AdaptDL

Petuum, Inc.

Apr 20, 2022

CONTENTS:

1 Why AdaptDL? 3
- 1.1 Efficient Resource Management . . . 3
- 1.2 Adaptive Batch Size Scaling . . . 4
- 1.3 Easy-to-use Elastic API . . . 4

2 Getting Started 5
- 2.1 Installation . . . 5
- 2.2 AdaptDL Command-line Interface . . . 9
- 2.3 AdaptDL with PyTorch . . . 14
- 2.4 Standalone Training . . . 17
- 2.5 Using the Adaptive Tune Trial Scheduler . . . 20
- 2.6 Adaptdl on Ray AWS . . . 23
- 2.7 adaptdl package . . . 26

Python Module Index 47

Index 49

AdaptDL is a *resource-adaptive* deep learning (DL) training and scheduling framework, and is part of the CASL open source project. The goal of AdaptDL is to make distributed DL easy and efficient in dynamic-resource environments such as shared clusters and the cloud.

AdaptDL consists of two components which can be used together with or separately from one another:

- **adaptdl-sched:** A cluster scheduler on Kubernetes optimized for distributed deep learning training.
- **adaptdl:** A library for adaptive batch sizes that can efficiently scale distributed training to many nodes.

Some core features offered by AdaptDL are:

- Elastically schedule distributed DL training jobs in shared clusters.
- Cost-aware resource auto-scaling in cloud computing environments (e.g. AWS).
- Automatic batch size and learning rate scaling for distributed training.

AdaptDL supports PyTorch training programs. TensorFlow support coming soon!

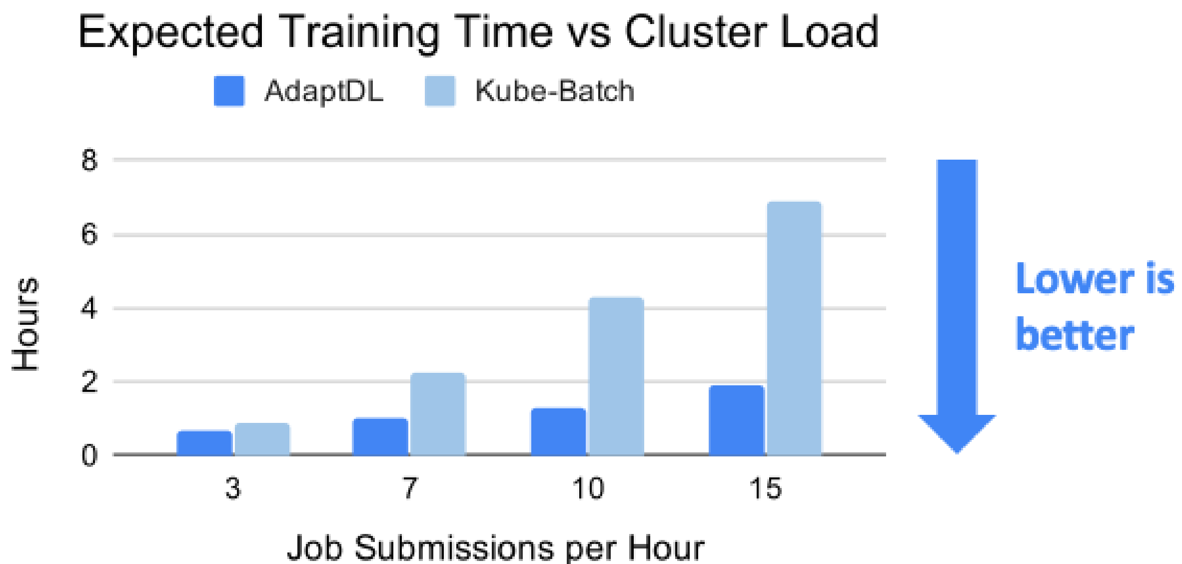
CHAPTER

ONE

WHY ADAPTDL?

1.1 Efficient Resource Management

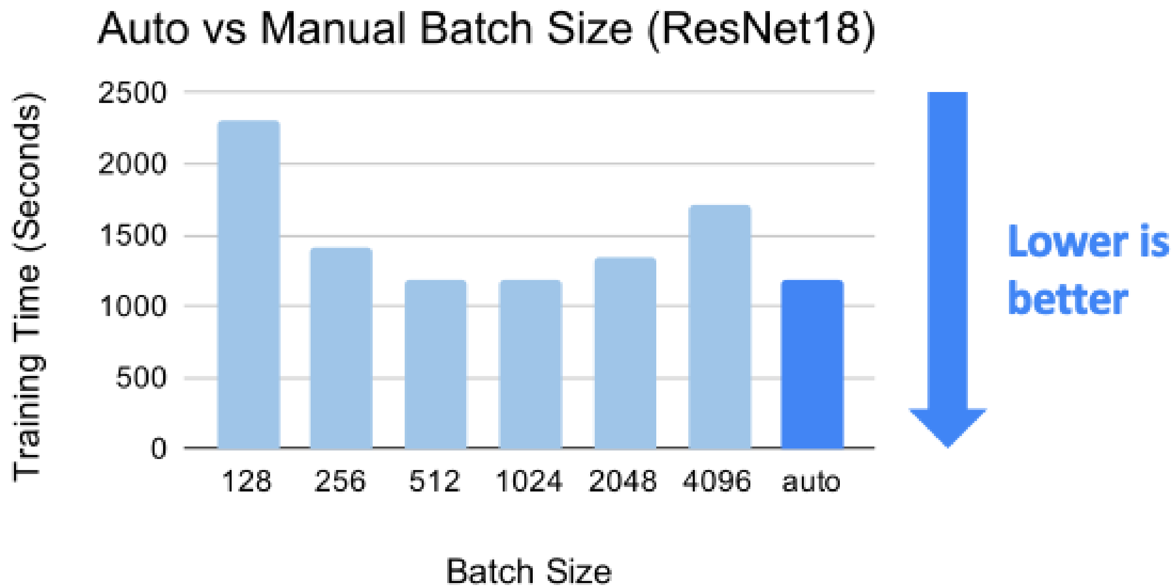
The AdaptDL scheduler directly optimizes cluster-wide training performance and resource utilization, by using a genetic algorithm to periodically optimize resource allocations for all jobs. Through elastic re-scaling, co-adapting batch sizes and learning rates, and avoiding network interference, AdaptDL significantly accelerates shared-cluster training when compared with alternative schedulers. For details, please see our OSDI'21 research paper.



In the cloud (e.g. AWS), AdaptDL auto-scales the size of the cluster based on how well those cluster resources are utilized. AdaptDL automatically provisions spot instances when available to reduce cost by up to 80%.

1.2 Adaptive Batch Size Scaling

Efficient distributed training requires careful selection of the batch size and learning rate, which can be tricky to find manually. AdaptDL offers automatic batch size and learning rate scaling, which enables efficient distributed training without requiring manual effort. To achieve this, AdaptDL measures the system performance and gradient noise scale during training, adaptively selects the most efficient batch size, and scales the learning rate using AdaScale.



1.3 Easy-to-use Elastic API

Making training programs run elastically can be challenging and error-prone. AdaptDL offers APIs which make it easy to enable elasticity for data-parallel PyTorch programs. Simply change a few lines of code, without heavy refactoring!

BEFORE:

```
torch.distributed.init_process_group("nccl")
model = torch.nn.parallel.DistributedDataParallel(model)
dataloader = torch.utils.data.DataLoader(dataset, batch_size=128)
for epoch in range(100):
    ...
```

AFTER:

```
adaptdl.torch.init_process_group("nccl")
model = adaptdl.torch.AdaptiveDataParallel(model, optimizer)
dataloader = adaptdl.torch.AdaptiveDataLoader(dataset, batch_size=128)
for epoch in adaptdl.torch.remaining_epochs_until(100):
    ...
```

CHAPTER

TWO

GETTING STARTED

AdaptDL consists of a *job scheduler* and an *adaptive training library*. They can be used in multiple ways:

1. Scheduling multiple training jobs on a shared Kubernetes cluster or the cloud (*Scheduler Installation*).
2. Adapting the batch size and learning rate for a single training job (*Standalone Training*).
3. As a Ray Tune Trial Scheduler (*Tune Trial Scheduler*).
4. As a single training job running on a Ray AWS cluster (*Ray AWS Launcher*)

2.1 Installation

The following pages describe how to deploy the AdaptDL scheduler to manage jobs in a shared Kubernetes cluster. If you are interested in training in a standalone setting, see *Standalone Training*.

2.1.1 Deploying MicroK8s for AdaptDL

This page describes how to deploy a single-node MicroK8s Kubernetes instance on which AdaptDL can be run. Refer to other pages if you want to run AdaptDL on *an existing Kubernetes cluster*, or on *an auto-scaling cluster with EKS*.

Note: The instructions on this page assume Ubuntu 18.04 or above with sudo access.

Installing MicroK8s

First, install MicroK8s using Snap:

```
$ sudo snap install microk8s --classic --channel=1.18/stable
```

The above command should install a barebones MicroK8s instance locally. Next, enable dns:

```
$ sudo microk8s enable dns
```

Enable gpu and storage:

```
$ sudo microk8s enable gpu storage
```

The above command enables pods to utilize GPUs if available, and allows local storage to be used for AdaptDL training checkpoints.

Initialize Helm, which is a package manager that can later be used to deploy the AdaptDL scheduler:

```
$ sudo microk8s enable helm
$ sudo microk8s helm init --stable-repo-url=https://charts.helm.sh/stable
$ sudo helm repo add stable https://charts.helm.sh/stable
```

Interacting with MicroK8s

Once MicroK8s is installed, you can interact with it via `microk8s.kubectl`, in the same way as using `kubectl` to interact with other Kubernetes instances:

```
$ sudo microk8s.kubectl get nodes
```

Example output:

```
NAME       STATUS    ROLES    AGE    VERSION
gpu00100   Ready     <none>   10m    v1.18.8
```

If you prefer to omit `sudo`, add your user to the `microk8s` group, and then re-login to your shell:

```
$ sudo usermod -a -G microk8s $USER
```

If you prefer to use `kubectl` rather than `microk8s.kubectl`:

```
$ mkdir -p $HOME/.kube
$ sudo microk8s kubectl config view --raw > $HOME/.kube/config
$ sudo chown -f -R $USER ~/.kube
```

The above step is recommended when later deploying AdaptDL onto MicroK8s.

Next Steps

Once your MicroK8s instance is installed and running, you can *deploy the AdaptDL scheduler*.

2.1.2 Provisioning EKS for AdaptDL

This page describes how to setup an AWS EKS cluster that auto-scales according to cluster load. Refer to other pages if you want to run AdaptDL on *an existing Kubernetes cluster*, or on *an a single node with MicroK8s*.

Note: The instruction on this page assume `eksctl`, `kubectl`, `helm` and `awscli` are installed locally. You can follow this guide to install all the tools needed.

Attention: This guide will provision AWS resources which will cost money. As of August 2020, you pay $0.10 per hour for each Amazon EKS cluster that you create. $0.30 GB-Month for the EFS storage and $0.526 per hour per `g4dn.xlarge` instance that you will end up using, starting with one. Note because the cluster is auto-scaling, additional instances will be spawned only when needed and you will be charged only for the duration of their lifetimes.

Provisioning the Cluster

You may use the provided manifest to create the cluster. Some configurations may be changed as per your preferences by downloading and modifying the file.

```
eksctl create cluster -f https://raw.githubusercontent.com/petuum/adaptdl/master/deploy/
↪eks/adaptdl-eks-cluster-on-demand.yaml
```

This will provision an elastic EKS cluster with name `adaptdl-eks-cluster` with 1 minimum and 4 maximum nodes in the `us-west-2` region. All nodes are on-demand `g4dn.xlarge` instances with a single GPU each. You can change the instance type and auto-scaling limits by changing `nodeGroups.instanceType`, `nodeGroups.minSize`, and `nodeGroups.maxSize`, respectively. You can also change the cluster `name`, AWS `region` of your choice.

Make sure the `CLUSTER_NAME` and `AWS_REGION` environment variables reflect the correct values after this step, for example:

```
export CLUSTER_NAME=adaptdl-eks-cluster
export AWS_REGION=us-west-2
```

Provisioning EFS

AdaptDL depends on a distributed filesystem like EFS to save and load checkpoints during training. You may follow the instructions from this website to provision an EFS volume for your cluster.

Next, install the EFS provisioner Helm chart. Make sure you have set the `FILE_SYSTEM_ID` environment variable according to the linked instructions.

```
helm repo add stable https://kubernetes-charts.storage.googleapis.com/

helm repo update

helm install stable/efs-provisioner \
--set efsProvisioner.efsFileSystemId=$FILE_SYSTEM_ID \
--set efsProvisioner.awsRegion=$AWS_REGION \
--generate-name
```

Installing the Cluster Autoscaler

```
helm repo add autoscaler https://kubernetes.github.io/autoscaler

helm repo update

helm install autoscaler/cluster-autoscaler-chart \
--set autoDiscovery.clusterName=$CLUSTER_NAME \
--set awsRegion=$AWS_REGION \
--generate-name
```

To verify that cluster-autoscaler has started, run:

```
kubectl --namespace=default get pods -l "app.kubernetes.io/name=aws-cluster-autoscaler-
↪chart"
```

Should show the Cluster Autoscaler pod as `Running`

Installing the NVIDIA Plugin

```
kubectl create -f https://raw.githubusercontent.com/NVIDIA/k8s-device-plugin/v0.6.0/
↪nvidia-device-plugin.yml
```

(Optional) Registry Access

If you will be using AdaptDL's insecure registry, you will need to add a new rule to the security group associated with the nodes of the cluster. You may need help from your AWS administrator to perform this step.

```
SECURITY_GROUP=$(aws cloudformation describe-stack-resources --stack-name \
eksctl-$CLUSTER_NAME-nodegroup-ng-1 --query \
'StackResources[?LogicalResourceId == `SG`].[PhysicalResourceId]' --output text)

aws ec2 authorize-security-group-ingress --group-id $SECURITY_GROUP \
--protocol tcp --port 32000 --cidr 0.0.0.0/0
```

Cleaning Up

Once you are done with the cluster, you can clean up all AWS resources with:

```
eksctl delete cluster --name $CLUSTER_NAME

for target in `aws efs describe-mount-targets --file-system-id $FILE_SYSTEM_ID --query
↪'MountTargets[].MountTargetId' --output text`; \
do aws efs delete-mount-target --mount-target-id $target; done

aws efs delete-file-system --file-system-id $FILE_SYSTEM_ID
```

Next Steps

Once your EKS cluster is provisioned and running, you can *deploy the AdaptDL scheduler*.

2.1.3 Installing the AdaptDL Scheduler

This page shows how to install the AdaptDL scheduler on an existing Kubernetes instance. If you do not have a running Kubernetes instance, you may refer to other pages to *deploy a single-node MicroK8s instance*, or to *provision an auto-scaling cluster on EKS*.

Note: The following instructions assume `kubectl` (installation instructions) and `helm` (installation instructions) are installed locally and configured with administrator access to an existing Kubernetes instance.

Install the AdaptDL Helm Chart

The AdaptDL scheduler can be installed in just one command using Helm:

```
$ helm install adaptdl adaptdl-sched --repo https://github.com/petuum/adaptdl/raw/helm-
↪repo \
  --namespace adaptdl --create-namespace --set docker-registry.enabled=true
```

The above command installs Kubernetes deployments for the AdaptDL scheduler service, as well as a Docker registry. The Docker registry is used to store intermediate Docker images when submitting jobs with the AdaptDL CLI.

> **Danger:** The Docker registry installed with the AdaptDL scheduler is *insecure*. Please install an alternative secure registry for serious use! The included Docker registry may be disabled by omitting the `--set docker-registry.enabled=true` option, and then the AdaptDL CLI may be configured to use the alternative secure registry.

Note: If installing AdaptDL together with the insecure registry, you may need to first install the Helm stable repository with the `helm repo add stable https://charts.helm.sh/stable` command.

Check that the AdaptDL scheduler and Docker registry are running:

```
$ kubectl get pods -n adaptdl
```

Example output:

```
adaptdl-adaptdl-sched-7d8b689f45-9ds8h   3/3     Running             0           2m37s
adaptdl-registry-7f45598964-t8df6        1/1     Running             0           2m37s
```

Next Steps

Once the AdaptDL scheduler is installed and running, you may *run AdaptDL jobs using the AdaptDL CLI*.

2.2 AdaptDL Command-line Interface

The following pages describe how to use AdaptDL CLI to submit and manage jobs in an AdaptDL-scheduled cluster. If you are instead interested in training in a standalone setting, see *Standalone Training*.

Note: The following instructions assume you have already installed the AdaptDL scheduler on Kubernetes. If you have not, see *Installation*.

2.2.1 Submitting a Simple Job

This page is an introduction to running AdaptDL jobs using a simple "Hello, world!" program. The goal is to show the basics of creating and interacting with AdaptDL jobs. For an introduction to modifying existing PyTorch code to use AdaptDL, please see *AdaptDL with PyTorch*.

Installation

```
python3 -m pip install adaptdl-cli
```

Writing a Simple Program

For the purpose of this guide, you will want a simple python script that produces output to `adaptdl.env.share_path()`, the directory used for your job for storing general files.

For example, you may copy the following code (into `hello_world/hello_world.py`):

```
import adaptdl.env
import os
import time

print("Hello, world!")

with open(os.path.join(adaptdl.env.share_path(), "foo.txt"), "w") as f:
    f.write("Hello, world!")

time.sleep(100)
```

Please note that stdout is only accessible while a job is still running. Therefore, the `time.sleep(100)` call is important for this tutorial.

Writing a Dockerfile

In order to run your application code, the job containers need access to the code directly. A simple method is to create a docker image containing the application.

Currently the `adaptdl` cli requires you to be able to push to and the cluster to be able to pull from a docker registry. This may be dockerhub, or it may be your own private docker registry. Please ensure that that is set up before proceeding.

Copy the following docker file into `hello_world/Dockerfile`:

```
FROM python:3.7-slim
RUN python3 -m pip install adaptdl

COPY hello_world.py /root/hello_world.py

ENV PYTHONUNBUFFERED=true
```

Tip: If the Dockerfile is not written carefully, the Docker build step can take a long time. Make sure to follow the best practices when writing your Dockerfile so your builds are as fast as possible:

1. Exploiting caching in Dockerfile to re-use layers and speed up builds

2. Using .dockerignore to minimize the size of your docker context.

In particular, you should (almost) always have a `.dockerignore` file that contains `.git` and other large files/directories which are not used in your containers.

Configuring the Job

AdaptDL jobs are specified as Kuberenetes Resource. The following yaml file defines the job specification for your hello world application:

Example (in `hello_world/adaptdljob.yaml`):

```
apiVersion: adaptdl.petuum.com/v1
kind: AdaptDLJob
metadata:
  generateName: hello-world-
spec:
  template:
    spec:
      containers:
      - name: main
        command:
        - python3
        - /root/hello_world.py
```

Submitting the Job

Run the following AdaptDL cli command from your client.

```
adaptdl submit hello_world
```

Note: If you are using Docker for Mac with AdaptDL's built-in insecure registry, the first run of `adaptdl submit` may fail with an error similar to:

```
Get https://host.docker.internal:59283/v2/: x509: certificate signed by unknown authority
```

You may need to restart Docker, and `adaptdl submit` should work thereafer.

This will create the AdaptDL Kubernetes job object for your application. Once this is created, the AdaptDL scheduler will recognize the job and schedule it for execution. Please note that for this command to work, the docker file created in step 3 must be located in *hello_world/Dockerfile* and the yaml created in step 4 must be located in *hello_world/adaptdljob.yaml*.

Monitoring the Job

Once the job object has been created, you can find more information about the job using

```
adaptdl ls
```

This should produce some output similar to

```
Name                                                          Status     Start(UTC)  ␣
↪ Runtime  Rplc  Rtrt
hello-world-kgjsc                                             Running    Aug-24 18:47␣
↪ 1 min    1     0
```

Once the `Status` is listed as `Running` and not `Pending`, then the AdaptDL scheduler has created pods for your AdaptDL job. Use the following command to find out more details about the pods:

```
kubectl get pods
```

This should produce an output that looks like

```
NAME                                                      READY    STATUS     RESTARTS␣
↪   AGE
adaptdl-adaptdl-sched-856cc685c4-hhdks                    3/3      Running    0        ␣
↪   8h
hello-world-kgjsc-a7fe6b49-e673-11ea-a27e-061e69fb5c39-0-0  1/1      Running    0        ␣
↪   20s
```

Note that this gets all of the pods in the default namespace, including the scheduler. To restrict this to just the pods created for your job, use `kubectl get pods | grep hello-world`.

When the phase is listed as `Running`, as opposed to `ContainerCreating`, then you can get the stdout and stderr logs via the following, (replacing `<pod-name>` with the name value you got from `kubectl get pods`):

```
kubectl logs <pod-name>
```

This should produce output of `Hello, world!`.

Please note that this method of getting stdout and stderr output requires the pod to still exist. However, when an AdaptDL job finishes or rescales, the worker pods are deleted. For more durable logging, it is advised to write to a file.

Retrieving Output Files

Use the following to copy result files to your client machine. Please replace `<adaptdl-job>` with the name value from the output of `adaptdl ls` in step 10:

```
adaptdl cp <adaptdl-job>:/adaptdl/share/foo.txt foo.txt
```

`foo.txt` on your local client should then contain `hello world`

Deleting the Job

Delete the job with kubectl: `kubectl delete adaptdljob <adaptdl-job>`. Again, replace the name parameter with the one from before. This will delete the AdaptDL kubernetes object from your job, which will also delete any running pods or other attached resources. Please note that this may cause files the job has written to to no longer be available.

(Advanced) External Registry

If possible, we recommend using a secure external Docker registry instead of the default insecure registry installed along with the AdaptDL scheduler. To do this, you'll need to export two environment variables to let AdaptDL know the full reponame to use, say `registry.example.com/adaptdl-submit`, along with registry credentials `mysecret`. Refer to this website for how to create one.

```
export ADAPTDL_SUBMIT_REPO=registry.example.com/adaptdl-submit
export ADAPTDL_SUBMIT_REPO_CREDS=mysecret
```

Then do `docker login` in with the registry credentials.

2.2.2 Integrating your AdaptDL job with TensorBoard

Tensorboard provides a simple way to collect and visualize model performance, statistics, and weights. AdaptDL provides integration with tensorboard across replicas via AdaptDL's command line interface.

AdaptDL provides a way to deploy a Tensorboard instance on your kubernetes cluster that your AdaptDL jobs can interact with. This tutorial demonstrates how to have your AdaptDL jobs write to Tensorboard and how to access the Tensorboard UI.

Modifying Your Code

The AdaptDL CLI provides the environment variable `ADAPTDL_TENSORBOARD_LOGDIR` as the log directory for AdaptDL TensorBoard deployments. Use `$ADAPTDL_TENSORBOARD_LOGDIR/<job-name>` for the particular job you are running via the following code:

```
os.path.join(os.getenv("ADAPTDL_TENSORBOARD_LOGDIR"),
                       adaptdl.env.job_id())
```

Following *AdaptDL with PyTorch*, the following changes are made to the `test` function to write to tensorboard:

```
with stats.synchronized():
    test_loss = stats["test_loss"] / len(test_loader.dataset)
    correct = stats["correct"]
    tensorboard_dir = os.path.join(os.getenv("ADAPTDL_TENSORBOARD_LOGDIR", "/tmp"),
                                   adaptdl.env.job_id())
    with SummaryWriter(tensorboard_dir) as writer:
        writer.add_scalar("Test/Loss", test_loss, epoch)
        writer.add_scalar("Test/Accuracy", 100. * correct / len(test_loader.dataset),␣
↪epoch)
```

See `mnist_tensorboard.py` for more context.

Deploying Tensorboard

Launch the AdaptDL TensorBoard deployment with

```
adaptdl tensorboard create my-tensorboard
```

This will create a deployment running tensorboard and a service to expose tensorboard's port.

Attaching TensorBoard

When creating your AdaptDL job via the adaptdl cli, use the flag `--tensorboard my-tensorboard`. This will attach the necessary persistent volume claims and environment variables to your AdaptDL job.

For example, to launch the Tensorboard MNIST example from above, run the following in your command line.

```
adaptdl submit . --tensorboard my-tensorboard -d tutorial/Dockerfile -f tutorial/
↪adaptdljob.yaml
```

Accessing TensorBoard

To access the GUI of your TensorBoard instance running in Kubernetes, you can start a proxy to it locally:

```
$ adaptdl tensorboard proxy my-tensorboard -p 8080
Proxying to TensorBoard instance my-tensorboard at http://localhost:8080
```

The proxy will keep running until you manually stop it by sending an interrupt. Now, you can view your TensorBoard instance by pointing your favorite browser to `http://localhost:8080`.

2.3 AdaptDL with PyTorch

This page describes the steps needed to modify a simple MNIST example to use AdaptDL. Please see `mnist_original.py` for the original version and `tutorial/mnist_step_<#>.py` for the resulting changes from each step number of this tutorial.``diff`` may be useful here to compare versions.

2.3.1 Initializing AdaptDL

Once the training model `model` with optimizer `optimizer` and (optional) learning rate scheduler `scheduler` have been created, register all three with the following commands:

```
adaptdl.torch.init_process_group("nccl" if torch.cuda.is_available()
                                 else "gloo")
model = adaptdl.torch.AdaptiveDataParallel(model, optimizer, scheduler)
```

Please note that `init_process_group` must be called before the `AdaptiveDataParallel` object is created

In the MNIST tutorial example (`mnist_step_1.py`), the changes will look like the following:

```
model = Net().to(device)
optimizer = optim.Adadelta(model.parameters(), lr=args.lr)

scheduler = StepLR(optimizer, step_size=1, gamma=args.gamma)
```

(continues on next page)

(continued from previous page)

```
adaptdl.torch.init_process_group("nccl" if torch.cuda.is_available()
                                 else "gloo") # Changed
model = adaptdl.torch.AdaptiveDataParallel(model, optimizer, scheduler) # Changed
```

2.3.2 Dataloading

AdaptDL requires you to use `adaptdl.torch.AdaptiveDataLoader`. This will require you to first have your dataset as a torch dataset object. From there, the `AdaptiveDataLoader` supports the same arguments as the standard PyTorch DataLoader class. Furthermore, the batchsize is not guaranteed to be the same as the `batch_size` argument. However, if batchsize autoscaling is not enabled (see part 3), then the global batchsize will be very close that provided via `batch_size`.

In the MNIST example (`mnist_step_2.py`), this is a matter of changing the dataloaders from

```
dataset1 = datasets.MNIST('../data', train=True, download=True,
                          transform=transform)
dataset2 = datasets.MNIST('../data', train=False,
                          transform=transform)
train_loader = torch.utils.data.DataLoader(dataset1, batch_size=64,
                                           num_workers=1, shuffle=True)
test_loader = torch.utils.data.DataLoader(dataset2, batch_size=64,
                                          num_workers=1, shuffle=True)
```

to

```
dataset1 = datasets.MNIST('../data', train=True, download=True,
                          transform=transform)
dataset2 = datasets.MNIST('../data', train=False,
                          transform=transform)
train_loader = adaptdl.torch.AdaptiveDataLoader(dataset1, drop_last=True, batch_size=64,
                                               num_workers=1, shuffle=True)
test_loader = adaptdl.torch.AdaptiveDataLoader(dataset2, batch_size=64,
                                              num_workers=1, shuffle=True)
```

Setting `drop_last=True` allows the dataloader to properly deal with remainders when dividing the dataset by the number of replicas

2.3.3 Adaptive Batch Size

Enable AdaptDL to automatically scale the batch size based off of throughput and gradient statistics via

```
data_loader.autoscale_batch_size(
    max_global_batchsize,
    local_bsz_bounds=(min_local_batchsize, max_local_batchsize))
```

Note: this will allow the batchsize to change dynamically in training via Adascale. Also note that this will generally require your optimizer to be SGD.

In the context of the MNIST example (`mnist_step_3.py`), the following change will need to be made:

```
train_loader = adaptdl.torch.AdaptiveDataLoader(dataset1, drop_last=True, **kwargs)
test_loader = adaptdl.torch.AdaptiveDataLoader(dataset2, **kwargs)

train_loader.autoscale_batch_size(1028, local_bsz_bounds=(32, 128))
```

Please note that this call is optional, but required to allow the global batchsize to change dynamically over time.

2.3.4 Training Loop

The core training loop requires the following change from:

```
for epoch in range(1, args.epochs + 1):
    train(args, model, device, train_loader, optimizer, epoch)
    test(model, device, test_loader)
    scheduler.step()
```

to

```
for epoch in adaptdl.torch.remaining_epochs_until(args.epochs): # Changed
    train(args, model, device, train_loader, optimizer, epoch)
    test(model, device, test_loader)
    scheduler.step()
```

The call `adaptdl.torch.remaning_epochs_until(args.epochs)` will resume the epochs and batches progressed when resuming from checkpoint after a job has been rescaled. See (`mnist_step_4.py`).

2.3.5 Statistics Accumulation

To calculate useful metrics like loss or accuracy across replicas, use the `adaptdl.torch.Accumulator` class, which is a `dict`-like object that sums across replicas when `synchronized` is called. However, outside of the `stats.synchronized()` context, get operations are not supported. Furthermore, calling `stats.synchronized()` forces blocking for synchronization across all replicas.

Whereas before collecting test data would look like:

```
def test(model, device, test_loader):
    model.eval()
    test_loss = 0
    correct = 0
    with torch.no_grad():
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)
            output = model(data)
            test_loss += F.nll_loss(output, target, reduction='sum').item()
            pred = output.argmax(dim=1, keepdim=True)
            correct += pred.eq(target.view_as(pred)).sum().item()

    test_loss /= len(test_loader.dataset)

    print('\nTest set: Average loss: {:.4f}, Accuracy: {}/{} ({:.0f}%)\n'.format(
        test_loss, correct, len(test_loader.dataset),
        100. * correct / len(test_loader.dataset)))
```

With AdaptDL statistics accumulation, it would look like:

```
def test(model, device, test_loader):
    model.eval()
    stats = adaptdl.torch.Accumulator() # Changed in step 5
    with torch.no_grad():
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)
            output = model(data)
            # CHANGED:
            stats["test_loss"] += F.nll_loss(output, target, reduction='sum').item()
            pred = output.argmax(dim=1, keepdim=True)
            stats["correct"] += pred.eq(target.view_as(pred)).sum().item()

    with stats.synchronized(): # Changed in step 5
        test_loss = stats["test_loss"] / len(test_loader.dataset) # Changed
        correct = stats["correct"] # Changed in step 5

        print('\nTest set: Average loss: {:.4f}, Accuracy: {}/{} ({:.0f}%)\n'.format(
            test_loss, correct, len(test_loader.dataset),
            100. * correct / len(test_loader.dataset)))
```

See (`mnist_step_5.py`) for the full changes.

2.4 Standalone Training

This tutorial shows how to run AdaptDL training code in a standalone setting, outside of an AdaptDL-scheduled cluster. Standalone training has no dependency on deploying Kubernetes or the AdaptDL scheduler. It can be useful for:

1. Distributed training with adaptive batch sizes in a dedicated cluster.
2. Local testing the training code before submitting to an AdaptDL cluster.

2.4.1 Local Training

Any training code that uses AdaptDL APIs can be run locally as a single process. All that's needed is to install the `adaptdl` package, and run the code as a regular python program.

```
$ python3 -m pip install adaptdl
```

As an example, we shall run the simple MNIST training script (`mnist_step_5.py`).

```
$ python3 mnist.py
```

Output:

```
WARNING:adaptdl.reducer:Could not connect to root, trying again...
INFO:adaptdl.reducer:Master waiting for connections on 0
INFO:adaptdl.reducer:rank 0 connecting to 0.0.0.0 on port 36405
INFO:adaptdl.torch:Initializing torch.distributed using tcp://0.0.0.0:39345?rank=0&world_
↪size=1
INFO:adaptdl.torch:torch.distributed initialized
```

(continues on next page)

(continued from previous page)

```
INFO:adaptdl.torch.epoch:starting at epoch 0
Train Epoch: 0 [0/60000 (0%)]   Loss: 2.318445
Train Epoch: 0 [640/60000 (1%)] Loss: 1.647522
...
...
...
Train Epoch: 13 [58880/60000 (98%)]  Loss: 0.003577
Train Epoch: 13 [59520/60000 (99%)]  Loss: 0.034688

Test set: Average loss: 0.0267, Accuracy: 9911/10000 (99%)
```

2.4.2 Manual Checkpoint-Restart

When a training program is running locally, a checkpoint can be triggered by sending an interrupt (CTRL-C in most terminals). The environment variable ADAPTDL_CHECKPOINT_PATH specifies where the checkpoint should be located.

```
$ mkdir mnist-checkpoint
$ ADAPTDL_CHECKPOINT_PATH=mnist-checkpoint python3 mnist.py
```

Output (after sending CTRL-C during training):

```
WARNING:adaptdl.reducer:Could not connect to root, trying again...
INFO:adaptdl.reducer:Master waiting for connections on 0
INFO:adaptdl.reducer:rank 0 connecting to 0.0.0.0 on port 51067
INFO:adaptdl.torch:Initializing torch.distributed using tcp://0.0.0.0:24997?rank=0&world_
↪size=1
INFO:adaptdl.torch:torch.distributed initialized
INFO:adaptdl.torch.epoch:starting at epoch 0
Train Epoch: 0 [0/60000 (0%)]    Loss: 2.318445
Train Epoch: 0 [640/60000 (1%)]  Loss: 1.647522
...
...
...
Train Epoch: 7 [30080/60000 (50%)]      Loss: 0.009690
Train Epoch: 7 [30720/60000 (51%)]      Loss: 0.010559
^CINFO:adaptdl._signal:Got SIGINT, exiting gracefully... Send signal again to force exit.
INFO:adaptdl._signal:Got SIGINT, exiting gracefully... Send signal again to force exit.
```

Training can be resumed by running the script with the same checkpoint path.

```
$ ADAPTDL_CHECKPOINT_PATH=mnist-checkpoint python3 mnist.py
```

Output:

```
WARNING:adaptdl.reducer:Could not connect to root, trying again...
INFO:adaptdl.reducer:Master waiting for connections on 0
INFO:adaptdl.reducer:rank 0 connecting to 0.0.0.0 on port 45371
INFO:adaptdl.torch:Initializing torch.distributed using tcp://0.0.0.0:23678?rank=0&world_
↪size=1
INFO:adaptdl.torch:torch.distributed initialized
INFO:adaptdl.torch.epoch:starting at epoch 7
```

(continues on next page)

(continued from previous page)

```
Train Epoch: 7 [0/60000 (0%)]   Loss: 0.070648
Train Epoch: 7 [640/60000 (2%)] Loss: 0.068212
...
...
...
Train Epoch: 13 [58880/60000 (98%)]     Loss: 0.081517
Train Epoch: 13 [59520/60000 (99%)]     Loss: 0.006973

Test set: Average loss: 0.0281, Accuracy: 9913/10000 (99%)
```

Whenever possible, it's recommended to test the training code locally in this way before submitting it to an AdaptDL-scheduled cluster.

2.4.3 Distributed Training

Training code that uses AdaptDL APIs can also be run on a distributed cluster, without requiring the AdaptDL scheduler. In this setting, the training job will run using the same number of replicas until it finishes, or until a checkpoint is manually triggered. Although the number of replicas is fixed, standalone distributed training can still benefit from the automatic batch size and learning rate scaling offered by AdaptDL.

The following environment variables need to be set for every replica:

- `ADAPTDL_MASTER_ADDR`: network address of the node running the rank 0 replica, must be accessible from all other replicas.
- `ADAPTDL_MASTER_PORT`: available port on the node running the rank 0 replica, must be accessible from all other replicas.
- `ADAPTDL_NUM_REPLICAS`: total number of replicas.
- `ADAPTDL_REPLICA_RANK`: integer rank from 0 .. K-1 for each replica, where K is the total number of replicas.

Assuming two nodes with hostnames `node-0` and `node-1`, on `node-0`:

```
$ ADAPTDL_MASTER_ADDR=node-0 ADAPTDL_MASTER_PORT=47000 \
  ADAPTDL_NUM_REPLICAS=2 ADAPTDL_REPLICA_RANK=0 python3 mnist.py
```

And on `node-1`:

```
$ ADAPTDL_MASTER_ADDR=node-0 ADAPTDL_MASTER_PORT=47000 \
  ADAPTDL_NUM_REPLICAS=2 ADAPTDL_REPLICA_RANK=1 python3 mnist.py
```

A checkpoint can be triggered by sending an interrupt to any of the replicas. The replica with rank 0 will save the checkpoint to the path specified by the `ADAPTDL_CHECKPOINT_PATH` environment variable, and then all replicas will exit.

Training can be resumed from the checkpoint using any number of replicas. However, each replica will need to be able to access the saved checkpoint. This means the checkpoint should be saved to a shared distributed filesystem such as NFS, or be manually copied to each node before resuming training.

2.5 Using the Adaptive Tune Trial Scheduler

This is a tutorial on using AdaptDL as a Tune Trial Scheduler. We'll go through an example that uses HyperOpt to tune hyperparameters like the learning rate, momentum and initial batch size. The batch size and number of replicas will be automatically adjusted by AdaptDL throughout the lifetimes of the trials so as to efficiently and fairly share the resources of the Ray cluster.

We'll be relying on the PyTorch *DistributedTrainable* Tune API documented here.

2.5.1 Setup

1. Install the required packages *pip install -U adaptdl-ray hyperopt*
2. Start the ray cluster.

2.5.2 Incorporating the AdaptDL API

In order to make use of the Adaptive functionality, we will need to change the trainable to include the AdaptDL API.

We don't change the model definition and test and train functions

```
class ConvNet(nn.Module):
    def __init__(self):
        super(ConvNet, self).__init__()
        # In this example, we don't change the model architecture
        # due to simplicity.
        self.conv1 = nn.Conv2d(1, 3, kernel_size=3)
        self.fc = nn.Linear(192, 10)

    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), 3))
        x = x.view(-1, 192)
        x = self.fc(x)
        return F.log_softmax(x, dim=1)


# Change these values if you want the training to run quicker or slower.
EPOCH_SIZE = 512
TEST_SIZE = 256


def train(model, optimizer, train_loader):
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        # We set this just for the example to run quickly.
        if batch_idx * len(data) > EPOCH_SIZE:
            return
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        output = model(data)
        loss = F.nll_loss(output, target)
```

(continues on next page)

(continued from previous page)

```
        loss.backward()
        optimizer.step()


def test(model, data_loader):
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for batch_idx, (data, target) in enumerate(data_loader):
            # We set this just for the example to run quickly.
            if batch_idx * len(data) > TEST_SIZE:
                break
            data, target = data.to(device), target.to(device)
            outputs = model(data)
            _, predicted = torch.max(outputs.data, 1)
            total += target.size(0)
            correct += (predicted == target).sum().item()
        else:
            return 0
    return correct / total
```

The trainable function *train_mnist* needs to change though.

```
+import adaptdl.torch as adl
+
 def train_mnist(config: Dict, checkpoint_dir: Optional[str] = None):
     # Data Setup
     mnist_transforms = transforms.Compose(
         [transforms.ToTensor(),
          transforms.Normalize((0.1307, ), (0.3081, ))])

-    train_loader = DataLoader(datasets.MNIST("~/data",
+    train_loader = adl.AdaptiveDataLoader(datasets.MNIST("~/data",
         train=True, download=True, transform=mnist_transforms),
         batch_size=64,
         shuffle=True)

-    test_loader = DataLoader(
+    test_loader = adl.AdaptiveDataLoader(
         datasets.MNIST("~/data", train=False, transform=mnist_transforms),
         batch_size=64,
         shuffle=True)
@@ -21,8 +23,9 @@

     model = ConvNet()
     model.to(device)
-    model = DistributedDataParallel(model)
+    model = adl.AdaptiveDataParallel(model, optimizer)

-    for i in range(10):
```

(continues on next page)

(continued from previous page)

```
+    for epoch in adl.remaining_epochs_until(config.get("epochs", 10)):
         train(model, optimizer, train_loader)
         acc = test(model, test_loader)
         # Send the current training result back to Tune
```

The changes essentially make the dataloaders and model elastic and restart-safe thus adding AdaptDL functionality. Now we need to use the the AdaptDL trial scheduler which can actually make decisions based on available cluster resources and trial characteristics.

We first create a trainable (class) and a search space for HyperOpt. We call *tune.run* and pass in *AdaptDLScheduler* as the trial scheduler for all the trials. The *AdaptDLScheduler* will first try to use GPUs on the Ray cluster. If it finds none, it will use CPUs to run the trials.

Full example can be found at hyperopt_example.py.

To run the example, simply run it from command line

```
$ python3 hyperopt_example.py

...
== Status ==
 Current time: 2021-10-26 12:55:14 (running for 00:04:55.09)
 Memory usage on this node: 2.1/31.2 GiB
 Using AdaptDL scheduling algorithm.
 Resources requested: 0/8 CPUs, 0/0 GPUs, 0.0/18.43 GiB heap, 0.0/9.21 GiB objects
 Result logdir: /tmp
 Number of trials: 4/4 (4 TERMINATED)
 +-------------------------------+------------+---------------------+----------+--------
↪+------------------+
 | Trial name                    | status     | loc                 |      acc |   iter␣
↪|   total time (s) |
 |-------------------------------+------------+---------------------+----------+--------
↪+------------------|
 | AdaptDLTrainable_7_2_cd64740f | TERMINATED | 192.168.1.196:20687 | 0.957576 |    102␣
↪|          92.0071 |
 | AdaptDLTrainable_1_2_cd64740e | TERMINATED | 192.168.1.196:21408 | 0.930804 |    102␣
↪|         115.433  |
 | AdaptDLTrainable_1_2_cd647410 | TERMINATED | 192.168.1.196:21407 | 0.953125 |    102␣
↪|          75.8803 |
 | AdaptDLTrainable_5_2_ceeea272 | TERMINATED | 192.168.1.196:21612 | 0.872396 |    102␣
↪|         102.775  |
 +-------------------------------+------------+---------------------+----------+--------
↪+------------------+

 Best trial config: {'bs': 960, 'epochs': 100, 'lr': 0.010874198064009714, 'momentum': 0.
↪5627724615056127}
 Best trial mean_accuracy: 0.8723958333333334
```

The trial names in the end can be interpreted as *AdaptDLTrainable_$num_replicas_$num_restarts_$trial_id*. Trials can expand or shrink based on the decisions of the AdaptDL optimizer and this gets reflected through their names.

2.6 Adaptdl on Ray AWS

The executable `adaptdl_on_ray_aws` allows you to run an AdaptDL job on an AWS-Ray cluster. The intention of this module is to allow you to get AdaptDL jobs working quickly, without the need to deploy kubernetes, and you to use Ray's cluster rescaling with AdaptDL's worker autoscaling.

2.6.1 Usage

Modifications to your training code

In order for your code to run, your training code will need to use AdaptDL. Please follow this tutorial for more information.

Your code should follow these properties:

- You do not need to make any calls to Ray
- Your code will also need to be able to run from the command line
- The code can take can take command line arguments via `sys.argv` and `argparse`
- The code is run as `__main__`
- Local imports from the same directory as `code.py` are supported

Deploying a Ray cluster on AWS EC2

You will need a ray cluster already deployed. Please see these instructions and tutorial for configuring and launching a ray cluster.

When creating the cluster, you will need configure the following:

- A dockerfile with these installed: * The pip requirements in *ray/aws/requirements.txt* * A working installation of pytorch-gpu * Whatever other pip dependencies you may require
- Sufficient disk space for the above docker image, and whatever disk space you may need to run your code
- Some maximum number of worker nodes

See *examples/cluster_config.yaml* for an example of the cluster.

To ensure that the ndoes have enough space for Docker to use, you will need to include something like the following *BlockDeviceMapping* configuration in all of the nodes:

```
node_config:
  InstanceType: <your instance type>
  BlockDeviceMappings:
    - DeviceName: /dev/sda1
      Ebs:
        VolumeSize: 100 #  Feel free to change this value
```

Just creating the EBS volume will not make it available for docker. You will also need to format and mount the volume as part of the initialization commands:

```
initialization_commands:
  - sudo usermod -aG docker $USER
  - sudo mkdir /docker_volume
```

(continues on next page)

(continued from previous page)

```
  - sudo mkfs -t xfs /dev/nvme1n1
  - sudo mount /dev/nmve1n1 /docker_volume -w
  - sudo dockerd --data-root /docker_volume &
```

If you find that your code does not have enough access to disk space, you can also mount an external volume (as provisioned above) to the runtime containers via:

```yaml
docker:
  image: <your-image-name>
  run_options:
  - -v '/<your-external-volume>:/<the-path-in-the-container>
```

Make sure that the permissions for the external volume are set properly.

Running your code

Once the cluster has been deployed, you will need the address and port of the cluster head. Generally, this will be of the form `<head-node-ip>:10001`. Make sure that you have access to that port via the AWS subnet and inbound rules.

On your local machine, make sure to install the pip package for `adaptdl_ray`. This package includes the launcher script, and will generally install it in `/usr/local/bin/adaptdl_on_ray_aws`.

If you have some AdaptDL training code runnable at `code.py` via `python3 code.py <command-line-args>`, you can run the training code on Ray via

`./usr/local/bin/adaptdl_on_ray_aws -u "ray://head-node-ip:10001" -f code.py -m <maximum-number-of-workers> --cpus <cpus-per-worker> --gpus <gpus-per-worker> -- <command-line-args>`

If your local version of Python does not match the cluster's, Ray will not work. In this case, one option is to run the command within a Docker container. Be sure to mount your code directory in the container, e.g. via `-v`.

Retrieving your trained model

In order to retrieve the result of your training code, you will need to manually save it to some external store. For example, you could write it to S3, or you could mount an EFS store to the cluster, and write it to that. See the Advanced Usage for more details on using EFS.

2.6.2 Example

To run the example code found in `examples/pytorch-cifar/main.py`, do the following:

1. Install the AWS CLI and authenticate.
2. Inside the `example/ray/aws` directory, run `ray up -y cluster.yaml -v`. Note: running this step will create an AWS EC2 cluster, which will cost money
3. Keep track of the ip and port `ray up` returns.
4. Install Docker or the exact Python version used by your cluster. You can determine the python version by running `ray attach <cluster-config-file`, and then running Python.

5. Still inside `example/ray/aws`, run `docker run <docker version> python3 adaptdl_ray.py -f main.py -m 3 -u ray://<ip>:<port> -- -autoscale-bsz`. If you are using Python. then install the requirements in `ray/aws/requirements.txt` and run `./usr/local/bin/adaptdl_on_ray_aws -f main.py -m 3 -u ray://<ip>:<port> -- -autoscale-bsz`.

2.6.3 Advanced Usage

Spot instances

AdaptDL on Ray AWS supports spot instances for the ray cluster. Each of the workers will listen to the for the spot instance termination notification. If a node is scheduled to be deleted, a checkpoint will be taken and the job will be rescaled to exclude and find a replacement for that node.

Dealing with Large Datasets

As workers can be rescheduled to fresh nodes, downloading large datasets to each worker can be expensive. For example, if a worker downloads data for 20 minutes when it is scheduled to a new node, then the other workers will be idle for 20 minutes as well, even if they already have the data. This is exacerbated if the autoscaler gradually increases the number of workers.

There are several options to deal with this:

1. Use Amazon S3 with an S3Dataset.
2. Use EFS to share the data between the nodes

Using S3

One difference with using an S3 Dataset in the Ray cluster versus on your local machine is ensuring that all of the nodes have the proper permissions. Please follow these instructions

Using EFS

EFS allows you to use a distributed filesystem with your EC2 cluster. To begin, you will need to create an EFS instance. Once that is done, use the `setup_commands` listed here to attach your EFS instance to the nodes.

Please note that using EFS will incur additional costs.

Imports

If you need Python modules that are local to your machine but not located in the same directory as your main script, set `--working-dir` to a directory that contains the main script and all the Python modules. The argument to `-f/--file` should then be the path to the main script relative to the argument to `--working-dir`.

Timeouts

There are two conditions where the job controller will need to wait for some reponse. In order to prevent a lack of response from permamently stopping the job, there are timeouts.

First, when the workers are terminated in order to perform a rescaling, the controller will wait to recieve a checkpoint object of the training state from worker 0. If the controller does not receive a checkpoint by the amount of time specified in `--checkpoint-timeout` (default 120 seconds), then the controller will use a previous version of the checkpoint, or restart from 0, if a previous checkpoint does not exist. Note that spot instances have around a 2 minute warning for termination.

Second, when the cluster is rescaling to more workers, it can take some time for the new workers to be ready. In addition, spot instances requests may never be fulfilled if their bid price is too low. The controller therefore waits for some time, up to the amount specified in `--cluster-rescale-timeout` (default 60), for the new nodes to be provisioned and ready. If the nodes are not ready by that time, it schedules up to the maximum supported by the current cluster. Please note that the new nodes need to download the Docker image set in the cluster config. As these images can be large, it may take 5-10 minutes for new nodes to be available.

2.7 adaptdl package

2.7.1 Subpackages

adaptdl.torch package

`class adaptdl.torch.Accumulator(*args, **kwargs)`

Bases: `collections.abc.MutableMapping`

This class helps aggregate simple statistics across all replicas in the current job, and across any number of checkpoint-restarts. Can be used to compute metrics like loss and accuracy, synchronized across each replica.

Accumulators imitate python dictionaries, but with a few key differences described below. Primarily, its usage and behavior depend on whether it is set to *accumulation mode* or to *synchronized mode*.

1. **Accumulation mode:** the accumulator is being updated on all replicas. Operations like `accum["key"] += val` or `accum.update(key=val)` will aggregate the updates locally on each replica, which are lazily synchronized in the background (either upon a checkpoint or a switch to synchronized mode). Each replica may make different updates, which are summed together when synchronized. While accumulation mode is enabled, all read operations on the accumulator will behave as if they were performed on an empty `dict`, ie. `len(accum)` will always return `0`. By default, all accumulators are set to accumulation mode.
2. **Synchronized mode:** the accumulator contains the same data on every replica, and the application must ensure that all write operations are exactly the same across all replicas. While in synchronized mode, the accumulator may be used as if it were a native python `dict`, and all read/write operations are supported. *`Accumulator.synchronized()`* may be used to enter synchronized mode. Upon entering synchronized mode, the accumulator will automatically sum all updates from all replicas to ensure the same data is available to each replica.

Using accumulators, many training/validation metrics can be computed easily and correctly in an elastic distributed setting. For example, a simple validation step which calculates a loss and accuracy can be implemented as follows:

```
accum = Accumulator()  # New accumulator starts in accumulation mode.

for epoch in remaining_epochs_until(60):
```

(continues on next page)

(continued from previous page)

```
for batch in validloader:
    ...
    accum["loss_sum"] += <loss summed within the batch>
    accum["correct"] += <number of correct predictions>
    accum["total"] += <total number of samples in the batch>

with accum.synchronized():  # Enter synchronized mode.
    accum["loss_avg"] = accum["loss_sum"] / accum["total"]
    accum["accuracy"] = accum["correct"] / accum["total"]
    print("Loss: {}, Accuracy: {}".format(
          accum["loss_avg"], accum["accuracy"]))
    accum.clear()
# Back to accumulation mode.
```

Parameters

- **args** – Positional arguments same as `dict`.
- **kwargs** – Keyword arguments same as `dict`.

`__iadd__`(*other*)

Supports the += operation, e.g. `accum += {key1: val1, key2: val2}`. Behaves the same way as `accum.update({key1: val1, key2: val2})`.

Parameters **other** – Mapping object or an iterable of key-update pairs.

`__isub__`(*other*)

Supports the -= operation, e.g. `accum -= {key1: val1, key2: val2}`. Behaves the same way as `accum.subtract({key1: val1, key2: val2})`.

Parameters **other** – Mapping object or an iterable of key-update pairs.

`__getitem__`(*key*)

Supports indexing, e.g. `val = accum[key]` and `accum[key] += 1`. The former (read access) should only be used when the accumulator is in synchronized mode.

Parameters **other** – Key used to access a value in the accumulator.

`subtract`(**args*, ***kwargs*)

Apply a collection of key-update pairs. Unlike `Accumulator.update()`, this method *subtracts* the updates from the accumulated values.

Parameters

- **args** – Positional arguments same as `Accumulator.update()`.
- **kwargs** – Keyword arguments same as `Accumulator.update()`.

`synchronized`()

A context manager which can be used to define the code to execute in *synchronized* mode. Within the context manager, any code can interact with this accumulator as if it were a regular Python `dict`. The application must ensure that whatever operations performed within this context block are the same across all replicas.

> **Warning:** Entering this context manager is a distributed synchronization point! Please ensure that all replicas enter this context manager at the same point in their code.

`update`(**args*, ***kwargs*)

Apply a collection of key-update pairs. Unlike `dict.update`, this method *additively* applies the updates to the accumulated values.

Parameters

- **args** – Positional arguments same as `dict.update`. Can be a mapping object or an iterable of key-update pairs.
- **kwargs** – Keyword arguments same as `dict.update`. Each keyword is the string key corresponding to the provided update.

class `adaptdl.torch.`**`AdaptiveDataLoader`**(*dataset*, *batch_size=1*, *shuffle=False*, ***kwargs*)

Bases: `torch.utils.data.dataloader.DataLoader`, *`adaptdl.torch.data.AdaptiveDataLoaderMixin`*

This class is a PyTorch DataLoader that also supports adaptive batch sizes and checkpoint-restart elasticity. Applications can typically use objects of this class as direct replacements for PyTorch DataLoaders. However, some notable differences are:

1. The `batch_size` argument defines the target total batch size across all replicas, rather than the local batch size on each replica.
2. Custom `sampler` and `batch_sampler` are not supported.
3. Iterating through the dataloader is only allowed from within an epoch loop (see *`adaptdl.torch.epoch`*), and only one dataloader loop is allowed at any given time.

Parameters

- **dataset** (*`torch.util.data.Dataset`*) – Dataset from which to load the data.
- **batch_size** (*`int`*) – The target total batch size across all replicas. The actual total batch size may be different due to rounding (each replica must have the same local batch size), or being scaled up using adaptive batch sizes.
- **shuffle** (*`bool`*) – Whether the data is reshuffled at every epoch.
- ****kwargs** – Keyword arguments passed to `torch.util.data.Dataloader`.

Raises **`ValueError`** – If `sampler` or `batch_sampler` are not `None`.

`__iter__`()

Iterate over batches of data. When adaptive batch size is disabled, stops after the entire dataset has been processed once in total by all replicas. This means if there are K replicas, then this method will iterate over ~1/K of the dataset. When adaptive batch size is enabled, stops after making enough statistical progress roughly equivalent to one pass over the dataset with non-adaptive batch size. In this case, the dataset may be processed more than once.

A checkpoint-restart may be triggered in-between each batch. In this case, the current iteration state will be saved and restored after the restart, and continue where it left off.

class `adaptdl.torch.`**`AdaptiveDataParallel`**(*model*, *optimizer*, *lr_scheduler=None*, *mp_scaler=None*, *scaling_rule: Optional[*adaptdl.torch.scaling_rules.ScalingRuleBase*] = None*, *name='adaptdl-dataparallel'*, ***kwargs*)

Bases: `torch.nn.parallel.distributed.DistributedDataParallel`

This class extends PyTorch DistributedDataParallel with support for adaptive batch sizes and checkpoint-restart elasticity. It automatically saves the given model, optimizer, and (optionally) LR scheduler whenever a checkpoint is triggered, and restores their states after restart. The optimizer is automatically patched with the chosen scaling rule.

Parameters

- **model** (*torch.nn.Module*) – Model to be distributed.
- **optimizer** (*torch.optim.Optimizer*) – Optimizer used to update the given
- **parameters** (*model's*) –
- **of** (*will be patched using subclass*) –

:param *adaptdl.torch.scaling_rules.ScalingRuleBase*.: :param scaling_rule: Scaling rule used to :type scaling_rule: ScalingRuleBase :param patch the given optimizer: :param default to AdaScale.: :param lr_scheduler: LR scheduler used :type lr_scheduler: torch.optim.lr_scheduler._LRScheduler :param to anneal the learning rate for the given optimizer.: :param name: Unique name for each instance of this class, needed only :type name: string :param if multiple instances exist.:

forward(**args*, ***kwargs*)

property gain

Current estimate of the AdaScale gain (r_t) value.

to_tensorboard(*writer*, *global_step*, *tag_prefix=''*)

Output some useful metrics to TensorBoard.

Parameters

- **writer** (*torch.utils.tensorboard.SummaryWriter*) – `SummaryWriter` object to output metrics to.
- **global_step** (*int*) – Global step value to record.
- **tag_prefix** (*str*) – Prefix added to each metric's tag.

training: bool

zero_grad(**args*, ***kwargs*)

Sets gradients of all model parameters to zero.

class adaptdl.torch.**ElasticSampler**(*dataset*, *shuffle=True*)

Bases: `torch.utils.data.sampler.Sampler`

A PyTorch Sampler which partitions data samples across multiple replicas, and supports deterministic continuing across checkpoint-restarts. Shuffling is deterministic for each epoch, and *ElasticSampler.set_epoch()* should be invoked to obtain different orderings in different epochs.

Parameters

- **dataset** (*torch.util.data.Dataset*) – The dataset to sample from.
- **shuffle** (*bool*) – Whether the data samples should be shuffled.

__iter__()

Iterate through the samples in the dataset, in the order defined for a set epoch, starting at a set index. Produces only the indices for the local replica.

Returns: Iterator over data sample indices.

`__len__`()

The total number of samples to be iterated through, starting at the set index, for the local replica.

Returns (int): Number of samples.

`set_epoch`(*epoch*, *index=0*)

Set the epoch to derive samples from. Optional argument `index` can be specified to start sampling from a particular index, e.g. after a checkpoint-restart.

Parameters

- **epoch** (*int*) – The epoch to sample from.
- **index** (*int*) – The index to start sampling from.

`adaptdl.torch.`**`current_dataloader`**()

Reference to the data loader currently being iterated.

Returns (AdaptiveDataLoaderHelper): Current data loader.

`adaptdl.torch.`**`current_epoch`**()

Get the current epoch while iterating with *`remaining_epochs_until()`*.

Returns The current epoch number if called from within a *`remaining_epochs_until()`* iteration, `None` otherwise.

Return type int or None

`adaptdl.torch.`**`finished_epochs`**()

Get the number of epochs finished using *`remaining_epochs_until()`*.

Returns The number of finished epochs. Equal to *`current_epoch()`* if called from within a *`remaining_epochs_until()`* iteration.

Return type int

`adaptdl.torch.`**`init_process_group`**(*backend*, *init_method=None*, *world_size=None*, *rank=None*)

Initializes the default distributed process group and the AdaptDL collectives module.

Parameters

- **backend** (*str or Backend*) – The backend to use. Use "nccl" for multi-GPU training else "gloo".
- **init_method** (*str, optional*) – URL specifying how to initialize the process group.
- **world_size** (*int, optional*) – Number of processes participating in the job
- **rank** (*int, optional*) – Rank of the current process (it should be a number between 0 and `world_size`-1).

If init_method, world_size and rank is NOT provided, typically in the Kubernetes environment, AdaptDL will try to infer them through environment variables ADAPTDL_MASTER_ADDR, ADAPTDL_NUM_REPLICAS and ADAPTDL_REPLICA_RANK respectively.

`adaptdl.torch.`**`remaining_epochs_until`**(*epoch*)

Iterate over epochs in a way that is consistent with checkpoint-restarts. For example:

```
for epoch in remaining_epochs_until(30):
    print(current_epoch())  # Should print 0 through 29

for epoch in remaining_epochs_until(60):
    print(current_epoch())  # Should print 30 through 59
```

If a checkpoint-restart happens during an epoch, all previous epochs will be skipped after the program restarts.

Parameters **epoch** (*int*) – The epoch number to end at (exclusively).

Raises **RuntimeError** – If invoked before a previous epoch loop has ended.

Submodules

adaptdl.torch.accumulator module

class adaptdl.torch.accumulator.**Accumulator**(**args*, ***kwargs*)

Bases: `collections.abc.MutableMapping`

This class helps aggregate simple statistics across all replicas in the current job, and across any number of checkpoint-restarts. Can be used to compute metrics like loss and accuracy, synchronized across each replica.

Accumulators imitate python dictionaries, but with a few key differences described below. Primarily, its usage and behavior depend on whether it is set to *accumulation mode* or to *synchronized mode*.

1. **Accumulation mode:** the accumulator is being updated on all replicas. Operations like `accum["key"] += val` or `accum.update(key=val)` will aggregate the updates locally on each replica, which are lazily synchronized in the background (either upon a checkpoint or a switch to synchronized mode). Each replica may make different updates, which are summed together when synchronized. While accumulation mode is enabled, all read operations on the accumulator will behave as if they were performed on an empty `dict`, ie. `len(accum)` will always return `0`. By default, all accumulators are set to accumulation mode.
2. **Synchronized mode:** the accumulator contains the same data on every replica, and the application must ensure that all write operations are exactly the same across all replicas. While in synchronized mode, the accumulator may be used as if it were a native python `dict`, and all read/write operations are supported. *Accumulator.synchronized()* may be used to enter synchronized mode. Upon entering synchronized mode, the accumulator will automatically sum all updates from all replicas to ensure the same data is available to each replica.

Using accumulators, many training/validation metrics can be computed easily and correctly in an elastic distributed setting. For example, a simple validation step which calculates a loss and accuracy can be implemented as follows:

```
accum = Accumulator()  # New accumulator starts in accumulation mode.

for epoch in remaining_epochs_until(60):

    for batch in validloader:
        ...
        accum["loss_sum"] += <loss summed within the batch>
        accum["correct"] += <number of correct predictions>
        accum["total"] += <total number of samples in the batch>

    with accum.synchronized():  # Enter synchronized mode.
        accum["loss_avg"] = accum["loss_sum"] / accum["total"]
        accum["accuracy"] = accum["correct"] / accum["total"]
        print("Loss: {}, Accuracy: {}".format(
              accum["loss_avg"], accum["accuracy"]))
        accum.clear()
    # Back to accumulation mode.
```

Parameters

- **args** – Positional arguments same as `dict`.
- **kwargs** – Keyword arguments same as `dict`.

`__iadd__`(*other*)

Supports the += operation, e.g. `accum += {key1: val1, key2: val2}`. Behaves the same way as `accum.update({key1: val1, key2: val2})`.

Parameters **other** – Mapping object or an iterable of key-update pairs.

`__isub__`(*other*)

Supports the -= operation, e.g. `accum -= {key1: val1, key2: val2}`. Behaves the same way as `accum.subtract({key1: val1, key2: val2})`.

Parameters **other** – Mapping object or an iterable of key-update pairs.

`__getitem__`(*key*)

Supports indexing, e.g. `val = accum[key]` and `accum[key] += 1`. The former (read access) should only be used when the accumulator is in synchronized mode.

Parameters **other** – Key used to access a value in the accumulator.

`subtract`(**args*, ***kwargs*)

Apply a collection of key-update pairs. Unlike `Accumulator.update()`, this method *subtracts* the updates from the accumulated values.

Parameters

- **args** – Positional arguments same as `Accumulator.update()`.
- **kwargs** – Keyword arguments same as `Accumulator.update()`.

`synchronized`()

A context manager which can be used to define the code to execute in *synchronized* mode. Within the context manager, any code can interact with this accumulator as if it were a regular Python `dict`. The application must ensure that whatever operations performed within this context block are the same across all replicas.

> **Warning:** Entering this context manager is a distributed synchronization point! Please ensure that all replicas enter this context manager at the same point in their code.

`update`(**args*, ***kwargs*)

Apply a collection of key-update pairs. Unlike `dict.update`, this method *additively* applies the updates to the accumulated values.

Parameters

- **args** – Positional arguments same as `dict.update`. Can be a mapping object or an iterable of key-update pairs.
- **kwargs** – Keyword arguments same as `dict.update`. Each keyword is the string key corresponding to the provided update.

adaptdl.torch.data module

class adaptdl.torch.data.**AdaptiveDataLoader**(*dataset*, *batch_size=1*, *shuffle=False*, ***kwargs*)

Bases: `torch.utils.data.dataloader.DataLoader`, `adaptdl.torch.data.AdaptiveDataLoaderMixin`

This class is a PyTorch DataLoader that also supports adaptive batch sizes and checkpoint-restart elasticity. Applications can typically use objects of this class as direct replacements for PyTorch DataLoaders. However, some notable differences are:

1. The `batch_size` argument defines the target total batch size across all replicas, rather than the local batch size on each replica.
2. Custom `sampler` and `batch_sampler` are not supported.
3. Iterating through the dataloader is only allowed from within an epoch loop (see `adaptdl.torch.epoch`), and only one dataloader loop is allowed at any given time.

Parameters

- **dataset** (*torch.util.data.Dataset*) – Dataset from which to load the data.
- **batch_size** (*int*) – The target total batch size across all replicas. The actual total batch size may be different due to rounding (each replica must have the same local batch size), or being scaled up using adaptive batch sizes.
- **shuffle** (*bool*) – Whether the data is reshuffled at every epoch.
- ****kwargs** – Keyword arguments passed to `torch.util.data.Dataloader`.

Raises ValueError – If `sampler` or `batch_sampler` are not `None`.

__iter__()

Iterate over batches of data. When adaptive batch size is disabled, stops after the entire dataset has been processed once in total by all replicas. This means if there are K replicas, then this method will iterate over ~1/K of the dataset. When adaptive batch size is enabled, stops after making enough statistical progress roughly equivalent to one pass over the dataset with non-adaptive batch size. In this case, the dataset may be processed more than once.

A checkpoint-restart may be triggered in-between each batch. In this case, the current iteration state will be saved and restored after the restart, and continue where it left off.

class adaptdl.torch.data.**AdaptiveDataLoaderHelper**(*batch_size=1*)

Bases: `object`

This class provides fine-grained control over adaptive training loops. It can be used for building more user-friendly custom data loaders, such as `AdaptiveDataLoader`.

Parameters batch_size (*int*) – The target total batch size across all replicas. The actual total batch size may be different due to rounding (each replica must have the same local batch size), or being scaled up using adaptive batch sizes.

property accumulation_steps

The number of batches returned by the dataloader before a step is taken.

autoscale_batch_size(*max_batch_size*, *local_bsz_bounds=None*, *gradient_accumulation=False*)

Enables adaptive batch size. Should be invoked once after the data loader object is created.

Parameters

- **max_batch_size** (*int*) – Maximum total batch size allowed.

- **local_bsz_bounds** (*tuple*) – A pair of (min_local_bsz, max_local_bsz), the min and max local batch sizes allowed on each replica.

Raises **ValueError** – If any of the provided batch size bounds are invalid.

`context()`

All iterators should be iterated under this context. It ensures proper cleanup of elastic context at the end of each epoch.

property current_batch_size

property current_index

The total number of data samples processed so far in the current loop. Includes the data processed by all replicas. `None` if this data loader is not currently being iterated.

property current_local_bsz

The current logical local batch size used by the dataloader. The batch size returned by the dataloader may be smaller if gradient accumulation is used

property end_index

(Optional) Can be used to track the end index of dataset across restarts.

`is_accum_step()`

Whether the current step's gradient will be accumulated.

`is_optim_step()`

Whether the optimizer step will be invoked in this step.

property local_bsz_bounds

The local batch size bounds on each replica. A pair of integers, (min_local_bsz, max_local_bsz).

property max_batch_size

The maximum total batch size allowed for adaptive batch size. `None` if adaptive batch size is disabled.

`profile(commit)`

Every iteration of every epoch should be profiled under this context. Note that, custom DataLoader writers should make sure that it gets called equal number of times on each replica.

Parameters **commit** (*bool*) – Whether to commit the profiled results.

`skipdone()`

Should be called just after entering the *_elastic* context to make sure that the dataloader loop is not replayed if has already finished before a restart.

`to_tensorboard(writer, global_step, tag_prefix='')`

Output some useful metrics to TensorBoard.

Parameters

- **writer** (*torch.utils.tensorboard.SummaryWriter*) – `SummaryWriter` object to output metrics to.
- **global_step** (*int*) – Global step value to record.
- **tag_prefix** (*str*) – Prefix added to each metric's tag.

`train()`

Set this data loader to be the one used for training. Only one data loader may be used for training.

property training

class adaptdl.torch.data.**AdaptiveDataLoaderMixin**(*batch_size*)

Bases: object

This class provides elastic functionality to any custom DataLoader which inherits it. It defines a member _elastic of type *AdaptiveDataLoaderHelper* which has useful methods and members to implement restart-safe, elastic DataLoaders. It also exposes public methods which can be used inside training loops directly from *AdaptiveDataLoader*.

property accumulation_steps

The number of batches returned by the dataloader before a step is taken.

autoscale_batch_size(*max_batch_size*, *local_bsz_bounds=None*, *gradient_accumulation=False*)

property current_batch_size

property current_local_bsz

to_tensorboard(*writer*, *global_step*, *tag_prefix=''*)

Output some useful metrics to TensorBoard.

Parameters

- **writer** (*torch.utils.tensorboard.SummaryWriter*) – SummaryWriter object to output metrics to.
- **global_step** (*int*) – Global step value to record.
- **tag_prefix** (*str*) – Prefix added to each metric's tag.

property training

class adaptdl.torch.data.**ElasticSampler**(*dataset*, *shuffle=True*)

Bases: torch.utils.data.sampler.Sampler

A PyTorch Sampler which partitions data samples across multiple replicas, and supports deterministic continuing across checkpoint-restarts. Shuffling is deterministic for each epoch, and *ElasticSampler.set_epoch()* should be invoked to obtain different orderings in different epochs.

Parameters

- **dataset** (*torch.util.data.Dataset*) – The dataset to sample from.
- **shuffle** (*bool*) – Whether the data samples should be shuffled.

__iter__()

Iterate through the samples in the dataset, in the order defined for a set epoch, starting at a set index. Produces only the indices for the local replica.

Returns: Iterator over data sample indices.

__len__()

The total number of samples to be iterated through, starting at the set index, for the local replica.

Returns (int): Number of samples.

set_epoch(*epoch*, *index=0*)

Set the epoch to derive samples from. Optional argument index can be specified to start sampling from a particular index, e.g. after a checkpoint-restart.

Parameters

- **epoch** (*int*) – The epoch to sample from.

- **index** (*int*) – The index to start sampling from.

adaptdl.torch.data.**current_dataloader**()

Reference to the data loader currently being iterated.

Returns (AdaptiveDataLoaderHelper): Current data loader.

adaptdl.torch.epoch module

This module provides tools for the top-level loop over epochs during training. AdaptDL expects the training program to be implemented as loop over several epochs, each containing a series of loops over datasets (e.g. one loop over the training set followed by one loop over the validation set). The program can be interrupted between every iteration of any dataset loop, trigger a checkpoint to be taken, and restarted using a different set of replicas.

Due to checkpoint-restarts, parts of the training program may be executed multiple times (e.g. once after each restart)! To avoid incorrect execution, ensure that your code is idempotent in the following locations:

1. Immediately before any epoch loop (using *remaining_epochs_until()*).
2. Immediately before any dataset loop (using *adaptdl.torch.data.AdaptiveDataLoader*).

Your code may be non-idempotent in other locations.

```
### IDEMPOTENT CODE ONLY ###

for epoch in remaining_epochs_until(30):

    ### IDEMPOTENT CODE ONLY ###

    for batch in train_loader:
        # ... any code ...

    ### IDEMPOTENT CODE ONLY ###

    for batch in valid_loader:
        # ... any code ...

    # ... any code ...

# ... any code ...

### END PROGRAM ###
```

For example, a common non-idempotent operation is learning-rate annealing:

```
for epoch in remaining_epochs_until(30):

    lr_scheduler.step()  # (A) WRONG!

    for batch in train_loader:
        # ...

    lr_scheduler.step()  # (B) WRONG!

    for batch in valid_loader:
```

(continues on next page)

(continued from previous page)

```
        # ...

    lr_scheduler.step()  # (C) OK!
```

Location (A) will be executed again after any checkpoint-restart during either the training or validation loop, resulting in the learning rate being annealed several times in one epoch! Similarly with location (B), if checkpoint-restart happens during the validation loop.

Location (C) results in the correct behavior, because (1) an epoch will not be repeated once it has finished, and (2) no checkpoint-restarts can occur between the learning rate annealing and the end of the epoch.

adaptdl.torch.epoch.**current_epoch**()

Get the current epoch while iterating with *remaining_epochs_until()*.

Returns The current epoch number if called from within a *remaining_epochs_until()* iteration, None otherwise.

Return type int or None

adaptdl.torch.epoch.**finished_epochs**()

Get the number of epochs finished using *remaining_epochs_until()*.

Returns The number of finished epochs. Equal to *current_epoch()* if called from within a *remaining_epochs_until()* iteration.

Return type int

adaptdl.torch.epoch.**remaining_epochs_until**(*epoch*)

Iterate over epochs in a way that is consistent with checkpoint-restarts. For example:

```
for epoch in remaining_epochs_until(30):
    print(current_epoch())  # Should print 0 through 29

for epoch in remaining_epochs_until(60):
    print(current_epoch())  # Should print 30 through 59
```

If a checkpoint-restart happens during an epoch, all previous epochs will be skipped after the program restarts.

Parameters **epoch** (*int*) – The epoch number to end at (exclusively).

Raises **RuntimeError** – If invoked before a previous epoch loop has ended.

adaptdl.torch.gradient_noise_scale module

class adaptdl.torch.gradient_noise_scale.**GradientNoiseScale**(*adp*, *optimizer*, *mp_scaler=None*, *num_replicas=None*, *accum_scale=None*)

Bases: object

This class tracks gradient related stats and takes care of gradient accumulation.

property accum_count

property accum_scale

gain(*scale*)

Current estimate of the GradientNoiseScale gain ratio.

Parameters **scale** (*float*) – The total scale to estimate the gain ratio for.

Returns (float): Estimate of gain ratio.

get_progress()

property raw_sqr_avg

property raw_var_avg

reset_accumulation()

reset accumulation calculations and gradients.

set_accum_scale(*accum_scale*)

set_progress(*progress*)

property should_zero_grad

sqr_avg()

Current estimate of the squared l2-norm of the true gradient (sigma squared).

Returns (float): Estimate of squared l2-norm.

var_avg()

Current estimate of the trace of the covariance of the true gradient (mu squared).

Returns (float): Estimate of trace of the covariance.

adaptdl.torch.iterator module

class adaptdl.torch.iterator.**AdaptiveBPTTIterator**(*dataset*, *batch_size*, *bptt_len*, ***kwargs*)

Bases: `torchtext.data.iterator.BPTTIterator`, `adaptdl.torch.data.AdaptiveDataLoaderMixin`

adaptdl.torch.parallel module

class adaptdl.torch.parallel.**AdaptiveDataParallel**(*model*, *optimizer*, *lr_scheduler=None*, *mp_scaler=None*, *scaling_rule: Optional[adaptdl.torch.scaling_rules.ScalingRuleBase] = None*, *name='adaptdl-dataparallel'*, ***kwargs*)

Bases: `torch.nn.parallel.distributed.DistributedDataParallel`

This class extends PyTorch DistributedDataParallel with support for adaptive batch sizes and checkpoint-restart elasticity. It automatically saves the given model, optimizer, and (optionally) LR scheduler whenever a checkpoint is triggered, and restores their states after restart. The optimizer is automatically patched with the chosen scaling rule.

Parameters

- **model** (*torch.nn.Module*) – Model to be distributed.
- **optimizer** (*torch.optim.Optimizer*) – Optimizer used to update the given
- **parameters** (*model's*) –
- **of** (*will be patched using subclass*) –

:param *adaptdl.torch.scaling_rules.ScalingRuleBase*.: :param scaling_rule: Scaling rule used to :type scaling_rule: ScalingRuleBase :param patch the given optimizer: :param default to AdaScale.: :param lr_scheduler: LR scheduler used :type lr_scheduler: torch.optim.lr_scheduler._LRScheduler :param to anneal the learning rate for the given optimizer.: :param name: Unique name for each instance of this class, needed only :type name: string :param if multiple instances exist.:

forward(**args*, ***kwargs*)

property gain

Current estimate of the AdaScale gain (r_t) value.

to_tensorboard(*writer*, *global_step*, *tag_prefix=''*)

Output some useful metrics to TensorBoard.

Parameters

- **writer** (*torch.utils.tensorboard.SummaryWriter*) – SummaryWriter object to output metrics to.
- **global_step** (*int*) – Global step value to record.
- **tag_prefix** (*str*) – Prefix added to each metric's tag.

training: bool

zero_grad(**args*, ***kwargs*)

Sets gradients of all model parameters to zero.

adaptdl.torch.scaling_rules module

class adaptdl.torch.scaling_rules.**AdaScale**

Bases: *adaptdl.torch.scaling_rules.ScalingRuleBase*

Implements the AdaScale algorithm for scaling the learning rate for distributed and large batch size training.

scale_lr(*scale*)

Calculate factors to be applied to lr for each parameter group.

class adaptdl.torch.scaling_rules.**LEGWScale**(*base_warmup_epochs*, *data_size*)

Bases: *adaptdl.torch.scaling_rules.ScalingRuleBase*

Implements the LEGWScale algorithm for scaling the learning rate.

Essentially, with LEGWScale, lr_factor is calculated based on training progress as follows: - when current_step < base_warmup_epoch * scale * steps_per_epoch:

*lr_factor = sqrt(scale) * progress_ratio* where `progress_ratio = current_step /

(scale * base_warmup_epochs * steps_per_epoch)`

- when current_step >= base_warmup_epoch * scale * steps_per_epoch: *lr_factor = sqrt(scale)*

In order to adapt LEGWScale to AdaptDL, *progress_ratio* is calculated differently as: *progress / (scale * base_warmup_epochs * steps_per_epoch)* where *progress* is the effective steps trained based on AdaptDL's estimation.

Argmuents: base_warmup_epochs: Base warmup epochs data_size: total number of samples in the dataset

scale_lr(*scale*)

class adaptdl.torch.scaling_rules.**LinearScale**

Bases: *adaptdl.torch.scaling_rules.ScalingRuleBase*

scale_lr(*scale*)

class adaptdl.torch.scaling_rules.**ScalingRuleBase**

Bases: object

Base class for scaling rules that has the ability to track gradient noise scale calculations. Its subclasses can be used in combination with adaptdl.torch.parallel.AdaptiveDataParallel and torch.optim.SGD.

```
optim = torch.optim.SGD(model, lr=0.001)
adascale = AdaScale()
model = AdaptiveDataParallel(model, optim, adascale)

for epoch in ...:
    for batch in ...:
        optim.zero_grad()
        loss = ...
        loss.backward()
        adascale.step()
```

initialize(*adp*, *optimizer*, *patch_optimizer=False*)

scale_lr(*scale*)

step(**args*, ***kwargs*)

Run one optimizer step. Essentially just invokes optimizer.step(*args, **kwargs) with a scaled learning rate.

Parameters

- **args** – Positional arguments passed to optimizer.step.
- **kwargs** – Keyword arguments passed to optimizer.step.

zero_grad(**args*, ***kwargs*)

class adaptdl.torch.scaling_rules.**SqrtScale**

Bases: *adaptdl.torch.scaling_rules.ScalingRuleBase*

scale_lr(*scale*)

2.7.2 Submodules

adaptdl.checkpoint module

This module provides functionality to Save and load arbitrary state as part of checkpoint-restart elasticity. The *State* class can be subclassed to define how to save/load any state to/from persistent storage, so it can be restored after the current job restarts and resumed from where it left off.

class adaptdl.checkpoint.**State**(*name*)

Bases: object

This class implements An arbitrary piece of state which can be saved and loaded as part of a checkpoint, and synchronized across all replicas. Should be sub-classed to define custom save, load, and sync logic.

`load`(*fileobj*)

This method should be overridden by subclasses to define how the state is loaded. Is invoked by *load_state* to load the state from persistent storage.

Parameters **fileobj** (*BinaryIO*) – A binary readable file object.

`save`(*fileobj*)

This method should be overridden by subclasses to define how the state is saved. Is invoked by *save_all_states* and *save_state* to save the state into persistent storage.

Parameters **fileobj** (*BinaryIO*) – A binary writable file object.

`sync`()

This method should be overridden by subclasses to define how the state is synchronized across replicas. This might be necessary to make sure the state is consistent before saving it to persistent storage. Is invoked by *save_state* before saving the state.

`adaptdl.checkpoint.load_state`(*state*)

Load the given *State* object from persistent storage. If the object was previously saved, then State.load will be invoked with a readable file object to load from.

Parameters **state** (State) – *State* object to load from persistent storage.

Returns *True* if state was previously saved and *State.load* was invoked, *False* otherwise.

`adaptdl.checkpoint.save_all_states`()

Invokes *save_state* on all *State* objects for which *State.skip* is True. This function can be used to trigger a global checkpoint and save every *State* in the current job.

`adaptdl.checkpoint.save_state`(*state*, *checkpoint_dir*, *sync=True*)

Saves a *State* object to persistent storage. First invokes *State.sync* on all replicas if *sync* is *True* (default), and then invokes *State.save* on the replica of rank 0 only. Note that we save state to a temporary folder first. Then, it will be renamed to the formal checkpoint folder after all states are saved.

Parameters

- **state** (State) – The *State* object to save to persistent storage.
- **sync** (*bool*) – Whether *State.sync* should be invoked.

adaptdl.collective module

This module contains simple collective communications primitives which operate on arbitrary python objects. It is meant to be general but *non-performant*. Only use these primitives if you are synchronizing small objects which can be efficiently pickled and operated on. For larger objects, use framework-specific functions, such as those provided by *torch.distributed*.

The functions in this module should be invoked *in the same order* across all replicas in the current job. Otherwise, their behavior is undefined and you may encounter unexpected bugs and errors.

`adaptdl.collective.allreduce`(*value*, *reduce_fn=<function default_reduce_fn>*)

Reduces a value across all replicas in such a way that they all get the final result. Blocks until this function is invoked by all replicas.

Parameters

- **value** (*object*) – The object which will be reduced together with all other replicas.
- **reduce_fn** (*Function*) – A reduction function which two objects as arguments, and returns the resulting reduced object.

Returns Resulting value after being reduced across all replicas.

Return type object

Raises `RuntimeError` – If this module has not been initialized.

`adaptdl.collective.allreduce_async`(*value*, *reduce_fn=<function default_reduce_fn>*)

Asynchronous version of the *allreduce* function. Does not block, instead returns a future which can be used to obtain the result later.

Parameters

- `value` (*object*) – The object which will be reduced together with all other replicas.
- `reduce_fn` (*Function*) – A reduction function which two objects as arguments, and returns the resulting reduced object.

Returns Object from which the result can be obtained later.

Return type *Future*

Raises `RuntimeError` – If this module has not been initialized.

`adaptdl.collective.broadcast`(*value*)

Broadcasts a value from the replica of rank 0 to all replicas. Blocks until this function is invoked by all replicas.

Parameters `value` (*object*) – The object which will be broadcasted from replica 0. Ignored on all other replicas.

Returns The value broadcasted from replica 0.

Return type object

Raises `RuntimeError` – If this module has not been initialized.

`adaptdl.collective.initialize`(*master_addr=None*, *master_port=None*, *replica_rank=None*, *num_replicas=None*)

Initialize this module, must be invoked before calling any other functions. This function will block until it has been invoked from all replicas.

Parameters

- `master_addr` – address of the replica with rank 0.
- `master_port` – free port of the replica with rank 0.
- `replica_rank` – rank of the current replica.
- `num_replicas` – total number of replicas.

Raises `RuntimeError` – If this module had already been initialized.

`adaptdl.collective.teardown`()

Teardown this module, will block until this function has been invoked from all replicas.

Raises `RuntimeError` – If this module has not been initialized.

adaptdl.env module

This module contains functions for retrieving the values of AdaptDL environment variables, or their defaults if unset.

`adaptdl.env.adaptdl_sched_version()`

A string which gives the AdaptDL version of scheduler. Determined by the environment variable `ADAPTDL_SCHED_VERSION` or `None`

Returns AdaptDL version of scheduler, or `None`.

Return type str

`adaptdl.env.checkpoint_path()`

Path to the directory used for saving and loading checkpoints. Determined by the environment variable `ADAPTDL_CHECKPOINT_PATH`, or `None` if unset. Setting this environment variable is required for checkpointing, and is automatically set in AdaptDL-scheduled clusters.

Returns checkpoint path or `None`.

Return type str

`adaptdl.env.from_ray()`

Returns True if the code is being called from Ray

`adaptdl.env.job_id()`

A string which uniquely identifies the current job in an AdaptDL-scheduled cluster. `None` if running standalone.

Returns unique job identifier or `None`.

Return type str

`adaptdl.env.master_addr()`

Network address of the rank 0 replica, required for distributed training. Determined by the environment variable `ADAPTDL_MASTER_ADDR`, or *0.0.0.0* if unset.

In AdaptDL-scheduled clusters, this environment variable is unset. The rank 0 replica is discovered dynamically by querying the supervisor (`supervisor_url()`).

Returns address of the rank 0 replica, or *0.0.0.0*.

Return type str

`adaptdl.env.master_port()`

Available port for the rank 0 replica, required for distributed training. Determined by the environment variable `ADAPTDL_MASTER_PORT`, or 0 if unset. Automatically set in AdaptDL-scheduled clusters.

Returns available port for the rank 0 replica, or 0.

Return type int

`adaptdl.env.num_nodes()`

Number of unique nodes being used for the current job. For example, if there are 4 nodes, each running 2 replicas, then this function returns 4. Determined by the environment variable `ADAPTDL_NUM_NODES`, or is equal to `num_replicas()` if unset. Thus, this environment variable only needs to be set if some node runs multiple replicas. Automatically set in AdaptDL-scheduled clusters.

Returns number of unique nodes, or the value of `num_replicas()`.

Return type int

adaptdl.env.**num_replicas**()

Total number of replicas, required for distributed training. For example, if there are 4 nodes, each running 2 replicas, then this function returns 8. Determined by the environment variable `ADAPTDL_NUM_REPLICAS`, or 1 if unset. Automatically set in AdaptDL-scheduled clusters.

Returns total number of replicas, or 1.

Return type int

adaptdl.env.**num_restarts**()

Number of times the current job was restarted. Determined by the environment variable `ADAPTDL_NUM_RESTARTS`, or 0 if unset. This value is mainly informational, and is automatically set in AdaptDL-scheduled clusters.

Returns number of restarts, or 0.

Return type int

adaptdl.env.**replica_rank**()

Rank of the current replica, required for distributed training. Each replica is assigned a unique rank from 0 to K-1, where K is the total number of replicas. Determined by the environment variable `ADAPTDL_REPLICA_RANK`, or 0 if unset. Automatically set in AdaptDL-scheduled clusters.

Returns rank of the current replica, or 0.

Return type int

adaptdl.env.**share_path**()

Path to a directory shared by all AdaptDL job replicas, which can be used by the application, e.g. for storing downloaded datasets or artifacts. Determined by the environment variable `ADAPTDL_SHARE_PATH`, or `None` if unset. Automatically set in AdaptDL-scheduled clusters.

Returns shared directory path or `None`.

Return type str

adaptdl.env.**supervisor_url**()

URL of the supervisor in an AdaptDL-scheduled cluster. The address of the rank 0 replica is dynamically discovered via the supervisor, instead of via the `ADAPTDL_MASTER_ADDR` environment variable.

Returns URL of the supervisor, or `None`.

Return type str

adaptdl.goodput module

class adaptdl.goodput.**GoodputFunction**(*perf_params*, *grad_params*, *init_batch_size*)

Bases: `object`

efficiency(*batch_size*)

evaluate(*num_nodes*, *num_replicas*, *atomic_bsz*, *accum_steps*)

optimize(*num_nodes*, *num_replicas*, *max_batch_size=None*, *atomic_bsz_range=None*, *accumulation=False*)

throughput(*num_nodes*, *num_replicas*, *atomic_bsz*, *accum_steps*)

class adaptdl.goodput.**GradParams**(*sqr*, *var*)

Bases: `tuple`

property sqr
Alias for field number 0

property var
Alias for field number 1

class adaptdl.goodput.**PerfParams**(*alpha_c*, *beta_c*, *alpha_n*, *beta_n*, *alpha_r*, *beta_r*, *gamma*)
Bases: `tuple`

property alpha_c
Alias for field number 0

property alpha_n
Alias for field number 2

property alpha_r
Alias for field number 4

property beta_c
Alias for field number 1

property beta_n
Alias for field number 3

property beta_r
Alias for field number 5

property gamma
Alias for field number 6

adaptdl.goodput.**fit_perf_params**(*num_nodes*, *num_replicas*, *atomic_bsz*, *accum_step_time*, *optim_step_time*)

adaptdl.reducer module

class adaptdl.reducer.**Future**(*reducer*, *key*)
Bases: `object`

result()

class adaptdl.reducer.**Reducer**(*rank*, *replicas*, *root_host*, *root_port*)
Bases: `object`

Simple asynchronous (all)reduce operations on python objects. Assumes all invokations to allreduce, allreduce_async, and Future.result happen in the same order across all processes.

allreduce(*obj*, *reduce_fn=<function default_reduce_fn>*)

allreduce_async(*obj*, *reduce_fn=<function default_reduce_fn>*)

broadcast(*obj*)
Broadcast a value from replica 0 to all other replicas. Currently uses allreduce with left-projection.

adaptdl.reducer.**default_reduce_fn**(*a*, *b*)

adaptdl.sched_hints module

adaptdl.sched_hints.**post_sched_hints**(*sched_hints*, *job_key*)

adaptdl.utils module

adaptdl.utils.**print_exc**(*function*)

A decorator that wraps the passed in function and prints any exceptions.

PYTHON MODULE INDEX

a

adaptdl, 26
adaptdl.checkpoint, 40
adaptdl.collective, 41
adaptdl.env, 43
adaptdl.goodput, 44
adaptdl.reducer, 45
adaptdl.sched_hints, 46
adaptdl.torch, 26
adaptdl.torch.accumulator, 31
adaptdl.torch.data, 33
adaptdl.torch.epoch, 36
adaptdl.torch.gradient_noise_scale, 37
adaptdl.torch.iterator, 38
adaptdl.torch.parallel, 38
adaptdl.torch.scaling_rules, 39
adaptdl.utils, 46

INDEX

Symbols

__getitem__() (*adaptdl.torch.Accumulator method*), 27
__getitem__() (*adaptdl.torch.accumulator.Accumulator method*), 32
__iadd__() (*adaptdl.torch.Accumulator method*), 27
__iadd__() (*adaptdl.torch.accumulator.Accumulator method*), 32
__isub__() (*adaptdl.torch.Accumulator method*), 27
__isub__() (*adaptdl.torch.accumulator.Accumulator method*), 32
__iter__() (*adaptdl.torch.AdaptiveDataLoader method*), 28
__iter__() (*adaptdl.torch.ElasticSampler method*), 29
__iter__() (*adaptdl.torch.data.AdaptiveDataLoader method*), 33
__iter__() (*adaptdl.torch.data.ElasticSampler method*), 35
__len__() (*adaptdl.torch.ElasticSampler method*), 29
__len__() (*adaptdl.torch.data.ElasticSampler method*), 35

A

accum_count (*adaptdl.torch.gradient_noise_scale.GradientNoiseScale property*), 37
accum_scale (*adaptdl.torch.gradient_noise_scale.GradientNoiseScale property*), 37
accumulation_steps (*adaptdl.torch.data.AdaptiveDataLoaderHelper property*), 33
accumulation_steps (*adaptdl.torch.data.AdaptiveDataLoaderMixin property*), 35
Accumulator (*class in adaptdl.torch*), 26
Accumulator (*class in adaptdl.torch.accumulator*), 31
adaptdl
    module, 26
adaptdl.checkpoint
    module, 40
adaptdl.collective
    module, 41
adaptdl.env
    module, 43
adaptdl.goodput
    module, 44
adaptdl.reducer
    module, 45
adaptdl.sched_hints
    module, 46
adaptdl.torch
    module, 26
adaptdl.torch.accumulator
    module, 31
adaptdl.torch.data
    module, 33
adaptdl.torch.epoch
    module, 36
adaptdl.torch.gradient_noise_scale
    module, 37
adaptdl.torch.iterator
    module, 38
adaptdl.torch.parallel
    module, 38
adaptdl.torch.scaling_rules
    module, 39
adaptdl.utils
    module, 46
adaptdl_sched_version() (*in module adaptdl.env*), 43
AdaptiveBPTTIterator (*class in adaptdl.torch.iterator*), 38
AdaptiveDataLoader (*class in adaptdl.torch*), 28
AdaptiveDataLoader (*class in adaptdl.torch.data*), 33
AdaptiveDataLoaderHelper (*class in adaptdl.torch.data*), 33
AdaptiveDataLoaderMixin (*class in adaptdl.torch.data*), 34
AdaptiveDataParallel (*class in adaptdl.torch*), 28
AdaptiveDataParallel (*class in adaptdl.torch.parallel*), 38
AdaScale (*class in adaptdl.torch.scaling_rules*), 39
allreduce() (*adaptdl.reducer.Reducer method*), 45
allreduce() (*in module adaptdl.collective*), 41
allreduce_async() (*adaptdl.reducer.Reducer method*), 45
allreduce_async() (*in module adaptdl.collective*), 42
alpha_c (*adaptdl.goodput.PerfParams property*), 45

alpha_n (*adaptdl.goodput.PerfParams property*), 45
alpha_r (*adaptdl.goodput.PerfParams property*), 45
autoscale_batch_size() (*adaptdl.torch.data.AdaptiveDataLoaderHelper method*), 33
autoscale_batch_size() (*adaptdl.torch.data.AdaptiveDataLoaderMixin method*), 35

B

beta_c (*adaptdl.goodput.PerfParams property*), 45
beta_n (*adaptdl.goodput.PerfParams property*), 45
beta_r (*adaptdl.goodput.PerfParams property*), 45
broadcast() (*adaptdl.reducer.Reducer method*), 45
broadcast() (*in module adaptdl.collective*), 42

C

checkpoint_path() (*in module adaptdl.env*), 43
context() (*adaptdl.torch.data.AdaptiveDataLoaderHelper method*), 34
current_batch_size (*adaptdl.torch.data.AdaptiveDataLoaderHelper property*), 34
current_batch_size (*adaptdl.torch.data.AdaptiveDataLoaderMixin property*), 35
current_dataloader() (*in module adaptdl.torch*), 30
current_dataloader() (*in module adaptdl.torch.data*), 36
current_epoch() (*in module adaptdl.torch*), 30
current_epoch() (*in module adaptdl.torch.epoch*), 37
current_index (*adaptdl.torch.data.AdaptiveDataLoaderHelper property*), 34
current_local_bsz (*adaptdl.torch.data.AdaptiveDataLoaderHelper property*), 34
current_local_bsz (*adaptdl.torch.data.AdaptiveDataLoaderMixin property*), 35

D

default_reduce_fn() (*in module adaptdl.reducer*), 45

E

efficiency() (*adaptdl.goodput.GoodputFunction method*), 44
ElasticSampler (*class in adaptdl.torch*), 29
ElasticSampler (*class in adaptdl.torch.data*), 35
end_index (*adaptdl.torch.data.AdaptiveDataLoaderHelper property*), 34
evaluate() (*adaptdl.goodput.GoodputFunction method*), 44

F

finished_epochs() (*in module adaptdl.torch*), 30
finished_epochs() (*in module adaptdl.torch.epoch*), 37
fit_perf_params() (*in module adaptdl.goodput*), 45
forward() (*adaptdl.torch.AdaptiveDataParallel method*), 29
forward() (*adaptdl.torch.parallel.AdaptiveDataParallel method*), 39
from_ray() (*in module adaptdl.env*), 43
Future (*class in adaptdl.reducer*), 45

G

gain (*adaptdl.torch.AdaptiveDataParallel property*), 29
gain (*adaptdl.torch.parallel.AdaptiveDataParallel property*), 39
gain() (*adaptdl.torch.gradient_noise_scale.GradientNoiseScale method*), 37
gamma (*adaptdl.goodput.PerfParams property*), 45
get_progress() (*adaptdl.torch.gradient_noise_scale.GradientNoiseScale method*), 38
GoodputFunction (*class in adaptdl.goodput*), 44
GradientNoiseScale (*class in adaptdl.torch.gradient_noise_scale*), 37
GradParams (*class in adaptdl.goodput*), 44

I

init_process_group() (*in module adaptdl.torch*), 30
initialize() (*adaptdl.torch.scaling_rules.ScalingRuleBase method*), 40
initialize() (*in module adaptdl.collective*), 42
is_accum_step() (*adaptdl.torch.data.AdaptiveDataLoaderHelper method*), 34
is_optim_step() (*adaptdl.torch.data.AdaptiveDataLoaderHelper method*), 34

J

job_id() (*in module adaptdl.env*), 43

L

LEGWScale (*class in adaptdl.torch.scaling_rules*), 39
LinearScale (*class in adaptdl.torch.scaling_rules*), 39
load() (*adaptdl.checkpoint.State method*), 40
load_state() (*in module adaptdl.checkpoint*), 41
local_bsz_bounds (*adaptdl.torch.data.AdaptiveDataLoaderHelper property*), 34

M

master_addr() (*in module adaptdl.env*), 43
master_port() (*in module adaptdl.env*), 43
max_batch_size (*adaptdl.torch.data.AdaptiveDataLoaderHelper property*), 34
module
    adaptdl, 26
    adaptdl.checkpoint, 40
    adaptdl.collective, 41
    adaptdl.env, 43

adaptdl.goodput, 44
adaptdl.reducer, 45
adaptdl.sched_hints, 46
adaptdl.torch, 26
adaptdl.torch.accumulator, 31
adaptdl.torch.data, 33
adaptdl.torch.epoch, 36
adaptdl.torch.gradient_noise_scale, 37
adaptdl.torch.iterator, 38
adaptdl.torch.parallel, 38
adaptdl.torch.scaling_rules, 39
adaptdl.utils, 46

N

num_nodes() (*in module adaptdl.env*), 43
num_replicas() (*in module adaptdl.env*), 43
num_restarts() (*in module adaptdl.env*), 44

O

optimize() (*adaptdl.goodput.GoodputFunction method*), 44

P

PerfParams (*class in adaptdl.goodput*), 45
post_sched_hints() (*in module adaptdl.sched_hints*), 46
print_exc() (*in module adaptdl.utils*), 46
profile() (*adaptdl.torch.data.AdaptiveDataLoaderHelper method*), 34

R

raw_sqr_avg (*adaptdl.torch.gradient_noise_scale.GradientNoiseScale property*), 38
raw_var_avg (*adaptdl.torch.gradient_noise_scale.GradientNoiseScale property*), 38
Reducer (*class in adaptdl.reducer*), 45
remaining_epochs_until() (*in module adaptdl.torch*), 30
remaining_epochs_until() (*in module adaptdl.torch.epoch*), 37
replica_rank() (*in module adaptdl.env*), 44
reset_accumulation() (*adaptdl.torch.gradient_noise_scale.GradientNoiseScale method*), 38
result() (*adaptdl.reducer.Future method*), 45

S

save() (*adaptdl.checkpoint.State method*), 41
save_all_states() (*in module adaptdl.checkpoint*), 41
save_state() (*in module adaptdl.checkpoint*), 41
scale_lr() (*adaptdl.torch.scaling_rules.AdaScale method*), 39
scale_lr() (*adaptdl.torch.scaling_rules.LEGWScale method*), 39
scale_lr() (*adaptdl.torch.scaling_rules.LinearScale method*), 40
scale_lr() (*adaptdl.torch.scaling_rules.ScalingRuleBase method*), 40
scale_lr() (*adaptdl.torch.scaling_rules.SqrtScale method*), 40
ScalingRuleBase (*class in adaptdl.torch.scaling_rules*), 40
set_accum_scale() (*adaptdl.torch.gradient_noise_scale.GradientNoiseScale method*), 38
set_epoch() (*adaptdl.torch.data.ElasticSampler method*), 35
set_epoch() (*adaptdl.torch.ElasticSampler method*), 30
set_progress() (*adaptdl.torch.gradient_noise_scale.GradientNoiseScale method*), 38
share_path() (*in module adaptdl.env*), 44
should_zero_grad (*adaptdl.torch.gradient_noise_scale.GradientNoiseScale property*), 38
skipdone() (*adaptdl.torch.data.AdaptiveDataLoaderHelper method*), 34
sqr (*adaptdl.goodput.GradParams property*), 44
sqr_avg() (*adaptdl.torch.gradient_noise_scale.GradientNoiseScale method*), 38
SqrtScale (*class in adaptdl.torch.scaling_rules*), 40
State (*class in adaptdl.checkpoint*), 40
step() (*adaptdl.torch.scaling_rules.ScalingRuleBase method*), 40
subtract() (*adaptdl.torch.Accumulator method*), 27
subtract() (*adaptdl.torch.accumulator.Accumulator method*), 32
supervisor_url() (*in module adaptdl.env*), 44
sync() (*adaptdl.checkpoint.State method*), 41
synchronized() (*adaptdl.torch.Accumulator method*), 27
synchronized() (*adaptdl.torch.accumulator.Accumulator method*), 32

T

teardown() (*in module adaptdl.collective*), 42
throughput() (*adaptdl.goodput.GoodputFunction method*), 44
to_tensorboard() (*adaptdl.torch.AdaptiveDataParallel method*), 29
to_tensorboard() (*adaptdl.torch.data.AdaptiveDataLoaderHelper method*), 34
to_tensorboard() (*adaptdl.torch.data.AdaptiveDataLoaderMixin method*), 35
to_tensorboard() (*adaptdl.torch.parallel.AdaptiveDataParallel method*), 39
train() (*adaptdl.torch.data.AdaptiveDataLoaderHelper method*), 34

training (*adaptdl.torch.AdaptiveDataParallel attribute*), 29
training (*adaptdl.torch.data.AdaptiveDataLoaderHelper property*), 34
training (*adaptdl.torch.data.AdaptiveDataLoaderMixin property*), 35
training (*adaptdl.torch.parallel.AdaptiveDataParallel attribute*), 39

U

update() (*adaptdl.torch.Accumulator method*), 28
update() (*adaptdl.torch.accumulator.Accumulator method*), 32

V

var (*adaptdl.goodput.GradParams property*), 45
var_avg() (*adaptdl.torch.gradient_noise_scale.GradientNoiseScale method*), 38

Z

zero_grad() (*adaptdl.torch.AdaptiveDataParallel method*), 29
zero_grad() (*adaptdl.torch.parallel.AdaptiveDataParallel method*), 39
zero_grad() (*adaptdl.torch.scaling_rules.ScalingRuleBase method*), 40