ElasticDL：同时提升集群利用率和研发效率的分布式深度学习框架

ElasticDL 是一个基于 TensorFlow 2.x 和 Kubernetes 的分布式深度学习编程框架。2019 年秋天的 Google Developer Day 活动中 来自蚂蚁金服的 ElasticDL 团队展示了 ElasticDL 的第一个开源版本。本文更新这大半年来 ElasticDL 项目的进展，尤其是性能优化和业务落地。

ElasticDL 的首要设计意图是简化分布式编程。它允许用户只提供用 TensorFlow 2.0 API 描述的模型，而不需要用户写分布式训练过程代码。用户的模型定义只要能在本地调通，即可在分布式环境下用大规模数据训练模型，从而提升研发效率。

同时，ElasticDL 提供的弹性调度的能力在实践中可以让集群的利用高达 90%。当集群资源不足时，一个训练作业里的进程减少；当其他作业结束释放资源后，进程数量随之增加。这样的做法比 TensorFlow distribution strategy 以及 TorchElastic 专注容错（进程减少的情况下作业不失败，但不会增加进程数量）更进一步。并且，因为 ElasticDL 作业容忍变化的 worker 数量，所以每个作业的启动都不必等待集群有足够的资源，而是可以见缝插针的尽早开始训练，从而缩短等待作业启动的时间，让研发人员可以尽快看到第一个迭代的结果，万一分布式训练有问题，也能尽早发现，从而进一步提升了研发效率。

简化分布式深度学习编程

为了从海量数据中学习规律，我们需要编写分布式深度学习程序来完成训练任务。这在工业场景中尤为常见。

可分布式深度学习程序的编写很难 —— 编程者既要了解深度学习，也要了解分布式系统开发。在一个分布式深度学习系统中，需要启动和监控若干个 workers。因为既要拆分训练数据给 workers，还要综合各个 worker 算出的 gradients 来更新模型，所以涉及通信（communication）和同步（synchronization）。此外，当 worker 数目很多时，作业在执行过程中有 worker 挂掉的概率也会变得很大。如果一个 worker 挂掉，则整个作业重启或者恢复到最近的 checkpoint (fault recovery)，那么重启之后可能又会有 worker 挂掉导致重启，于是作业不断陷入重启和恢复，永远也无法完成。这进一步要求编程者具备设计容错（fault tolerance）系统的能力。其实不仅分布式深度学习，其他分布式机器学习程序、分布式离线和在线数据处理程序等各种分布式程序的写作，都对编程者有类似上述要求。

一个常见的解决思路是为特定类型的作业提供分布式编程框架，让用户只需要完形填空一样补上业务逻辑，而分布式计算（包括通信、同步、和容错）都由框架的代码来完成。一个典型的例子是离线数据处理程序用 MapReduce 框架来写。不管是 Google MapReduce 还是 Hadoop MapReduce，用户基本都只需填写 map 和 reduce 两个函数的实现即可。类似的，在线数据流系统基于 Storm 和 Flink 来写，用户只需提供 bolts 和 nuts 这样的业务逻辑定义。

在 ElasticDL 之前，蚂蚁金服的同事们使用过多种框架和类似框架的高层 API。这些方案大都基于 TensorFlow 和 Kubernetes。

1. TensorFlow Estimator 作为构建在 TensorFlow 之上的一层 API，允许用户只需定义模型，而训练过程封装在一个函数调用里。利用 Kubeflow 提供的 TF operator，我们可以将该训练过程以分布式作业的方式启动在 Kubernetes 上。这个方案的局限是：它仅支持 TensorFlow 的 graph mode，不支持 eager execution；而 eager execution 可以大幅简化调试，尤其方便跟踪网络各层输出。
2. Keras API 支持 TensorFlow 2.x 和 eager execution。目前 TensorFlow 2.x Keras API 还暂不支持 ParameterServer 分布式策略，对 AllReduce 分布式策略提供了实验性的支持。

3. Horovod 对用户代码有侵入性，用户除了必须熟悉 TensorFlow API 之外，还需学习 Horovod API。

以上三个方案的共同局限是，虽然具备一定的容错能力，不过不支持弹性调度。而且它们都依赖部署 Kubernetes operator，了解 Kubernetes 对 AI 专家来说颇有挑战。

方案	模型定义方式	分布式执行机制
Estimator	TensorFlow Estimator API	Kubeflow TF-operator
Keras	TensorFlow Keras API	Kubeflow TF-operator
Horovod	Horovod with TensorFlow	Kubeflow MPI-operator
ElasticDL	TensorFlow Keras API	ElasticDL master process per job

针对这些局限，我们设计和开发了 ElasticDL 分布式计算框架。用户定义可以用 TensorFlow 2.x 的 Keras API 来定义模型。并且，分布式执行不要求 Kubernetes 集群有任何特殊配置，而是利用每个作业里的 master 进程来协调训练数据分配、通信、同步、和容错 —— 这也是 ElasticDL 除了容错，支持弹性调度的原因。

基于 ElasticDL 框架的编程

就像 MapReduce 框架中只需要用户完形填空两个函数：map 和 reduce，ElasticDL 需要用户填写 forward、loss、optimizer、feed 函数。其中 forward 定义深度学习的前向计算过程 (forward pass)，ElasticDL 会调用 TensorFlow eager execution 的 GradientTape 机制来自动推导对应的后向计算过程（backward pass）；loss 函数返回模型训练时使用的损失函数；optimizer 函数返回模型训练时使用的优化器；feed 定制化训练数据到 TensorFlow 模型输入（tensors）的转换过程。

所有这些函数的编程只需要了解 TensorFlow API，不需要对分布式训练有任何背景知识。写完之后，用户可以在单机上用小数据做调试验证。如果通过，可以不做任何代码修改就提交到 Kubernetes 集群上做分布式的容错的大规模训练。

不同于 Kubeflow/TF-operator 给每个集群部署一个 Kubernetes Operator 的方式，ElasticDL 为每个作业引入一个 master 进程。通过调用 Kubernetes API，master 进程了解集群情况；同时，作为作业的一部分，master 还了解深度学习作业的特点 —— 包括利用 Python inspection 机制了解上述各个函数的特点，其中调用的 API 函数等。所以，master 有非常充分的信息来做更优的调度。比如 master 可以请 Kubernetes 把两个 worker 启动在同一台物理机上，共用一个 GPU —— 当一个 worker 读数据的时候，请另外一个 worker 来做计算，从而始终保持较高的 GPU 利用率。

一个例子

我们用一个 MNIST 手写数字识别的例子来说明。

```
def forward():
    inputs = tf.keras.Input(shape=(28, 28), name="image")
    x = tf.keras.layers.Reshape((28, 28, 1))(inputs)
    x = tf.keras.layers.Conv2D(32, kernel_size=(3, 3), activation="relu")(x)
    x = tf.keras.layers.Conv2D(64, kernel_size=(3, 3), activation="relu")(x)
    x = tf.keras.layers.BatchNormalization()(x)
    x = tf.keras.layers.MaxPooling2D(pool_size=(2, 2))(x)
    x = tf.keras.layers.Dropout(0.25)(x)
    x = tf.keras.layers.Flatten()(x)
```

```
    outputs = tf.keras.layers.Dense(10)(x)
    return tf.keras.Model(inputs=inputs, outputs=outputs, name="mnist_model")
```

除了模型定义之外，用户还需要指定 feed, loss，optimizer 函数。

```
def loss(labels, predictions):
    labels = tf.reshape(labels, [-1])
    return tf.reduce_mean(
        input_tensor=tf.nn.sparse_softmax_cross_entropy_with_logits(
            logits=predictions, labels=labels
        )
    )

def optimizer(lr=0.1):
    return tf.optimizers.SGD(lr)

def feed(dataset, mode, _):
    def _parse_data(record):
        if mode == Mode.PREDICTION:
            feature_description = {
                "image": tf.io.FixedLenFeature([28, 28], tf.float32)
            }
        else:
            feature_description = {
                "image": tf.io.FixedLenFeature([28, 28], tf.float32),
                "label": tf.io.FixedLenFeature([1], tf.int64),
            }
        r = tf.io.parse_single_example(record, feature_description)
        features = {
            "image": tf.math.divide(tf.cast(r["image"], tf.float32), 255.0)
        }
        if mode == Mode.PREDICTION:
            return features
        else:
            return features, tf.cast(r["label"], tf.int32)

    dataset = dataset.map(_parse_data)

    if mode == Mode.TRAINING:
        dataset = dataset.shuffle(buffer_size=1024)
    return dataset
```

上述每个函数都很容易做单独测试(unit test)。而且，利用 TensorFlow 2.x eager execution，上述函数很容易 log 每一层的输出。基于个特点，ElasticDL worker 在调用 forward 函数的时候，可以打印中间结果，便于调试和复现问题。

ElasticDL 的弹性训练过程

给定上述模型定义，ElasticDL 的 master 进程按照 asynchronous 或者 synchronous SGD 方法，协调 workers 来做训练。当使用 asynchronous SGD 方法时，master 会启动一个高性能的 parameter server，供各个 workers 使用。当使用 synchronous SGD

时，ElasticDL 使用和才云科技合作研发的一个 Kubernetes-native 的 fault-tolerable AllReduce 实现 FTlib。

Master 负责动态数据划分

弹性训练过程的一个容易被忽略的前提是动态数据划分（dynamic data partitioning）。在用 MPI 写分布式程序的时候，因为作业中进程数量是恒定的，所以经常采用静态数据划分的做法 —— 在训练之前把训练数据预先分成 N 个文件，对应作业中的 N 个 worker 进程。这个做法在弹性调度的时候就失效了 —— 因为弹性调度时，作业中的进程数量是可变的。为此，需要实现动态数据划分。

ElasticDL 的动态数据划分是基于索引的。ElasticDL 要求训练数据是一个或者多个 RecordIO 格式的文件，或者是 MaxCompute 数据库系统中的表（table）。这两种数据源都允许 master 进程在开始训练之前，在基本存储单元（block）间快速跳跃着扫描数据，把数据分成小段，称之为任务（task）。每个 task 包括的内容如下：

1. 文件名或者表名，
2. 第一条记录相对于文件（或者表）开始处的偏移（offset），
3. 这个 task 里的总记录数。

扫描结果是很多 tasks，master 把这些 tasks 放进一个 TODO 队列里。这个队列不一定需要是 master 进程里的数据结构，可以是放在 etcd 里的 —— 因为 etcd 是不死的，所以 master 即使被高优先级作业抢占了，这个信息也不会丢失；可以通过在资源富余时重启 master 进程来恢复作业状态。

扫描和划分数据的同时，master 开始请 Kubernetes 启动 workers，总数不超过用户指定的数量 N（最大并发度）。每当一个 worker 启动起来了，master 会收到 Kubernetes 发来的通知；master 在一个 etcd 数据结构里记录“活着”的 workers。

扫描和划分数据结束之后，master 就依次从 TODO 队列里取出 task，通过 gRPC 发给某一个活着的 worker，同时 master 把这个 task 挪进 DOING 队列里。接收到 task 的 worker 负责打开文件（或者表），并且从指定的 offset 开始依次读取记录，并且更新本地模型。根据用户选择的 asynchronous 或者 synchronous 算法，workers 会通过调用 parameter server 或者 AllReduce 来协调更新全局模型。

当一个 worker 处理完了接收到的 task，它通过 gRPC 返回一个表示成功的标记；master 就把这个 task 从 DOING 队列挪到 DONE 队列了。当所有 task 都从 TODO 挪进了 DONE，则说明一个 epoch 完成了。

如果一个 worker 失败了（比如被更高优先级作业抢占了），则 master 的 gRPC call 会 timeout；此时，master 把对应的 task 从 DOING 队列挪回 TODO 队列了。下一次有 worker 完成 task 时，master 会把这个 task 再发出去。这里有一个细节：有的 task 可能被某个 worker 使用了一部分，也因此影响到了模型更新；此时 worker 被抢占，那么这部分已经被处理的数据会因为 task 的下一次分发，被重复使用。不过这个并不影响机器学习训练要求数据统计一致性的假设。而且其他动态数据划分方法造成的数据复用情况可能更严重。

Woker 调用 TensorFlow Eager Execution

ElasticDL worker 接收到的一个 task 通常包括多个 minibatches。对于每个 task，worker 打开对应的文件或者表，随后做如下操作：

1. 读取一个 mini-batch 的训练数据。
2. 用本地模型（local model）作为参数调用用户定义的 forward 函数以计算 cost。如果模型很大，则部分参数可能来自于 parameter server。

3. 给定 cost，worker 利用 TensorFlow eager execution 的 GradientTape 机制，进行 backward 计算，得到梯度（gradient）。
4. 如果是 synchronous SGD，此时 worker 调用 AllReduce 实现 FTlib 来同步 gradients 并且更新模型。如果是 asynchronous SGD，worker 不定时的向 parameter server 上传 gradients，也不定时地从 parameter server 获取全局模型参数。

高效训练的优化

相对于 2019 年秋季 ElasticDL 在 Google Developer Day 上亮相时的状态，最近几个月 ElasticDL 项目针对性能优化做了很多工作。当时 ElasticDL 使用 Redis 作为 parameter server。现在有了自己的用 Go 语言写的 parameter server。相对于 Redis，ElasticDL parameter server 可以做一些深度学习计算，从而减少 worker 和 parameter server 之间通信的次数。

这个变化和其他优化工作一起让同样的训练作业，总体训练时间下降了约 13 倍。最近一个基于 DeepFM 模型的试验展示，用两个 parameter server 进程和四个 workers 进程来训练，10 个 epochs 的总体时间从 1350 秒（ElasticDL 的 2019 年 9 月版本）下降到 106 秒（2020 年 2 月版本）。这些优化策略包括：

- 在 parameter server 上惰性初始化 (lazy initialize) embedding vectors —— 在使用到 vector 的时候才初始化。
- 把一个 embedding table 拆分到多个 parameter server 进程里以均衡存储与通信负载。
- worker 从 PS 请求 embedding vectors 时，先滤除重复的 embedding ID，只取回不同 ID 的 vectors，从而减少通信量。
- worker 向 PS 发送梯度时，先把相同 ID 的梯度进行合并（调用 TensorFlow 的 embedding vector combinanation 函数），从而减少通信量。

弹性调度提升集群利用率

ElasticDL 实现的弹性调度和刚性调度（gang scheduling） 是对应的。刚性调度的简洁不求甚解的描述是：一个作业里的 n 个进程，运行时如果有一个进程挂了（比如被更高优先级的作业抢占了资源），则整个作业挂掉。等资源足够再启动所有的 n 个进程了，则可以重启（或者从最近的 checkpoint 恢复）。

上文提到的几种分布式运行 TensorFlow 作业的方式都使用了 Kubeflow 项目提供的 Kubernetes operators，支持在 Kubernetes 上分布式地运行 TensorFlow 作业。因为 TensorFlow runtime 目前支持一定程度的容错，所以作业执行过程中，如果有一些 workers 挂了，剩下的可以继续。不过不支持因为日后资源富余，恢复 workers 数量。Facebook 开源的 TorchElastic 也是类似的扩展，支持分布式 PyTorch 作业。它号称 Elastic，其实是 job 失败后从 checkpoint 重启。XGBoost、MXNet 社区也习惯于复用 Kubeflow 的 Kubernetes operator。用 MPI 写的程序也可以用 Kubeflow 拉起。

而弹性调度（elastic scheduling）实现的是训练作业运行过程中，进程数量的变化不影响作业进行。具体的说，如果一个或者几个进程被高优先级的作业抢占，剩下的进程不受影响地继续进行。如果将来资源丰沛了，系统可以加几个进程，此时作业仍然不受影响地继续运行。

上文简述了 ElasticDL 实现弹性调度的机制，包括动态数据分配以及由 master 来启动、监控、和管理 workers，而不依赖 Kubernetes operator。本节展示三个 benchmark 试验，帮助大家直观地了解 ElasticDL 对集群利用率和研发效率的同时提升。

实验一：多个 AI 训练作业并发

考虑两个 AI 训练作业需要的资源总和略超过集群的情况：如果没有 elastic scheduling，则两个作业顺序执行。第二个作业的发起人需要等很久 —— 用户体验不好。并且任何时刻只有一个作业在运行 —— 集群资源用不满。而如果有 elastic scheduling，则两个作业并发执行，虽然后启动的作业拿不到期待的全部资源，但是也马上就开始执行了 —— 用户体验好，而且因为作业并发集群被用满。

我们做了一个实验来验证上述好处。这个实验可以在 ASI 集群和开源 Kubernetes 集群上复现。实验结果如下图。

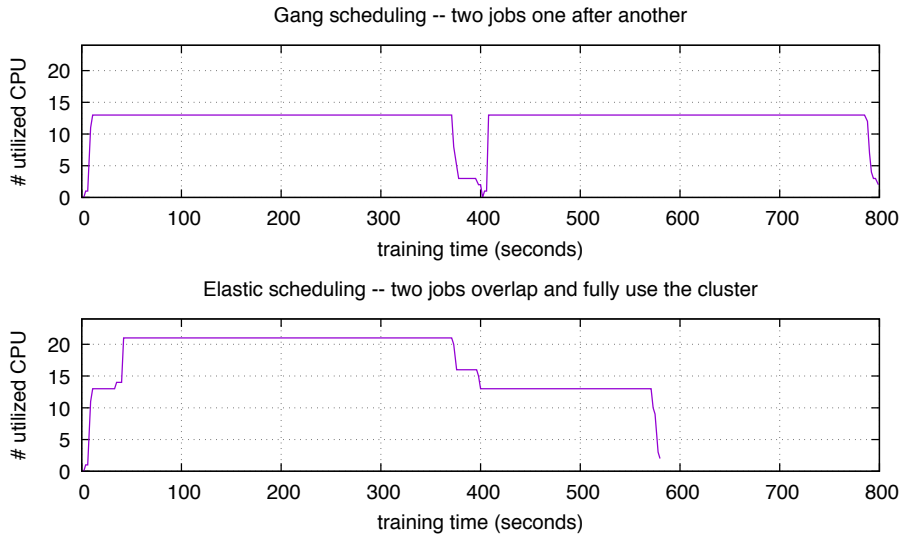

Figure 1: overlap jobs

上图对应的实验里，我们用 gang scheduling 的方式提交了两个训练作业，每个作业都需要 13 个 CPU。而 Google Cloud 上租用的实验集群总 CPU 数是 24，不足同时运行两个作业，所以依次运行它们。可以看到第一个作业在 395 秒时结束。随后集群花了一点时间调度，然后开始运行第二个作业，直到 795 秒时结束。

下图对应的实验里，我们用 ElasticDL 来执行同样的两个训练作业。第一个作业提交之后的 30 秒，我们提交了第二个作业。第二个作业马上就开始运行，用满了集群剩下的资源，而不需要等到第一个作业结束。在 395 秒时，第一个作业结束。随后，在 580 秒时，第二个作业也结束了。因为弹性调度，使得两个作业尽量同时运行，所以总结束时间比也上图要早。

总结：

- 用户等待作业启动时间几乎是 0。这对于 AI 工作很重要，因为用户最关注的是第一个迭代尽快开始—— 如果第一个迭代失败了，很可能是用户程序的 bug。另外，深度学习模型往往需要手动调优，学习率、optimizer、activation 等配置如果不合理，往往在前几个迭代就能发现；因此第一个迭代能立刻开始，对模型调优的工作效率提高有很大帮助。
- 集群利用率高。第二个实验（elastic scheduling）执行期间，有一段时间集群利用率是 100%；其他时间也不低于第一个实验（gang scheduling）。
- 作业完成更快。第二个试验里，两个作业用了约 580 秒；第一个实验里需要约 795 秒。

实验二：AI 作业和在线服务混布

运行各种在线服务的生产集群，通常需要留出余量资源，以应付突然增长的用户请求量。我们希望利用这些“余量”来做 AI 训练，从而提升集群利用率。下面实验验证：通过用较低优先级运行 ElasticDL 训练作业，在用户请求增加的时候，Kubernetes 自动扩容在线服务（NGINX）；此时 ElasticDL 作业自动释放资源，配合在线服务的扩容。当流量高峰过去之后，Kubernetes 自动缩容 NGINX 服务，此时，ElasticDL 自动利用释放的资源。

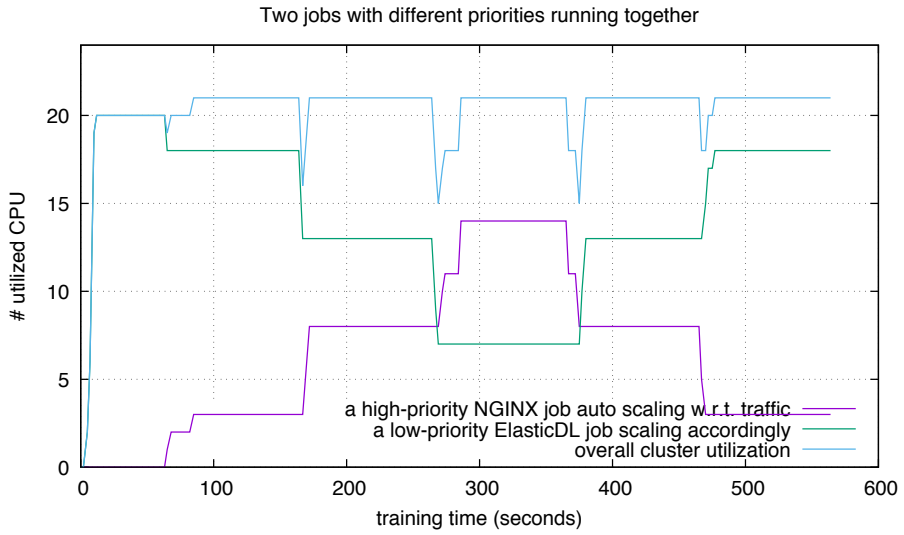

Figure 2: auto react

图中紫色曲线是 NGINX 服务使用的 CPU 数量，随用户请求数量变化。绿色曲线是 ElasticDL 训练作业使用的 CPU 数量，随 NGINX 的资源需求自动变化。蓝色曲线是集群的总体资源利用率 —— 保持在 90% 以上。

实验三：训练时更改 worker 数量不影响收敛性

有用户担心训练过程中 worker 的数量发生变化，会导致不收敛。实际情况下从未发生这类问题。用 ElasticDL 和用 gang scheduling 分别训练 Wide & Deep model 和 xDeepFM model，收敛曲线如下：

可以看到，采用 gang scheduling 持续用 4 个或者 8 个 workers，和用 ElasticDL 并且 worker 数量在 4 到 8 之间变化，得到的收敛曲线很难分辨。差别在自然误差范围之内。

总结

蚂蚁金服从事的金融行业涉及支付、微贷、和保险等业务。和搜索、广告、推荐不同，金融业务的流程要复杂得多 —— 包括对用户信用的预判以及和其他金融机构的联动 —— 每一个用户请求对应很多处理步骤；而搜索、广告、推荐业务里针对每个用户请求的 AI 处理步骤少得多。行业特点导致蚂蚁金服要训练的模型的类型繁多，呈现更长尾的特点。也对工具提升研发效率提出了高要求。ElasticDL 正是针对这些特点设计的。

同时，对集群的利用率提升是各行各业都关注的。在很多公司和行业，AI 集群的利用率通常在 30% 以下。当通过全面实现弹性调度，把集群利用率提升到 90% 左右时，相当于空手套白狼地

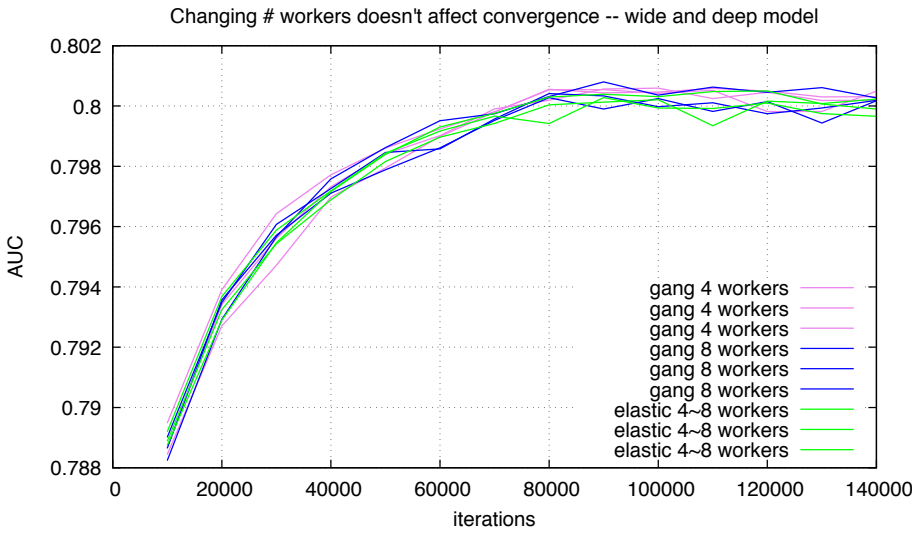

Figure 3: wide-n-deep training coverges

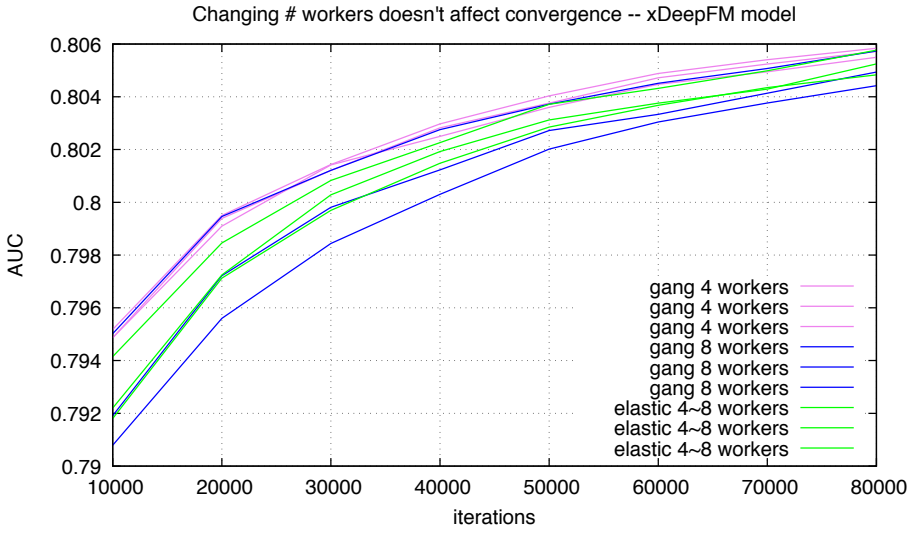

Figure 4: xdeepfm training converges

把集群规模扩大了为原来的三倍多。因此节省的硬件投资可能高达数千万甚至数亿元人民币。

ElasticDL 的设计和实现依托了 TensorFlow 2.x 提供的高效率的模型描述 API。也依赖了 TensorFlow eager execution 提供的 GradientTape 机制 —— 使得 ElasticDL 可以在不改变 TensorFlow runtime 的情况下，结合 Kubernetes 实现彻底的弹性调度（进程数可增也可减），从而实现了减少作业启动的等待时间，提升集群利用率，和提升研发效率的效果。

目前 ElasticDL 在阿里系结合 PAI 平台在推广。PAI 平台提供的拖拽式编程模式进一步降低了端到端机器学习流程的研发门槛。希望接下来 ElasticDL 团队可以有更多结合业务实践的分享。