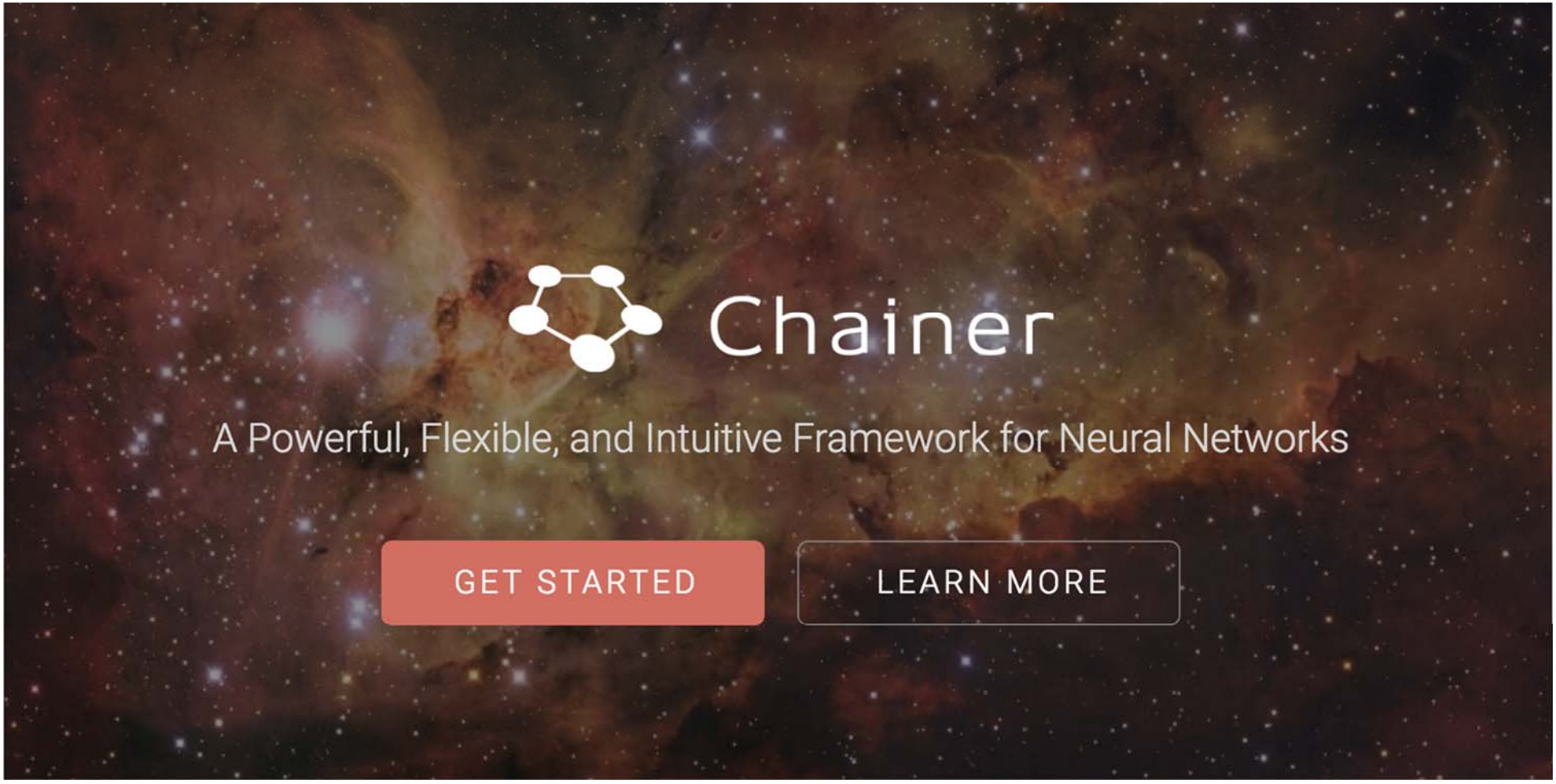
第1回
ディープラーニング分散学習ハッカソン
<ChainerMN紹介 + スパコンでの実行方法>

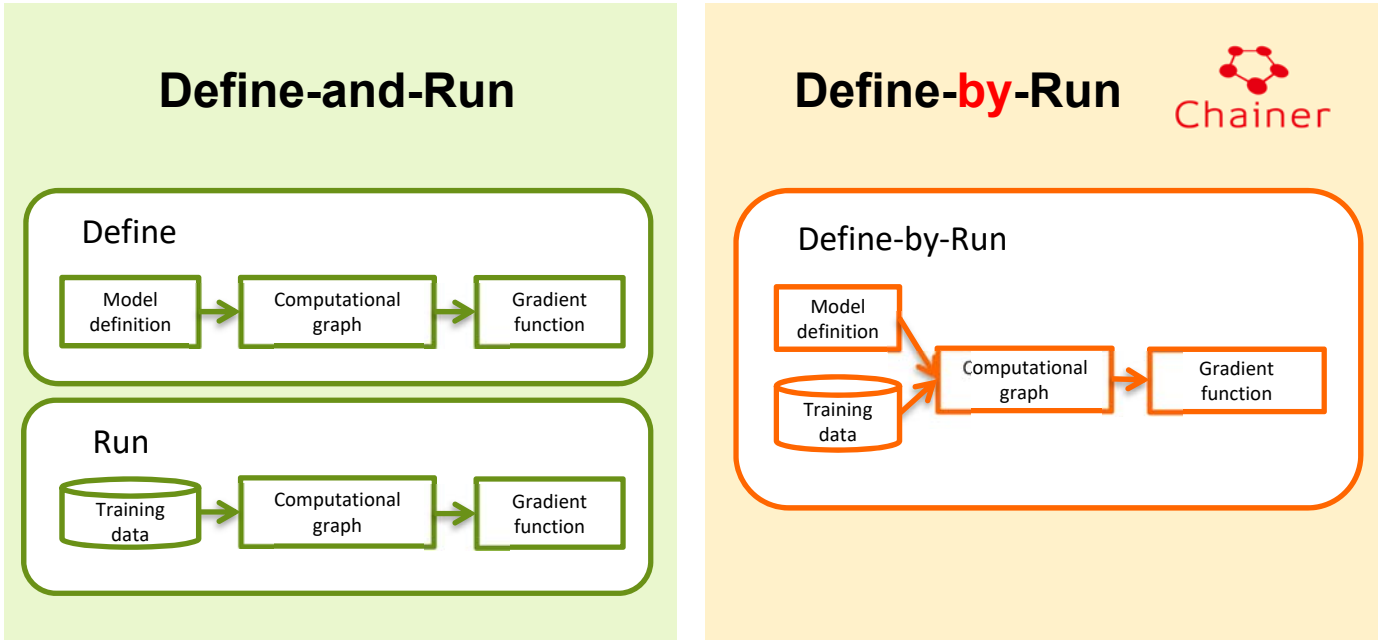
チューター
福田圭祐（PFN）
鈴木脩司（PFN）



https://chainer.org/

Preferred Networks

Chainer:
A Flexible Deep Learning Framework



Caffe2, TensorFlow etc.

PyTorch, TensorFlow(Eager Execution) etc.

Preferred Networks

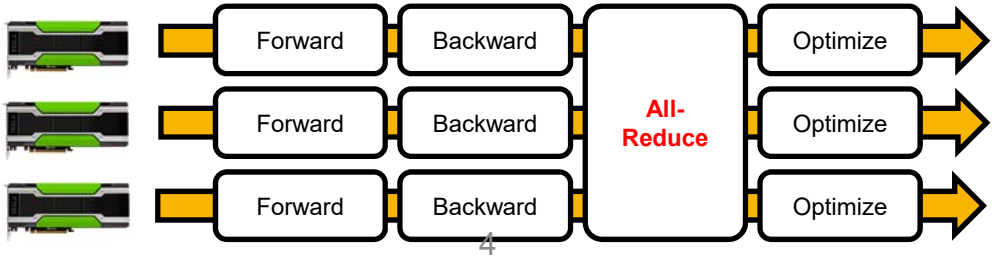
ChainerMN: Distributed Training with Chainer

- Bundled with Chainer (from Chainer v5)
- Enables multi-node distributed deep learning using NVIDIA NCCL2

Features

- **Scalable**: Near-linear scaling with hundreds of GPUs
- **Flexible**: Even GANs, dynamic NNs, and RL are applicable

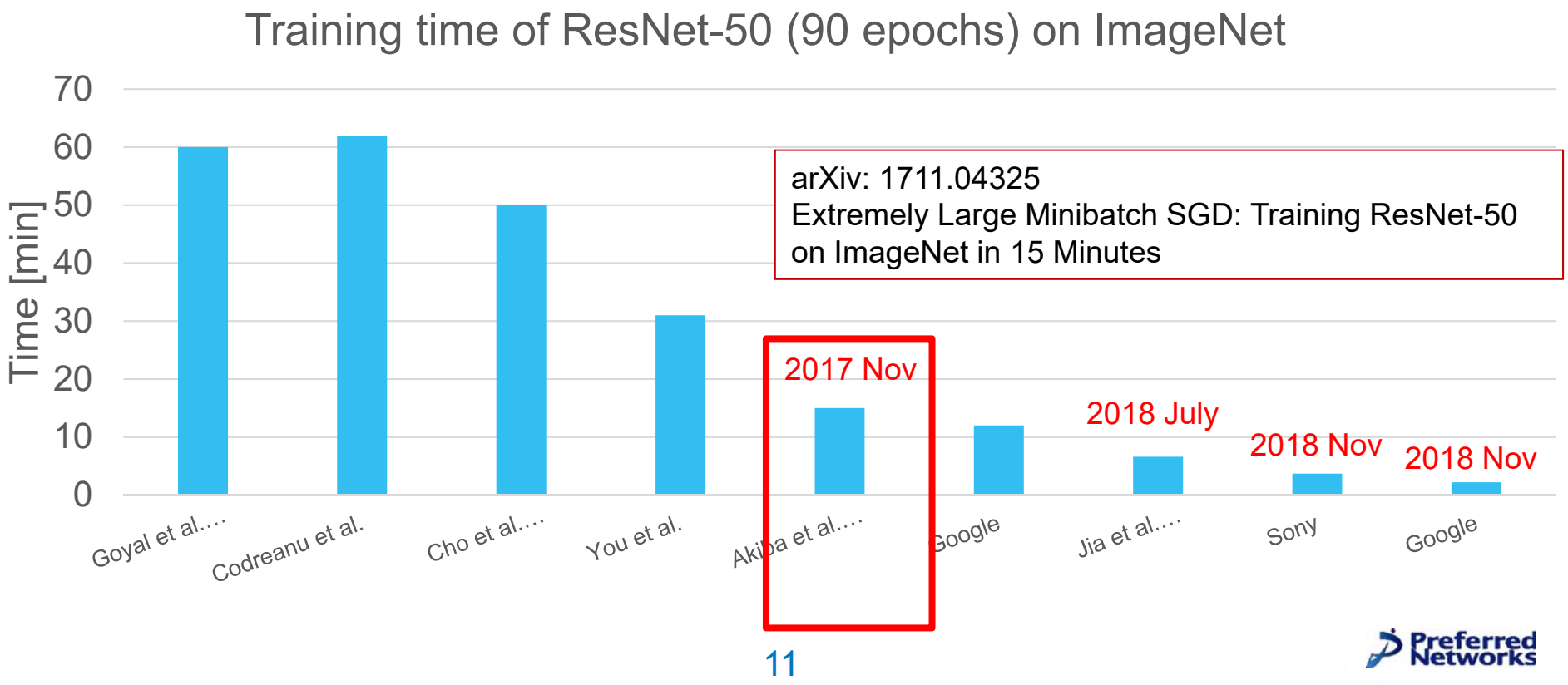
Distributed Training with ChainerMN



Preferred Networks

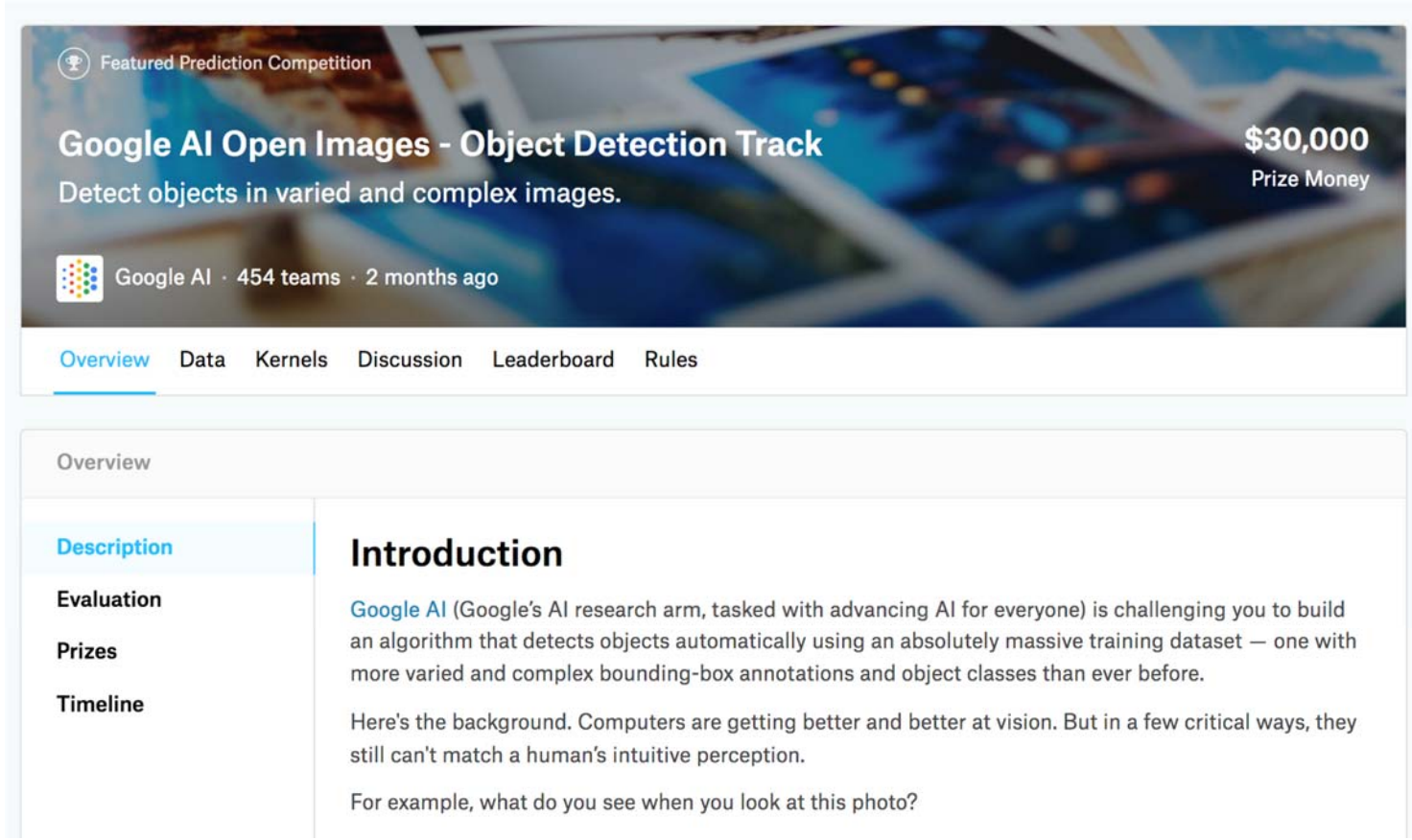
CHAINER活用事例

Achievement on MN-1a: ImageNet in 15 minutes



Preferred Networks

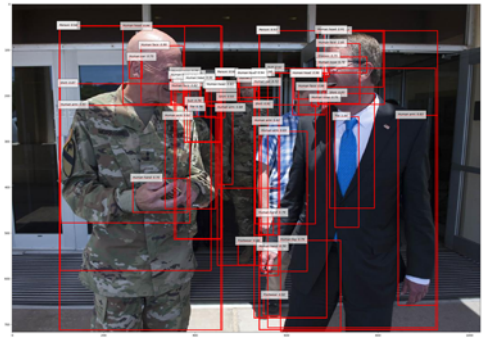
Achievement on MN-1b: PFDet in OIC 2018



Preferred Networks

Achievement on MN-1b: PFDet in OIC 2018

- Google AI Open Images - Object Detection Track
  - Competition using Largest-class image dataset
  - 12 million bounding boxes, 1.7 million images
  - 454 competitiors
  - Approx. 500GB (annotated subset)
- Object detection: much harder than object recognition task



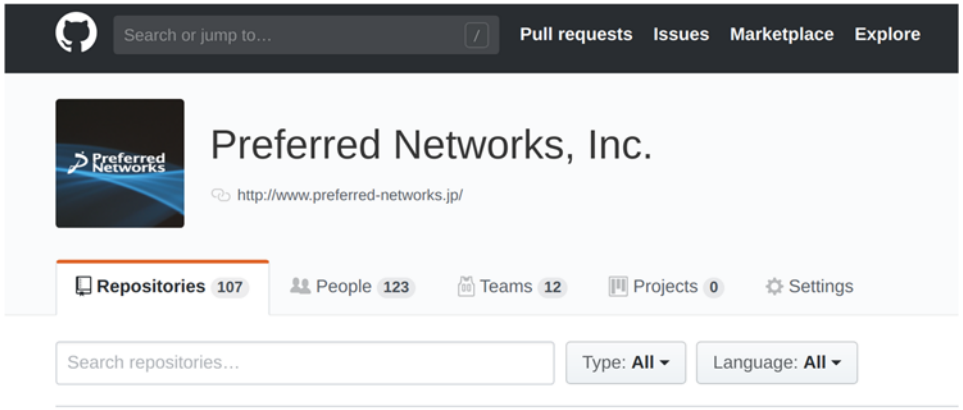
Preferred Networks

Achievement on MN-1b: PFDet in OIC 2018

- We won the 2nd position (0.023% diff to the 1st)

#	△pub	Team Name	Kernel	Team Members	Score	Entries	Last
1	▲2	kivajok			0.58657	102	2mo
2	▼1	PFDet		+3	0.58634	49	2mo
3	▼1	Avengers			0.58616	64	2mo
4	—	XJTU			0.58348	22	2mo
5	—	ikciting		+5	0.56801	39	2mo
6	—	Sogou_MM			0.53909	105	2mo
7	—	QLearning			0.53709	20	2mo

Opensourcing PFDet:



PFDet: 2nd Place Solution to Open Images Challenge 2018 Object Detection Track

Takuya Akiba* Tommi Kerola* Yusuke Niitani* Toru Ogawa* Shotaro Sano* Shuji Suzuki*
Preferred Networks, Inc.
{akiba,tommi,niitani,ogawa,sano,ssuzuki}@preferred.jp

Abstract

We present a large-scale object detection system by team PFDet. Our system enables training with huge datasets using 512 GPUs, handles sparsely verified classes, and massive class imbalance. Using our method, we achieved 2nd place in the Google AI Open Images Object Detection Track 2018 on Kaggle. [1]

1. Introduction

Open Images Detection Dataset V4 (OID) [6] is currently the largest publicly available object detection dataset, including 1.7M annotated images with 12M bounding boxes. The diversity of images in training datasets is the driving force of the generalizability of machine learning models. Successfully trained models on OID would push the frontier of object detectors with the help of data.

Training a deep learning model on OID with low parallelization would lead to prohibitively long training times, as is the case for training with other large-scale datasets [2]. We follow the work of MegDet [11] and use multi-node batch normalization to stably train an object detector with batch size of 512. Using ChainerMN [1], a distributed deep learning library, we demonstrate highly scalable parallelization over 512 GPUs.

OID is different from its predecessors, such as MS COCO [8], not merely in terms of the sheer number of images, but also regarding the annotation style. In the predecessors, instances of all classes covered by the dataset are always exhaustively annotated, whereas in OID, for each image, instances of classes not verified to exist in the image are not annotated. This is a realistic approach to expanding the number of classes covered by the dataset, because without sparsifying the annotated classes, the number of annotations required may explode as the total number of classes increases.

*The authors contributed equally and they are ordered alphabetically.
[1] https://www.kaggle.com/c/google-ai-open-images-object-detection-track

The problem with sparsifying the annotated classes is that most of the CNN-based object detectors learn by assuming that all regions outside of the ground truth boxes belong to the background. Thus, in OID, these learning methods would falsely treat a bounding box as the background when an unverified instance is inside the box. We find that the sparse annotation often leads to invalid labels, especially for classes that are parts of the other classes, which we call *part classes* and *subject classes*, respectively. For instance, a human arm usually appears inside the bounding box of a person. Based on this finding, we propose *co-occurrence loss*. For bounding box proposals that are spatially close to the ground truth boxes with a subject class annotation, co-occurrence loss ignores all learning signals for classifying the part classes of the subject class. This reduces noise in the training signal, and we found this leads to a significant performance improvement for part classes.

In addition to the previously mentioned uniqueness of OID, the dataset poses an unprecedented class imbalance for an object detection dataset. The instances of the rarest class *Pressure Cooker* are annotated in only 13 images, but the instances of the most common class *Person* are annotated in more than 800k images. The ratio of the occurrence of the most common and the least common class is 183 times larger than in MS COCO [8]. Typically, this class imbalance can be tackled by over-sampling images containing instances of rare classes. However, this technique may suffer from degraded performance for common classes, as the number of images with these classes decreases within the same number of training epochs.

As a practical method to solve class imbalance, we train models exclusively on rare classes and ensemble them with the rest of the models. We find this technique beneficial especially for the first 250 rarest classes, sorted by their occurrence count.

Our final model integrates solutions to the three noteworthy challenges of the OID dataset: a large number of images, sparsely verified classes, and massive class imbalance. We use Feature Pyramid Network (FPN) [7] with SE-ResNeXt-101 and SENet-154 [4] as backbones trained with

1

Technical report is already on arXiv:
arXiv:1809.00778

Preferred Networks

PFNの計算環境

MN-1: An in-house supercomputer

- **MN-1a** (Sep. '17~)
  - 1024 NVIDIA Tesla P100
  - InfiniBand FDR
  - Peak 9.3 Peta FLOPS (SP)
  - #227 in Top500 Nov. 2018
- **MN-1b** (July. '18~)
  - 512 NVIDIA Tesla V100 32GB
  - InfiniBand EDR
  - Peak 56 Peta (tensor) Flops

- Targeting Exa FL ops by 2020

Preferred Networks

CHAINERプログラムを秒速でMN化する手順

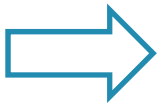
ChainerMNの動作の概要

- ChainerMNは、MPIという通信ライブラリの上に構築されています
  - （MPI＝並列計算で標準的に使われる通信ライブラリ）

- 要点
  - `mpiexec`というコマンドで起動する（後述）
  - 基本的に、サーバー／クライアントという概念はない（全プロセスが互いに対等）
  - 個々のプロセスはRankというプロセス番号を持つ（0～N-1）
  - すべてのプロセスが同じプログラムを実行する（ただし、Rankによって条件分岐することが可能）

```
device = …
chainer.cuda.get_device_from_id(device).use()
…
optimizer = chainer.optimizers.Adam()
```

これだけの変更で（最小限の）MN化！



```
import chainermn
…
comm = chainermn.create_communicator('pure_nccl')
device = comm.intra_rank
chainer.cuda.get_device_from_id(device).use()
…
optimizer = chainermn.create_multi_node_optimizer(
      chainer.optimizers.Adam(), comm)
…
if comm.rank == 0:  # LogReportなどを代表一人だけが出力するようにする
    trainer.extend( … )
```

MN化のその後

- まずは秒速MN化でReedbush上で動作確認をしましょう
- その後やること：
  - `scatter_dataset`
    （プロセス間で学習データに重複が無いように分割）
  - `evaluator`の並列化
    （現在はrank 0だけのevaluationの結果が記録される）
  - Multi node BN （必要に応じて）

普段、どうやって学習を実行していますか？
（フレームワークによらず）

1. 研究室／自社クラスタ
2. クラウド（AWS etc.）
3. スパコン

Reedbushスパコンにおける**Chainer**の使い方

- スパコンは、普通の計算機と環境が（すこし）違います

- 主な違い：
  - 計算機の種類が、**計算ノード**と**ログインノード**に分かれている
  - 計算実行するために、**ジョブスケジューラ**と呼ばれるソフトウェアを使う必要がある
  - 計算ノードからは、インターネット環境にアクセスできない
  - 計算ノードからは、HOME も見えない

Preferred Networks

Reedbushスパコンにおける**Chainer**の使い方

- スパコンは、普通の計算機と環境が（すこし）違います

- 主な違い：
  - 計算機の種類が、**計算ノード**と**ログインノード**に分かれている
  - 計算実行するために、**ジョブスケジューラ**と呼ばれるソフトウェアを使う必要がある
  - 計算ノードからは、インターネット環境にアクセスできない
  - 計算ノードからは、HOME も見えない

Preferred Networks

Reedbushスパコンにおける**Chainer**の使い方

- なので… 以下のことを実行しておく必要があります
  - インターネットからダウンロードの必要があるものは、あらかじめログインノード上でダウンロードしておく
  - Lustreという共有ファイルシステムを事実上のHOMEとして使う
  - PythonモジュールもLustre上にインストールする
  - 環境変数に気をつける（non-interactive環境になるため）
  - ジョブスクリプトと呼ばれるものを書く必要がある

Reedbushスパコンにおける**Chainer**の使い方

使い方説明：

https://bit.ly/2MsNzuA

Preferred Networks

Reedbushスパコンにおける**Chainer**の使い方

手順（1）

Quick Start Guideの項目4〜14を参考に、初期設定を行い、ログインできることを確認する

Preferred Networks

Reedbushスパコンにおける**Chainer**の使い方

環境設定

計算ノードからはHOMEは見えないので、Lustreを事実上のHOMEとして用いる。

```
$ cd /lustre/$(id -ng)/$USER/
```

id -ng コマンドでグループ名
$USER 環境変数（あるいは **id -nu** コマンド)でユーザー名がわかる

Reedbushスパコンにおける**Chainer**の使い方

環境設定ファイル `env.sh` を作成する（内容は、このgistの別ファイル参照）

```
$ vi env.sh
```

好きなエディタで env.sh の内容を作成して、
/lustre/$(id -ng)/$USER/env.sh として保存しましょう
（完全コピペでOK）

Preferred Networks

Reedbushスパコンにおける**Chainer**の使い方

env.shを読み込む。この操作はログインのたびに毎回行うので、 `.bash_profile` 等に書くと良い。

```
$ . /lustre/$(id -ng)/$USER/env.sh
```


Preferred Networks

Reedbushスパコンにおける**Chainer**の使い方

pip

次に、pipで必要なモジュールをインストールする。env.sh内に定義されている `PYTHONUSERBA`
トールされる

```
# pip install は、ファイルシステムの調子によって、数分〜5分かかる場合があるので気長に待つ
$ pip install --user mpi4py
$ pip install --user cupy-cuda91==5.1.0
$ pip install --user chainer==5.1.0
```

cupy ではなく、 cupy-cuda91 を使いましょう

きちんとバージョンを指定しましょう

Preferred Networks

Reedbushスパコンにおける**Chainer**の使い方

MNISTの実行準備

ChainerのMNISTサンプルは、初回実行時に `HOME` ディレクトリにMNISTデータをダウンロードする。計算ノードからはインターネットにアクセスできないので、ログインノードでMNISTを一回実行してデータをダウンロードさせる。この実行はChainerMNである必要はない。（まちがえてmasterブランチのtrain_mnist.pyをダウンロードするとエラーで実行できないので注意）（なお、手動でデータをコピーしても良い）

```
# NOTE: env.shの実行を忘れないように
$ wget https://raw.githubusercontent.com/chainer/chainer/v5.1.0/examples/mnist/train_mnist.py -O train_mnis
$ python train_mnist_single.py -e 1
```

次に、ChainerMN用の train_mnist.py をダウンロードする。（まちがえてmasterブランチのtrain_mnist.pyをダウンロードするとエラーで実行できないので注意）

```
$ wget https://raw.githubusercontent.com/chainer/chainer/v5.1.0/examples/chainermn/mnist/train_mnist.py
```

最後に、ジョブを実行するためのジョブスクリプトを記述する（内容はこのGistの別ファイル参照）

Preferred Networks

Reedbushスパコンにおける**Chainer**の使い方

最後に、ジョブを実行するためのジョブスクリプトを記述する

```
$ vi job.sh
```


Preferred Networks

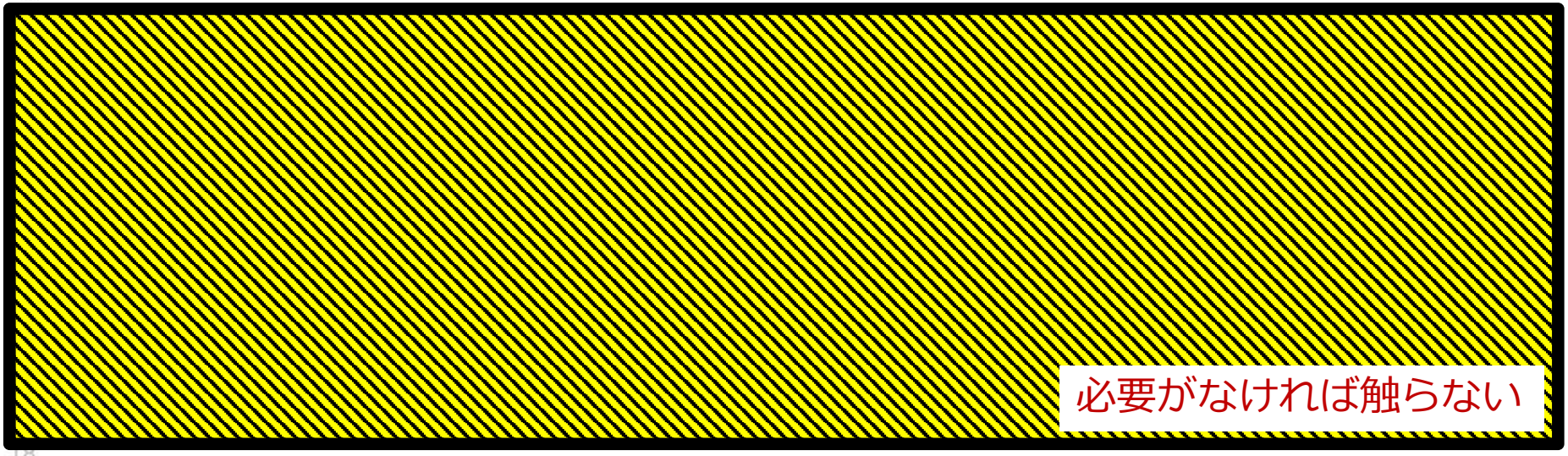
job.sh

```
#!/bin/bash
#PBS -q h-debug               # <-------------- 投入するキューを指定する
#PBS -l select=2:mpiprocs=2:ompthreads=1  #<-- 実行したいノード数と、ノードあたりプロセス数を記述する(ここでは合計4プロセス)
#PBS -W group_list=pz0xxx     # <-------------- 自分のグループ名に書き換える
#PBS -l walltime=00:15:00     # <-------------- 実行時間上限を指定する

GROUP=$(id -ng)
MYDIR=/lustre/${GROUP}/${USER}
export HOME=$MYDIR

. $MYDIR/env.sh

cd $MYDIR

# デバッグ出力: Chainer と CuPy のバージョンを表示
python -c "import chainer; print(chainer.__version__)"
python -c "import cupy; print(cupy.cuda.nccl.get_version())"

mpiexec -x PYTHONUSERBASE \
        python ./train_mnist.py -g --communicator pure_nccl

```

job.sh

```
1  #!/bin/bash
2  #PBS -q h-debug                # <-------------- 投入するキューを指定する
3  #PBS -l select=2:mpiprocs=2:ompthreads=1   #<-- 実行したいノード数と、ノードあたりプロセス数を記述する(ここでは合計4プロセス)
4  #PBS -W group_list=pz0xxx      # <-------------- 自分のグループ名に書き換える
5  #PBS -l walltime=00:15:00      # <-------------- 実行時間上限を指定する
```

ジョブを投げるたびに調整するところ




```
19 mpiexec -x PYTHONUSERBASE \
20         python ./train_mnist.py -g --communicator pure_nccl
21
```

あなたのプログラム

Reedbushスパコンにおける**Chainer**の使い方

```
#PBS -l select=2:mpiprocs=2:ompthreads=1
```

ノード数＝２　　ノードあたりのGPU数＝２

→　**２ｘ２＝４GPUで実行**

- 「ノードあたりのGPU数」は増やせないので２で固定がおすすめ
- 「ノード数」を変更することで、全体のGPU数を調整
- 開発・デバッグ中は「１ｘ２」もしくは「２ｘ２」くらいがおすすめ

Reedbushスパコンにおける**Chainer**の使い方

おまじない

```
mpiexec -x PYTHONUSERBASE ¥
```

```
python ./train_mnist.py -g --communicator pure_nccl
```

引数はスクリプトの書き方によるが、
コミュニケーターは pure_nccl がおすすめ
（というか、プログラムに書いてしまっても良い）

Preferred Networks