MLBench Documentation

Release 2020

MLBench development team

Jul 02, 2021

MLBENCH

A public and reproducible collection of reference implementations and benchmark suite for distributed machine learning algorithms, frameworks and systems.

- Project website: https://mlbench.github.io/
- Free software: Apache Software License 2.0
- Documentation: https://mlbench.readthedocs.io.

CHAPTER

ONE

FEATURES

- For reproducibility and simplicity, we currently focus on standard **supervised ML**, including standard deep learning tasks as well as classic linear ML models.
- We provide **reference implementations** for each algorithm, to make it easy to port to a new framework.
- Our goal is to benchmark all/most currently relevant **distributed execution frameworks**. We welcome contributions of new frameworks in the benchmark suite.
- We provide **precisely defined tasks** and datasets to have a fair and precise comparison of all algorithms, frameworks and hardware.
- Independently of all solver implementations, we provide universal **evaluation code** allowing to compare the result metrics of different solvers and frameworks.
- Our benchmark code is easy to run on **public clouds**.

CHAPTER

TWO

REPOSITORIES

MLBench consists of 5 Github repositories:

- Documentation: http://github.com/mlbench/mlbench-docs
- Helm Charts for Kubernetes: http://github.com/mlbench/mlbench-helm
- Python Core Library: http://github.com/mlbench/mlbench-core
- Benchmark Implementations: http://github.com/mlbench/mlbench-benchmarks
- Dashboard: http://github.com/mlbench/mlbench-dashboard

CHAPTER

THREE

COMMUNITY

Mailing list: https://groups.google.com/d/forum/mlbench

Contact Email: mlbench-contact@googlegroups.com

3.1 Benchmarks

The results can be found here: *benchmark-task-results*

3.1.1 Benchmark Metrics

The basic metric for comparison is *Time to Accuracy*, i.e. training time of the system until a specified target accuracy is reached (where accuracy will be test and/or training accuracy).

The variable dimensions are:

- Algorithm - limited number of prescribed standard algorithms, according to strict reference implementations provided
- Hardware - GPU(s) - CPU(s) - Memory
- Scalability - Number of workers
- Network - Impact of bandwidth and latency

3.1.2 Benchmark Task Descriptions

We here provide precise descriptions of the official benchmark tasks. The task are selected to be representative of relevant machine learning workloads in both industry and in the academic community. The main goal here is a fair, reproducible and precise comparison of most state-of-the-art algorithms, frameworks, and hardware.

For each task, we provide a reference implementation, as well as results for different systems.

Task 0: Communication Backend Raw Performance

This task consists of benchmarking the communication backends for different frameworks and operations.

0.a All-reduce

In this task, tensors of increasing size in `np.logspace(0, 8, num=80)` are sent between workers, 100 times for each tensor size. This allows for measuring and comparing the communication times for each backend for an *all-reduce* operation.

Task 1: Image Classification

1a. Resnet-20, CIFAR-10

Image classification is one of the most important problems in computer vision and a classic example of supervised machine learning.

1. **Model** We benchmark two model architectures of Deep Residual Networks (ResNets) based on prior work by He et al. The first model (m1) is based on the ResNets defined in [HZRS15]. The second version (m2) is based on the ResNets defined in [HZRS16]. For this benchmark implementation, we use 20 layers ResNet called ResNet-20 using the first version stated previously.
2. **Dataset** The CIFAR-10 dataset containing a set of images used to train machine learning and computer vision models. It contains 60,000 32x32 color images in 10 different classes, with 6000 images per class. The 10 different classes represent airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks.

   The train / test split as provided in the dataset is used. The test dataset contains 10,000 images with exactly 1000 randomly-selected images per each class. The rest 50,000 images are training samples.
3. **Training Algorithm** We use standard synchronous SGD [DAM+16] as the optimizer (distributed mini-batch SGD with synchronous all-reduce communication before each mini-batch update).
   - number of machines `k = 1, 2, 4, 8, 16, 32`
   - minibatch size per worker `b = 128`
   - maximum epochs: 164
   - learning rate
     - learning rate η : 0.02
     - decay: We reduce the learning rate when a plateau in the validation loss is reached for 2 consecutive epochs
     - scaling and warmup: apply `linear scaling rule` mentioned in [GDollarG+17]. The learning rate is scaled from η to $\eta \times k$ within the first $log_2(num\_workers)$.
   - optimizer: `CentralizedSGD(momentum=0.9, nesterov=True, weight_decay=1e-4, dampening=0, by_layer=False)`
   - loss : `CrossEntropyLoss`

   Besides, in each round workers access disjoint set of datapoints.

Implementation details:

1. **Data Preprocessing** We followed the same approach described in [HZRS15].

2. **Selection of Framework & Systems** We aim to provide the same algorithm in multiple frameworks, primarily focusing on PyTorch and Tensorflow. For the systems, kubernetes allows easy transferability of our code. While initial results reported are from google kubernetes engine, AWS will be supported very soon.
3. **Environments for Scaling Task** We use a single process per node environment, with one GPU per process (i.e. one GPU per node). The bandwidth between two nodes is around 7.5Gbit/s. `MPI`, `GLOO` or `NCCL` are used for communication.

1b. Resnet-?, ImageNet

TODO

Task 2: Linear Learning

2.a Logistic Regression, Epsilon 2008

1. **Model** We benchmark Logistic Regression with L2 regularization.
2. **Dataset** The epsilon dataset is an artificial and dense dataset which is used for Pascal large scale learning challenge in 2008. It contains 400,000 training samples and 100,000 test samples with 2000 features.
3. **Training Algorithm** We use standard synchronous SGD [DAM+16] as the optimizer (that is distributed mini-batch SGD with synchronous all-reduce communication before each mini-batch).
   - number of machines `k = 1, 2, 4, 8, 16`
   - minibatch size per worker `b = 128`
   - maximum epochs: 164
   - learning rate
     - learning rate η : 4
     - decay: We reduce the learning rate when a plateau in the validation loss is reached for 2 consecutive epochs
     - scaling: The learning rate is scaled from η to $\eta \times k$ for k workers
   - optimizer: `CentralizedSGD(momentum=0, nesterov=False, weight_decay=0, dampening=0, by_layer=False)`
   - loss: `BCELossRegularized` (Binary Cross-Entropy Loss with regularization)
   - regularization parameters: $L1 = 0, L2 = 0.0000025$

Implementation details:

1. **Data Preprocessing** Dataset is pre-processed prior to training, and stored on [here](https://storage.googleapis.com/mlbench-datasets/libsvm). The pre-processing script can be found under *mlbench_core/dataset/util/pytorch/libsvm.py*, and generates *.lmdb* files from the original dataset. One can easily generate the used dataset by running:

```
$ python mlbench_core/dataset/util/pytorch/libsvm.py epsilon [test | train]␣
↪[dest_dir]
```

2. **Selection of Framework & Systems** We aim to provide the same algorithm in multiple frameworks, primarily focusing on PyTorch and Tensorflow. For the systems, kubernetes allows easy transferability of our code. While initial results reported are from google kubernetes engine, AWS will be supported very soon.

3. **Environments for Scaling Task** We use a single process per node environment, with no GPU acceleration. The bandwidth between two nodes is around 7.5Gbit/s. `MPI` or `GLOO` are used for communication.

Task 3: Language Modelling

3a. LSTM, Wikitext2

1. **Model** We benchmark the AWD-LSTM model.
2. **Dataset** The Wikitext2 dataset is used. contains text for language modelling. The train set contains 2088628 tokens, and the validation set 217646 tokens. The vocabulary is made up of 33278 words.
3. **Training Algorithm** We use standard synchronous SGD as the optimizer (that is distributed mini-batch SGD with synchronous all-reduce communication before each mini-batch).
   - number of machines `k = 1, 2, 4, 8, 16, 32`
   - minibatch size per worker `b = 80` sentences
   - backpropagation through time: `bptt_len = 70`
   - Sequence length: Sampled at random for each batch using the following rule `X ~ Bernouilli(0.05), seq_len = max(min_seq_len, Normal(bptt_len / (1 + X), 5))`
     - Validation sequence length: 10
   - maximum epochs: 750
   - optimization
     - learning rate per batch $\eta = 30 \times \frac{seq\_len}{bptt\_len}$
     - weight decay: $1.2e - 6$
     - scaling: We use linear warm-up to scale the learning rate η to $\eta \times \sqrt{num\_workers}$
     - warm-up period: We use a warm-up period of $num\_workers * 3$ epochs
     - gradient clipping: Gradient norm is clipped at 0.25
   - **scheduling:**
     - If Perplexity stops improving for 5 epochs, optimizer is switched to Averaged SGD
   - **loss: `CrossEntropyLoss`**
     - Activation regularization $\alpha = 2$
     - Temporal Activation reg. $\beta = 1$

Implementation details:

1. **Selection of Framework & Systems** We aim to provide the same algorithm in multiple frameworks, primarily focusing on PyTorch and Tensorflow. For the systems, kubernetes allows easy transferability of our code.
2. **Environments for Scaling Task** We use a single process per node environment, with no GPU acceleration. The bandwidth between two nodes is around 7.5Gbit/s. `MPI`, `GLOO` or *NCCL* are used for communication.

Task 4: Machine Translation

4.a LSTM, WMT16 EN-DE

1. **Model** We benchmark the GNMT Machine Translation Model [WSC+16], which follows the sequence-to-sequence learning framework, and uses stacked residual LSTM connections in the encoder and decoder modules. The residual connections allow for deeper stacked LSTM layers, as without residuals, the stack typically suffer from vanishing/exploding gradients when too many layers are used.
2. **Dataset** The WMT-16 dataset containing a set of translated sentences from multiple languages. We exclusively use English-German translation from this dataset.

   The model is trained on sentences of maximum length 75 tokens, and tested on sentences of max length 150 tokens.
3. **Training Algorithm** We use Synchronous distributed Adam as the optimizer, which is similar to [DAM+16], but uses Adam's update rule: Before each weight update, gradients on all workers are summed using an `all_reduce` operation; that way, all workers share their gradients and obtain the same weight update.

   Also, this training algorithm uses mixed precision training (explained below).

   - number of machines `k = 1, 2, 4, 8, 16, 32`
   - global minibatch size `b = 2048` sentences[1]
   - maximum epochs: 8
   - learning rate (Figure 1. left plot)
     - `initial_learning_rate = 0.0`
     - `base_learning_rate = 2.0e-3`
     - decay: We decay by 0.5 after having gone through 40% of total training, and then for every 5% for maximum 4 times
     - scaling and warmup: We use 200 warmup steps, where the learning rate is exponentially increased from `initial_learning_rate` to `base_learning_rate`
   - optimizer: `Adam(betas=(0.9, 0.999), eps=1e-8, weight_decay=0, amsgrad=False)`
   - loss: `LabelSmoothingLoss` (Negative Log-Likelihood with smoothing)
   - gradient clipping: max norm of 5.0
   - Loss Scaling
     - `initial_scale = 2**10`
     - `scale_factor = 2` (downscale and upscale)
     - `max_scale = 2**13`
     - `scale_window = 128` (steps after upscale if no overflow/underflow)

Implementation details:

[1] Fitting batch in memory: In order to achieve the mentioned global batch size, one needs to fit a batch size of `bw = b / world_size` on each worker. Depending on the used hardware, this cannot be achieved as it takes up too much memory. For that, we compute the gradients on multiple batches of smaller size `bs` and aggregate them, before applying the weight update. The frequency at which we update is called `update_freq`.
This implies that the global batch size is `b = bs * world_size * update_freq`. For the used hardware, `max(bs) = bs_max = 128` is the maximum value we can fit in memory. Thus, we have `update_freq = max(1, b / (bs_max * world_size))` thus `bs = b / (world_size * update_freq)`

1. **Data Preprocessing** The data needs to be downloaded and pre-processed and tokenized using the pre-processing script *mlbench_core/dataset/nlp/pytorch/wmt16/preprocess/preprocess.py* before training. The pre-processed data is available on our S3
2. **Mixed Precision Training** In order to have faster backward and forward passes, our model's weights and gradients are cast into `float16` prior to training. `float32` weights are still kept in memory and used by the optimizer to update weights. We use our own `FP16Optimizer`. Since `float16` has lower precision than `float32`, it is necessary to have a loss scaler:
   - Start with `loss_scale = initial_scale`
   - Before each backward pass, inflate the loss by `loss_scale` (in `float16`) to avoid under-flows
   - Before weight update, deflate gradients by `loss_scale` (in `float32`) to keep precision
   - Clip gradient norm to be `grad_clip`
   - Check if gradient norm is `nan` or `inf` (in `float16`). If True, `loss_scale = loss_scale / scale_factor`. If False, update weights.
   - If after `scale_window` updates, no overflow/underflow detected, `loss_scale = loss_scale * scale_factor`
3. **Selection of Framework & Systems** We currently only have this reference implementation in PyTorch. For the systems, kubernetes allows easy transferability of our code. While initial results reported are from Google Kubernetes engine, AWS will be supported very soon.
4. **Environments for Scaling Task** We use a single process per node environment, with one GPU per process (i.e. one GPU per node). The bandwidth between two nodes is around 7.5Gbit/s. `MPI` or `NCCL` are used for communication.

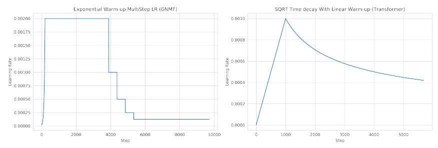


Fig. 1: Figure 1: Learning rate scheduler for GNMT and Transformer

4.b Transformer, WMT17 EN-DE

1. **Model** We benchmark the Transformer Model, using attention mechanisms based on the paper "Attention Is All You need" [VSP+17] that. The implementation is based on a combination of NVIDIA's implementation of fairseq 's transformer. Our implementation differs from MLPerf's in one subtle way: the *FusedLayerNorm* layers are changed to native torch *LayerNorm*, as its performance has increased since. Also, instead of using *FusedAdam*, we use *Adam*. One part of the *MultiheadAttention* module needs a cuda extension, that makes training significantly faster than torch's native *MultiheadAttention*
2. **Dataset** The WMT-17 dataset containing a set of translated sentences from multiple languages. We exclusively use English-German translation from this dataset.

   The model is trained and tested on sentences of maximum length 80 tokens.
3. **Training Algorithm** We use Synchronous distributed Adam as the optimizer, which is similar to [DAM+16], but uses Adam's update rule: Before each weight update, gradients on all workers are summed using an `all_reduce` operation and divided by `world_size * update_frequency`; that way, all workers share their gradients and obtain the same weight update.

   Also, this training algorithm uses mixed precision training (explained below).

- number of machines `k = 1, 2, 4, 8, 16, 32`
- global minibatch size `b = 2**17` tokens[2].
- maximum epochs: 10
- learning rate (Figure 1. right plot)
  - `initial_learning_rate = 0.0`
  - `base_learning_rate = 1.976e-3`
  - decay: We decay by $\sqrt{N}$ after warmup
  - scaling and warmup: We use 1000 warmup steps, where the learning rate is linearly increased from `initial_learning_rate` to `base_learning_rate`
- optimizer: `Adam(betas=(0.9, 0.98), eps=1e-9, weight_decay=0, amsgrad=False)`
- loss: `LabelSmoothingLoss` (Negative Log-Likelihood with smoothing)
- Loss Scaling
  - `initial_scale = 2**7`
  - `scale_factor = 2` (downscale and upscale)
  - `scale_window = 2000` (steps after upscale if no overflow/underflow)

Implementation details:

1. **Data Preprocessing** The data needs to be downloaded and pre-processed and tokenized using the pre-processing script *mlbench_core/dataset/nlp/pytorch/wmt17/preprocess/preprocess.py* before training. The pre-processed data is available on our S3 storage
2. **Mixed Precision Training** In order to have faster backward and forward passes, our model's weights and gradients are cast into `float16` prior to training. `float32` weights are still kept in memory and used by the optimizer to update weights. We use our own *FP16Optimizer*. Since `float16` has lower precision than `float32`, it is necessary to have a loss scaler:
   - Start with `loss_scale = initial_scale`
   - Before each backward pass, inflate the loss by `loss_scaling` (in `float16`) to avoid under-flows
   - Before weight update, deflate gradients by `loss_scaling * full_batch_size / (world_size * update_freq)` (in `float32`) to keep precision, where `full_batch_size` is the batch size over all workers (sum of number of tokens on this batch for each worker).
   - Check if gradient norm is `nan` or `inf` (in `float16`). If True, `loss_scale = loss_scale / scale_factor`. If False, update weights.
   - If after `scale_window` updates, no overflow/underflow detected, `loss_scale = loss_scale * scale_factor`
3. **Selection of Framework & Systems** We currently only have this reference implementation in PyTorch. For the systems, kubernetes allows easy transferability of our code. While initial results reported are from google kubernetes engine, AWS will be supported very soon.
4. **Environments for Scaling Task** We use a single process per node environment, with one GPU per process (i.e. one GPU per node). The bandwidth between two nodes is around 7.5Gbit/s. `MPI` or `NCCL` are used for communication.

[2] Fitting batch in memory: We apply the same technique as in[?]. For the used hardware, we have `max(bs) = bs_max = 8192` and `update_freq = max(1, b / (bs_max * world_size))` to get `bs = b / (world_size * update_freq)`

3.1.3 Benchmark Results

Here we present the results for scaling tasks. All results were generated on the Google Cloud Kubernetes Engine.

Task 0: Communication Backend

0.a PyTorch All-reduce

1. **Frameworks**
   - PyTorch 1.5.0
2. **Communication Backends:**
   - MPI (OpenMPI), GLOO and NCCL
3. **System Hardware**
   - machine type: n1-standard-4 instances on GCP with 15GB memory and 4 virtual CPUs.
   - available CPUs: 3 CPUs available for pod (1 for Kubernetes management)
   - GPU: *NVIDIA® Tesla® T4* (16GB GDDR6, Turing arch)
4. **Pricing**
   - *n1-standard-4*: $0.2092/hour (regular), $0.0440/hour (preemptible)
   - *NVIDIA® Tesla® T4*: $0.35/hour (regular), $0.11/hour (preemptible)

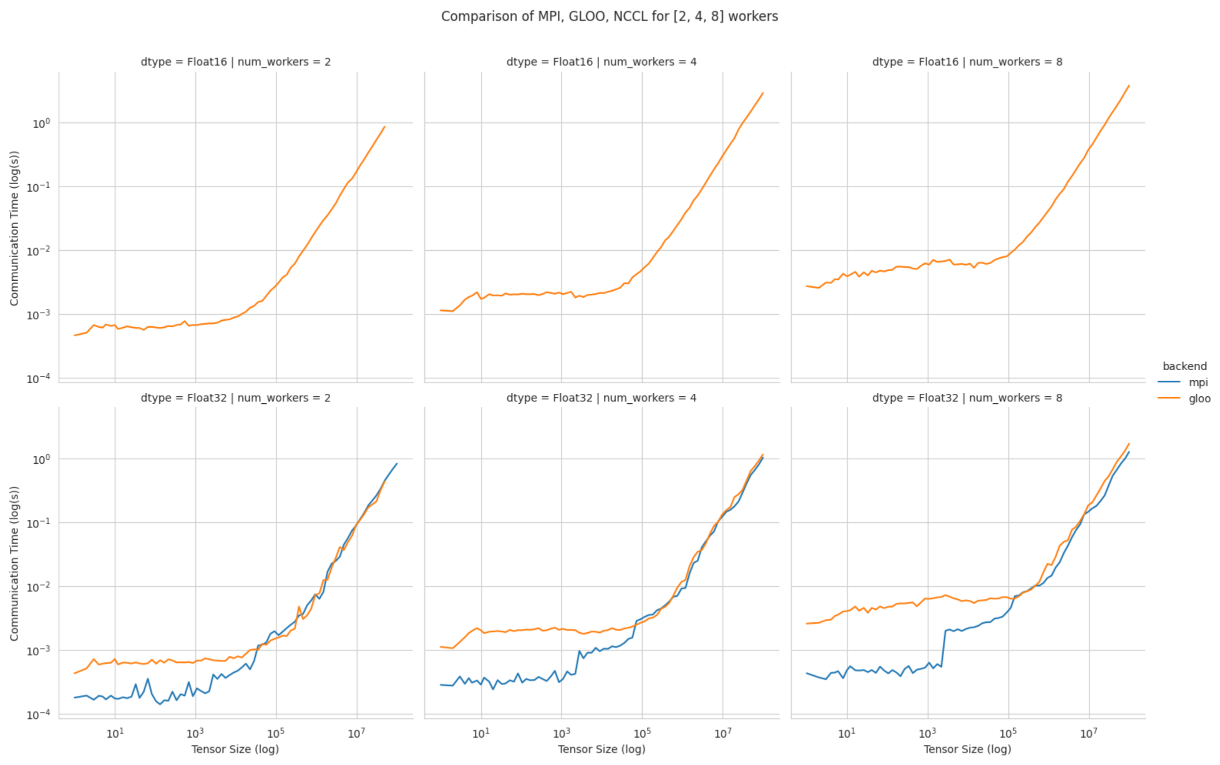


Fig. 2: Communication for 2 to 8 workers (CPU tensors)

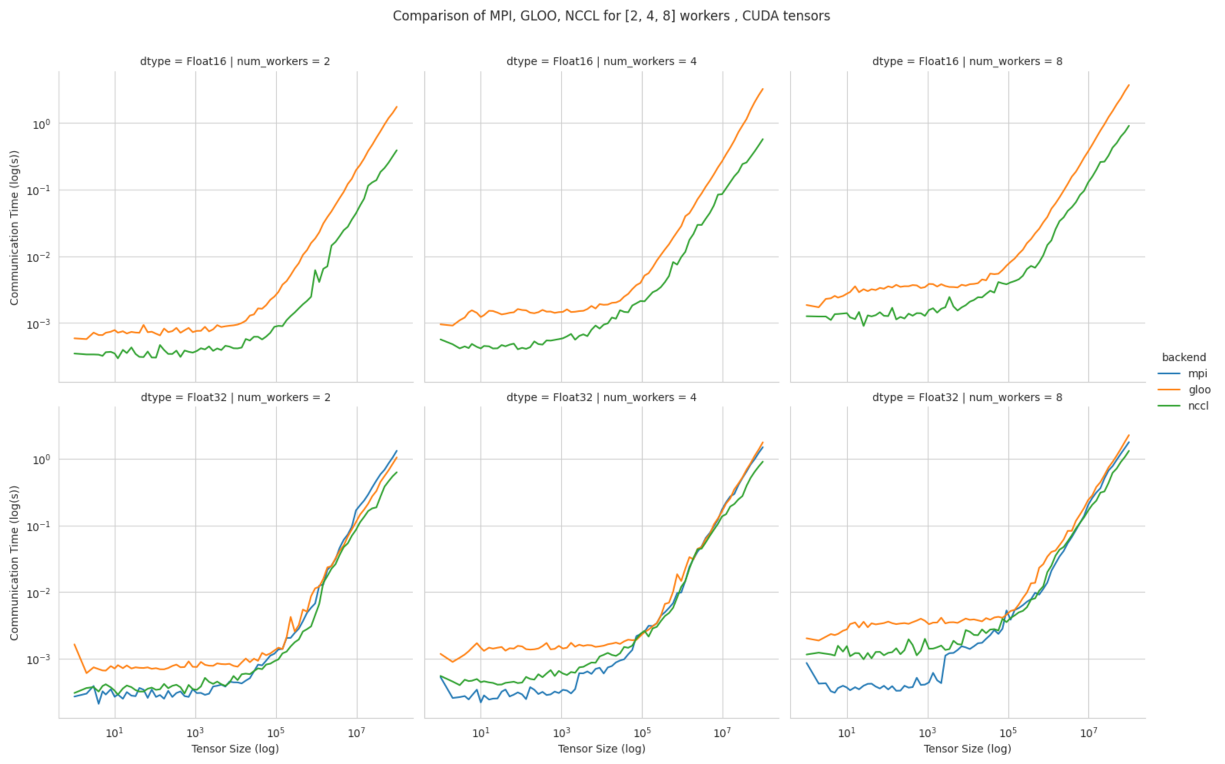


Fig. 3: Communication for 2 to 8 workers (GPU tensors)

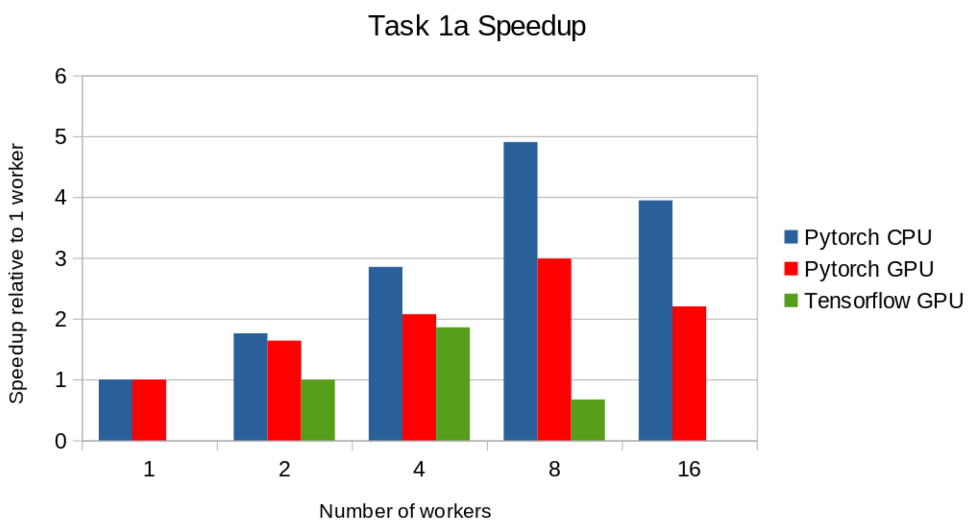
- The first figure shows the communication times between 2, 4, and 8 workers for `float32` and `float16` CPU tensors, for GLOO and MPI backends.
- The second figure shows the communication times between 2, 4, and 8 workers for `float32` and `float16` GPU tensors, for GLOO, MPI and NCCL backends.
- MPI and GLOO both support CPU tensor communication, while NCCL only supports GPU tensors.
- NCCL and GLOO both support `float16` communication, while MPI only supports `float32`.
- This graph allows for a quantitative comparison of the different backends, and to study their advantages/disadvantages.
- We can see that MPI behaves better than GLOO for small `float32` CPU tensors, with similar performance they get larger.
- MPI seems to be less affected by increased cluster sizes.
- MPI and NCCL have comparable performance for 2 workers (for small tensors), but NCCL gets slower as cluster size increases.
- NCCL always has better performance for very large tensors, and supports `float16`, while MPI doesn't.
- GLOO has poor performance compared to others, but has the main advantage to be the only backend supporting `float16` training on CPU.

Task 1: Image Classification

1a. Resnet-20, CIFAR-10

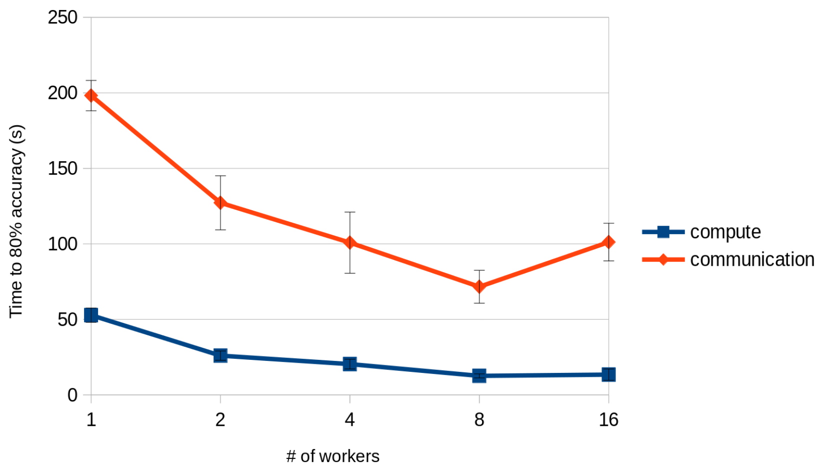
1. **Frameworks**
   - PyTorch 1.5.0
   - Tensorflow
2. **Communication Backends:**
   - MPI (OpenMPI), GLOO and NCCL
3. **System Hardware**
   - machine type: n1-standard-4 instances on GCP with 15GB memory and 4 virtual CPUs.
   - available CPUs: 3 CPUs available for pod (1 for Kubernetes management)
   - GPU: *NVIDIA® Tesla® T4* (16GB GDDR6, Turing arch)
4. **Metric** Time to Accuracy of 80% on validation set.
5. **Pricing**
   - *n1-standard-4*: $0.2092/hour (regular), $0.0440/hour (preemptible)
   - *NVIDIA® Tesla® T4*: $0.35/hour (regular), $0.11/hour (preemptible)

- The next figure shows the speedup in training times to 80% accuracy relative to training on one node[3]. The baseline time for 1 worker for the PyTorch CPU implementation is 5895 s, for the PyTorch GPU implementation 407 s and for the Tensorflow GPU implementation 1191 s.

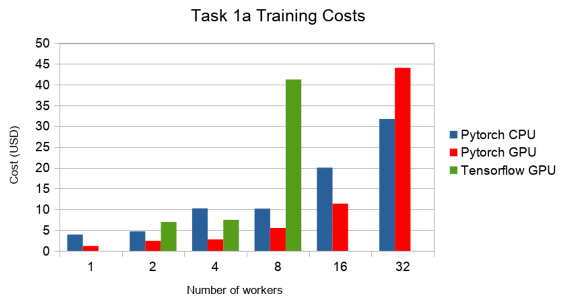


- This figure shows the time spent in compute and communication for the PyTorch GPU implementation on 1, 2, 4, 8 and 16 workers.

[3] Training on CPU shows speedup with increasing number of nodes up to 32 nodes. For the Pytorch implementation on the GPU, speedups plateau at 4 nodes and decrease for 32 nodes. Tensorflow GPU numbers are only available up to 8 nodes, as more nodes lead to an Out-Of-Memory error on the GPU. This benchmark is still a work in progress and this issue will be fixed in a future release. Also since Tensorflow requires at least one parameter-server and a worker to run, it can't be run on a single machine. As such, the results between PyTorch and Tensorflow are not directly comparable. Tuning the Tensorflow parameter-server in size when growing the number of total machines might require further tuning



- The next figure compares the cost of experiment. Note that a regular *n1-standard-4* instance costs $0.19 per hour and a preemptible one costs only $0.04. *NVIDIA® Tesla® K80* GPUs (preemtpible) cost $0.135 per hour. All costs shown are for premtible instances.



1b. Resnet-?, ImageNet

TODO

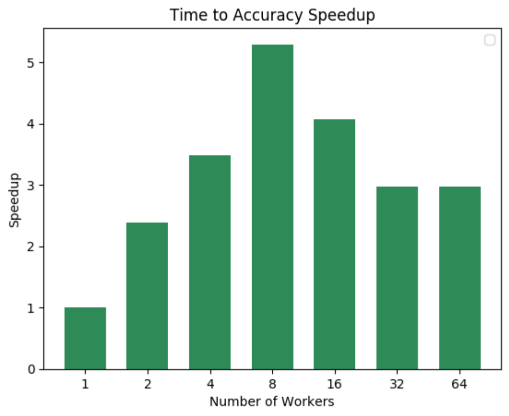
Task 2: Linear Learning

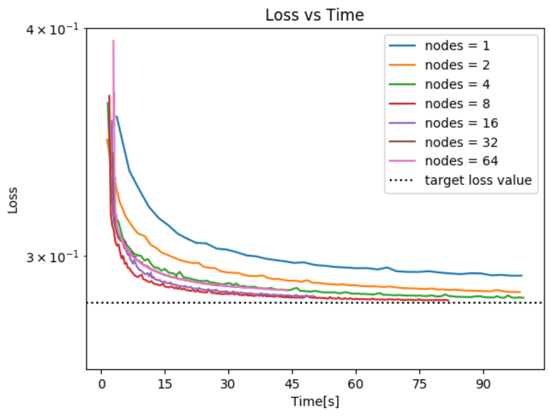
2.a Logistic Regression, Epsilon 2008

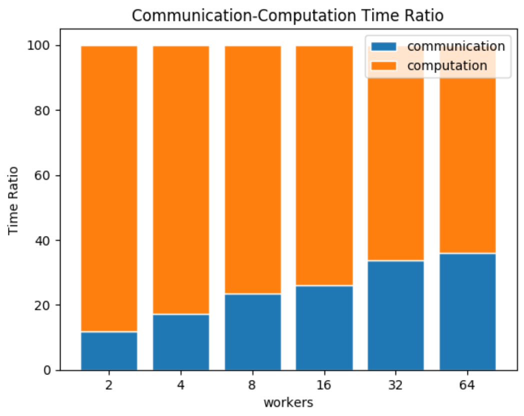
1. **Frameworks**
   - PyTorch 1.5.0
2. **Communication Backends:**
   - MPI (OpenMPI), GLOO
3. **System Hardware**
   - machine type: n1-standard-4 instances on GCP with 15GB memory and 4 virtual CPUs.
   - available CPUs: 3 CPUs available for pod (1 for Kubernetes management)
4. **Metric** Time to Accuracy of 80% on validation set.
5. **Pricing**

  - *n1-standard-4*: $0.2092/hour (regular), $0.0440/hour (preemptible)

- First figure shows the speedup of time to accuracy, for test accuracy of 89%, as the size of the cluster increases. Even though initially the speedup grows with the number of nodes added to the cluster, the benefit starts dropping for a cluster bigger than 16 nodes. This is mostly due to the issue of large-batch training. As the local batch-size of each worker is fixed, the global batch-size increases with the number of workers. Hence, while increasing batch size up to a point makes the training faster, beyond a certain point it will no longer reduce the number of training steps required, making it slower to reach the same accuracy.
- Second figure illustrates how the loss value drops over time for various number of nodes. The black dotted line shows the target loss value, which is 0.2828 for this particular dataset.
- Last figure shows the average communication-computation time ratio for a node in the cluster. As we expected, the more workers we have, the more time is spent in communication.







Task 3: Language Modelling

3a. LSTM, Wikitext2

1. **Frameworks**
   - PyTorch 1.7.0
2. **Communication Backends:**
   - NCCL
3. **System Hardware**

- machine type: n1-standard-4 instances on GCP with 15GB memory and 4 virtual CPUs.
- available CPUs: 3 CPUs available for pod (1 for Kubernetes management)

4. **Metric** Time to Perplexity of 70 on validation set

Task 4: Machine Translation

4.a LSTM, WMT16 EN-DE

1. **Frameworks**
   - PyTorch 1.5.0
2. **Communication Backends:**
   - NCCL
3. **System Hardware**
   - machine type: n1-standard-4 instances on GCP with 15GB memory and 4 virtual CPUs.
   - available CPUs: 3 CPUs available for pod (1 for Kubernetes management)
   - GPU: *NVIDIA® Tesla® T4* (16GB GDDR6, Turing arch)
4. **Metric** Time to BLEU-Score of 24.0 on test set.
5. **Pricing**
   - *n1-standard-4*: $0.2092/hour (regular), $0.0440/hour (preemptible)
   - *NVIDIA® Tesla® T4*: $0.35/hour (regular), $0.11/hour (preemptible)
   - Training on 1 node ~10$

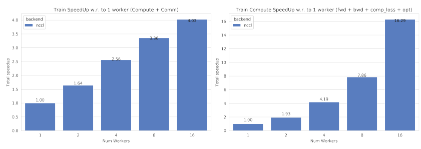


Fig. 4: Speedups for Task 4a: with communication times (left), without (right)

This figures shows the absolute speedup (left), and the compute speedup (right). The compute speedup doesn't account for communication times, and is only used as an indicator to see the maximum achievable speedup with lightspeed communication.

A few interesting points:

- Overall speedups seem to follow logarithmic scaling for this configuration.
- Scaling the number of compute nodes gives perfect linear scaling for this task
- Using more powerful communication hardware (e.g. `NVLink®`) will positively affect speedups.

This figure shows the total time spent in each step for all cluster sizes.

- Total time and compute step times follow an exponential decay with the increase of number of nodes.
- Time spent optimizing doesn't seem to follow the same path, but increases are insignificant (~10 seconds), and are due to additional compute steps (averaging tensors, computations related to Mixed precision) when using distribution

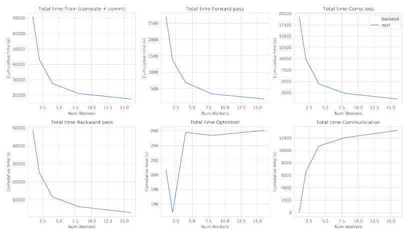


Fig. 5: Step times for task 4a

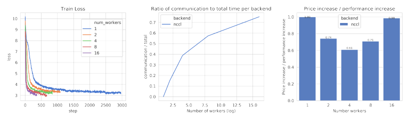


Fig. 6: Train loss (right), Ratio of communication to total time, Price index for Task 4a

- Total communication time increases also logarithmically

This figure shows, the train losses (right), Ratio of communication to total time, and a price index. The price index is computed as follows $index = \frac{price\_increase}{performance\_increase}$

The center graph is useful, as it depicts the limits of distribution for this model, using the described hardware. We can see that after 8 workers, communication takes up more than 50% of total time,

The right-most graph, shows the worthiness of distribution:

- The price increase is less than the performance increase for 2, 4, and 8 workers. This suggests that distribution is worth the price increase
- The 4 workers case seems to be the best price-performance trade-off.
- Training on 4 workers costs ~18$, but is 2.56 times faster.

4.b Transformer, WMT17 EN-DE

1. **Frameworks**
    - PyTorch 1.5.0
2. **Communication Backends:**
    - NCCL
3. **System Hardware**
    - machine type: n1-standard-4 instances on GCP with 15GB memory and 4 virtual CPUs.
    - available CPUs: 3 CPUs available for pod (1 for Kubernetes management)
    - GPU: *NVIDIA® Tesla® T4* (16GB GDDR6, Turing arch)
4. **Metric** Time to BLEU-Score of 25.0 on test set.
5. **Pricing**
    - *n1-standard-4*: $0.2092/hour (regular), $0.0440/hour (preemptible)
    - *NVIDIA® Tesla® T4*: $0.35/hour (regular), $0.11/hour (preemptible)
    - Training on 1 node ~5.5$

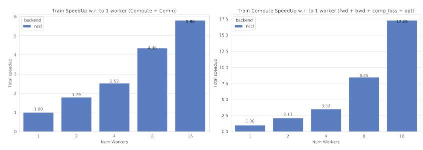


Fig. 7: Speedups for Task 4b: with communication times (left), without (right)

This figures shows the absolute speedup (left), and the compute speedup (right). The compute speedup doesn't account for communication times, and is only used as an indicator to see the maximum achievable speedup with lightspeed communication.

A few interesting points:

- Overall speedups follow a similar path than Task 4a, with even better speedups when not considering communication.
- Linear speedup of compute implies a nearly perfect scaling for this task.
- Using more powerful communication hardware (e.g. `NVLink®`) will also positively affect speedups.

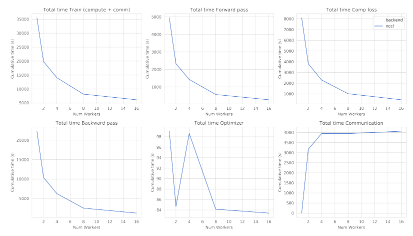


Fig. 8: Step times for task 4b

This figure shows the total time spent in each step for all cluster sizes. We can observe very similar behaviour than Task 4a in all step times.

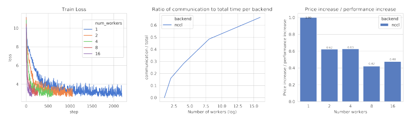


Fig. 9: Train loss (right), Ratio of communication to total time, Price index for Task 4b

This figure shows, the train losses (right), Ratio of communication to total time, and a price index. Communication times ratio is lower than Task 4a for more workers, but still reaches over 50% for 8 workers.

The price index however, has a very different shape:

- All price indices are below one.
- This suggests that distributing this particular model on the mentioned hardware is always beneficial despite the price increase.
- Training Transformer models seems to always benefit from distribution.
- Training on most optimal configuration (8 workers) costs ~9.3$ and is 4.36 times faster.

3.1.4 Benchmark Task Implementations

For details on the available Benchmark implementations, please see Benchmark Implementations .

References

3.2 Prerequisites

3.2.1 Kubernetes

MLBench uses Kubernetes as basis for the distributed cluster. This allows for easy and reproducible installation and use of the framework on a multitude of platforms.

Since mlbench manages the setup of nodes for experiments and uses Kubernetes to monitor the status of worker pods, it needs to be installed with a service-account that has permission to manage and monitor `Pods` and `StatefulSets`.

Additionally, *helm* requires a kubernetes user account with the `cluster-admin` role to deploy applications to a kubernetes cluster.

To use MLBench, one would need to install kubectl

On Ubuntu/Debian:

```
$ sudo apt-get update && sudo apt-get install -y apt-transport-https gnupg2
$ curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo apt-key add -
$ echo "deb https://apt.kubernetes.io/ kubernetes-xenial main" | sudo tee -a /etc/apt/
↪sources.list.d/kubernetes.list
$ sudo apt-get update
$ sudo apt-get install -y kubectl
```

3.2.2 Google Cloud

GCloud SDK (Required)

To use MLBench with GCLoud, it requires the gcloud CLI to be installed and authenticated on the client machine.

On Ubuntu/Debian:

```
$ echo "deb [signed-by=/usr/share/keyrings/cloud.google.gpg] https://packages.cloud.
↪google.com/apt cloud-sdk main" | sudo tee -a /etc/apt/sources.list.d/google-cloud-sdk.
↪list
$ sudo apt-get install apt-transport-https ca-certificates gnupg
$ curl https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo apt-key --keyring /
↪usr/share/keyrings/cloud.google.gpg add -
$ sudo apt-get update && sudo apt-get install google-cloud-sdk
$ gcloud init
```

Note: In order to set you credentials for gcloud, you need to run the commands `gcloud auth login` and `gcloud auth application-default login`

Manually creating a cluster (Optional)

The GCloud SDK allows for manual cluster creation Please refer to Kubernetes Quickstart for more information

If you're planning to use GPUs in your cluster, see the GPUs article, especially the "Installing NVIDIA GPU device drivers" section.

When creating a GKE cluster, make sure to use version `1.15` or above of kubernetes, as there is an issue with DNS resolution in earlier version. You can do this with the `--cluster-version=1.15` flag for the `gcloud container clusters create` command.

Make sure credentials for your cluster are installed correctly (use the correct zone for your cluster):

Example of cluster creation:

```
$ gcloud container clusters create dummy-2 --zone=europe-west1-b \
    --cluster-version="1.15" --enable-network-policy \
    --machine-type=n1-standard-4 --num-nodes=2 --disk-type=pd-standard \
    --disk-size=50 --scopes=storage-full
```

If you would like to add GPU acceleration, add the following parameter `--accelerator type=${GPU_TYPE}, count=${NUM_GPUS}`

3.2.3 Helm (Required)

Helm charts are like recipes to install Kubernetes distributed applications. They consist of templates with some logic that get rendered into Kubernetes deployment *.yaml* files They come with some default values, but also allow users to override those values.

Helm can be found here, and only needs to be installed if manual cluster installation is needed (i.e. manually install MLBench on a cluster)

On Ubuntu/Debian:

```
$ curl https://baltocdn.com/helm/signing.asc | sudo apt-key add -
$ sudo apt-get install apt-transport-https --yes
$ echo "deb https://baltocdn.com/helm/stable/debian/ all main" | sudo tee /etc/apt/
↪sources.list.d/helm-stable-debian.list
$ sudo apt-get update
$ sudo apt-get install helm
```

3.3 Installation

Make sure to read *Prerequisites* before installing mlbench.

Then, the library can be installed directly using `pip`:

```
$ pip install mlbench-core
```

This will install the `mlbench` CLI to the current environment, and will allow creation/deletion of clusters, as well as creating runs.

In addition to installation of CLI, we have provided alternative ways of installing *mlbench-core* depending on the use-case. These are alternatives and allow installation of extra libraries which are not needed for CLI use. Here are all the installation possibilities:

```
$ pip install mlbench-core[test] # Install with test requirements
$ pip install mlbench-core[lint] # Install with lint requirements
$ pip install mlbench-core[torch] # Install only with torch requirements
$ pip install mlbench-core[tensorflow] # Install only with tensorflow requirements
$ pip install mlbench-core[dev] # Install all dependencies for development (all of the␣
↪above)
```

```
$ mlbench --help
   Usage: mlbench [OPTIONS] COMMAND [ARGS]...

   Console script for mlbench_cli.

   Options:
     --version  Print mlbench version
     --help     Show this message and exit.

   Commands:
     charts             Chart the results of benchmark runs Save generated...
     create-cluster     Create a new cluster.
     delete             Delete a benchmark run
     delete-cluster     Delete a cluster.
     download           Download the results of a benchmark run
     get-dashboard-url  Returns the dashboard URL of the current cluster
     list-clusters      List all currently configured clusters.
     run                Start a new run for a benchmark image
     set-cluster        Set the current cluster to use.
     status             Get the status of a benchmark run, or all runs if no...
```

3.3.1 Cluster & Run Deployment (using CLI)

One can easily deploy a cluster on both AWS and GCloud, using the `mlbench` CLI.

For example, one can create a GCloud cluster by running:

```
$ mlbench create-cluster gcloud 3 my-cluster
[...]
MLBench successfully deployed
```

Which creates a cluster called `my-cluster-3` with 3 nodes (See `mlbench create-cluster gcloud --help` for more options).

Once created, experiments can be run using:

```
$ mlbench run my-run 2

Benchmark:

 [0]     PyTorch Cifar-10 ResNet-20
 [1]     PyTorch Cifar-10 ResNet-20 (Scaling LR)
 [2]     PyTorch Linear Logistic Regression
 [3]     PyTorch Machine Translation GNMT
 [4]     PyTorch Machine Translation Transformer
```

(continues on next page)

(continued from previous page)

```
  [5]      Tensorflow Cifar-10 ResNet-20 Open-MPI
  [6]      PyTorch Distributed Backend benchmarking
  [7]      Custom Image


Selection [0]: 1

[...]

Run started with name my-run-2
```

A few handy commands for quickstart:

- To obtain the dashboard URL: `mlbench get-dashboard-url`.
- To see the state of the experiment: `mlbench status my-run-2`.
- To download the results of the experiment: `mlbench download my-run-2`.
- To delete the cluster: `mlbench delete-cluster gcloud my-cluster-3`

Kubernetes in Docker (Debugging only)

MLBench also supports deployment of dashboard and tasks to a local cluster. This uses the KIND technology and can be easily deployed using the CLI. This can be used for debugging code or development, but is not meant to

Click here to download and install KIND.

```
$ mlbench create-cluster kind 3 my-cluster
[...]
MLBench successfully deployed
```

This creates a local cluster of 3 "nodes", as well as a local docker registry on port 5000. This allows for deploying runs using local docker images. To do that, one needs to push their docker image to the local repository:

```
$ docker tag <repo>/<image-name>:<tag> localhost:5000/<image-name>:<tag>
$ docker push localhost:5000/<image-name>:<tag>
```

You can now use the image `localhost:5000/<image-name>:<tag>` in MLBench's dashboard to run a task.

3.3.2 Manual helm chart deployment (Optional)

Helm Chart installation

The manual deployment requires the repo mlbench-helm to be cloned, and helm to be installed *Helm (Required)*

MLBench's Helm charts can also be deployed manually on a running Kubernetes cluster. For that, it is needed to have the credentials for the cluster in the `kubectl` config. For example, to obtain the credentials for a GCloud Kubernetes cluster, one should run

```
$ gcloud container clusters get-credentials --zone ${MACHINE_ZONE} ${CLUSTER_NAME}
```

This will setup `kubectl` for the cluster.

Then to deploy the dashboard on the running cluster, we need to apply our values to the existing helm template, and deploy it onto the cluster

```
$ cd mlbench-helm
$ helm template ${RELEASE_NAME} . \
    --set limits.workers=${NUM_NODES-1} \
    --set limits.gpu=${NUM_GPUS} \
    --set limits.cpu=${NUM_CPUS-1} | kubectl apply -f -
```

Where :

- RELEASE_NAME represents the cluster name (called `my-cluster-3` in the example above)
- NUM_NODES is the maximum number of worker nodes available. This sets the maximum number of nodes that can be chosen for an experiment in the UI/CLI.
- NUM_GPUS is the number of gpus requested by each worker pod.
- NUM_CPUS is the maximum number of CPUs (Cores) available on each worker node. Uses Kubernetes notation (*8* or *8000m* for 8 cpus/cores). This is also the maximum number of Cores that can be selected for an experiment in the UI

This will deploy the helm charts with the corresponding images to each node, and will set the hardware limits.

Note:

Get the application URL by running these commands:

```
$ export NODE_PORT=$(kubectl get --namespace default -o jsonpath="{.spec.ports[0].
↪nodePort}" services ${RELEASE_NAME}-mlbench-master)
$ export NODE_IP=$(gcloud compute instances list|grep $(kubectl get nodes --
↪namespace default -o jsonpath="{.items[0].status.addresses[0].address}") |awk '
↪{print $5}')
$ gcloud compute firewall-rules create --quiet mlbench --allow tcp:$NODE_PORT,tcp:
↪$NODE_PORT
$ echo http://$NODE_IP:$NODE_PORT
```

Danger: The last command opens up a firewall rule to the google cloud. Make sure to delete the rule once it's not needed anymore:

```
$ gcloud compute firewall-rules delete --quiet mlbench
```

One can also create a new `myvalues.yml` file with custom limits:

```
limits:
  workers:
  cpu:
  gpu:

gcePersistentDisk:
  enabled:
  pdName:
```

- `limits.workers` is the maximum number of worker nodes available to mlbench. This sets the maximum number of nodes that can be chosen for an experiment in the UI. By default mlbench starts 2 workers on startup.
- `limits.cpu` is the maximum number of CPUs (Cores) available on each worker node. Uses Kubernetes notation (*8* or *8000m* for 8 cpus/cores). This is also the maximum number of Cores that can be selected for an experiment in the UI
- `limits.gpu` is the number of gpus requested by each worker pod.
- `gcePersistentDisk.enabled` create resources related to NFS persistentVolume and persistentVolumeClaim.
- `gcePersistentDisk.pdName` is the name of persistent disk existed in GKE.

Caution: If `workers`, `cpu` or `gpu` are set higher than available in the cluster, Kubernetes will not be able to allocate nodes to mlbench and the deployment will hang indefinitely, without throwing an exception. Kubernetes will just wait until nodes that fit the requirements become available. So make sure the cluster actually has the requested requirements.

Note: To use `gpu` in the cluster, the nvidia device plugin should be installed. See *Plugins* for details

Note: Use commands like `gcloud compute disks create --size=10G --zone=europe-west1-b my-pd-name` to create persistent disk.

Note: The GCE persistent disk will be mounted to */datasets/* directory on each worker.

Caution: Google installs several pods on each node by default, limiting the available CPU. This can take up to 0.5 CPU cores per node. So make sure to provision VM's that have at least 1 more core than the amount of cores you want to use for you mlbench experiment. See here for further details on node limits.

Plugins

In `values.yaml`, one can optionally install Kubernetes plugins by turning on/off the following flags:

- `weave.enabled`: If true, install the weave network plugin.
- `nvidiaDevicePlugin.enabled`: If true, install the nvidia device plugin.

3.4 Component Overview

mlbench consists of two components, the **Master** and the **Worker** Docker containers.

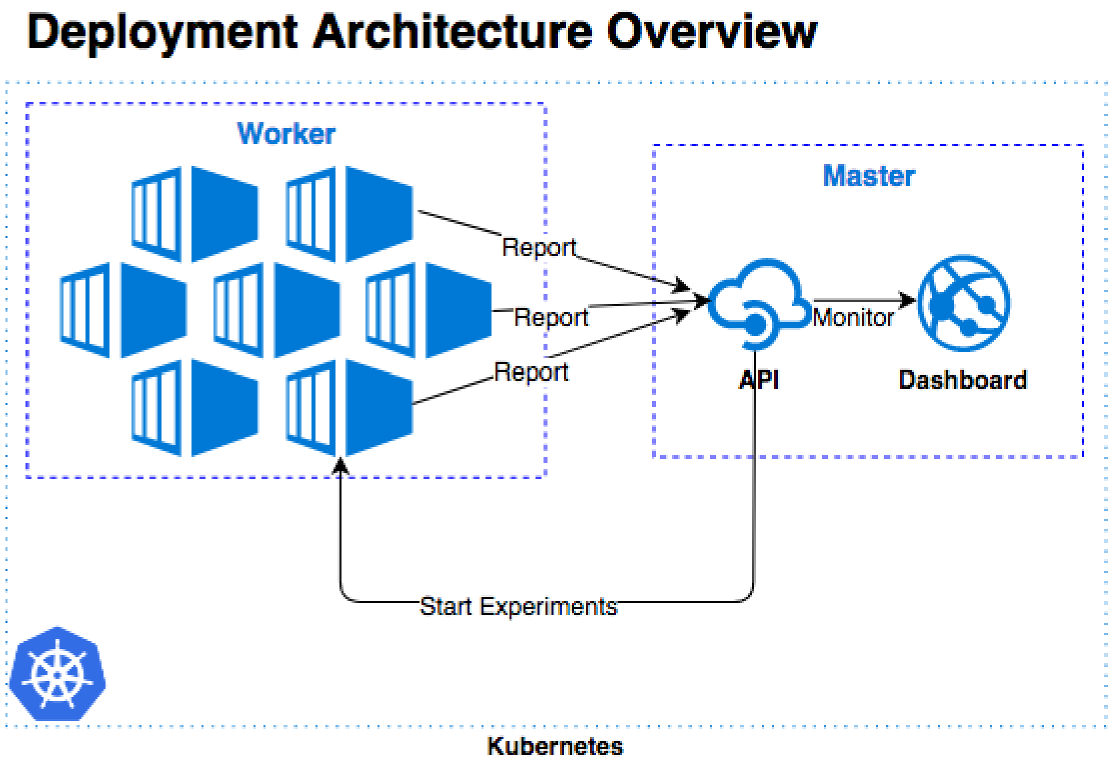


Fig. 10: Deployment Overview: Relation between Worker and Master

3.4.1 Master

The master contains the Dashboard, the main interface for the project. The dashboard allows you to start and run a distributed ML experiment and visualizes the progress and result of the experiment. It allows management of the mlbench nodes in the Kubernetes cluster and for most users constitutes the sole way they interact with the mlbench project.

It also contains a REST API that can be use instead of the Dashboard, as well as being used for receiving data from the Dashboard.

The Master also manages the `StatefulSet` of worker through the Kubernetes API.

The code for the Master can be found in the mlbench-dashboard repository.

3.4.2 Worker

The worker images contain all the boilerplate code needed for a distributed ML model as well as the actual model code. There is a docker image for each Benchmark implementation. They take care of training the distributed model, with some configuration supplied by the Master.

Worker nodes send status information to the metrics API of the Master to inform it of the progress and state of the current run.

All official reference implementations along with useful base images can be found in the mlbench-benchmarks repository.

3.4.3 mlbench-core

mlbench-core is a Python package that contains functionality to interact with the Master node, such as writing metrics. It also contains common code used across multiple benchmark implementations and implementation independent helper functions.

The code can be found in the mlbench-core repository.

3.4.4 Helm Chart

The Helm chart allows automated installation of the MLBench framework into a Kubernetes cluster.

It can be found in the mlbench-helm repository.

3.5 Tutorials

Also check out our Blog for good tips and the latest news!

3.5.1 Adding an existing PyTorch training script into MLBench

In this tutorial, we will go through the process of adapting existing distributed PyTorch code to work with the MLBench framework. This allows you to run your models in the MLBench environment and easily compare them with our reference implementations as baselines to see how well your code performs.

MLBench is designed to easily be used with third-party models, allowing for quick and fair comparisons by standardizing the data distribution, evaluation dataset and providing evaluation code. It saves all of the hassle that's needed to implement your own baselines for comparison.

We will adapt the code from the official PyTorch distributed tutorial to run in MLBench. If you're unfamiliar with that tutorial, it might be worth giving it a quick look so you know what we're working with.

Adapting the Code

To get started, create a new directory `mlbench-pytorch-tutorial` and copy the train_dist.py file into it.

The official tutorial spawns multiple parallel processes on a single machine, but we want to run the code on multiple machines, so first we need to replace the initialization functionality with our own.

Replace

```python
if __name__ == "__main__":
    size = 2
    processes = []
    for rank in range(size):
        p = Process(target=init_processes, args=(rank, size, run))
        p.start()
        processes.append(p)

    for p in processes:
        p.join()
```

with

```python
if __name__ == "__main__":
    parser = argparse.ArgumentParser(description='Process run parameters')
    parser.add_argument('--run_id', type=str, help='The id of the run')
    parser.add_argument('--rank', type=int, help='The rank of this worker')
    parser.add_argument('--hosts', type=str, help='The list of hosts')
    args = parser.parse_args()
    init_processes(args.rank, args.run_id, args.hosts)
```

and add

```python
import argparse
```

to the top of the file.

We also need to change the `init_processes` method to reflect our previous changes, along with setting the `WORLD_SIZE` and `RANK` environment variables:

```python
def init_processes(rank, run_id, hosts, backend='gloo'):
    """ Initialize the distributed environment. """
    hosts = hosts.split(',')
    os.environ['MASTER_ADDR'] = hosts[0] # first worker is the master worker
    os.environ['MASTER_PORT'] = '29500'
    world_size = len(hosts)
    os.environ['WORLD_SIZE'] = str(world_size)
    os.environ['RANK'] = str(rank)
    dist.init_process_group(backend, rank=rank, world_size=len(world_size))
    run(rank, world_size, run_id)
```

Next, we need to change the signature of the `run` method to add the `run_id` parameter. The `run_id` is a unique identifier automatically assigned by MLBench to identify an individual run and all its data and performance metrics.

```python
def run(rank, size, run_id):
```

At this point, the script could technically already run in MLBench. However, you would not be able to see any reported results or intermediate stats during training. Results are shown either in the Dashboard (where you can see them in real time) or can be downloaded at any time during the run from the command line. So let's add some reporting functionality.

The PyTorch script reports loss to `stdout`, but we can easily report the loss to MLBench as well. First we need to import the relevant MLBench functionality by adding the following line to the imports at the top of the file:

```python
from mlbench_core.utils import Tracker
from mlbench_core.evaluation.goals import task1_time_to_accuracy_goal
from mlbench_core.evaluation.pytorch.metrics import TopKAccuracy
from mlbench_core.controlflow.pytorch import validation_round
```

`task1_time_to_accuracy_goal` measures the time taken to reach 80% accuracy.

After this we can simply create a `Tracker` object and use it to report the loss and add metrics (`TopKAccuracy`) to track. We add code to record the timing of different steps with `tracker.record_batch_step()`. We have to tell the tracker that we're in the training loop by calling `tracker.train()` and that the epoch is done by calling `tracker.epoch_end()`. The loss is recorded with `tracker.record_loss()`.

```python
def run(rank, size, run_id):
    """ Distributed Synchronous SGD Example """
```

(continues on next page)

(continued from previous page)

```
    torch.manual_seed(1234)
    train_set, bsz = partition_dataset()
    model = Net()
    optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.5)
    metrics = [                                     # Add metrics to gather
        TopKAccuracy(topk=1),
        TopKAccuracy(topk=5)
    ]
    loss_func = nn.NLLLoss()

    tracker = Tracker(metrics, run_id, rank)        # Instantiate a Tracker

    num_batches = ceil(len(train_set.dataset) / float(bsz))

    tracker.start()                                 # Start the tracker

    for epoch in range(10):
        tracker.train()                             # Record training start

        epoch_loss = 0.0
        for data, target in train_set:
            tracker.batch_start()                   # Record batch start

            optimizer.zero_grad()
            output = model(data)

            tracker.record_batch_step('forward')    # Record batch forward step

            loss = loss_func(output, target)
            epoch_loss += loss.data.item()

            tracker.record_batch_step('loss')       # Record batch loss step

            loss.backward()

            tracker.record_batch_step('backward')   # Record batch backward step

            average_gradients(model)
            optimizer.step()

            tracker.batch_end()                     # Record batch end

        tracker.record_loss(epoch_loss, num_batches, log_to_api=True)

        logging.debug('Rank %s, epoch %s: %s',
                      dist.get_rank(), epoch,
                      epoch_loss / num_batches)       # Print to stderr

        tracker.epoch_end()                         # Record epoch end

        if tracker.goal_reached:                    # Goal reached
            logging.debug("Goal Reached!")
```

(continues on next page)

(continued from previous page)

```
            return
```

That's it. Now the training will report the loss of each worker back to the Dashboard and the output result files. On the Dashboard, you will also see a nice graph showing this data.

For the official tasks, we also need to report validation stats to the tracker and use the official validation code. Rename the current `partition_dataset()` method to `partition_dataset_train` and add a new partition method to load the validation set:

```python
def partition_dataset_val():
    """ Partitioning MNIST validation set"""
    dataset = datasets.MNIST(
        './data',
        train=False,
        download=True,
        transform=transforms.Compose([
            transforms.ToTensor(),
            transforms.Normalize((0.1307, ), (0.3081, ))
        ]))
    size = dist.get_world_size()
    bsz = int(128 / float(size))
    partition_sizes = [1.0 / size for _ in range(size)]
    partition = DataPartitioner(dataset, partition_sizes)
    partition = partition.use(dist.get_rank())
    val_set = torch.utils.data.DataLoader(
        partition, batch_size=bsz, shuffle=True)
    return val_set, bsz
```

Then load the validation set and add the goal for the official task (The Task 1a goal is used for illustration purposes in this example):

```python
def run(rank, size, run_id):
    """ Distributed Synchronous SGD Example """
    torch.manual_seed(1234)
    train_set, bsz = partition_dataset_train()
    val_set, bsz_val = partition_dataset_val()
    model = Net()
    optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.5)
    metrics = [
        TopKAccuracy(topk=1),
        TopKAccuracy(topk=5)
    ]
    loss_func = nn.NLLLoss()

    goal = task1_time_to_accuracy_goal

    tracker = Tracker(metrics, run_id, rank, goal=goal)

    num_batches = ceil(len(train_set.dataset) / float(bsz))
    num_batches_val = ceil(len(val_set.dataset) / float(bsz_val))

    tracker.start()
```

Now all that is needed is to add the validation loop code (`validation_round()`) to run validation in the `run()`

function. We also check if the goal is reached and stop training if it is. `validation_round()` evaluates the metrics on the validation set and reports the results to the Dashboard.

```
tracker.record_loss(epoch_loss, num_batches, log_to_api=True)

logging.debug('Rank %s, epoch %s: %s',
              dist.get_rank(), epoch,
              epoch_loss / num_batches)

validation_round(val_set, model, loss_func, metrics, run_id, rank,
                  'fp32', transform_target_type=None, use_cuda=False,
                  max_batch_per_epoch=num_batches_val, tracker=tracker)

tracker.epoch_end()

if tracker.goal_reached:
    logging.debug("Goal Reached!")
    return
```

The full code (with some additional improvements) is in our Github Repo.

Creating a Docker Image for Kubernetes

To actually run our code, we need to wrap it in a Docker Image. We could create one from scratch, but it's easier to use the PyTorch Base image provided by MLBench, which already includes everything you might need for executing a PyTorch model.

Create a new file called `Dockerfile` in the `mlbench-pytorch-tutorial` directory and add the following code:

```
1  FROM mlbench/mlbench-pytorch-base:latest
2  
3  RUN pip install mlbench-core
4  
5  # The reference implementation and user defined implementations are placed here.
6  # ADD ./requirements.txt /requirements.txt
7  # RUN pip install --no-cache-dir -r /requirements.txt
8  
9  RUN mkdir /codes
10 ADD ./train_dist.py /codes/train_dist.py
11 
12 EXPOSE 29500
13 
14 ENV PYTHONPATH /codes
```

The `mlbench-pytorch-base:latest` image already contains all necessary libraries, but if your image requires additional python libraries, you can add them with the commands on lines 6 and 7, along with adding a `requirements.txt` file.

In order for Kubernetes to access the image, you have to build and upload it to a Docker registry that's accessible to Kubernetes, for instance Docker Hub (Make sure to change the Docker image and repo name accordingly):

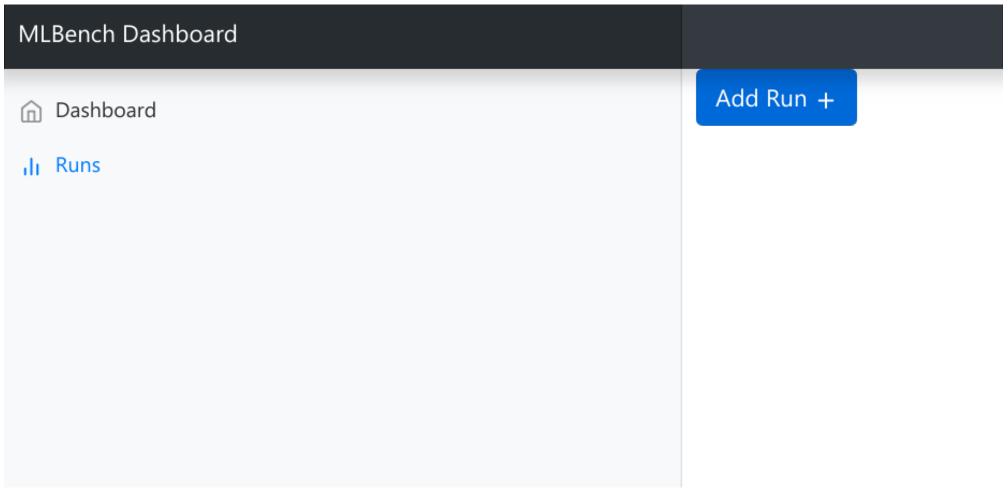
```
$ docker login
$ docker build -t <user|organisation>/<name>:latest mlbench-pytorch-tutorial/
$ docker push mlbench/pytorch-tutorial:latest
```

The image is now built and available for running in MLBench.

Running the code in MLBench

Navigate to the MLBench Dashboard and go to the `Runs` page.

Create a new Run:

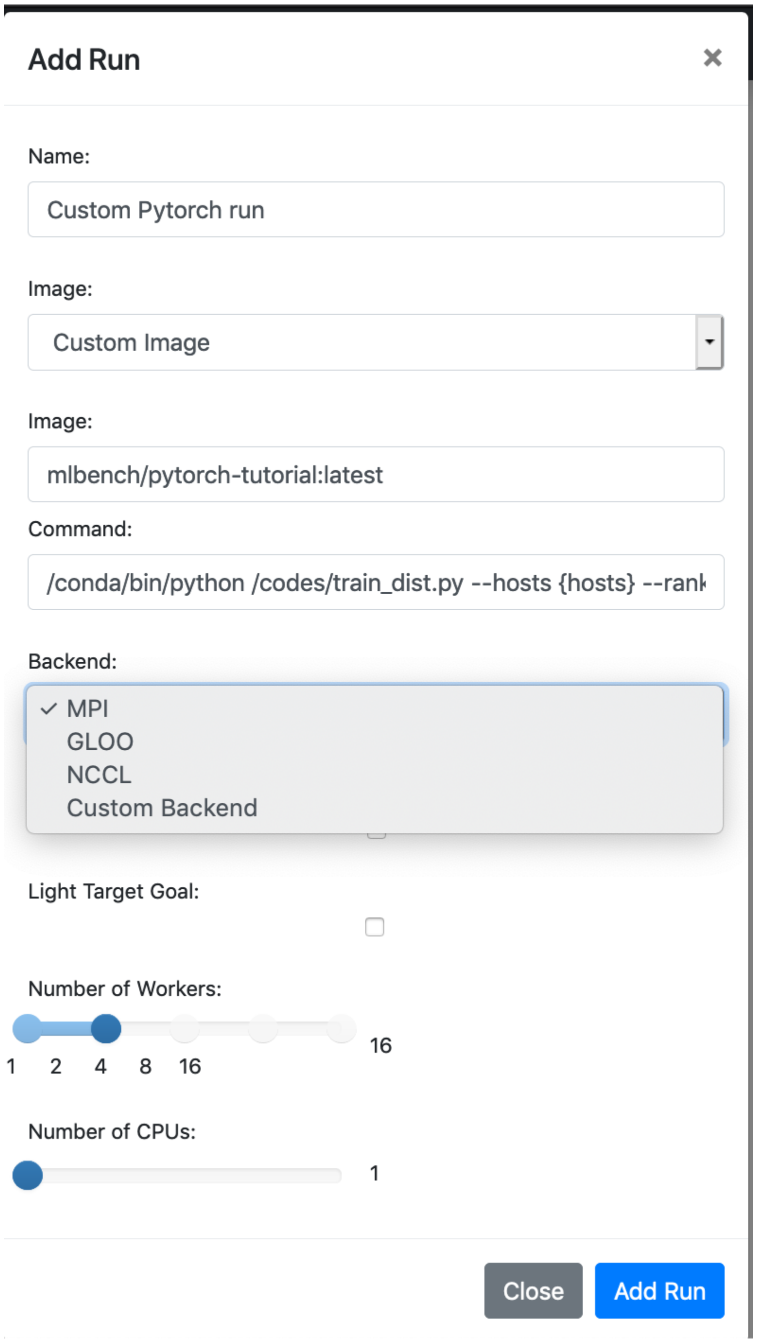


Enter the URL of the newly uploaded Docker image (The host can be left out if you use Docker Hub). Then enter the command to execute on each worker:

```
/conda/bin/python /codes/train_dist.py --hosts {hosts} --rank {rank} --run_id {run_id}
```

The values in brackets will be substituted by MLBench with the correct values and passed to our script.

We also need to choose which backend we want to run on (in our case, MPI) and set the number of workers on which we want to execute our run.


Add Run

Name: Custom Pytorch run

Image: Custom Image

Image: mlbench/pytorch-tutorial:latest

Command: /conda/bin/python /codes/train_dist.py --hosts {hosts} --rank

Backend: ✓ MPI

GLOO

NCCL

Custom Backend

Light Target Goal:

Number of Workers: 16

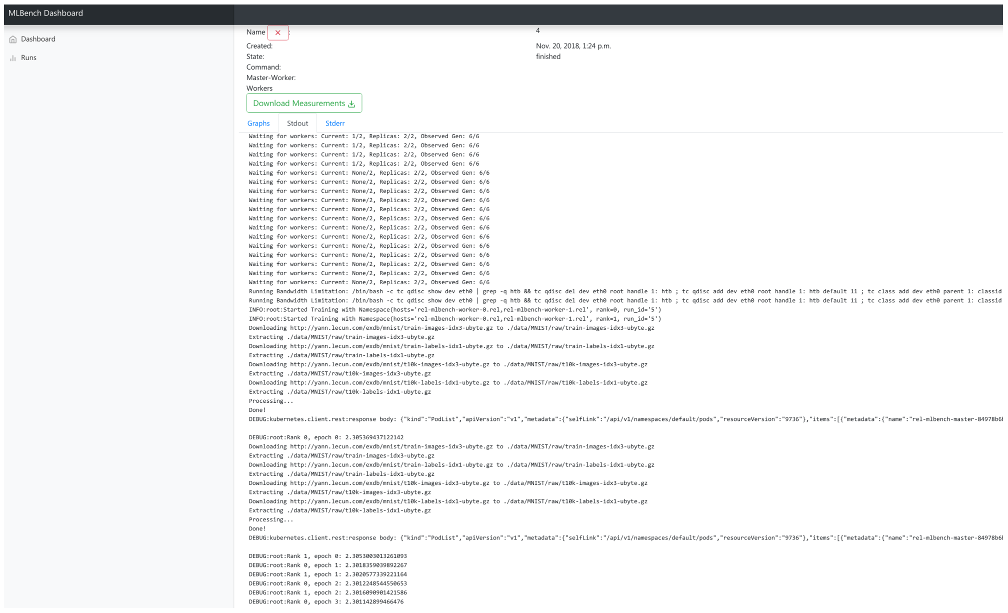
Number of CPUs: 1

Now we're all set to start our experiment. Hit `Add Run` and that's it. You just ran a custom model on MLBench. If you are only running from the command line, you can execute:

```
mlbench run custom-pytorch-run 2
```

When prompted, choose `Custom Image` and enter the image and execution command.

If you are using the Dashboard, you should see a graph of the training loss of each worker, along with the combined `stdout` and `stderr` of all workers. If you are running from the command line, you will see these printed to your terminal and will be able to access the training data and results using `mlbench download <run_name>` (check out our tutorial on *Using the MLBench Command-Line Interface* for more information).



Name: 4
Created: Nov. 20, 2018, 1:24 p.m.
State: finished
Command:
Master-Worker:
Workers

```
Waiting for workers: Current: 1/2, Replicas: 2/2, Observed Gen: 6/6
Waiting for workers: Current: 1/2, Replicas: 2/2, Observed Gen: 6/6
Waiting for workers: Current: 1/2, Replicas: 2/2, Observed Gen: 6/6
Waiting for workers: Current: 1/2, Replicas: 2/2, Observed Gen: 6/6
Waiting for workers: Current: None/2, Replicas: 2/2, Observed Gen: 6/6
Waiting for workers: Current: None/2, Replicas: 2/2, Observed Gen: 6/6
Waiting for workers: Current: None/2, Replicas: 2/2, Observed Gen: 6/6
Waiting for workers: Current: None/2, Replicas: 2/2, Observed Gen: 6/6
Waiting for workers: Current: None/2, Replicas: 2/2, Observed Gen: 6/6
Waiting for workers: Current: None/2, Replicas: 2/2, Observed Gen: 6/6
Waiting for workers: Current: None/2, Replicas: 2/2, Observed Gen: 6/6
Waiting for workers: Current: None/2, Replicas: 2/2, Observed Gen: 6/6
Waiting for workers: Current: None/2, Replicas: 2/2, Observed Gen: 6/6
Waiting for workers: Current: None/2, Replicas: 2/2, Observed Gen: 6/6
Waiting for workers: Current: None/2, Replicas: 2/2, Observed Gen: 6/6
Waiting for workers: Current: None/2, Replicas: 2/2, Observed Gen: 6/6
Waiting for workers: Current: None/2, Replicas: 2/2, Observed Gen: 6/6
Running Bandwidth Limitation: /bin/bash -c tc qdisc show dev eth0 | grep -q htb && tc qdisc del dev eth0 root handle 1: htb ; tc qdisc add dev eth0 root handle 1: htb default 11 ; tc class add dev eth0 parent 1: classid
Running Bandwidth Limitation: /bin/bash -c tc qdisc show dev eth0 | grep -q htb && tc qdisc del dev eth0 root handle 1: htb ; tc qdisc add dev eth0 root handle 1: htb default 11 ; tc class add dev eth0 parent 1: classid
INFO:root:Started Training with Namespace(hosts='rel-mlbench-worker-0.rel,rel-mlbench-worker-1.rel', rank=0, run_id='5')
INFO:root:Started Training with Namespace(hosts='rel-mlbench-worker-0.rel,rel-mlbench-worker-1.rel', rank=1, run_id='5')
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz to ./data/MNIST/raw/train-images-idx3-ubyte.gz
Extracting ./data/MNIST/raw/train-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz to ./data/MNIST/raw/train-labels-idx1-ubyte.gz
Extracting ./data/MNIST/raw/train-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz to ./data/MNIST/raw/t10k-images-idx3-ubyte.gz
Extracting ./data/MNIST/raw/t10k-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz to ./data/MNIST/raw/t10k-labels-idx1-ubyte.gz
Extracting ./data/MNIST/raw/t10k-labels-idx1-ubyte.gz
Processing...
Done!
DEBUG:kubernetes.client.rest:response body: {"kind":"PodList","apiVersion":"v1","metadata":{"selfLink":"/api/v1/namespaces/default/pods","resourceVersion":"9736"},"items":[{"metadata":{"name":"rel-mlbench-master-84978b6

DEBUG:root:Rank 0, epoch 0: 2.305369437122142
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz to ./data/MNIST/raw/train-images-idx3-ubyte.gz
Extracting ./data/MNIST/raw/train-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz to ./data/MNIST/raw/train-labels-idx1-ubyte.gz
Extracting ./data/MNIST/raw/train-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz to ./data/MNIST/raw/t10k-images-idx3-ubyte.gz
Extracting ./data/MNIST/raw/t10k-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz to ./data/MNIST/raw/t10k-labels-idx1-ubyte.gz
Extracting ./data/MNIST/raw/t10k-labels-idx1-ubyte.gz
Processing...
Done!
DEBUG:kubernetes.client.rest:response body: {"kind":"PodList","apiVersion":"v1","metadata":{"selfLink":"/api/v1/namespaces/default/pods","resourceVersion":"9736"},"items":[{"metadata":{"name":"rel-mlbench-master-84978b6

DEBUG:root:Rank 1, epoch 0: 2.3053003013261093
DEBUG:root:Rank 0, epoch 1: 2.3018359039892267
DEBUG:root:Rank 1, epoch 1: 2.3020577339221164
DEBUG:root:Rank 0, epoch 2: 2.3012248544550653
DEBUG:root:Rank 1, epoch 2: 2.3016090901421586
DEBUG:root:Rank 0, epoch 3: 2.301142899466476
```

3.5.2 Using the MLBench Command-Line Interface

In this tutorial we'll introduce the CLI and show you how easy it is to get it up and running.

Please beware any costs that might be incurred by running this tutorial on the Google cloud. Usually costs should only be on the order of 5-10USD. We don't take any responsibility for the costs incurred

Install the mlbench-core python package by running:

```
pip install mlbench-core
```

After installation, mlbench is usable by calling the `mlbench` command.

MLBench supports multiple clouds, but for the purposes of this tutorial we will focus on Google Cloud. To create a new Google cloud cluster, simply run (this might take a couple of minutes):

```
$ mlbench create-cluster gcloud 3 my-cluster
[...]
MLBench successfully deployed
```

This creates a cluster with 3 nodes called `my-cluster-3` and sets up the mlbench deployment in that cluster. Note that the number of nodes should always be 1 higher than the maximum number of workers you want to run.

To start an experiment, simply run:

```
$ mlbench run my-run 2

Benchmark:

[0]     PyTorch Cifar-10 ResNet-20
[1]     PyTorch Cifar-10 ResNet-20 (Scaling LR)
[2]     PyTorch Linear Logistic Regression
[3]     PyTorch Machine Translation GNMT
[4]     PyTorch Machine Translation Transformer
[5]     Tensorflow Cifar-10 ResNet-20 Open-MPI
```

(continues on next page)

(continued from previous page)

```
[6]     PyTorch Distributed Backend benchmarking
[7]     Custom Image

Selection [0]: 1
Backend:

[0]     MPI
[1]     GLOO
[2]     NCCL
[3]     Custom Backend

Selection [0]: 0

[...]

Run started with name my-run-2
```

You will be prompted to select the benchmark image you want to run (or to specify a custom image). Afterwards, a new benchmark run will be started in the cluster with 2 workers.

You can also start multiple runs at the same time, which will be scheduled as nodes become available:

```
$ mlbench run my-run 2 4 8 16

Benchmark:

[0]     PyTorch Cifar-10 ResNet-20
[1]     PyTorch Cifar-10 ResNet-20 (Scaling LR)
[2]     PyTorch Linear Logistic Regression
[3]     PyTorch Machine Translation GNMT
[4]     PyTorch Machine Translation Transformer
[5]     Tensorflow Cifar-10 ResNet-20 Open-MPI
[6]     PyTorch Distributed Backend benchmarking
[7]     Custom Image


Selection [0]: 1
Backend:

[0]     MPI
[1]     GLOO
[2]     NCCL
[3]     Custom Backend

Selection [0]: 0

[...]

Run started with name my-run-2
Run started with name my-run-4
Run started with name my-run-8
Run started with name my-run-16
```

which would start runs with 2, 4, 8 and 16 workers, respectively.

To see the status of a run, execute:

```
$ mlbench status my-run-2
[...]
id      name    created_at            finished_at state
---     ------  -----------            ----------- -----
1       my-run-2 2019-11-11T13:35:06             started
No Validation Loss Data yet
No Validation Precision Data yet
```

After the first round of validation, this command also outputs the current validation loss and precision.

To download the results of a current or finished run, use:

```
$ mlbench download my-run-2
```

which will download all the metrics of the run as a zip file. This file also contains the official benchmark result once the run finishes, in the form of the `official_result.txt`.

You can also access all the information of the run in the dashboard. To get the dashboard URL, simply run:

```
$ mlbench get-dashboard-url
[...]
http://34.76.223.123:32535
```

Don't forget to delete the cluster once you're done!

```
$ mlbench delete-cluster gcloud my-cluster-3
[...]
```

NOTE: if you created a cluster in a non-default zone using the `-z` flag, you also need to delete it by passing the same flag and argument to `mlbench delete-cluster`.

```
# create cluster in europe-west2-b (non-default)
$ mlbench create-cluster gcloud -z europe-west2-b 3 my-cluster

# delete cluster
$ mlbench delete-cluster gcloud -z europe-west2-b my-cluster-3
```

3.5.3 Using Kubernetes-in-Docker (KIND) for development and debugging

Developing distributed applications can be a burden because it requires a cluster of machines to be available. This induces additional costs that are really not necessary. Luckily, KIND can be very helpful.

KIND allows for deployment of a kubernetes cluster locally on your machine, using docker, and unlocks testing and development without an available "real" cluster of machines.

To deploy a KIND cluster locally, use the following command:

```
$ mlbench create-cluster kind 3 my-cluster
[...]
MLBench successfully deployed
```

This will create a "kind" cluster of size 3 nodes, called `my-cluster`. The dashboard will be available on one running image, and the two workers allow you to run some code.

Additionally, this command will also deploy a local docker registry at `localhost:5000`, which allows the use of local images instead of having to pull them for a remote location, and connects the created cluster to it (through docker networks).

```
$ docker ps
CONTAINER ID        IMAGE                     COMMAND                  CREATED                ␣
↪STATUS              PORTS                     NAMES
54bc6050b3a1        kindest/node:v1.15.12    "/usr/local/bin/entr..."   5 minutes ago         ␣
↪   Up 5 minutes                              my-cluster-3-worker
3b5579d64a78        kindest/node:v1.15.12    "/usr/local/bin/entr..."   5 minutes ago         ␣
↪   Up 5 minutes        127.0.0.1:40583->6443/tcp   my-cluster-3-control-plane
d4612a2c913c        kindest/node:v1.15.12    "/usr/local/bin/entr..."   5 minutes ago         ␣
↪   Up 5 minutes                              my-cluster-3-worker2
3624c7f747e3        registry:2               "/entrypoint.sh /etc..."   4 days ago            ␣
↪   Up 17 hours         0.0.0.0:5000->5000/tcp      kind-registry
```

To push an image to the local registry (and have it available for the cluster), use the following commands:

```
$ docker tag <repo>/<image>:<tag> localhost:5000/<image>:<tag>
$ docker push localhost:5000/<image>:<tag>
```

At this point, the image `<image>:<tag>` will be available for use locally.

```
$ mlbench run test 2

[0]     PyTorch Cifar-10 ResNet-20
[1]     PyTorch Cifar-10 ResNet-20 (DDP)
[2]     PyTorch Linear Logistic Regression
[3]     PyTorch Language Modeling (AWD-LSTM)
[4]     PyTorch Machine Translation GNMT
[5]     PyTorch Machine Translation Transformer
[6]     Tensorflow Cifar-10 ResNet-20 Open-MPI
[7]     PyTorch Distributed Backend benchmarking
[8]     Custom Image

Selection [3]: 8
Image:  localhost:5000/<image>:<tag>
Command: <command-to-run>
```

The command to run the image should be a python script with arguments `run_id, rank, hosts and backend`. For official images, we use the command:

`/conda/bin/python /codes/main.py --run_id {run_id} --rank {rank} --hosts {hosts} --backend {backend}`

3.6 Developer Guide

3.6.1 Development Workflow

Git Flow is used for features etc. This automatically handles pull requests. Make sure to install the commandline tool at the link above

3.6.2 Code Style

Python code should follow PEP8 guidelines. flake8 checks PEP8 compliance

3.7 Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

3.7.1 Types of Contributions

Report Bugs

Report bugs at https://github.com/mlbench/mlbench/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with “bug” and “help wanted” is open to whoever wants to implement it.

Implement Features

Look through the GitHub issues for features. Anything tagged with “enhancement” and “help wanted” is open to whoever wants to implement it.

Write Documentation

mlbench could always use more documentation, whether as part of the official mlbench docs, in docstrings, or even on the web in blog posts, articles, and such.

Submit Feedback

The best way to send feedback is to file an issue at https://github.com/mlbench/mlbench/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

3.7.2 Get Started!

Ready to contribute? Here's how to set up *mlbench* for local development.

1. Install the Prerequisites;
2. Follow the installation guide;
3. Create a branch for local development:

   ```
   $ git checkout -b name-of-your-bugfix-or-feature
   ```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

   ```
   $ flake8 mlbench tests
   $ python setup.py test or py.test
   $ tox
   ```

   To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

   ```
   $ git add .
   $ git commit -m "Your detailed description of your changes."
   $ git push origin name-of-your-bugfix-or-feature
   ```

7. Submit a pull request through the GitHub website.

3.7.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.7, 3.4, 3.5 and 3.6, and for PyPy. Check https://travis-ci.org/mlbench/mlbench/pull_requests and make sure that the tests pass for all supported Python versions.

3.7.4 Tips

To run a subset of tests:

```
$ py.test tests.test_mlbench
```

3.7.5 Deploying

A reminder for the maintainers on how to deploy. Make sure all your changes are committed (including an entry in HISTORY.rst). Then run:

```
$ bumpversion patch # possible: major / minor / patch
$ git push
$ git push --tags
```

3.7.6 Repositories

mlbench-core

To contribute to mlbench-core, you first need to clone the mlbench-core repository, create a new branch for the feature, make the changes and then create a new local commit. For local development and testing, you do not need to push the changes to Github. You need to create a development release on PyPi with the changes. To do this, you need an account that has permission to do releases on the mlbench-core PyPi project. Then, inside the git repository you need to run:

```
$ bumpversion --verbose --allow-dirty --no-tag --no-commit dev
```

This will bump the version of the development release. You need to be aware that if someone else published a development release on PyPI since your last release, bumpversion will not take this into account. In this case, you need to manually bump the version. To do this, you first need to check what is the latest dev release on PyPI. Let us assume that the latest version on PyPI is `2.4.0.dev240`. Now, you need to enter the version `2.4.0-dev241` in the files `setup.py`, `setup.cfg` and `mlbench_core/__init__.py`. You should also be careful that the formatting of the version in the files is different than on PyPI. However, the files will already contain some version, so you only need to change the numbering and not the formatting. After you have done this, you need to build and upload the release by running the following commands inside the git repository:

```
$ python setup.py sdist bdist_wheel
$ python -m twine upload dist/*
```

If everything is successful, you should be able to see your release on PyPI. Now, to test this release you need to go to the benchmark directory and locate the file requirements.txt. Inside, there should be a line for mlbench-core specifying the version. You should replace the version with the one you just released. In the previous example you would need to specify `mlbench-core==2.4.0-dev241`. Depending on your changes, you may want to modify the code for the benchmark in the file `main.py`. This is necessary for example when you add a new optimizer and you want to test the benchmark using it. In this case, you need to replace the previous optimizer with the new one in `main.py`. After you are done with the changes, you need to build and push the docker image to Docker Hub. This can be done by running the `docker build` and `docker push` commands inside the benchmark repository. In order to be able to push to Docker Hub, you need to create an account and login using the command `docker login`. Once you have the image on Docker Hub, you can use it as a custom image in MLBench when starting a run either through the CLI or the dashboard. When prompted for the image location, you only need to specify the repository and image name because MLBench automatically looks for the image on Docker Hub. You have to note that this procedure is only required when making changes that need to be tested by running a benchmark. For example, if you want to simply make changes in the CLI, you can modify the file `cli.py` and test it locally using:

```
$ python cli.py <specific-command>
```

mlbench-benchmarks

MLBench offers a choice between different optimizers, learning rate schedulers etc, so you might be interested in modifying the existing benchmark implementations to use different components. To do this, you can follow a similar approach as described in the previous section. You have to clone the `mlbench-benchmarks` repository and modify the `main.py` file of the corresponding benchmark. You could also write your own implementations of some components and combine them with MLBench. You can find detailed description on the process of adapting existing PyTorch models to use with MLBench in the Tutorials section. The main focus here is the case where you want to try out different options which are not part of the official implementation, but are still available in mlbench-core. In addition to `main.py`, you might need to modify the files `requirements.txt` if you use additional libraries, and the `Dockerfile` if you want to include additional files or require any additional setup. However, for most use cases, only the `main.py` file needs to be modified. Once you are done with the modification, you can follow the same procedure as in the previous section, to build and push the new image, and then use it as a custom image in MLBench.

mlbench-helm

The MLBench installation through the CLI automatically uses the latest version of the master branch in the `mlbench-helm` repository. To test your own version of the helm chart without changing the master branch you first need to push your changes to a different branch in `mlbench-helm`. Then, you need to change the file `mlbench_core/cli/cli.py` inside the `mlbench-core` repository. This file contains the CLI functionalities and you can test MLBench by running it locally. To use your own version of the helm chart, you need to locate the code for creating the `ChartBuilder` object in the function you want to use. MLBench has different functions for different cloud providers. For testing, you can pick one provider, find the function for creating a cluster on that provider and modify the `ChartBuilder` object to use your own branch. For example, let's say that you have pushed your changes to the branch `new-feature` in the helm repository. Then, you should change the value of the `source.reference` field of the `ChartBuilder` object to `new-feature`. Now, when you run the command for creating the cluster, it will install MLBench using your own helm chart instead of the default one.

mlbench-dashboard

When you want to test changes to the dashboard you first need to build, tag the image and then push it to a repository on Docker Hub. Let's say you want to push the image to the repository `user/mlbench_master` with the tag `testing`. You can do that by running the following command inside the root of the `mlbench-dashboard` repository:

```
$ docker login
$ docker build -f Docker/Dockerfile -t user/mlbench_master:testing .
$ docker push user/mlbench_master:testing
```

Once you push the image, you can modify the file `values.yaml` in `mlbench-helm` to use your new image. You need to modify the values of `master.image.repository` and `master.image.tag`. In our example, you would set the repository to `user/mlbench_master` and the tag to `testing`. From there, you can use the instructions from the previous section to use the new chart with the CLI. Alternatively, you could skip the step of creating a branch on the helm repository and use the `custom-value` argument of the functions for creating clusters using the CLI. As an example, to customize the helm chart directly from the CLI, when creating a cluster on Google Cloud you could use the following command:

```
$ mlbench create-cluster gcloud 3 my-cluster --custom-value master.image.repository=user/
↪mlbench_master --custom-value master.image.tag=testing
```

mlbench-docs

To contribute to the documentation, you simply need to modify the relevant .rst file inside the repository and create a pull request. Once the pull request is accepted and merged, the changes will automatically be published on the website with the next release.

3.8 Change Log

3.8.1 MLBench Core

v3.0.0

v3.0.0 (2020-12-07)

Full Changelog

Implemented enhancements:

- Support multiple clusters in CLI #91
- Add notebook/code to visualize results #72
- Support AWS in CLI #33
- Fix rnn language model #303 (ehoelzl)
- Transformer language translation #99 (ehoelzl)

Fixed bugs:

- Training code keeps running for PyTorch after training is done #26

Closed issues:

- Remove loss argument for metric computation #295

- Update PyTorch to 1.7 #286
- Refactor optimizer and chose more appropriate names #284
- fails to create kind cluster #277
- Refactor CLI #253
- Dependabot couldn’t authenticate with https://pypi.python.org/simple/ #252
- Unify requirements/setup.py versions #244
- isort failing on all PRs #227
- torch.div is not supported in PyTorch 1.6 #223
- Refactor common functionality for tiller and helm #108
- Add GPU support for AWS in CLI #104
- Change CPU limit to #CPUs - 1 #101
- Add –version flag #97
- Cluster creation/deletion errors with non-default zone #94
- Add command to list runs #86
- RefreshError from gcloud #83
- Run new benchmarks and document costs #82
- Make nvidia k80 default GPU #80
- Fix random seeds #79
- benchmark against torch.nn.parallel.DistributedDataParallel MPSG #75
- upgrade to pytorch 1.5 #74
- Provide comparison to competitors #66
- Add some integration tests #64
- Remove stale branches #62
- Add PowerSGD optimizer #59
- Add RNN Language Model #54
- Use torch.nn.DataParallel for intra-node computation #46
- Add CLI support for DIND #42
- Port over functionality from Language Model benchmark to the core library #34
- make results reproducible from command-line #24
- Contribution and docs section on README.md #17
- test new torch.distributed #15

Merged pull requests:

- Bugfix KIND cli #307 (ehoelzl)
- Update README.md to show new badge #306 (ehoelzl)
- Create manual.yml #305 (ehoelzl)
- Switch to github actions #304 (ehoelzl)

- Bump sphinx from 3.3.0 to 3.3.1 #301 (dependabot[bot])
- Remove loss from metric argument #297 (ehoelzl)
- Fix translators #294 (ehoelzl)
- Update pytorch #292 (ehoelzl)
- Bump sphinx from 3.2.1 to 3.3.0 in /docs #288 (dependabot[bot])
- Refactor optimizers #285 (ehoelzl)
- Bump isort from 5.5.4 to 5.6.4 #283 (dependabot[bot])
- Bump sphinx-autoapi from 1.5.0 to 1.5.1 #280 (dependabot[bot])
- Add gpu functionality on AWS #278 (mmilenkoski)
- Catch exceptions when creating/deleting clusters #276 (ehoelzl)
- Fix doc #275 (ehoelzl)
- Fix AWS deployment #274 (mmilenkoski)
- Create dependabot.yml #260 (ehoelzl)
- Merge requirements & Update doc #259 (ehoelzl)
- Bump google-api-python-client from 1.9.3 to 1.12.1 #246 (dependabot-preview[bot])
- Bump numpy from 1.19.0 to 1.19.2 #245 (dependabot-preview[bot])
- Bump boto3 from 1.14.6 to 1.14.50 #234 (dependabot-preview[bot])
- Fix isort errors #233 (mmilenkoski)
- Bump pytest-mock from 3.1.1 to 3.3.1 #231 (dependabot-preview[bot])
- Bump isort from 4.3.21 to 5.4.2 #221 (dependabot-preview[bot])
- Bump sphinx from 3.0.4 to 3.2.1 #220 (dependabot-preview[bot])
- Bump grpcio from 1.29.0 to 1.31.0 #207 (dependabot-preview[bot])
- Bump spacy from 2.3.0 to 2.3.2 #182 (dependabot-preview[bot])
- Downgrade Sphinx #162 (ehoelzl)
- Add developer docs #161 (Panaetius)
- Fp optimizer changes #160 (ehoelzl)
- Bump wcwidth from 0.1.9 to 0.2.5 #156 (dependabot-preview[bot])
- Bump all versions and add doc test #152 (Panaetius)
- Bump torchvision from 0.6.0 to 0.6.1 #151 (dependabot-preview[bot])
- Bump numpy from 1.18.5 to 1.19.0 #150 (dependabot-preview[bot])
- Bump torch from 1.5.0 to 1.5.1 #148 (dependabot-preview[bot])
- Bump google-auth from 1.17.2 to 1.18.0 #147 (dependabot-preview[bot])
- Bump sphinx-rtd-theme from 0.4.3 to 0.5.0 #144 (dependabot-preview[bot])
- Bump spacy from 2.2.4 to 2.3.0 #142 (dependabot-preview[bot])
- Bump sphinx from 3.1.0 to 3.1.1 #140 (dependabot-preview[bot])
- Bump dill from 0.3.1.1 to 0.3.2 #138 (dependabot-preview[bot])

- Update dependencies #137 (Panaetius)
- Bump spacy from 2.2.3 to 2.2.4 #135 (dependabot-preview[bot])
- Bump numpy from 1.16.6 to 1.18.5 #133 (dependabot-preview[bot])
- Bump freezegun from 0.3.12 to 0.3.15 #129 (dependabot-preview[bot])
- Bump tabulate from 0.8.6 to 0.8.7 #128 (dependabot-preview[bot])
- Bump deprecation from 2.0.6 to 2.1.0 #125 (dependabot-preview[bot])
- Bump pytest-black from 0.3.8 to 0.3.9 #124 (dependabot-preview[bot])
- Bump sphinx-rtd-theme from 0.4.2 to 0.4.3 #123 (dependabot-preview[bot])
- Bump sphinx from 1.8.1 to 3.1.0 #121 (dependabot-preview[bot])
- Bump pytest-mock from 1.10.0 to 3.1.1 #120 (dependabot-preview[bot])
- Bump torchtext from 0.5.0 to 0.6.0 #118 (dependabot-preview[bot])
- Bump torchvision from 0.5.0 to 0.6.0 #117 (dependabot-preview[bot])
- Adds support for multiple clusters #115 (Panaetius)
- Bump click from 7.0 to 7.1.2 #114 (dependabot-preview[bot])
- Bump google-cloud-container from 0.3.0 to 0.5.0 #113 (dependabot-preview[bot])
- Bump appdirs from 1.4.3 to 1.4.4 #112 (dependabot-preview[bot])
- Bump sphinxcontrib-bibtex from 0.4.0 to 1.0.0 #111 (dependabot-preview[bot])
- Bump sphinx-autoapi from 1.3.0 to 1.4.0 #110 (dependabot-preview[bot])
- Remove unused arguments in create_aws #109 (mmilenkoski)
- Fix Random seeds, Add new tracker stats #107 (ehoelzl)
- Add return_code check in test_cli #106 (mmilenkoski)
- Add AWS support in CLI #103 (mmilenkoski)
- Update test_cli.py #100 (giorgiosav)
- Adds a chart command to cli #95 (Panaetius)
- Add support for kind cluster creation in the CLI #93 (mmilenkoski)

v2.4.0

v2.4.0 (2020-04-20)

Full Changelog

Implemented enhancements:

- Switch to black for code formatting #35

Closed issues:

- Travis tests run only for Python 3.6 #65
- Downloading results fails if `--output` option is not provided #57
- Remember user input in mlbench run #56

- Aggregate the gradients by model, instead of by layers. #45
- Update docker images to CUDA10, mlbench-core module to newest #43
- Upgrade PyTorch to 1.4 #40

Merged pull requests:

- Pytorch v1.4.0 #68 (ehoelzl)
- Fix ci #67 (ehoelzl)
- Add aggregation by model #61 (ehoelzl)
- Remember user input in mlbench run #60 (mmilenkoski)
- Add default name of output file in CLI #58 (mmilenkoski)
- Cli adaptation #55 (ehoelzl)
- Update tags and patch version to 2.3.2 #52 (ehoelzl)
- Add get_optimizer to create optimizer object #48 (mmilenkoski)

v2.3.2

v2.3.2 (2020-04-07)

Full Changelog

Implemented enhancements:

- Add NCCL & GLOO Backend support #49
- Add NCCL & GLOO Backend support #47 (giorgiosav)

Fixed bugs:

- math ValueError with 1-node cluster #38

Merged pull requests:

- num_workers fix #51 (giorgiosav)
- Adds centralized Adam implementation #41 (mmilenkoski)

v2.3.1

2.3.1 (2020-03-09)

Full Changelog

Implemented enhancements:

- Customize Communication Scheme For Sparsified/Quantizatized/Decentralized scenarios #12

v2.3.0

v2.3.0 (2019-12-23)

Full Changelog

v2.2.1

v2.2.1 (2019-12-16)

Full Changelog

v2.2.0

v2.2.0 (2019-11-11)

Full Changelog

Implemented enhancements: - `initialize_backends` can now be called as context manager - Improved CLI to run multiple runs in parallel

v2.1.1

v2.1.1 (2019-11-11)

Full Changelog

v2.1.0

v2.1.0 (2019-11-4)

Full Changelog

Implemented enhancements:

- Added CLI for MLBench runs

v2.0.0

v2.0.0 (2019-06-13)

Full Changelog

v1.4.4

v1.4.4 (2019-05-28)

Full Changelog

v1.4.3

v1.4.3 (2019-05-23)

Full Changelog

v1.4.2

v1.4.2 (2019-05-21)

Full Changelog

v1.4.1

v1.4.1 (2019-05-16)

Full Changelog

v1.4.0

v1.4.0 (2019-05-02)

Full Changelog

Implemented enhancements:

- Split Train and Validation in Tensorflow #22

v1.3.4

v1.3.4 (2019-03-20)

Full Changelog

Implemented enhancements:

- in controlflow, don't mix train and validation #20

Fixed bugs:

- Add metrics logging for Tensorflow #19

v1.3.3

v1.3.3 (2019-02-26)

Full Changelog

v1.3.2

v1.3.2 (2019-02-13)

Full Changelog

v1.3.1

v1.3.1 (2019-02-13)

Full Changelog

v1.3.0

v1.3.0 (2019-02-12)

Full Changelog

v1.2.1

v1.2.1 (2019-01-31)

Full Changelog

v1.2.0

v1.2.0 (2019-01-30)

Full Changelog

v1.1.1

v1.1.1 (2019-01-09)

Full Changelog

v1.1.0

v1.1.0 (2018-12-06)

Full Changelog

Fixed bugs:

- Bug when saving checkpoints #13

Implemented enhancements:

- Adds Tensorflow Controlflow, Dataset and Model code
- Adds Pytorch linear models
- Adds sparsified and decentralized optimizers

v1.0.0

1.0.0 (2018-11-15)

Implemented enhancements:

- Add API Client to mlbench-core #6
- Move to google-style docs #4
- Add Imagenet Dataset for pytorch #3
- Move worker code to mlbench-core repo #1

3.8.2 MLBench Helm

v3.0.0

v3.0.0 (2020-12-07)

Full Changelog

Implemented enhancements:

- Add DIND Setup Script #4
- Add Amazon Cloud setup script #3

Closed issues:

- Add integration tests for newer versions of Kubernetes #23
- Add deployment on KIND rather than Minikube #21
- Use of GCloud script #19
- Can not configure NVIDIA on AWS #17
- Migrate to Kubernetes API v1 #15
- Deployment on minikube requires kubernetes 1.15 #13
- Remove obsolete info in `values.yaml` #12

- mlbench worker pods not created #11

Merged pull requests:

- Add workflow #25 (ehoelzl)
- Update to v1 #24 (ehoelzl)
- Update doc requirements #22 (ehoelzl)
- Remove AWS and GCloud scripts #20 (ehoelzl)
- Removes unused entries from values.yaml #18 (Panaetius)
- Switch to eksctl for aws deployment #16 (mmilenkoski)
- Add setup script for kind with local registry #14 (mmilenkoski)

v2.0.0

Implemented enhancements:

- Added GKE and AWS Setup Scripts

3.8.3 MLBench Dashboard

v3.0.0

v3.0.0 (2020-12-07)

Full Changelog

Implemented enhancements:

- Allow running of custom code #9
- Define Job resource for mpirun execution #2
- Create Kubernetes Job to execute mpirun #1

Closed issues:

- Add integration tests #86
- Dependabot couldn't authenticate with https://pypi.python.org/simple/ #74
- Fix dashboard scheduling #49
- Add ability to stop run before end #48
- Make sure all results are well zipped #44
- Prevent user from inserting invalid run names #28
- Travis tests run only for Python 3.6 #24
- Remove stale branches #23

Merged pull requests:

- Switch to actions #121 (ehoelzl)
- Bump sphinx from 3.3.0 to 3.3.1 in /docs #120 (dependabot[bot])
- Fix stream disconnection #115 (ehoelzl)

- Update images #114 (ehoelzl)
- Fix integration tests #113 (ehoelzl)
- Bump rq-scheduler from 0.8.3 to 0.10.0 #109 (dependabot[bot])
- Bump sphinx from 3.2.1 to 3.3.0 in /docs #108 (dependabot[bot])
- Bump fakeredis from 1.4.3 to 1.4.4 #102 (dependabot-preview[bot])
- Bump pytest from 6.0.2 to 6.1.2 #101 (dependabot-preview[bot])
- Bump pytest-django from 3.10.0 to 4.1.0 #100 (dependabot-preview[bot])
- Bump tox from 3.20.0 to 3.20.1 #96 (dependabot-preview[bot])
- Change 'Benchmarks' to 'Benchmark Implementations' #93 (ehoelzl)
- Add integration tests #91 (ehoelzl)
- Bump pytest-kind from 20.5.3 to 20.10.0 #89 (dependabot-preview[bot])
- Add tests #75 (ehoelzl)
- Bugfix #60 (ehoelzl)
- Bump watchdog from 0.8.3 to 0.10.3 #58 (dependabot-preview[bot])
- Bump uwsgi from 2.0.17 to 2.0.19.1 #57 (dependabot-preview[bot])
- Bump sphinx from 1.7.1 to 3.1.1 #52 (dependabot-preview[bot])
- Bump tox from 2.9.1 to 3.15.2 #46 (dependabot-preview[bot])
- Bump sphinx-rtd-theme from 0.4.0 to 0.4.3 #45 (dependabot-preview[bot])
- Bump django-constance from 2.2.0 to 2.6.0 #43 (dependabot-preview[bot])
- Bump pytest-black from 0.3.8 to 0.3.9 #42 (dependabot-preview[bot])
- Bump flake8 from 3.5.0 to 3.8.3 #40 (dependabot-preview[bot])
- Bump redis from 2.10.6 to 3.5.3 #38 (dependabot-preview[bot])
- Bump pip from 10.0.1 to 20.1.1 #37 (dependabot-preview[bot])
- Bump bumpversion from 0.5.3 to 0.6.0 #34 (dependabot-preview[bot])
- Bump django from 2.2.12 to 2.2.13 #33 (dependabot[bot])
- Bump django from 2.2.12 to 2.2.13 in /Docker #32 (dependabot[bot])
- Add backend benchmark #31 (ehoelzl)
- Add transformer image #30 (ehoelzl)

v2.0.0

Implemented enhancements:

- Added Download of Task Goals
- Fixed some performance issues

v1.1.0

Implemented enhancements:

- Added new Tensorflow Benchmark Image
- Remove Bandwidth limiting
- Added ability to run custom images in dashboard

3.8.4 MLBench Benchmarks

v3.0.0

v3.0.0 (2020-12-07)

Full Changelog

Implemented enhancements:

- Update PyTorch base to 1.7 #64
- Add NLP/machine translation Transformer benchmark task #33
- Repair Logistic regression Model #30
- Add NLP/machine translation RNN benchmark task #27
- Add NLP benchmark images & task #24
- Add Gloo support to PyTorch images #23
- Add NCCL support to PyTorch images #22
- documentation: clearly link ref code to benchmark tasks #14
- Add time-to-accuracy speedup plot #7
- Update GKE documentation to use kubernetes version 1.10.9 #4
- Add tensorflow cifar10 benchmark #3
- Transformer language translation #51 (ehoelzl)

Fixed bugs:

- Change Tensorflow Benchmark to use OpenMPI #8

Closed issues:

- Clean-up tasks #63
- Support for local run #59
- task implementations: delete choco, name tasks nlp/language-model and nlp/translation #55
- remove open/closed division distinction #47
- [Not an Issue] Comparing 3 backends on multi-node single-gpu env #44
- Create light version of the base image for development #43
- No unit tests #40
- Remove stale branches #39

- Remove Communication backend from image name #36
- pytorch 1.4 #34
- create light version (in addition to full) for resource heavy benchmark tasks #19
- add script to compute official results from raw results (time to acc for example) #18

Merged pull requests:

- Add workflow #68 (ehoelzl)
- Fix rnn language model #67 (ehoelzl)
- Update pytorch #65 (ehoelzl)
- Adapt optimizer imports #62 (ehoelzl)
- Translation changes #61 (ehoelzl)
- Change 'Benchmarks' to 'Benchmark Implementations' #60 (ehoelzl)
- Add generic worker #58 (ehoelzl)
- Rename tasks #57 (ehoelzl)
- Add link to task description #56 (ehoelzl)
- Fix tasks #54 (ehoelzl)
- Add backend benchmark code and image #53 (ehoelzl)
- Update nccl #52 (ehoelzl)
- Remove open/closed division from benchmarks #49 (mmilenkoski)
- Pytorch 1.5.0 #48 (giorgiosav)
- Refactor controlflow #46 (ehoelzl)
- Add Image Recognition Benchmark with DistributedDataParallel #42 (mmilenkoski)
- Pytorch v1.4.0 #41 (ehoelzl)
- Add aggregation by model #38 (ehoelzl)
- Add NCCL & GLOO support to images #35 (giorgiosav)
- Rnn language translation #32 (ehoelzl)
- Linear model #28 (ehoelzl)
- Fix ci #26 (ehoelzl)
- [WIP]Add LSTM language model #25 (Panaetius)

v2.0.0

Implemented enhancements:

- Added Goals to PyTorch Benchmark
- Updated PyTorch Tutorial code
- Changed all images to newest `mlbench-core` version.

v1.1.0

Implemented enhancements:

- Added Tensorflow Benchmark

3.9 Indices and tables

- genindex
- modindex
- search

BIBLIOGRAPHY

[DAM+16] Dipankar Das, Sasikanth Avancha, Dheevatsa Mudigere, Karthikeyan Vaidyanathan, Srinivas Sridharan, Dhiraj D. Kalamkar, Bharat Kaul, and Pradeep Dubey. Distributed deep learning using synchronous stochastic gradient descent. *CoRR*, 2016. URL: http://arxiv.org/abs/1602.06709, arXiv:1602.06709.

[GDollarG+17] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

[HZRS15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, 2015. URL: http://arxiv.org/abs/1512.03385, arXiv:1512.03385.

[HZRS16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. *CoRR*, 2016. URL: http://arxiv.org/abs/1603.05027, arXiv:1603.05027.

[VSP+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, 2017. URL: http://arxiv.org/abs/1706.03762, arXiv:1706.03762.

[WSC+16] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Lukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. Google's neural machine translation system: bridging the gap between human and machine translation. *CoRR*, 2016. URL: http://arxiv.org/abs/1609.08144, arXiv:1609.08144.