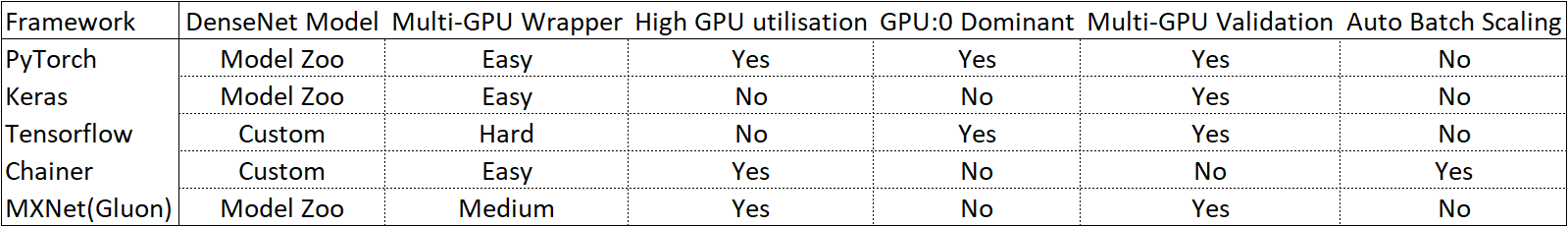
# 多GPU框架比较

[GitHub Link](https://github.com/ilkarman/DeepLearningFrameworks#2-training-time-densenet-121-on-chestxray---image-recognition-multi-gpu)

有关最新更新，请关注上的重新发布（即将出现）。

既然如此，就很难对每个人都深入研究。在这篇文章中，我采取了Tensorflow，Pythorch，MXNet，Keras，和Chainer和train amodel。



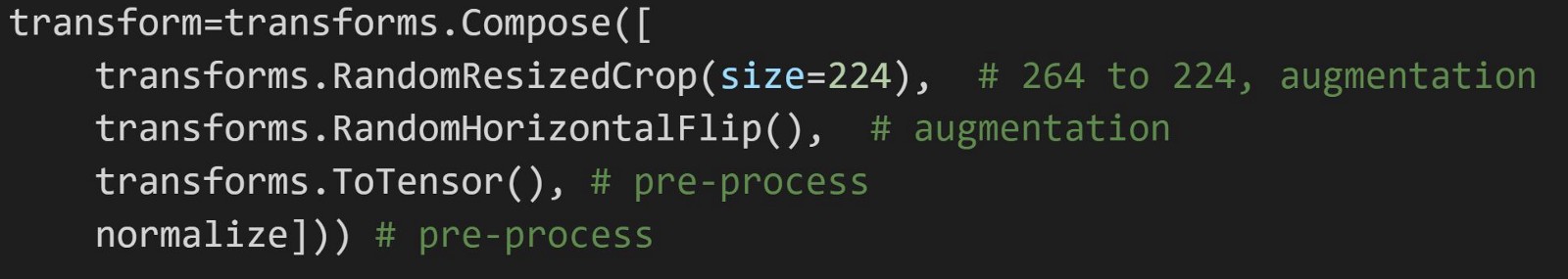
笔记本的例子，训练一个数据并行模型（DenseNet121在ImageNet上预先训练）跨越4个v100使用本地数据加载器在飞行中预处理图像，以及图像增强和验证。

### 数据

真正的数据集由12120个PNG的灰度胸片组成，调整到（264px，264px）并保存到磁盘。框架本地数据加载器用于动态加载、预处理数据和执行一些增强（随机水平翻转和随机裁剪到224px）。

合成数据集只是随机（224224，224）矩阵的numpy数组，加载到内存中，用于衡量数据加载（增强和验证阶段）对训练性能的影响程度。

最初我使用的是全尺寸图像（超过10000x1000px），但我注意到，即使使用异步数据加载程序，将图像大小调整到264也会限制训练（即使使用一个GPU）。因此，将图像大小调整为264并首先保存到磁盘。一个可能的进一步扩展可以将它们转换为jpeg，从而从&heoldsimd中受益。最近优化了用于加载和处理JPEG的库。数据加载管道可以用torchvision.transforms进行简洁的总结：



### 安装程序

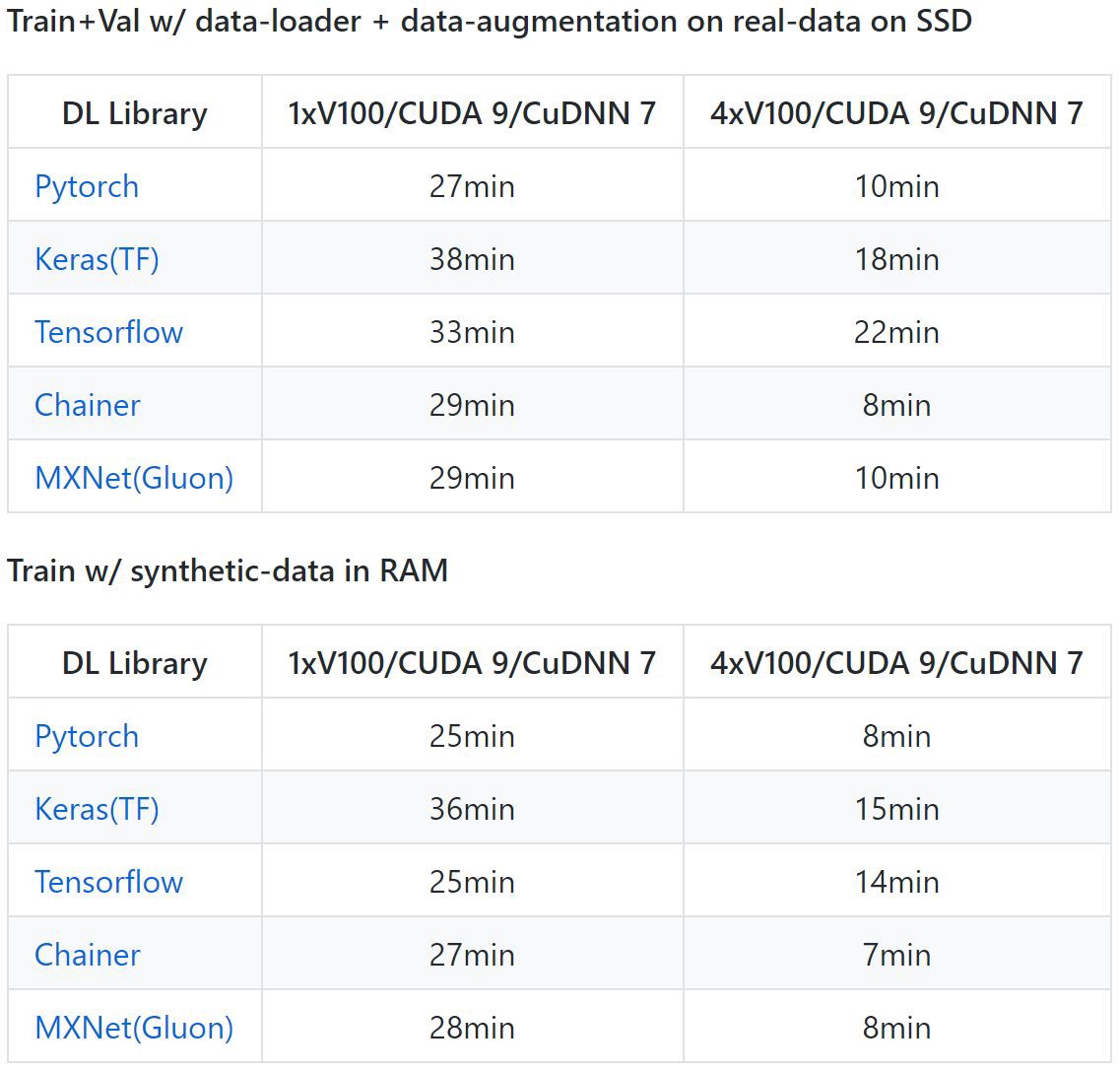
在这些笔记本中，我们使用（w/Imagenet weights），在最后一个完全连接的（14）层应用sigmoid激活和二进制交叉熵损失来解释非互斥标签。该模型被训练了5个阶段（测试数据集的AUC约为0.82），计时如下。

使用Pytorch、Keras、Tensorflow和MXNet，可以完全受益于数据并行模式，该模式涉及通过gpu数量手动增加批大小（有效地运行更大的批大小）。对于固定数量的时间段，这意味着GPU的梯度更新次数更少，因此我采用了一个简单的线性缩放规则，其中学习速率随着GPU数量的增加而增加。然而，Chainer是唯一一个根据gpu数量自动调整批处理大小并调整优化程序的框架。

### 总体结果

请按照GitHub链接获取更新结果

我计算出匹配相同数量的时期所需的时间（而不是相同的评估指标，这需要更长的时间）。



### 摘要

这个小比较的想法是使用相对高级（单节点）的数据并行包装器并评估性能。对于单节点，使用分布式包装器（如asor）的多gpu训练可能会导致更快的计时（每个进程将绑定到单个gpu并在单个节点上执行多进程分布式训练）。这是我们目前正在进行的工作，将成为下一篇博客文章的主题。

大多数框架执行数据并行训练的方式是：（i）跨gpu复制模型，（ii）将输入分散到多个gpu，（iii）输出[i]=副本，（i v）将输出[i]收集到主gpu，（v）计算损失和分割梯度，（vi）通过副本向后，最后（vii）在所有副本之间求和梯度。在Pythorch中，所有这些都是通过一个简单的呼叫来处理的：！

将输出收集到主gpu意味着计算损失的API调用对用户来说不那么复杂，只需：loss（output，target），因为所有东西都驻留在cuda:0（主gpu）上。然而，从gpu利用率图表中可以看出，这意味着损失计算是在gpu:0上完成的，并且通过不收集输出，而是将目标分散到多个gpu上并对每个gpu执行损失计算，有一个小的性能改进潜力。

ytorch团队在提供支持方面令人难以置信（我认为这是判断框架的一个关键标准），而且由于他们的帮助，您可以检查哪个覆盖了PyTorch的默认API来计算多个gpu上的损失，并缩短了几秒钟（以牺牲更多的代码为代价）。这也是直截了当的。

Py火炬本身是一个非常灵活的框架，它使得重写默认（更重要的标准）更容易。例如，根据我的经验，去掉ROI头只留下RPN（对一类对象检测有用）是很容易做到的。总的来说，我认为贡献者的支持和修改的容易程度比培训的速度更重要。

然而，当使用灵活的API（如Gluon和Py火炬）时，需要注意的是，不会意外地妨碍培训过程。至少应该避免CPU上的变量（可能是因为日志丢失）上的GPU上的变量（模型的输出）。而当真正追求性能时（并计算多个GPU上的损失，而不是收集到一个），重要的是将损失列表（每个GPU一个）记录到位于同一GPU上的变量（而不是将所有损失记录到GPU:0）。例如，对于胶子，我们会创建一个变量来记录损耗，如so:[mx.nd.zeros（（1），ctx=c）For c in ctx]和PyTorch:[torch.FloatTensor（1）.fill\_0.cuda（i）For i in range（GPU\_COUNT）]

Chainer使用了一个非常有趣的API，它更像是一个分布式包装器，并使用后台自动调整批处理大小，以获得一个非常令人印象深刻的gpu利用率图表。来自社区的支持（特别是通过slack）非常棒，notebook使用acaffection加载DenseNet（将批处理规范和缩放合并到单个批处理规范层，然后删除不必要的中间输出）。很高兴看到Chainer最终（优化）ONNX导入，让用户可以访问更广泛的模型zoo，并反对caffee功能。

下面是我对每个框架的更详细的总结，以及gpu利用率（顶部面板）和gpu内存利用率（底部面板）可视化。在大多数框架中，我们可以看到5个峰值在开始时（训练+对真实数据的验证），一个小峰值用于测试，然后在结束时持续运行5个阶段对合成数据（没有验证）。

GPU:0是绿线，默认情况下，应该很容易发现哪些框架收集输出到GPU:0来计算损失，以及哪些框架跨多个GPU进行损失计算。

同样有趣的是，在开始时看到小的上勾号，可能是自动调整的（选择最佳卷积前向算法）

我认为有两个因素可能有助于提高大多数框架的性能：

1. 将经过优化的二进制文件与pip安装捆绑在一起，以处理常见的功能，如图像加载和大小调整（以避免数据加载程序阻碍GPU）；例如，在PIL与OpenCV中调整图像大小的时间相当长。Soumith有一个，演示了如何安装pillow simd和turbojpeg；要从中受益，我应该将我的PNG转换为JPEG。
2. 优化训练后的验证周期（在训练结束前预取数据，以确保在交换内存时尽可能少的停机时间），因为在大量的时间段中，这会增加并自动使用所有GPU进行验证。您可以在“感谢亚当的帮助”中看到此预取的示例。

最后，对我来说一个有趣的发现是，最大化GPU利用并不一定转化为更好的性能：

* 增加用于验证的数据加载器的批处理大小（因为我们不存储渐变，我们有更多的内存，可以这样做）会降低性能，可能是因为gpu必须等待整个大批处理就绪后才能开始。解决此问题的一种可能方法（具有足够大的CPU内存）是在训练循环开始之前调用验证数据集上的iter（）（以便在后台进行预取），但是可能会干扰实际（训练）阶段的加载。
* 将数据加载器的num\_进程设置为cpu\_count（24）实际上是次优的，对于这个例子来说，6似乎是最快的（可能我的处理太轻，并且在头上创建了打开/关闭进程）。将这一点发挥到极致，对于合成数据（当不需要预处理时），只使用一个过程可以提供最快的时间。

### Pythorch（笔记本）

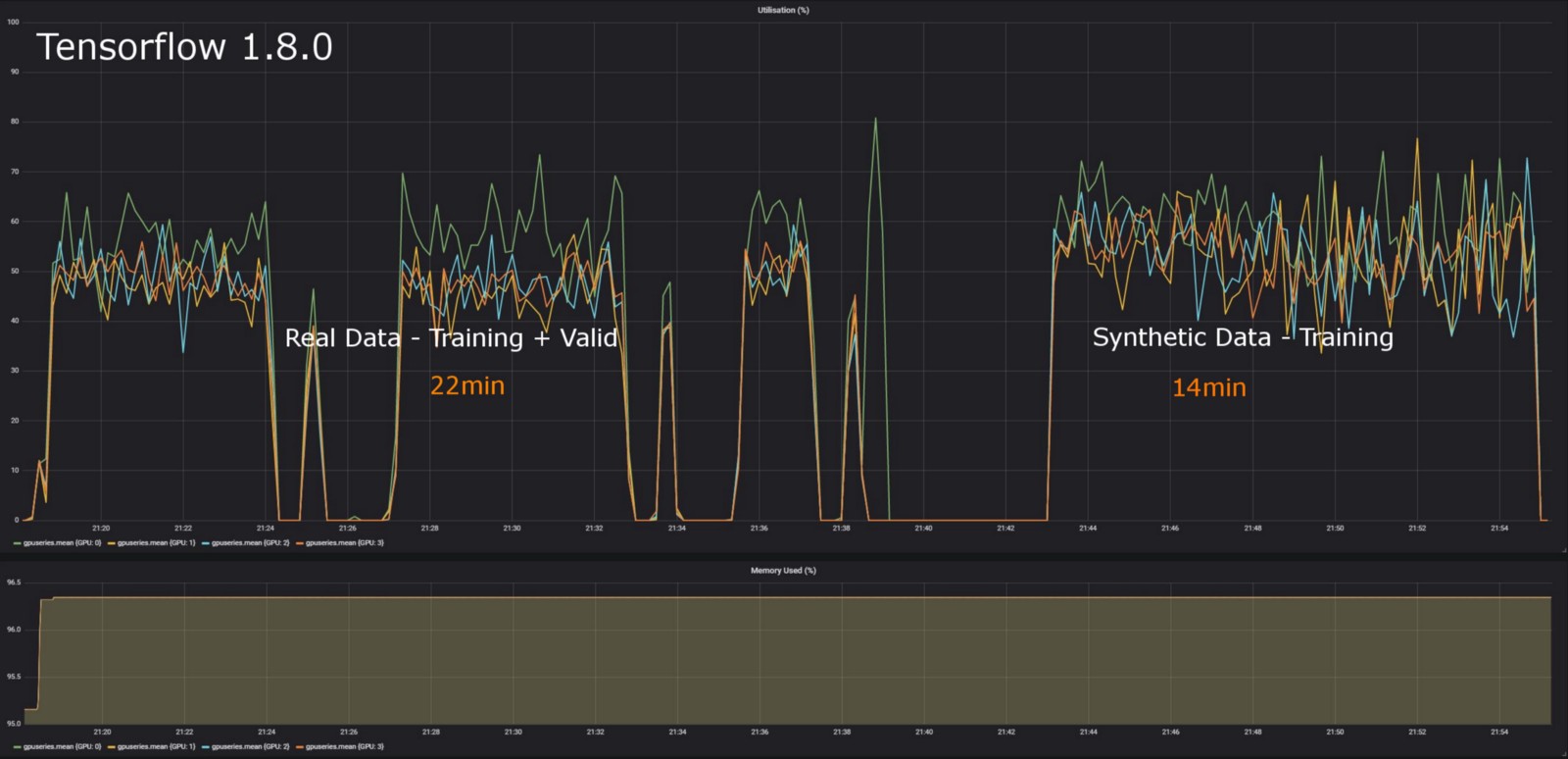
* PyTorch是最容易使用的框架，在实验结束时成为我的最爱。用户友好性似乎是免费的，因为它是最快的框架之一。
* GPU内存利用类似于Chainer和Gluon
* 由于上述原因，与胶子相比，Pythorch的gpu利用率图表更为gpu-0密集。
* PyTrac的训练代码是超级灵活的，但是这也为用户积累了更多的责任，而不是通过积累某些变量（损失或预测）以次优的方式来进行瓶颈训练。
* torchvision.transforms非常方便，拥有最多的转换；因此，看到MXNet使用MXNet.glion.data.vision.transforms跟随PyTorch非常高兴。它们的高级API看起来几乎相同。
* 奇怪的是，MXNet的数据加载器比PyTorch的实时数据加载器快了几分钟，尽管它拥有几乎相同的高级API，这表明它们已经优化了一些后端功能。
* 我没注意到使用torchvision有什么区别，也许我应该把PNG转换成JPEG？不幸的是，我找不到英特尔的IPP来让accdimage正常工作（上一次GitHub提交是2017年11月，所以我不确定它是否得到维护）。我觉得看到一些与pip捆绑在一起的常用函数（图像加载和调整大小）作为优化后的二进制文件安装，以加快数据加载程序的速度，这将是非常好的。

### Keras（笔记本）



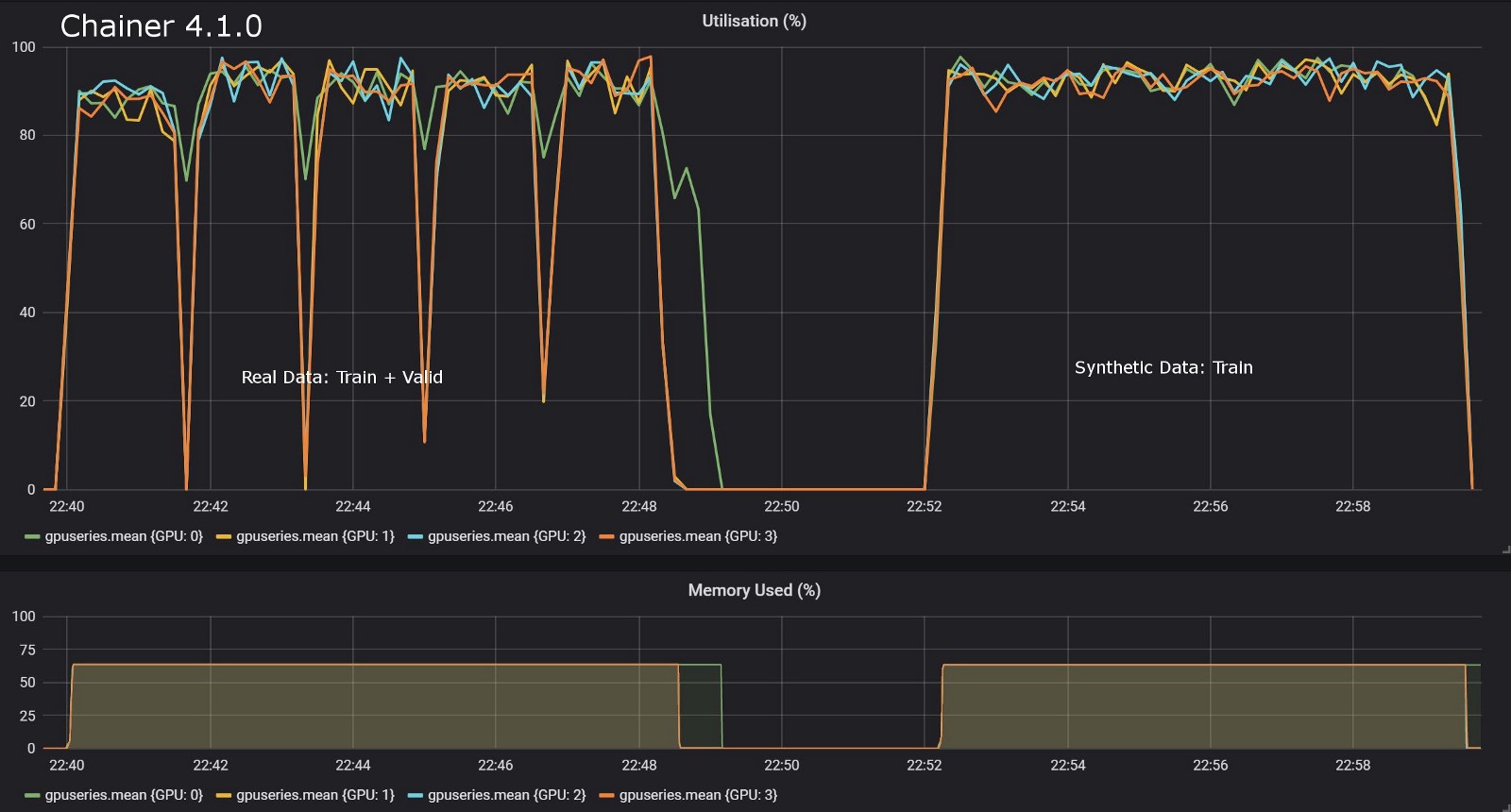
* 使用TF作为后端与Keras并行运行数据非常简单。这个速度几乎与我的原始TF实现相当（而在实际数据上，更快的数据加载程序稍有好处）。
* 然而，GPU利用率图似乎强调了GPU未被充分利用（例如，与Pythorch或MXNet相比）。
* 不幸的是，在编写本文时，在运行多个gpu时，ModelCheckpoint出现了问题，所以我对所有笔记本都忽略了这一点。
* Keras有一个非常快速的数据加载器
* 我发现ImageDataGenerator.flow\_from\_directory（）有点太严格了 - 子文件夹变成了标签，这成为多标签的问题，因为图像将在文件夹之间复制。有点老套，但我会在创建标签后覆盖它们。我觉得，允许生成器接受{file\_name:label，}字典将使整个过程更容易。
* 就在发表这篇文章之前，我注意到Keras 2.1.5将把图像发送到ImageDataGenerator.preprocessing\_function（）而不是CHW数组（keras2.1.4和之前的版本）。由于大多数预处理代码都是对数组进行操作的，这意味着我必须将图像转换为数组进行处理，然后将其转换回图像（稍后再转换为数组）。因此，我决定继续使用Keras 2.1.4 - 替代方法是继承Sequence类并编写自己的数据加载器。
* Keras的fit\_生成器允许用户在多处理和多线程之间进行选择。我发现后一种方法更适合这个实验，因为调整大小和随机裁剪不需要太多处理器（因为图像被调整到264并保存到磁盘）。我用未调整大小的图像（1000px+）做了一个实验，结果证明多个处理速度更快。因此，我假设如果处理很轻，并且工作负载主要是IO绑定的，那么多线程将更快。如果能得到更多的指导，那就太好了。

### Tensorflow（笔记本）



* 我找不到非常奇怪的anfor Tensorflow，因为它不是一个非常新的模型（可用于Keras TF），所以我使用了一个使用TF slim by pudae（非常有用）创建的版本。
* 很有可能某些优化版本的代码（可能使用不同的api）会更快，但是我没有得到任何关于如何改进笔记本电脑的帮助。
* 为了减少Tensorflow代码的冗长，tf.data.Dataset和tf.estimator.estimator高级api是一个非常受欢迎的（最近的）添加，我建议每个人都充分地研究它们。
* 我创建的TF数据加载器最终比Keras数据加载器慢。我找不到很多“好的”TF数据加载器的例子（可能是因为对tfrecords的关注？）所以不确定这是否是因为我的实现缺乏或者Keras真的更快。我注意到，使用TF 1.8，现在可以使用TF.contrib.data.prefetch\_to\_device（）预取到GPU内存。
* 一般来说，TF有许多不同的api可以用于相同的目的。也许他们调用同一个后端，而高级方法是为了向后兼容而保留的，但是我相信这会让用户很困惑，因为他们知道自己是在使用最佳函数还是应该弃用的东西。不过，我明白为什么这很难避免，因为TF需要比任何其他框架更多的低级锅炉板代码，而且这种代码会不断得到改进，因此会更改更多的功能（而不仅仅是使用这种代码的高级功能）。
* 创建一个多gpu示例被证明是非常困难的，并且需要编写大量的低级代码（例如平均梯度）。与编写高级API相比，编写大量低级代码的问题有两个方面：1）很难判断我的实现是否有效；2）随着版本的提高，我必须不断更改此代码以利用性能改进（而不是在后台自动执行）。提供一个“好”的例子可以解决第一个问题，而不是第二个问题。因此，将多个gpu合并为tf.estimator（）中的标志将是非常好的。由于使用Keras和raw TF对合成数据（避免使用数据加载器）进行了类似的计时，我想假设我正确地包装了多个gpu。
* 处理非常gpu:0重（如图所示），这可能在某个地方造成了瓶颈。
* 对于tf.slim模型，is\_训练标志似乎只影响批量正常化的完成方式。我认为在is\_training=False的情况下，TF将使用保存的均值和方差作为批处理范数，否则它将存活下来。在他们GitHub页面上的演练笔记本中，is\_training被设置为True进行验证，这就是我所遵循的。我认为应该改名为其他名称，例如batch\_norm\_use\_saved。
* 在编写本文时，我无法将验证设置为在每个纪元之后发生，而是tf.estimator.EvalSpec（）有一个trottle\_sec方法，该方法将在特定的秒数之后执行验证。这就是为什么GPU的使用率只有3个峰值，而不是5个（我猜不出时机，只有3个验证，而不是5个）。
* GPU利用率图表显示了培训和验证之间的大量停机时间，而其他框架则没有。这是因为TF会在训练结束后将模型保存到磁盘，然后从磁盘加载它以开始验证，然后再次恢复训练。我不知道如何避免这一点，因为这花费了上表中相当多的时间。
* 因此，我个人认为（作为一个局外人），Google希望人们使用Tensorflow通过Keras而不是原始TF（或者至少，这会取代TF.estimator？）。有太多的锅炉板代码不断更新，很难保持你的TF代码的最佳状态，除非你每天都在使用它。此外，还有更多的预训练模型可用于凯拉斯。对于我们正在研究的多节点分布式训练示例，我们决定完全不使用raw TF，而只使用。

### 链接器（笔记本）



* Chainer的gpu利用率是所有其他框架中最高和最一致的。它的平均速度也是最快的一次（在对默认函数进行了一些修改之后）。gpu的利用率并不是gpu:0占主导地位，而且似乎均匀地分散了负载。
* 在编写本文时，Chainer没有可用的预先训练模型DenseNet121，因此我找到了一个由shicai编写的，可以使用Chainer.links.caffe.caffefefunction加载的模型。
* Chainer的API可能是所有框架中最不同的，我非常喜欢使用它。我认为，如果Chainer改进了model zoo（和caffefefunction导入），或者最终完成了ONNX导入和导出，这将吸引更多的人使用这个框架。我发现这个库对于图像增强的变换也非常有用。
* 要加载模型，我必须编写一个方法，将批处理规范化epsilonparam截断为1e-5，这有点奇怪，因为prototxt已经是1e-5，我想知道这是否是某种导入错误。
* CaffeFunction方法非常通用，可以保存网络中所有层的结果（不只是最后一层），这会使内存膨胀很多，意味着我只能使用一批32个。我创建了另一个方法来覆盖它，该方法只存储DenseNet所需的层（请记住，组之间存在一些连接），这减少了内存并允许我增加批处理大小。对函数进行修改（由社区提供）将批处理规范和缩放合并到一个批处理规范层中，显著提高了性能（相对于默认值）
* 奇怪的是，使用multi-gpu的测试AUC指标比所有其他框架的都要低得多，我不知道为什么。采用同样的线性缩放规则对这一点的危害更大，所以在假设乐观者是自动缩放的情况下，我没有改变学习率。我需要进一步调查。
* 从上图（和代码中）我们可以看到，只有一个GPU用于验证，如果所有四个都可以使用（类似于PyTorch），那么这可能会进一步提高性能
* Chainer使用NCCL进行单节点多gpu通信（chainermn用于跨节点的分布式Chainer）。有关更多详细信息，请参阅此。

### MXNet胶子（笔记本）



* 胶子API与PyTorch有很强的相似性，因此使用起来非常方便和友好。
* GPU利用率图表看起来非常好（尤其是在培训+验证期间的下降峰值比其他框架小得多，这可能是数据加载器的一个优势）。
* Gluon需要稍加修改，以找出在使用打印丢失方法阻塞之前异步发送的最佳批数，否则将收到内存不足错误。

### 致谢

非常感谢您创建和运行Gluon笔记本，Soumith Chintala&Adam Paszke&Teng Li提供Pythorch支持，gpu监视器和Chainer相关帮助