MLBench

Optional Semester Project

Student **Martin Milenkoski**

Supervisor **Martin Jaggi**

École Polytechnique Fédérale de Lausanne
Lausanne, Switzerland
June 2020

Table of Contents

Introduction **2**

Project structure **2**

Core repository 2

Benchmarks repository 3

Helm repository 3

Dashboard repository 3

Documentation repository 3

Results repository 4

Comparison to MLPerf **4**

Benchmark Suite 4

Hyperparameter tuning 5

Results reporting 6

Advantages of MLBench 6

Contribution guidelines **6**

Contributing to mlbench-core 6

Contributing to mlbench-benchmarks 8

Contributing to mlbench-helm 8

Contributing to mlbench-dashboard 9

Contributing to mlbench-docs 9

Kubernetes in Docker **9**

Command Line Interface 10

Manual setup script 11

Implementation details 11

Amazon Web Services **12**

Manual setup script 12

Command Line Interface 12

Implementation details 13

Optimizers **14**

Centralized ADAM 14

PowerSGD 14

DistributedDataParallel **15**

Additional contributions **16**

Future work **16**

Conclusion **16**

Pull requests **17**

References **18**

Introduction

MLBench is a framework for benchmarking distributed machine learning algorithms. The main goals of MLBench are to provide a fair benchmarking suite for software and hardware systems and to provide reference implementations for the most common distributed machine learning applications.

In this report, I will summarize my contributions to the framework. Additionally, I will explain the things I learned and the obstacles I faced while using and contributing to the project. During my semester project, I realized that it is not straightforward for new contributors, especially students, to join the project. To this end, I will try to provide a description of the project structure, and explain the process of local development, testing and contribution. I hope that these instructions will be useful for future contributors and will ease the development process. Additionally, I will provide my opinion on what is the appropriate way to export each section of my report .

Project structure

The MLBench codebase is split in several repositories. At first, it is not very easy to understand their purpose. In this section, I will try to summarize the main aspects of each repository.

Core repository

This is the main code repository. It contains the main functionalities of MLBench, and it is the repository where most of the contributions by the core team are done. In this subsection, I will explain the main components of MLBench that reside inside this repository.

- CLI - This is the command line interface for MLBench. Users can use the CLI to create clusters, obtain the dashboard url, start runs, view the status of a run, delete a run or download the results.
- Control flow - The control flow consists of functionalities for running the training and validation steps of different models. It can be reused in different benchmarks and makes the benchmark development easier for users.
- Data Loaders - The data loaders are objects used for loading and preprocessing the train and test datasets for different models.
- Learning Rate (LR) Schedulers - The LR schedulers allow users to adjust the learning rate based on the number of epochs. Different algorithms use different scheduling schemes, and these schemes are implemented as different LR schedulers in MLBench.
- Models - The models are the neural network architectures used for different tasks. The users of MLBench do not have to reimplement the models, but they can adjust some of their parameters.

- Optimizers - The optimizers contain the optimization logic. They dictate how the gradients from different cluster nodes are combined and distributed.
- API - the API class communicates with the dashboard through a REST API, in order to send the user commands to the cluster and obtain the server responses.
- Unit tests - The unit tests are used to make sure that updates to the codebase do not break the core functionalities of the framework.

Benchmarks repository

This repository contains reference implementations of the different tasks. In some cases, it can also contain modified implementations for comparison. Users can use the existing implementations directly, modify the hyperparameters or create their own implementations. The benchmarks provide a type of blueprint on how the implementation should look like. As mentioned above, MLBench provides reference implementations for DataLoaders, Optimizer, LRSchedulers etc, so it is very easy to reuse some parts, while modifying others. In one of the following sections, I will present how I modified one of the benchmarks to use the DistributedDataParallel package while reusing the rest of the code.

Helm repository

Helm is a tool used to install and manage Kubernetes applications. Helm uses a packaging format called charts. A chart is a collection of files that describe a related set of Kubernetes resources. The MLBench helm repository contains the helm chart used for installing MLBench on a Kubernetes cluster. This repository contains the file *values.yaml* which contains the default configuration for MLBench. It also contains the directory *templates* where we can find all templates for generating valid Kubernetes manifests. This is done by combining the templates with the values provided in *values.yaml.* Additionally, this repository contains scripts for manual deployment of MLBench on different cloud providers. Currently, scripts for deploying to Google Cloud, Amazon Web Services (AWS) and Kubernetes in Docker (KIND) are provided. The AWS and KIND scripts are my contribution, and I will explain them in more detail in a later section.

Dashboard repository

This repository contains the code for the MLBench dashboard. It is written in Django, and allows users to perform some of the same functionalities as the CLI, but using a nice user interface. Users can start runs, view the status, download the results, and monitor the output and error log in real time. The last functionality is also very useful for debugging purposes both for core developers of MLBench and for users. The dashboard can only be accessed once the user has created a cluster and deployed MLBench through the CLI.

Documentation repository

This repository contains the files used for generating the MLBench documentation on Read the Docs.

Results repository

This repository contains the official results from the reference implementations for each task. At the moment of writing, the results are not yet available, but they should be in the near future.

Where can this section be exported? This section can be exported as a blog post to introduce the project structure for newcomers. There is a section called *Components* in the documentation that is a more technical description of some of the individual parts of MLBench. Parts of this section may be used to expand that documentation.

Comparison to MLPerf

MLPerf [1] is a broad benchmark suite for measuring the performance of machine learning (ML) software frameworks, ML hardware platforms and ML cloud platforms [2]. In my project, I researched the features of MLPerf and compared them to the features of MLBench. I identified the main similarities and differences. I will present my findings in this section.

Benchmark Suite

The most important aspect of any benchmark suite is the benchmarks it provides. For this reason, I first compare the benchmarks provided by the two frameworks. In Table 1, I present the benchmarks offered by MLPerf. For each benchmark, I show the default dataset, target goal and reference model.

Benchmark	Dataset	Quality Target	Reference Model
Image classification	ImageNet (224x224)	75.9% Top-1 Accuracy	Resnet-50 v1.5
Object detection (light weight)	COCO 2017	23% mAP	SSD-ResNet34
Object detection (heavy weight)	COCO 2017	0.377 Box min AP, 0.339 Mask min AP	Mask R-CNN
Translation (recurrent)	WMT English-German	24.0 BLEU	GNMT
Translation (non-recurrent)	WMT English-German	25.0 BLEU	Transformer
Recommendation	Undergoing modification		
Reinforcement learning	N/A	Pre-trained checkpoint	Mini Go

Table 1. MLPerf Benchmark Suite

In Table 2, I present the benchmarks offered by MLBench with their default datasets, target goals and reference models.

Benchmark	Dataset	Quality Target	Reference Implementation Model
Image classification	CIFAR10 (32x32)	80% Top-1 Accuracy	Resnet-20
Translation (recurrent)	WMT14 EN-DE	24.0 BLEU	GNMT
Translation (non-recurrent)	WMT17 EN-DE	25.0 BLEU	Transformer

Table 2. MLBench Benchmark Suite

As we can see, MLPerf currently offers more benchmarks than MLBench. For this reason, we might need to consider extending our benchmark suite in the future. Furthermore, we can see some other differences. For example, the two frameworks use different datasets, target goals and reference models for the image recognition task.

Hyperparameter tuning

MLPerf restricts the set of hyperparameters that can be tuned. It also allows users to borrow hyperparameters from others. MLBench currently provides exact values for all hyperparameters. I opened a Github issue to discuss the flexibility of our benchmarks [3]. I believe it would be useful to define a policy on what hyperparameters can be changed by users. One possible idea is to reuse the same policy as the one used in MLPerf. In Table 3, I show which hyperparameters are modifiable for different models in MLPerf.

Model	Modifiable Hyperparmeters
All that use SGD	Batch size, Learning-rate schedule parameters
SSD-ResNet-34	Maximum samples per training patch
Mask R-CNN	Number of image candidates
GNMT	Learning-rate decay function, Learning rate, Decay start, Decay interval, Warmup function, Warmup steps
Transformer	Optimizer: Adam (Kingma & Ba, 2015) or Lazy Adam, Learning rate, Warmup steps
NCF	Optimizer: Adam or Lazy Adam, Learning rate, β1, β2

Table 3. MLPerf modifiable hyperparameters.

Results reporting

Both MLBench and MLPerf use end-to-end time to accuracy. However, MLBench reports how much time was used on communication and how much on computing, while MLPerf does not. Furthermore, MLBench reports more fine grained results in comparison to MLPerf. On the other hand, MLPerf reports the average of several runs, while MLBench reports only the result of a single run. While MLPerf allows for distributed training, it does not distinguish the results obtained from single node and multi-node training. Moreover, MLPerf does not distinguish between single GPU and multiple GPUs per node. While the number of nodes and GPUs per node are reported, there does not seem to be any fine grained reporting on the amount of time spent on communication and computation. MLPerf reports only one number for all possible scenarios - time to accuracy. For this reason, MLPerf can not accurately show the effects of scaling the number of nodes or GPUs. They are also not able to pinpoint the reason for an improved or decreased performance of a model because their reported results are not fine grained. On the other hand, MLBench is fully focused on distributed training, can show the effects of scaling, can identify the bottlenecks in the model performance and can accurately show the effects of the model hyperparameters on different parts like communication and computation. In this way, MLBench offers a much more powerful and versatile benchmarking suite than the one offered by MLPerf.

Advantages of MLBench

As mentioned before, MLBench is fully focused on distributed training. It offers fine grained metrics for the execution time of different steps. Additionally, it is much more user-friendly. It offers a command line interface and a dashboard for execution and monitoring of runs. Finally, it has a built-in support for cloud providers like Google Cloud and AWS. In MLPerf, users have to set up everything themselves.

Where can this section be exported? I believe that it would be best to export this section as a blog post that highlights the advantages of MLBench. Additionally, we can also add it as a section in the official documentation.

Contribution guidelines

As mentioned in the introduction, I think that MLBench is missing guidelines for new contributors, especially ones who are not as familiar with the development practices used in the project. For this reason, I believe it is worthwhile to write some guidelines to ease the development process for new contributors. In this section, I will describe some of the common development flows that I used when contributing to MLBench.

Contributing to *mlbench-core*

The first use case I want to discuss is when we want to add a functionality to the core MLBench repository and then test this functionality with some benchmark, either a reference one or our own. For example, this is useful if we want to add a new optimizer, learning rate scheduler, data loader etc, and then run a benchmark to test our implementation. To do this,

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

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

This will bump the version of the development release. We need to be aware that if someone else published a development release on PyPI since our last release, *bumpversion* will not take this into account. In this case, we need to manually bump the version. To do this, we 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, we need to enter the version 2.4.0-dev241 in the files *setup.py*, *setup.cfg* and *mlbench_core/__init__.py*. We 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 we only need to change the numbering and not the formatting. After we have done this, we 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, we should be able to see our release on PyPI. Now, to test this release we 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. We should replace the version with the one we just released. In our previous example we would need to specify *mlbench-core==2.4.0-dev241.* Depending on our changes, we may want to modify the code for the benchmark in the file *main.py.* This is necessary for example when we add a new optimizer and we want to test the benchmark using it. In this case, we need to replace the previous optimizer with the new one in *main.py.* After we are done with the changes, we 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, we need to create an account and login using the command *docker login.* Once we have the image on Docker Hub, we 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, we only need to specify the repository and image name because MLBench automatically looks for the image on Docker Hub. We have to note that this procedure is only required when making changes that need to be tested by running a benchmark. For example, if we want to simply make changes in the CLI, we can modify the file *cli.py* and test it locally using:

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

Contributing to *mlbench-benchmarks*

As discussed before, MLBench offers a choice between different optimizers, learning rate schedulers etc, so we might be interested in modifying the existing benchmark implementations to use different components. To do this, we can follow a similar approach as described in the previous section. We have to clone the *mlbench-benchmarks* repository and modify the file *main.py* file of the corresponding benchmark. We could also write our own implementations of some components and combine them with MLBench. The MLBench blog contains a detailed description on the process of adapting existing PyTorch models to use with MLBench, so I will not go into details regarding this. My main focus here is the case where we 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,* we might need to modify the files *requirements.txt* if we use additional libraries, and the *Dockerfile* if we 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 we are done with the modification, we 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.

Contributing to *mlbench-helm*

The MLBench installation through the CLI automatically uses the latest version of the master branch in the *mlbench-helm* repository. To test our own version of the helm chart without changing the master branch we first need to push our changes to a different branch in *mlbench-helm.* Then, we need to change the file *mlbench_core/cli/cli.py* inside the *mlbench-core* repository. As we mentioned before, this file contains the CLI functionalities and we can test MLBench by running it locally. To use our own version of the helm chart, we need to locate the code for creating the *ChartBuilder* object in the function we want to use. MLBench has different functions for different cloud providers. For testing, we can pick one provider, find the function for creating a cluster on that provider and modify the *ChartBuilder* object to use our own branch. For example, let's say that we have pushed our changes to the branch *new-feature* in the helm repository. Then, we should specify that branch using the *source.reference* parameter like this:

```
chart = ChartBuilder(
            {
            "name": "mlbench-helm",
             "source": {
                 "type": "git",
                 "location": "https://github.com/mlbench/mlbench-helm"
                 "reference": "new-feature"
                 },
             }
        )
```

Now, when we run the command for creating the cluster, it will install MLBench using our own helm chart instead of the default one. Another approach for testing the changes to the helm chart is to use the script for manual setup for different cloud providers. However, in this

way, we can not use the CLI for testing. Therefore, I would recommend using the first approach for testing changes to the helm chart.

Contributing to *mlbench-dashboard*

When we want to test changes to the dashboard we first need to build, tag the image and then push it to a repository on Docker Hub. Let's say we want to push the image to the repository *user/mlbench_master* with the tag *testing.* We 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 we push the image, we can modify the file *values.yaml* in *mlbench-helm* to use our new image. We need to modify the values of *master.image.repository* and *master.image.tag.* In our example, we would set the repository to *user/mlbench_master* and the tag to *testing.* From there, we can use the instructions from the previous section to use the new chart with the CLI. Alternatively, we 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 we 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
```

Contributing to *mlbench-docs*

To contribute to the documentation, we 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.

Where can this section be exported? I believe it would be best to export this section of the report in the official documentation. There is a subsection called *Contributing* under the section *Additional Info* in the documentation, but it is not as detailed as the one I provided here, so we can expand it with this documentation.

Kubernetes in Docker

Using cloud providers like Google Cloud and Amazon Web Services is required when we want to obtain official results and test the real performance of machine learning algorithms. However, when working on features and improvements to the MLBench framework, we do not have to incur additional costs by deploying and testing on a cloud provider. We can deploy a Kubernetes cluster on our local machine using Kubernetes in Docker (KIND). During my project, I developed a script for manual setup of a KIND cluster which is available

as *kind-with-registry.sh* in the *mlbench-helm* repository. Additionally, I integrated the process for deploying a KIND cluster directly in the CLI. In this way, users can create a KIND cluster and then use the CLI to create runs, view their status and download the results. In this section, I will outline the process for deploying a KIND cluster and I will describe the work I did on this task.

Command Line Interface

Users can use the MLBench CLI to deploy a Kubernetes cluster on their local machine. A cluster of 3 nodes with a name *my-cluster* can be created with the following command:

```
$ mlbench create-cluster kind 3 my-cluster
```

The first argument specifies the number of nodes and the second one specifies the cluster name. Additionally, the command accepts the following options:

```
  -r, --registry_name   The name of the local Docker registry
                        (default: kind-registry)
  -p, --registry_port   The port to use for the local registry
                        (default: 5000)
  -h, --host_port       The host port for the local registry
                        (default: 5000)
  -c, --num-cpus        The number of CPUs to use (default: 1)
  -g, --num-gpus        The number of GPUs to use (default: 0)
  -v, --custom-value    Custom value for the helm chart
```

The commands for interacting with the cluster are the same as the ones in the case of Google Cloud described in the MLBench documentation.

In order to use an image in a KIND cluster we need to push it to the local registry. I will illustrate the process using the image *mlbench/pytorch-cifar10-resnet-scaling:2.3.0.* We need to pull the image from Docker Hub, re-tag it and push it to the local registry. If we assume that the local registry runs on port 5000, which is the default one in the CLI, we can achieve this with the following set of commands:

```
$ docker pull mlbench/pytorch-cifar10-resnet-scaling:2.3.0
$ docker tag mlbench/pytorch-cifar10-resnet-scaling:2.3.0
localhost:5000/pytorch-cifar10-resnet-scaling:2.3.0
$ docker push localhost:5000/pytorch-cifar10-resnet-scaling:2.3.0
```

Once the image is pushed on the local registry, we can use it as a custom image to start runs in MLBench. When prompted for the image name, we need to specify the new name and tag. In our example, that would be *localhost:5000/pytorch-cifar10-resnet-scaling:2.3.0.* After we are done with the cluster we can delete it using the following command:

```
$ mlbench delete-cluster kind my-cluster-3
```

The only argument is the name of the cluster.

Manual setup script

I also created a script for manual setup of a KIND cluster. It is available in the *mlbench-helm* repository in the file *kind-with-registry.sh.* The script creates a cluster with two worker nodes. To create a cluster, we can simply run the script. However, in this case MLBench is not automatically installed, so we need to install it by running the following command in the helm repository:

```
$ helm upgrade --wait --recreate-pods -f values.yaml --timeout 900s
--install rel .
```

Before doing this, we need to ensure that we have provided all values in the file *values.yml.*

This will install MLbench as a helm release with the name *rel.* To obtain the link to the MLBench dashboard, we can run the following commands:

```
$ export NODE_PORT=$(kubectl get --namespace default -o
jsonpath="{.spec.ports[0].nodePort}" services rel-mlbench-master)
$ export NODE_IP=$(kubectl get nodes --namespace default -o
jsonpath="{.items[0].status.addresses[0].address}")
$ echo http://$NODE_IP:$NODE_PORT
```

From there, we can start and monitor runs in the dashboard. To delete the cluster, we can run:

```
$ kind delete cluster <cluster-name>
```

The default name of the cluster is *kind,* but it can be modified by setting the environment variable *$KIND_CLUSTER_NAME.*

Implementation details

Both the manual setup and the CLI function follow a similar procedure for deploying MLBench on a KIND cluster. First, we check if a registry with the specified name is already running on the local machine. If not, we have to start it using *docker.* Afterwards, we need to create the config for the KIND cluster. In the config, we need to specify the IP address and port of the local registry and the number of workers. Finally, we can create the KIND cluster with the created config using the *kind* command. In the CLI, the calls to *docker* are implemented through the Docker SDK for Python. Currently, there is no KIND SDK for Python, so we need to call the *kind* command from Python using *subprocess.Popen.* In bash,

we can provide the config directly in the script, but in Python we use a temporary file to write the script and use the file in the *kind* command.

Where can this section be exported? I already included instructions on how to manually set up the KIND cluster in the section *Installation* in the official documentation. There is a subsection on that page called *Automated Setup* that describes how to use the CLI to setup a Google Cloud cluster, and I believe that we should expand it to also describe the setup of a KIND cluster using the CLI.

Amazon Web Services

Manual setup script

As part of the project, I also fixed the script for manual setup of an AWS cluster. Previously, the script was using the *kops* library, but it was not working properly. I fixed the issue by migrating to *eksctl* which is the recommended way for deploying Kubernetes clusters on AWS. The manual script is available in the file *aws_setup.sh* in *mlbench-helm*. We can create an AWS cluster and install MLBench using the following two commands:

```
$ ./aws_setup.sh create-cluster
$ ./aws_setup.sh install-chart
```

To delete the cluster, we can use the following command:

```
$ ./aws_setup.sh delete-cluster
```

The default parameters in the script can be changed using environment variables. The available parameters are listed at the top of the script.

Command Line Interface

Additionally, I integrated the AWS cluster creation process in the MLBench CLI. Here, I mostly use the AWS SDK for Python named *boto3.* A cluster of 3 nodes with a name *my-cluster* can be created with the following command:

```
$ mlbench create-cluster aws 3 my-cluster
```

The first argument specifies the number of nodes and the second one specifies the cluster name. Additionally, the command accepts the following options:

```
  -k, --kubernetes-version   The Kubernetes version to use in the
                             cluster (default: 1.15)
  -t, --machine-type         The AWS machine type (default: t2.medium)
  -c, --num-cpus             The number of CPUs to use (default: 1)
```

```
  -g, --num-gpus              The number of GPUs to use (default: 0)
  -v, --custom-value          Custom value for the helm chart
  -a, --ami-id                The id of the Amazon machine image
                              (default: ami-06d4f570358b1b626)
  -a, --ssh-key               The name of the SSH key to use for the
                              Workers (default: eksNodeKey)
```

After we are done with the cluster we can delete it using the following command:

```
$ mlbench delete-cluster aws my-cluster-3
```

Implementation details

The bash script creates a cluster using *eksctl.* This can be done with a single command using the appropriate options. For installing MLBench on the cluster, we use helm. Additionally, we need to allow public access to the port hosting MLBench on the master node. This is accomplished using the AWS CLI. The implementation in Python is a bit more involved, since there is no single call that takes care of all configuration like in the case of *eksctl.* In this subsection, I will give a high-level overview of the process. First, we need to create a CloudFormation stack to host the cluster. A stack is a collection of AWS resources that we can manage as a single unit. We use *boto3 cloudformation client* to create the stack using a sample template provided by AWS. Creating a stack will create a security group and subnets for the cluster. A security group in AWS acts as a virtual firewall for our instances to control inbound and outbound traffic. A subnet is a range of IP addresses allocated to our stack. Using *boto3 ec2 client,* we ensure that all instances created in our cluster will receive a public IP address upon launch. This is important in order to be able to access the cluster from our local machine. Next, we need to create an IAM role in our AWS account that has a permission to create an EKS cluster. This is done using an *iam client.* IAM stands for Identity and Access Management. It is a service that allows us to manage access to AWS resources and services. An IAM role is an identity with specific permissions. It can be assigned to multiple users in the AWS account. We use the created role to create an EKS cluster. EKS stands for Elastic Kubernetes Service. It is the AWS service that enables creation and management of Kubernetes clusters on AWS. To create the cluster, we also need to provide the security group and subnets associated with the *cloudformation* stack we created earlier. In this way, we assign the EKS cluster to the stack. We also need to specify the kubernetes version we want to use. By default, we use kubernetes 1.15. This is because some of the resources used in MLBench have been deprecated in Kubernetes 1.16. This is something that we should address in the near future. Ideally, we should use the latest Kubernetes version across all cloud providers. Once we have created the EKS cluster, we need to create a group of worker instances. This is done by creating a new CloudFormation stack that will manage them. Users can specify a SSH key they want to use as an option in the command. If they don't, we will use the one with the name *eksNodeKey.* If it doesn't already exist, the function will create it using *boto3 ec2 client.* Finally, we create the nodes using *boto3 cloudformation client.* In order for the nodes to be able to communicate with the cluster, they need some way to authenticate themselves. For this purpose, we use the AWS IAM authenticator. The authenticator generates a token using the AWS credentials of the current

user, and the nodes can use that token to authenticate with the cluster. The cluster also uses the AWS IAM authenticator to verify the token. To enable this authentication method, we simply create a config map that allows the server to verify the authenticity of the nodes. We do this with the Kubernetes SDK for Python. Once we have completed this, the cluster is ready and we can install MLBench using tiller and pyhelm. In the function for deleting the cluster, we first delete the stack with the worker nodes, then the EKS cluster and finally the stack for the EKS cluster. We do not delete the role and SSH key because AWS does not charge for those, and in this way we can avoid recreating them every time we create a cluster. The users can manually delete them if needed.

Where can this section be exported? I believe that we should explain the AWS setup process in the section *Installation* in the official documentation. We should add a subsection for the manual setup and expand the subsection *Automated Setup* to include information for setting up an AWS cluster using the CLI.

Optimizers

MLBench provides implementations for different distributed optimization algorithms. During my project, I worked on the implementation of centralized ADAM and PowerSGD. I will explain these contributions in this section.

Centralized ADAM

Adam [4] is one of the most popular optimization algorithms for training deep neural networks in recent years. In this project, I created a simple extension of the algorithm for the distributed setting. For this purpose, I created a class *CentralizedADAM* which inherits from the PyTorch class *torch.optim.Adam.* This class overrides the constructor and the *step* function. Apart from the standard Adam parameters, the constructor accepts the method for averaging the models and creates the appropriate aggregator. Currently, the only accepted method is the average. This is the case for all optimizers in MLBench. The step function aggregates the gradients of individual nodes using an *all_reduce* operation and performs a step with the aggregated gradient. In the future, we might want to consider more sophisticated implementations of distributed ADAM like the one described in [5].

PowerSGD

PowerSGD [6] introduces a gradient compression method that alleviates the communication bottleneck in distributed optimization. In my project, I adapted the PowerSGD code for the MLBench framework. For this purpose, I created a new aggregation class called *PowerAggregation.* This class contains the function *reduce* which is mostly reused from the original *PowerSGD* code, with slight modifications to adapt the interface and some function calls to the MLBench framework. Currently, the class supports only aggregation by model, which is the aggregation used in the original paper. I believe that it does not make sense to use aggregation by layer in the context of PowerSGD due to the compression scheme. The class also overrides the default *_agg_gradients_by_model* function because it requires a

different format of the input. Currently, only the average method for aggregation is accepted, similar to all other reducers in MLBench. Additionally, I create a new optimizer class called *PowerSGD* which uses *PowerAggregation* for aggregation. This class inherits from *torch.optim.SGD,* which is the default implementation of Stochastic Gradient Descent in PyTorch. *PowerSGD* augments the parent class by aggregating the gradients of the different worker nodes and performing an optimization step using the aggregated gradient.

DistributedDataParallel

DistributedDataParallel is the default implementation of data parallelism across multiple machines in PyTorch. It is a good benchmark for comparing the performance of the reference implementations in MLBench. For this purpose, I modified the reference implementation of the Image Recognition task to use DIstributedDataParallel instead of the default optimization scheme provided by MLBench. This is a good example for users on how to modify specific parts of the benchmarks while reusing most of the code. For this reason, I will use this opportunity to explain in detail what modifications I needed to make. I was able to reuse most of the code from the original implementation. The modification in *main.py* were the following:

- I had to wrap the model in a *torch.nn.parallel.DistributedDataParallel* object. This is done to distribute the model on the workers.
- I had to change the optimizer from *CentralizedSGD* to *torch.optim.SGD* which is the default implementation of stochastic gradient descent in PyTorch. This is done because DDP takes care of the communication of gradients between workers, so we do not have to do it ourselves.
- I had to change the default behaviour to train on a GPU because we wanted to compare our models to the Multi-Process Single-GPU mode of DDP.
- I had to change the docstring to point out that this is not a reference implementation and describe the modifications.

The files *Dockerfile* and *requirements.txt* were reused completely from the original benchmark. The README was also slightly modified to explain the modifications from the reference implementation. I compared the performance of the reference implementation and DDP on the light goal using two different backends. I summarize the results in Table 4.

Implementation	Backend	Total time (s)	Compute time (s)	Communication time (s)
DDP	MPI	97.4	93.2	3.4
Reference	MPI	91	20.5	68
DDP	NCCL	26.2	18.2	7.2
Reference	NCCL	25.8	15.2	9.6

Table 4. Performance of the Reference and DDP implementation on the Image Recognition task

As we can see, our implementation is comparable or better, depending on the backend. This validates the implementation of our optimization algorithms. It is interesting that when using MPI, DDP needs much more compute time than the reference implementation. I double checked the correctness of the results, but I am not sure what is the reason for this.

Where can this section be exported? I believe that we can export this section as a blog post where we describe the process of modifying the reference implementation and compare the performance of our implementation and the DistributedDataParallel implementation. Potentially, we could split the modification part and the comparison part in two separate blog posts.

Additional contributions

In the previous sections, I mentioned some of the bigger contributions that I made on the MLBench codebase. In this section, I want to point out some of the contributions that did not justify the inclusion of an additional section. For example, I worked on the inclusion of the GLOO backend in the benchmarks. This work was later incorporated and extended by Giorgio in his work on backends. I created a function that generates an optimizer object based on the name passed as argument by the user. This was done so that users can select an optimizer using a command line argument. This has not been included in the benchmarks yet, since the discussion on the benchmark flexibility is still ongoing. I also worked on some documentation issues like removing the mentions of the open and closed division and deleting the section on the deprecated Docker in Docker package and including a new section for Kubernetes in Docker. Finally, I fixed several bugs in the software that arose during my project. There is a separate section at the end of the report listing all my pull requests.

Future work

There are many possible aspects in which we can extend the framework in the future. First and foremost, we need to generate the official benchmark results and write better documentation, especially for new users and contributors. I hope that some of the work in this report can be reused in that direction. Next, we should write some blog posts on the official results and promote the framework. In the longer run, we can maybe consider extending our benchmark suite and defining clear policies on hyperparameter modification and benchmark submission. Support for other cloud providers like Microsoft Azure is also something that may be useful for many users. Finally, keeping the framework up to date with all the underlying libraries is something that we should strive to achieve and maintain.

Conclusion

In this semester project, my contributions were mainly in software. In this report, I listed the most important features that I contributed to MLBench. Additionally, I did a bit of competition

research with the case of MLPerf. I dedicated a part of this report to documenting the project structure and providing contribution guidelines for future contributors. Finally, I also included user guidelines for some of the features that I contributed. I believe that these can easily be extended to blog posts on the MLBench blog, and can be very valuable to new users and contributors.

Pull requests

This is the list of all Pull Requests that I created and resolved during my project:

- Remove dimension check in TopKAccuracy
- Add Gloo support to PyTorch images
- Adds centralized Adam implementation
- Add get_optimizer to create optimizer object
- Add default name of output file in CLI
- Remember user input in mlbench run
- Remove dind-script.sh from docs
- Add PowerSGD optimizer
- Refactor pack_tensors
- Add Image Recognition Benchmark with DistributedDataParallel
- Add setup script for kind with local registry
- Add KIND installation instructions
- Add support for kind cluster creation in the CLI
- Remove open/closed division from benchmarks
- Remove open/closed division from the docs
- Switch to eksctl for aws deployment
- Add AWS support in CLI
- Add return_code check in test_cli

References

[1] P. Mattson *et al.*, “MLPerf Training Benchmark,” *ArXiv191001500 Cs Stat*, Mar. 2020, Accessed: Jun. 07, 2020. [Online]. Available: http://arxiv.org/abs/1910.01500.

[2] S. Condon, “Google, Nvidia tout advances in AI training with MLPerf benchmark results,” *ZDNet*. https://www.zdnet.com/article/google-nvidia-tout-advances-in-ai-training-with-mlperf-benchmark-results/ (accessed Jun. 07, 2020).

[3] “Benchmark flexibility design discussion · Issue #85 · mlbench/mlbench-core,” *GitHub*. https://github.com/mlbench/mlbench-core/issues/85 (accessed Jun. 07, 2020).

[4] D. P. Kingma and J. Ba, “Adam: A Method for Stochastic Optimization,” *ArXiv14126980 Cs*, Jan. 2017, Accessed: Jun. 07, 2020. [Online]. Available: http://arxiv.org/abs/1412.6980.

[5] P. Nazari, D. A. Tarzanagh, and G. Michailidis, “DADAM: A Consensus-based Distributed Adaptive Gradient Method for Online Optimization,” *ArXiv190109109 Cs Math Stat*, May 2019, Accessed: Jun. 07, 2020. [Online]. Available: http://arxiv.org/abs/1901.09109.

[6] T. Vogels, S. P. Karimireddy, and M. Jaggi, “PowerSGD: Practical Low-Rank Gradient Compression for Distributed Optimization,” in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d\textquotesingle Alché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 14259–14268.