hi-ml

InnerEye

Nov 03, 2021

WORKING WITH AZURE

This toolbox helps to simplify and streamline work on deep learning models for healthcare and life sciences, by providing tested components (data loaders, pre-processing), deep learning models, and cloud integration tools.

The *hi-ml* toolbox provides

- Functionality to easily run Python code in Azure Machine Learning services
- Low-level and high-level building blocks for Machine Learning / AI researchers and practitioners.

CHAPTER

ONE

FIRST STEPS: HOW TO RUN YOUR PYTHON CODE IN THE CLOUD

The simplest use case for the `hi-ml` toolbox is taking a script that you developed, and run it inside of Azure Machine Learning (AML) services. This can be helpful because the cloud gives you access to massive GPU resource, you can consume vast datasets, and access multiple machines at the same time for distributed training.

1.1 Setting up AzureML

You need to have an AzureML workspace in your Azure subscription. Download the config file from your AzureML workspace, as described here. **Put this file (it should be called `config.json`) into the folder where your script lives**, or one of its parent folders. You can use parent folders up to the last parent that is still included in the `PYTHONPATH` environment variable: `hi-ml` will try to be smart and search through all folders that it thinks belong to your current project.

1.2 Using the AzureML integration layer

Consider a simple use case, where you have a Python script that does something - this could be training a model, or pre-processing some data. The `hi-ml` package can help easily run that on Azure Machine Learning (AML) services.

Here is an example script that reads images from a folder, resizes and saves them to an output folder:

```
from pathlib import Path
if __name__ == '__main__':
    input_folder = Path("/tmp/my_dataset")
    output_folder = Path("/tmp/my_output")
    for file in input_folder.glob("*.jpg"):
        contents = read_image(file)
        resized = contents.resize(0.5)
        write_image(output_folder / file.name)
```

Doing that at scale can take a long time. **We'd like to run that script in AzureML, consume the data from a folder in blob storage, and write the results back to blob storage**, so that we can later use it as an input for model training.

You can achieve that by adding a call to `submit_to_azure_if_needed` from the `hi-ml` package:

```
from pathlib import Path
from health_azure import submit_to_azure_if_needed
if __name__ == '__main__':
    current_file = Path(__file__)
    run_info = submit_to_azure_if_needed(compute_cluster_name="preprocess-ds12",
```

(continues on next page)

(continued from previous page)

```
                                        input_datasets=["images123"],
                                        # Omit this line if you don't create an output␣
↪dataset (for example, in
                                        # model training scripts)
                                        output_datasets=["images123_resized"],
                                        default_datastore="my_datastore")
    # When running in AzureML, run_info.input_datasets and run_info.output_datasets will␣
↪be populated,
    # and point to the data coming from blob storage. For runs outside AML, the paths␣
↪will be None.
    # Replace the None with a meaningful path, so that we can still run the script␣
↪easily outside AML.
    input_dataset = run_info.input_datasets[0] or Path("/tmp/my_dataset")
    output_dataset = run_info.output_datasets[0] or Path("/tmp/my_output")
    files_processed = []
    for file in input_dataset.glob("*.jpg"):
        contents = read_image(file)
        resized = contents.resize(0.5)
        write_image(output_dataset / file.name)
        files_processed.append(file.name)
    # Any other files that you would not consider an "output dataset", like metrics, etc,
↪ should be written to
    # a folder "./outputs". Any files written into that folder will later be visible in␣
↪the AzureML UI.
    # run_info.output_folder already points to the correct folder.
    stats_file = run_info.output_folder / "processed_files.txt"
    stats_file.write_text("\n".join(files_processed))
```

Once these changes are in place, you can submit the script to AzureML by supplying an additional `--azureml` flag on the commandline, like `python myscript.py --azureml`.

Note that you do not need to modify the argument parser of your script to recognize the `--azureml` flag.

1.3 Essential arguments to `submit_to_azure_if_needed`

When calling `submit_to_azure_if_needed`, you can to supply the following parameters:

- `compute_cluster_name` (**Mandatory**): The name of the AzureML cluster that should run the job. This can be a cluster with CPU or GPU machines. See here for documentation
- `entry_script`: The script that should be run. If omitted, the `hi-ml` package will assume that you would like to submit the script that is presently running, given in `sys.argv[0]`.
- `snapshot_root_directory`: The directory that contains all code that should be packaged and sent to AzureML. All Python code that the script uses must be copied over. This defaults to the current working directory, but can be one of its parents. If you would like to explicitly skip some folders inside the `snapshot_root_directory`, then use `ignored_folders` to specify those.
- `conda_environment_file`: The conda configuration file that describes which packages are necessary for your script to run. If omitted, the `hi-ml` package searches for a file called `environment.yml` in the current folder or its parents.

You can also supply an input dataset. For data pre-processing scripts, you can add an output dataset (omit this for ML training scripts).

- To use datasets, you need to provision a data store in your AML workspace, that points to your training data in blob storage. This is described here.
- `input_datasets=["images123"]` in the code above means that the script will consume all data in folder `images123` in blob storage as the input. The folder must exist in blob storage, in the location that you gave when creating the datastore. Once the script has run, it will also register the data in this folder as an AML dataset.
- `output_datasets=["images123_resized"]` means that the script will create a temporary folder when running in AML, and while the job writes data to that folder, upload it to blob storage, in the data store.

For more examples, please see *examples.md*. For more details about datasets, see *here*.

1.4 Additional arguments you should know about

`submit_to_azure_if_needed` has a large number of arguments, please check the API documentation for an exhaustive list. The particularly helpful ones are listed below.

- `experiment_name`: All runs in AzureML are grouped in "experiments". By default, the experiment name is determined by the name of the script you submit, but you can specify a name explicitly with this argument.
- `environment_variables`: A dictionary with the contents of all environment variables that should be set inside the AzureML run, before the script is started.
- `docker_base_image`: This specifies the name of the Docker base image to use for creating the Python environment for your script. The amount of memory to allocate for Docker is given by `docker_shm_size`.
- `num_nodes`: The number of nodes on which your script should run. This is essential for distributed training.
- `tags`: A dictionary mapping from string to string, with additional tags that will be stored on the AzureML run. This is helpful to add metadata about the run for later use.

1.5 Conda environments, Alternate pips, Private wheels

The function `submit_to_azure_if_needed` tries to locate a Conda environment file in the current folder, or in the Python path, with the name `environment.yml`. The actual Conda environment file to use can be specified directly with:

```
run_info = submit_to_azure_if_needed(
    ...
    conda_environment_file=conda_environment_file,
```

where `conda_environment_file` is a `pathlib.Path` or a string identifying the Conda environment file to use.

The basic use of Conda assumes that packages listed are published Conda packages or published Python packages on PyPI. However, during development, the Python package may be on Test.PyPI, or in some other location, in which case the alternative package location can be specified directly with:

```
run_info = submit_to_azure_if_needed(
    ...
    pip_extra_index_url="https://test.pypi.org/simple/",
```

Finally, it is possible to use a private wheel, if the package is only available locally with:

```
run_info = submit_to_azure_if_needed(
    ...
    private_pip_wheel_path=private_pip_wheel_path,
```

where `private_pip_wheel_path` is a `pathlib.Path` or a string identifying the wheel package to use. In this case, this wheel will be copied to the AzureML environment as a private wheel.

CHAPTER

TWO

CONNECTING TO AZURE

2.1 Authentication

The `hi-ml` package uses two possible ways of authentication with Azure. The default is what is called "Interactive Authentication". When you submit a job to Azure via `hi-ml`, this will use the credentials you used in the browser when last logging into Azure. If there are no credentials yet, you should see instructions printed out to the console about how to log in using your browser.

We recommend using Interactive Authentication.

Alternatively, you can use a so-called Service Principal, for example within build pipelines.

2.2 Service Principal Authentication

A Service Principal is a form of generic identity or machine account. This is essential if you would like to submit training runs from code, for example from within an Azure pipeline. You can find more information about application registrations and service principal objects here.

If you would like to use Service Principal, you will need to create it in Azure first, and then store 3 pieces of information in 3 environment variables — please see the instructions below. When all the 3 environment variables are in place, your Azure submissions will automatically use the Service Principal to authenticate.

2.2.1 Creating the Service Principal

1. Navigate back to aka.ms/portal
2. Navigate to `App registrations` (use the top search bar to find it).
3. Click on `+ New registration` on the top left of the page.
4. Choose a name for your application e.g. `MyServicePrincipal` and click `Register`.
5. Once it is created you will see your application in the list appearing under `App registrations`. This step might take a few minutes.
6. Click on the resource to access its properties. In particular, you will need the application ID. You can find this ID in the `Overview` tab (accessible from the list on the left of the page).
7. Create an environment variable called `HIML_SERVICE_PRINCIPAL_ID`, and set its value to the application ID you just saw.
8. You need to create an application secret to access the resources managed by this service principal. On the pane on the left find `Certificates & Secrets`. Click on `+ New client secret` (bottom of the page), note down

your token. Warning: this token will only appear once at the creation of the token, you will not be able to re-display it again later.

9. Create an environment variable called `HIML_SERVICE_PRINCIPAL_PASSWORD`, and set its value to the token you just added.

2.2.2 Providing permissions to the Service Principal

Now that your service principal is created, you need to give permission for it to access and manage your AzureML workspace. To do so:

1. Go to your AzureML workspace. To find it you can type the name of your workspace in the search bar above.
2. On the `Overview` page, there is a link to the Resource Group that contains the workspace. Click on that.
3. When on the Resource Group, navigate to `Access control`. Then click on `+ Add` > `Add role assignment`. A pane will appear on the the right. Select `Role` > `Contributor`. In the `Select` field type the name of your Service Principal and select it. Finish by clicking `Save` at the bottom of the pane.

2.2.3 Azure Tenant ID

The last remaining piece is the Azure tenant ID, which also needs to be available in an environment variable. To get that ID:

1. Log into Azure
2. Via the search bar, find "Azure Active Directory" and open it.
3. In the overview of that, you will see a field "Tenant ID"
4. Create an environment variable called `HIML_TENANT_ID`, and set that to the tenant ID you just saw.

CHAPTER

THREE

DATASETS

3.1 Key concepts

We'll first outline a few concepts that are helpful for understanding datasets.

3.1.1 Blob Storage

Firstly, there is Azure Blob Storage. Each blob storage account has multiple containers - you can think of containers as big disks that store files. The `hi-ml` package assumes that your datasets live in one of those containers, and each top level folder corresponds to one dataset.

3.1.2 AzureML Data Stores

Secondly, there are data stores. This is a concept coming from Azure Machine Learning, described here. Data stores provide access to one blob storage account. They exist so that the credentials to access blob storage do not have to be passed around in the code - rather, the credentials are stored in the data store once and for all.

You can view all data stores in your AzureML workspace by clicking on one of the bottom icons in the left-hand navigation bar of the AzureML studio.

One of these data stores is designated as the default data store.

3.1.3 AzureML Datasets

Thirdly, there are datasets. Again, this is a concept coming from Azure Machine Learning. A dataset is defined by

- A data store
- A set of files accessed through that data store

You can view all datasets in your AzureML workspace by clicking on one of the icons in the left-hand navigation bar of the AzureML studio.

3.1.4 Preparing data

To simplify usage, the `hi-ml` package creates AzureML datasets for you. All you need to do is to

- Create a blob storage account for your data, and within it, a container for your data.
- Create a data store that points to that storage account, and store the credentials for the blob storage account in it

From that point on, you can drop a folder of files in the container that holds your data. Within the `hi-ml` package, just reference the name of the folder, and the package will create a dataset for you, if it does not yet exist.

3.2 Using the datasets

The simplest way of specifying that your script uses a folder of data from blob storage is as follows: Add the `input_datasets` argument to your call of `submit_to_azure_if_needed` like this:

```
from health_azure import submit_to_azure_if_needed
run_info = submit_to_azure_if_needed(...,
                                     input_datasets=["my_folder"],
                                     default_datastore="my_datastore")
input_folder = run_info.input_datasets[0]
```

What will happen under the hood?

- The toolbox will check if there is already an AzureML dataset called "my_folder". If so, it will use that. If there is no dataset of that name, it will create one from all the files in blob storage in folder "my_folder". The dataset will be created using the data store provided, "my_datastore".
- Once the script runs in AzureML, it will download the dataset "my_folder" to a temporary folder.
- You can access this temporary location by `run_info.input_datasets[0]`, and read the files from it.

More complicated setups are described below.

3.2.1 Input and output datasets

Any run in AzureML can consume a number of input datasets. In addition, an AzureML run can also produce an output dataset (or even more than one).

Output datasets are helpful if you would like to run, for example, a script that transforms one dataset into another.

You can use that via the `output_datasets` argument:

```
from health_azure import submit_to_azure_if_needed
run_info = submit_to_azure_if_needed(...,
                                     input_datasets=["my_folder"],
                                     output_datasets=["new_dataset"],
                                     default_datastore="my_datastore")
input_folder = run_info.input_datasets[0]
output_folder = run_info.output_datasets[0]
```

Your script can now read files from `input_folder`, transform them, and write them to `output_folder`. The latter will be a folder on the temp file system of the machine. At the end of the script, the contents of that temp folder will be uploaded to blob storage, and registered as a dataset.

3.2.2 Mounting and downloading

An input dataset can be downloaded before the start of the actual script run, or it can be mounted. When mounted, the files are accessed via the network once needed - this is very helpful for large datasets where downloads would create a long waiting time before the job start.

Similarly, an output dataset can be uploaded at the end of the script, or it can be mounted. Mounting here means that all files will be written to blob storage already while the script runs (rather than at the end).

Note: If you are using mounted output datasets, you should NOT rename files in the output folder.

Mounting and downloading can be triggered by passing in `DatasetConfig` objects for the `input_datasets` argument, like this:

```
from health_azure import DatasetConfig, submit_to_azure_if_needed
input_dataset = DatasetConfig(name="my_folder", datastore="my_datastore", use_
↪mounting=True)
output_dataset = DatasetConfig(name="new_dataset", datastore="my_datastore", use_
↪mounting=True)
run_info = submit_to_azure_if_needed(...,
                                     input_datasets=[input_dataset],
                                     output_datasets=[output_dataset])
input_folder = run_info.input_datasets[0]
output_folder = run_info.output_datasets[0]
```

3.2.3 Local execution

For debugging, it is essential to have the ability to run a script on a local machine, outside of AzureML. Clearly, your script needs to be able to access data in those runs too.

There are two ways of achieving that: Firstly, you can specific an equivalent local folder in the `DatasetConfig` objects:

```
from pathlib import Path
from health_azure import DatasetConfig, submit_to_azure_if_needed
input_dataset = DatasetConfig(name="my_folder",
                              datastore="my_datastore",
                              local_folder=Path("/datasets/my_folder_local"))
run_info = submit_to_azure_if_needed(...,
                                     input_datasets=[input_dataset])
input_folder = run_info.input_datasets[0]
```

Secondly, you can check the returned path in `run_info`, and replace it with something for local execution. `run_info.input_datasets[0]` will be `None` if the script runs outside of AzureML, and no `local_folder` is available.

```
from pathlib import Path
from health_azure import submit_to_azure_if_needed
run_info = submit_to_azure_if_needed(...,
                                     input_datasets=["my_folder"],
                                     default_datastore="my_datastore")
input_folder = run_info.input_datasets[0] or Path("/datasets/my_folder_local")
```

3.2.4 Making a dataset available at a fixed folder location

Occasionally, scripts expect the input dataset at a fixed location, for example, data is always read from `/tmp/mnist`. AzureML has the capability to download/mount a dataset to such a fixed location. With the `hi-ml` package, you can trigger that behaviour via an additional option in the `DatasetConfig` objects:

```
from health_azure import DatasetConfig, submit_to_azure_if_needed
input_dataset = DatasetConfig(name="my_folder",
                              datastore="my_datastore",
                              use_mounting=True,
                              target_folder="/tmp/mnist")
run_info = submit_to_azure_if_needed(...,
                                     input_datasets=[input_dataset])
# Input_folder will now be "/tmp/mnist"
input_folder = run_info.input_datasets[0]
```

3.2.5 Dataset versions

AzureML datasets can have versions, starting at 1. You can view the different versions of a dataset in the AzureML workspace. In the `hi-ml` toolbox, you would always use the latest version of a dataset unless specified otherwise. If you do need a specific version, use the `version` argument in the `DatasetConfig` objects:

```
from health_azure import DatasetConfig, submit_to_azure_if_needed
input_dataset = DatasetConfig(name="my_folder",
                              datastore="my_datastore",
                              version=7)
run_info = submit_to_azure_if_needed(...,
                                     input_datasets=[input_dataset])
input_folder = run_info.input_datasets[0]
```

CHAPTER

FOUR

HYPERPARAMETER SEARCH VIA HYPERDRIVE

HyperDrive runs can start multiple AzureML jobs in parallel. This can be used for tuning hyperparameters, or executing multiple training runs for cross validation. To use that with the `hi-ml` package, simply supply a HyperDrive configuration object as an additional argument. Note that this object needs to be created with an empty `run_config` argument (this will later be replaced with the correct `run_config` that submits your script.)

The example below shows a hyperparameter search that aims to minimize the validation loss `val_loss`, by choosing one of three possible values for the learning rate commandline argument `learning_rate`.

```
from azureml.core import ScriptRunConfig
from azureml.train.hyperdrive import GridParameterSampling, HyperDriveConfig,␣
↪PrimaryMetricGoal, choice
from health_azure import submit_to_azure_if_needed
hyperdrive_config = HyperDriveConfig(
            run_config=ScriptRunConfig(source_directory=""),
            hyperparameter_sampling=GridParameterSampling(
                parameter_space={
                    "learning_rate": choice([0.1, 0.01, 0.001])
                }),
            primary_metric_name="val_loss",
            primary_metric_goal=PrimaryMetricGoal.MINIMIZE,
            max_total_runs=5
        )
submit_to_azure_if_needed(..., hyperdrive_config=hyperdrive_config)
```

For further examples, please check the *example scripts here*, and the HyperDrive documentation.

CHAPTER

FIVE

USING CHEAP LOW PRIORITY VMS

By using Low Priority machines in AzureML, we can run training at greatly reduced costs (around 20% of the original price, see references below for details). This comes with the risk, though, of having the job interrupted and later re-started. This document describes the inner workings of Low Priority compute, and how to best make use of it.

Because the jobs can get interrupted, low priority machines are not suitable for production workload where time is critical. They do offer a lot of benefits though for long-running training jobs or large scale experimentation, that would otherwise be expensive to carry out.

5.1 Setting up the Compute Cluster

Jobs in Azure Machine Learning run in a "compute cluster". When creating a compute cluster, we can specify the size of the VM, the type and number of GPUs, etc. Doing this via the AzureML UI is described here. Doing it programmatically is described here

One of the setting to tweak when creating the compute cluster is whether the machines are "Dedicated" or "Low Priority":

- Dedicated machines will be permanently allocated to your compute cluster. The VMs in a dedicated cluster will be always available, unless the cluster is set up in a way that it removes idle machine. Jobs will not be interrupted.
- Low priority machines effectively make use of spare capacity in the data centers, you can think of them as "dedicated machines that are presently idle". They are available at a much lower price (around 20% of the price of a dedicated machine). These machines are made available to you until they are needed as dedicated machines somewhere else.

In order to get a compute cluster that operates at the lowest price point, choose

- Low priority machines
- Set "Minimum number of nodes" to 0, so that the cluster removes all idle machines if no jobs are running.

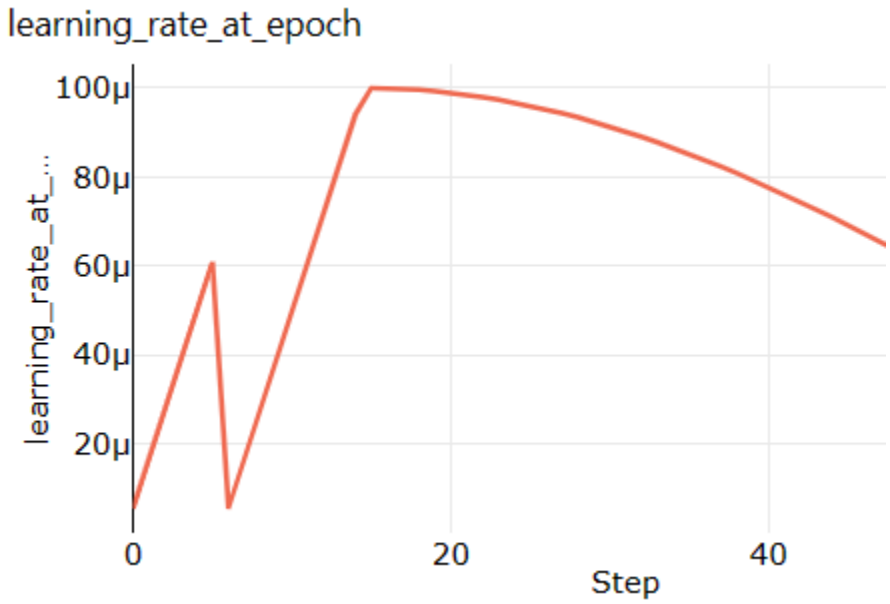
For details on pricing, check this Azure price calculator, choose "Category: GPU". The price for low priority VMs is given in the "Spot" column

5.2 Behaviour of Low Priority VMs

Jobs can be interrupted at any point, this is called "low priority preemption". When interrupted, the job stops - there is no signal that we can make use of to do cleanup or something. All the files that the job has produced up to that point in the `outputs` and `logs` folders will be saved to the cloud.

At some later point, the job will be assigned a virtual machine again. When re-started, all the files that the job had produced in its previous run will be available on disk again where they were before interruption, mounted at the same path. That is, if the interrupted job wrote a file `outputs/foo.txt`, this file will be accessible as `outputs/foo.txt` also after the restart.

Note that all AzureML-internal log files that the job produced in a previous run will be **overwritten** (this behaviour may change in the future). That is in contrast to the behaviour for metrics that the interrupted job had saved to AzureML already (for example, metrics written by a call like `Run.log("loss", loss_tensor.item())`): Those metrics are already stored in AzureML, and will still be there when the job restarts. The re-started job will then **append** to the metrics that had been written in the previous run. This typically shows as sudden jumps in metrics, as illustrated here:



In this example, the learning rate was increasing for the first 6 or so epochs. Then the job got preempted, and started training from scratch, with the initial learning rate and schedule. Note that this behaviour is only an artifact of how the metrics are stored in AzureML, the actual training is doing the right thing.

How do you verify that your job got interrupted? Usually, you would see a warning displayed on the job page in the AzureML UI, that says something along the lines of "Low priority compute preemption warning: a node has been preempted." . You can use kinks in metrics as another indicator that your job got preempted: Sudden jumps in metrics after which the metric follows a shape similar to the one at job start usually indicates low priority preemption.

Note that a job can be interrupted more than one time.

5.3 Best Practice Guide for Your Jobs

In order to make best use of low priority compute, your code needs to be made resilient to restarts. Essentially, this means that it should write regular checkpoints, and try to use those checkpoint files if they already exist. Examples of how to best do that are given below.

In addition, you need to bear in mind that the job can be interrupted at any moment, for example when it is busy uploading huge checkpoint files to Azure. When trying to upload again after restart, there can be resource collisions.

5.3.1 Writing and Using Recovery Checkpoints

When using PyTorch Lightning, you can add a checkpoint callback to your trainer, that ensures that you save the model and optimizer to disk in regular intervals. This callback needs to be added to your `Trainer` object. Note that these recovery checkpoints need to be written to the `outputs` folder, because only files in this folder get saved to Azure automatically when the job gets interrupted.

When starting training, your code needs to check if there is already a recovery checkpoint present on disk. If so, training should resume from that point.

Here is a code snippet that illustrates all that:

```
import re
from pathlib import Path
import numpy as np
from health_ml.utils import AzureMLLogger
from pytorch_lightning import Trainer
from pytorch_lightning.callbacks import ModelCheckpoint

RECOVERY_CHECKPOINT_FILE_NAME = "recovery_"
CHECKPOINT_FOLDER = "outputs/checkpoints"


def get_latest_recovery_checkpoint():
    all_recovery_files = [f for f in Path(CHECKPOINT_FOLDER).glob(RECOVERY_CHECKPOINT_
↪FILE_NAME + "*")]
    if len(all_recovery_files) == 0:
        return None
    # Get recovery checkpoint with highest epoch number
    recovery_epochs = [int(re.findall(r"[\d]+", f.stem)[0]) for f in all_recovery_files]
    idx_max_epoch = int(np.argmax(recovery_epochs))
    return str(all_recovery_files[idx_max_epoch])


recovery_checkpoint = ModelCheckpoint(dirpath=CHECKPOINT_FOLDER,
                                      filename=RECOVERY_CHECKPOINT_FILE_NAME + "{epoch}",
                                      period=10)
trainer = Trainer(default_root_dir="outputs",
                  callbacks=[recovery_checkpoint],
                  logger=[AzureMLLogger()],
                  resume_from_checkpoint=get_latest_recovery_checkpoint())
```

5.4 Additional Optimizers and Other State

In order to be resilient to interruption, your jobs need to save all their state to disk. In PyTorch Lightning training, this would include all optimizers that you are using. The "normal" optimizer for model training is saved to the checkpoint by Lightning already. However, you may be using callbacks or other components that maintain state. As an example, training a linear head for self-supervised learning can be done in a callback, and that callback can have its own optimizer. Such callbacks need to correctly implement the `on_save_checkpoint` method to save their state to the checkpoint, and `on_load_checkpoint` to load it back in.

For more information about persisting state, check the PyTorch Lightning documentation .

CHAPTER

SIX

COMMANDLINE TOOLS

6.1 Run TensorBoard

From the command line, run the command

`himl-tb`

specifying one of `[--experiment] [--latest_run_file] [--run]`

This will start a TensorBoard session, by default running on port 6006. To use an alternative port, specify this with `--port`.

If `--experiment` is provided, the most recent Run from this experiment will be visualised. If `--latest_run_file` is provided, the script will expect to find a RunId in this file. Alternatively you can specify the Runs to visualise via `--run`. This can be a single run id, or multiple ids separated by commas. This argument also accepts one or more run recovery ids, although these are not recommended since it is no longer necessary to provide an experiment name in order to recovery an AML Run.

By default, this tool expects that your TensorBoard logs live in a folder named 'logs' and will create a similarly named folder in your root directory. If your TensorBoard logs are stored elsewhere, you can specify this with the `--log_dir` argument.

If you choose to specify `--experiment`, you can also specify `--num_runs` to view and/or `--tags` to filter by.

If your AML config path is not ROOT_DIR/config.json, you must also specify `--config_file`.

To see an example of how to create TensorBoard logs using PyTorch on AML, see the AML submitting script which submits the following pytorch sample script. Note that to run this, you'll need to create an environment with pytorch and tensorboard as dependencies, as a minimum. See an example conda environemnt. This will create an experiment named 'tensorboard_test' on your Workspace, with a single run. Go to outputs + logs -> outputs to see the tensorboard events file.

6.2 Download files from AML Runs

From the command line, run the command

`himl-download`

specifying one of `[--experiment] [--latest_run_file] [--run]`

If `--experiment` is provided, the most recent Run from this experiment will be downloaded. If `--latest_run_file` is provided, the script will expect to find a RunId in this file. Alternatively you can specify the Run to download via `--run`. This can be a single run id, or multiple ids separated by commas. This argument also accepts one or more run recovery ids, although these are not recommended since it is no longer necessary to provide an experiment name in order to recovery an AML Run.

The files associated with your Run will be downloaded to the location specified with `--output_dir` (by default ROOT_DIR/outputs)

If you choose to specify `--experiment`, you can also specify `--tags` to filter by.

If your AML config path is not `ROOT_DIR/config.json`, you must also specify `--config_file`.

6.3 Creating your own command line tools

When creating your own command line tools that interact with the Azure ML ecosystem, you may wish to use the `AmlRunScriptConfig` class for argument parsing. This gives you a quickstart way for accepting command line arguments to specify the following

- experiment: a string representing the name of an Experiment, from which to retrieve AML runs
- tags: to filter the runs within the given experiment
- num_runs: to define the number of most recent runs to return from the experiment
- run: to instead define one or more run ids from which to retrieve runs (also supports the older format of run recovery ideas although these are obsolete now)
- latest_run_file: to instead provide a path to a file containing the id of your latest run, for retrieval.
- config_path: to specify a config.json file in which your workspace settings are defined

You can extend this list of arguments by creating a child class that inherits from AMLRunScriptConfig.

6.3.1 Defining your own argument types

Additional arguments can have any of the following types: `bool`, `integer`, `float`, `string`, `list`, `class/class instance` with no additional work required. You can also define your own custom type, by providing a custom class in your code that inherits from `CustomTypeParam`. It must define 2 methods:

1. `_validate(self, x: Any)`: which should raise a `ValueError` if x is not of the type you expect, and should also make a call `super()._validate(val)`
2. `from_string(self, y: string)` which takes in the command line arg as a string (`y`) and returns an instance of the type that you want. For example, if your custom type is a tuple, this method should create a tuple from the input string and return that. An example of a custom type can be seen in our own custom type: `RunIdOrListParam`, which accepts a string representing one or more run ids (or run recovery ids) and returns either a List or a single RunId object (or RunRecoveryId object if appropriate)

6.3.2 Example:

```
class EvenNumberParam(util.CustomTypeParam):
    """ Our custom type param for even numbers """

    def _validate(self, val: Any) -> None:
        if (not self.allow_None) and val is None:
            raise ValueError("Value must not be None")
        if val % 2 != 0:
            raise ValueError(f"{val} is not an even number")
        super()._validate(val)  # type: ignore
```

(continues on next page)

(continued from previous page)

```
    def from_string(self, x: str) -> int:
        return int(x)


class MyScriptConfig(util.AmlRunScriptConfig):
    # example of a generic param
    simple_string: str = param.String(default="")
    # example of a custom param
    even_number = EvenNumberParam(2, doc="your choice of even number")
```

CHAPTER

SEVEN

DOWNLOADING FROM/ UPLOADING TO AZURE ML

All of the below functions will attempt to find a current workspace, if running in Azure ML, or else will attempt to locate 'config.json' file in the current directory, and its parents. Alternatively, you can specify your own Workspace object or a path to a file containing the workspace settings.

7.1 Download files from an Azure ML Run

To download all files from an AML Run, given its run id, perform the following:

```
from pathlib import Path
from health_azure import download_files_from_run_id
run_id = "example_run_id_123"
output_folder = Path("path/to/save")
download_files_from_run_id(run_id, output_folder)
```

Here, "path_to_save" represents the folder in which we want the downloaded files to be stored. E.g. if your run contains the files ["abc/def/1.txt", "abc/2.txt"] and you specify the prefix "abc" and the output_folder "my_outputs", you'll end up with the files ["my_outputs/abc/def/1.txt", "my_outputs/abc/2.txt"]

If you wish to specify the file name(s) to be downloaded, you can do so with the "prefix" parameter. E.g. prefix="outputs" will download all files within the "output" folder, if such a folder exists within your Run.

There is an additional parameter, "validate_checksum" which defaults to False. If True, will validate MD5 hash of the data arriving (in chunks) to that being sent.

Note that if your code is running in a distributed manner, files will only be downloaded onto nodes with local rank = 0. E.g. if you have 2 nodes each running 4 processes, the file will be downloaded by CPU/GPU 0 on each of the 2 nodes. All processes will be synchronized to only exit the downloading method once it has completed on all nodes/ranks.

7.2 Downloading checkpoint files from a run

To download checkpoint files from an Azure ML Run, perform the following:

```
from pathlib import Path
from health_azure import download_checkpoints_from_run_id
download_checkpoints_from_run_id("example_run_id_123", Path("path/to/checkpoint/directory
↪"))
```

All files within the checkpoint directory will be downloaded into the folder specified by "path/to/checkpoint_directory".

Since checkpoint files are often large and therefore prone to corruption during download, by default, this function will validate the MD5 hash of the data downloaded (in chunks) compared to that being sent.

Note that if your code is running in a distributed manner, files will only be downloaded onto nodes with local rank = 0. E.g. if you have 2 nodes each running 4 processes, the file will be downloaded by CPU/GPU 0 on each of the 2 nodes. All processes will be synchronized to only exit the downloading method once it has completed on all nodes/ranks.

7.3 Downloading files from an Azure ML Datastore

To download data from an Azure ML Datastore within your Workspace, follow this example:

```
from pathlib import Path
from health_azure import download_from_datastore
download_from_datastore("datastore_name", "prefix", Path("path/to/output/directory") )
```

where "prefix" represents the path to the file(s) to be downloaded, relative to the datastore "datastore_name". Azure will search for files within the Datastore whose paths begin with this string. If you wish to download multiple files from the same folder, set equal to that folder's path within the Datastore. If you wish to download a single file, include both the path to the folder it resides in, as well as the filename itself. If the relevant file(s) are found, they will be downloaded to the folder specified by <output_folder>. If this directory does not already exist, it will be created. E.g. if your datastore contains the paths ["foo/bar/1.txt", "foo/bar/2.txt"] and you call this function with file_prefix="foo/bar" and output_folder="outputs", you would end up with the files ["outputs/foo/bar/1.txt", "outputs/foo/bar/2.txt"]

This function takes additional parameters "overwrite" and "show_progress". If True, overwrite will overwrite any existing local files with the same path. If False and there is a duplicate file, it will skip this file. If show_progress is set to True, the progress of the file download will be visible in the terminal.

7.4 Uploading files to an Azure ML Datastore

To upload data to an Azure ML Datastore within your workspace, perform the following:

```
from pathlib import Path
from health_azure import upload_to_datastore
upload_to_datastore("datastore_name", Path("path/to/local/data/folder"), Path("path/to/
↪datastore/folder") )
```

Where "datastore_name" is the name of the registered Datastore within your workspace that you wish to upload to and "path/to/datastore/folder" is the relative path within this Datastore that you wish to upload data to. Note that the path to local data must be a folder, not a single path. The folder name will not be included in the remote path. E.g. if you specify the local_data_dir="foo/bar" and that contains the files ["1.txt", "2.txt"], and you specify the remote_path="baz", you would see the following paths uploaded to your Datastore: ["baz/1.txt", "baz/2.txt"]

This function takes additional parameters "overwrite" and "show_progress". If True, overwrite will overwrite any existing remote files with the same path. If False and there is a duplicate file, it will skip this file. If show_progress is set to True, the progress of the file upload will be visible in the terminal.

CHAPTER

EIGHT

EXAMPLES

Note: All examples below contain links to sample scripts that are also included in the repository. The experience is **optimized for use on readthedocs**. When navigating to the sample scripts on the github UI, you will only see the `.rst` file that links to the `.py` file. To access the `.py` file, go to the folder that contains the respective `.rst` file.

8.1 Basic integration

The sample examples/1/sample.py is a script that takes an optional command line argument of a target value and prints all the prime numbers up to (but not including) this target. It is simply intended to demonstrate a long running operation that we want to run in Azure. Run it using e.g.

```
cd examples/1
python sample.py -n 103
```

The sample examples/2/sample.py shows the minimal modifications to run this in AzureML. Firstly create an AzureML workspace and download the config file, as explained here. The config file should be placed in the same folder as the sample script. A sample Conda environment file is supplied. Import the hi-ml package into the current environment. Finally add the following to the sample script:

```
from health_azure import submit_to_azure_if_needed
    ...
def main() -> None:
    _ = submit_to_azure_if_needed(
        compute_cluster_name="lite-testing-ds2",
        wait_for_completion=True,
        wait_for_completion_show_output=True)
```

Replace `lite-testing-ds2` with the name of a compute cluster created within the AzureML workspace. If this script is invoked as the first sample, e.g.

```
cd examples/2
python sample.py -n 103
```

then the output will be exactly the same. But if the script is invoked as follows:

```
cd examples/2
python sample.py -n 103 --azureml
```

then the function `submit_to_azure_if_needed` will perform all the required actions to run this script in AzureML and exit. Note that:

- code after `submit_to_azure_if_needed` is not run locally, but it is run in AzureML.

- the print statement prints to the AzureML console output and is available in the `Output + logs` tab of the experiment in the `70_driver_log.txt` file, and can be downloaded from there.
- the command line arguments are passed through (apart from –azureml) when running in AzureML.
- a new file: `most_recent_run.txt` will be created containing an identifier of this AzureML run.

A sample script examples/2/results.py demonstrates how to programmatically download the driver log file.

8.2 Output files

The sample examples/3/sample.py demonstrates output file handling when running on AzureML. Because each run is performed in a separate VM or cluster then any file output is not generally preserved. In order to keep the output it should be written to the `outputs` folder when running in AzureML. The AzureML infrastructure will preserve this and it will be available for download from the `outputs` folder in the `Output + logs` tab.

Make the following additions:

```
    from health_azure import submit_to_azure_if_needed
    run_info = submit_to_azure_if_needed(
    ...
    parser.add_argument("-o", "--output", type=str, default="primes.txt", required=False,
↪ help="Output file name")
    ...
    output = run_info.output_folder / args.output
    output.write_text("\n".join(map(str, primes)))
```

When running locally `submit_to_azure_if_needed` will create a subfolder called `outputs` and then the output can be written to the file `args.output` there. When running in AzureML the output will be available in the file `args.output` in the Experiment.

A sample script examples/3/results.py demonstrates how to programmatically download the output file.

8.3 Output datasets

The sample examples/4/sample.py demonstrates output dataset handling when running on AzureML.

In this case, the following parameters are added to `submit_to_azure_if_needed`:

```
    from health_azure import submit_to_azure_if_needed
    run_info = submit_to_azure_if_needed(
        ...
        default_datastore="himldatasets",
        output_datasets=["himl_sample4_output"],
```

The `default_datastore` is required if using the simplest configuration for an output dataset, to just use the blob container name. There is an alternative that doesn't require the `default_datastore` and allows a different datastore for each dataset:

```
from health_azure import DatasetConfig, submit_to_azure_if_needed
    ...
    run_info = submit_to_azure_if_needed(
        ...
        output_datasets=[DatasetConfig(name="himl_sample4_output", datastore=
↪"himldatasets")]
```

(continues on next page)

(continued from previous page)

Now the output folder is constructed as follows:

```
    output_folder = run_info.output_datasets[0] or Path("outputs") / "himl_sample4_output
↪"
    output_folder.mkdir(parents=True, exist_ok=True)
    output = output_folder / args.output
```

When running in AzureML `run_info.output_datasets[0]` will be populated using the new parameter and the output will be written to that blob storage. When running locally `run_info.output_datasets[0]` will be None and a local folder will be created and used.

A sample script examples/4/results.py demonstrates how to programmatically download the output dataset file.

For more details about datasets, see *here*

8.4 Input datasets

This example trains a simple classifier on a toy dataset, first creating the dataset files and then in a second script training the classifier.

The script examples/5/inputs.py is provided to prepare the csv files. Run the script to download the Iris dataset and create two CSV files:

```
cd examples/5
python inputs.py
```

The training script examples/5/sample.py is modified from https://github.com/Azure/MachineLearningNotebooks/blob/master/how-to-use-azureml/ml-frameworks/scikit-learn/train-hyperparameter-tune-deploy-with-sklearn/train_iris.py to work with input csv files. Start it to train the actual classifier, based on the data files that were just written:

```
cd examples/5
python sample.py
```

8.4.1 Including input files in the snapshot

When using very small datafiles (in the order of few MB), the easiest way to get the input data to Azure is to include them in the set of (source) files that are uploaded to Azure. You can run the dataset creation script on your local machine, writing the resulting two files to the same folder where your training script is located, and then submit the training script to AzureML. Because the dataset files are in the same folder, they will automatically be uploaded to AzureML.

However, it is not ideal to have the input files in the snapshot: The size of the snapshot is limited to 25 MB. It is better to put the data files into blob storage and use input datasets.

8.4.2 Creating the dataset in AzureML

The suggested way of creating a dataset is to run a script in AzureML that writes an output dataset. This is particularly important for large datasets, to avoid the usually low bandwith from a local machine to the cloud.

This is shown in examples/6/inputs.py: This script prepares the CSV files in an AzureML run, and writes them to an output dataset called `himl_sample6_input`. The relevant code parts are:

```
run_info = submit_to_azure_if_needed(
    compute_cluster_name="lite-testing-ds2",
    default_datastore="himldatasets",
    output_datasets=["himl_sample6_input"])
# The dataset files should be written into this folder:
dataset = run_info.output_datasets[0] or Path("dataset")
```

Run the script:

```
cd examples/6
python inputs.py --azureml
```

You can now modify the training script examples/6/sample.py to use the newly created dataset `himl_sample6_input` as an input. To do that, the following parameters are added to `submit_to_azure_if_needed`:

```
run_info = submit_to_azure_if_needed(
    compute_cluster_name="lite-testing-ds2",
    default_datastore="himldatasets",
    input_datasets=["himl_sample6_input"])
```

When running in AzureML, the dataset will be downloaded before running the job. You can access the temporary folder where the dataset is available like this:

```
input_folder = run_info.input_datasets[0] or Path("dataset")
```

The part behind the `or` statement is only necessary to keep a reasonable behaviour when running outside of AzureML: When running in AzureML `run_info.input_datasets[0]` will be populated using input dataset specified in the call to `submit_to_azure_if_needed`, and the input will be downloaded from blob storage. When running locally `run_info.input_datasets[0]` will be `None` and a local folder should be populated and used.

The `default_datastore` is required if using the simplest configuration for an input dataset. There are alternatives that do not require the `default_datastore` and allows a different datastore for each dataset, for example:

```
from health_azure import DatasetConfig, submit_to_azure_if_needed
    ...
    run_info = submit_to_azure_if_needed(
        ...
        input_datasets=[DatasetConfig(name="himl_sample7_input", datastore="himldatasets
↪"],
```

For more details about datasets, see *here*

8.4.3 Uploading the input files manually

An alternative to writing the dataset in AzureML (as suggested above) is to create them on the local machine, and upload them manually directly to Azure blob storage.

This is shown in examples/7/inputs.py: This script prepares the CSV files and uploads them to blob storage, in a folder called `himl_sample7_input`. Run the script:

```
cd examples/7
python inputs_via_upload.py
```

As in the above example, you can now modify the training script examples/7/sample.py to use an input dataset that has the same name as the folder where the files just got uploaded. In this case, the following parameters are added to `submit_to_azure_if_needed`:

```
    run_info = submit_to_azure_if_needed(
        ...
        default_datastore="himldatasets",
        input_datasets=["himl_sample7_input"],
```

8.5 Hyperdrive

The sample examples/8/sample.py demonstrates adding hyperparameter tuning. This shows the same hyperparameter search as in the AzureML sample.

Make the following additions:

```
from azureml.core import ScriptRunConfig
from azureml.train.hyperdrive import HyperDriveConfig, PrimaryMetricGoal, choice
from azureml.train.hyperdrive.sampling import RandomParameterSampling
    ...
def main() -> None:
    param_sampling = RandomParameterSampling({
        "--kernel": choice('linear', 'rbf', 'poly', 'sigmoid'),
        "--penalty": choice(0.5, 1, 1.5)
    })

    hyperdrive_config = HyperDriveConfig(
        run_config=ScriptRunConfig(source_directory=""),
        hyperparameter_sampling=param_sampling,
        primary_metric_name='Accuracy',
        primary_metric_goal=PrimaryMetricGoal.MAXIMIZE,
        max_total_runs=12,
        max_concurrent_runs=4)

    run_info = submit_to_azure_if_needed(
        ...
        hyperdrive_config=hyperdrive_config)
```

Note that this does not make sense to run locally, it should always be run in AzureML. When invoked with:

```
cd examples/8
python sample.py --azureml
```

this will perform a Hyperdrive run in AzureML, i.e. there will be 12 child runs, each randomly drawing from the parameter sample space. AzureML can plot the metrics from the child runs, but to do that, some small modifications are required.

Add in:

```
    run = run_info.run
    ...
    args = parser.parse_args()
    run.log('Kernel type', np.str(args.kernel))
    run.log('Penalty', np.float(args.penalty))
    ...
    print('Accuracy of SVM classifier on test set: {:.2f}'.format(accuracy))
    run.log('Accuracy', np.float(accuracy))
```

and these metrics will be displayed on the child runs tab in the Experiment page on AzureML.

8.6 Controlling when to submit to AzureML and when not

By default, the `hi-ml` package assumes that you supply a commandline argument `--azureml` (that can be anywhere on the commandline) to trigger a submission of the present script to AzureML. If you wish to control it via a different flag, coming out of your own argument parser, use the `submit_to_azureml` argument of the function `health.azure.himl.submit_to_azure_if_needed`.

CHAPTER

NINE

LOGGING METRICS WHEN TRAINING MODELS IN AZUREML

This section describes the basics of logging to AzureML, and how this can be simplified when using PyTorch Lightning. It also describes helper functions to make logging more consistent across your code.

9.1 Basics

The mechanics of writing metrics to an ML training run inside of AzureML are described here.

Using the `hi-ml-azure` toolbox, you can simplify that like this:

```
from health_azure import RUN_CONTEXT
...
RUN_CONTEXT.log(name="name_of_the_metric", value=my_tensor.item())
```

Similarly you can log strings (via the `log_text` method) or figures (via the `log_image` method), see the documentation.

9.2 Using PyTorch Lightning

The `hi-ml` toolbox relies on `pytorch-lightning` for a lot of its functionality. Logging of metrics is described in detail here

`hi-ml` provides a Lightning-ready logger object to use with AzureML. You can add that to your trainer as you would add a Tensorboard logger, and afterwards see all metrics in both your Tensorboard files and in the AzureML UI. This logger can be added to the `Trainer` object as follows:

```
from health_ml.utils import AzureMLLogger
from pytorch_lightning.loggers import TensorBoardLogger
tb_logger = TensorBoardLogger("logs/")
azureml_logger = AzureMLLogger()
trainer = Trainer(logger=[tb_logger, azureml_logger])
```

You do not need to make any changes to your logging code to write to both loggers at the same time. This means that, if your code correctly writes to Tensorboard in a local run, you can expect the metrics to come out correctly in the AzureML UI as well after adding the `AzureMLLogger`.

9.3 Making logging consistent when training with PyTorch Lightning

A common problem of training scripts is that the calls to the logging methods tend to run out of sync. The `.log` method of a `LightningModule` has a lot of arguments, some of which need to be set correctly when running on multiple GPUs.

To simplify that, there is a function `log_on_epoch` that turns synchronization across nodes on/off depending on the number of GPUs, and always forces the metrics to be logged upon epoch completion. Use as follows:

```
from health_ml.utils import log_on_epoch
from pytorch_lightning import LightningModule

class MyModule(LightningModule):
    def training_step(self, *args, **kwargs):
        ...
        loss = my_loss(y_pred, y)
        log_on_epoch(self, loss)
        return loss
```

9.3.1 Logging learning rates

Logging learning rates is important for monitoring training, but again this can add overhead. To log learning rates easily and consistently, we suggest either of two options:

- Add a `LearningRateMonitor` callback to your trainer, as described here
- Use the `hi-ml` function `log_learning_rate`

The `log_learning_rate` function can be used at any point the training code, like this:

```
from health_ml.utils import log_learning_rate
from pytorch_lightning import LightningModule

class MyModule(LightningModule):
    def training_step(self, *args, **kwargs):
        ...
        log_learning_rate(self, "learning_rate")
        loss = my_loss(y_pred, y)
        return loss
```

`log_learning_rate` will log values from all learning rate schedulers, and all learning rates if a scheduler returns multiple values. In this example, the logged metric will be `learning_rate` if there is a single scheduler that outputs a single LR, or `learning_rate/1/0` to indicate the value coming from scheduler index 1, value index 0.

CHAPTER

TEN

PERFORMANCE DIAGNOSTICS

The `hi-ml` toolbox offers several components to integrate with PyTorch Lightning based training workflows:

- The `AzureMLProgressBar` is a replacement for the default progress bar that the Lightning Trainer uses. Its output is more suitable for display in an offline setup like AzureML.
- The `BatchTimeCallback` can be added to the trainer to detect performance issues with data loading.

10.1 AzureMLProgressBar

The standard PyTorch Lightning is well suited for interactive training sessions on a GPU machine, but its output can get confusing when run inside AzureML. The `AzureMLProgressBar` class can replace the standard progress bar, and optionally adds timestamps to each progress event. This makes it easier to later correlate training progress with, for example, low GPU utilization showing in AzureML's GPU monitoring.

Here's a code snippet to add the progress bar to a PyTorch Lightning Trainer object:

```
from health_ml.utils import AzureMLProgressBar
from pytorch_lightning import Trainer

progress = AzureMLProgressBar(refresh_rate=100, print_timestamp=True)
trainer = Trainer(callbacks=[progress])
```

This produces progress information like this:

```
2021-10-20T06:06:07Z Training epoch 18 (step 94):    5/5 (100%) completed. 00:00 elapsed,
↪ total epoch time ~ 00:00
2021-10-20T06:06:07Z Validation epoch 18:    2/2 (100%) completed. 00:00 elapsed, total␣
↪epoch time ~ 00:00
2021-10-20T06:06:07Z Training epoch 19 (step 99):    5/5 (100%) completed. 00:00 elapsed,
↪ total epoch time ~ 00:00
...
```

10.2 BatchTimeCallback

This callback can help diagnose issues with low performance of data loading. It captures the time between the end of a training or validation step, and the start of the next step. This is often indicative of the time it takes to retrieve the next batch of data: When the data loaders are not performant enough, this time increases.

The `BatchTimeCallback` will detect minibatches where the estimated data loading time is too high, and print alerts. These alerts will be printed at most 5 times per epoch, for a maximum of 3 epochs, to avoid cluttering the output.

Note that it is common for the first minibatch of data in an epoch to take a long time to load, because data loader processes need to spin up.

The callback will log a set of metrics:

- `timing/train/batch_time [sec] avg` and `timing/train/batch_time [sec] max`: Average and maximum time that it takes for batches to train/validate
- `timing/train/batch_loading_over_threshold [sec]` is the total time wasted per epoch in waiting for the next batch of data. This is computed by looking at all batches where the batch loading time was over the threshold `max_batch_load_time_seconds` (that is set in the constructor of the callback), and totalling the batch loading time for those batches.
- `timing/train/epoch_time [sec]` is the time for an epoch to complete.

10.2.1 Caveats

- In distributed training, the performance metrics will be collected at rank 0 only.
- The time between the end of a batch and the start of the next batch is also impacted by other callbacks. If you have callbacks that are particularly expensive to run, for example because they actually have their own model training, the results of the `BatchTimeCallback` may be misleading.

10.2.2 Usage example

```
from health_ml.utils import BatchTimeCallback
from pytorch_lightning import Trainer

batchtime = BatchTimeCallback(max_batch_load_time_seconds=0.5)
trainer = Trainer(callbacks=[batchtime])
```

This would produce output like this:

```
Epoch 18 training: Loaded the first minibatch of data in 0.00 sec.
Epoch 18 validation: Loaded the first minibatch of data in 0.00 sec.
Epoch 18 training took 0.02sec, of which waiting for data took 0.01 sec total.
Epoch 18 validation took 0.00sec, of which waiting for data took 0.00 sec total.
```

CHAPTER

ELEVEN

NOTES FOR DEVELOPERS

11.1 Creating a Conda environment

To create a separate Conda environment with all packages that `hi-ml` requires for running and testing, use the provided `environment.yml` file. Create a Conda environment called `himl` from that via

```
conda env create --file environment.yml
conda activate himl
```

11.2 Installing `pyright`

We are using static typechecking for our code via `mypy` and `pyright`. The latter requires a separate installation outside the Conda environment. For WSL, these are the required steps (see also here):

```
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.38.0/install.sh | bash
nvm install node
npm install -g pyright
```

11.3 Using specific versions `hi-ml` in your Python environments

If you'd like to test specific changes to the `hi-ml` package in your code, you can use two different routes:

- You can clone the `hi-ml` repository on your machine, and use `hi-ml` in your Python environment via a local package install:

```
pip install -e <your_git_folder>/hi-ml
```

- You can consume an early version of the package from `test.pypi.org` via `pip`:

```
pip install --extra-index-url https://test.pypi.org/simple/ hi-ml==0.1.0.post165
```

- If you are using Conda, you can add an additional parameter for `pip` into the Conda `environment.yml` file like this:

```
name: foo
dependencies:
  - pip=20.1.1
```

(continues on next page)

(continued from previous page)

```
  - python=3.7.3
  - pip:
      - --extra-index-url https://test.pypi.org/simple/
      - hi-ml==0.1.0.post165
```

11.4 Common things to do

The repository contains a makefile with definitions for common operations.

- `make check`: Run `flake8` and `mypy` on the repository.
- `make test`: Run `flake8` and `mypy` on the repository, then all tests via `pytest`
- `make pip`: Install all packages for running and testing in the current interpreter.
- `make conda`: Update the hi-ml Conda environment and activate it

11.5 Building documentation

To build the sphinx documentation, you must have sphinx and related packages installed (see `build_requirements.txt` in the repository root). Then run:

```
cd docs
make html
```

This will build all your documentation in `docs/build/html`.

11.6 Setting up your AzureML workspace

- In the browser, navigate to the AzureML workspace that you want to use for running your tests.
- In the top right section, there will be a dropdown menu showing the name of your AzureML workspace. Expand that.
- In the panel, there is a link "Download config file". Click that.
- This will download a file `config.json`. Move that file to the root folder of your `hi-ml` repository. The file name is already present in `.gitignore`, and will hence not be checked in.

11.7 Creating and Deleting Docker Environments in AzureML

- Passing a `docker_base_image` into `submit_to_azure_if_needed` causes a new image to be built and registered in your workspace (see docs for more information).
- To remove an environment use the az ml environment delete function in the AzureML CLI (note that all the parameters need to be set, none are optional).

11.8 Testing

For all of the tests to work locally you will need to cache your AzureML credentials. One simple way to do this is to run the example in `src/health/azure/examples` (i.e. run `python elevate_this.py --message='Hello World' --azureml` or `make example`) after editing `elevate_this.py` to reference your compute cluster.

When running the tests locally, they can either be run against the source directly, or the source built into a package.

- To run the tests against the source directly in the local `src` folder, ensure that there is no wheel in the `dist` folder (for example by running `make clean`). If a wheel is not detected, then the local `src` folder will be copied into the temporary test folder as part of the test process.
- To run the tests against the source as a package, build it with `make build`. This will build the local `src` folder into a new wheel in the `dist` folder. This wheel will be detected and passed to AzureML as a private package as part of the test process.

11.9 Creating a New Release

To create a new package release, follow these steps:

- Double-check that `CHANGELOG.md` is up-to-date: It should contain a section for the next package version with subsections Added/Changed/...
- On the repository's github page, click on "Releases", then "Draft a new release"
- In the "Draft a new release" page, click "Choose a tag". In the text box, enter a (new) tag name that has the desired version number, plus a "v" prefix. For example, to create package version 0.12.17, create a tag `v0.12.17`. Then choose "+ Create new tag" below the text box.
- Enter a "Release title" that highlights the main feature(s) of this new package version.
- Click "Auto-generate release notes" to pull in the titles of the Pull Requests since the last release.
- Before the auto-generated "What's changed" section, add a few sentences that summarize what's new.
- Click "Publish release"

CHAPTER

TWELVE

CONTRIBUTING TO THIS TOOLBOX

We welcome all contributions that help us achieve our aim of speeding up ML/AI research in health and life sciences. Examples of contributions are

- Data loaders for specific health & life sciences data
- Network architectures and components for deep learning models
- Tools to analyze and/or visualize data

All contributions to the toolbox need to come with unit tests, and will be reviewed when a Pull Request (PR) is started. If in doubt, reach out to the core `hi-ml` team before starting your work.

Please look through the existing folder structure to find a good home for your contribution.

12.1 Submitting a Pull Request

If you'd like to submit a PR to the codebase, please ensure you:

- Include a brief description
- Link to an issue, if relevant
- Write unit tests for the code - see below for details.
- Add appropriate documentation for any new code that you introduce
- Ensure that you modified CHANGELOG.md and described your PR there.
- Only publish your PR for review once you have a build that is passing. You can make use of the "Create as Draft" feature of GitHub.

12.2 Code style

- We use `flake8` as a linter, and `mypy` and pyright for static typechecking. Both tools run as part of the PR build, and must run without errors for a contribution to be accepted. `mypy` requires that all functions and methods carry type annotations, see mypy documentation
- We highly recommend to run all those tools *before* pushing the latest changes to a PR. If you have `make` installed, you can run both tools in one go via `make check` (from the repository root folder)

12.3 Unit testing

- DO write unit tests for each new function or class that you add.
- DO extend unit tests for existing functions or classes if you change their core behaviour.
- DO try your best to write unit tests that are fast. Very often, this can be done by reducing data size to a minimum. Also, it is helpful to avoid long-running integration tests, but try to test at the level of the smallest involved function.
- DO ensure that your tests are designed in a way that they can pass on the local machine, even if they are relying on specific cloud features. If required, use `unittest.mock` to simulate the cloud features, and hence enable the tests to run successfully on your local machine.
- DO run all unit tests on your dev machine before submitting your changes. The test suite is designed to pass completely also outside of cloud builds.
- DO NOT rely only on the test builds in the cloud (i.e., run test locally before submitting). Cloud builds trigger AzureML runs on GPU machines that have a far higher CO2 footprint than your dev machine.
- When fixing a bug, the suggested workflow is to first write a unit test that shows the invalid behaviour, and only then start to code up the fix.

12.4 Correct Sphinx Documentation

Common mistakes when writing docstrings:

- There must be a separating line between a function description and the documentation for its parameters.
- In multi-line parameter descriptions, continuations on the next line must be indented.
- Sphinx will merge the class description and the arguments of the constructor `__init__`. Hence, there is no need to write any text in the constructor, only the classes' parameters.
- Use >>> to include code snippets. PyCharm will run intellisense on those to make authoring easier.
- To generate the Sphinx documentation on your dev machine, run `make html` in the `./docs` folder, and then open `./docs/build/html/index.html`

Example:

```
class Foo:
    """
    This is the class description.

    The following block will be pretty-printed by Sphinx. Note the space between >>> and␣
↪the code!

    Usage example:
        >>> from module import Foo
        >>> foo = Foo(bar=1.23)
    """

    ANY_ATTRIBUTE = "what_ever."
    """Document class attributes after the attribute."""

    def __init__(self, bar: float = 0.5) -> None:
```

(continues on next page)

(continued from previous page)

```
        """
        :param bar: This is a description for the constructor argument.
            Long descriptions should be indented.
        """
        self.bar = bar

    def method(self, arg: int) -> None:
        """
        Method description, followed by an empty line.

        :param arg: This is a description for the method argument.
            Long descriptions should be indented.
        """
```

CHAPTER

THIRTEEN

WHOLE SLIDE IMAGES

Computational Pathology works with image files that can be very large in size, up to many GB. These files may be too large to load entirely into memory at once, or at least too large to act as training data. Instead they may be split into multiple tiles of a much smaller size, e.g. 224x224 pixels before being used for training. There are two popular libraries used for handling this type of image:

- OpenSlide
- cuCIM

but they both come with trade offs and complications.

In development there is also tifffile, but this is untested.

13.1 OpenSlide

There is a Python interface for OpenSlide at openslide-python, but this first requires the installation of the OpenSlide library itself. This can be done on Ubuntu with:

```
apt-get install openslide-tools
```

On Windows follow the instructions here and make sure that the install directory is added to the system path.

Once the shared library/dlls are installed, install the Python interface with:

```
pip install openslide-python
```

13.2 cuCIM

cuCIM is much easier to install, it can be done entirely with the Python package: cucim. However, there are the following caveats:

- It requires a GPU, with NVIDIA driver 450.36+
- It requires CUDA 11.0+
- It supports only a subset of tiff image files.

The suitable AzureML base Docker images are therefore the ones containing `cuda11`, and the compute instance must contain a GPU.

13.3 Performance

An exploratory set of scripts are at slide_image_loading for comparing loading images with OpenSlide or cuCIM, and performing tiling using both libraries.

13.3.1 Loading and saving at lowest resolution

Four test tiff files are used:

- a 44.5 MB file with level dimensions: ((27648, 29440), (6912, 7360), (1728, 1840))
- a 19.9 MB file with level dimensions: ((5888, 25344), (1472, 6336), (368, 1584))
- a 5.5 MB file with level dimensions: ((27648, 29440), (6912, 7360), (1728, 1840)), but acting as a mask
- a 2.1 MB file with level dimensions: ((5888, 25344), (1472, 6336), (368, 1584)), but acting as a mask

For OpenSlide the following code:

```
    with OpenSlide(str(input_file)) as img:
        count = img.level_count
        dimensions = img.level_dimensions

        print(f"level_count: {count}")
        print(f"dimensions: {dimensions}")

        for k, v in img.properties.items():
            print(k, v)

        region = img.read_region(location=(0, 0),
                                 level=count-1,
                                 size=dimensions[count-1])
        region.save(output_file)
```

took an average of 29ms to open the file, 88ms to read the region, and 243ms to save the region as a png.

For cuCIM the following code:

```
    img = cucim.CuImage(str(input_file))

    count = img.resolutions['level_count']
    dimensions = img.resolutions['level_dimensions']

    print(f"level_count: {count}")
    print(f"level_dimensions: {dimensions}")

    print(img.metadata)

    region = img.read_region(location=(0, 0),
                             size=dimensions[count-1],
                             level=count-1)
    np_img_arr = np.asarray(region)
    img2 = Image.fromarray(np_img_arr)
    img2.save(output_file)
```

took an average of 369ms to open the file, 7ms to read the region and 197ms to save the region as a png, but note that it failed to handle the mask images.

13.3.2 Loading and saving as tiles at the medium resolution

Test code created tiles of size 224x224 pilfes, loaded the mask images, and used occupancy levels to decide which tiles to create and save from level 1 - the middle resolution. This was profiled against both images, as above.

For cuCIM the total time was 4.7s, 2.48s to retain the tiles as a Numpy stack but not save them as pngs. cuCIM has the option of cacheing images, but is actually made performance slightly worse, possibly because the natural tile sizes in the original tiffs were larger than the tile sizes.

For OpenSlide the comparable total times were 5.7s, and 3.26s.

CHAPTER

FOURTEEN

HEALTH_AZURE PACKAGE

14.1 Functions

create_run_configuration(workspace, ..., [, ...])	Creates an AzureML run configuration, that contains information about environment, multi node execution, and Docker.
create_script_run([snapshot_root_directory, ...])	Creates an AzureML ScriptRunConfig object, that holds the information about the snapshot, the entry script, and its arguments.
download_files_from_run_id(run_id, output_folder)	For a given Azure ML run id, first retrieve the Run, and then download all files, which optionally start with a given prefix.
download_checkpoints_from_run_id(run_id, ...)	Given an Azure ML run id, download all files from a given checkpoint directory within that run, to the path specified by output_path.
download_from_datastore(datastore_name, ...)	Download file(s) from an Azure ML Datastore that are registered within a given Workspace.
fetch_run(workspace, run_recovery_id)	Finds an existing run in an experiment, based on a recovery ID that contains the experiment ID and the actual RunId.
get_most_recent_run(run_recovery_file, workspace)	Gets the name of the most recently executed AzureML run, instantiates that Run object and returns it.
get_workspace([aml_workspace, ...])	Retrieve an Azure ML Workspace from one of several places:
is_running_in_azure_ml([aml_run])	Returns True if the given run is inside of an AzureML machine, or False if it is on a machine outside AzureML.
set_environment_variables_for_multi_node()	Sets the environment variables that PyTorch Lightning needs for multi-node training.
split_recovery_id(id_str)	Splits a run ID into the experiment name and the actual run.
submit_run(workspace, experiment_name, ...)	Starts an AzureML run on a given workspace, via the script_run_config.
submit_to_azure_if_needed([...])	Submit a folder to Azure, if needed and run it.
torch_barrier()	This is a barrier to use in distributed jobs.
upload_to_datastore(datastore_name, ..., [, ...])	Upload a folder to an Azure ML Datastore that is registered within a given Workspace.

14.1.1 create_run_configuration

health_azure.**create_run_configuration**(*workspace*, *compute_cluster_name*, *conda_environment_file=None*, *aml_environment_name=''*, *environment_variables=None*, *pip_extra_index_url=''*, *private_pip_wheel_path=None*, *docker_base_image=''*, *docker_shm_size=''*, *num_nodes=1*, *max_run_duration=''*, *input_datasets=None*, *output_datasets=None*)

Creates an AzureML run configuration, that contains information about environment, multi node execution, and Docker.

Parameters

- **workspace** (Workspace) – The AzureML Workspace to use.
- **aml_environment_name** (str) – The name of an AzureML environment that should be used to submit the script. If not provided, an environment will be created from the arguments to this function (conda_environment_file, pip_extra_index_url, environment_variables, docker_base_image)
- **max_run_duration** (str) – The maximum runtime that is allowed for this job in AzureML. This is given as a floating point number with a string suffix s, m, h, d for seconds, minutes, hours, day. Examples: '3.5h', '2d'
- **compute_cluster_name** (str) – The name of the AzureML cluster that should run the job. This can be a cluster with CPU or GPU machines.
- **conda_environment_file** (Optional[Path]) – The conda configuration file that describes which packages are necessary for your script to run.
- **environment_variables** (Optional[Dict[str, str]]) – The environment variables that should be set when running in AzureML.
- **docker_base_image** (str) – The Docker base image that should be used when creating a new Docker image.
- **docker_shm_size** (str) – The Docker shared memory size that should be used when creating a new Docker image.
- **pip_extra_index_url** (str) – If provided, use this PIP package index to find additional packages when building the Docker image.
- **private_pip_wheel_path** (Optional[Path]) – If provided, add this wheel as a private package to the AzureML workspace.
- **conda_environment_file** – The file that contains the Conda environment definition.
- **input_datasets** (Optional[List[*DatasetConfig*]]) – The script will consume all data in folder in blob storage as the input. The folder must exist in blob storage, in the location that you gave when creating the datastore. Once the script has run, it will also register the data in this folder as an AzureML dataset.
- **output_datasets** (Optional[List[*DatasetConfig*]]) – The script will create a temporary folder when running in AzureML, and while the job writes data to that folder, upload it to blob storage, in the data store.
- **num_nodes** (int) – The number of nodes to use in distributed training on AzureML.

Return type RunConfiguration

Returns

14.1.2 create_script_run

health_azure.**create_script_run**(*snapshot_root_directory=None*, *entry_script=None*, *script_params=None*)

Creates an AzureML ScriptRunConfig object, that holds the information about the snapshot, the entry script, and its arguments.

Parameters

- **entry_script** (Union[Path, str, None]) – The script that should be run in AzureML.
- **snapshot_root_directory** (Optional[Path]) – The directory that contains all code that should be packaged and sent to AzureML. All Python code that the script uses must be copied over.
- **script_params** (Optional[List[str]]) – A list of parameter to pass on to the script as it runs in AzureML. If empty (or None, the default) these will be copied over from sys.argv, omitting the –azureml flag.

Return type ScriptRunConfig

Returns

14.1.3 download_files_from_run_id

health_azure.**download_files_from_run_id**(*run_id*, *output_folder*, *prefix=''*, *workspace=None*, *workspace_config_path=None*, *validate_checksum=False*)

For a given Azure ML run id, first retrieve the Run, and then download all files, which optionally start with a given prefix. E.g. if the Run creates a folder called "outputs", which you wish to download all files from, specify prefix="outputs". To download all files associated with the run, leave prefix empty.

If not running inside AML and neither a workspace nor the config file are provided, the code will try to locate a config.json file in any of the parent folders of the current working directory. If that succeeds, that config.json file will be used to instantiate the workspace.

If function is called in a distributed PyTorch training script, the files will only be downloaded once per node (i.e, all process where is_local_rank_zero() == True). All processes will exit this function once all downloads are completed.

Parameters

- **run_id** (str) – The id of the Azure ML Run
- **output_folder** (Path) – Local directory to which the Run files should be downloaded.
- **prefix** (str) – Optional prefix to filter Run files by
- **workspace** (Optional[Workspace]) – Optional Azure ML Workspace object
- **workspace_config_path** (Optional[Path]) – Optional path to settings for Azure ML Workspace
- **validate_checksum** (bool) – Whether to validate the content from HTTP response

Return type None

14.1.4 download_checkpoints_from_run_id

health_azure.**download_checkpoints_from_run_id**(*run_id*, *checkpoint_path_or_folder*, *output_folder*, *aml_workspace=None*, *workspace_config_path=None*)

Given an Azure ML run id, download all files from a given checkpoint directory within that run, to the path specified by output_path. If running in AML, will take the current workspace. Otherwise, if neither aml_workspace nor workspace_config_path are provided, will try to locate a config.json file in any of the parent folders of the current working directory.

Parameters

- **run_id** (str) – The id of the run to download checkpoints from
- **checkpoint_path_or_folder** (str) – The path to the either a single checkpoint file, or a directory of checkpoints within the run files. If a folder is provided, all files within it will be downloaded.
- **output_folder** (Path) – The path to which the checkpoints should be stored
- **aml_workspace** (Optional[Workspace]) – Optional AML workspace object
- **workspace_config_path** (Optional[Path]) – Optional workspace config file

Return type None

14.1.5 download_from_datastore

health_azure.**download_from_datastore**(*datastore_name*, *file_prefix*, *output_folder*, *aml_workspace=None*, *workspace_config_path=None*, *overwrite=False*, *show_progress=False*)

Download file(s) from an Azure ML Datastore that are registered within a given Workspace. The path to the file(s) to be downloaded, relative to the datastore <datastore_name>, is specified by the parameter "prefix". Azure will search for files within the Datastore whose paths begin with this string. If you wish to download multiple files from the same folder, set <prefix> equal to that folder's path within the Datastore. If you wish to download a single file, include both the path to the folder it resides in, as well as the filename itself. If the relevant file(s) are found, they will be downloaded to the folder specified by <output_folder>. If this directory does not already exist, it will be created. E.g. if your datastore contains the paths ["foo/bar/1.txt", "foo/bar/2.txt"] and you call this function with file_prefix="foo/bar" and output_folder="outputs", you would end up with the files ["outputs/foo/bar/1.txt", "outputs/foo/bar/2.txt"]

If not running inside AML and neither a workspace nor the config file are provided, the code will try to locate a config.json file in any of the parent folders of the current working directory. If that succeeds, that config.json file will be used to instantiate the workspace.

Parameters

- **datastore_name** (str) – The name of the Datastore containing the blob to be downloaded. This Datastore itself must be an instance of an AzureBlobDatastore.
- **file_prefix** (str) – The prefix to the blob to be downloaded
- **output_folder** (Path) – The directory into which the blob should be downloaded
- **aml_workspace** (Optional[Workspace]) – Optional Azure ML Workspace object
- **workspace_config_path** (Optional[Path]) – Optional path to settings for Azure ML Workspace
- **overwrite** (bool) – If True, will overwrite any existing file at the same remote path. If False, will skip any duplicate file.

- **show_progress** (bool) – If True, will show the progress of the file download

Return type None

14.1.6 fetch_run

health_azure.**fetch_run**(*workspace*, *run_recovery_id*)

Finds an existing run in an experiment, based on a recovery ID that contains the experiment ID and the actual RunId. The run can be specified either in the experiment_name:run_id format, or just the run_id.

Parameters

- **workspace** (Workspace) – the configured AzureML workspace to search for the experiment.
- **run_recovery_id** (str) – The Run to find. Either in the full recovery ID format, experiment_name:run_id or just the run_id

Return type Run

Returns The AzureML run.

14.1.7 get_most_recent_run

health_azure.**get_most_recent_run**(*run_recovery_file*, *workspace*)

Gets the name of the most recently executed AzureML run, instantiates that Run object and returns it.

Parameters

- **run_recovery_file** (Path) – The path of the run recovery file
- **workspace** (Workspace) – Azure ML Workspace

Return type Run

Returns The Run

14.1.8 get_workspace

health_azure.**get_workspace**(*aml_workspace=None*, *workspace_config_path=None*)

Retrieve an Azure ML Workspace from one of several places:

1. If the function has been called during an AML run (i.e. on an Azure agent), returns the associated workspace
2. If a Workspace object has been provided by the user, return that
3. If a path to a Workspace config file has been provided, load the workspace according to that.

If not running inside AML and neither a workspace nor the config file are provided, the code will try to locate a config.json file in any of the parent folders of the current working directory. If that succeeds, that config.json file will be used to instantiate the workspace.

Parameters

- **aml_workspace** (Optional[Workspace]) – If provided this is returned as the AzureML Workspace.

- **workspace_config_path** (Optional[Path]) – If not provided with an AzureML Workspace, then load one given the information in this config

Return type Workspace

Returns An AzureML workspace.

14.1.9 is_running_in_azure_ml

health_azure.**is_running_in_azure_ml**(*aml_run=<azureml.core.run._OfflineRun object>*)

Returns True if the given run is inside of an AzureML machine, or False if it is on a machine outside AzureML. When called without arguments, this functions returns True if the present code is running in AzureML. Note that in runs with "compute_target='local'" this function will also return True. Such runs execute outside of AzureML, but are able to log all their metrics, etc to an AzureML run.

Parameters **aml_run** (Run) – The run to check. If omitted, use the default run in RUN_CONTEXT

Return type bool

Returns True if the given run is inside of an AzureML machine, or False if it is a machine outside AzureML.

14.1.10 set_environment_variables_for_multi_node

health_azure.**set_environment_variables_for_multi_node**()

Sets the environment variables that PyTorch Lightning needs for multi-node training.

Return type None

14.1.11 split_recovery_id

health_azure.**split_recovery_id**(*id_str*)

Splits a run ID into the experiment name and the actual run. The argument can be in the format 'experiment_name:run_id', or just a run ID like user_branch_abcde12_123. In the latter case, everything before the last two alphanumeric parts is assumed to be the experiment name.

Parameters **id_str** (str) – The string run ID.

Return type Tuple[str, str]

Returns experiment name and run name

14.1.12 submit_run

health_azure.**submit_run**(*workspace*, *experiment_name*, *script_run_config*, *tags=None*, *wait_for_completion=False*, *wait_for_completion_show_output=False*)

Starts an AzureML run on a given workspace, via the script_run_config.

Parameters

- **workspace** (Workspace) – The AzureML workspace to use.
- **experiment_name** (str) – The name of the experiment that will be used or created. If the experiment name contains characters that are not valid in Azure, those will be removed.
- **script_run_config** (Union[ScriptRunConfig, HyperDriveConfig]) – The settings that describe which script should be run.

- **tags** (Optional[Dict[str, str]]) – A dictionary of string key/value pairs, that will be added as metadata to the run. If set to None, a default metadata field will be added that only contains the commandline arguments that started the run.
- **wait_for_completion** (bool) – If False (the default) return after the run is submitted to AzureML, otherwise wait for the completion of this run (if True).
- **wait_for_completion_show_output** (bool) – If wait_for_completion is True this parameter indicates whether to show the run output on sys.stdout.

Return type Run

Returns An AzureML Run object.

14.1.13 submit_to_azure_if_needed

health_azure.**submit_to_azure_if_needed**(*compute_cluster_name=''*, *entry_script=None*, *aml_workspace=None*, *workspace_config_file=None*, *snapshot_root_directory=None*, *script_params=None*, *conda_environment_file=None*, *aml_environment_name=''*, *experiment_name=None*, *environment_variables=None*, *pip_extra_index_url=''*, *private_pip_wheel_path=None*, *docker_base_image=''*, *docker_shm_size=''*, *ignored_folders=None*, *default_datastore=''*, *input_datasets=None*, *output_datasets=None*, *num_nodes=1*, *wait_for_completion=False*, *wait_for_completion_show_output=False*, *max_run_duration=''*, *submit_to_azureml=None*, *tags=None*, *after_submission=None*, *hyperdrive_config=None*)

Submit a folder to Azure, if needed and run it. Use the commandline flag –azureml to submit to AzureML, and leave it out to run locally.

Parameters

- **after_submission** (Optional[Callable[[Run], None]]) – A function that will be called directly after submitting the job to AzureML. The only argument to this function is the run that was just submitted. Use this to, for example, add additional tags or print information about the run.
- **tags** (Optional[Dict[str, str]]) – A dictionary of string key/value pairs, that will be added as metadata to the run. If set to None, a default metadata field will be added that only contains the commandline arguments that started the run.
- **aml_environment_name** (str) – The name of an AzureML environment that should be used to submit the script. If not provided, an environment will be created from the arguments to this function.
- **max_run_duration** (str) – The maximum runtime that is allowed for this job in AzureML. This is given as a floating point number with a string suffix s, m, h, d for seconds, minutes, hours, day. Examples: '3.5h', '2d'
- **experiment_name** (Optional[str]) – The name of the AzureML experiment in which the run should be submitted. If omitted, this is created based on the name of the current script.
- **entry_script** (Union[Path, str, None]) – The script that should be run in AzureML
- **compute_cluster_name** (str) – The name of the AzureML cluster that should run the job. This can be a cluster with CPU or GPU machines.

- **conda_environment_file** (Union[Path, str, None]) – The conda configuration file that describes which packages are necessary for your script to run.
- **aml_workspace** (Optional[Workspace]) – There are two optional parameters used to glean an existing AzureML Workspace. The simplest is to pass it in as a parameter.
- **workspace_config_file** (Union[Path, str, None]) – The 2nd option is to specify the path to the config.json file downloaded from the Azure portal from which we can retrieve the existing Workspace.
- **snapshot_root_directory** (Union[Path, str, None]) – The directory that contains all code that should be packaged and sent to AzureML. All Python code that the script uses must be copied over.
- **ignored_folders** (Optional[List[Union[Path, str]]]) – A list of folders to exclude from the snapshot when copying it to AzureML.
- **script_params** (Optional[List[str]]) – A list of parameter to pass on to the script as it runs in AzureML. If empty (or None, the default) these will be copied over from sys.argv, omitting the –azureml flag.
- **environment_variables** (Optional[Dict[str, str]]) – The environment variables that should be set when running in AzureML.
- **docker_base_image** (str) – The Docker base image that should be used when creating a new Docker image.
- **docker_shm_size** (str) – The Docker shared memory size that should be used when creating a new Docker image.
- **pip_extra_index_url** (str) – If provided, use this PIP package index to find additional packages when building the Docker image.
- **private_pip_wheel_path** (Union[Path, str, None]) – If provided, add this wheel as a private package to the AzureML workspace.
- **default_datastore** (str) – The data store in your AzureML workspace, that points to your training data in blob storage. This is described in more detail in the README.
- **input_datasets** (Optional[List[Union[str, *DatasetConfig*]]]) – The script will consume all data in folder in blob storage as the input. The folder must exist in blob storage, in the location that you gave when creating the datastore. Once the script has run, it will also register the data in this folder as an AzureML dataset.
- **output_datasets** (Optional[List[Union[str, *DatasetConfig*]]]) – The script will create a temporary folder when running in AzureML, and while the job writes data to that folder, upload it to blob storage, in the data store.
- **num_nodes** (int) – The number of nodes to use in distributed training on AzureML.
- **wait_for_completion** (bool) – If False (the default) return after the run is submitted to AzureML, otherwise wait for the completion of this run (if True).
- **wait_for_completion_show_output** (bool) – If wait_for_completion is True this parameter indicates whether to show the run output on sys.stdout.
- **submit_to_azureml** (Optional[bool]) – If True, the codepath to create an AzureML run will be executed. If False, the codepath for local execution (i.e., return immediately) will be executed. If not provided (None), submission to AzureML will be triggered if the commandline flag '–azureml' is present in sys.argv
- **hyperdrive_config** (Optional[HyperDriveConfig]) – A configuration object for Hyperdrive (hyperparameter search).

Return type *AzureRunInfo*

Returns If the script is submitted to AzureML then we terminate python as the script should be executed in AzureML, otherwise we return a AzureRunInfo object.

14.1.14 torch_barrier

health_azure.**torch_barrier**()

This is a barrier to use in distributed jobs. Use it to make all processes that participate in a distributed pytorch job to wait for each other. When torch.distributed is not set up or not found, the function exits immediately.

Return type None

14.1.15 upload_to_datastore

health_azure.**upload_to_datastore**(*datastore_name*, *local_data_folder*, *remote_path*, *aml_workspace=None*, *workspace_config_path=None*, *overwrite=False*, *show_progress=False*)

Upload a folder to an Azure ML Datastore that is registered within a given Workspace. Note that this will upload all files within the folder, but will not copy the folder itself. E.g. if you specify the local_data_dir="foo/bar" and that contains the files ["1.txt", "2.txt"], and you specify the remote_path="baz", you would see the following paths uploaded to your Datastore: ["baz/1.txt", "baz/2.txt"]

If not running inside AML and neither a workspace nor the config file are provided, the code will try to locate a config.json file in any of the parent folders of the current working directory. If that succeeds, that config.json file will be used to instantiate the workspace.

Parameters

- **datastore_name** (str) – The name of the Datastore to which the blob should be uploaded. This Datastore itself must be an instance of an AzureBlobDatastore
- **local_data_folder** (Path) – The path to the local directory containing the data to be uploaded
- **remote_path** (Path) – The path to which the blob should be uploaded
- **aml_workspace** (Optional[Workspace]) – Optional Azure ML Workspace object
- **workspace_config_path** (Optional[Path]) – Optional path to settings for Azure ML Workspace
- **overwrite** (bool) – If True, will overwrite any existing file at the same remote path. If False, will skip any duplicate files and continue to the next.
- **show_progress** (bool) – If True, will show the progress of the file download

Return type None

14.2 Classes

AzureRunInfo(input_datasets, ...)	This class stores all information that a script needs to run inside and outside of AzureML.
DatasetConfig(name[, datastore, version, ...])	Contains information to use AzureML datasets as inputs or outputs.

14.2.1 AzureRunInfo

class health_azure.**AzureRunInfo**(*input_datasets*, *output_datasets*, *run*, *is_running_in_azure_ml*, *output_folder*, *logs_folder*)

Bases: `object`

This class stores all information that a script needs to run inside and outside of AzureML. It is return from *submit_to_azure_if_needed*, where the return value depends on whether the script is inside or outside AzureML.

Please check the source code for detailed documentation for all fields.

14.2.2 DatasetConfig

class health_azure.**DatasetConfig**(*name*, *datastore=''*, *version=None*, *use_mounting=None*, *target_folder=''*, *local_folder=None*)

Bases: `object`

Contains information to use AzureML datasets as inputs or outputs.

Parameters

- **name** (`str`) – The name of the dataset, as it was registered in the AzureML workspace. For output datasets, this will be the name given to the newly created dataset.
- **datastore** (`str`) – The name of the AzureML datastore that holds the dataset. This can be empty if the AzureML workspace has only a single datastore, or if the default datastore should be used.
- **version** (`Optional[int]`) – The version of the dataset that should be used. This is only used for input datasets. If the version is not specified, the latest version will be used.
- **use_mounting** (`Optional[bool]`) – If True, the dataset will be "mounted", that is, individual files will be read or written on-demand over the network. If False, the dataset will be fully downloaded before the job starts, respectively fully uploaded at job end for output datasets. Defaults: False (downloading) for datasets that are script inputs, True (mounting) for datasets that are script outputs.
- **target_folder** (`str`) – The folder into which the dataset should be downloaded or mounted. If left empty, a random folder on /tmp will be chosen.
- **local_folder** (`Optional[Path]`) – The folder on the local machine at which the dataset is available. This is used only for runs outside of AzureML.

Methods Summary

to_input_dataset(workspace, dataset_index)	Creates a configuration for using an AzureML dataset inside of an AzureML run.
to_output_dataset(workspace, dataset_index)	Creates a configuration to write a script output to an AzureML dataset.

Methods Documentation

to_input_dataset(*workspace*, *dataset_index*)

Creates a configuration for using an AzureML dataset inside of an AzureML run. This will make the AzureML dataset with given name available as a named input, using INPUT_0 as the key for dataset index 0.

Parameters

- **workspace** (Workspace) – The AzureML workspace to read from.
- **dataset_index** (int) – Suffix for using datasets as named inputs, the dataset will be marked INPUT_{index}

Return type DatasetConsumptionConfig

to_output_dataset(*workspace*, *dataset_index*)

Creates a configuration to write a script output to an AzureML dataset. The name and datastore of this new dataset will be taken from the present object.

Parameters

- **workspace** (Workspace) – The AzureML workspace to read from.
- **dataset_index** (int) – Suffix for using datasets as named inputs, the dataset will be marked OUTPUT_{index}

Return type OutputFileDatasetConfig

Returns

CHAPTER

FIFTEEN

HEALTH_ML.UTILS PACKAGE

15.1 Functions

log_learning_rate(module[, name])	Logs the learning rate(s) used by the given module.
log_on_epoch(module[, name, value, metrics, ...])	Write a dictionary with metrics and/or an individual metric as a name/value pair to the loggers of the given module.

15.1.1 log_learning_rate

health_ml.utils.**log_learning_rate**(*module*, *name='learning_rate'*)

Logs the learning rate(s) used by the given module. Multiple learning rate schedulers and/or multiple rates per scheduler are supported. The learning rates are logged under the given name. If multiple scheduler and/or multiple rates are used, a suffix like "/0/1" is added, to indicate the learning rate for scheduler 0, index 1, for example. Learning rates are logged on epoch.

Parameters

- **module** (LightningModule) – The module for which the learning rates should be logged.
- **name** (str) – The name to use when logging the learning rates.

Return type None

15.1.2 log_on_epoch

health_ml.utils.**log_on_epoch**(*module*, *name=None*, *value=None*, *metrics=None*, *reduce_fx=<built-in method mean of type object>*, *sync_dist=None*, *sync_dist_op='mean'*)

Write a dictionary with metrics and/or an individual metric as a name/value pair to the loggers of the given module. Metrics are always logged upon epoch completion. The metrics in question first synchronized across GPUs if DDP with >1 node is used, using the sync_dist_op (default: mean). Afterwards, they are aggregated across all steps via the reduce_fx (default: mean). Metrics that are fed in as plain numbers rather than tensors (for example, plain Python integers) are converted to tensors before logging, to enable synchronization across GPUs if needed.

Parameters

- **name** (Optional[str]) – The name of the metric to log.
- **value** (Optional[Any]) – The actual value of the metric to log.
- **metrics** (Optional[Mapping[str, Any]]) – A dictionary with metrics to log.

- **module** (LightningModule) – The PyTorch Lightning module where the metrics should be logged.
- **sync_dist** (Optional[bool]) – If not None, use this value for the sync_dist argument to module.log. If None, set it automatically depending on the use of DDP. Set this to False if you want to log metrics that are only available on Rank 0 of a DDP job.
- **reduce_fx** (Callable) – The reduce function to apply to the per-step values, after synchronizing the tensors across GPUs. Default: torch.mean
- **sync_dist_op** (Any) – The reduce operation to use when synchronizing the tensors across GPUs. This must be a value recognized by sync_ddp: 'sum', 'mean', 'avg'

Return type None

15.2 Classes

AzureMLLogger()	A Pytorch Lightning logger that stores metrics in the current AzureML run.
AzureMLProgressBar([refresh_rate, ...])	A PL progress bar that works better in AzureML.
BatchTimeCallback([...])	This callback provides tools to measure batch loading time and other diagnostic information.

15.2.1 AzureMLLogger

class health_ml.utils.**AzureMLLogger**

Bases: pytorch_lightning.loggers.base.LightningLoggerBase

A Pytorch Lightning logger that stores metrics in the current AzureML run. If the present run is not inside AzureML, nothing gets logged.

Methods Summary

experiment()	Return the experiment object associated with this logger.
log_hyperparams(params)	Logs the given model hyperparameters to AzureML as a table.
log_metrics(metrics[, step])	Writes the given metrics dictionary to the AzureML run context.
name()	Return the experiment name.
version()	Return the experiment version.

Methods Documentation

experiment()
Return the experiment object associated with this logger.

Return type Any

log_hyperparams(*params*)
Logs the given model hyperparameters to AzureML as a table. Namespaces are converted to dictionaries. Nested dictionaries are flattened out.

Return type None

log_metrics(*metrics*, *step=None*)
Writes the given metrics dictionary to the AzureML run context. If the metrics dictionary has an *epoch* key, the *step* value (x-axis for plots) is left empty. If there is no *epoch* key, the *step* value is taken from the function argument. This is the case for metrics that are logged with the *on_step=True* flag.

Parameters

- **metrics** (Dict[str, float]) – A dictionary with metrics to log. Keys are strings, values are floating point numbers.
- **step** (Optional[int]) – The trainer global step for logging.

Return type None

name()
Return the experiment name.

Return type Any

version()
Return the experiment version.

Return type int

15.2.2 AzureMLProgressBar

class health_ml.utils.**AzureMLProgressBar**(*refresh_rate=50*, *print_timestamp=True*, *write_to_logging_info=False*)
Bases: pytorch_lightning.callbacks.progress.base.ProgressBarBase

A PL progress bar that works better in AzureML. It prints timestamps for each message, and works well with a setup where there is no direct access to the console.

Usage example:

```
>>> from health_ml.utils import AzureMLProgressBar
>>> from pytorch_lightning import Trainer
>>> progress = AzureMLProgressBar(refresh_rate=100)
>>> trainer = Trainer(callbacks=[progress])
```

Parameters

- **refresh_rate** (int) – The number of steps after which the progress should be printed out.
- **print_timestamp** (bool) – If True, each message that the progress bar prints will be prefixed with the current time in UTC. If False, no such prefix will be added.

- **write_to_logging_info** (bool) – If True, the progress information will be printed via logging.info. If False, it will be printed to stdout via print.

Attributes Summary

PROGRESS_STAGE_PREDICT	A string that indicates that the trainer loop is presently in prediction mode.
PROGRESS_STAGE_TEST	A string that indicates that the trainer loop is presently in testing mode.
PROGRESS_STAGE_TRAIN	A string that indicates that the trainer loop is presently in training mode.
PROGRESS_STAGE_VAL	A string that indicates that the trainer loop is presently in validation mode.
is_disabled	**rtype** bool
is_enabled	**rtype** bool
refresh_rate	**rtype** int

Methods Summary

disable()	You should provide a way to disable the progress bar.
enable()	You should provide a way to enable the progress bar.
on_predict_batch_end(*args, **kwargs)	Called when the predict batch ends.
on_predict_epoch_start(trainer, pl_module)	Called when the predict epoch begins.
on_test_batch_end(*args, **kwargs)	Called when the test batch ends.
on_test_epoch_start(trainer, pl_module)	Called when the test epoch begins.
on_train_batch_end(*args, **kwargs)	Called when the train batch ends.
on_train_epoch_start(trainer, pl_module)	Called when the train epoch begins.
on_validation_batch_end(*args, **kwargs)	Called when the validation batch ends.
on_validation_start(trainer, pl_module)	Called when the validation loop begins.
start_stage(stage, total_num_batches)	Sets the information that a new stage of the PL loop is starting.
update_progress(batches_processed)	Writes progress information once the refresh interval is full.

Attributes Documentation

PROGRESS_STAGE_PREDICT = 'Prediction'
A string that indicates that the trainer loop is presently in prediction mode.

PROGRESS_STAGE_TEST = 'Testing'
A string that indicates that the trainer loop is presently in testing mode.

PROGRESS_STAGE_TRAIN = 'Training'
A string that indicates that the trainer loop is presently in training mode.

PROGRESS_STAGE_VAL = 'Validation'
A string that indicates that the trainer loop is presently in validation mode.

is_disabled

Return type bool

is_enabled

Return type bool

refresh_rate

Return type int

Methods Documentation

disable()
You should provide a way to disable the progress bar.

The Trainer will call this to disable the output on processes that have a rank different from 0, e.g., in multi-node training.

Return type None

enable()
You should provide a way to enable the progress bar.

The Trainer will call this in e.g. pre-training routines like the learning rate finder to temporarily enable and disable the main progress bar.

Return type None

on_predict_batch_end(**args*, ***kwargs*)
Called when the predict batch ends.

Return type None

on_predict_epoch_start(*trainer*, *pl_module*)
Called when the predict epoch begins.

Return type None

on_test_batch_end(**args*, ***kwargs*)
Called when the test batch ends.

Return type None

on_test_epoch_start(*trainer*, *pl_module*)
Called when the test epoch begins.

Return type None

on_train_batch_end(**args*, ***kwargs*)
Called when the train batch ends.

Return type None

on_train_epoch_start(*trainer*, *pl_module*)
Called when the train epoch begins.

Return type None

on_validation_batch_end(**args*, ***kwargs*)
Called when the validation batch ends.

Return type None

on_validation_start(*trainer*, *pl_module*)
Called when the validation loop begins.

Return type None

start_stage(*stage*, *total_num_batches*)
Sets the information that a new stage of the PL loop is starting. The stage will be available in self.stage, total_num_batches in self.total_num_batches. The time when this method was called is recorded in self.stage_start_time

Parameters

- **stage** (str) – The string name of the stage that has just started.
- **total_num_batches** (int) – The total number of batches that need to be processed in this stage. This is used only for progress reporting.

Return type None

update_progress(*batches_processed*)
Writes progress information once the refresh interval is full.

Parameters **batches_processed** (int) – The number of batches that have been processed for the current stage.

Return type None

15.2.3 BatchTimeCallback

class health_ml.utils.**BatchTimeCallback**(*max_batch_load_time_seconds=0.5*, *max_load_time_warnings=3*, *max_load_time_epochs=5*)
Bases: pytorch_lightning.callbacks.base.Callback

This callback provides tools to measure batch loading time and other diagnostic information. It prints alerts to the console or to *logging* if the batch loading time is over a threshold for several epochs. Metrics for loading time, as well as epoch time, and maximum and average batch processing time are logged to the loggers that are set up on the module. In distributed training, all logging to the console and to the Lightning loggers will only happen on global rank 0.

The loading time for a minibatch is estimated by the difference between the start time of a minibatch and the end time of the previous minibatch. It will consequently also include other operations that happen between the end of a batch and the start of the next one. For example, computationally expensive callbacks could also drive up this time.

Usage example:

```
>>> from health_ml.utils import BatchTimeCallback
>>> from pytorch_lightning import Trainer
>>> batchtime = BatchTimeCallback(max_batch_load_time_seconds=0.5)
>>> trainer = Trainer(callbacks=[batchtime])
```

Parameters

- **max_batch_load_time_seconds** (float) – The maximum expected loading time for a minibatch (given in seconds). If the loading time exceeds this threshold, a warning is printed. The maximum number of such warnings is controlled by the other arguments.
- **max_load_time_warnings** (int) – The maximum number of warnings about increased loading time that will be printed per epoch. For example, if max_load_time_warnings=3, at most 3 of these warnings will be printed within an epoch. The 4th minibatch with loading time over the threshold would not generate any warning anymore. If set to 0, no warnings are printed at all.
- **max_load_time_epochs** (int) – The maximum number of epochs where warnings about the loading time are printed. For example, if max_load_time_epochs=2, and at least 1 batch with increased loading time is observed in epochs 0 and 3, no further warnings about increased loading time would be printed from epoch 4 onwards.

Attributes Summary

BATCH_TIME	The name that is used to log the execution time per batch.
EPOCH_TIME	The name that is used to log the execution time per epoch
EXCESS_LOADING_TIME	The name that is used to log the time spent loading all the batches that exceeding the loading time threshold.
METRICS_PREFIX	The prefix for all metrics collected by this callback.
TRAIN_PREFIX	The prefix for all metrics collected during training.
VAL_PREFIX	The prefix for all metrics collected during validation.

Methods Summary

batch_end(is_training)	Shared code to keep track of minibatch loading times.
batch_start(batch_idx, is_training)	Shared code to keep track of minibatch loading times.
get_timers(is_training)	Gets the object that holds all metrics and timers, for either the validation or the training epoch.
log_metric(name_suffix, value, is_training)	Write a metric given as a name/value pair to the currently trained module.
on_fit_start(trainer, pl_module)	This is called at the start of training.
on_train_batch_end(trainer, pl_module, ...)	Called when the train batch ends.
on_train_batch_start(trainer, pl_module, ...)	Called when the train batch begins.
on_train_epoch_start(trainer, pl_module)	Called when the train epoch begins.
on_validation_batch_end(trainer, pl_module, ...)	Called when the validation batch ends.
on_validation_batch_start(trainer, ...)	Called when the validation batch begins.

continues on next page

Table 7 – continued from previous page

on_validation_epoch_end(trainer, pl_module)	This is a hook called at the end of a training or validation epoch.
on_validation_epoch_start(trainer, pl_module)	Called when the val epoch begins.
write_and_log_epoch_time(is_training)	Reads the IO timers for either the training or validation epoch, writes them to the console, and logs the time per epoch.

Attributes Documentation

BATCH_TIME = 'batch_time [sec]'
The name that is used to log the execution time per batch.

EPOCH_TIME = 'epoch_time [sec]'
The name that is used to log the execution time per epoch

EXCESS_LOADING_TIME = 'batch_loading_over_threshold [sec]'
The name that is used to log the time spent loading all the batches that exceeding the loading time threshold.

METRICS_PREFIX = 'timing/'
The prefix for all metrics collected by this callback.

TRAIN_PREFIX = 'train/'
The prefix for all metrics collected during training.

VAL_PREFIX = 'val/'
The prefix for all metrics collected during validation.

Methods Documentation

batch_end(*is_training*)
Shared code to keep track of minibatch loading times. This is only done on global rank zero.

Parameters **is_training** (bool) – If true, this has been called from *on_train_batch_end*, otherwise it has been called from *on_validation_batch_end*.

Return type None

batch_start(*batch_idx*, *is_training*)
Shared code to keep track of minibatch loading times. This is only done on global rank zero.

Parameters

- **batch_idx** (int) – The index of the current minibatch.
- **is_training** (bool) – If true, this has been called from *on_train_batch_start*, otherwise it has been called from *on_validation_batch_start*.

Return type None

get_timers(*is_training*)
Gets the object that holds all metrics and timers, for either the validation or the training epoch.

Return type EpochTimers

log_metric(*name_suffix*, *value*, *is_training*, *reduce_max=False*)
Write a metric given as a name/value pair to the currently trained module. The full name of the metric is composed of a fixed prefix "timing/", followed by either "train/" or "val/", and then the given suffix.

Parameters

- **name_suffix** (str) – The suffix for the logged metric name.
- **value** (float) – The value to log.
- **is_training** (bool) – If True, use "train/" in the metric name, otherwise "val/"
- **reduce_max** (bool) – If True, use torch.max as the aggregation function for the logged values. If False, use torch.mean

Return type None

on_fit_start(*trainer*, *pl_module*)
This is called at the start of training. It stores the model that is being trained, because it will be used later to log values.

Return type None

on_train_batch_end(*trainer*, *pl_module*, *outputs*, *batch*, *batch_idx*, *dataloader_idx*)
Called when the train batch ends.

Return type None

on_train_batch_start(*trainer*, *pl_module*, *batch*, *batch_idx*, *dataloader_idx*)
Called when the train batch begins.

Return type None

on_train_epoch_start(*trainer*, *pl_module*)
Called when the train epoch begins.

Return type None

on_validation_batch_end(*trainer*, *pl_module*, *outputs*, *batch*, *batch_idx*, *dataloader_idx*)
Called when the validation batch ends.

Return type None

on_validation_batch_start(*trainer*, *pl_module*, *batch*, *batch_idx*, *dataloader_idx*)
Called when the validation batch begins.

Return type None

on_validation_epoch_end(*trainer*, *pl_module*)
This is a hook called at the end of a training or validation epoch. In here, we can still write metrics to a logger.

Return type None

on_validation_epoch_start(*trainer*, *pl_module*)
Called when the val epoch begins.

Return type None

write_and_log_epoch_time(*is_training*)
Reads the IO timers for either the training or validation epoch, writes them to the console, and logs the time per epoch.

Parameters **is_training** (bool) – If True, show and log the data for the training epoch. If False, use the data for the validation epoch.

Return type None

CHAPTER

SIXTEEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX

O

P

R

S

T

U

V

W