases = create_phases(pct_start)

```

```

12 sched_lr = combine_scheds(phases, cos_1cycle_anneal(1r/10., 1r, 1r/1e5))
13 sched_mom = combine_scheds(phases, cos_1cycle_anneal(0.95, 0.85, 0.95))
14 cbsched = [
15     ParamScheduler('lr', sched_lr),
16     ParamScheduler('mom', sched_mom)]
17 learn = Learner(arch(), data, loss_func, lr=lr, cb_funcs=cbfs,
18 opt_func=opt_func)
19 learn.fit(5, cbs=cbsched)

```

epoch	train_loss	train_accuracy	valid_loss	valid_accuracy	time
0	1.701341	0.486505	1.767135	0.510000	00:38
1	1.488563	0.590507	1.830278	0.514000	00:38
2	1.345791	0.651078	1.440738	0.638000	00:38
3	1.174334	0.727393	1.005328	0.792000	00:38
4	1.019539	0.790213	0.912079	0.824000	00:39

- 在Imagenette上的acc在5个Epoch上是82.4%, 在Imagenette官网上面, 128的5epoch是84.6%acc
- 在128pixel的Imagenette上训练5个epoch对ImageNet训练更大的模型有很多影响, 所以你可以通过不尝试训练巨型模型来学到很多东西。希望大家在leaderboard上打败Jeremy的记录。
- Image wolf只有小狗的照片, 但是hard很多。

cnn_learner

- 我们可以重构所有的东西, 把回调和所有的东西加入到CNN learner中。

```

1 #export
2 def cnn_learner(arch, data, loss_func, opt_func, c_in=None, c_out=None,
3                 lr=1e-2, cuda=True, norm=None, progress=True, mixup=0,
4                 xtra_cb=None, **kwargs):
5     cbfs = [partial(AvgStatsCallback, accuracy)]+listify(xtra_cb)
6     if progress: cbfs.append(ProgressCallback)
7     if cuda: cbfs.append(CudaCallback)
8     if norm: cbfs.append(partial(BatchTransformXCallback, norm))
9     if mixup: cbfs.append(partial(MixUp, mixup))
10    arch_args = {}
11    if not c_in: c_in = data.c_in
12    if not c_out: c_out = data.c_out
13    if c_in: arch_args['c_in']=c_in
14    if c_out: arch_args['c_out']=c_out
15    return Learner(arch(**arch_args), data, loss_func, opt_func=opt_func,
16    lr=lr, cb_funcs=cbfs, **kwargs)
17 learn = cnn_learner(xresnet34, data, loss_func, opt_func,
18 norm=norm_imagenette)
19 learn.fit(5, cbsched)

```

epoch	train_loss	train_accuracy	valid_loss	valid_accuracy	time
0	1.696614	0.493175	1.633222	0.540000	00:24
1	1.511717	0.580968	1.535915	0.576000	00:24
2	1.387270	0.632232	1.332997	0.656000	00:23
3	1.200042	0.710253	1.002243	0.810000	00:24
4	1.036241	0.781681	0.922973	0.834000	00:23

Imagenet

你可以在fastai imagenet训练脚本中看到所有这些内容。它和我们目前看到的是一样的，除了它也处理多gpu训练。那么这种方法的效果如何呢？

- 我们训练了60个epoch，得到了5.9%的误差，而PyTorch官方的resnet在90个epoch中得到了7.5%的误差！93.3%
- 我们的xresnet 50训练甚至超过标准的resnet 152，后者训练超过50%的epoch，有3倍的layer。
- 我们的xresnet50是建立在基本的常识之上的，并得到了惊人的结果。所以Jeremy真的不认为我们都的要去搜索神经架构和超参数优化。只要使用良好的常识思维就可以得到很好的模型。
- 现在我们有一个很好的模型之后，希望能够进行迁移学习！

Platform	Model	Acc@1	Acc@5	Rank@1	Rank@5	Input Size
Keras 2.2.4	nasnetlarge	80.83	95.27	1	1	331
Keras 2.2.4	inceptionresnetv2	78.93	94.45	2	2	299
PyTorch 1.0	resnet152	77.62	93.81	3	3	224
Keras 2.2.4	xception	77.18	93.49	4	4	299
PyTorch 1.0	densenet161	76.92	93.49	5	5	224

fastai xresnet:
 94.1% acc@5
 224 size
 60 epochs

图像分类：迁移学习/微调 fine-tune

([Jump to 第12课视频](#), (笔记本: [11a transfer learning.ipynb](#))

回想一下 fastai 第1部分中熟悉的“one two”训练组合，以便在图像分类任务上获得良好的结果：

1. 获取预训练的模型权重的 ResNet 权重
 2. 为您的新任务创建模型的新“头部head”部分。
 3. 冻结除头部以外的所有层。
 4. 为头部运行几个周期的训练。
 5. 解冻所有层并再运行几个训练周期。
- 如何从头开始做迁移学习呢？将image wolf上的学习模型，迁移到Pets数据集中。
 - 所有的模块都没有用fastai库的东西，都是我们自己写的。

让我们实现使这成为可能所需的代码。

```
1 | #export
```

```

2 from exp.nb_11 import *
3 path = datasets untar_data(datasets.URLS.IMAGEWOOF_160)
4 size = 128
5 bs = 64
6
7 tfms = [make_rgb, RandomResizedCrop(size, scale=(0.35,1)), np_to_float,
PILRandomFlip()]
8 val_tfms = [make_rgb, CenterCrop(size), np_to_float]
9 il = ImageList.from_files(path, tfms=tfms)
10 sd = SplitData.split_by_func(il, partial(grandparent_splitter,
valid_name='val'))
11 ll = label_by_func(sd, parent_labeler, proc_y=categoryProcessor())
12 ll.valid.x.tfms = val_tfms
13 data = ll.to_databunch(bs, c_in=3, c_out=10, num_workers=8)
14 len(il)
15 loss_func = LabelsSmoothingCrossEntropy()
16 opt_func = adam_opt(mom=0.9, mom_sqr=0.99, eps=1e-6, wd=1e-2)
17 learn = cnn_learner(xresnet18, data, loss_func, opt_func,
norm=norm_imagenette)
18 def sched_1cycle(lr, pct_start=0.3, mom_start=0.95, mom_mid=0.85,
mom_end=0.95):
19     phases = create_phases(pct_start)
20     sched_lr = combine_scheds(phases, cos_1cycle_anneal(lr/10., lr,
lr/1e5))
21     sched_mom = combine_scheds(phases, cos_1cycle_anneal(mom_start, mom_mid,
mom_end))
22     return [ParamScheduler('lr', sched_lr),
23             ParamScheduler('mom', sched_mom)]
24 lr = 3e-3
25 pct_start = 0.5
26 cbsched = sched_1cycle(lr, pct_start)
27 learn.fit(40, cbsched)

```

```

1 st = learn.model.state_dict()
2 type(st)
3 ', '.join(st.keys())
4 st['10.bias']
5 mdl_path = path/'models'
6 mdl_path.mkdir(exist_ok=True)

```

epoch	train_loss	train_accuracy	valid_loss	valid_accuracy	time
0	2.107170	0.264493	2.124692	0.262000	00:12
1	1.948093	0.336759	1.993934	0.308000	00:11
2	1.839806	0.400915	2.051041	0.282000	00:11
3	1.751581	0.444516	1.903694	0.370000	00:11
4	1.694040	0.469167	1.967755	0.368000	00:11
5	1.655241	0.493095	1.682255	0.466000	00:11
6	1.617931	0.505862	2.150986	0.322000	00:11
7	1.577291	0.524570	2.255864	0.354000	00:11
8	1.520226	0.558054	1.916284	0.444000	00:11
22	1.055159	0.768749	1.284904	0.666000	00:11
23	1.024775	0.781596	1.210474	0.710000	00:12
24	0.999277	0.795487	1.167366	0.726000	00:12
25	0.981030	0.804721	1.222561	0.708000	00:12
26	0.944283	0.821583	1.124248	0.730000	00:12
27	0.927088	0.828007	1.083567	0.754000	00:12
28	0.900515	0.842460	1.059961	0.770000	00:12
29	0.862564	0.857556	1.038407	0.788000	00:11
30	0.839758	0.867914	1.032012	0.776000	00:12
31	0.825747	0.871527	1.018186	0.776000	00:11
32	0.789751	0.891762	1.014238	0.792000	00:12
33	0.767951	0.902842	0.999521	0.796000	00:12
34	0.754325	0.907660	1.005709	0.794000	00:12
35	0.735343	0.915449	0.989576	0.804000	00:11
36	0.721974	0.924201	0.982883	0.800000	00:12
37	0.716628	0.928617	0.985535	0.810000	00:12
38	0.708484	0.930464	0.988875	0.804000	00:12
39	0.715810	0.928296	0.992937	0.796000	00:12

- 做了40个epoch，因为我们要做迁移学习。
- 11s做一个epoch，最后得到70.6%的acc

- 但是这个模型没有训练猫，有点棘手。而且这个模型只接受过不到 10,000 张图像的训练
- 所以我们这样的模型迁移还有效吗？work吗？我们很有兴趣！
- 也可以保存整个模型，包括架构，但这非常繁琐，我们不推荐这样做。相反，只需保存参数，并直接重新创建模型。
 - 把模型放进一个字典里面，子弹是有顺序的。可以轻松将模块转换为字典。

```

1 | torch.save(st, mdl_path/'iw5')
2 | # pickle也可以实现相同功能

```

Pets

```

1 | pets = datasets untar_data(datasets.URLS.PETS)
2 | pets.ls()
3 | pets_path = pets/'images'
4 | il = ImageList.from_files(pets_path, tfms=tfms)
5 | il
6 | #export
7 | def random_splitter(fn, p_valid): return random.random() < p_valid
8 | random.seed(42)
9 | sd = splitData.split_by_func(il, partial(random_splitter, p_valid=0.1))
10 | sd
11 |
12 | n = il.items[0].name; n
13 |
14 | re.findall(r'^(.*)_\\d+.jpg$', n)[0]
15 |
16 | def pet_labeler(fn): return re.findall(r'^(.*)_\\d+.jpg$', fn.name)[0]
17 |
18 | proc = CategoryProcessor()
19 |
20 | ll = label_by_func(sd, pet_labeler, proc_y=proc)
21 |
22 | ', '.join(proc.vocab)
23 |
24 | ll.valid.x.tfms = val_tfms
25 |
26 | c_out = len(proc.vocab)
27 |
28 | data = ll.to_databunch(bs, c_in=3, c_out=c_out, num_workers=8)
29 | learn = cnn_learner(xresnet18, data, loss_func, opt_func,
30 | norm=norm_imagenette)
| learn.fit(5, cbsched)

```

epoch	train_loss	train_accuracy	valid_loss	valid_accuracy	time
0	3.477494	0.085946	3.421262	0.095436	00:07
1	3.298791	0.134843	3.404232	0.096819	00:07
2	3.061584	0.195740	3.288172	0.179806	00:07
3	2.750766	0.291135	2.701614	0.280775	00:07
4	2.457208	0.386381	2.501944	0.377593	00:07

- 拿到数据后，从头开始训练的。只有30%的acc。不是很好，也许通过迁移学习可以做的更好！

Custom Head

在笔记本中，他们想使用ImageWoof预先训练的模型来微调宠物Pets数据集。我们可以将ImageWoof模型作为layer_name: tensor的字典保存到磁盘上。PyTorch模型可以使用`st = learn.model.state_dict()`。

- 让我们回顾一下加载回预先训练过的ImageWoof模型的过程。首先，我们需要创造一个学习者：

```
1 | learn = cnn_learner(xresnet18, data, loss_func, opt_func, c_out=10,
2 | norm=norm_imagenette)
```

- ImageWoof有10个激活的，所以我们需要匹配这一所以权重匹配：`c_out=10`。

然后我们可以加载 ImageWoof 状态字典并将权重加载到我们的 Learner：

```
1 | st = torch.load(md1_path/'iw5')
2 | m = learn.model
3 | m.load_state_dict(st)
```

- 现在这只是恢复后的 ImageWoof 模型。我们想要改变它，以便它可以用于新的数据集，所以我们取下最后一个线性层的10类，并替换为一个宠物数据集的37类。我们可以通过搜索指向神经网络`nn.AdaptiveAvgPool2d`层的索引切点来找到我们想要切割模型的点。
`nn.AdaptiveAvgPool2d`层即是head之前的倒数第二层，找到我们想要切割模型的点。切割模型为：`m_cut = m[:cut]`。

```
1 | cut = next(i for i,o in enumerate(m.children()) if
2 |     isinstance(o,nn.AdaptiveAvgPool2d))
3 | m_cut = m[:cut]
4 | xb,yb = get_batch(data.valid_dl, learn)
5 | pred = m_cut(xb)
6 | pred.shape
7 | ni = pred.shape[1]
```

- 我们的新头的输出数量是 37，那么输入呢？我们可以判断，我们可以很容易地通过切割模型运行一批：`ni = m_cut(xb).shape[1]`。
- 因为我们的输出得是cut模型的输入，所以运行一个batch来看一下。
- 我们现在可以为 Pets 模型创建新的头部：

```
1 | m_new = nn.Sequential(
2 |     m_cut, AdaptiveConcatPool2d(), Flatten(),
3 |     nn.Linear(ni*2, data.c_out))
```

```
1 | #export
2 | class AdaptiveConcatPool2d(nn.Module):
3 |     def __init__(self, sz=1):
4 |         super().__init__()
5 |         self.output_size = sz
6 |         self.ap = nn.AdaptiveAvgPool2d(sz)
7 |         self.mp = nn.AdaptiveMaxPool2d(sz)
8 |     def forward(self, x): return torch.cat([self.mp(x), self.ap(x)], 1)
```

```

9 nh = 40
10
11 m_new = nn.Sequential(
12     m_cut, AdaptiveConcatPool2d(), Flatten(),
13     nn.Linear(ni*2, data.c_out))
14 learn.model = m_new
15 learn.fit(5, cbsched)

```

epoch	train_loss	train_accuracy	valid_loss	valid_accuracy	time
0	2.882488	0.292485	2.210701	0.441217	00:07
1	2.046879	0.512224	2.200438	0.477178	00:07
2	1.841778	0.595170	1.897576	0.568465	00:07
3	1.619549	0.674966	1.697158	0.645920	00:07
4	1.422663	0.757162	1.566284	0.710927	00:07

- 我们还使用AdaptiveConcatPool2d，它只是将平均池和最大池合并成一个 $2 \times ni$ 大小的向量。这个双池是一个快速的技巧，它比只做另一个提供了一点提升。
- 通过这种简单的迁移学习，我们可以在4个epoch后获得71%的宠物。没有迁移学习，我们只得到37%。

把所有的步骤放在一起:把整个过程放在一个函数中:

adapt_model and gradual unfreezing

```

1 def adapt_model(learn, data):
2     cut = next(i for i,o in enumerate(learn.model.children())
3                 if isinstance(o,nn.AdaptiveAvgPool2d))
4     m_cut = learn.model[:cut]
5     xb,yb = get_batch(data.valid_dl, learn)
6     pred = m_cut(xb)
7     ni = pred.shape[1]
8     m_new = nn.Sequential(
9         m_cut, AdaptiveConcatPool2d(), Flatten(),
10        nn.Linear(ni*2, data.c_out))
11    learn.model = m_new

```

则权重加载和模型适配简单化为:

```

1 learn = cnn_learner(xresnet18, data, loss_func, opt_func, c_out=10,
2 norm=norm_imagenette)
3 learn.model.load_state_dict(torch.load(md1_path/'iw5'))
4 adapt_model(learn, data)

```

Freezing Layers

- 最开始只训练前面几层，所以要把后面的层给冻结起来。

你可以通过关闭grad来冻结freeze layers:

```

1 | for p in learn.model[0].parameters(): p.requires_grad_(False)
2 | learn.fit(3, sched_1cycle(1e-2, 0.5))

```

epoch	train_loss	train_accuracy	valid_loss	valid_accuracy	time
0	2.757396	0.308535	2.517377	0.396957	00:05
1	2.483888	0.419979	2.228861	0.463347	00:05
2	2.078623	0.533673	2.046422	0.543568	00:05

- 训练好了head，继续解冻，然后训练后面的层：

```

1 | for p in learn.model[0].parameters(): p.requires_grad_(True)
2 | learn.fit(5, cbsched, reset_opt=True)

```

epoch	train_loss	train_accuracy	valid_loss	valid_accuracy	time
0	1.920109	0.572671	2.039839	0.556017	00:05
1	1.931590	0.573871	2.086156	0.511757	00:05
2	1.978338	0.550022	2.024737	0.528354	00:05
3	1.913799	0.580171	1.973329	0.560166	00:05
4	1.824862	0.608820	1.958072	0.562932	00:05

- 让我们做一个One Two训练组合。

- 冻结body，训练head3个epoch得到54%。
- 解冻unfreezing和训练其余的模型5个epoch得到56%(!)

这比不进行微调要好 (30%)，但有趣的是，**当我们在没有冻结的情况下进行微调时我们得到了更好的结果71%。**，但是比不微调差 (71%)
为什么它不起作用？

每次在你的神经网络中发生奇怪的事情，几乎可以肯定是由于批量规范batch normalization。因为批量标准让一切都很奇怪。: D

- 预训练模型中的 batchnorm 层已经学习了不同数据集 (ImageWoof) 的均值和标准差。当我们训练冻结身体并训练头部时，头部正在使用一组不同的批量规范统计数据进行学习。
- 当我们解冻 body 时，batchnorm 统计现在可以改变，这有效地导致地面从我们刚刚改变的后面的层下面转移
- 冻结了模型的部分，bn减去的均值和标准差是ImageWolf数据集的。但是Pet数据集有不同的均值和方差，但在模型内部，我们解冻模型的时候还是ImageWolf的均值和方差。
 - fastai是一门课程提醒大家注意这一点。之前这些东西都是隐藏在库里的。56%和71%是比较大的差别
- 好消息是，这个很容易修复。诀窍是：不要冻结所有的body参数，冻结所有的非bn层的参数。bn层的参数不要冻结。
 - 也就是说，我们在微调的时候，也微调bn层的权重和偏差！！！
- 修复：在进行部分层训练时，不要冻结 batchnorm 层中的权重。

下面是对层进行冻结和解冻的函数，它跳过了批量层：

```

1 | def set_grad(m, b):
2 |     if isinstance(m, (nn.Linear, nn.BatchNorm2d)): return
3 |     if hasattr(m, 'weight'):
4 |         for p in m.parameters(): p.requires_grad_(b)

```

我们可以使用 PyTorch `apply` 方法将此函数应用于我们的模型：

```
1 | learn.model.apply(partial(set_grad, b=False));
```

Batch norm transfer

```

1 | learn = cnn_learner(xresnet18, data, loss_func, opt_func, c_out=10,
2 | norm=norm_imagenette)
3 | learn.model.load_state_dict(torch.load(md1_path/'iw5'))
4 | adapt_model(learn, data)
5 | def apply_mod(m, f):
6 |     f(m)
7 |     for l in m.children(): apply_mod(l, f)
8 |
9 | def set_grad(m, b):
10 |     if isinstance(m, (nn.Linear, nn.BatchNorm2d)): return
11 |     if hasattr(m, 'weight'):
12 |         for p in m.parameters(): p.requires_grad_(b)
13 |
14 | apply_mod(learn.model, partial(set_grad, b=False))
| learn.fit(3, sched_1cycle(1e-2, 0.5))

```

epoch	train_loss	train_accuracy	valid_loss	valid_accuracy	time
0	2.642298	0.338533	2.342353	0.449516	00:06
1	2.123198	0.500225	2.059914	0.520055	00:05
2	1.857212	0.591120	1.849981	0.586445	00:06

- 我们冻结的时候没有冻结bn层，让bn的参数一起微调。可以发现，这样比之前冻结了bn的效果要好。从54%提升到了58%！

```

1 | apply_mod(learn.model, partial(set_grad, b=True))
2 | learn.fit(5, cbsched, reset_opt=True)

```

epoch	train_loss	train_accuracy	valid_loss	valid_accuracy	time
0	1.750263	0.637468	1.853926	0.603043	00:07
1	1.745434	0.635368	2.028596	0.531120	00:07
2	1.685524	0.649768	1.923829	0.543568	00:07
3	1.531872	0.709464	1.635953	0.683264	00:07
4	1.380929	0.778911	1.578964	0.704011	00:07

- 更重要的是，我们解冻后训练，我们的acc又回到了70%
- fastai可能是第一个这样做的库，如果是TensorFlow或者其他库，你需要自己写代码，来确保不同冻结bn层的权重。
- pytorch自己有一个apply_mod函数，功能跟我们上面的类似

```
1 | learn.model.apply(partial(set_grad, b=False));
```

Discriminative LR and param groups

- 我们要研究区别学习率。
 - 是一种简单的方法来做 layer freezing，可以将学习率设置为0，那么该层就相当于冻结了。
- 所以我们要做的是将学习率参数分为两组或更多组，用来分离的函数为bn_splitter

```
1 | learn = cnn_learner(xresnet18, data, loss_func, opt_func, c_out=10,
2 | norm=norm_imagenette)
3 | learn.model.load_state_dict(torch.load(md1_path/'iw5'))
4 | adapt_model(learn, data)
5 |
6 | def bn_splitter(m):
7 |     def _bn_splitter(l, g1, g2):
8 |         if isinstance(l, nn.BatchNorm2d): g2 += l.parameters()
9 |         elif hasattr(l, 'weight'): g1 += l.parameters()
10 |         for ll in l.children(): _bn_splitter(ll, g1, g2)
11 |         # 两组参数，如果是bn，就放在g2，其他的放在g1
12 |         g1,g2 = [],[]
13 |         _bn_splitter(m[0], g1, g2)
14 |
15 |         g2 += m[1:].parameters()
16 |         return g1,g2
17 |
18 | a,b = bn_splitter(learn.model)
19 | test_eq(len(a)+len(b), len(list(m.parameters())))
20 |
```

- 我们的Learner里面也有一个splitter函数，将模型传给splitter

在09b_learner.ipynb中，在fit函数中把model传递给了splitter，然后分为两个参数组g1和g2，然后把g1和g2给opt_fun函数，来创建优化器。

```
1 | class Learner():
2 |     def __init__(self, model, data, loss_func, opt_func=sgd_opt, lr=1e-2,
3 |                  splitter=param_getter,
4 |                  cbs=None, cb_funcs=None):
5 |
6 |         self.model, self.data, self.loss_func, self.opt_func, self.lr, self.splitter =
7 |         model, data, loss_func, opt_func, lr, splitter
8 |         # 这里有splitter函数
9 |     def fit(self, epochs, cbs=None, reset_opt=False):
10 |         # NEW: pass callbacks to fit() and have them removed when done
11 |         self.add_cbs(cbs)
12 |         # NEW: create optimizer on fit(), optionally replacing existing
13 |         if reset_opt or not self.opt:
14 |             self.opt = self.opt_func(self.splitter(self.model), lr=self.lr)
```

- 可能会出错，比如可能不训练模型的最后一层，比如以相同的学习率训练所有层。很难知道模型是否糟糕，是否代码搞错了。我们需要一种方法来debug，不能只看内部，而是要看通过网络的东西。
- 我们在创建learner时就将Splitter传递过去了。
- 所以需要看model内部，使用回调函数。

1 | Learner.ALL_CBS

- 现在我们定义一个函数，print_det打印细节，放在debugcallback的函数中。

```

1 #export
2 from types import SimpleNamespace
3 cb_types = SimpleNamespace(**{o:o for o in Learner.ALL_CBS})
4 cb_types.after_backward
5
6 #export
7 class DebugCallback(callback):
8     _order = 999
9     # 我们可以指定，我们想要debug哪一个callback函数
10    def __init__(self, cb_name, f=None): self.cb_name, self.f = cb_name, f
11    def __call__(self, cb_name):
12        if cb_name==self.cb_name:
13            if self.f: self.f(self.run)
14        else: set_trace()

```

S

```

1 #export
2 def sched_1cycle(lrs, pct_start=0.3, mom_start=0.95, mom_mid=0.85,
3 mom_end=0.95):
4     phases = create_phases(pct_start)
5     sched_lr = [combine_scheds(phases, cos_1cycle_anneal(lr/10., lr,
6     lr/1e5))]
6         for lr in lrs]
7     sched_mom = combine_scheds(phases, cos_1cycle_anneal(mom_start, mom_mid,
8     mom_end))
8     return [ParamScheduler('lr', sched_lr),
9             ParamScheduler('mom', sched_mom)]
9
10 disc_lr_sched = sched_1cycle([0, 3e-2], 0.5)

```

```

1 # Learner自带一个splitter
2 learn = cnn_learner(xresnet18, data, loss_func, opt_func,
3                      c_out=10, norm=norm_imagenette, splitter=bn_splitter)
4
5 learn.model.load_state_dict(torch.load(md1_path/'iw5'))
6 adapt_model(learn, data)
7
8 def _print_det(o): # 打印有多少个param_groups, 有多少个超参数, 然后马上停止训练
9     print (len(o.opt.param_groups), o.opt.hypers)
10    raise CancelTrainException()
11
12 learn.fit(1, disc_lr_sched + [DebugCallback(cb_types.after_batch,
13                                _print_det)])

```

一旦fit，就立马停止。会打印出来。2个参数组。超参数为如下：第一组参数组的lr=0，第二组为0.03.跟上面的disc_lr_sched是一样的。

```

epoch train_loss train_accuracy valid_loss valid_accuracy time
2 [{"mom": 0.9499999999999997, "mom_sqr": 0.99, "eps": 1e-06, "wd": 0.01, "lr": 0.0, "sqr_mom": 0.99}, {"mom": 0.9499999999999997, "mom_sqr": 0.99, "eps": 1e-06, "wd": 0.01, "lr": 0.00300000000000512, "sqr_mom": 0.99}]

```

```
1 | learn.fit(3, disc_lr_sched)
```

epoch	train_loss	train_accuracy	valid_loss	valid_accuracy	time
0	2.586969	0.352332	2.324385	0.453665	00:07
1	2.029975	0.523024	1.969945	0.526971	00:07
2	1.648762	0.661167	1.638924	0.668050	00:07

```

1 | disc_lr_sched = sched_1cycle([1e-3,1e-2], 0.3)
2 | learn.fit(5, disc_lr_sched)

```

epoch	train_loss	train_accuracy	valid_loss	valid_accuracy	time
0	1.514395	0.727764	1.885501	0.583679	00:07
1	1.637216	0.672416	1.746767	0.611342	00:07
2	1.527710	0.706915	1.727681	0.636238	00:07
3	1.394199	0.765562	1.560399	0.687414	00:07
4	1.298138	0.797360	1.526475	0.686030	00:07

Querstion:你为什么反对在深度学习中使用交叉验证?

如果您没有足够的数据来创建一个合理大小的验证集，交叉验证是一个非常有用的技术，可以获得一个合理大小的验证集。

在大多数研究都是50-60排的时候，它特别流行。如果只有1000行，那就没有意义了——无论如何统计意义都是存在的。

我不反对它，只是大多数时候你不需要它。因为如果你在验证集中有1000个东西，而你只关心它是正负1%，那就完全没有意义了。

看看您的验证集的准确性在每次运行中有多大的变化，如果变化太大，您无法做出您需要做出的决定，那么您可以添加交叉验证。

question2: 你调试深度学习的技巧是什么？

一开始就不要犯错误！

做到这一点的唯一方法是使**代码简单到不可能有错误**，并检查每个中间结果，以确保没有错误。

否则你可能会像我上个月那样花上一个月的时间。一个月前，我在ImageNet上的准确率达到了94.1%。

然后我开始尝试各种调整，但似乎都没有帮助。作为一个理智的检查，我决定重复之前的训练，我不能重复它-我得到了93.5%。(在AWS上，每次6小时的培训花费了我150美元!)

所以即使意识到它坏了也是一个很大的过程。**当你编写了错误的深度学习代码时，它会以你甚至察觉不到的方式被破坏**，而且可能已经broken for few weeks。

- 你需要成为一个伟大的科学家来进行深度学习——将你的结果记录下来。我可以回到我的日志，看看我什么时候得到了94.1%的结果，所以我可以将fastai恢复到那个时间的提交，然后重新运行并成功地复制结果。
- 所以Jeremy现在必须弄清楚，到底哪一个改变让fastai库导致了broken。所以我尝试做的第一件事试图找到一种方法来快速找出是否有问题，但是在进行了几次运行并将它们绘制在Excel中之后，很明显训练是相同的，直到第50个epoch为止。
- 所以没有捷径，Jeremy对分搜索一个模块，，查看在那个差异中发生变化的15个模块，直到我最终发现它在混合精度模块中，然后我经历了混合位置模块中发生的每个变化，就像5,000美元后，终于找到了上了一行代码，在Opt中少了4个字母。
- 然后我可以在此期间对fastai所做的更改进行二分，直到我在混合精度模块中发现了bug。这个bug很微妙，直到纪元50年代才出现!找到它花了5000美元.....

这个微小的差异是如此微不足道，以至于使用fastai的人都没有注意到这个错误，只有在试图获得SOTA图像网结果时才会注意到。**以至于没有人使用该库甚至注意到它不起作用我不知道它没有起作用，直到我开始尝试你知道以68美元的价格在ImageNet上稳定地获得结果**

这些类型的“软bug”在深度学习中很常见，真的很难检测和追踪!!

我的意思是调试很难，而且更糟糕的是，大多数时候你都不知道，所以我的意思是说老实说，**训练模型很糟糕，深度学习是一种悲惨的经历，你应该不这样做**，但另一方面，它会给你带来比其他任何东西都更好的结果，它正在接管世界，所以要么那样做要么被其他人吃掉。

- 编写普通代码要容易的多。
- 测试时的思维方式完全不同，你不会想要重复的测试，而是想要随机性的测试。看看某些东西是否只是偶尔发生变化，想要随机性测试。想要是否某个时候会break。
- **所以一旦发生这种情况，我们就会尝试编写一个每次都失败的测试你知道**，一旦你意识到这件事有问题，**你就会试图找到一种方法让它每次都失败**，但是调试很困难。最后你只需要通过每一步查看你的数据确保它看起来很合理，然后尽量不要一开始就犯错误。

ULMFiT From Scratch

Question: 科学日志，记录日志 scientific journal是什么意思？

当你看看历史上的伟大科学家时，他们都有出色的科学期刊实践。

就我而言，它是一个名为Windows记事本的软件，我将内容粘贴到底部，当我想查找内容时，我按Ctrl-F。它只需要记录你正在做什么和结果是什么。因为取得突破的科学家通常会取得突破，因为他们看到了不应该发生的事情，然后他们会说：“哦！这很奇怪！这是怎么回事？”

例如，惰性气体的发现是因为一位科学家在烧杯中看到了一个小气泡，他们很确定那里不应该有气泡。大多数人会忽略它，但他们研究了气泡并发现了惰性气体。

另一个例子是青霉素。

我发现在深度学习中也是如此。我花了很多时间研究批量归一化和迁移学习，因为几年前在 Keras 中，我得到了糟糕的迁移学习结果，我认为应该更准确。我想——“哦，这很奇怪。” - 我花了数周时间改变我能改变的一切，然后几乎随机地尝试改变批处理规范。

所有这些摆弄 - 90% 不会导致任何地方，但其他 10% 你将无法弄清楚，除非你可以回去确认实验的结果确实发生了。

您可以记录日期、github 提交、日志、命令行等任何内容，以确保您可以返回并在以后重现实验。

- ULMFiT 是应用于 NLP 的 AWD-LSTM 的迁移学习
- 最近在应用于 NLP 的迁移学习领域出现了许多突破性的创新——例如 GPT2、BERT。这些都是基于目前非常热门的 Transformers，所以人们可能会认为 LSTM 不再被使用或不再有趣。但是，当您查看最近的竞争机器学习结果 (NB 记录 2019 年) 时，您会看到 ULMFiT 击败了 BERT - 来自[poleval2019](#):
- **Task 6: Automatic cyberbullying detection**

Subtask 6.1

Best system: n-waves ULMFiT

Prize winner: n-waves ULMFiT

System name	Precision	Recall	F1	Accuracy
n-waves ULMFiT	66.67	52.24	58.58	90.10
Przetak	66.35	51.49	57.98	90.00
ULMFiT + SentencePiece + BranchingAttention	52.90	54.48	53.68	87.40
ensamble spacy + tpot + BERT	52.71	50.75	51.71	87.30
ensamble + fastai	52.71	50.75	51.71	87.30
ensenble spacy + tpot	43.09	58.21	49.52	84.10
Rafal-1	41.08	56.72	47.65	83.30
Rafal-2	41.38	53.73	46.75	83.60

- 杰里米说.....:

RNN 在过去绝对不是真的。文本的 Transformer 和 CNNs 有很多问题。他们没有状态 state。因此，如果您正在进行语音识别，对于您查看的每个样本，您必须一次又一次地对其周围的所有样本进行全面分析。所以实在是太浪费了。

而 RNN 有状态。但是，当您想要进行研究和改变事物时，它们就很麻烦且难以处理。RNN，尤其是 AWD-LSTM，已经对如何仔细地规范化它们进行了大量研究。斯蒂芬·梅里蒂 (Stephen Merity) 做了大量工作，研究了所有不同的规范化方式。在 RNN 世界之外没有类似的东西。目前，对于大多数现实世界的任务，我的转到选择仍然是 ULMFiT。我还没有看到Transformer 赢得比赛或在实际工作中使用。

有很多东西不是文本序列——基因组学、化学键分析和药物发现。人们正在寻找 NLP 之外的令人兴奋的 ULMFiT 应用。ULMFiT 将是我们的重点，稍后我们也会了解 transformer。

- ULMFiT 不光是在文本上的迁移训练，最近在基因组学应用、化学键分析和药物发现方面也有很多最先进的结果，有很多东西是序列，结果证明我们还只是在冰山一角，因为大多数正在研究药物发现或化学键合或基因组学的人。
- 不仅对 NLP 有用，还对各种序列分类任务有用。真的有趣。

以下是在 fastai v3 的第 1 部分中看到的对 ULMFiT 管道的回顾：

ULMFiT is transfer learning applied to AWD-LSTM



- 所以基本过程将是在一些大数据集上创建一个语言模型，注意语言模型是一个非常笼统的术语，它意味着预测序列中的下一个项目，因此它可以是一个音频和语言模型，可以预测一段音乐或语音中的下一个样本，可以预测序列中的下一个基因组或其他任何正确的东西，这就是我所做的指语言模型。
- 然后对语言模型微调，如IMDb。预处理数据集。然后对于分类来进行微调

```
1 #export
2 from exp.nb_11a import *
```

Data

```
1 path = datasets untar_data(datasets.URLs.IMDB)
2 path.ls()
3 [PosixPath('/home/jupyter/.fastai/data/imdb/unsup'),
4 PosixPath('/home/jupyter/.fastai/data/imdb/tmp_clas'),
5 PosixPath('/home/jupyter/.fastai/data/imdb/1d.pkl'),
6 PosixPath('/home/jupyter/.fastai/data/imdb/test'),
7 PosixPath('/home/jupyter/.fastai/data/imdb/train'),
8 PosixPath('/home/jupyter/.fastai/data/imdb/README'),
9 PosixPath('/home/jupyter/.fastai/data/imdb/models'),
10 PosixPath('/home/jupyter/.fastai/data/imdb/tmp_lm'),
11 PosixPath('/home/jupyter/.fastai/data/imdb/vocab')]
```

预处理文本

笔记本：[12_text.ipynb](#)

我们将使用由 50,000 条带标签的电影评论（正面或负面）和 50,000 条未标记的评论组成的 IMDB 数据集。它包含一个 `train` 文件夹、一个 `test` 文件夹和一个 `unsup`（不受监督的）文件夹。

我们需要做的第一件事是 `ItemList` 为文本创建一个数据块子类：

```
1 #export
2 def read_file(fn):
3     with open(fn, 'r', encoding = 'utf8') as f: return f.read()
4
5 class TextList(ItemList):
6     @classmethod
7         def from_files(cls, path, extensions='.txt', recurse=True, include=None,
8 **kwargs):
9             return cls(get_files(path, extensions, recurse=recurse,
10 include=include), path, **kwargs)
11
12     def get(self, i):
13         if isinstance(i, Path): return read_file(i)
14         return i
```

这很容易，因为我们重用了之前编写的大量代码。我们已经有了这个 `get_files` 功能，它现在搜索 `.txt` 文件而不是图像。然后我们重写 `get` 现在调用一个 `read_file` 读取文本文件的函数。现在我们可以加载数据集：

```
1 il = TextList.from_files(path, include=['train', 'test', 'unsup'])
2 len(il.items)
3 100000
```

如果我们查看其中一项，它将只是 IMDB 电影评论的原始文本。

我们可以把它放到一个模型中——它需要是数字。所以我们需要对它进行**标记和数值化**。

```
1 txt = il[0]
2 txt
3 -----
4 'Comedian Adam Sandler\'s last theatrical release "I Now Pronounce You Chuck
and Larry" served as a loud and proud plea for tolerance of the g
```

对于文本分类，我们将像以前一样通过祖父母文件夹进行分割，但对于语言建模，我们将所有文本放在一边，只保留10%。

```

1 sd = splitData.split_by_func(il, partial(random_splitter, p_valid=0.1))
2 sd
3 SplitData
4 Train: TextList (89885 items)
5 [PosixPath('/home/jupyter/.fastai/data/imdb/unsup/30860_0.txt'),
 PosixPath('/home/jupyter/.fastai/data/imdb/unsup/36250_0.txt'),
 PosixPath('/home/jupyter/.fastai/data/imdb/unsup/24690_0.txt'),
 PosixPath('/home/jupyter/.fastai/data/imdb/unsup/21770_0.txt'),
 PosixPath('/home/jupyter/.fastai/data/imdb/unsup/9740_0.txt'),
 PosixPath('/home/jupyter/.fastai/data/imdb/unsup/40778_0.txt'),
 PosixPath('/home/jupyter/.fastai/data/imdb/unsup/44512_0.txt'),
 PosixPath('/home/jupyter/.fastai/data/imdb/unsup/22672_0.txt'),
 PosixPath('/home/jupyter/.fastai/data/imdb/unsup/25946_0.txt'),
 PosixPath('/home/jupyter/.fastai/data/imdb/unsup/40866_0.txt')...]
6 Path: /home/jupyter/.fastai/data/imdb
7 Valid: TextList (10115 items)
8 [PosixPath('/home/jupyter/.fastai/data/imdb/unsup/1041_0.txt'),
 PosixPath('/home/jupyter/.fastai/data/imdb/unsup/38186_0.txt'),
 PosixPath('/home/jupyter/.fastai/data/imdb/unsup/16367_0.txt'),
 PosixPath('/home/jupyter/.fastai/data/imdb/unsup/47167_0.txt'),
 PosixPath('/home/jupyter/.fastai/data/imdb/unsup/58_0.txt'),
 PosixPath('/home/jupyter/.fastai/data/imdb/unsup/49861_0.txt'),
 PosixPath('/home/jupyter/.fastai/data/imdb/unsup/306_0.txt'),
 PosixPath('/home/jupyter/.fastai/data/imdb/unsup/18238_0.txt'),
 PosixPath('/home/jupyter/.fastai/data/imdb/unsup/34952_0.txt'),
 PosixPath('/home/jupyter/.fastai/data/imdb/unsup/24288_0.txt')...]
9 Path: /home/jupyter/.fastai/data/imdb

```

Tokenizing

我们需要先对数据集进行分词，即将一个句子拆分为单独的词。这些标记是经过一些调整的基本单词或标点符号：`don't` 例如在 `do` 和 `and` 之间拆分 `n't`。我们将为此使用处理器，并结合[spacy 库](#)。

在标记化之前，我们将对文本进行一些预处理以清理它们（我们看到那里有一些 HTML 代码）。在我们将句子拆分为标记之前应用这些规则。

```

1 #export
2 import spacy,html
3 #export
4 #special tokens
5 UNK, PAD, BOS, EOS, TK REP, TK_WREP, TK_UP, TK_MAJ = "xxunk xxpad xxbos
xxeos xxrep xxwrep xxup xxmaj".split()
6
7 def sub_br(t):
8     "Replaces the <br /> by \n"
9     re_br = re.compile(r'<\s*br\s*/?>', re.IGNORECASE)
10    return re_br.sub("\n", t)
11
12 def spec_add_spaces(t):
13     "Add spaces around / and #"
14     return re.sub(r'([/#])', r' \1 ', t)
15
16 def rm_useless_spaces(t):
17     "Remove multiple spaces"
18     return re.sub(' {2,}', ' ', t)
19

```

```

20 def replace_rep(t):
21     "Replace repetitions at the character level: cccc -> TK REP 4 c"
22     def _replace_rep(m:Collection[str]) -> str:
23         c,cc = m.groups()
24         return f' {TK REP} {len(cc)+1} {c} '
25     re_rep = re.compile(r'(\s)(\1{3,})')
26     return re_rep.sub(_replace_rep, t)
27
28 def replace_wrep(t):
29     "Replace word repetitions: word word word -> TK_WREP 3 word"
30     def _replace_wrep(m:Collection[str]) -> str:
31         c,cc = m.groups()
32         return f' {TK_WREP} {len(cc.split())+1} {c} '
33     re_wrep = re.compile(r'(\b\w+\w+)(\1{3,})')
34     return re_wrep.sub(_replace_wrep, t)
35
36 def fixup_text(x):
37     "Various messy things we've seen in documents"
38     re1 = re.compile(r' +')
39     x = x.replace(';', '').replace('amp;', '&').replace('#146;',
40     '').replace(
41         '\nbsp;', ' ').replace('#36;', '$').replace('\\n',
42         '\n').replace('quot;', '"').replace(
43             '<br />', '\n').replace('\\\\', '').replace('<unk>', UNK).replace(
44             '@@', '.').replace(
45                 '@-@', '-').replace('\\\\', '\\ ')
46     return re1.sub(' ', html.unescape(x))
47 # 预处理规则:
48 default_pre_rules = [fixup_text, replace_rep, replace_wrep, spec_add_spaces,
49 rm_useless_spaces, sub_br]
50 default_spec_tok = [UNK, PAD, BOS, EOS, TK REP, TK_WREP, TK_UP, TK_MAJ]

```

1 | replace_rep('cccc')

如果我们找到 BR / 我们用新行替换它或者我们找到一个斜杠或者一个散列，如果我们在一行中发现两个以上的空格，我们就在它周围放空格，我们只把它变成一个空格，然后我们就有了这些特殊的标记，这就是它们作为字符串的样子

1 | ' xxrep 4 c '

1 | replace_wrep('word word word word word ')

1 | ' xxwrep 5 word '

这些是：

- `fixup_text`：修复了文档中看到的各种乱七八糟的东西。例如，HTML 工件。
- `replace_rep`：在字符级别替换重复：!!!!! -> TK REP 5 !
- `replace_wrep`：替换单词重复：word word word -> TK_WREP 3 word
- `spec_add_spaces`：在 / 和周围添加空格 #
- `rm_useless_spaces`：如果我们发现一行中有两个以上的空格，就用一个空格替换它们
- `sub_br`：替换
用\n

为什么做 replace_rep 和 replace_wrep？让我们想象一条推文说：“这太棒了！！！！！！！！！！！！！！！！！！！！！！！！！！”。我们可以将感叹号视为一个标记，因此我们将拥有一个专门为 21 个感叹号的词汇项。你可能不会再看到它，所以它甚至不会出现在你的词汇中，如果它出现了，那将是非常罕见的，以至于你将无法从中学到任何有趣的东西。它也与有 20 或 22 个感叹号的情况有很大不同。但是一些感叹号确实是有意义的，我们知道这与只有一个感叹号的情况不同。如果我们用 xxrep 21 替代它！，那么这只是三个标记，模型可以从中了解到许多重复的感叹号是一个具有特定语义的一般概念。

另一种选择是将我们的感叹号序列连续变成 21 个标记，但现在我们要求我们的 LSTM 保持该状态 21 个时间步长，这需要做更多的工作，但它不会做作为一份好工作。

我们在 NLP 中试图做的是让我们的词汇表中的东西尽可能有意义。

在使用标记化之后，spacey 我们应用了更多规则：

- replace_all_caps：将所有大写中的标记替换为其较低版本并在 TK_UP 之前插入。
- deal_caps：将所有 Capitalized 标记替换为其较低版本并在 TK_MAJ 之前添加。
- add_eos_bos：在文档开始/结束的标记列表的任一侧添加流前(BOS)和流**结束(EOS)标记。事实证明，这些令牌非常重要。当模型遇到一个 EOS 标记时，它知道它在文档的末尾并且下一个文档是新的。所以它必须学会以某种方式重置它的状态。

例如：

```
1 #export
2 def replace_all_caps(x):
3     "Replace tokens in ALL CAPS by their lower version and add `TK_UP` before."
4     res = []
5     for t in x:
6         if t.isupper() and len(t) > 1: res.append(TK_UP);
7     res.append(t.lower())
8     else: res.append(t)
9     return res
10
11 def deal_caps(x):
12     "Replace all Capitalized tokens in by their lower version and add `TK_MAJ` before."
13     res = []
14     for t in x:
15         if t == '': continue
16         if t[0].isupper() and len(t) > 1 and t[1:].islower():
17             res.append(TK_MAJ)
18             res.append(t.lower())
19     return res
20
21 def add_eos_bos(x): return [BOS] + x + [EOS]
22
23 default_post_rules = [deal_caps, replace_all_caps, add_eos_bos]
```

```
1 | replace_all_caps(['I', 'AM', 'SHOUTING'])
```

```
1 | ['I', 'xxup', 'am', 'xxup', 'shouting']
```

```
1 | deal_caps(['My', 'name', 'is', 'Jeremy'])
```

```
1 | ['xxmaj', 'my', 'name', 'is', 'xxmaj', 'jeremy']
```

用 `with` 标记 spacey 很慢，因为 spacey 做事情非常小心。spacey 有一个复杂的基于解析器的分词器，使用它会大大提高你的准确性，所以值得使用。幸运的是，标记化是令人尴尬的并行，因此我们可以使用多处理来加快速度。

```
1 | #export
2 | from spacy.symbols import ORTH
3 | from concurrent.futures import ProcessPoolExecutor
4 |
5 | def parallel(func, arr, max_workers=4):
6 |     if max_workers<2: results = list(progress_bar(map(func, enumerate(arr)),
7 |                                         total=len(arr)))
8 |     else:
9 |         with ProcessPoolExecutor(max_workers=max_workers) as ex:
10 |             return list(progress_bar(ex.map(func, enumerate(arr)),
11 |                                     total=len(arr)))
12 |     if any([o is not None for o in results]): return results
13 | #export
14 | class TokenizeProcessor(Processor):
15 |     def __init__(self, lang="en", chunksize=2000, pre_rules=None,
16 |                  post_rules=None, max_workers=4):
17 |         self.chunksize, self.max_workers = chunksize, max_workers
18 |         self.tokenizer = spacy.blank(lang).tokenizer
19 |         for w in default_spec_tok:
20 |             self.tokenizer.add_special_case(w, [{ORTH: w}])
21 |         self.pre_rules = default_pre_rules if pre_rules is None else
22 |                         pre_rules
23 |         self.post_rules = default_post_rules if post_rules is None else
24 |                         post_rules
25 |
26 |     def proc_chunk(self, args):
27 |         i,chunk = args
28 |         chunk = [compose(t, self.pre_rules) for t in chunk]
29 |         docs = [[d.text for d in doc] for doc in self.tokenizer.pipe(chunk)]
30 |         docs = [compose(t, self.post_rules) for t in docs]
31 |         return docs
32 |
33 |     def __call__(self, items):
34 |         toks = []
35 |         if isinstance(items[0], Path): items = [read_file(i) for i in items]
36 |         chunks = [items[i:i+self.chunksize] for i in (range(0, len(items),
37 |                                         self.chunksize))]
38 |         toks = parallel(self.proc_chunk, chunks,
39 |                         max_workers=self.max_workers)
40 |         return sum(toks, [])
41 |
42 |     def proc1(self, item): return self.proc_chunk([item])[0]
43 |
44 |     def deprocess(self, toks): return [self.deproc1(tok) for tok in toks]
45 |     def deproc1(self, tok):    return " ".join(tok)
```

```
1 | tp = TokenizeProcessor()
```

```
1 | txt[:250]
```

```
1 | # 下面是一个示例原始输入的样子:  
2 | 'Comedian Adam Sandler\'s last theatrical release "I Now Pronounce You Chuck  
and Larry" served as a loud and proud plea for tolerance of the gay  
community. The former "Saturday Night Live" funnyman\'s new movie "You Don\'t  
Mess with the Zohan" (*** out o'
```

```
1 | ' • '.join(tp(i1[:100])[0])[:400]
```

```
1 | # 这是标记化后的样子:  
2 | 'xxbos • xxmaj • comedian • xxmaj • adam • xxmaj • sandler • \'s • last •  
theatrical • release • " • i • xxmaj • now • xxmaj • pronounce • xxmaj • you  
• xxmaj • chuck • and • xxmaj • larry • " • served • as • a • loud • and •  
proud • plea • for • tolerance • of • the • gay • community • . • xxmaj • the  
• former • " • xxmaj • saturday • xxmaj • night • xxmaj • live • " • funnyman  
• \'s • new • movie • '
```

Numericalizing

一旦我们标记了我们的文本，我们就用一个单独的数字替换每个标记，这称为数字化。同样，我们使用处理器来执行此操作（与没有太大区别 `CategoryProcessor`）。

```
1 | #export  
2 | import collections  
3 |  
4 | class NumericalizeProcessor(Processor):  
5 |     def __init__(self, vocab=None, max_vocab=60000, min_freq=2):  
6 |         self.vocab, self.max_vocab, self.min_freq = vocab, max_vocab, min_freq  
7 |  
8 |     def __call__(self, items):  
9 |         #The vocab is defined on the first use.  
10 |         if self.vocab is None:  
11 |             freq = Counter(p for o in items for p in o)  
12 |             self.vocab = [o for o,c in freq.most_common(self.max_vocab) if c  
13 | >= self.min_freq]  
14 |             for o in reversed(default_spec_tok):  
15 |                 if o in self.vocab: self.vocab.remove(o)  
16 |                 self.vocab.insert(0, o)  
17 |             if getattr(self, 'otoi', None) is None:  
18 |                 self.otoi = collections.defaultdict(int,{v:k for k,v in  
19 | enumerate(self.vocab)})  
20 |             return [self.proc1(o) for o in items]  
21 |         def proc1(self, item): return [self.otoi[o] for o in item]  
22 |  
23 |         def deprocess(self, idxs):  
24 |             assert self.vocab is not None  
25 |             return [self.deproc1(idx) for idx in idxs]  
26 |         def deproc1(self, idx): return [self.vocab[i] for i in idx]
```

标记和数字化文本需要一段时间，所以最好做一次然后序列化它。

```
1 proc_tok,proc_num = TokenizeProcessor(max_workers=8),NumericalizeProcessor()  
2 %time ll = label_by_func(sd, lambda x: 0, proc_x = [proc_tok,proc_num])
```

S

```
1 ll.train.x_obj(0)  
2 'xxbos xxmaj comedian xxmaj adam xxmaj sandler \'s last theatrical release "  
i xxmaj now xxmaj pronounce xxmaj you xxmaj chuck and xxmaj la
```

S

```
1 pickle.dump(ll, open(path/'ld.pkl', 'wb'))  
2 ll = pickle.load(open(path/'ld.pkl', 'rb'))
```

Batching RNN 训练的批处理文本

批处理语言模型数据需要比处理图像数据更加小心。让我们通过一个示例文本进行批处理：

```
1 stream = """  
2 In this notebook, we will go back over the example of classifying movie  
reviews we studied in part 1 and dig deeper under the surface.  
3 First we will look at the processing steps necessary to convert text into  
numbers and how to customize it. By doing this, we'll have another example of  
the Processor used in the data block API.  
4 Then we will study how we build a language model and train it.\n  
5 """
```

让我们使用**6**的**批量大小**。这个序列恰好分成了6个长度为15的片段，所以我们的6个序列的长度是15。

1

S

1

S

1

S

1

结论

FastAI v3 还有两个教训：

- [第 13 课 \(2019\) - 深度学习的 Swift 基础知识](#)
- [第 14 课 \(2019\) - Swift: C 互操作; 协议; 把这一切放在一起](#)

我不打算在这两堂课上制作大量笔记，因为重点是使用 Swift 进行深度学习和 *Swift for Tensorflow*. 关于 Python 在深度学习方面的弱点以及被 Swift 或 Julia 等更好的语言破坏的空间，有很好的讨论。虽然这是一个非常有趣的话题，但对我来说，此时投入我的精力来写这件事并没有多大用处（这些笔记中的每一个都非常辛苦！），我想转移到其他项目上。这些讲座的代码量也非常大，并提供了 Swift 的编码介绍。当其他地方有无数更好的教程时，这里就不值得复制了。我也意识到这些讲座发表已经一年多了，一年后我不清楚 Swift for Tensorflow 是否在社区中获得了很大的动力。与此同时，Julia 似乎是 Python 的一个更有前途的竞争对手。

1 |

s

1 |

s

1 |

s

1 |

s

1 |

s

1 |

s

1 |

s

1 |

s

1 |

s

1 |

s

1 |

s

1 |

s

1 |

s

1 |

s

1 |

s

1 |

s

1 |

s

1 |

s

1 |