AIACC-Training 1.5.0 使用文档

1. 概述

AIACC-Training（曾用名 Ali-perseus / Perseus-Training）是阿里云神龙计算加速团队基于阿里云IAAS资源层上研发和维护的高效人工智能分布式训练加速组件，旨在实现开源兼容和客户无感地加速阿里云上广大用户的分布式训练任务。

- AIACC-Training 支持PyTorch / TensorFlow / MXNet / Caffe 四种目前市面上主流的AI框架层的分布式训练任务加速。
- 在接口层面上，AIACC-Training兼容PyTorch DDP 和 Horovod的API，对于原生的分布式训练做到无感的性能加速。
- 在底层加速上，在底层加速上，针对阿里云网络基础设施和AI 数据并行分布式训练的特性，AIACC-Training的优化带来显著性能的提升。

AIACC-Training详细架构图如下：

AIACC-Training（AIACC训练加速）支持的加速特性包括但不限于以下功能：

- 梯度融合通信的自适应多流，自适应梯度融合，对带宽密集的网络模型，性能提升显著（50%至300%的提升）
- 高度优化的在线/离线梯度协商机制，将大规模节点下梯度协商的开销降低一到两个数量级
- 分级的Allreduce，支持FP16梯度压缩及混合精度压缩
- 支持Gossip风格梯度压缩技术
- 支持Multistep梯度通信优化
- 针对RDMA/eRDMA网络的深度优化
- 针对MXNet的API扩展，支持insighface类型的数据+模型并行和syncbn operator 增强
- 支持GroupComm的分组通信算子，可以方便快速构建数据并行+模型并行的复杂通信组合训练

基于阿里云普通的32Gbps VPC 网络，AIACC-Training打破了Dawnbench 分布式训练的性能和成本记录

斯坦福Dawnbench Imagenet训练速度世界第一

斯坦福Dawnbench Imagenet训练成本世界第一

本节为AIACC–Training 1.5 的概述。本文档的第二节介绍了AIACC–Training 在PyTorch / TensorFlow /MXNet / Caffe的使用步骤。第三节介绍在我们支持的客户在阿里云的GPU实例上运行分布式训练的典型应用及AIACC–Trainig带来的性能提升。第四节为附录，介绍了AIACC–Training使用过程中可选的环境变量、常用问题和针对PyTorch / TensorFlow / MXNet的常用的适配方式

2. 软件使用

AIACC–Training 支持 PyTorch / TensorFlow / MXNet / Caffe等主流人工智能框架的分布式训练。在接口层面上，目前AIACC–Training兼容PyTorch DDP 和 Horovod的API，对于原生使用上述分布式训练方法的分布式训练代码可以做到无感的性能加速。以下部分将介绍AIACC–Training 1.5.0的软件环境部署方式及针对 PyTorch / TensorFlow / MXNet / Caffe 等框架介绍使用AIACC–Training 进行加速的具体步骤

2.0 环境部署

前提条件

已创建阿里云GPU实例，且GPU实例满足以下要求：

- 操作系统为Alibaba Cloud Linux 、CentOS7.x或Ubuntu16.04以上
- 已安装NVDIA Driver和CUDA10.0或以上版本

支持的深度学习框架

AIACC–Training提供对应不同深度学习框架版本的软件包，，目前支持的框架版本如下表:

CUDA版本	框架类型	框架版本
10.0	PyTorch	"1.2.0", "1.3.0"
	TensorFlow	"1.14.0", "1.15.0", "2.0.0"
	MXNet	"1.4.1", "1.5.0", "1.7.0"
10.1	PyTorch	"1.6.0", "1.5.1", "1.4.0"
	TensorFlow	"2.1.0", "2.2.0", "2.3.0"
	MXNet	"1.4.1", "1.5.0", "1.6.0", "1.7.0", "1.9.0"
10.2	PyTorch	"1.5.1", "1.6.0", "1.8.0", "1.8.2", "1.9.0", "1.10.0"
	MXNet	"1.9.0"
11.0	PyTorch	"1.7.0", "1.7.1"
	TensorFlow	"2.4.0"
	MXNet	"1.9.0"

备注：

- 目前TensorFlow和MXNet只支持Python3.6
- PyTorch支持Python3.6~3.9，具体与PyTorch官网提供的版本支持一致
- 如您使用的框架版本不在支持范围内，请提交工单获得支持。

使用方式

AIACC-Training提供多种安装方式：

方式一：在已有的AI软件环境中使用AIACC-Training

若您已布置好深度学习训练环境，可以通过以下方式安装AIACC-Training。
安装前除了需要满足前提条件以外，还需要以下的环境依赖：

- 已安装Python3和对应的pip
- 已安装深度学习框架（PyTorch、TensorFlow或MXNet）

1. 自动安装AIACC-Training软件包（推荐）

AIACC-Training提供对应框架版本的Python软件包，可以通过一键脚本自动安装：

Shell 复制代码

```
wget https://ali-perseus-release.oss-cn-huhehaote.aliyuncs.com/install_AIACC-Training.sh && bash install_AIACC-Training.sh
```

注意：

- 脚本默认是使用 `python3` 作为Python的命令。若您使用其他的Python命令，可在脚本的最后添加，比如使用 `python` 作为Python的命令，可通过以下命令一键安装：

Shell 复制代码

```
wget https://ali-perseus-release.oss-cn-huhehaote.aliyuncs.com/install_AIACC-Training.sh && bash install_AIACC-Training.sh python
```

- 重新安装深度学习框架之后，也需要重新安装AIACC-Training。

2. 手动安装AIACC-Training软件包

您可以使用pip手动安装最新的AIACC-Training软件包：

- 如果您使用的是Pytorch，请使用以下命令安装AIACC-Training：

Bash 复制代码

```
pip install --force-reinstall https://ali-perseus-release.oss-cn-huhehaote.aliyuncs.com/cuda${cuda_version}/perseus_torch-1.5.0%2B${framework_version}-cp${python_version}-cp${python_version}m-linux_x86_64.whl --trusted-host mirrors.aliyun.com -i http://mirrors.aliyun.com/pypi/simple/
```

- 如果您使用TensorFlow或MXNET，请使用以下命令安装AIACC-Training：

Bash 复制代码

```
pip install --force-reinstall https://ali-perseus-release.oss-cn-huhehaote.aliyuncs.com/cuda${cuda_version}/perseus_${framework}-1.5.0%2B${framework_version}-py2.py3-none-manylinux1_x86_64.whl --trusted-host mirrors.aliyun.com -i http://mirrors.aliyun.com/pypi/simple/
```

参数说明：

1. `${cuda_version}` 为 cuda的版本（除去“.”号），比如cuda 11.0对应 `cuda_version=110`
2. `${framework}` 为框架的类型，可选torch/tensorflow/mxnet，比如 `framework=torch`
3. `${framework_version}` 为框架版本号，比如PyTorch 1.7.1对应 `framework_version=1.7.1`
4. `${python_version}` 为python的版本号（除去“.”号），比如python3.6对应 `python_version=36`

以PyTorch1.7.1 CUDA11.0 Pyhon3.6为例，一键安装命令如下：

Bash 复制代码

```
cuda_version=110          #注意：此处不带“.”号
framework=torch
framework_version=1.7.1
python_version=36
pip install --force-reinstall https://ali-perseus-release.oss-cn-huhehaote.aliyuncs.com/cuda${cuda_version}/perseus_${framework}-1.5.0%2B${framework_version}-cp${python_version}-cp${python_version}m-linux_x86_64.whl --trusted-host mirrors.aliyun.com -i http://mirrors.aliyun.com/pypi/simple/
```

以TensorFlow1.15.0 CUDA10.0 Pyhon3.6为例，一键安装命令如下：

Bash 复制代码

```
cuda_version=100          #注意：此处不带“.”号
framework=tensorflow
framework_version=1.15.0
python_version=36
pip install --force-reinstall https://ali-perseus-release.oss-cn-huhehaote.aliyuncs.com/cuda${cuda_version}/perseus_${framework}-1.5.0%2B${framework_version}-py2.py3-none-manylinux1_x86_64.whl --trusted-host mirrors.aliyun.com -i http://mirrors.aliyun.com/pypi/simple/
```

注意：

- 对于Python3.8或以上的whl包，请使用以下的下载地址模版：

```
https://ali-perseus-release.oss-cn-huhehaote.aliyuncs.com/cuda${cuda_version}/perseus_${framework}-1.5.0%2B${framework_version}-cp${python_version}-cp${python_version}-linux_x86_64.whl
```

- 重新安装深度学习框架之后，也需要重新安装AIACC-Training。

方式二：使用装有AIACC-Training的Conda环境

Conda是一款开源跨平台的软件包和环境管理系统。您可以一键创建包含AIACC-Training的Conda环境，该环境已安装CUDA Toolkit、Python3、深度学习框架和最新的AIACC-Training软件等，方便您快速搭建和管理不同的深度学习框架和版本，并通过AIACC-Training显著提升训练性能。
使用之前，除了需要满足前提条件以外，还需要安装最新的Miniconda。并且根据需要的框架版本和环境信息，一键创建相应的Conda环境。

```
conda env create -f https://ali-perseus-release.oss-cn-huhehaote.aliyuncs.com/conda/latest/${framework}_${framework_version}_cu${cuda_version}_py${python_version}.yaml
```

以PyTorch1.7.1 CUDA11.0 Pyhon3.6为例，一键创建Conda环境的命令如下：

```
cuda_version=11.0  #注意：此处需带“.”号，此版本可小于或等于ECS已安装的CUDA版本
framework=torch    #tensorflow/mxnet/torch分别对应TensorFlow/MXNet/PyTorch框架
framework_version=1.7.1
python_version=36
conda env create -f https://ali-perseus-release.oss-cn-huhehaote.aliyuncs.com/conda/latest/${framework}_${framework_version}_cu${cuda_version}_py${python_version}.yaml
```

注意：

- 若提示找不到该Conda URL，则表示暂不支持该框架版本，请参考完整的支持列表。

方式三：装有AIACC-Training的Docker镜像

您可以下载已安装好AIACC-Training的Docker镜像，镜像内已安装CUDA、Python3、深度学习框架和最新的AIACC-Training软件等，帮助您快速部署深度学习环境，管理不同的CUDA环境，并通过AIACC-Training显著提升训练性能。

使用之前除了需要满足前提条件以外，还需要下列的环境依赖：

- 已安装Docker（若您使用的操作系统为Alibaba Cloud Linux 2，请参考部署并使用Docker）
- 已安装NVIDIA-Container

然后根据需要的框架版本和环境信息，下载对应的Docker镜像。

Bash 复制代码

```
docker pull registry.cn-beijing.aliyuncs.com/cto_office/perseus-training:${OS_type}-cu${cuda_version}-${framework}${framework_version}-py${python_version}-latest
```

参数说明：

简称	说明	示例值
os_type	Docker镜像的系统类型。注意：Docker内的系统类型与ECS实例的系统类型无关	centos7, ubuntu1804
cuda_version	CUDA的版本号。注意：此处需带“.”号，此版本可小于或等于ECS已安装的CUDA版本	11.0, 10.2, 10.1, 10.0
framework	深度学习框架类型的缩写，tf/mx/pt分别对应TensorFlow/MXNet/PyTorch框架	tf, mx, pt
framework_ver	深度学习框架对应的版本号，以“xx.xx.xx”形式。请参考支持列表	2.4.0, 1.7.0
python_version	Python版本号。注意：此处不带“.”号，36/37/38对应Python3.6/3.7/3.8	36, 37, 38, 39

以CentOS7，CUDA11.0，TensorFlow 2.4.0的镜像为例，命令如下：

```bash
os_type=centos7           #注意：Docker内的系统类型与ECS实例的系统类型无关
cuda_version=11.0         #注意：此处需带“.”号，此版本可小于或等于ECS已安装的CUDA版本
framework=tf              #tf/mx/pt
framework_version=2.4.0
python_version=36         #注意：此处不带“.”号，如36/37/38
docker pull registry.cn-beijing.aliyuncs.com/cto_office/perseus-training:${os_type}-cu${cuda_version}-${framework}${framework_version}-py${python_version}-latest
```

注意：

- 若提示找不到该Docker镜像，则表示暂不支持该框架版本，请参考完整的支持列表。
- 若使用容器进行分布式训练， `docker run` 启动需要增大shm的分配，参数如下： `--shm-size=1g --ulimit memlock=-1`

2.1 PyTorch

自从PyTorch 1.x 发布迭代后，使用PyTorch 原生自带的DDP 进行分布式训练逐渐行程了主流。关于PyTorch DDP的介绍可以参考PyTorch，本节主要介绍使用PyTorch DDP 兼容的方式运行AIACC 加速的适应方法。（同时在PyTorch 分布式训练上，AIACC同样兼容horovod的使用方式，针对horovod api的使用，详见附录：PyTorch+horovod适配）

2.1.1 代码适配与运行

1. 代码适配：仅需在训练代码main 函数对应的文件上增加一行导入aiacc，但是必须确保导入perseus的位置要在导入torch之前。如下：

```python
import perseus  # aiacc的导入一定要放在torch的导入之前
import torch
import torch.nn as nn
import torch.distributed as dist
......
```

2. 启动脚本：运行ddp的分布式训练使用标准的torch.distributed.launch 方式，如下命令行示例所示

```python
### 以下示例展示使用双机进行pytorch ddp方式的启动脚本，每台机器挂8张卡

### 机器一上的命令
python -m torch.distributed.launch --nproc_per_node=8 --nnodes=2 --node_rank=0 --master_addr=${machine1_ip} --master_port=6007 --use_env ${TRAIN_SCRIPT}

### 机器二上的命令
python -m torch.distributed.launch --nproc_per_node=8 --nnodes=2 --node_rank=1 --master_addr=${machine1_ip} --master_port=6007 --use_env ${TRAIN_SCRIPT}
```

注意：

1. 脚本中的${machine1_ip}为机器一的内网ip 如 192.168.2.211（通过ifconfig查看）
2. 脚本中的${TRAIN_SCRIPT}为实际训练的脚本
3. 对于PyTorch 1.9.0或以上的版本，需要在启动命令上加入 `--standalone` 参数

2.1.2 示范用例

AIACC-Training软件包路径中包含示例代码，具体操作如下：

1. 进入示例代码目录：

```shell
cd `echo $(python -c "import perseus; print(perseus)") | cut -d\' -f 4 | sed "s/\_\_init\_\_\.py//"`examples/
```

3. 启动训练

采用原生ddp的启动方式运行PyTorch_ddp_benchmark.py，以下为单机8卡训练的启动命令：

```shell
NP=8
ADDR=localhost
PORT=6006
python -m torch.distributed.launch --nproc_per_node=$NP --nnodes=1 --node_rank=0 --master_addr=$ADDR --master_port=$PORT \
  pytorch_ddp_benchmark.py
```

2.2 TensorFlow

TensorFlow目前主流的进行数据分布式训练的方式是Horovod， AIACC-Training 1.5 支持Horovod API兼容的方式对TensorFlow 分布式训练进行加速。

2.2.1 代码适配与运行

1. 代码修改：

适配AIACC-Training的方式与Horovod一致，如果代码是使用Horovod进行分布式训练，只需要修改import一行即可。

```python
import perseus.tensorflow.horovod as hvd
```

如果训练代码是单机训练，请继续参考附录：TensorFlow 适配，升级为分布式训练代码。

2. 启动训练：

训练脚本通过perseusrun启动来进行分布式训练。比如，8卡2机的启动方式如下，此处IP为内网IP。

```bash
perseusrun -np 16 -H ${IP1}:8,${IP2}:8 python xxx.py
```

2.2.2 示范用例

AIACC-Training软件包路径中包含示例代码，该目录下有针对sess，Keras，Estimator，tf2 eager等的示例代码。具体操作如下：

1. 进入示例代码目录：

```shell
cd `echo $(python -c "import perseus; print(perseus)") | cut -d\' -f 4 |
sed "s/\_\_init\_\_\.py//"`examples/
```

2. 启动分布式训练。

比如对Keras形式的MNIST训练模型启动单机2卡的启动命令如下：

```shell
perseusrun -np 2 -H localhost:2 python tensorflow_keras_mnist.py
```

备注：关于启动方式的更多介绍，请参考启动方式说明

2.3 MXNet

MXNet支持不同的分布式训练方式，原始版本是基于KV Store存储的PS方式，AIACC-Training1.5能够支持创建KV Store的方式对MXNet分布式训练进行加速，此外MXNet也支持horovod的分布式训练方式，AIACC-

Training1.5也能够无缝兼容horovod的API版本，本节主要讲解基于horovod的API版本适配，基于kvstore版本的详见附录：基于kvstore的代码适配。

2.3.1 代码适配与运行

适配AIACC-Training的方式与Horovod一致，如果代码是使用Horovod进行分布式训练，只需要修改import一行即可。

```python
import perseus.mxnet as hvd
```

注：其他完整的适配过程见附录：基于horovod的代码适配

2.3.2 示范用例

AIACC-Training软件包路径中包含示例代码，具体操作如下：

1. 进入示例代码目录：

```shell
cd `echo $(python -c "import perseus; print(perseus)") | cut -d\' -f 4 | sed "s/\_\_init\_\_\.py//"`examples/
```

2. 启动分布式训练。

比如启动单机8卡的MNiST训练命令如下：

```shell
perseusrun -np 8 -H localhost:8 python $examples_path/mxnet_mnist.py
```

2.4 Caffe

Caffe的支持由于官方不再更新，所以AIACC没有提供新的feature，已有的支持方式详见附录

3. 典型分布式训练的优化案例

客户	模型	框架	规模GPUs	训练速度提升
某智能芯片厂商	图像分类	MXNet	256	100%
某智能芯片厂商	人脸识别	MXNet	256	200%
某汽车厂商	FaceNet	PyTorch	32	100%
某手机厂商	BERT	TensorFlow	32	30%
某手机厂商	GPT2	PyTorch	32	30%
某人工智能厂商	Faster-RCNN	MXNet/hvd/byteps	128	30%
某人工智能厂商	InsightFace	MXNet/hvd/byteps	128	200%
某在线教育	ESPnet	PyTorch-DP	16	30%
某在线教育	ESPnet2	PyTorch-DDP	16	30%+
某在线教育	CTR	PyTorch	32	80%
某在线教育	OCR	PyTorch	32	30%
某手机厂商	图像分类	PyTorch	128	25%
某手机厂商	MAE	PyTorch	32	30%
某研究院	GPT2	PyTorch+Megtragon	32	30%
某社交媒体	MMDetection2	PyTorch	32	30%
某金融智能	InsightFace	PyTorch	32	50%
某手机厂商	Detection2	PyTorch	64	25%
某视觉团队	insightface	MXNet	64	50%
某游戏厂商	ResNet	PyTorch	32	30%
某城市大脑	InsightFace	MXNet	16	42%
某制药科技	Autoencoder	PyTorch	32	30%
某自动驾驶	swin-transformer	PyTorch	32	70%

典型客户案例介绍

阿里云助力图森未来提升性能、加速模型迭代

阿里云神龙AI加速引擎帮助vivo将训练性能提升30%-70%

4. 附录

4.1 环境变量

默认情况下，不需要设定任何环境变量，以下环境变量可以改变AIACC-Training的默认行为，可以根据需要设定：

Perseus 配置开关	功能描述	建议
PERSEUS_ALLREDUCE_NANCHECK	设置是否检查梯度值为NaN： 0： 关闭此功能、 1： 开启运行时NaN检查 默认值为0，表示不启用检查。	
PERSEUS_ALLREDUCE_DTYPE	选择GPU之前通信的梯度压缩模式： 0：开启Float16的梯度压缩 1： 关闭梯度压缩 2： 混合精度，节点内不压缩梯度， 节点间采用Float16 默认值为0，表示开启float16压缩。	相对于纯float32训练，若发现精度降低，建议选择2-混合精度。 一般状况，建议不用设置此值，选择默认fp16精度压缩。 对于启用了AMP的场景，建议选择1-关闭梯度压缩。
PERSEUS_ALLREDUCE_MODE	选择节点间的AllReduce通信模式，分为2种： 0： 所有节点采用一层allreduce进行通信 1： 若存在多节点，每节点多卡，会做两级通信，节点内部先reduce到一张卡上，然后每个节点一张卡参与第二级通信 默认为Perseus自动选择。若不设置，Perseus自动选择AllReduce通信模式	建议不要设置此值，有Perseus自动选择最优值
PERSEUS_ALLREDUCE_STREAMS	设定多流通信的上限流数。默认值为4，取值范围：[1, 12]	大多数情况建议不要设置此值。当同时满足以下两种条件，可以考虑设置更高的流数： 1. 带宽较高，例如GN6V大实例，带宽32Gbps，或者SCC网络下采用tcp通信 2. 每张卡有超过200MB以上空闲显存 若显存富裕，扩展性不及预期，可适当增大流数至8.
PERSEUS_ALLREDUCE_FUSION	设定梯度融合粒度，以设定值16为例，梯度融合粒度为所有梯度总和的1/16，参数范围：[0, 128]，无默认值，此时AIACC会自动设置最优值	建议不要设置此值

	动设置最优值	
PERSEUS_ACCUMULATE_N_STEPS （1.3.0+）	设置本地梯度累积的step数，即multistep方法实现local gradient accumulate。默认值为1， 可设置为2，4，8等值	需要大batch size 而显存不足的情况可以通过本地累积梯度实现大batch size；或希望降低通信量时,可利用该功能模拟扩大n倍batch size并保持同等的epoch数,通信量将降至1/n。 注：local gradient accumulate相当于增大训练的batch size，超参如学习率等需要适配该大batchsize
PERSEUS_DOWNSAMPLE_N_ELEMENTS （1.3.0+）	设置梯度采用gossip 压缩方式的粒度，可设置为2、4、8 等值，默认值为1	gossip梯度压缩可使用于step size较大的情况，可以减少梯度的通信规模。在imagenet/resnet50 8*64 batch size下设置为2/4/8 均能保证精度，同时通信量降低50%/75%/87.5%
PERSEUS_GRADIENT_MOMENTUM （1.3.0+）	设置梯度的momentum值，与PERSEUS_DOWNSAMPLE_N_ELEMENTS一起使用，默认值为1	在MomentumSGD 的imagenet训练中可设置为0.9
PERSEUS_NCCL_ENABLE （需要特殊版本，请提交工单获得支持）	0. 关闭NCCL混合链路支持 – 默认值 1. 启用NCCL混合链路支持	当使用SCC机型时，可以启用此feature，在训练是同时使用RDMA链路与VPC链路，达到带宽叠加的效果，详细使用方式，请提交工单获得支持
PERSEUS_ALLREDUCE_GRADIENT_SCALE （1.3.0+）	设置梯度scale系数，默认值为10 仅当PERSEUS_ALLREDUCE_DTYPE 为0或者2时生效。	当通信环节存在FP32到FP16精度压缩时（即PERSEUS_ALLREDUCE_DTYPE 为0或者2时），此参数生效。从FP32到FP16的转换时会乘以此系数，反向则除以此系数。若初始loss值非常大导致出现NaN需要降低此值。
PERSEUS_OFFLINE_NEG （1.3.2 +）	设置离线梯度协商模式，默认为0，设置为1时开启离线梯度协商	对于TensorFlow的部分模型，若模型参数个数较多，扩展性不佳，建议开启离线协商；对于其他框架，仅当大量采用syncbn layer时需要开启此选项；所有其它的case，可保持关闭
PERSEUS_PERF_CHECK_N_STEPS (1.3.2+)	设置异常性能检查频次，默认值为0，表示不开启	PERSEUS_PERF_CHECK_N_STEPS=100表示每隔100个step启动一次异常检测，若出现GPU卡异常，会在终端打

		印出性能异常的卡的信息。 注：该功能与TensorFlow xla不兼容，请勿在xla环境下开启
PERSEUS_MASTER_PORT (1.5.0+)	设置启动master的端口号，默认为6666 仅当PyTorch launcher启动DDP训练时生效	默认PyTorch训练时会启动一个rendezvous服务，AIACC会启动类似的一个rendezvous服务，只需保证两者端口不同即可，因为两者是共用同一台master_addr地址。
PERSEUS_NCCL_NETWORK_INTERFACE (1.5.0+)	设置nccl通信使用的网卡信息，默认为eth0	如有特殊网卡配置，可以根据实际情况进行调整
PERSEUS_GLOO_NETWORK_INTERFACE (1.5.0+)	设置gloo通信使用的网卡信息，默认为eth0	如有特殊网卡配置，可以根据实际情况进行调整
GLOO_TIMEOUT_SECONDS (1.4.0+)	设置gloo通信的超时时间，以秒为单位，默认为60	如果有复杂逻辑或者网络环境问题导致hang的发生，可以增大该超时参数
PERSEUS_CHANGE_HVD_ALLGATHER (1.5.0+)	设置Allgather的计算模式，默认为0，兼容DDP/mpi4py的方式。 若设置为1，则为horovod的实现方式。 示例： 假如输入为2个tensor: tensor1=[0,0]，tensor2=[1,1] 使用参数0则输出为tensor([[0,0], [1,1]]) 使用参数1则输出为tensor([0,0,1,1])	在使用PyTorch原始DDP训练时使用默认即可，在horovod实现的时候，可以考虑设置为1进行切换；对应在PyTorch上的syncbn的实现也会不同，性能会有少许差异
PERSEUS_USE_DDP_LAUNCHER (1.5.0+)	设置PyTorch DDP训练的模式切换，默认为1，兼容DDP原始的launcher启动方式； 若设置为0，则修改为horovod的mpirun启动方式	在使用PyTorch原始DDP训练时使用默认即可，在horovod实现的时候，需要设置为0进行切换；

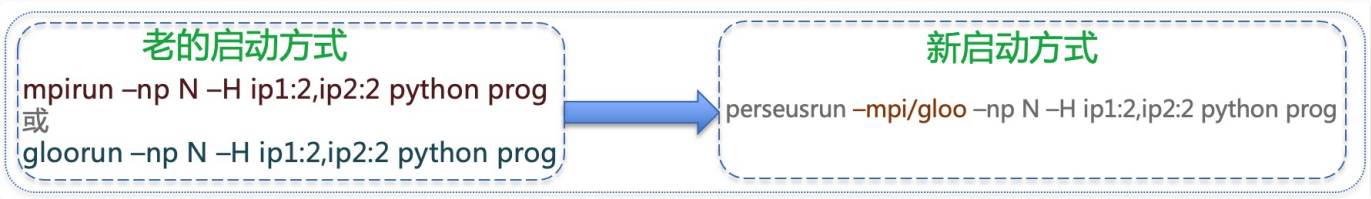
环境变量可加在启动命令 `perseusrun` 之前，比如开启混合精度并设置梯度scale系数为5的启动命令如下：

Plain Text | 复制代码

```
PERSEUS_ALLREDUCE_DTYPE=2 PERSEUS_ALLREDUCE_GRADIENT_SCALE=5 perseusrun xxx
```

4.2 启动方式说明

为了更好地使用AIACC-Training加速分布式训练， `perseusrun` 提供了统一的分布式启动入口。整合了不同的底层通讯设施和多种训练模式，同时兼容了分布式训练与弹性训练，新旧启动方式升级如下。



使用方式如下：

- 单机启动命令如下，默认使用gloo的backend：

```shell
perseusrun -np NP [-H localhost:N] -- COMMAND [ARG [ARG...]]
```

- 多机：

比如，使用MPI为backend的双机启动命令如下：

```shell
perseusrun --mpi -np NP -H host1:N,host2:N -- COMMAND [ARG [ARG...]]
```

比如，使用gloo为backend的双机启动命令如下：

```shell
perseusrun --gloo -np NP -H host1:N,host2:N -- COMMAND [ARG [ARG...]]
```

其中：

- N为每个实例启动的进程数，一般为该ECS实例的GPU个数
- NP为总的启动进程数，NP = N * {ECS实例总数}
- host1, host2...各个ECS实例的内网IP
- program.py为您的训练的Python程序，请对应修改名字

若需要更多的功能，请输入 `perseusrun -h` 获得详细介绍。

示例：

```bash
# 启动单节点-8卡训练,使用默认的gloo backend
perseusrun -np 8 -H localhost:8 -- python train.py --model resnet50
perseusrun -np 8 -- python train.py --model resnet50

# 启动双节点-每节点8进程的训练,并使用mpi作为通信backend
perseusrun --mpi -np 16 -H host1:8,host2:8 -- python train.py --model resnet50

# 启动四节点-每节点8进程的训练,并使用gloo作为通信backend
perseusrun --gloo -np 32 -H host1:8,host2:8,host3:8,host4:8 -- python train.py --model resnet50
```

4.3 常见问题

4.3.1 框架无关

- **容器内运行多卡分布式训练，触发 `NCCL unhandled error`**

  设置 `NCCL_DEBUG=INFO`，若发现如下的log信息，则解决方法为再启动容器时加入参数 *`--shm-size=1g --ulimit memlock=-1`* 加到nvidia-docker run的命令行中去

```bash

hzh-perseus-5868d9dfdb-q664k:34486:37433 [3] include/shm.h:28 NCCL WARN Call to posix_fallocate failed : No space left on device
hzh-perseus-5868d9dfdb-q664k:34486:37433 [3] NCCL INFO include/shm.h:41 -> 2

```

- **Perseus无法正常启动，遇到框架相关的 `Undefined symbols`**

  排查perseus的版本和框架版本是否兼容。Perseus的版本号第一部分是perseus自身的版本，第二部分是对应的框架版本，对应的框架版本必须与当前所使用的框架版本一致。

- **Perseus启动遇到链接错误：`libcuda.so.1: cannot open shared object file: No such file or directory`**

排查是否安装了CUDA驱动以及CUDA SDK

- **Perseus启动遇到 `libcudart.so.X.Y: cannot open shared object file: No such file or...` 类似的错误**

  排查cuda版本是否与当前的perseus版本一致

- **容器环境下，出现类似unhandled system error，错误位置在 `nccl_comm.cpp` 内**

  可能与shm设置相关，需要修改容器启动参数增大shm的分配，参数如下：--shm-size=1g --ulimit memlock=-1

- **Perseus报错 `python: double free or corruption`**

  1. 可能原因I：是不同rank中传入的相同tensor_name的shape或size不一致，本质上用法也是不正确的，不同shape无法进行allreduce操作

  2. 可能原因II：不同rank执行的时序有较大差异，可以在epoch的结束后进行一次同步，比如MXNet下面可以使用： `kv._barrier(); mx.nd.waitall()`

- **训练结束后hang住，没有正常退出**

  python层面结束后会触发exit信号给perseus后端，但是python层面并没有真正结束，有可能是模型代码中存在额外的数据读取进程保持着无法正常结束主进程，一种fix方法是在所有运行结束后执行主进程显示退出: `import sys; sys.exit(0)`

- **启动环节异常的慢，查看系统的线程数非常多，有大量omp相关的线程**

  一般单GPU分配4个以内的omp线程是合适的， `export OMP_NUM_THREADS=4` 或者更低可以解决此类问题。

4.3.2 TensorFlow

- **显存OOM**
  a. nvidia-smi 检测启动过程中，各显卡占用显存是否均衡增加，单process对应单GPU，每个process所使用显存应该接近。
  b. 若存在多个process 绑定同一张卡，需要检查config.gpu_options.visible_device_list是否正确设置。
  c. 若存在显存直接被用尽，尝试增加 config.allow_growth = True
  d. 若启用了xla：
    i. 可以考虑增加参数config.gpu_options.per_process_gpu_memory_fraction = 0.9 或者0.8。

- **快速判断是否是梯度通信带来的性能瓶颈**

  可以将适配的代码中 `DistributedOptimizer(opt)` 注释掉，注释掉后将不会产生梯度通信，可以进一步排查数据IO、CPU预处理等性能瓶颈

- **对数据集做shard的注意事项**

  由于AIACC-Training是多个进程启动同一份训练代码，所以需要做对数据集做数据划分，让每个进程处理和训练不同的子数据集。TensorFlow为tf.data.Dataset类提供了自动切分数据的shard()接口，可结合size()，local_rank()或rank()函数进行自定义的数据切分，比如

```python
dataset = tf.dataset.shard(hvd.size(), hvd.rank())
```

注意，为保证算法的准确性shard使用要放在数据集shuffle操作之前；为了性能的考虑，shard操作不要放在repeat操作之后，否则会带来额外的预处理负担，严重影响性能

4.3.3 PyTorch

- **模型数据存放位置冲突，导致错误 `Input type (CUDAFloatTensor) and weight type (CPUFloatTensor) should be the same`**

一般是因为存在模型的参数不在GPU中，而输入数据在GPU中，一般是遗漏了model的cuda操作，可以通过增加model.cuda()将模型转移到GPU上来解决该问题。

- **出现错误 `RuntimeError: all tensors must be on devices[0]`**

1. 需要判断是否在程序初始化时候指定了gpu_ids，如 `torch.cuda.set_device(hvd.local_rank())`

2. 还有可能就是采用DataParallel的时候没有统一设置，需要设置一致的 `device_ids` ，如 `nn.DataParallel(model.cuda(), device_ids=str(hvd.local_rank()))`

- **加载模型显存OOM**

一般是模型数据太大导致，可以通过将模型导入为CPU内存来解决，如 `torch.load(pretrain, map_location='cpu')`

- **异常退出**

1. 首先需要确保修改步骤已经按照上述的修改说明。

2. 可能的原因就是多个模型的组合问题，多个模型的named_parameters需要进行合并，然后再传递给perseus进行optimizer的包装。

3. 另外多个模型组合的时候广播参数是需要广播所有参数的，因此也需要进行state_dict的参数合并，具体代码实例可以参照2.2.3 PyTorch支持部分。

- **端口占用**

1. 首先需要保证已有的端口是否被其他进程占用，如果有可以使用 pkill python 进行进程杀除
2. 在torch1.9+的版本，默认走torch.distributed.run的启动方式，会使用默认的 rdzv_backend作为训练的launcher，但是在启动训练之前就已经创建了服务，而代码里如果有dist.init_process_group()则会继续创建该服务，导致冲突，一种解决方法就是增加配置选项：--standalone

4.3.4 MXNet

- **额外显存占用问题。单机8卡为例，0号卡，被其余7个process均占用了200MB到500MB的显存，从而导致0号卡显存被耗尽**

此问题的root cause在于MXNet内部的cpu_pinned memory分配机制默认会使用0号卡。解决方法是参照MXNet章节，重新绑定GPU卡。

- **运行时显示NDArray相关的symbol没有定义 `Undefined symbols`**

此问题发生在pip安装的1.4之前的MXNet版本，由于1.4版本之前的pip安装，没有export libMXNet.so中对于perseus必要的symbol，因此导致此问题。解决方法有两个，第一个是升级到MXNet1.4或者更新的版本，第

二个是重新编译安装MXNet。

- **启动速度较慢**

1. 检查cpu的负载，若占比很高可尝试通过设置

```bash
export MXNET_USE_OPERATOR_TUNING=0
export MXNET_USE_NUM_CORES_OPERATOR_TUNING=1
export OMP_NUM_THREADS=1
```

2. 可减小preprocess的线程数，因为Perseus下训练模式为单process，单GPU，之前的默认线程数设置会太大，可以根据本节点的GPU数目，等比例缩小preprocess的线程数。比如preprocess的线程数默认值为24，而本节点GPU数目为8，那么将preprocess的线程数降低为3或者4会比较合适。

- **单机正常，多机异常退出**

可能是乌龙错误，人为导致多机中存在正在运行训练的机器，则运行多机会发生 `create cusolver handle failed` 的错误，可以使用 `mpirun` 去执行 `nvidia-smi` 检查多机中是否存在运行的机器

4.4 PyTorch + Horovod API适配

（请更新到Perseus1.3.2 或者更新版本使用PyTorch的相关加速功能）

1. 需要 import perseus.torch.horovod as hvd
2. 执行hvd.init
3. 将当前process绑定对应的GPU卡：

```python
torch.cuda.set_device(hvd.local_rank())
```

4. 通常情况下，学习率需要增大hvd.size()倍，上面有代码示例。部分模型不需要增大，例如bert模型。
5. 重载optimizer：

```python
optimizer = hvd.DistributedOptimizer(
optimizer,named_parameters=model.named_parameters())
```

注意：如果是多个模型，有多个named_parameters，则需要进行合并处理，例如

```python
all_named_parameters = []
for name, value in model1.named_parameters():
    all_named_parameters.append((name, value))
for name, value in model2.named_parameters():
    all_named_parameters.append((name, value))
optimizer = hvd.DistributedOptimizer(
                            optimizer,
named_parameters=all_named_parameters)
```

6. 广播全局变量参数到所有节点：

```python
hvd.broadcast_parameters(model.state_dict(), root_rank=0)
hvd.broadcast_optimizer_state(optimizer, root_rank=0)
```

注意：如果是多个模型，有多个state_dict，则需要进行合并处理，例如

```python
all_state_dict={}
all_state_dict.update(model1.state_dict())
all_state_dict.update(model2.state_dict())
hvd.broadcast_parameters(all_state_dict, root_rank=0)
```

7. 数据集划分：

```python
train_sampler = torch.utils.data.distributed.DistributedSampler(
    train_dataset, num_replicas=hvd.size(), rank=hvd.rank())
loader = torch.utils.DataLoader(
    train_dataset, batch_size=batch_size, sampler=train_sampler,
**kwargs)
```

8. 模型单机单卡化：

perseus运行的是单机单卡程序，然后下发给各个GPU，进行单机多卡或分布式训练，所以程序需要为单机单卡的运行方式。

- 原来为

```python
model = nn.DataParallel(model.cuda())
```

- 修改为：

```python
# 方式1
model = nn.DataParallel(model.cuda(), device_ids=[hvd.local_rank()])
# 方式2，cuda()默认会去找前面第三步骤设置的当前process绑定的GPU卡
model = model.cuda()
```

只需要在 `process 0` 上保存checkpoint，防止其他process冲突，类似的还有verbose、tensorboardX。

9. 保存checkpoint

```python
save_checkpoint = True if hvd.rank() == 0 else False
verbose = 1 if hvd.rank() == 0 else 0
log_writer = tensorboardX.SummaryWritter(log_dir) if hvd.rank() == 0 else None
```

10. 加载checkpoint

```python
if hvd.rank() == 0:
    checkpoint = torch.load(filepath)
    model.load_state_dict(checkpoint['model'])
    optimizer.load_state_dict(checkpoint['optimizer'])
```

11. 单机8卡运行

```shell
mpirun -allow-run-as-root -np 8 -npernode 8 -x NCCL_DEBUG=INFO ./train.sh
```

12. 4机8卡运行

```shell
mpirun -allow-run-as-root -bind-to none -np 32 -npernode 8 \
                    -x NCCL_DEBUG=INFO -x PATH -x LD_LIBRARY \
                    -x PERSEUS_ALLREDUCE_STREAMS=8
                    -hostfile mpi_host.txt ./train.sh
```

4.5 TensorFlow + Horovod API适配

本小节介绍如何使用Horovod兼容API进行TensorFlow 分布式训练的基本步骤，下面是将原始训练代码适配到AIACC-Traninig的一般过程：

1. AIACC-Training for TensorFlow支持Horovod的API。适配AIACC-Training的方式与Horovod一致，**如果之前是使用Horovod进行分布式训练，只需替换第一步import模块即可。**

```python
import perseus.tensorflow.horovod as hvd
```

如果训练代码是非分布式代码，请继续参考以下步骤升级为Horovod接口的分布式训练代码。

2. 在main函数的开头部分，执行 `hvd.init()`
   - *init()* —— 初始化Perseus Horovod模块，必须在使用其他任何perseus API之前调用

```python
hvd.init()
```

3. 训练数据切分：需要每个进程使用不同的数据，来达到数据并行训练的目的。可以手动或自动进行数据切分。TensorFlow为tf.data.Dataset类提供了自动切分数据的shard()接口，可结合size()，local_rank()或rank()函数进行自定义的数据切分，比如

```python
# 每个节点使用同一份数据
dataset = tf.dataset.shard(hvd.size(), hvd.rank())

# 已将数据切分到每个节点上，每个节点使用数据的不同子集
dataset = tf.dataset.shard(hvd.size(), hvd.local_rank())
```

其中：

   - *size()* —— 返回所有参与训练的GPU卡数，例如双节点，每节点8张卡，会返回16。也可用来调整学

习率，以及checkpoint保存batch number等等，在上述示例中也有用到此API。

- *rank()* —— 返回当前GPU在所有节点所有GPU中的编号，例如四台8卡机器，rank()是0-31的范围。
- *local_rank()* —— 返回当前GPU在本节点上的编号，例如两台8卡机器，每台机器上的卡的local_rank()是0-7的范围，也可用于绑定进程到相应的GPU卡。

注意：

- 为保证算法的准确性和保证性能，shard()要放在数据集的shuffle()和repeat()操作之前。
- 请勿对evaluation或test数据集进行切分，否则导致各进程的评估结果不一致。

4. 通常情况下，对于训练的step数和warmup step参数，需要除以总的进程数 `hvd.size()` ，学习率需要对应增大 `hvd.size()` 倍（部分模型不需要增大，如BERT模型，具体根据训练收敛情况作判断）。

Plain Text 复制代码

```
step = step // hvd.size()
learning_rate = learning_rate * hvd.size()
```

5. 重载Optimizer：需要将原先单机的optimizer使用 `hvd.DistributedOptimizer()` 函数重载。使用范例如下：

Python 复制代码

```
# original optimizer
tf_optimizer = tf.train.AdamOptimizer(learning_rate)

# AIACC-Training: wrap the Distributed Optimizer.
optimizer = hvd.DistributedOptimizer(tf_optimizer)
```

其中：

- *DistributedOptimizer(TensorFlow_Optimizer)* —— 重载optimizer，使其可以进行AIACC分布式训练。输入参数为标准的TensorFlow下的Optimizer，输出为被AIACC重载过的Optimizer。

6. 将当前进程绑定对应的GPU卡：

- **对于TensorFlow 1.x版本：**

Python 复制代码

```
config = tf.ConfigProto()
config.gpu_options.allow_growth= True
config.gpu_options.visible_device_list =str(hvd.local_rank())
```

然后根据不同情况，将config传入相应的函数中，

- 若使用tf.Session.run()，需要在初始化tf.Session时传入config：

```bash
with tf.Session(config=config) as sess:
  sess.run(...)
```

- 若使用Estimator，需要在初始化Estimator时，以 `tf.estimator.RunConfig(session_config=config)` 封装传入：

```bash
mnist_classifier = tf.estimator.Estimator(
    model_fn=cnn_model_fn, model_dir=model_dir,
    config=tf.estimator.RunConfig(session_config=config))
```

- 若使用Keras，需要封装后传入：

```python
from tensorflow.keras import backend as K
K.set_session(tf.Session(config=config))
```

- **对于TensorFlow 2.x版本：**

```python
gpus = tf.config.experimental.list_physical_devices('GPU')
for gpu in gpus:
    tf.config.experimental.set_memory_growth(gpu, True)
if gpus:
    tf.config.experimental.set_visible_devices(gpus[hvd.local_rank()], 'GPU')
```

7. 在开始训练之前，需要同步所有训练进程的初始状态，根据不同的使用形式进行适配：
   a. 对于传统的tf.Session.run()形式，需在训练前先运行 `tf.Session.run(hvd.broadcast_global_variables(0))`

b. 对于Keras，需在训练的callbacks中增加 `hvd.keras.callbacks.BroadcastGlobalVariablesCallback(0)`
c. 对于Estimator，在session hook中加入 `hvd.BroadcastGlobalVariablesHook(0)`

8. checkpoint或模型只需要在root rank上保存，其它rank设置为none。此步骤是必须的，可以防止各个进程都保存模型而发生覆盖冲突。

```python
checkpoint_dir = './checkpoints' if hvd.rank() == 0 else None
```

9. 最后，训练脚本通过perseusrun启动来进行分布式训练。比如，8卡2机的启动方式如下，此处IP为内网IP。

```
perseusrun -np 16 -H {IP1:8},{IP2:8} python xxx.py
```

备注：关于启动方式的更多介绍，请参考启动方式说明
您也可以通过以下命令进入示例代码目录，找到sess，Keras，Estimator，tf2 eager等对应的适配例子作参考。

```shell
cd `echo $(python -c "import perseus; print(perseus)") | cut -d\' -f 4 | sed "s/\_\_init\_\_\.py//"`examples/
```

4.6 MXNet 适配

4.6.1 基于KVStore方式的代码适配

Perseus KVStore 新API

为了支持insightface中的特殊的数据并行+模型并行，Perseus KVStore增加了下列新的API

- **local_rank:** 返回当前GPU worker在本节点内部的序号，以此序号来建立对应的gpu context即可；在上面的例子中，是在启动的shell脚本中获得当前的GPU序号作为参数传入python，更方便的方式是在python内部，直接使用local_rank来作为当前的GPU id来创建ctx，更为方便。
- **init(key_name, ndarray, param_only = false):** init 增加了一个参数param_only 用来同步非普通梯度。

例如feature map数据，allreduce精度数据等，其他任何需要一次性同步的参数。另一个判定准侧是，只要update操作不是由kvstore来完成的ndarray，都需要加param_only参数。

- **push(key_name, ndarray, op = PerseusOp.Sum)**: push参数增加了一个op参数，默认是sum，支持Sum，Max，Min这是用来同步softmax layer的输出。

以下为基于kvstore方式的代码适配方式：

1. 使用Perseus kvstore

通过git diff命令对比修改前后的代码，可以看到只需import perseus MXNet module，并替换kvstore的生成就可以了。

Bash 复制代码

```
diff --git a/example/image-classification/common/fit.py b/example/image-classification/common/fit.py
index 9412b6f..3a6e9a0 100755
--- a/example/image-classification/common/fit.py
+++ b/example/image-classification/common/fit.py
@@ -22,6 +22,7 @@ import time
 import re
 import math
 import mxnet as mx
+import perseus.mxnet as perseus_kv


 def _get_lr_scheduler(args, kv):
@@ -146,7 +147,8 @@ def fit(args, network, data_loader, **kwargs):
     # kvstore
-    kv = mx.kvstore.create(args.kv_store)
+    kv = perseus_kv.create(args.kv_store) if args.kv_store == 'dist_sync_perseus' else mx.kvstore.create(args.kv_store)
     if args.gc_type != 'none':
         kv.set_gradient_compression({'type': args.gc_type,
                                      'threshold': args.gc_threshold})
```

2. 选取对应的GPU device id

AIACC-Training 通过重载KVStore实现了对MXNet分布式训练的支持，在API上与原生KVStore基本兼容，因此模型代码本身不需要太多的修改，最大的变化是使用AIACC-Training后，单进程将绑定在单张GPU卡上，模型代码中的ctx的设定需要稍作修改，如下代码片段所示，将当前process绑定到kv.local_rank所对应的GPU卡上。Perseus kvstore增加了一个local_rank的api，方便进行此操作。

```python
ctx = []
cvd = os.environ['DEVICES'].strip()
if 'perseus' in args.kv_store:
    import perseus.mxnet as perseus
    ctx.append(mx.gpu(kv.local_rank))

```

3. 分布式training的launch：

MXNet的launch方式是基于PS server的，我们没有PS server存在，所以launch方法不同。Perseus采用的是MPI的launch方式。和其它框架类似，Perseus的单机多卡与多机多卡的launch方式基本一致，不再支持原生MXNet下的单机多卡在单一Process中的模式。

以下为4机8卡训练的launch示例：

准备如下的脚本，命名为config.sh，重点是其中的MXNet_VISIBLE_DEVICE的设定，因为我们会采用mpirun来运行，所以会使用MPI的环境变量来推导此进程所对应的GPU 设备id，然后设定此环境变量，同时要将此id作为参数传递到module代码中去创建对应的ctx。

```bash
#!/bin/sh
let GPU=OMPI_COMM_WORLD_RANK % OMPI_COMM_WORLD_LOCAL_SIZE

export OMP_NUM_THREADS=4

MXNET_VISIBLE_DEVICE=$GPU python train_imagenet.py \
                                --network resnet \
                                --num-layers 50 \
                                --kv-store dist_sync_perseus \
                                --gpus $GPU …
```

准备好上面的config.sh后，使用下面的mpirun 命令就可以launch一个4机8卡的训练（总共32卡）

```bash
mpirun –np 32 –npernode 8 –hostfile mpi_host.txt  ./config.sh
```

其中的mpi_host.txt就是一个普通的mpi machine file，和MXNet的ssh launcher的host file也类似，下面是一个简单的示例：

```
192.168.0.1
192.168.0.2
192.168.0.3
192.168.0.4
```

开始训练之后，每个GPU都是一个单独的进程，都有各自的输出，可以类比之前的多机下，每个机器的输出，总性能是所有单process性能的总和。

注意开源版本MXNet会默认占据系统所有的CPU资源，因此在最初的启动阶段，会占用较多的CPU时间，启动速度会很慢。

这个问题可以用下面的几个环境变量解决。
export MXNet_USE_OPERATOR_TUNING=0
export MXNet_USE_NUM_CORES_OPERATOR_TUNING=1
export OMP_NUM_THREADS=1

4.6.2 基于horovod方式的代码适配

1. 导入python代码包

```python
import perseus.mxnet as hvd
```

2. 执行初始化

```python
hvd.init()
```

3. 绑定GPU卡

```bash
# rank and size
rank = hvd.rank()
num_workers = hvd.size()
local_rank = hvd.local_rank()

# Horovod: pin GPU to local rank
context = mx.gpu(local_rank)
```

4. 创建优化器（通常情况下，学习率需要增大hvd.size()倍；部分模型不需要增大，例如bert模型）

```bash
learning_rate = ...
optimizer_params = {'learning_rate': learning_rate * hvd.size()}
opt = mx.optimizer.create(optimizer, **optimizer_params)
```

5. 优化器重载

```bash
# Horovod: create DistributedTrainer, a subclass of gluon.Trainer
trainer = hvd.DistributedTrainer(params, opt)
```

6. 广播参数

```bash
# Horovod: fetch and broadcast parameters
params = net.collect_params()
if params is not None:
    hvd.broadcast_parameters(params)
```

7. 启动训练

使用下面的mpirun 命令就可以launch一个4机8卡的训练（总共32卡）

```bash
mpirun -np 32 -npernode 8 -hostfile mpi_host.txt  ./train.sh
```

其中的mpi_host.txt就是一个普通的mpi machine file，和MXNet的ssh launcher的host file也类似，下面是一个简单的示例：

```
192.168.0.1
192.168.0.2
192.168.0.3
192.168.0.4
```

开始训练之后，每个GPU都是一个单独的进程，都有各自的输出，可以类比之前的多机下，每个机器的输出，总性能是所有单process性能的总和。

4.7 MXNet SyncBatchNorm使用及性能测试

perseus的sync-bn实现基于MXNet官方代码src/operator/contrib/sync_batch_norm-inl.h的计算逻辑，并通过加载libperseus_MXNet.so调用Perseus通信的API，在operator内部实现sync-bn，且支持单机local模式以及全局global模式。

- 功能背景：针对object-detection等小batchsize场景，继续使用每个GPU单独的BatchNorm计算的mean和var信息有较大的偏差，会带来一定的精度损失，通过使用SyncBatchNorm可以弥补对统计信息的内部偏移，真正发挥理论上BN层的作用，即便在大规模分布式的情况下也能达到更高的期望精度，总之，SyncBatchNorm相比原始BatchNorm能够在牺牲一定训练性能的情况下，提高收敛精度的上限。
- 使用步骤：

1. 使用perseus-MXNet-sync-bn.patch补丁：

```bash
patch -p1 < perseus-mxnet-sync-bn.patch
```

2. 编译MXNet源码：

```bash
make USE_OPENCV=1 USE_BLAS=openblas USE_CUDA=1
USE_CUDA_PATH=/usr/local/cuda USE_CUDNN=1 USE_DIST_KVSTORE=1 USE_NCCL=1
USE_LIBJPEG_TURBO=1 MPI_ROOT=/usr/local -j24
```

3. 调用sync-bn模型：
    - 实现基于原始MXNet官方代码，因此兼容sync-bn的原始使用方法，只是名称变为

PerseusSyncBatchNorm如 mx.gluon.contrib.nn.PerseusSyncBatchNorm(comm_scope=0), mx.sym.contrib.PerseusSyncBatchNorm(comm_scope=0)，只是额外增加了一个修改模式的可选参数。

4. 修改模式：
    - local：局部平均，每次forward、backward计算均值和方差后同步只在单机内部分别进行；默认为此模式comm_scope=0；
    - global：全局平均，每次forward、backward时计算均值和方差在全局进行同步，需要修改BN定义的参数，comm_scope=1；

- 精度测试：
    - 基于gluoncv实现的faster-rcnn模型，适配perseus，并进行原始BatchNorm以及PerseusSyncBatchNorm的对比
    - 单卡batchsize 为2，单机8卡的精度数据如下图所示，可以看到从第一个epoch开始，到最终20个epoch结束，SyncBatchNorm达到的精度均高于BatchNorm，最高的mAP从31.3提升到34.6。

4.8 Caffe 适配

通过将Backward计算和Update计算之间增加集合通信能力实现Caffe的Ali-Perseus-Training的支持。其中Ali-Perseus-Training的功能实现在libperseus-caffe.so的库文件内。
将upstream的BVLC Caffe编译成Ali-Perseus-Training分布式框架需要如下步骤
第一步，Patches Upstream BVLC Caffe

- 首先确认BVLC caffe的版本为

```
99bd99795dcdf0b1d3086a8d67ab1782a8a08383
```

- 其次patch 如下修改

```
git apply perseus-dist-1.2.0/patches/bvlc_caffe/1_perseus_distribute_train.patch
git apply perseus-dist-1.2.0/patches/bvlc_caffe/2_cudnn_batchnorm.patch
git apply perseus-dist-1.2.0/patches/bvlc_caffe/3_data_augmentation.patch
```

其中第一个patch给caffe增加Ali-Perseus-Training功能，第二个patch是增加cudnn batch norm的支持，第三个batch增加了image data layer的数据增强功能。若仅需要分布式训练功能只需第一个patch即可
第二步，重新编译Caffe
运行如下命令重新编译caffe生成Ali-Perseus-Training支持，其中输出binary在$HOME/perseus-caffe-dist

```bash
cd $HOME/caffe/build
cmake -DCMAKE_INSTALL_PREFIX=$HOME/perseus-caffe-dist -DBLAS=open -DUSE_PERSEUS=ON -DPERSEUS_LIBRARY_PATH=/root/caffe/libperseus-caffe.so ..
make all -j16 && make instal
```

第三步，运行分布式的Caffe训练命令

以下命令表示在两台机器上运行，每台机器有八块GPU卡（–npernode 8），总共有16个进程（–np 16）

【注】这里运行Caffe train 命令不用指定gpu id，Ali–Perseus–Training会根据起的进程数自动进行GPU分配。hostfile如上例，指定进行分布式运行的机器网址。

```bash
mpirun --allow-run-as-root -np 16 -npernode 8 \
    -machinefile /root/hostfile \
    --mca bind-to none \
    --mca btl_tcp_if_include eth0 \
    --mca orte_keep_fqdn_hostnames t \
    -x NCCL_IB_DISABLE=1 \
    -x NCCL_SOCKET_IFNAME=eth0 \
    -x LD_LIBRARY_PATH \
    -x NCCL_DEBUG=INFO \
    /root/perseus-caffe-dist/bin/caffe train --solver solver.prototxt
```