hi-ml

InnerEye

Jul 28, 2023

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

To run your code in the cloud, you need to have an AzureML workspace in your Azure subscription. Please follow the *instructions here* to create an AzureML workspace if you don't have one yet.

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__)
```

(continues on next page)

(continued from previous page)

```
    run_info = submit_to_azure_if_needed(compute_cluster_name="preprocess-ds12",
                                         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

SETTING UP AZURE

If you already have an AzureML workspace available, you can go straight to *the last step*.

To set up all your Azure resources, you need to have:

- A subscription to Azure.
- An account that has “Owner” permissions to the Azure subscription.

There are two ways to set up all necessary resources, either via the Azure portal or via the Azure Command-line Interface (CLI). We recommend the CLI because all necessary resources can be easily created via a single script.

2.1 Creating an AzureML workspace via the Azure Portal

If you prefer to create your workspace via the web UI on the Azure Portal, please follow the steps below.

- Create a resource group.
- Create a storage account for datasets.
- Create an AzureML workspace, compute instances and compute clusters.

2.2 Creating an AzureML workspace via the Azure Command-line Tools

A pureley command-line driven setup is possible via the Azure Command-line Tools. These tools are available for multiple platforms, including Linux, Mac, and Windows.

After downloading the command-line tools, you can run the following command add the `ml` extension that is required to create an AzureML workspace:

```
az extension add --name ml
```

2.2.1 Documentation

- Creating AzureML workspaces via the CLI.
- Creating Azure storage accounts via the CLI.
- Schema for creating datastores.
- Creating datastores via the CLI.
- Create a shared access signature.

2.2.2 Collecting the necessary information

Find out which Azure data centre locations you can use:

```
az account list-locations -o table
```

You will need the location names (second column in the table) to create resources in the right geographical regions. Choosing the right region can be particularly important if your data governance requires the data to be processed inside certain geographical boundaries.

For the storage account, please choose an SKU from based on your needs as described here. Most likely, `Standard_LRS` will be the right SKU for you.

2.2.3 Creating resources

The script below will create

- an AzureML workspace.
- a storage account to hold the datasets, with a container called `datasets`.
- a datastore that links the AzureML workspace to the storage account.
- a resource group for all of the above items.

In the script, you will need to replace the values of the following variables:

- `location` - the location of the Azure datacenter you want to use.
- `prefix` - the prefix you want to use for the resources you create. This will also be the name of the AzureML workspace.

```
export location=uksouth     # The Azure location where the resources should be created
export prefix=himl          # The name of the AzureML workspace. This is also the prefix␣
↪for all other resources.
export container=datasets
export datastorefile=datastore.yaml
az group create \
    --name ${prefix}rg \
    --location ${location}
az storage account create \
    --name ${prefix}data \
    --resource-group ${prefix}rg \
    --location ${location} \
    --sku Standard_LRS
az storage container create \
```

(continues on next page)

(continued from previous page)

```
    --account-name ${prefix}data \
    --name ${container} \
    --auth-mode
az ml workspace create \
    --resource-group ${prefix}rg \
    --name ${prefix} \
    --location ${location}
key=$(az storage account keys list --resource-group ${prefix}rg --account-name ${prefix}
↪data --query [0].value -o tsv)
cat >${datastorefile} <<EOL
\$schema: https://azuremlschemas.azureedge.net/latest/azureBlob.schema.json
name: datasets
type: azure_blob
description: Pointing to the `${container}` container in the ${prefix}data storage␣
↪account.
account_name: ${prefix}data
container_name: ${container}
credentials:
  account_key: ${key}
EOL
az ml datastore create --file ${datastorefile} --resource-group ${prefix}rg --workspace-
↪name ${prefix}
rm ${datastorefile}
```

Note that the datastore will use the storage account key to authenticate. If you want to use Shared Access Signature (SAS) instead, replace the creation of the datastore config file in the above script with the following command:

```
key=$(az storage container generate-sas --account-name ${prefix}data --name ${container}␣
↪--permissions acdlrw --https-only --expiry 2024-01-01 -o tsv)
cat >${datastorefile} <<EOL
\$schema: https://azuremlschemas.azureedge.net/latest/azureBlob.schema.json
name: ${name}
type: azure_blob
description: Pointing to the `${container}` container in the ${prefix}data storage␣
↪account.
account_name: ${prefix}data
container_name: ${container}
credentials:
  sas_token: ${key}
EOL
```

You can adjust the expiry date of the SAS token and the permissions of the SAS token (full read/write permission in the script above). For further options, run `az storage container generate-sas --help`

2.2.4 Creating compute clusters and permissions

Now that you have created the core AzureML workspace, you need to create a compute cluster.

To adjust permissions, find the AzureML workspace that you just created in the Azure Portal. Add yourself and your team members with "Contributor" permissions to the workspace, following the guidelines here.

2.3 Accessing the workspace

The hi-ml toolbox relies on a workspace configuration file called config.json to access the right AzureML workspace. This file can be downloaded from the UI of the workspace. It needs to be placed either in your copy of the hi-ml repository, or in your repository that uses the hi-ml package.

- In the browser, navigate to the AzureML workspace that you want to use for running your training job.
- 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. Copy that file to the root folder of your repository.

The file config.json should look like this:

```
{
  "subscription_id": "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx",
  "resource_group": "myresourcegroup",
  "workspace_name": "myworkspace"
}
```

As an alternative to keeping the config.json file in your repository, you can specify the necessary information in environment variables. The environment variables are:

- HIML_SUBSCRIPTION_ID: The subscription ID of the AzureML workspace, taken from the subscription_id field in the config.json file.
- HIML_RESOURCE_GROUP: The resource group of the AzureML workspace, taken from the resource_group field in the config.json file.
- HIML_WORKSPACE_NAME: The name of the AzureML workspace, taken from the workspace_name field in the config.json file.

When accessing the workspace, the hi-ml toolbox will first look for the config.json file. If it is not found, it will fall back to the environment variables. For details, see the documentation of the get_workspace function in readthedocs.

CHAPTER

THREE

CONNECTING TO AZURE

3.1 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.

3.2 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.

3.2.1 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.

3.2.2 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.

3.2.3 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

FOUR

DATASETS

4.1 Key concepts

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

4.1.1 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.

4.1.2 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.

4.1.3 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.

4.1.4 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.

4.2 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:

```python
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.

4.2.1 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:

```python
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.

4.2.2 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]
```

4.2.3 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 specify 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, if `local_folder` is not specified, then the dataset will either be downloaded or mounted to a temporary folder locally, depending on the `use_mounting` flag. The path to it will be available in `run_info` as above.

```
input_folder = run_info.input_datasets[0]
```

Note that mounting the dataset locally is only supported on Linux because it requires the use of the native package libfuse, which must first be installed. Also, if running in a Docker container, it must be started with additional arguments. For more details see here: azureml.data.filedataset.mount.

4.2.4 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]
```

This is also true when running locally - if local_folder is not specified and an AzureML workspace can be found, then the dataset will be downloaded or mounted to the target_folder.

4.2.5 Overwriting existing output datasets

When creating an output dataset with the same name as an existing dataset, the default behaviour of hi-ml is to overwrite the existing datasets. This is as if a run fails during the upload stage, corrupt files may be created. Allowing overwriting means that these corrupt datasets will not cause errors. If you wish to disable this behaviour, it can be controlled using the overwrite_existing parameter (only available in sdk v1, hence setting strictly_aml_v1=True):

```
from health_azure import DatasetConfig, submit_to_azure_if_needed
output_dataset = DatasetConfig(name="my_folder",
                               datastore="my_datastore",
                               overwrite_existing=False,
                              )

# fails if output dataset already exists:
run_info = submit_to_azure_if_needed(...,
                                     output_datasets=[output_dataset],
                                     strictly_aml_v1=True,
                                    )

```

4.2.6 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,
```

(continues on next page)

(continued from previous page)

```
                              )
run_info = submit_to_azure_if_needed(...,
                                     input_datasets=[input_dataset],
                                    )
input_folder = run_info.input_datasets[0]
```

CHAPTER

FIVE

HYPERPARAMETER SEARCH VIA HYPERDRIVE (AML SDK V1)

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

SIX

HYPERPARAMETER SEARCH IN AML SDK V2

There is no concept of a HyperDriveConfig in AML SDK v2. Instead, hyperparameter search arguments are passed into a command, and then the 'sweep' method is called AML docs. To specify a hyperparameter search job you must specify the method `get_parameter_tuning_args` in your Container. This should return a dictionary of the arguments to be passed in to the command. For example:

```
    def get_parameter_tuning_args(self) -> Dict[str, Any]:
        from azure.ai.ml.entities import Choice
        from health_azure.himl import (MAX_TOTAL_TRIALS_ARG, PARAM_SAMPLING_ARG,␣
↪SAMPLING_ALGORITHM_ARG,
                                       PRIMARY_METRIC_ARG, GOAL_ARG)

        values = [0.1, 0.5, 0.9]
        argument_name = "learning_rate"
        param_sampling = {argument_name: Choice(values)}
        metric_name = "val/loss"

        hparam_args = {
            MAX_TOTAL_TRIALS_ARG: len(values),
            PARAM_SAMPLING_ARG: param_sampling,
            SAMPLING_ALGORITHM_ARG: "grid",
            PRIMARY_METRIC_ARG: metric_name,
            GOAL_ARG: "Minimize"

        }
        return hparam_args
```

Additional parameters, sampling strategies, limits etc. are described in the link above. Note that each job that is created will receive an additional command line argument <`argument_name`> and it is your job to update the script to be able to parse and use this argument.

CHAPTER

SEVEN

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.

7.1 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.

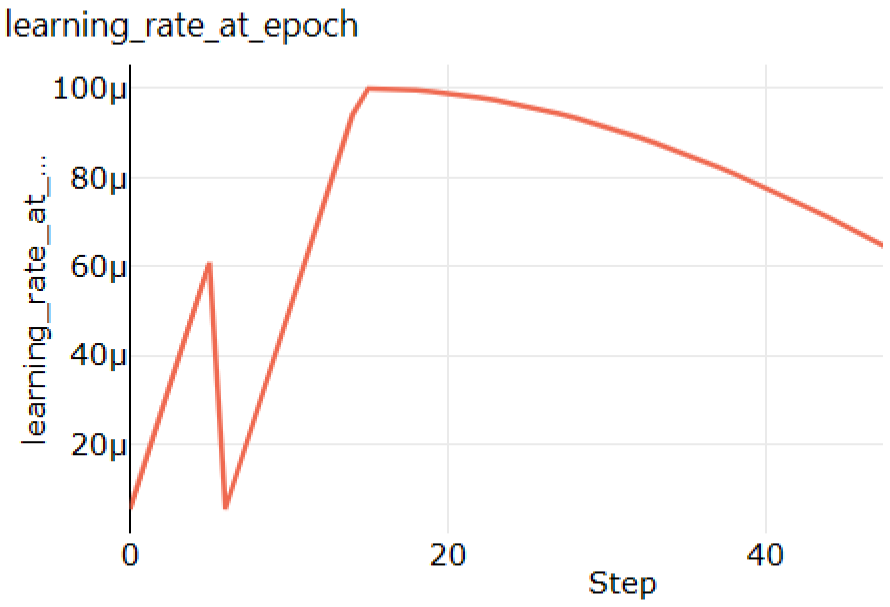
For details on pricing, check this Azure price calculator, choose "Category: GPU". The price for low priority VMs is given in the "Spot" column

7.2 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.

7.3 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.

7.3.1 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())
```

7.4 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

EIGHT

COMMANDLINE TOOLS

8.1 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.

8.2 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`.

8.3 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.

8.3.1 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)

8.3.2 Example:

```python
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

NINE

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.

9.1 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.

9.2 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.

9.3 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.

9.4 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

TEN

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.

10.1 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` if using the old AzureML runtime, or `std_log.txt` if using the new AzureML runtime, 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.

10.2 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.

10.3 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(
```

(continues on next page)

(continued from previous page)

```
        ...
        output_datasets=[DatasetConfig(name="himl_sample4_output", datastore=
↪"himldatasets")]
```

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*

10.4 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
```

10.4.1 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.

10.4.2 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*

10.4.3 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"],
```

10.5 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.

10.6 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`.

10.7 Training with k-fold cross validation in Azure ML

It is possible to create a parent run on Azure ML that is associated with one or more child runs (see here for further information.) This is useful in circumstances such as k-fold cross-validation, where individual child run perform validation on a different data split. When a HyperDriveRun is created in Azure ML, it follows this same principle and generates multiple child runs, associated with one parent.

To train with k-fold cross validation using `submit_to_azure_if_needed`, you must do two things.

1. Call the helper function `create_crossval_hyperdrive_config` to create an AML HyperDriveConfig object representing your parent run. It will have one child run for each of the k-fold splits you request, as follows

   ```
   from health_azure import create_crossval_hyperdrive_config

    hyperdrive_config = create_crossval_hyperdrive_config(num_splits,
                                                          cross_val_index_arg_
   ↪name=cross_val_index_arg_name,
                                                          metric_name=metric_name)
   ```

   where:

   - `num_splits` is the number of k-fold cross validation splits you require
   - `cross_val_index_arg_name` is the name of the argument given to each child run, whose value denotes which split that child represents (this parameter defaults to 'cross_validation_split_index', in which case, supposing you specified 2 cross validation splits, one would receive the arguments ['–cross_validation_split_index' '0'] and the other would receive ['–cross_validation_split_index' '1']]. It is up to you to then use these args to retrieve the correct split from your data.

  - `metrics_name` represents the name of a metric that you will compare your child runs by. **NOTE** the run will expect to find this metric, otherwise it will fail as described here You can log this metric in your training script as follows:

```
from azureml.core import Run

# Example of logging a metric called <metric_name> to an AML Run.
loss = <my_loss_calc>
run_log = Run.get_context()
run_log.log(metric_name, loss)
```

See the documentation here for further explanation.

2. The hyperdrive_config returned above must be passed into the function `submit_to_azure_if_needed` as follows:

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

This will create a parent (HyperDrive) Run, with `num_cross_validation_split` children - each one associated with a different data split.

10.8 Retrieving the aggregated results of a cross validation/ HyperDrive run

You can retrieve a Pandas DataFrame of the aggregated results from your cross validation run as follows:

```
from health_azure import aggregate_hyperdrive_metrics

df = aggregate_hyperdrive_metrics(run_id, child_run_arg_name)
```

where:

- `run_id` is a string representing the id of your HyperDriveRun. Note that this **must** be an instance of an AML HyperDriveRun.
- `child_run_arg_name` is a string representing the name of the argument given to each child run to denote its position relative to other child runs (e.g. this arg could equal 'child_run_index' - then each of your child runs should expect to receive the arg '–child_run_index' with a value <= the total number of child runs)

If your HyperDrive run has 2 children, each logging the metrics epoch, accuracy and loss, the result would look like this:

```
|              | 0               | 1                  |
|--------------|-----------------|--------------------|
| epoch        | [1, 2, 3]       | [1, 2, 3]          |
| accuracy     | [0.7, 0.8, 0.9] | [0.71, 0.82, 0.91] |
| loss         | [0.5, 0.4, 0.3] | [0.45, 0.37, 0.29] |
```

here each column is one of the splits/ child runs, and each row is one of the metrics you have logged to the run.

It is possible to log rows and tables in Azure ML by calling run.log_table and run.log_row respectively. In this case, the DataFrame will contain a Dictionary entry instead of a list, where the keys are the table columns (or keywords provided to log_row), and the values are the table values. e.g.

```
|                 | 0                                          | 1                                          
↪                |
|-----------------|--------------------------------------------|---------------------------
↪---------------|
| accuracy_table |{'epoch': [1, 2], 'accuracy': [0.7, 0.8]} | {'epoch': [1, 2], 'accuracy
↪': [0.8, 0.9]} |
```

It is also posisble to log plots in Azure ML by calling run.log_image and passing in a matplotlib plot. In this case, the DataFrame will contain a string representing the path to the artifact that is generated by AML (the saved plot in the Logs & Outputs pane of your run on the AML portal). E.g.

```
|                 | 0                                        | 1                                       
↪          |
|-----------------|------------------------------------------|-----------------------------
↪----------|
| accuracy_plot  | aml://artifactId/ExperimentRun/dcid.... | aml://artifactId/
↪ExperimentRun/dcid...|
```

10.9 Modifying checkpoints stored in an AzureML run

The script in examples/modify_checkpoint/modify_checkpoint.py shows how checkpoints can be downloaded from an AzureML run, modified, and the uploaded back to a newly created run.

This can be helpful for example if networks architecture changed, but you do not want to re-train the stored models with the new code.

The essential bits are:

- Download files from a run via `download_files_from_run_id`
- Modify the checkpoints
- Create a new run via `create_aml_run_object`
- Then use `Run.upload_folder` to upload all modified checkpoints to that new run. From there, they can be consumed in a follow-up training run again via `download_files_from_run_id`

CHAPTER

ELEVEN

SUBMITTING JOBS TO SINGULARITY USING AMULET

PLEASE NOTE: Amulet is only intended for those with access to internal Microsoft compute resources. To access Amulet you must have an identity associated with the Microsoft tenant. This means you must be a Microsoft employee or approved external user.

The documentation below describes how to submit `hi-ml` jobs via Amulet. In addition, we also provide a *minimal sample script* that shows the essential parts of a trainer script and how it interacts with Amulet.

11.1 Install Amulet

This package is not included in the hi-ml environment definition, since these instructions only apply to users associated with the Microsoft tenant. See the instructions for installing.

11.2 Create an Azure ML Storage Account

As stated in the Amulet docs, a Storage Account is required for storing information about your experiments, outputs of jobs etc. The Storage Account must be of type Storage V2. See the docs for steps on setting up.

11.3 Add your Singularity Workspace

```
amlt workspace add WORKSPACE_NAME --subscription SUBSCRIPTION_ID --resource-group␣
↪RESOURCE_GROUP
amlt workspace set-default VC_NAME WORKSPACE_NAME
```

11.4 Create or checkout a project

As stated in the docs, an Amulet project "usually corresponds to a single research endeavor, e.g. a publication". Projects contain experiments which contain jobs. To create a project:

```
amlt project create <your-project-name> <storage-account-name>
```

To manage existing projects, use:

```
amlt project {list|checkout|remove}
```

11.5 Create a configuration file

A configuration (yaml) file is required to specify your job. For example, to run the HelloWorld model via the hi-ml runner:

```yaml
description: Hello World on Singularity

environment:
  image: azureml/openmpi3.1.2-cuda10.2-cudnn7-ubuntu18.04:latest
  conda_yaml_file: $CONFIG_DIR/hi-ml/supercomputer_environment.yml

code:
  # local directory of the code. this will be uploaded to the server.
  # $CONFIG_DIR is expanded to the directory of this config file
  local_dir: $CONFIG_DIR

# list of jobs to run
jobs:
- name: HelloWorld
  sku: G1
  command:
  - python hi-ml/src/health_ml/runner.py --model=health_cpath.TilesPandaImageNetMIL --
↪tune_encoder --batch_size=2
```

Note that SKU here refers to the number of GPUs/CPUs to reserve, and its memory. In this case we have specified 1 GPU. For other options, see the docs.

You can specify multiple jobs here. There are a variety of arguments for controlling factors such as `sla_tier`, whether the job is `preemptible`, the job `priority` and more. For full details see the docs

11.6 Submit the job to Singularity

```
amlt run <path-to-config> <experiment-name> -t <target-cluster>
```

11.7 To run a specific job from a config

If you have multiple jobs specified in your config file, it is possible to submit just one of them as follows:

```
amlt run <path-to-config> <experiment-name> :<job_name> -t <target-cluster>
```

11.8 Running a distributed job

There are multiple ways to distribute a job with Amulet. The recommended way is to add the following section to your config file

```
env_defaults:
  NODES: 1
  GPUS: 8
  MEM: 32
```

Then you should update your job definition as follows:

```
jobs:
- name: <job name>
  sku: ${NODES}x${MEM}G${GPUS}
  command:
  - python <script> <args>
  process_count_per_node: ${GPUS}
```

Additional settings and other methods for distributing can be found here.

For training jobs using PyTorch Lightning, the field `process_count_per_node` can be set to 0 or omitted altogether. This indicates to Amulet that the user is responsible for spawning the additional processes. This is the case for PyTorch Lightning, which will later spawn 1 process per GPU.

11.9 View your job

Once your job is running, you can view it in the Azure ML UI. Alternatively, you can check on the status using the `amlt` CLI as follows:

```
amlt status <exp-name> :<job-name-1>
```

Similarly, to get the STDOUT from your job, run

```
amlt logs <exp-name> :<job-name-1>
```

To view all of your runs in one place, run:

```
amlt browse
```

This will launch a Flask app which allows you to view the runs within your project

11.10 Download the /logs outputs of your job

Amulet provides a simple way to download the logs from your job:

```
amlt results <exp-name> :<job-name-1>
```

CHAPTER TWELVE

EXAMPLE TRAINER SCRIPT FOR AMULET

The following example shows a simple PyTorch Lightning trainer script that makes use of the Amulet environment.

You will need a folder that contains the following files:

- A Conda environment definition, like this.
- An Amulet configuration file, like this. The configuration file should live in your repository's root folder, and to the Conda environment definition.
- A script that uses the Amulet environment, like this

The script has a large number of comments around the correct use of the Amulet environment - please read them carefully if you want to base your training code on that example.

To submit this example script as-is via Amulet, follow these steps:

- Check out the hi-ml repository.
- Follow the onboarding instructions in the *Amulet overview* section, and create an Amulet project in the root folder of the repository.
- Modify `docs/source/amulet/config.yml` to point to your storage account: replace the `<storage_account_name>` placeholder with the name of your storage account.
- Once Amulet is installed, submit the example jobs with the following command:

```
amlt run docs/source/amulet/config.yml <experiment_name> -t <name_of_your_compute_target>
```

CHAPTER

THIRTEEN

LOGGING METRICS WHEN TRAINING MODELS IN AND OUTSIDE 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.

13.1 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.

13.2 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 import Trainer
from pytorch_lightning.loggers import TensorBoardLogger

tb_logger = TensorBoardLogger("logs/")
azureml_logger = AzureMLLogger(enable_logging_outside_azure_ml=False)
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`.

13.3 Logging to AzureML when running outside AzureML

You may still see the need to run some of your training jobs on an individual VM, for example small jobs or for debugging. Keeping track of the results in those runs can be tricky, and comparing or sharing them even more.

All results that you achieve in such runs outside AzureML can be written straight into AzureML using the `AzureMLLogger`. Its behaviour is as follows:

- When instantiated inside a run in AzureML, it will write metrics straight to the present run.
- When instantiated outside an AzureML run, it will create a new `Run` object that writes its metrics straight through to AzureML, even though the code is not running in AzureML.

This behaviour is controlled by the `enable_logging_outside_azure_ml` argument. With the following code snippet, you can to use the `AzureMLLogger` to write metrics to AzureML when the code is inside or outside AzureML:

```
from health_ml.utils import AzureMLLogger
from pytorch_lightning.loggers import TensorBoardLogger
from pytorch_lightning import Trainer

tb_logger = TensorBoardLogger("logs/")
azureml_logger = AzureMLLogger(enable_logging_outside_azure_ml=True)
trainer = Trainer(logger=[tb_logger, azureml_logger])
```

If this is executed on a VM outside an AzureML run, you will see additional information printed to the console like this:

```
Writing metrics to run ed52cfac-1b85-42ea-8ebe-2f90de21be6b in experiment azureml_logger.
To check progress, visit this URL: https://ml.azure.com/runs/ed52cfac-1b85-42ea-8ebe-
↪2f90de21be...
```

Clicking on the URL will take you to the AzureML web page, where you can inspect the metrics that the run has written so far.

There are a few points that you should note:

Experiments: Each run in AzureML is associated with an experiment. When executed in an AzureML run, the `AzureMLLogger` will know which experiment to write to. Outside AzureML, on your VM, the logger will default to using an experiment called `azureml-logger`. This means that runs inside and outside AzureML end up in different experiments. You can customize this like in the following code snippet, so that the submitted runs and the runs outside AzureML end up in the same experiment:

```
from health_azure import submit_to_azure_if_needed
from health_ml.utils import AzureMLLogger
from pytorch_lightning import Trainer

experiment_name = "my_new_architecture"
submit_to_azure_if_needed(compute_cluster_name="nd24",
                          experiment_name=experiment_name)
azureml_logger = AzureMLLogger(enable_logging_outside_azure_ml=True, experiment_
↪name=experiment_name)
trainer = Trainer(logger=[azureml_logger])
```

Snapshots: The run that you are about the create can follow the usual pattern of AzureML runs, and can create a full snapshot of all code that was used in the experiment. This will greatly improve reproducibility of your experiments. By default, this behaviour is turned off, though. You can provide an additional argument to the logger, like

AzureMLLogger(snapshot_directory='/users/me/code/trainer') to include the given folder in the snapshot. In addition, you can place a file called .amlignore to exclude previous results, or large checkpoint files, from being included in the snapshot (see here for details)

13.4 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
```

13.4.1 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

FOURTEEN

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.

14.1 `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:

```python
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
...
```

14.2 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.

14.2.1 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.

14.2.2 Usage example

```python
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

FIFTEEN

RUNNING ML EXPERIMENTS WITH HI-ML

The hi-ml toolbox is capable of training any PyTorch Lighting (PL) model inside of AzureML, making use of these features:

- Training on a local GPU machine or inside of AzureML without code changes
- Working with different models in the same codebase, and selecting one by name
- Distributed training in AzureML
- Logging via AzureML's native capabilities
- Evaluation of the trained model on new datasets

This can be used by invoking the hi-ml runner and providing the name of the container class, like this: `himl-runner --model=MyContainer`.

There is a fully working example HelloContainer, that implements a simple 1-dimensional regression model from data stored in a CSV file. You can run that from the command line by `himl-runner --model=HelloWorld`.

15.1 Specifying the model to run

The `--model` argument specifies the name of a class that should be used for model training. The class needs to be a subclass of `LightningContainer`, see below. There are different ways of telling the runner where to find that class:

- If just providing a single class name, like `--model=HelloWorld`, the class is expected somewhere in the `health_ml.configs` namespace. It can be in any module/folder inside of that namespace.
- If the class is outside of the `health_ml.configs` (as would be normal if using the `himl-runner` from a package), you need to provide some "hints" where to start searching. It is enough to provide the start of the namespace string: for example, `--model health_cpath.PandaImageNetMIL` is effectively telling the runner to search for the `PandaImageNetMIL` class *anywhere* in the `health_cpath` namespace. You can think of this as `health_cpath.*.PandaImageNetMIL`

15.2 Running ML experiments in Azure ML

To train in AzureML, use the flag `--cluster` to specify the name of the cluster in your Workspace that you want to submit the job to. So the whole command would look like:

```
himl-runner --model=HelloWorld --cluster=my_cluster_name
```

You can also specify `--num_nodes` if you wish to distribute the model training.

When starting the runner, you need to do that from a directory that contains all the code that your experiment needs: The current working directory will be used as the root of all data that will be copied to AzureML to run your experiment. (the only exception to this rule is if you start the runner from within an enlistment of the HI-ML GitHub repository).

AzureML needs to know which Python/Conda environment it should use. For that, the runner needs a file `environment.yml` that contains a Conda environment definition. This file needs to be present either in the current working directory or one of its parents. To specify a Conda environment that is located elsewhere, you can use

```
himl-runner --model=HelloWorld --cluster=my_cluster_name --conda_env=/my/folder/to/
↪special_environment.yml
```

15.3 Setup - creating your model config file

In order to use these capabilities, you need to implement a class deriving from `health_ml.lightning_container.LightningContainer`. This class encapsulates everything that is needed for training with PyTorch Lightning:

For example:

```
class MyContainer(LightningContainer):
   def __init__(self):
       super().__init__()
       self.azure_datasets = ["folder_name_in_azure_blob_storage"]
       self.local_datasets = [Path("/some/local/path")]
       self.max_epochs = 42

   def create_model(self) -> LightningModule:
       return MyLightningModel()

   def get_data_module(self) -> LightningDataModule:
       return MyDataModule(root_path=self.local_dataset)
```

The `create_model` method needs to return a subclass of PyTorch Lightning's LightningModule, that has all the usual PyTorch Lightning methods required for training, like the `training_step` and `forward` methods. E.g:

```
class MyLightningModel(LightningModule):
    def __init__(self):
        self.layer = ...
    def training_step(self, *args, **kwargs):
        ...
    def forward(self, *args, **kwargs):
        ...
    def configure_optimizers(self):
        ...
    def test_step(self, *args, **kwargs):
        ...
```

The `get_data_module` method of the container needs to return a DataModule (inheriting from a PyTorch Lightning DataModule) which contains all of the logic for downloading, preparing and splitting your dataset, as well as methods for wrapping the train, val and test datasets respectively with DataLoaders. E.g:

```
class MyDataModule(LightningDataModule):
    def __init__(self, root_path: Path):
        # All data should be read from the folder given in self.root_path
        self.root_path = root_path
    def train_dataloader(self, *args, **kwargs) -> DataLoader:
        # The data should be read off self.root_path
        train_dataset = ...
        return DataLoader(train_dataset, batch_size=5, num_workers=5)
    def val_dataloader(self, *args, **kwargs) -> DataLoader:
        # The data should be read off self.root_path
        val_dataset = ...
        return DataLoader(val_dataset, batch_size=5, num_workers=5)
    def test_dataloader(self, *args, **kwargs) -> DataLoader:
        # The data should be read off self.root_path
        test_dataset = ...
        return DataLoader(test_dataset, batch_size=5, num_workers=5)
```

So, the **full file** would look like:

```
from pathlib import Path
from torch.utils.data import DataLoader
from pytorch_lightning import LightningModule, LightningDataModule
from health_ml.lightning_container import LightningContainer

class MyLightningModel(LightningModule):
    def __init__(self):
        self.layer = ...
    def training_step(self, *args, **kwargs):
        ...
    def forward(self, *args, **kwargs):
        ...
    def configure_optimizers(self):
        ...
    def test_step(self, *args, **kwargs):
        ...

class MyDataModule(LightningDataModule):
    def __init__(self, root_path: Path):
        # All data should be read from the folder given in self.root_path
        self.root_path = root_path
    def train_dataloader(self, *args, **kwargs) -> DataLoader:
        # The data should be read off self.root_path
        train_dataset = ...
        return DataLoader(train_dataset, batch_size=5, num_workers=5)
    def val_dataloader(self, *args, **kwargs) -> DataLoader:
        # The data should be read off self.root_path
        val_dataset = ...
        return DataLoader(val_dataset, batch_size=5, num_workers=5)
    def test_dataloader(self, *args, **kwargs) -> DataLoader:
        # The data should be read off self.root_path
        test_dataset = ...
        return DataLoader(test_dataset, batch_size=5, num_workers=5)
```

(continues on next page)

(continued from previous page)

```python
class MyContainer(LightningContainer):
    def __init__(self):
        super().__init__()
        self.azure_datasets = ["folder_name_in_azure_blob_storage"]
        self.local_datasets = [Path("/some/local/path")]
        self.max_epochs = 42

    def create_model(self) -> LightningModule:
        return MyLightningModel()

    def get_data_module(self) -> LightningDataModule:
        return MyDataModule(root_path=self.local_dataset)
```

By default, config files will be looked for in the folder "health_ml.configs". To specify config files that live elsewhere, use a fully qualified name for the parameter `--model` - e.g. "MyModule.Configs.my_config.py"

15.4 Outputting files during training

The Lightning model returned by `create_model` needs to write its output files to the current working directory. When running inside of AzureML, the output folders will be directly under the project root. If not running inside AzureML, a folder with a timestamp will be created for all outputs and logs.

When running in AzureML, the folder structure will be set up such that all files written to the current working directory are later uploaded to Azure blob storage at the end of the AzureML job. The files will also be later available via the AzureML UI.

15.5 Trainer arguments

All arguments that control the PyTorch Lightning `Trainer` object are defined in the class `TrainerParams`. A `LightningContainer` object inherits from this class. The most essential one is the `max_epochs` field, which controls the `max_epochs` argument of the `Trainer`.

For example:

```python
from pytorch_lightning import LightningModule, LightningDataModule
from health_ml.lightning_container import LightningContainer

class MyContainer(LightningContainer):
    def __init__(self):
        super().__init__()
        self.max_epochs = 42

    def create_model(self) -> LightningModule:
        return MyLightningModel()

    def get_data_module(self) -> LightningDataModule:
        return MyDataModule(root_path=self.local_dataset)
```

15.5.1 Optimizer and LR scheduler arguments

To the optimizer and LR scheduler: the Lightning model returned by `create_model` should define its own `configure_optimizers` method, with the same signature as `LightningModule.configure_optimizers`, and returns a tuple containing the Optimizer and LRScheduler objects

15.6 Run inference with a pretrained model

You can use the hi-ml-runner in inference mode only by switching the `--run_inference_only` flag on and specifying the model weights by setting `--src_checkpoint` argument. With this flag, the model will be evaluated on the test set only. There is also an option for evaluating the model an a full dataset, described further below.

15.6.1 Specifying the checkpoint to use

When running inference on a trained model, you need to provide a model checkpoint that should be used. This is done via the `--src_checkpoint` argument. This supports three types of checkpoints:

- A local path where the checkpoint is stored `--src_checkpoint=local/path/to/my_checkpoint/model.ckpt`
- A remote URL from where to download the weights `--src_checkpoint=https://my_checkpoint_url.com/model.ckpt`
- An AzureML run id where checkpoints are saved in `outputs/checkpoints`. For this specific use case, you can experiment with different checkpoints by setting `--src_checkpoint` according to the format `<azureml_run_id>:<optional/custom/path/to/checkpoints/><filename.ckpt>`. If no custom path is provided (e.g., `--src_checkpoint=AzureML_run_id:best.ckpt`), we assume the checkpoints to be saved in the default checkpoints folder `outputs/checkpoints`. If no filename is provided (e.g., `--src_checkpoint=AzureML_run_id`), the last epoch checkpoint `outputs/checkpoints/last.ckpt` will be loaded.

Refer to *Checkpoints Utils* for more details on how checkpoints are parsed.

15.6.2 Running inference on the test set

When supplying the flag `--run_inference_only` on the commandline, no model training will be run, and only inference on the test set will be done:

- The model weights will be loaded from the location specified by `--src_checkpoint`
- A PyTorch Lightining `Trainer` object will be instantiated.
- The test set will be read out from the data module specified by the `get_data_module` method of the `LightningContainer` object.
- The model will be evaluated on the test set, by running `trainer.test`. Any special logic to use during the test step will need to be added to the model's `test_step` method.

Running the following command line will run inference using the `MyContainer` model with weights from the checkpoint saved in the AzureMl run `MyContainer_XXXX_yyyy` at the best validation loss epoch `/outputs/checkpoints/best_val_loss.ckpt`.

```
himl-runner --model=Mycontainer --run_inference_only --src_checkpoint=MyContainer_XXXX_
↪yyyy:best_val_loss.ckpt
```

15.6.3 Running inference on a full dataset

When supplying the flag `--mode=eval_full` on the commandline, no model training will be run, and the model will be evaluated on a dataset different from the training/validation/test dataset. This dataset is loaded via the `get_eval_data_module` method of the container.

- The model weights will be loaded from the location specified by `--src_checkpoint`
- A PyTorch Lightning `Trainer` object will be instantiated.
- The test set will be read out from the data module specified by the `get_eval_data_module` method of the `LightningContainer` object. The data module itself can read data from a mounted Azure dataset, which will be made availabe for the container at the path `self.local_datasets`. In a typical use-case, all the data in that dataset will be put into the `test_dataloader` field of the data module.
- The model will be evaluated on the test set, by running `trainer.test`. Any special logic to use during the test step will need to added to the model's `test_step` method.

Running the following command line will run inference using the `MyContainer` model with weights from the checkpoint saved in the AzureMl run `MyContainer_XXXX_yyyy` at the best validation loss epoch `/outputs/checkpoints/best_val_loss.ckpt`.

```
himl-runner --model=Mycontainer --src_checkpoint=MyContainer_XXXX_yyyy:best_val_loss.
↪ckpt --mode=eval_full --azure_datasets=my_new_dataset
```

The example code snippet here shows how to add a method that reads the inference dataset. In this example, we assume that the `MyDataModule` class has an argument `splits` that specifies the fraction of data to go into the training, validation, and test data loaders.

```
class MyContainer(LightningContainer):
    def __init__(self):
        super().__init__()
        self.azure_datasets = ["folder_name_in_azure_blob_storage"]
        self.local_datasets = [Path("/some/local/path")]
        self.max_epochs = 42

    def create_model(self) -> LightningModule:
        return MyLightningModel()

    def get_data_module(self) -> LightningDataModule:
        return MyDataModule(root_path=self.local_dataset, splits=(0.7, 0.2, 0.1))

    def get_eval_data_module(self) -> LightningDataModule:
        return MyDataModule(root_path=self.local_dataset, splits=(0.0, 0.0, 1.0))
```

15.7 Resume training from a given checkpoint

Analogously, one can resume training by setting `--src_checkpoint` and `--resume_training` to train a model longer. The pytorch lightning trainer will initialize the lightning module from the given checkpoint corresponding to the best validation loss epoch as set in the following comandline.

```
himl-runner --model=Mycontainer --cluster=my_cluster_name --src_checkpoint=MyContainer_
↪XXXX_yyyy:best_val_loss.ckpt --resume_training
```

Warning: When resuming training, one should make sure to set `container.max_epochs` greater than the last epoch of the specified checkpoint. A misconfiguration exception will be raised otherwise:

```
pytorch_lightning.utilities.exceptions.MisconfigurationException: You restored a␣
↪checkpoint with current_epoch=19, but you have set Trainer(max_epochs=4).
```

15.8 Logging to AzureML when running outside AzureML

The runner offers the ability to log metrics to AzureML, even if the present training is not running inside of AzureML. This adds an additional level of traceability for runs on GPU VMs, where there is otherwise no record of any past training.

You can trigger this behaviour by specifying the `--log_from_vm` flag. For the `HelloWorld` model, this will look like:

```
himl-runner --model=HelloWorld --log_from_vm
```

For logging to work, you need have a `config.json` file in the current working directory (or one of its parent folders) that specifies the AzureML workspace itself. When starting the runner, you will be asked to authenticate to AzureML.

There are two additional flags that can be used to control the logging behaviour:

- The `--experiment` flag sets which AzureML experiment to log to. By default, the experiment name will be the name of the model class (`HelloWorld` in the above example).
- The `--tag` flag sets the display name for the AzureML run. You can use that to give your run a memorable name, and later easily find it in the AzureML UI.

The following command will log to the experiment `my_experiment`, in a run that is labelled `my_first_run` in the UI:

```
himl-runner --model=HelloWorld --log_from_vm --experiment=my_experiment --tag=my_first_
↪run
```

15.9 Starting experiments with different seeds

To assess the variability of metrics, it is often useful to run the same experiment multiple times with different seeds. There is a built-in functionality of the runner to do this. When adding the commandline flag `--different_seeds=3`, your experiment will get run 3 times with seeds 0, 1 and 2. This is equivalent to starting the runner with arguments `--random_seed=0`, `--random_seed=1` and `--random_seed=2`.

These runs will be started in parallel in AzureML via the HyperDrive framework. It is not possible to run with different seeds on a local machine, other than by manually starting runs with `--random_seed=0` etc.

15.10 Common problems with running in AML

1. `"Your total snapshot size exceeds the limit <SNAPSHOT_LIMIT>"`. Cause: The size of your source directory is larger than the limit that AML sets for snapshots. Solution: check for cache files, log files or other files that are not necessary for running your experiment and add them to a `.amlignore` file in the root directory. Alternatively, you can see Azure ML documentation for instructions on increasing this limit, although it will make your jobs slower.
2. `"FileNotFoundError"`. Possible cause: Symlinked files. Azure ML SDK v2 will resolve the symlink and attempt to upload the resolved file. Solution: Remove symlinks from any files that should be uploaded to Azure ML.

CHAPTER

SIXTEEN

CHECKPOINT UTILS

Hi-ml toolbox offers different utilities to parse and download pretrained checkpoints that help you abstract checkpoint downloading from different sources. Refer to CheckpointParser for more details on the supported checkpoints format. Here's how you can use the checkpoint parser depending on the source:

- For a local path, simply pass it as shown below. The parser will further check if the provided path exists:

```
from health_ml.utils.checpoint_utils import CheckpointParser

download_dir = 'outputs/checkpoints'
checkpoint_parser = CheckpointParser(checkpoint='local/path/to/my_checkpoint/model.ckpt')
print('Checkpoint', checkpoint_parser.checkpoint, 'is a local file', checkpoint_parser.
↪is_local_file)
local_file = parser.get_or_download_checkpoint(download_dir)
```

- To download a checkpoint from a URL:

```
from health_ml.utils.checpoint_utils import CheckpointParser, MODEL_WEIGHTS_DIR_NAME

download_dir = 'outputs/checkpoints'
checkpoint_parser = CheckpointParser('https://my_checkpoint_url.com/model.ckpt')
print('Checkpoint', checkpoint_parser.checkpoint, 'is a URL', checkpoint_parser.is_url)
# will dowload the checkpoint to download_dir/MODEL_WEIGHTS_DIR_NAME
path_to_ckpt = checkpoint_parser.get_or_download_checkpoint(download_dir)
```

- Finally checkpoints from an Azure ML runs can be reused by providing an id in this format `<AzureML_run_id>:<optional/custom/path/to/checkpoints/><filename.ckpt>`. If no custom path is provided (e.g., `<AzureML_run_id>:<filename.ckpt>`) the checkpoint will be downloaded from the default checkpoint folder (e.g., `outputs/checkpoints`) If no filename is provided, (e.g., `src_checkpoint=<AzureML_run_id>`) the latest checkpoint will be downloaded (e.g., `last.ckpt`).

```
from health_ml.utils.checpoint_utils import CheckpointParser

checkpoint_parser = CheckpointParser('AzureML_run_id:best.ckpt')
print('Checkpoint', checkpoint_parser.checkpoint, 'is a AML run', checkpoint_parser.is_
↪aml_run_id)
path_azure_ml_ckpt = checkpoint_parser.get_or_download_checkpoint(download_dir)
```

If the Azure ML run is in a different workspace, a temporary SAS URL to download the checkpoint can be generated as follow:

```
cd hi-ml-cpath
python src/health_cpath/scripts/generate_checkpoint_url.py --run_id=AzureML_run_id:best_
↪val_loss.ckpt --expiry_days=10
```

(continues on next page)

(continued from previous page)

```
```

N.B: config.json should correspond to the original workspace where the AML run lives.

16.1 Use cases

CheckpointParser is used to specify a `src_checkpoint` to resume training from a given checkpoint, or run inference with a pretrained model, as well as ssl_checkpoint for computation pathology self supervised pretrained encoders.

CHAPTER

SEVENTEEN

HI-ML TOOLS FOR COMPUTATIONAL PATHOLOGY

The directory `hi-ml-cpath` contains code for runnning experiments in Computational Pathology.

The tools for computational pathology are best used directly from the Git repository. You can also use the `hi-ml-cpath` PyPi package to re-use the code in your own projects, for example the deep learning architectures.

17.1 Setting up your computer

Please follow the instructions in README to set up your local Python environment.

17.2 Onboarding to Azure

Please follow the *instructions here* to create an AzureML workspace if you don't have one yet. You will also need to download the workspace configuration file, as described here, so that your code knows which workspace to access.

17.3 Creating datasets

In our example models, we are working with two public datasets, PANDA and TCGA-Crck.

Please follow the *detailed instructions* to download and prepare these datasets in Azure.

17.4 Training models

- *Train a DeepMIL model with an ImageNet encoder on the PANDA dataset (whole slides)*
- *Train an SSL encoder on the TCGA-Crck dataset (tiles)*

17.5 Visualizing data and results in Digital Slide Archive DSA

- Setting up DSA
- Using DSA to look at data and model results

17.6 New Model configurations

To define your own model configuration, place a class definition in the directory `health_cpath.configs`. The class should inherit from a LightningContainer. As an example, please check the HelloWorld model or the base class for the MIL models.

17.7 Mount datasets

If you would like to inspect or analyze the datasets that are stored in Azure Blob Storage, you can either download them or mount them. "Mounting" here means that the dataset will be loaded on-demand over the network (see also the docs). This is ideal if you expect that you will only need a small number of files, or if the disk of your machine is too small to download the full dataset.

You can mount the dataset by executing this script in `<root>/hi-ml-cpath`:

```
python src/histopathology/scripts/mount_azure_dataset.py --dataset_id PANDA
```

After a few seconds, this may bring up a browser to authenticate you in Azure, and let you access the AzureML workspace that you chose by downloading the `config.json` file. If you get an error message saying that authentication failed (error message contains "The token is not yet valid (nbf)"), please ensure that your system's time is set correctly and then try again. On WSL, you can use `sudo hwclock -s`.

Upon success, the script will print out:

```
Dataset PANDA will be mounted at /tmp/datasets/PANDA.
```

CHAPTER

EIGHTEEN

USING PUBLIC DATASETS PANDA AND TCGA-CRCK

18.1 Setting up your storage account and tools

Your Azure account needs to have permissions to write to the storage account. You need to have, for example, "Storage Blob Data Contributor" permissions.

Your storage account should have a container called `datasets`. Verify in the Azure Portal that this container exists: You can do that by going to the "Containers" section in the left-hand navigation. If there is no such container, create one with the "+ Container" button.

To upload datasets to Azure, we recommend using azcopy. You will first need to log in, by calling `azcopy login`. Follow the instructions at the prompt.

18.2 PANDA Dataset

The PANDA dataset was released with the Prostate cANcer graDe Assessment (PANDA) Challenge. The dataset is available from Kaggle. To get access to the dataset, you need to register with Kaggle, then press "Download all". This will download a 200GB ZIP file.

18.2.1 Uploading

Now unzip the ZIP file. Then rename the folder in which the files reside to `PANDA`. To double-check, there should now be a file `PANDA/train.csv` in your current directory.

```
head PANDA/train.csv  # just to check if we are in the right folder
azcopy copy PANDA https://<your_storage_account>.blob.core.windows.net/datasets/ --
↪recursive
```

18.3 TCGA-Crck Dataset

This dataset contains histological images from patients with colorectal cancer, available from here.

To download and prepare the dataset, please run the following commands in a Linux shell in the root folder of the git repository. To prepare:

- In the last statement, where we upload the full dataset to Azure, replace `<your_storage_account>` with the name of your Azure storage account.

- For Python to pick up the paths in `hi-ml-cpath/src/histopathology/scripts/tcga_dataset_prep.py`, you need to add the `hi

Note: Depending on the speed of your internet connection, this script can run for several hours because it downloads a total of more than 20GB of files. It is advantageous to run the script in a Virtual Machine in Azure (downloading can be easily automated because no authentication is required).

```
mkdir TCGA-Crck
cd TCGA-Crck
# Download the files and unzip into folder broken down by Train/Test
for file in CRC_DX_TRAIN_MSIMUT.zip CRC_DX_TRAIN_MSS.zip
do
    wget https://zenodo.org/record/2530835/files/$file
    unzip $file -d CRC_DX_TRAIN
    rm $file
done
for file in CRC_DX_TEST_MSIMUT.zip CRC_DX_TEST_MSS.zip
do
    wget https://zenodo.org/record/2530835/files/$file
    unzip $file -d CRC_DX_TEST
    rm $file
done
# Create a summary file dataset.csv with all file paths and class labels
cd ..
export PYTHONPATH=`pwd`/hi-ml-cpath/src
python hi-ml-cpath/src/histopathology/scripts/tcga_dataset_prep.py
# Upload
azcopy copy TCGA-Crck https://<your_storage_account>.blob.core.windows.net/datasets/ --
↪recursive
```

18.4 Making your storage account accessible to AzureML

As a last step, you need to ensure that AzureML has access to your storage account. For that, you need to create a datastore. A datastore is an abstraction layer on top of the plain storage account, where AzureML stores an account key or access token that allows it to later download the data to the training machine.

To create the datastore, please follow the instructions here. Once the datastore is created, mark it as the default datastore.

CHAPTER

NINETEEN

DEEPMIL MODEL FOR TUMOR GRADING ON PANDA DATASET

19.1 Background

The repository contains the configuration for training Deep Multiple Instance Learning (DeepMIL) models for ISUP score prediction on the PANDA challenge dataset. The models are developed to reproduce the results described in Myronenko et al. 2021 and the example in the Project-MONAI tutorial.

A ResNet50 encoder that was pre-trained on ImageNet is downloaded on-the-fly (from here at the start of the training run.

19.2 Preparations

Please follow the instructions in the Readme file to create a Conda environment and activate it, and the instructions to set up Azure.

You will also need to run the dataset preparations for the PANDA dataset, as described here.

19.3 Running the model as-is

If you have a GPU available, you can run training on that machine, by executing in `<root>/hi-ml-cpath`:

```
conda activate HimlHisto
python ../hi-ml/src/health_ml/runner.py --model health_cpath.
↪SlidesPandaImageNetMILBenchmark
```

Running the model will automatically mount (download on-the-fly) the PANDA dataset from Azure. To enable that, you will be asked to log into Azure a few seconds after the start of the script. This will either pop up a browser window automatically, or give you a prompt on the console to open the browser.

Once the authentication is completed, it will access the AzureML workspace that you chose by downloading the `config.json` file. If you get an error message saying that authentication failed, "The token is not yet valid (nbf)", please ensure that your system's time is set correctly (on WSL, use `sudo hwclock -s`) and then try again.

However, the GPU demand for this model is rather high. We recommend running in AzureML, on a GPU compute cluster. You can run the training in the cloud by simply appending name of the compute cluster, `--cluster=<your_cluster_name>`. In addition, you can turn on fine-tuning of the encoder, which will improve the results further:

```
conda activate HimlHisto
python ../hi-ml/src/health_ml/runner.py --model health_cpath.
↪SlidesPandaImageNetMILBenchmark --tune_encoder --cluster=<your_cluster_name>
```

Then the script will output "Successfully queued run number . . ." and a line prefixed "Run URL: . . .". Open that URL to view the submitted run in AzureML, view progress, metrics, etc.

19.4 Expected results

The best runs so far uses Transformer Pooling layer similar to the one implemented in Myronenko et. al 2021 (`pool_type=TransformerPoolingBenchmark.__name__`), in combintation with fine-tuning the encoder.

`SlidesPandaImageNetMIL` model trained with fine-tuning, cross validation metrics (mean $\pm$ std):

- Validation accuracy: 0.7828 $\pm$ 0.0037
- Validation AUROC: 0.9473 $\pm$ 0.002
- Validation QWK: 0.8793 $\pm$ 0.0074

For internal reference, this was run `HD_0e805b91-319d-4fde-8bc3-1cea3a6d08dd` on `innereye4ws`.

19.5 Model variants

Six different pooling layers can be used by changing parameter `pool_type` in the configuration file, or chosen on the commandline or from CLI. Available pooling layers include:

- `pool_type=AttentionLayer.__name__`: Attention layer from Ilse et al. 2018
- `pool_type=GatedAttentionLayer.__name__`: Gated attention layer from Ilse et al. 2018
- `pool_type=MaxPoolingLayer.__name__`: Max pooling layer returns frequency normalized weights and the maximum feature vector over the first axis
- `pool_type=MeanPoolingLayer.__name__`: Mean pooling layer returns uniform weights and the average feature vector over the first axis
- `pool_type=TransformerPooling.__name__`: Transformer pooling layer
- `pool_type=TransformerPoolingBenchmark.__name__`: Transformer pooling layer used in Myronenko et al. 2021

Use the optional runner argument `--mount_in_azureml` to mount the PANDA dataset on AzureML, instead of downloading it. This will mean that the job starts faster, but may not run at maximum speed because of network bottlenecks.

19.6 Cross-validation

To use cross-validation, supply the additional commandline flag `--crossval_count=5` for 5-fold cross-validation, like:

```
python ../hi-ml/src/health_ml/runner.py --model health_cpath.
↪idesPandaImageNetMILBenchmark --crossval_count=5 --cluster=<your_cluster_name>
```

Cross-validation will start 5 training runs in parallel. For this reason, cross-validation can only be used in AzureML.

To compute aggregated metrics of the hyperdrive run in Azure ML, replace the `run_id` in `hi-ml-cpath/src/histopathology/scripts/aggregate_metrics_crossvalidation.py` with the Run ID of the hyperdrive run, and run the script as follows:

```
conda activate HimlHisto
python hi-ml-cpath/src/histopathology/scripts/aggregate_metrics_crossvalidation.py
```

CHAPTER

TWENTY

DEEPMIL MODEL FOR GENETIC MUTATION ON TCGA-CRCK

20.1 Background

The repository contains the configuration for training a Deep Multiple Instance Learning (DeepMIL) models for microsatellite instability (MSI) prediction on the TCGA-CRCk dataset. Instructions to download and prepare the dataset are *here*. The dataset is composed of tiles.

The models are developed to reproduce the results described in the DeepSMILE papers Schirris et al. 2021 and Schirris et al. 2022.

A ResNet18 encoder that was pre-trained on ImageNet is downloaded on-the-fly (from here at the start of the training run when using the ImageNet configuration. The SSL MIL configuration requires the checkpoint of a pre-trained SSL encoder (ResNet50) on the TCGA-CRCk dataset. The SSL encoder that can be obtained following the instructions on *how to train a custom SSL encoder on a pre-tiled dataset*.

20.2 Preparations

Please follow the instructions in the Readme file to create a Conda environment and activate it, and the instructions to set up Azure to run in the cloud.

20.3 Running the model

You can run the model in the same way you run the *benchmark model on PANDA*. If you have GPU available locally, you can run training on that machine, by executing in `<root>/hi-ml-histopathology`:

```
conda activate HimlHisto
python ../hi-ml/src/health_ml/runner.py --model health_cpath.TcgaCrckImageNetMIL
```

If you setup a GPU cluster in Azure, you can run the training in the cloud by simply appending name of the compute cluster:

```
conda activate HimlHisto
python ../hi-ml/src/health_ml/runner.py --model health_cpath.TcgaCrckImageNetMIL --
↪cluster=<your_cluster_name>
```

Assuming you pre-trained your own SSL encoder and updated the checkpoint path in the SSLencoder class, the SSL MIL model configuration can be run using the flag `--model histopathology.TcgaCrckSSLMIL`.

20.4 Expected results

ImageNet MIL (ResNet18)

- Test AUROC: 0.706 $\pm$ 0.041
- Test F1 score: 0.490 $\pm$ 0.041

SSL MIL (ResNet50)

- Test AUROC: 0.825 $\pm$ 0.065
- Test F1 score: 0.516 $\pm$ 0.171

CHAPTER

TWENTYONE

PRE-TRAINING OF AN IMAGE ENCODER USING SELF-SUPERVISED LEARNING

Often in histopathology, we only have access to weak labels, e.g., a single label for an entire Whole Slide Image (WSI). However, papers like DeepSmiles show that we can use unlabeled tiles from WSI to pre-train an image encoder using Self-Supervised Learning (SSL). In hi-ml, we have implemented two popular self-supervised learning methods SimCLR and BYOL. We will use the TCGA-CRCk dataset, as seen in Kather et al. 2019, the dataset comes with binary WSI labels (microsatellite stable or instable). We will use the TCGA-CRCk dataset as an example to show how to set up SSL training in hi-ml. If you want to use your own dataset, you will find instructions at the end of this section.

21.1 Example: Train an image encoder using SSL on the TCGA-CRCk locally

The TCGA-CRCk dataset consists of colorectal tumor tiles extracted from Formalin-Fixed, Paraffin-Embedded (FFPE) WSIs from the Cancer Genome Atlas (TCGA) with accompanying binarized MicroSatellite Instability (MSI) labels. In the case of TCGA-CRCk, the dataset is already tiled, i.e., the WSI are not available. In *public_datasets.md* you will find instructions on how to download and setup the TCGA-CRCk dataset.

To train an image encoder using SSL locally run this in the `hi-ml-cpath` folder, with the `HimlHisto` conda enviroment activated:

```
python ../hi-ml/src/health_ml/runner.py --model SSL.CRCK_SimCLR
```

The model class `CRCK_SimCLR` is the config used to train a SSL model on TCGA-CRCk. It houses everything, e.g., the model, the dataset, checkpointing, etc. Here, we need to define some important parameters:

1. The type of image encoder we want to train, the type of SSL (SimCLR or BYOL) we want to use, and the batch_size.

- `ssl_encoder=EncoderName.resnet50`
- `ssl_training_type=SSLTrainingType.SimCLR`
- `ssl_training_batch_size=48`

1. The dataset we want to use for training the image encoder and the linear model we only use for evaluation of the image encoder. In theory, they could be two different datasets.

- `ssl_training_dataset_name=SSL_Dataset_TCGA_CRCK`
- `linear_head_dataset_name=SSL_Dataset_TCGA_CRCK`

1. Model checkpointing: We use PyTorch lightning checkpointing. Among others, we define the validation metric, where the `online_evaluator` is the same as the `linear_head`. In the case of TCGA_CRCK, we use AUC ROC as the validation metric.

- `model_monitor_metric='ssl_online_evaluator/val/AreaUnderRocCurve'`

In the parent class of `CRCK_SimCLR`, `HistoSSLContainer` the data augmentations are defined. Data augmentation is one of the most important components of SSL training. Currently, we have hardcoded the data augmentation used in the SimCLR paper. These are the following:

- `RandomResizedCrop(size=224)`
- `RandomHorizontalFlip(p=0.5)`
- `RandomApply([ColorJitter(brightness=0.8, contrast=0.8, saturation=0.8, hue=0.2)], 0.8)`
- `RandomGrayscale(p=0.2)`
- `GaussianBlur(int(224 * 0.1) + 1)`

While not optimized for WSI we observe good performance using these augmentations. The data augmentations are wrapped by `DualViewTransformWrapper` to return two augmented versions per tile, as required by the majority of SSL methods.

21.2 Train on Azure

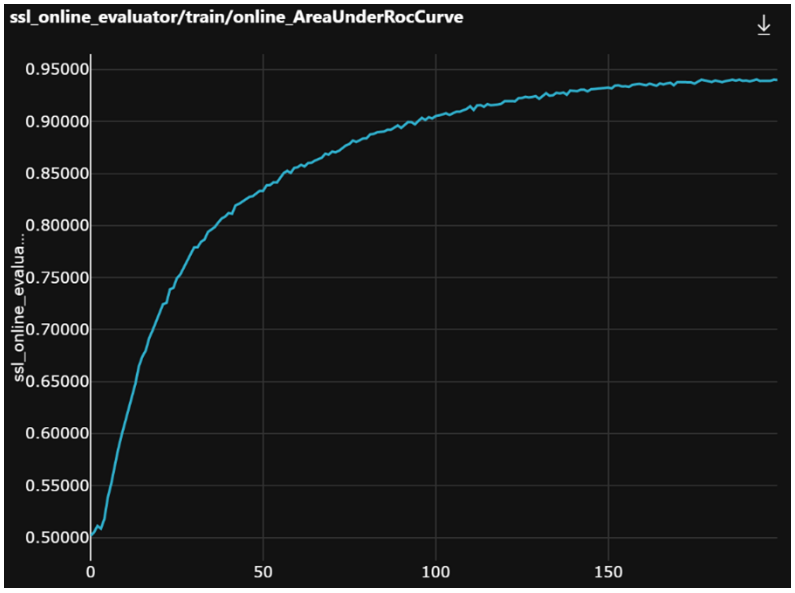
In the case of SimCLR, the effective batch_size (batch_size * GPU) should be as big as possible. The SSL models in hi-ml natively supports distributed training using multiple GPUs. We recommend using 8 GPUs for running the SimCLR model on the TCGA-CRCk dataset. Assuming you are using a total of 8 GPUs (e.g. 1 node with 8 GPUs or 2 nodes with 4 GPUs) in Azure you can start training with the following command in the repository root folder:

```
python hi-ml/src/health_ml/runner.py --model SSL.CRCK_SimCLR --cluster CLUSTER_NAME --
↪conda_env hi-ml-cpath/environment.yml
```

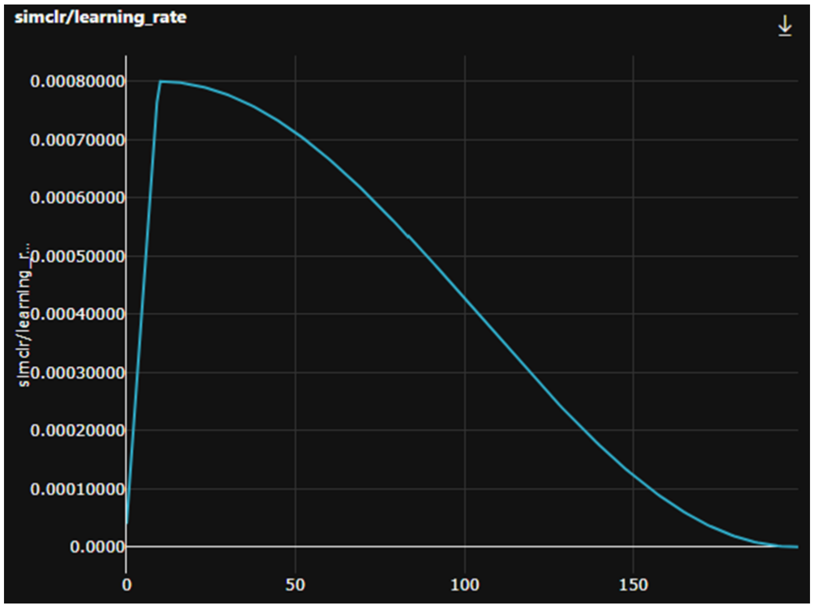
A SimCLR run with 200 epochs, 8 GPUs, and a batch size of 48 (per GPU) takes about 6 hours. On Azure we use Standard_ND40rs_v2 (40 cores, 672 GB RAM, 2900 GB disk, 8 x NVIDIA Tesla V100).

Let’s have a look at the training behavior.

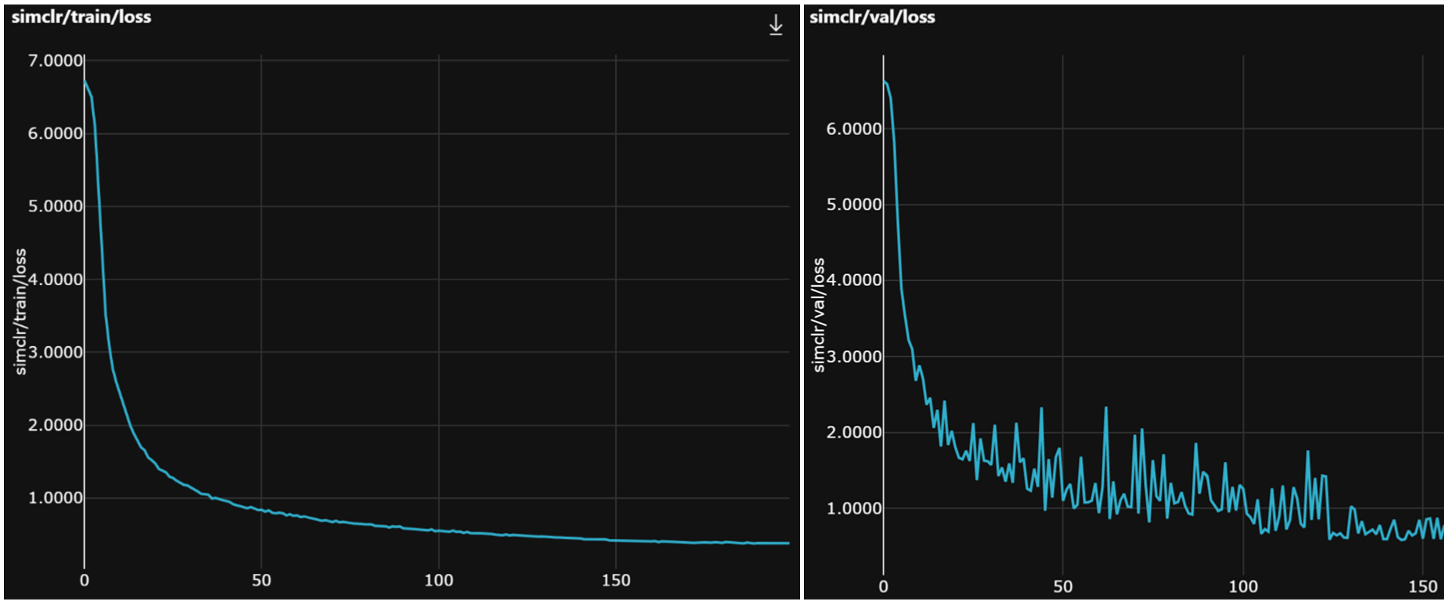
As mentioned previously, using the WSI label for each tile of the same slide and a linear head on the outputs of the image encoder to monitor training works quite well. We see a smooth and steady increase of the validation metric.



In addition, we are using a cosine learning rate schedule with a fixed warm up of 10 epochs. Note: The SSL code in hi-ml automatically scales the learning rate to the number of GPUs used during training, as described here.



Last, the training and validation loss curves are expected to look like this.



After training, we can use the pre-trained image encoder on downstream tasks like microsatellite stable/instable prediction on TCGA-CRCk. You only have to specify the path to the checkpoint of the SSL image encoder in the setup function of `DeepSMILECrck`.

21.3 Using your own dataset

For scripts that help you tile your own dataset please see `histopathology/preprocessing/tiling.py`. In the case of TCGA-CRCk, the dataset is already tiled. `TcgaCrck_TilesDataset` is a child of `TilesDataset`. For a `TilesDataset` we assume that each tile has a unique tile index and a label. Since we assume that we are working with weakly labelled WSI we do not have access to the real tile labels. However, we found that monitoring the SSL training using the slide label for each tile works sufficiently well. I.e., if the WSI has a positive label then every tile from the WSI has a positive label. Last, the unique index for each tile is used to make sure we don't use twice the same tile in one training epoch during SSL training.

Subsequently, the TCGA-CRCk dataset is wrapped in `TcgaCrck_TilesDatasetReturnImageLabel`. Here the data augmentations are applied and the `__getitem__` method is defined.

The dataset is then wrapped one last time in `TcgaCrck_TilesDatasetWithReturnIndex`, where we inherit the ability to return the tile index from `DatasetWithReturnIndex`.

CHAPTER

TWENTYTWO

DIGITAL SLIDE ARCHIVE

The Digital Slide Archive (DSA) is "a containerized web-based platform for the analysis, visualization, management and annotation of whole-slide digital pathology imaging data". It is an open-source project based on Kitware's data management platform Girder.

22.1 Azure deployment

We have deployed the DSA on Azure to visualize our data and interpret our models and experiments. Below are instructions to replicate our deployment using your own data.

22.1.1 Host

The first step is creating a Linux virtual machine (VM) to host the DSA. The code can be downloaded from GitHub and run using Docker. By default, the application runs at port 8080. To use HTTPS, an SSL certificate must be obtained and port 443 may be specified instead within the Docker configuration. The DSA uses Cherrypy as the underlying server engine. Therefore, Cherrypy must be configured to use the SSL certificate installed in the VM. Ports in a VM can be opened using network security group rules.

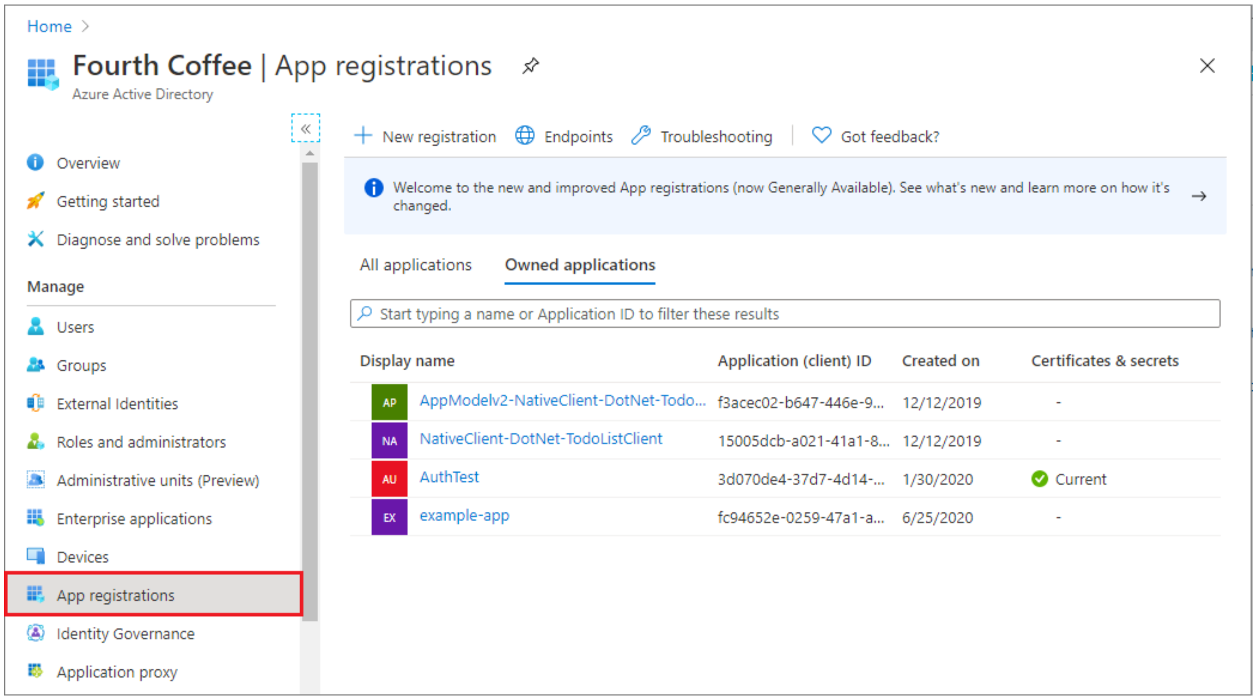
22.1.2 Storage

The datasets we use are securely stored in Azure Blob Storage containers. Containers can be mounted on the host machine using BlobFuse. Then, Girder assetstores can be created and fed data from the mounted containers.

22.1.3 Authentication

We have added Microsoft as a provider in the OAuth2 Login plugin for Girder. The authentication settings widget describes the steps needed to generate the provider credentials.

First, our deployed DSA must be registered in Azure Active Directory (Azure AD) as an application.



Welcome to the new and improved App registrations (now Generally Available). See what's new and learn more on how it's changed.

Display name	Application (client) ID	Created on	Certificates & secrets
AP AppModelv2-NativeClient-DotNet-Todo...	f3acec02-b647-446e-9...	12/12/2019	-
NA NativeClient-DotNet-TodoListClient	15005dcb-a021-41a1-8...	12/12/2019	-
AU AuthTest	3d070de4-37d7-4d14-...	1/30/2020	✓ Current
EX example-app	fc94652e-0259-47a1-a...	6/25/2020	-

Once the DSA app has been registered, a client ID (also called *application ID*) will be generated. Then, a client secret (also called *application password*) must also be generated. Finally, the *tenant ID* (the Azure ID of your organization) *may* be specified so that access is restricted to users belonging to your tenant. These three strings will be used to configure authentication in the OAuth2 Login plugin settings.

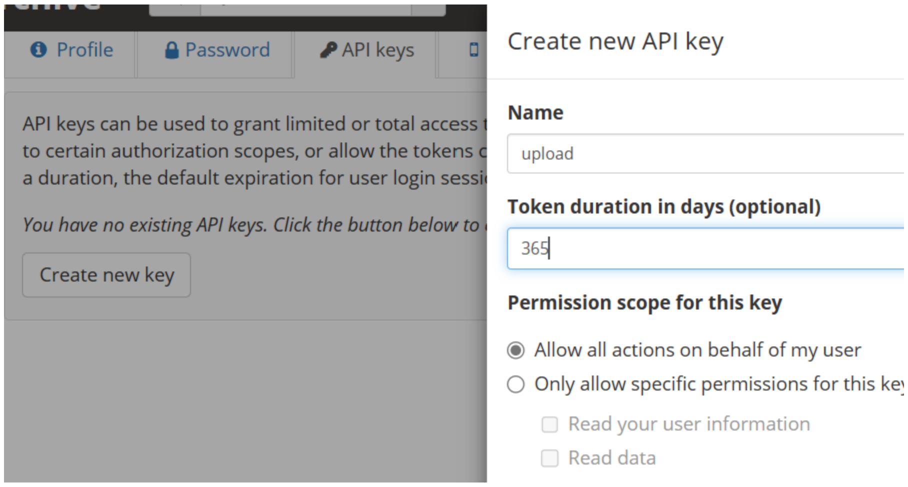
22.1.4 Authorization

User permissions can be set to different data collections in an assetstore, to maximize protection of sensitive data.

22.1.5 Creating API Keys

You can create an API key for a user in DSA directly on the web.

- Navigate to your DSA and log in
- Click on “Users” in the navigation bar
- Click on the user you want to create an API key for
- On the top right, there is a menu “Actions”, choose “Edit user”
- You will see 4 tabs with user information, click on “API Keys”
- Choose a name for the key, a duration in days. It is important to choose “Allow all actions on behalf of this user”!
- Click “Create”. You will see the list of keys, press “show” to reveal the actual key value. Copy it.
- Set the key as an environment variable `DSA_API_KEY` (in `bash`, this would be `export DSA_API_KEY=<the key>`)
- In addition, you can also set the URL for your DSA instance as an environment variable `DSA_URL` (in `bash`, this would be `export DSA_URL=<the url>`)



API keys can be used to grant limited or total access
to certain authorization scopes, or allow the tokens
a duration, the default expiration for user login sessi

You have no existing API keys. Click the button below to

Create new key

Create new API key

Name: upload

Token duration in days (optional): 365

Permission scope for this key

(●) Allow all actions on behalf of my user
( ) Only allow specific permissions for this ke
[ ] Read your user information
[ ] Read data

22.2 Visualizing Azure Machine Learning results

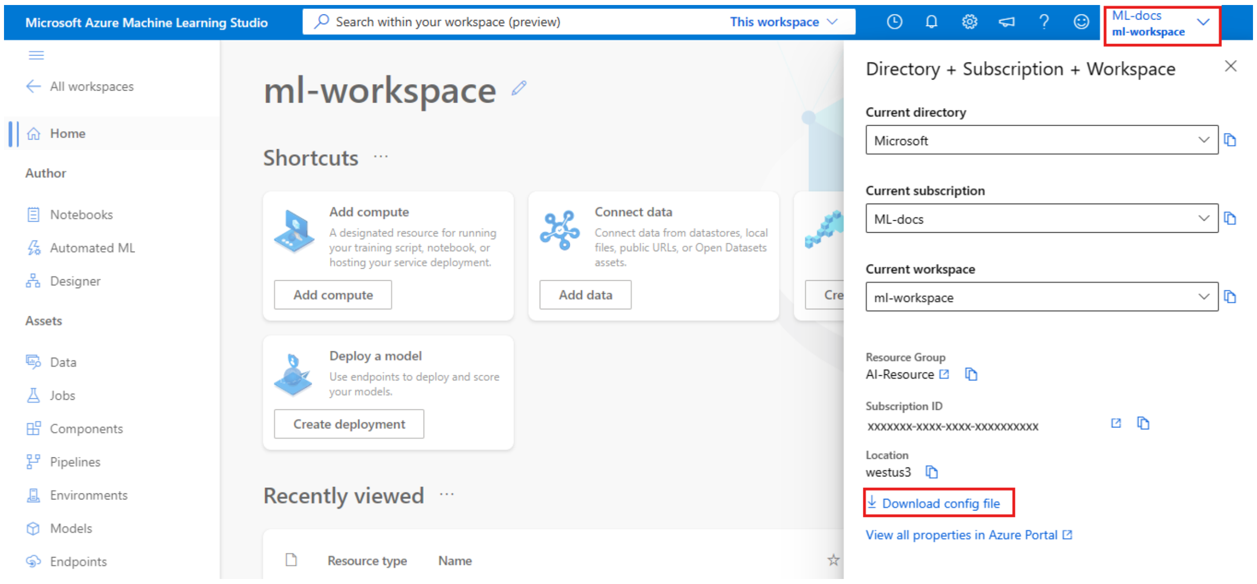
The Girder RESTful API may be used to upload annotations to DSA items programmatically. An example of a use case is creating annotations to visualize attention maps generated by a deep learning model. The `girder` module in this repository includes tools to download training results from Azure Machine Learning (Azure ML) and upload annotations to a deployed DSA. For example:

```
SCRIPT="hi-ml-cpath/src/histopathology/utils/girder.py"
python $SCRIPT \
    --run-id "Experiment_transformer_Gleason_run_0" \
    --dsa-url "https://my-deployed-dsa.azure.com/" \
    --dsa-key "AHKZ42Ks24kSH5Fxt3354ryKzCxamjqM" \
    --workspace-config "config.json" \
```

The DSA URL and API key may be specified in environment variables `DSA_URL` and `DSA_API_KEY` instead, see above. The workspace configuration file contains information related to the Azure Machine Learning workspace:

```
{
    "subscription_id": "22f7beb4-1b54-9ee7-5255-6dbcc8da000b",
    "resource_group": "myresourcegroup",
    "workspace_name": "myworkspace"
}
```

It can be downloaded from the workspace website:


Microsoft Azure Machine Learning Studio
Search within your workspace (preview)
This workspace
ML-docs
ml-workspace
All workspaces
Home
Author
Notebooks
Automated ML
Designer
Assets
Data
Jobs
Components
Pipelines
Environments
Models
Endpoints
ml-workspace
Shortcuts
Add compute
A designated resource for running your training script, notebook, or hosting your service deployment.
Add compute
Connect data
Connect data from datastores, local files, public URLs, or Open Datasets assets.
Add data
Cre
Deploy a model
Use endpoints to deploy and score your models.
Create deployment
Recently viewed
Resource type
Name
Directory + Subscription + Workspace
Download config file
View all properties in Azure Portal

Current directory
Microsoft

Current subscription
ML-docs

Current workspace
ml-workspace

Resource Group
AI-Resource

Subscription ID
xxxxxxx-xxxx-xxxx-xxxxxxxxxx

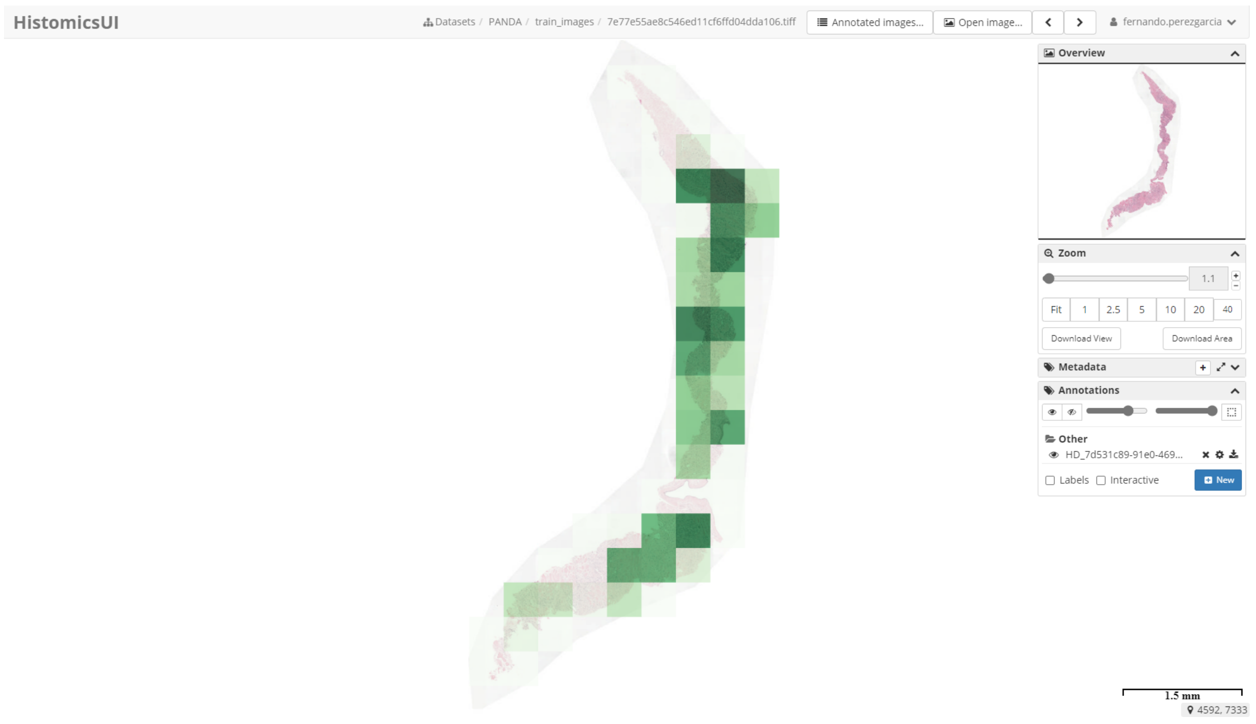
Location
westus3

The script uses the Azure SDK and the Python client for the Girder API to:

1. Log into Azure ML
2. Download training results
3. Generate JSON annotations for each slide
4. Search for the slide by slide ID in DSA by using full text search
5. Upload the annotations to the deployed DSA

For a full description of all the options, add `--help` to the arguments.

Below is an example of an attention map overlaid on a slide from the PANDA dataset:



22.2.1 Upload into a folder

The `girder.py` script can also upload the annotations into a folder in DSA. This is helpful if there are multiple variants of the same slide in DSA, but located in different folders. Simple text search would then return multiple results.

To use the folder functionality, you need to supply the name of the folder via `--folder`. For example, to upload the annotations to folder `foo` in collection `Collection`, add ``–folder Collection1/foo`.

When supplying a folder argument, searching for the slide where annotations will be added works differently:

1. Firstly, all slides in the folder are retrieved.
2. Then, the script tries to identify which of the slides in the folder contains the value from the `slide_id` field.
3. If there is exactly one matching slide, the annotations are added to that slide. Annotations that match zero or more than one slide are ignored.

CHAPTER

TWENTYTHREE

CREATING MONTAGES FROM WHOLE SLIDE IMAGES (WSIS)

For working with large amounts of histology data, it is often useful to create montages of the data. Montages are a collection of images that are stitched together to form a single image. Montages are useful for visualizing large amounts of data at once, and can be used to create a single image that can be used for analysis. The `hi-ml-cpath` toolbox contains scripts that help with the creation of montages from whole slide images (WSIs).

Creating montages can be very time-consuming. It can hence be helpful to run the process in the cloud. The montage creation code provided here can be run in AzureML very easily.

23.1 Types of data for montage creation

1. Montages can be created from a folder of images, by specifying the name of the folder and a glob pattern, like `**/foo_*.tiff`.
2. Montages can be created by first reading a file called `dataset.csv` located in a folder. `dataset.csv` is effectively a Pandas DataFrame, with each row corresponding to a single image. More details on the format of `dataset.csv` can be found below.

Montage creation works for all WSI image formats that are supported by either of the two possible backends:

- `openslide` supports `.tif(f)`, `.ndpi`, `.scn` and others
- `cucim` supports `.svs`, `.tiff` and others

23.2 Setup

- Check out the `hi-ml` repository via `git clone https://github.com/microsoft/hi-ml`
- Run the following commands:

```
cd hi-ml-cpath
make env
conda activate HimlHisto
make pip_local
```

All the commands listed below assume that

- you have activated the Conda environment `HimlHisto`
- your current working directory is `<repo root>/hi-ml-cpath`

23.3 Creating montages from a folder with files

The following command will create a montage from all files in the folder `/data` that match the pattern `**/*.tiff`.

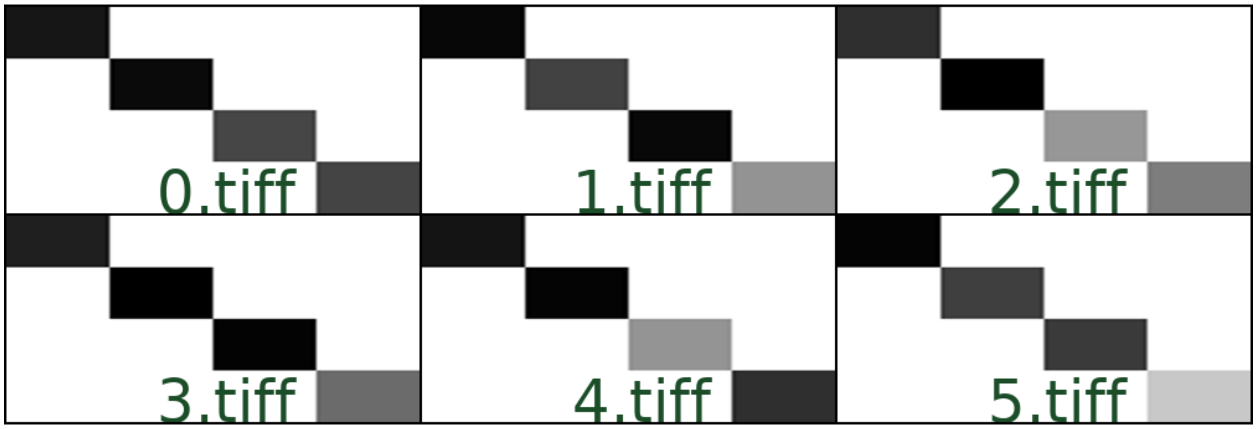
```
python src/health_cpath/scripts/create_montage.py --dataset /data --image_glob_pattern
↪'**/*.tiff' --level 2 --width 1000 --output_path montage1
```

This will create a montage from all TIFF files in folder `/data`. Each TIFF file is read as a multi-level image, and level 2 is read for creating the montage.

The `--width` argument determines the width in pixel of the output image. The height of the output image is determined automatically. Note that the width given here, 1000, is suitable only for montages from a very small number of files (say, 10). See below for more details on this commandline option.

This will create two images in the folder specified by the `--output_path montage1` argument, hence outputting the files `montage1/montage.jpg` and `montage1/montage.png`.

Here's an example how this could look like for a folder with 6 images, `0.tiff` through `5.tiff`:



23.4 Creating montages from a `dataset.csv` file

If the montage creation script is only pointed to a folder, without providing a glob pattern, it assumes that a file `dataset.csv` is present. A montage will be created from only the images listed in `dataset.csv`. In addition, an optional `label` column will be added to the text that is overlayed onto the images itself.

The dataset file should be a CSV file, with each row corresponding to a single image. When working with a `dataset.csv` file, the following columns are handled:

Column name	Contents	Required?
`image`	The path of the image that should be loaded	Required
`slide_id`	A unique identifier for the slide	Required
`label`	An additional string that will be placed on the montage, This could be `0`, `1`, `tumour`, …	Optional
`mask`	The path of an additional image that will rendered next to the image given in `image`	Optional

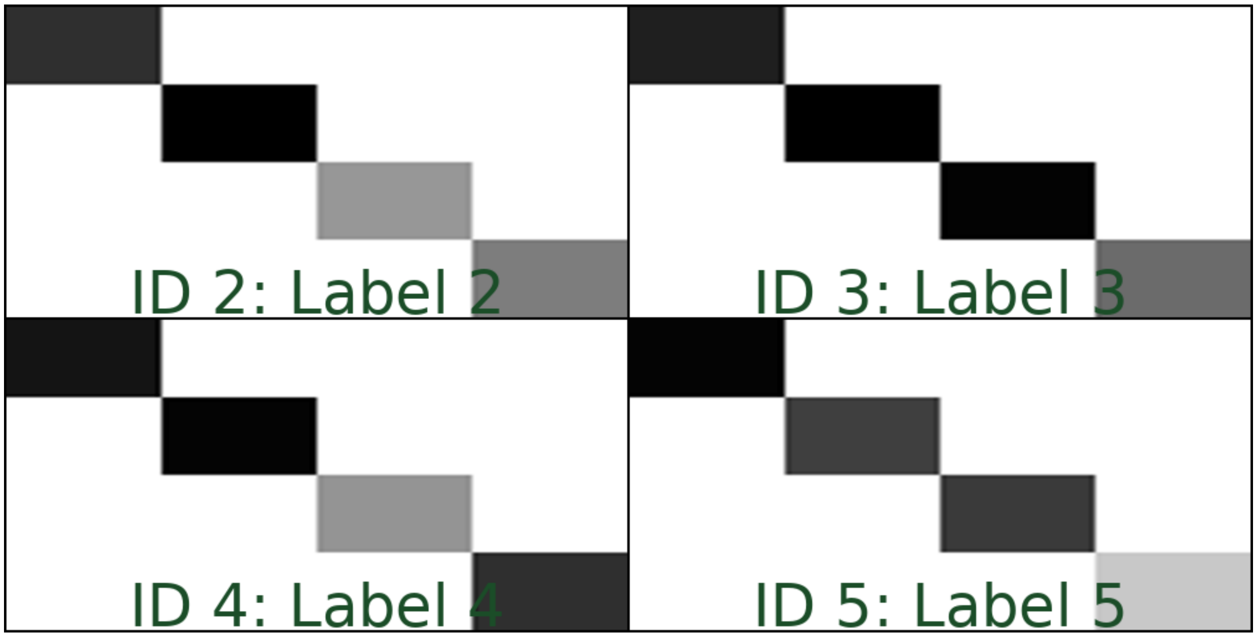
Consider this example dataset file:

```
image,slide_id,label
2.tiff,ID 2,Label 2
3.tiff,ID 3,Label 3
4.tiff,ID 4,Label 4
5.tiff,ID 5,Label 5
```

Run montage creation with the following command:

```
python src/health_cpath/scripts/create_montage.py --dataset /data --level 2 --width 1000␣
↪--output_path montage1
```

This would produce (assuming that the images `2.tiff`, `3.tiff`, `4.tiff`, and `5.tiff` are present in the folder `/data`) a montage similar to this one:



23.4.1 Using inclusion or exclusion lists

When creating montages from a `dataset.csv` file, it is possible to create montages from only a specific subset of rows, or all rows apart from those in a given list.

- Use the `--exclude_by_slide_id exclude.txt` argument to point to a file with a list of slide IDs that should be excluded from the montage.
- Use the `--include_by_slide_id include.txt` argument to point to a file with a list of slide IDs for which the montage should be created.

The files `exclude.txt` and `include.txt` should contain one slide ID per line.

23.5 Other commandline options

- Use `--width=20000` to set the width of the output montage image. The height of the output image is determined automatically. Mind that the default value is 60_000, suitable for several hundreds of input images. If you want to try out montage creation on a small set of files (say, 10), ensure that you set the width to a reasonably small value, like `--width=1000`
- Use `--parallel=2` to specify the number of parallel processes that should be used for creating image thumbnails. Thumbnails are created in a first step, using multiple processes, and then the thumbnails are stitched into the final montage in the main process.
- Use `--backend=cucim` to switch the image reader backend to `CuCIM`. The default backend is `openslide`.

23.6 Running in Azure

The `create_montage.py` script can be run in AzureML by adding 3 commandline arguments.

To set up Azure and AzureML:

- Follow the steps in the *AzureML onboarding*.
- At the end of the onboarding you will download a file `config.json` from your AzureML workspace to your repository root folder.
- To understand datasets, please read through the *AzureML datasets* documentation. Then create an AzureML datastore that points to your Azure Blob Storage account.
- Upload your WSIs to a folder in Azure Blob Storage. This can be done most efficiently via azcopy. `azcopy` can also copy directly across cloud providers, for example from AWS to Azure.

The following command will upload all files in the folder `my_test_slides` to a container `datasets` in your Azure Blob Storage account called `mystorage`, creating a folder `my_test_slides` in the storage account in the process:

```
azcopy copy my_test_slides https://mystorage.blob.core.windows.net/datasets/ --recursive
```

The following command will then create a run in AzureML that executes montage creation from that folder:

```
python src/health_cpath/scripts/create_montage.py --dataset my_test_slides --level 2 --
↪width 1000 --cluster <clustername> --conda_env environment.yml --datastore
↪<datastorename>
```

In this command, replace the following:

- Replace `my_test_slides` with the name of the folder in blob storage where you uploaded your WSIs.
- `clustername` is the name of a compute cluster where your job will execute)
- `datastorename` is the name of an AzureML datastore, essential a pointer to your blob storage account plus the credentials that are necessary to access it. For the above example, the data store needs to point to storage account `mystorage` and container `datasets`.

The command above will only run for a minute or less - it will mostly create a snapshot of the code and send that off to the cloud for execution. At the end you will see a link printed out that takes you to the AzureML portal, where you can monitor the progress of the run.

Once the run is completed, you will find two files `montage.jpg` and `montage.png` in the tab "Outputs" of the run, and an option to download it to your machine.

CHAPTER

TWENTYFOUR

HISTOPATHOLOGY SCRIPTS

Please see links below for details on the available arguments for each of the histopathology scripts.

24.1 create_montage

This script can create a high-resolution montage of all images in a folder or datasets.

Create an overview image with thumbnails of all slides in a dataset.

```
usage: python create_montage.py --dataset <dataset_folder> --image_glob_pattern '**/*.
↪tiff' --width 1000
```

24.1.1 Named Arguments

--cluster	The name of the GPU or CPU cluster inside the AzureML workspacethat should execute the job. To run on your local machine, omit this argument. Default: “”
--datastore	The name of the AzureML datastore where the dataset is defined. Default: “”
--dataset	The name of the AzureML dataset to use for creating the montage. The dataset will be mounted automatically. Use an absolute path to a folder on the local machine to bypass mounting. Default: “”
--conda_env	The Conda environment file that should be used when submitting the present run to AzureML. If not specified, the hi-ml-cpath environment file will be used. Default: hi-ml/hi-ml-cpath/environment.yml
--wait_for_completion	If True, wait for AML Run to complete before proceeding. If False, submit the run to AML and exit Default: False
--docker_shm_size	The shared memory in the Docker image for the AzureML VMs. Default: “100g”

--workspace_config_path The path to the AzureML workspace configuration file. If not specified, the configuration file in the current folder or one of its parents will be used.

--display_name The display name of the AzureML run. If not specified, a default name will be used.

Default: “”

--level Resolution downsample level, e.g. if lowest resolution is 40x and the available downsample levels are [1.0, 4.0, 16.0] then level = 1 corresponds to 10x magnification

Default: 1

--exclude_by_slide_id Provide a file that contains slide IDs that should be excluded. File format is CSV, the first column is used as the slide ID. If the file is empty, no slides will be excluded.

--include_by_slide_id Provide a file that contains slide IDs that should be included. File format is CSV, the first column is used as the slide ID. If the file is empty, no montage will be produced.

--image_glob_pattern When provided, use this pattern in rglob to find the files that should be included in the montage. Example: ‘**/*.tiff’ to find all TIFF files recursive. You may have to escape the pattern in your shell.

Default: “”

--width The width of the montage in pixels

Default: 60000

--output_path The folder where the montage will be saved

Default: outputs

--parallel The number of parallel processes to use when creating the montage.

Default: 8

--backend The backend to use for reading the slides. Can be ‘openslide’ or ‘cucim’

Default: “openslide”

CHAPTER

TWENTYFIVE

HI-ML MULTIMODAL TOOLBOX

This toolbox provides models for multimodal health data. The code is available on GitHub and Hugging Face .

25.1 Getting started

The best way to get started is by running the phrase grounding notebook and the *examples*. All the dependencies will be installed upon execution, so Python 3.9 and Jupyter are the only requirements to get started.

The notebook can also be run on Binder, without the need to download any code or install any libraries:

25.2 Installation

The latest version can be installed using `pip`:

```
pip install --upgrade hi-ml-multimodal
```

25.2.1 Development

For development, it is recommended to clone the repository and set up the environment using conda:

```
git clone https://github.com/microsoft/hi-ml.git
cd hi-ml-multimodal
make env
```

This will create a `conda` environment named `multimodal` and install all the dependencies to run and test the package.

You can visit the API documentation for a deeper understanding of our tools.

25.3 Examples

For zero-shot classification of images using text prompts, please refer to the example script that utilises a small subset of Open-Indiana CXR dataset for pneumonia detection in chest X-ray images. Please note that the examples and models are not intended for deployed use cases (commercial or otherwise), which is currently out-of-scope.

25.4 Hugging Face

While the GitHub repository provides examples and pipelines to use our models, the weights and model cards are hosted on Hugging Face .

25.5 Credit

If you use our code or models in your research, please cite our recent ECCV and CVPR papers:

Boecking, B., Usuyama, N. et al. (2022). Making the Most of Text Semantics to Improve Biomedical Vision–Language Processing. In: Avidan, S., Brostow, G., Cissé, M., Farinella, G.M., Hassner, T. (eds) Computer Vision – ECCV 2022. ECCV 2022. Lecture Notes in Computer Science, vol 13696. Springer, Cham. https://doi.org/10.1007/978-3-031-20059-5_1

Bannur, S., Hyland, S., et al. (2023). Learning to Exploit Temporal Structure for Biomedical Vision-Language Processing. In: CVPR 2023.

25.5.1 BibTeX

```
@InProceedings{10.1007/978-3-031-20059-5_1,
    author="Boecking, Benedikt and Usuyama, Naoto and Bannur, Shruthi and Castro, Daniel␣
↪C. and Schwaighofer, Anton and Hyland, Stephanie and Wetscherek, Maria and Naumann,␣
↪Tristan and Nori, Aditya and Alvarez-Valle, Javier and Poon, Hoifung and Oktay, Ozan",
    editor="Avidan, Shai and Brostow, Gabriel and Ciss{\'e}, Moustapha and Farinella,␣
↪Giovanni Maria and Hassner, Tal",
    title="Making the Most of Text Semantics to Improve Biomedical Vision--Language␣
↪Processing",
    booktitle="Computer Vision -- ECCV 2022",
    year="2022",
    publisher="Springer Nature Switzerland",
    address="Cham",
    pages="1--21",
    isbn="978-3-031-20059-5"
}

@inproceedings{bannur2023learning,
    title={Learning to Exploit Temporal Structure for Biomedical Vision{\textendash}
↪Language Processing},
    author={Shruthi Bannur and Stephanie Hyland and Qianchu Liu and Fernando P\'{e}rez-
↪Garc\'{i}a and Maximilian Ilse and Daniel C. Castro and Benedikt Boecking and Harshita␣
↪Sharma and Kenza Bouzid and Anja Thieme and Anton Schwaighofer and Maria Wetscherek␣
↪and Matthew P. Lungren and Aditya Nori and Javier Alvarez-Valle and Ozan Oktay},
    booktitle={Conference on Computer Vision and Pattern Recognition 2023},
```

(continues on next page)

(continued from previous page)

```
    year={2023},
    url={https://openreview.net/forum?id=5jScn5xsbo}
}
```

CHAPTER

TWENTYSIX

API

26.1 Vision-language processing (VLP)

vlp	Visual-language processing tools

26.1.1 health_multimodal.vlp

Visual-language processing tools

inference_engine	Tools related to joint image and text inference

health_multimodal.vlp.inference_engine

Tools related to joint image and text inference

Classes

ImageTextInferenceEngine(...)	Functions related to inference on `ImageTextModel`.

class `health_multimodal.vlp.inference_engine.`**`ImageTextInferenceEngine`**(*image_inference_engine*, *text_inference_engine*)

Functions related to inference on `ImageTextModel`.

static **`convert_similarity_to_image_size`**(*similarity_map*, *width*, *height*, *resize_size*, *crop_size*, *val_img_transform=None*, *interpolation='nearest'*)

Convert similarity map from raw patch grid to original image size, taking into account whether the image has been resized and/or cropped prior to entering the network.

Return type `ndarray`

`get_similarity_map_from_raw_data`(*image_path*, *query_text*, *interpolation='nearest'*)

Return a heatmap of the similarities between each patch embedding from the image and the text embedding.

Parameters

- **`image_path`** (`Path`) – Path to the input chest X-ray, either a DICOM or JPEG file.
- **`query_text`** (`str`) – Input radiology text phrase.

- **interpolation** (`str`) – Interpolation method to upsample the heatmap so it matches the input image size. See `torch.nn.functional.interpolate()` for more details.

Return type `ndarray`

Returns A heatmap of the similarities between each patch embedding from the image and the text embedding, with the same shape as the input image.

get_similarity_score_from_raw_data(*image_path*, *query_text*)

Compute the cosine similarity score between an image and one or more strings.

If multiple strings are passed, their embeddings are averaged before L2-normalization.

Parameters

- **image_path** (`Path`) – Path to the input chest X-ray, either a DICOM or JPEG file.
- **query_text** (`Union`[`List`[`str`], `str`]) – Input radiology text phrase.

Return type `float`

Returns The similarity score between the image and the text.

to(*device*)

Move models to the specified device.

Return type `None`

class `health_multimodal.vlp.`**ImageTextInferenceEngine**(*image_inference_engine*, *text_inference_engine*)

Functions related to inference on `ImageTextModel`.

static **convert_similarity_to_image_size**(*similarity_map*, *width*, *height*, *resize_size*, *crop_size*, *val_img_transform=None*, *interpolation='nearest'*)

Convert similarity map from raw patch grid to original image size, taking into account whether the image has been resized and/or cropped prior to entering the network.

Return type `ndarray`

get_similarity_map_from_raw_data(*image_path*, *query_text*, *interpolation='nearest'*)

Return a heatmap of the similarities between each patch embedding from the image and the text embedding.

Parameters

- **image_path** (`Path`) – Path to the input chest X-ray, either a DICOM or JPEG file.
- **query_text** (`str`) – Input radiology text phrase.
- **interpolation** (`str`) – Interpolation method to upsample the heatmap so it matches the input image size. See `torch.nn.functional.interpolate()` for more details.

Return type `ndarray`

Returns A heatmap of the similarities between each patch embedding from the image and the text embedding, with the same shape as the input image.

get_similarity_score_from_raw_data(*image_path*, *query_text*)

Compute the cosine similarity score between an image and one or more strings.

If multiple strings are passed, their embeddings are averaged before L2-normalization.

Parameters

- **image_path** (`Path`) – Path to the input chest X-ray, either a DICOM or JPEG file.
- **query_text** (`Union`[`List`[`str`], `str`]) – Input radiology text phrase.

Return type `float`

Returns The similarity score between the image and the text.

`to`(*device*)
Move models to the specified device.

Return type `None`

26.2 Image processing

image	Image-related tools

26.2.1 health_multimodal.image

Image-related tools

inference_engine
utils

health_multimodal.image.inference_engine

Classes

ImageInferenceEngine(image_model, transform)	Encapsulate inference-time operations on an image model.

`class health_multimodal.image.inference_engine.`**`ImageInferenceEngine`**(*image_model*, *transform*)
Encapsulate inference-time operations on an image model.

Parameters

- **`img_model`** – Trained image model
- **`transform`** (`Compose`) – Transform to apply to the image after loading. Must return a torch.Tensor that can be input directly to the image model.

`get_projected_global_embedding`(*image_path*)
Compute global image embedding in the joint latent space.

Parameters **`image_path`** (`Path`) – Path to the image to compute embeddings for.

Return type `Tensor`

Returns Torch tensor containing l2-normalised global image embedding [joint_feature_dim,] where joint_feature_dim is the dimensionality of the joint latent space.

`get_projected_patch_embeddings`(*image_path*)
Compute image patch embeddings in the joint latent space, preserving the image grid.

Parameters **`image_path`** (`Path`) – Path to the image to compute embeddings for.

Return type `Tuple[Tensor, Tuple[int, int]]`

Returns A tuple containing the image patch embeddings and the shape of the original image (width, height) before applying transforms.

`load_and_transform_input_image`(*image_path*, *transform*)

Read an image and apply the transform to it.

1. Read the image from the given path
2. Apply transform
3. Add the batch dimension
4. Move to the correct device

Parameters `return_original_shape` – Whether to return an extra tuple that has the original shape of the image before the transforms. The tuple returned contains (width, height).

Return type `Tuple[Tensor, Tuple[int, int]]`

health_multimodal.image.utils

Functions

get_image_inference([image_model_type])	Create a `ImageInferenceEngine` for the image model.

Classes

ImageModelType(value)	An enumeration.

class `health_multimodal.image.utils.`**`ImageModelType`**(*value*)

An enumeration.

`health_multimodal.image.utils.`**`get_image_inference`**(*image_model_type=ImageModelType.BIOVIL_T*)

Create a `ImageInferenceEngine` for the image model.

Parameters `image_model_type` (*ImageModelType*) – The type of image model to use, *BIOVIL* or *BIOVIL_T*.

The model is downloaded from the Hugging Face Hub. The engine can be used to get embeddings from text prompts or masked token predictions.

Return type *ImageInferenceEngine*

io
transforms

health_multimodal.image.data.io

Functions

load_image(path)	Load an image from disk.
remap_to_uint8(array[, percentiles])	Remap values in input so the output range is $[0, 255]$.

`health_multimodal.image.data.io.`**`load_image`**(*path*)

Load an image from disk.

The image values are remapped to $[0, 255]$ and cast to 8-bit unsigned integers.

Parameters **`path`** (`Path`) – Path to image.

Return type `Image`

Returns Image as `Pillow` `Image`.

`health_multimodal.image.data.io.`**`remap_to_uint8`**(*array*, *percentiles=None*)

Remap values in input so the output range is $[0, 255]$.

Percentiles can be used to specify the range of values to remap. This is useful to discard outliers in the input data.

Parameters

- **`array`** (`ndarray`) – Input array.
- **`percentiles`** (`Optional[Tuple[float, float]]`) – Percentiles of the input values that will be mapped to `0` and `255`. Passing `None` is equivalent to using percentiles `(0, 100)` (but faster).

Return type `ndarray`

Returns Array with `0` and `255` as minimum and maximum values.

health_multimodal.image.data.transforms

Functions

create_chest_xray_transform_for_inference(...)	Defines the image transformation pipeline for Chest-Xray datasets.
infer_resize_params(val_img_transforms)	Given the validation transforms pipeline, extract the sizes to which the image was resized and cropped, if any.

Classes

ExpandChannels()	Transforms an image with one channel to an image with three channels by copying pixel intensities of the image along the 1st dimension.

`class health_multimodal.image.data.transforms.`**`ExpandChannels`**

Transforms an image with one channel to an image with three channels by copying pixel intensities of the image along the 1st dimension.

health_multimodal.image.data.transforms.**create_chest_xray_transform_for_inference**(*resize*, *center_crop_size*)

Defines the image transformation pipeline for Chest-Xray datasets.

Parameters

- **resize** (int) – The size to resize the image to. Linear resampling is used. Resizing is applied on the axis with smaller shape.
- **center_crop_size** (int) – The size to center crop the image to. Square crop is applied.

Return type Compose

health_multimodal.image.data.transforms.**infer_resize_params**(*val_img_transforms*)

Given the validation transforms pipeline, extract the sizes to which the image was resized and cropped, if any.

Return type Tuple[Optional[int], Optional[int]]

encoder
model
modules
resnet
transformer
types

health_multimodal.image.model.encoder

Functions

get_encoder_from_type(img_encoder_type)	Returns the encoder class for the given encoder type.
get_encoder_output_dim(module, device)	Calculate the output dimension of an encoder by making a single forward pass.
restore_training_mode(module)	Restore the training mode of a module after some operation.

Classes

ImageEncoder(img_encoder_type)	Image encoder trunk module for the ImageModel class.
MultiImageEncoder(img_encoder_type)	Multi-image encoder trunk module for the ImageModel class.

class health_multimodal.image.model.encoder.**ImageEncoder**(*img_encoder_type*)

Image encoder trunk module for the ImageModel class.

:param img_encoder_type [Type of image encoder model to use, either "resnet18_multi_image" or] "resnet50_multi_image".

Initializes internal Module state, shared by both nn.Module and ScriptModule.

`forward`(*current_image*, *return_patch_embeddings=False*)
Get image global and patch embeddings

Return type `Union[Tensor, Tuple[Tensor, Tensor]]`

`reload_encoder_with_dilation`(*replace_stride_with_dilation=None*)
Workaround for enabling dilated convolutions after model initialization.

Parameters **replace_stride_with_dilation** (`Optional[Sequence[bool]]`) – Replace the 2x2 standard convolution stride with a dilated convolution in each layer in the last three blocks of ResNet architecture.

Return type `None`

`class health_multimodal.image.model.encoder.`**`MultiImageEncoder`**(*img_encoder_type*)
Multi-image encoder trunk module for the `ImageModel` class. It can be used to encode multiple images into combined latent representation. Currently it only supports two input images but can be extended to support more in future.

Parameters **img_encoder_type** (`str`) – Type of image encoder model to use: either `"resnet18"` or `"resnet50"`.

Initializes internal Module state, shared by both nn.Module and ScriptModule.

`forward`(*current_image*, *previous_image=None*, *return_patch_embeddings=False*)
Get image global and patch embeddings

Return type `Union[Tensor, Tuple[Tensor, Tensor]]`

`reload_encoder_with_dilation`(*replace_stride_with_dilation=None*)
Workaround for enabling dilated convolutions after model initialization.

Parameters **replace_stride_with_dilation** (`Optional[Sequence[bool]]`) – Replace the 2x2 standard convolution stride with a dilated convolution in each layer in the last three blocks of ResNet architecture.

Return type `None`

`health_multimodal.image.model.encoder.`**`get_encoder_from_type`**(*img_encoder_type*)
Returns the encoder class for the given encoder type.

Parameters **img_encoder_type** (`str`) – Encoder type. {RESNET18, RESNET50, RESNET18_MULTI_IMAGE, RESNET50_MULTI_IMAGE}

Return type *`ImageEncoder`*

`health_multimodal.image.model.encoder.`**`get_encoder_output_dim`**(*module*, *device*)
Calculate the output dimension of an encoder by making a single forward pass.

Parameters

- **module** (`Module`) – Encoder module.
- **device** (`device`) – Compute device to use.

Return type `int`

`health_multimodal.image.model.encoder.`**`restore_training_mode`**(*module*)
Restore the training mode of a module after some operation.

Parameters **module** (`Module`) – PyTorch module.

Return type `Generator[None, None, None]`

health_multimodal.image.model.model

Classes

BaseImageModel()	Abstract class for image models.
ImageModel(img_encoder_type, joint_feature_size)	Image encoder module
MultiImageModel(**kwargs)	Initializes internal Module state, shared by both nn.Module and ScriptModule.

`class health_multimodal.image.model.model.`**`BaseImageModel`**

Abstract class for image models.

Initializes internal Module state, shared by both nn.Module and ScriptModule.

`abstract forward``(`**args*, ***kwargs*`)`

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

Return type *ImageModelOutput*

`class health_multimodal.image.model.model.`**`ImageModel`**`(`*img_encoder_type*, *joint_feature_size*, *freeze_encoder=False*, *pretrained_model_path=None*, ***downstream_classifier_kwargs*`)`

Image encoder module

Initializes internal Module state, shared by both nn.Module and ScriptModule.

`create_downstream_classifier``(`***kwargs*`)`

Create the classification module for the downstream task.

Return type *MultiTaskModel*

`forward``(`*x*`)`

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

Return type *ImageModelOutput*

`get_patchwise_projected_embeddings``(`*input_img*, *normalize*`)`

Get patch-wise projected embeddings from the CNN model.

Parameters

- **`input_img`** (`Tensor`) – input tensor image [B, C, H, W].

- **normalize** (`bool`) – If `True`, the embeddings are L2-normalized.

Returns projected_embeddings tensor of embeddings in shape [batch, n_patches_h, n_patches_w, feature_size].

Return type `Tensor`

train(*mode=True*)
Switch the model between training and evaluation modes.

Return type `Any`

class health_multimodal.image.model.model.**MultiImageModel**(***kwargs*)
Initializes internal Module state, shared by both nn.Module and ScriptModule.

forward(*current_image*, *previous_image=None*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

Return type *ImageModelOutput*

health_multimodal.image.model.modules

Classes

MLP(input_dim, output_dim[, hidden_dim, ...])	Fully connected layers to map between image embeddings and projection space where pairs of images are compared.
MultiTaskModel(input_dim, ...)	Torch module for multi-task classification heads.

class health_multimodal.image.model.modules.**MLP**(*input_dim*, *output_dim*, *hidden_dim=None*, *use_1x1_convs=False*)
Fully connected layers to map between image embeddings and projection space where pairs of images are compared.

Parameters

- **input_dim** (`int`) – Input embedding feature size
- **hidden_dim** (`Optional[int]`) – Hidden layer size in MLP
- **output_dim** (`int`) – Output projection size
- **use_1x1_convs** (`bool`) – Use 1x1 conv kernels instead of 2D linear transformations for speed and memory efficiency.

Initializes internal Module state, shared by both nn.Module and ScriptModule.

forward(*x*)
forward pass of the multi-layer perceptron

Return type `Tensor`

class health_multimodal.image.model.modules.**MultiTaskModel**(*input_dim*, *classifier_hidden_dim*, *num_classes*, *num_tasks*)

Torch module for multi-task classification heads. We create a separate classification head for each task and perform a forward pass on each head independently in forward(). Classification heads are instances of *MLP*.

Parameters

- **input_dim** (int) – Number of dimensions of the input feature map.
- **classifier_hidden_dim** (Optional[int]) – Number of dimensions of hidden features in the MLP.
- **num_classes** (int) – Number of output classes per task.
- **num_tasks** (int) – Number of classification tasks or heads required.

Initializes internal Module state, shared by both nn.Module and ScriptModule.

forward(*x*)

Returns [batch_size, num_tasks, num_classes] tensor of logits.

Return type Tensor

health_multimodal.image.model.resnet

Functions

resnet18([pretrained, progress])	ResNet-18 model from "Deep Residual Learning for Image Recognition".
resnet50([pretrained, progress])	ResNet-50 model from "Deep Residual Learning for Image Recognition".

Classes

ResNetHIML(**kwargs)	Wrapper class of the original torchvision ResNet model.

class health_multimodal.image.model.resnet.**ResNetHIML**(***kwargs*)

Wrapper class of the original torchvision ResNet model.

The forward function is updated to return the penultimate layer activations, which are required to obtain image patch embeddings.

Initializes internal Module state, shared by both nn.Module and ScriptModule.

forward(*x*, *return_intermediate_layers=False*)

ResNetHIML forward pass. Optionally returns intermediate layers using the `return_intermediate_layers` argument.

Parameters **return_intermediate_layers** (bool) – If `True`, return layers x0-x4 as a tuple, otherwise return x4 only.

Return type Union[Tensor, Tuple[Tensor, Tensor, Tensor, Tensor, Tensor]]

health_multimodal.image.model.resnet.**resnet18**(*pretrained=False*, *progress=True*, ***kwargs*)

ResNet-18 model from "Deep Residual Learning for Image Recognition".

Parameters

- **pretrained** (bool) – If True, returns a model pre-trained on ImageNet.
- **progress** (bool) – If True, displays a progress bar of the download to stderr.

Return type *ResNetHIML*

health_multimodal.image.model.resnet.**resnet50**(*pretrained=False*, *progress=True*, ***kwargs*)

ResNet-50 model from "Deep Residual Learning for Image Recognition".

Parameters

- **pretrained** (bool) – If True, returns a model pre-trained on ImageNet
- **progress** (bool) – If True, displays a progress bar of the download to stderr.

Return type *ResNetHIML*

health_multimodal.image.model.transformer

Classes

Block(dim, num_heads[, mlp_ratio, qkv_bias, ...])	Encapsulates multi-layer perceptron and multi-head self attention modules into a block.
MultiHeadAttentionLayer(dim[, num_heads, ...])	Multi-head self attention module
MultiHeadAttentionOutput(mha_output[, attention])	
SinePositionEmbedding([embedding_dim, ...])	This is a more standard version of the position embedding, very similar to the one used by the Attention is all you need paper, generalized to work on images.
VisionTransformerPooler(input_dim, grid_shape)	**type input_dim** int

class health_multimodal.image.model.transformer.**Block**(*dim*, *num_heads*, *mlp_ratio=1.0*, *qkv_bias=False*, *drop=0.0*, *attn_drop=0.0*, *drop_path=0.0*, *act_layer=<class 'torch.nn.modules.activation.GELU'>*, *norm_layer=<class 'torch.nn.modules.normalization.LayerNorm'>*)

Encapsulates multi-layer perceptron and multi-head self attention modules into a block.

The content builds on top of the TIMM library (vision_transformer.py) and differs by the following:

- **This implementation uses spatio-temporal positional embeddings instead of 2D positional embeddings only,** and they are taken into account within the forward pass of each ViT block.
- **Utilises the custom defined *MultiHeadAttentionLayer* which does not apply *self-attention* only but can be** generalised to arbitrary (query, key, value) tuples. This can be valuable to process more than 2 scans.

Positional and type embeddings are handled in a similar fashion as DETR object localisation paper https://alcinos.github.io/detr_page/, where a fixed set of sine/cos positional embeddings are used in an additive manner to Q and K tensors.

Initializes internal Module state, shared by both nn.Module and ScriptModule.

`forward`(*x*, *pos_and_type_embed*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

Return type `Tensor`

`class health_multimodal.image.model.transformer.`**`MultiHeadAttentionLayer`**(*dim*, *num_heads=8*, *qkv_bias=False*, *attn_drop=0.0*, *proj_drop=0.0*)

Multi-head self attention module

The content builds on top of the TIMM library (vision_transformer.py) and differs by the following:

- **Defines a custom *MultiHeadAttentionLayer* which does not only apply *self-attention* but it can be** generalised to arbitrary (query, key, value) input tuples. This feature can be valuable to process more than 2 scans at a time.
- *Self-attention* specific use-case can still be invoked by calling the *forward_as_mhsa* method.

Initializes internal Module state, shared by both nn.Module and ScriptModule.

`forward`(*k*, *q*, *v*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

Return type *`MultiHeadAttentionOutput`*

`class health_multimodal.image.model.transformer.`**`MultiHeadAttentionOutput`**(*mha_output*, *attention=None*)

`class health_multimodal.image.model.transformer.`**`SinePositionEmbedding`**(*embedding_dim=64*, *temperature=10000*, *normalize=False*, *scale=None*)

This is a more standard version of the position embedding, very similar to the one used by the Attention is all you need paper, generalized to work on images.

class health_multimodal.image.model.transformer.**VisionTransformerPooler**(*input_dim*, *grid_shape*, *num_heads=8*, *num_blocks=3*, *norm_layer=functools.partial(<class 'torch.nn.modules.normalization.LayerNo* *eps=1e-06)*)

Parameters

- **input_dim** (int) – Input feature dimension (i.e., channels in old CNN terminology)
- **grid_shape** (Tuple[int, int]) – Shape of the grid of patches per image
- **num_heads** (int) – Number of self-attention heads within the MHA block
- **num_blocks** (int) – Number of blocks per attention layer
- **norm_layer** (Any) – Normalisation layer

***self.type_embed*: Is used to characterise prior and current scans, and** create permutation variance across modalities/series.

Initializes internal Module state, shared by both nn.Module and ScriptModule.

forward(*current_image*, *previous_image=None*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

Return type `Tensor`

health_multimodal.image.model.types

Classes

ImageEncoderType(value)	An enumeration.
ImageEncoderWeightTypes(value)	An enumeration.
ImageModelOutput(img_embedding, ...)	

class health_multimodal.image.model.types.**ImageEncoderType**(*value*)

An enumeration.

class health_multimodal.image.model.types.**ImageEncoderWeightTypes**(*value*)

An enumeration.

class health_multimodal.image.model.types.**ImageModelOutput**(*img_embedding*, *patch_embeddings*, *projected_global_embedding*, *class_logits*, *projected_patch_embeddings*) `class health_multimodal.image.`**`BaseImageModel`**

Abstract class for image models.

Initializes internal Module state, shared by both nn.Module and ScriptModule.

`abstract` **`forward`**`(`**args*, ***kwargs*`)`

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

Return type *`ImageModelOutput`*

`class health_multimodal.image.`**`ImageEncoderType`**`(`*value*`)`

An enumeration.

`class health_multimodal.image.`**`ImageInferenceEngine`**`(`*image_model*, *transform*`)`

Encapsulate inference-time operations on an image model.

Parameters

- **`img_model`** – Trained image model
- **`transform`** (`Compose`) – Transform to apply to the image after loading. Must return a torch.Tensor that can be input directly to the image model.

`get_projected_global_embedding``(`*image_path*`)`

Compute global image embedding in the joint latent space.

Parameters **`image_path`** (`Path`) – Path to the image to compute embeddings for.

Return type `Tensor`

Returns Torch tensor containing l2-normalised global image embedding [joint_feature_dim,] where joint_feature_dim is the dimensionality of the joint latent space.

`get_projected_patch_embeddings``(`*image_path*`)`

Compute image patch embeddings in the joint latent space, preserving the image grid.

Parameters **`image_path`** (`Path`) – Path to the image to compute embeddings for.

Return type `Tuple[Tensor, Tuple[int, int]]`

Returns A tuple containing the image patch embeddings and the shape of the original image (width, height) before applying transforms.

`load_and_transform_input_image``(`*image_path*, *transform*`)`

Read an image and apply the transform to it.

1. Read the image from the given path
2. Apply transform
3. Add the batch dimension
4. Move to the correct device

Parameters **`return_original_shape`** – Whether to return an extra tuple that has the original shape of the image before the transforms. The tuple returned contains (width, height).

Return type Tuple[Tensor, Tuple[int, int]]

`class health_multimodal.image.`**`ImageModel`**(*img_encoder_type*, *joint_feature_size*, *freeze_encoder=False*, *pretrained_model_path=None*, ***downstream_classifier_kwargs*)

Image encoder module

Initializes internal Module state, shared by both nn.Module and ScriptModule.

`create_downstream_classifier`(***kwargs*)

Create the classification module for the downstream task.

Return type *MultiTaskModel*

`forward`(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

Return type *ImageModelOutput*

`get_patchwise_projected_embeddings`(*input_img*, *normalize*)

Get patch-wise projected embeddings from the CNN model.

Parameters

- **input_img** (`Tensor`) – input tensor image [B, C, H, W].
- **normalize** (`bool`) – If `True`, the embeddings are L2-normalized.

Returns projected_embeddings tensor of embeddings in shape [batch, n_patches_h, n_patches_w, feature_size].

Return type Tensor

`train`(*mode=True*)

Switch the model between training and evaluation modes.

Return type Any

`health_multimodal.image.`**`get_image_inference`**(*image_model_type=ImageModelType.BIOVIL_T*)

Create a *ImageInferenceEngine* for the image model.

Parameters **image_model_type** (*ImageModelType*) – The type of image model to use, *BIOVIL* or *BIOVIL_T*.

The model is downloaded from the Hugging Face Hub. The engine can be used to get embeddings from text prompts or masked token predictions.

Return type *ImageInferenceEngine*

26.3 Text processing

text	Text-related tools

26.3.1 health_multimodal.text

Text-related tools

inference_engine
utils

health_multimodal.text.inference_engine

Classes

TextInferenceEngine(tokenizer, text_model)	Text inference class that implements functionalities required to extract sentence embedding, similarity and MLM prediction tasks.

`class health_multimodal.text.inference_engine.`**`TextInferenceEngine`**(*tokenizer*, *text_model*)

Text inference class that implements functionalities required to extract sentence embedding, similarity and MLM prediction tasks.

Parameters

- **`tokenizer`** (`BertTokenizer`) – A BertTokenizer object.
- **`text_model`** (`BertForMaskedLM`) – Text model either default HuggingFace class

`get_embeddings_from_prompt`(*prompts*, *normalize=True*, *verbose=True*)

Generate L2-normalised embeddings for a list of input text prompts.

Parameters

- **`prompts`** (`Union[str, List[str]]`) – Input text prompt(s) either in string or list of string format.
- **`normalize`** (`bool`) – If True, L2-normalise the embeddings.
- **`verbose`** (`bool`) – If set to True, tokenized words are displayed in the console.

Return type `Tensor`

Returns Tensor of shape (batch_size, embedding_size).

`get_pairwise_similarities`(*prompt_set_1*, *prompt_set_2*)

Compute pairwise cosine similarities between the embeddings of the given prompts.

Return type `Tensor`

`is_in_eval`()

Returns True if the model is in eval mode.

Return type `bool`

predict_masked_tokens(*prompts*)

Predict masked tokens for a single or list of input text prompts.

Requires models to be trained with a MLM prediction head.

Parameters **prompts** (Union[str, List[str]]) – Input text prompt(s) either in string or list of string format.

Return type List[List[str]]

Returns Predicted token candidates (Top-1) at masked position.

tokenize_input_prompts(*prompts*, *verbose=True*)

Tokenizes the input sentence(s) and adds special tokens as defined by the tokenizer. :type prompts: Union[str, List[str]] :param prompts: Either a string containing a single sentence, or a list of strings each containing

a single sentence. Note that this method will not correctly tokenize multiple sentences if they are input as a single string.

Parameters **verbose** (bool) – If set to True, will log the sentence after tokenization.

Return type Any

Returns A 2D tensor containing the tokenized sentences

health_multimodal.text.utils

Functions

get_bert_inference([bert_encoder_type])	Create a TextInferenceEngine for a text encoder model.
get_biovil_t_bert()	Load the BioViL-T Bert model and tokenizer from the Hugging Face Hub.
get_cxr_bert()	Load the CXR-BERT model and tokenizer from the Hugging Face Hub.

Classes

BertEncoderType(value)	An enumeration.

class health_multimodal.text.utils.**BertEncoderType**(*value*)

An enumeration.

health_multimodal.text.utils.**get_bert_inference**(*bert_encoder_type=BertEncoderType.BIOVIL_T_BERT*)

Create a TextInferenceEngine for a text encoder model.

Parameters **bert_encoder_type** (*BertEncoderType*) – The type of text encoder model to use, *CXR_BERT* or *BIOVIL_T_BERT*.

The model is downloaded from the Hugging Face Hub. The engine can be used to get embeddings from text prompts or masked token predictions.

Return type *TextInferenceEngine*

`health_multimodal.text.utils.`**`get_biovil_t_bert`**`()`

Load the BioViL-T Bert model and tokenizer from the Hugging Face Hub.

Return type `Tuple`[*`CXRBertTokenizer`*, *`CXRBertModel`*]

`health_multimodal.text.utils.`**`get_cxr_bert`**`()`

Load the CXR-BERT model and tokenizer from the Hugging Face Hub.

Return type `Tuple`[*`CXRBertTokenizer`*, *`CXRBertModel`*]

io

health_multimodal.text.data.io

Classes

TextInput(tokenizer)	Text input class that can be used for inference and deployment.

class `health_multimodal.text.data.io.`**`TextInput`**`(`*tokenizer*`)`

Text input class that can be used for inference and deployment.

Implements tokenizer related operations and ensure that input strings conform with the standards expected from a BERT model.

Parameters **`tokenizer`** (`BertTokenizer`) – A BertTokenizer object.

`assert_special_tokens_not_present``(`*prompt*`)`

Check if the input prompts contain special tokens.

Return type `None`

`tokenize_input_prompts``(`*prompts*, *verbose*`)`

Tokenizes the input sentence(s) and adds special tokens as defined by the tokenizer. :type prompts: `Union`[`str`, `List`[`str`]] :param prompts: Either a string containing a single sentence, or a list of strings each containing

a single sentence. Note that this method will not correctly tokenize multiple sentences if they are input as a single string.

Parameters **`verbose`** (`bool`) – If set to True, will log the sentence after tokenization.

Return type `Any`

Returns A 2D tensor containing the tokenized sentences

configuration_cxrbert
modelling_cxrbert

health_multimodal.text.model.configuration_cxrbert

Classes

CXRBertConfig([projection_size])	Config class for CXR-BERT model.
CXRBertTokenizer(**kwargs)	

class health_multimodal.text.model.configuration_cxrbert.**CXRBertConfig**(*projection_size=128*, ***kwargs*)

Config class for CXR-BERT model.

Parameters **projection_size** (int) – Dimensionality of the joint latent space.

class health_multimodal.text.model.configuration_cxrbert.**CXRBertTokenizer**(***kwargs*)

health_multimodal.text.model.modelling_cxrbert

Classes

BertProjectionHead(config)	Projection head to be used with BERT CLS token.
CXRBertModel(config)	Implements the CXR-BERT model outlined in the manuscript: Boecking et al. "Making the Most of Text Semantics to Improve Biomedical Vision-Language Processing", 2022 https://link.springer.com/chapter/10.1007/978-3-031-20059-5_1.
CXRBertOutput	

class health_multimodal.text.model.modelling_cxrbert.**BertProjectionHead**(*config*)

Projection head to be used with BERT CLS token.

This is similar to `BertPredictionHeadTransform` in HuggingFace.

Parameters **config** (*CXRBertConfig*) – Configuration for BERT.

Initializes internal Module state, shared by both nn.Module and ScriptModule.

forward(*hidden_states*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

Return type Tensor

class health_multimodal.text.model.modelling_cxrbert.**CXRBertModel**(*config*)

Implements the CXR-BERT model outlined in the manuscript: Boecking et al. "Making the Most of Text Semantics to Improve Biomedical Vision-Language Processing", 2022 https://link.springer.com/chapter/10.1007/978-3-031-20059-5_1

Extends the HuggingFace BertForMaskedLM model by adding a separate projection head. The projection "[CLS]" token is used to align the latent vectors of image and text modalities.

Initializes internal Module state, shared by both nn.Module and ScriptModule.

`config_class`
alias of `health_multimodal.text.model.configuration_cxrbert.CXRBertConfig`

`forward`(*input_ids*, *attention_mask*, *token_type_ids=None*, *position_ids=None*, *head_mask=None*, *inputs_embeds=None*, *output_attentions=None*, *output_hidden_states=None*, *output_cls_projected_embedding=None*, *return_dict=None*, ***kwargs*)
The [*BertForMaskedLM*] forward method, overrides the *__call__* special method.

<Tip>

Although the recipe for forward pass needs to be defined within this function, one should call the [*Module*] instance afterwards instead of this since the former takes care of running the pre and post processing steps while the latter silently ignores them.

</Tip>

Parameters

- **`input_ids`** (*torch.LongTensor* of shape *(batch_size, sequence_length)*) – Indices of input sequence tokens in the vocabulary.

  Indices can be obtained using [*BertTokenizer*]. See [*PreTrainedTokenizer.encode*] and [*PreTrainedTokenizer.__call__*] for details.

  [What are input IDs?](../glossary#input-ids)

- **`attention_mask`** (*torch.FloatTensor* of shape *(batch_size, sequence_length)*, *optional*) – Mask to avoid performing attention on padding token indices. Mask values selected in *[0, 1]*:

  – 1 for tokens that are **not masked**,

  – 0 for tokens that are **masked**.

  [What are attention masks?](../glossary#attention-mask)

- **`token_type_ids`** (*torch.LongTensor* of shape *(batch_size, sequence_length)*, *optional*) – Segment token indices to indicate first and second portions of the inputs. Indices are selected in *[0, 1]*:

  – 0 corresponds to a *sentence A* token,

  – 1 corresponds to a *sentence B* token.

  [What are token type IDs?](../glossary#token-type-ids)

- **`position_ids`** (*torch.LongTensor* of shape *(batch_size, sequence_length)*, *optional*) – Indices of positions of each input sequence tokens in the position embeddings. Selected in the range *[0, config.max_position_embeddings - 1]*.

  [What are position IDs?](../glossary#position-ids)

- **`head_mask`** (*torch.FloatTensor* of shape *(num_heads,)* or *(num_layers, num_heads)*, *optional*) – Mask to nullify selected heads of the self-attention modules. Mask values selected in *[0, 1]*:

  – 1 indicates the head is **not masked**,

  – 0 indicates the head is **masked**.

- **inputs_embeds** (*torch.FloatTensor* of shape *(batch_size, sequence_length, hidden_size)*, *optional*) – Optionally, instead of passing *input_ids* you can choose to directly pass an embedded representation. This is useful if you want more control over how to convert *input_ids* indices into associated vectors than the model's internal embedding lookup matrix.
- **output_attentions** (*bool*, *optional*) – Whether or not to return the attentions tensors of all attention layers. See *attentions* under returned tensors for more detail.
- **output_hidden_states** (*bool*, *optional*) – Whether or not to return the hidden states of all layers. See *hidden_states* under returned tensors for more detail.
- **return_dict** (*bool*, *optional*) – Whether or not to return a [*~file_utils.ModelOutput*] instead of a plain tuple.
- **labels** (*torch.LongTensor* of shape *(batch_size, sequence_length)*, *optional*) – Labels for computing the masked language modeling loss. Indices should be in *[-100, 0, ..., config.vocab_size]* (see *input_ids* docstring) Tokens with indices set to *-100* are ignored (masked), the loss is only computed for the tokens with labels in *[0, ..., config.vocab_size]*

Returns

A [*transformers.modeling_outputs.MaskedLMOutput*] or a tuple of *torch.FloatTensor* (if *return_dict=False* is passed or when *config.return_dict=False*) comprising various elements depending on the configuration ([*BertConfig*]) and inputs.

- **loss** (*torch.FloatTensor* of shape *(1,)*, *optional*, returned when *labels* is provided) – Masked language modeling (MLM) loss.
- **logits** (*torch.FloatTensor* of shape *(batch_size, sequence_length, config.vocab_size)*) – Prediction scores of the language modeling head (scores for each vocabulary token before SoftMax).
- **hidden_states** (*tuple(torch.FloatTensor)*, *optional*, returned when *output_hidden_states=True* is passed or when *config.output_hidden_states=True*) – Tuple of *torch.FloatTensor* (one for the output of the embeddings + one for the output of each layer) of shape *(batch_size, sequence_length, hidden_size)*.

  Hidden-states of the model at the output of each layer plus the initial embedding outputs.
- **attentions** (*tuple(torch.FloatTensor)*, *optional*, returned when *output_attentions=True* is passed or when *config.output_attentions=True*) – Tuple of *torch.FloatTensor* (one for each layer) of shape *(batch_size, num_heads, sequence_length, sequence_length)*.

  Attentions weights after the attention softmax, used to compute the weighted average in the self-attention heads.

Return type [*transformers.modeling_outputs.MaskedLMOutput*] or *tuple(torch.FloatTensor)*

Example:

```python >>> from transformers import BertTokenizer, BertForMaskedLM >>> import torch

```
>>> tokenizer = BertTokenizer.from_pretrained("bert-base-uncased")
>>> model = BertForMaskedLM.from_pretrained("bert-base-uncased")
```

```
>>> inputs = tokenizer("The capital of France is [MASK].", return_tensors="pt")
>>> labels = tokenizer("The capital of France is Paris.", return_tensors="pt")[
↪"input_ids"]
```
```

```
>>> outputs = model(**inputs, labels=labels)
>>> loss = outputs.loss
>>> logits = outputs.logits
```
```

**get_projected_text_embeddings**(*input_ids*, *attention_mask*, *normalize_embeddings=True*)

Returns l2-normalised projected cls token embeddings for the given input token ids and attention mask. The joint latent space is trained using a contrastive objective between image and text data modalities.

**Parameters**

- **input_ids** (Tensor) – (batch_size, sequence_length)
- **attention_mask** (Tensor) – (batch_size, sequence_length)
- **normalize_embeddings** (bool) – Whether to l2-normalise the embeddings.

**Return type** Tensor

**Returns** (batch_size, projection_size)

class health_multimodal.text.model.modelling_cxrbert.**CXRBertOutput**

class health_multimodal.text.**CXRBertConfig**(*projection_size=128*, ***kwargs*)

Config class for CXR-BERT model.

**Parameters** **projection_size** (int) – Dimensionality of the joint latent space.

class health_multimodal.text.**CXRBertModel**(*config*)

Implements the CXR-BERT model outlined in the manuscript: Boecking et al. "Making the Most of Text Semantics to Improve Biomedical Vision-Language Processing", 2022 https://link.springer.com/chapter/10.1007/978-3-031-20059-5_1

Extends the HuggingFace BertForMaskedLM model by adding a separate projection head. The projection "[CLS]" token is used to align the latent vectors of image and text modalities.

Initializes internal Module state, shared by both nn.Module and ScriptModule.

**config_class**

alias of *health_multimodal.text.model.configuration_cxrbert.CXRBertConfig*

**forward**(*input_ids*, *attention_mask*, *token_type_ids=None*, *position_ids=None*, *head_mask=None*, *inputs_embeds=None*, *output_attentions=None*, *output_hidden_states=None*, *output_cls_projected_embedding=None*, *return_dict=None*, ***kwargs*)

The [*BertForMaskedLM*] forward method, overrides the *__call__* special method.

<Tip>

Although the recipe for forward pass needs to be defined within this function, one should call the [*Module*] instance afterwards instead of this since the former takes care of running the pre and post processing steps while the latter silently ignores them.

</Tip>

**Parameters**

- **input_ids** (*torch.LongTensor* of shape *(batch_size, sequence_length)*) – Indices of input sequence tokens in the vocabulary.

  Indices can be obtained using [*BertTokenizer*]. See [*PreTrainedTokenizer.encode*] and [*PreTrainedTokenizer.__call__*] for details.

  [What are input IDs?](../glossary#input-ids)
```

- `attention_mask` (*torch.FloatTensor* of shape *(batch_size, sequence_length)*, *optional*) – Mask to avoid performing attention on padding token indices. Mask values selected in *[0, 1]*:
  - 1 for tokens that are **not masked**,
  - 0 for tokens that are **masked**.

  [What are attention masks?](../glossary#attention-mask)
- `token_type_ids` (*torch.LongTensor* of shape *(batch_size, sequence_length)*, *optional*) – Segment token indices to indicate first and second portions of the inputs. Indices are selected in *[0, 1]*:
  - 0 corresponds to a *sentence A* token,
  - 1 corresponds to a *sentence B* token.

  [What are token type IDs?](../glossary#token-type-ids)
- `position_ids` (*torch.LongTensor* of shape *(batch_size, sequence_length)*, *optional*) – Indices of positions of each input sequence tokens in the position embeddings. Selected in the range *[0, config.max_position_embeddings - 1]*.

  [What are position IDs?](../glossary#position-ids)
- `head_mask` (*torch.FloatTensor* of shape *(num_heads,)* or *(num_layers, num_heads)*, *optional*) – Mask to nullify selected heads of the self-attention modules. Mask values selected in *[0, 1]*:
  - 1 indicates the head is **not masked**,
  - 0 indicates the head is **masked**.
- `inputs_embeds` (*torch.FloatTensor* of shape *(batch_size, sequence_length, hidden_size)*, *optional*) – Optionally, instead of passing *input_ids* you can choose to directly pass an embedded representation. This is useful if you want more control over how to convert *input_ids* indices into associated vectors than the model's internal embedding lookup matrix.
- `output_attentions` (*bool*, *optional*) – Whether or not to return the attentions tensors of all attention layers. See *attentions* under returned tensors for more detail.
- `output_hidden_states` (*bool*, *optional*) – Whether or not to return the hidden states of all layers. See *hidden_states* under returned tensors for more detail.
- `return_dict` (*bool*, *optional*) – Whether or not to return a [*~file_utils.ModelOutput*] instead of a plain tuple.
- `labels` (*torch.LongTensor* of shape *(batch_size, sequence_length)*, *optional*) – Labels for computing the masked language modeling loss. Indices should be in *[-100, 0, ..., config.vocab_size]* (see *input_ids* docstring) Tokens with indices set to *-100* are ignored (masked), the loss is only computed for the tokens with labels in *[0, ..., config.vocab_size]*

Returns

A [*transformers.modeling_outputs.MaskedLMOutput*] or a tuple of *torch.FloatTensor* (if *return_dict=False* is passed or when *config.return_dict=False*) comprising various elements depending on the configuration ([*BertConfig*]) and inputs.

- **loss** (*torch.FloatTensor* of shape *(1,)*, *optional*, returned when *labels* is provided) – Masked language modeling (MLM) loss.

- **logits** (*torch.FloatTensor* of shape *(batch_size, sequence_length, config.vocab_size)*) – Prediction scores of the language modeling head (scores for each vocabulary token before SoftMax).
- **hidden_states** (*tuple(torch.FloatTensor)*, *optional*, returned when *output_hidden_states=True* is passed or when *config.output_hidden_states=True*) – Tuple of *torch.FloatTensor* (one for the output of the embeddings + one for the output of each layer) of shape *(batch_size, sequence_length, hidden_size)*.

  Hidden-states of the model at the output of each layer plus the initial embedding outputs.
- **attentions** (*tuple(torch.FloatTensor)*, *optional*, returned when *output_attentions=True* is passed or when *config.output_attentions=True*) – Tuple of *torch.FloatTensor* (one for each layer) of shape *(batch_size, num_heads, sequence_length, sequence_length)*.

  Attentions weights after the attention softmax, used to compute the weighted average in the self-attention heads.

Return type [*transformers.modeling_outputs.MaskedLMOutput*] or *tuple(torch.FloatTensor)*

Example:

```python >>> from transformers import BertTokenizer, BertForMaskedLM >>> import torch

```
>>> tokenizer = BertTokenizer.from_pretrained("bert-base-uncased")
>>> model = BertForMaskedLM.from_pretrained("bert-base-uncased")
```

```
>>> inputs = tokenizer("The capital of France is [MASK].", return_tensors="pt")
>>> labels = tokenizer("The capital of France is Paris.", return_tensors="pt")[
↪"input_ids"]
```

```
>>> outputs = model(**inputs, labels=labels)
>>> loss = outputs.loss
>>> logits = outputs.logits
```
```

get_projected_text_embeddings(*input_ids*, *attention_mask*, *normalize_embeddings=True*)

Returns l2-normalised projected cls token embeddings for the given input token ids and attention mask. The joint latent space is trained using a contrastive objective between image and text data modalities.

Parameters

- **input_ids** (Tensor) – (batch_size, sequence_length)
- **attention_mask** (Tensor) – (batch_size, sequence_length)
- **normalize_embeddings** (bool) – Whether to l2-normalise the embeddings.

Return type Tensor

Returns (batch_size, projection_size)

class health_multimodal.text.**CXRBertOutput**

class health_multimodal.text.**CXRBertTokenizer**(***kwargs*)

class health_multimodal.text.**TextInferenceEngine**(*tokenizer*, *text_model*)

Text inference class that implements functionalities required to extract sentence embedding, similarity and MLM prediction tasks.

Parameters

- **tokenizer** (BertTokenizer) – A BertTokenizer object.

- **`text_model`** (`BertForMaskedLM`) – Text model either default HuggingFace class

`get_embeddings_from_prompt`(*prompts*, *normalize=True*, *verbose=True*)
Generate L2-normalised embeddings for a list of input text prompts.

Parameters

- **`prompts`** (`Union[str, List[str]]`) – Input text prompt(s) either in string or list of string format.
- **`normalize`** (`bool`) – If True, L2-normalise the embeddings.
- **`verbose`** (`bool`) – If set to True, tokenized words are displayed in the console.

Return type `Tensor`

Returns Tensor of shape (batch_size, embedding_size).

`get_pairwise_similarities`(*prompt_set_1*, *prompt_set_2*)
Compute pairwise cosine similarities between the embeddings of the given prompts.

Return type `Tensor`

`is_in_eval`()
Returns True if the model is in eval mode.

Return type `bool`

`predict_masked_tokens`(*prompts*)
Predict masked tokens for a single or list of input text prompts.

Requires models to be trained with a MLM prediction head.

Parameters **`prompts`** (`Union[str, List[str]]`) – Input text prompt(s) either in string or list of string format.

Return type `List[List[str]]`

Returns Predicted token candidates (Top-1) at masked position.

`tokenize_input_prompts`(*prompts*, *verbose=True*)
Tokenizes the input sentence(s) and adds special tokens as defined by the tokenizer. :type prompts: `Union[str, List[str]]` :param prompts: Either a string containing a single sentence, or a list of strings each containing

a single sentence. Note that this method will not correctly tokenize multiple sentences if they are input as a single string.

Parameters **`verbose`** (`bool`) – If set to True, will log the sentence after tokenization.

Return type `Any`

Returns A 2D tensor containing the tokenized sentences

`health_multimodal.text.`**`get_bert_inference`**(*bert_encoder_type=BertEncoderType.BIOVIL_T_BERT*)
Create a *`TextInferenceEngine`* for a text encoder model.

Parameters **`bert_encoder_type`** (*`BertEncoderType`*) – The type of text encoder model to use, *CXR_BERT* or *BIOVIL_T_BERT*.

The model is downloaded from the Hugging Face Hub. The engine can be used to get embeddings from text prompts or masked token predictions.

Return type *`TextInferenceEngine`*

26.4 Common utils

common	General utils

26.4.1 health_multimodal.common

General utils

device
visualization

health_multimodal.common.device

Functions

get_module_device(module)	Returns the device of the module

`health_multimodal.common.device.`**`get_module_device`**`(`*module*`)`

Returns the device of the module

Return type `device`

health_multimodal.common.visualization

Functions

plot_phrase_grounding_similarity_map(...[, ...])	Plot visualization of the input image, the similarity heatmap and the heatmap isolines.

`health_multimodal.common.visualization.`**`plot_phrase_grounding_similarity_map`**`(`*image_path*, *similarity_map*, *bboxes=None*`)`

Plot visualization of the input image, the similarity heatmap and the heatmap isolines.

Parameters

- **`image_path`** (`Path`) – Path to the input image.
- **`similarity_map`** (`ndarray`) – Phase grounding similarity map of the same size as the image.
- **`bboxes`** (`Optional[List[Tuple[float, float, float, float]]]`) – Optional list of bounding boxes to plot on the image.

Return type `Figure`

CHAPTER

TWENTYSEVEN

TRAINING OF SELF-SUPERVISED MODELS

The code present in the InnerEye/ML/SSL folder allows you to train self-supervised models using SimCLR or BYOL. This code runs as a " bring-your-own-model" self-contained module (see docs/bring_your_own_model.md) .

Here, we provide implementations for four datasets to get you kickstarted with self-supervised models:

- Toy CIFAR datasets: CIFAR10 and CIFAR100
- Medical Chest-Xray datasets: RSNA Pneumonia Detection Challenge data (30k scans, labels indicating presence of pneumonia), NIH Chest-Xray (112k Chest-Xray scans) or CheXpert (228k scans).

27.1 Multi-dataset support

During self-supervised training, a separate linear classifier is trained on top of learnt image embeddings. In this way, users can continuously monitor the representativeness of learnt image embeddings for a given downstream classification task. More importantly, the framework allows users to specify multiple datasets and data loaders for SimCLR/BYOL training and evaluation. For instance, a BYOL encoder can be learnt using a dataset that does not contain any target labels and embeddings can be evaluated throughout training on a separate dataset containing class labels. To enable this functionality, our SSLContainer takes two dataset names parameters: `ssl_training_dataset_name` to indicate which dataset to use for SSL training and `linear_head_dataset_name` to indicate which dataset for the classification task used to monitor embeddings quality during training.

27.2 Quick start guide

Here we described how to quickly start a training job with our ready made configs.

27.2.1 Example 1: training a SimCLR or BYOL model on CIFAR10

To kick-off a training for a SimCLR and BYOL models on CIFAR10, simply run

```
python ML/runner.py --model=CIFAR10BYOL
python ML/runner.py --model=CIFAR10SimCLR
```

For this dataset, it will automatically take care of downloading the dataset to your machine prior to starting training.

27.2.2 Example 2: training a BYOL model on Chest-Xray data

Step 0: Get the data

If you run on your local machine:

Prior to starting training a model on this dataset, you will need to download it from Kaggle to your machine:

- To use the RSNA Pneumonia Detection Challenge data: please download from here. Make sure to download all images and the `dataset.csv` file to your data folder. Please note that the labels are here merely used for monitoring purposes.
- To get the NIH dataset: please download from here. Make sure you include also the csv files in your data folder, the code assumes the dataset files and all the images lie in your data folder as downloaded from Kaggle. In particular, do not modify the original csv filenames (e.g. the code expects to find the `Data_Entry_2017.csv` file within the data directory).
- To get the CheXpert data: please download from here, the code assumes the dataset files and all the images lie in your data folder as downloaded.

If you run on AML

In order to train models on AML you will need to upload the datasets listed above to your storage account and get their dataset_id to pass to your model config.

Step 1: Update your model config

We provide sample configs to train models on NIH data both for BYOL and SimCLR. You can use them as they are except for the dataset location fields:

- If you're running locally set the `local_dataset` parameter to point to your data folder.
- If you're running on AML: you need to update the `RSNA_AZURE_DATASET_ID` and `NIH_AZURE_DATASET_ID` variables to point to the name of the NIH and Kaggle dataset in your own workspace.

Step 2: Launch the training job

Example to train a SSL model with BYOL on the NIH dataset and monitor the embeddings quality on the Kaggle RSNA Pneumonia Challenge classification task:

```
python ML/runner.py --model=NIH_RSNA_BYOL
```

27.3 Configuring your own SSL models:

27.3.1 About SSLContainer configuration

All SSL models are derived from the SSLcontainer class. See the config class in ML/configs/ssl for some examples of specific model configurations (all derived from this container).

If you wish to create your own model config for SSL training, you will need to create a child class and parametrize it with the following available arguments:

- `ssl_training_dataset_name`: the name of the dataset to train the SSL encoder on, a member of the SSL-DatasetName class (don't forget to update this class if you're adding a new dataset ;)),
- `linear_head_dataset_name`: the name of the dataset to train to linear head on top of the classifier for monitoring purposes,
- `azure_dataset_id`: the id of the AML dataset to use for SSL training,
- `extra_azure_dataset_ids`: dataset_id to use for linear head training, expected to be provided as a list [data-id],
- `ssl_encoder`: name of the encoder to train, member of `EncoderName` class, currently supported are resnet50, resnet101 and densenet121,
- `ssl_training_type`: which SSL algorithm to use, member of `SSLType` choice between BYOL and SimCLR,
- `ssl_training_batch_size`: batch size of SSL training. This is the number of examples processed by a single GPU. Multiply this by the number of GPUs to get the effective batch size.
- `linear_head_batch_size`: batch size for linear head training (used for monitor of SSL embeddings quality). This is the number of examples processed by a single GPU. Multiply this by the number of GPUs to get the effective batch size.
- `ssl_augmentation_config`: path to yaml config for augmentation to use during SSL training. Only used for NIH/Kaggle datasets.
- `linear_head_augmentation_config`: path to yaml config for augmentation to use for linear head training. Only used for NIH/Kaggle datasets,
- `use_balanced_binary_loss_for_linear_head`: whether to use balanced loss for linear head training,
- `random_seed`: seed for the run,
- `num_epochs`: number of epochs to train for.

In case you wish to first test your model locally, here some optional arguments that can be useful:

- `local_dataset`: path to local dataset, if passed the azure dataset will be ignored
- `is_debug_model`: if True it will only run on the first batch of each epoch
- `drop_last`: if False (True by default) it will keep the last batch also if incomplete

27.3.2 Creating your own datamodules:

To use this code with your own data, you will need to:

1. Define your own Lightening Container that inherits from `SSLContainer` as described in the paragraph above.
2. Create a dataset class that reads your new dataset, inheriting from both `VisionDataset` and `DatasetWithReturnIndex`. See for example how we constructed `RSNAKaggleCXR` class. WARNING: the first positional argument of your dataset class constructor MUST be the data directory ("root"), as VisionDataModule expects this in the prepare_data step.
3. In your own container update the `DatasetToClassMapping` member of the class so that the code knows which data class to associate to your new dataset name.
4. Create a yaml configuration file that contains the augmentations specific to your dataset. The yaml file will be consumed by the `create_transforms_from_config` function defined in the `InnerEye.ML.augmentations.transform_pipeline` module (see next paragraph for more details). Alternatively, overwrite the `_get_transforms` method. To simplify this step, we have defined a series of standard operations in `SSL/transforms_utils.py` . You could for example construct a transform pipeline similar to the one created inside `create_transform_from_config` inside your own method.

5. Update all necessary parameters in the model config (cf. previous paragraph)

Once all these steps are updated, the code in the base SSLContainer class will take care of creating the corresponding datamodules for SSL training and linear head monitoring.

27.3.3 About the augmentation configuration yaml file

The augmentations used for SSL training for all Chest-X-rays models are parametrized via a yaml config file. The path to this config as to be passed in the model config (cf. section above). We provide two defaults configs: `cxr_ssl_encoder_augmentations.yaml` is used to define the augmentations used for BYOL/SimCLR training; the `cxr_linear_head_augmentations.yaml` config defines the augmentations to used for the training of the linear head (used for monitoring purposes). The meaning of each config argument is detailed in `ssl_model_config.py`

WARNING: this file will be ignored for CIFAR examples where we use the default pl-bolts augmentations.

27.4 Finetuning a linear head on top of a pretrained SSL model.

Alongside with the modules to train your SSL models, we also provide examplary modules that allow you to build a classifier on top of a pretrained SSL model. The base class for these modules is `SSLClassifierContainer`. It builds on top of the `SSLContainer` with additional command line arguments allowing you to specify where to find the checkpoint for your pretrained model. For this you have two options:

- If you are running locally, you can provide the local path to your pretrained model checkpoint via `--src_checkpoint`.
- If your are running on AML, use the `pretraining_run_recovery_id` field. Providing this field, will mean that AML will automatically download the checkpoints to the current node, will pick the latest checkpoint to build the classifier on top. Beware not to confuse `pretraining_run_recovery_id` with `run_recovery_id` as the latter is use to continue training on the same model (which is not the case here).

The code will then automatically extract the encoder part of the loaded SSL model to initialize your classifier. You can then also choose whether you want to freeze the weights of your encoder or not via the `--freeze_encoder=True/False` argument. By default, this is set to True.

27.4.1 Example for CIFAR

We provide an example of such a classifier container for CIFAR named `SSLClassifierCIFAR`. To launch a finetuning run for this model on CIFAR10, just run

```
python ML/runner.py --model=SSLClassifierCIFAR --pretraining_run_recovery_id={THE_ID_TO_
↪YOUR_SSL_TRAINING_JOB}
```

27.4.2 Example for CXR

Similarly, we provide class to allow you to simply start a finetuning job for CXR model in `CXRImageClassifier`. By default, this will launch a finetuning job on the RSNA Pneumonia dataset. To start the run:

```
python ML/runner.py --model=CXRImageClassifier --pretraining_run_recovery_id={THE_ID_TO_
↪YOUR_SSL_TRAINING_JOB}
```

or for a local run

```
python ML/runner.py --model=CXRImageClassifier --src_checkpoint={LOCAL_PATH_TO_YOUR_SSL_
↪CHECKPOINT}
```

CHAPTER

TWENTYEIGHT

CONTRIBUTING TO THIS TOOLBOX

We welcome all contributions that help us achieve our aim of speeding up Machine Learning (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
- Bug reports
- Documentation improvements, including fixing typos
- Suggestions about codebase improvements

All contributions to the toolbox need to come with unit tests, and will be reviewed when a pull request 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.

For a full set of design and guidelines, please check our coding guidelines documentation.

CHAPTER

TWENTYNINE

DESIGN AND CODING GUIDELINES

29.1 Design Phase

For non-trivial changes, please communicate your plans early on to reach maximum impact and to avoid duplicate work. Create an issue or a discussion that describes the problem that you are trying to solve, and describe why this adds a positive contribution to the project.

Tag one of the core `hi-ml` team (@ant0nsc, @mebristo, @fepegar) to get input on your problem. In particular, you should find out if this has already been worked on or thought about. This will help you to get to know about existing code fragments that can simplify your task, and integrate better with existing code.

We recommend to work with a core `hi-ml` team member to create a design for your work. Depending on the size of the task at hand, a design can be anywhere between a short description of your plan in a GitHub discussion or issue, or it can be a shared document where you list your design options and invite feedback. Your design partner will later also be well positioned to review your pull request (PR) because they will be already familiar with your plan.

When working on a large contribution, we suggest to break it down into a set of small PRs that are more manageable for you and for the reviewers (see below in the section "Scope of PRs")

29.2 Setting up your development environment

Please see our *detailed instructions*.

29.3 Coding Guidelines

29.3.1 Naming

Please follow the general PEP 8 Python rules for naming:

- Variables, function and method names should use `snake_case` (lower case with under scores separating words)
- Class names should follow `PascalCase` (first letter of each word capitalized).

To improve readability, functions or methods that return boolean values should follow a `is_...`, `has_...`, `use...` pattern, like `is_status_ok` instead of `ok`.

29.3.2 Static Analysis, Linting and Styling

We use `flake8` as a linter, `mypy` and pyright for static typechecking, and `black` for styling. All these tools run as part of the PR workflow, and must run without errors for a contribution to be accepted.

`mypy` requires that all functions and methods carry type annotations. See the `mypy` documentation for more information.

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).

29.3.3 Black styling

We use `black` for styling all of our code. To pass the tests carried out in `make check` (or `make black`) you can run the following command to reformat your changed file as per our guidelines:

```
black <path_to_python_file>
```

You can also run `black` on the whole repository by running the following command from the head of the repo:

```
black .
```

Black VS Code Integration

If you're using VS Code as your IDE, you can avoid having to call `black` every time you want to style some files by installing the `black` formatter extension and adding the following to your vscode `settings.json` file:

```
  "[python]": {
      "editor.formatOnSave": true,
      "editor.defaultFormatter": "ms-python.black-formatter",
    },
```

Now every time you save a file it will automatically be formatted by `black`.

Black pre-commit hook

If you do not style your code yourself, then the pre-commit hook on your PR will attempt to do this automatically. Please be aware that this may cause conflicts if you're working on an open pull request. We highly recommend styling your code before submitting!

29.3.4 Documentation

For general information around docstrings, please check PEP 257.

We use Sphinx for generating documentation. We recommend using a VSCode extension for auto-generating documentation templates, for example `njpwerner.autodocstring`. This extension is already pre-configured in our VSCode workspace settings file.

Reminders about 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 `__init__` method. Hence, there is no need to write any text in the docstring for `__init__` other than the arguments.
- Use >>> to include code snippets.

To generate the Sphinx documentation on your machine, run

```
cd docs
make html
```

Then open the results in `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. Should end with a period."""

    def __init__(self, bar: float = 0.5) -> None:
        """
        :param bar: This is a description for the constructor argument.
            Long descriptions should be indented.
        """
        self.bar = bar

    def method(self, arg: int) -> None:
        """Short method description, followed by an empty line. Sentences should end␣
↪with a period.

        Longer descriptions can be added as well.
        Argument names like ``arg`` are rendered nicely if enclosed in double backticks.

        :param arg: This is a description for the method argument.
            Long descriptions should be indented.
        :raises ValueError: If something bad happens.
        """
        do_something()  # comments should start with "  # "
```

For more information, check the Sphinx-RTD tutorial.

29.4 Testing

29.4.1 Why do we write unit tests?

If you think that there are sensitive parts in your code, where hidden errors can impact the project outcome, please take your time to protect parts of your code. As the team grows, so does the number of users of the repository, and we may not be able to monitor changes on a frequent basis, but your tests can ensure that the expected behavior is preserved. We are not making it strictly mandatory, however, contributors should take into account that their code may have negative downstream impact.

We do not need to test every part of the code depending on the project constraints, e.g., quality and test-coverage vs learning and hypothesis-testing. However, we expect a justification in each PR for this decision and it should be reviewed by team members. Please regularly monitor the code coverage results that are posted as comments on each pull request.

In particular, please do write unit tests for anything that affects training /results, such as data preprocessing, losses, and metrics.

29.4.2 Testing Do's / Don'ts

- 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 on your development machine.
- DO NOT rely only on the test results in the cloud (i.e., run test locally before submitting). Apart from the obvious delay in getting your results, CI runs 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.

29.4.3 Testing of Scripts

Depending on the project needs, we may write scripts for doing one-off operations (as an example, have a look at himl_download.py. How carefully should we test those?

- Any scripts that perform operations that we anticipate others to need as well should be designed for re-usability and be tested. "Designed for re-use" here would mean, for example: The script should not contain any hard-coded setup that is specific to my machine or user account. If that's not achievable, document carefully what others need to prepare so that they can run this script.
- Scripts are inherently difficult to test. Testing becomes a lot easier if the script is mainly a front-end to functions that live somewhere else in the codebase.If the script is written as a thin wrapper around library functions, these library functions can be tested in isolation as part of the normal unit test suite.

Writing arguments parsers is particularly error prone. Consider using automatically generated parsers, like we use in the `hi-ml`: Starting point is a class that describes inputs to a function. A parser can be generated automatically from these classes.

Lastly, if you are considering adding a script to your project, also consider the following: If the script should be called, for example, after an AzureML run to collect results, can this be automated further, and make the script obsolete? Reasons for this approach:

- People tend to forget that there is a script to do X already, and may re-do the task in question manually.
- Any script that requires input from the user also has a chance to be provided with the wrong input, leading to friction or incorrect results. In a programmatic scenario, where the script is called automatically, this chance of errors is greatly minimized.

29.5 What not to commit

- DO NOT check in files taken from or derived from private datasets.
- DO NOT check in any form of credentials, passwords or access tokens.
- Do not check in any code that contains absolute paths (for example, paths that only work on your machine).
- Avoid checking in large files (anything over 100 kB). If you need to commit large files, consider adding them via Git LFS.

29.5.1 Jupyter Notebooks

Notebooks can easily become obsolete over time and may not work with code changes. Also, testing them is generally difficult. Please follow these guidelines for notebooks:

- If you are planning to use a notebook, avoid committing into the repository unless it is part of a project demo.
- If the notebook is used to document results, then please render the notebook and place the results in a separate document outside the repository.
- Bear in mind that notebooks are also an easy way of leaking sensitive data: They can for example contain images derived from private datasets. You should clear the notebook outputs before committing it.

29.6 Review / Pull Requests

29.6.1 Scope of Pull Requests

PRs should ideally implement a single change. If in doubt, err on the side of making the PR too small, rather than too big.

- Small PRs help reduce the load on the reviewer.
- Avoid adding unrelated changes to an existing PR.
- PRs should be modular: we can iterate on PRs, and any positive delta is a contribution.

Please follow the guidance on Github flow.

Try gauging the value of your contribution for yourself by asking the following questions:

- Will this change bring the team closer to achieving their project goals?

- Will someone else understand my code?
- Will they be able to use my code?
- Will they be able to extend or build on top of my code?

29.6.2 Pull Request Process

- When creating a PR, do add a summary of your contribution in the PR description.
- The template PR description also contains a checklist for the PR author.
- Link your PR to a GitHub issue that describes the problem/feature that you are working on.
- For collecting early feedback on your work, please use a Draft Pull Request. These PRs are marked with a grey icon in the Github UI, and send a clear signal that the code there is not yet ready for review. When submitting a draft PR, all the checks will be run as for a normal PR.
- Once your work is ready for review, click the "Ready for Review" button on the Github PR page, and, if you want, assign reviewers for your PR.

29.6.3 Pull Request Titles

To enable good auto-generated changelogs, we prefix all PR titles with a category string, like `BUG: Fix out of bounds error when using small images`. Those category prefixes must be in upper case, followed by a colon (`:`). Valid categories are

- `ENH` for enhancements, new capabilities
- `BUG` for bugfixes
- `STY` for stylistic changes (for example, refactoring) that do not impact the functionality
- `DOC` for changes to documentation only
- `DEL` for removing something from the codebase
- `TEST` for adding or modifying tests
- `FIX` to fix something that is not a `BUG` such as a typo
- `MNT` maintenance, to upgrade package version, packaging, linting, CI etc.
- `PERF` performance related

29.7 Notes on Branching

We should treat the main branch as a collection of code that is

- Of high-quality
- Readable and suitable for re-use
- Well documented

Use a dedicated dev branch for development, debugging and analysis work. Once you are sure that the work in the dev branch adds enough value to reach our project objectives, then use a pull request to get your contribution into the main branch.

CHAPTER

THIRTY

SETTING UP THE DEVELOPMENT ENVIRONMENT

30.1 Development environment

We suggest using Visual Studio Code (VSCode), available for multiple platforms here. On Windows system, we recommend using WSL, the Windows Subsystem for Linux, because some PyTorch features are not available on Windows. Inside VSCode, please install the extensions that are recommended for this project - they are available in `.vscode/extensions.json` in the repository root.

30.2 Opening the repository

Once you have the repository on your computer, you can open either all projects at once or individual projects separately in VSCode.

- To open all projects at once, use VSCode's "Open Workspace from File", and select `himl-projects.code-workspace`.
- To open individual projects, use VSCode's "Open Folder", and select one of the folders `hi-ml-azure`, `hi-ml`, or `hi-ml-cpath`

30.3 Creating a Conda environment

Different projects in this repository use different Conda environments:

- The `himl` Conda environment should be used for work on the `hi-ml` and `hi-ml-azure` projects.
- The `HimlHisto` Conda environment should be used for work on `hi-ml-cpath`.

Please select the right Python interpreter for your project (or all projects if using the `himl-projects` workspace) inside VSCode, by choosing "Python: Select Interpreter" from the command palette (Ctrl-Shift-P on VSCode for Windows)

To create the Conda environment `himl`, please use either

```
conda env create --file hi-ml/environment.yml
```

or use `make` in the repository root folder:

```
make env
```

Please see the project-specific README files for instructions how to set up the other Conda environments.

30.4 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
```

Close your terminal and re-open it, then run:

```
nvm install node
npm install -g pyright
```

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

30.6 Common things to do

The repository contains a makefile with definitions for common operations.

- `make check`: Run `flake8`, `mypy` and `black` 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

30.7 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`.

30.8 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 both of the folders `hi-ml/testhiml` and `hi-ml/testazure` The file `config.json` is already present in `.gitignore`, and will hence not be checked in.

30.9 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).

30.10 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.

30.10.1 Test discovery in VSCode

All tests in the repository should be picked up automatically by VSCode. In particular, this includes the tests in the `hi-ml-cpath` folder, which are not always necessary when working on the core `hi-ml` projects.

30.11 Creating a New Release

To create a new package release, follow these steps:

- 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"

30.12 Troubleshooting

30.12.1 Debugging a test in VSCode fails on Windows

- Symptom: Debugging just does not seem to do anything
- Check: Debug Console shows error `from _sqlite3 import *: ImportError: DLL load failed: The specified module could not be found.`
- Fix: see here
- Run `conda info --envs` to see where your Conda environment lives, then place `sqlite3.dll` into the `DLLs` folder inside of the environment

CHAPTER

THIRTYONE

SOFTWARE DEVELOPMENT PROCESS

This document provides a high-level overview of the software development process that our team uses for their repositories (most importantly InnerEye-DeepLearning and hi-ml).

For detailed guidance for developers, please refer to the *coding guidelines*.

31.1 Version Control

Software code versioning is done via GitHub. The code with the highest level of quality control is in the "main" branch. Ongoing development happens in separate branches. Once development in the branch is finished, a Pull Request (PR) process is started to integrate the branch into "main".

31.2 Development Process

The design and development of the software in this repository is roughly separated into an **initiation**, **prototyping**, and a **finalization** phase. The initiation phase can be skipped for minor changes, for example an update to documentation.

31.2.1 Initiation

During the initiation phase, the following steps are carried out:

- Collection of a set of requirements.
- Creating a suggested design for the change.
- Review of the design with member of the core team.

The deliverables of this phase are a detailed design of the proposed change in a GitHub Issue or a separate document.

31.2.2 Prototyping

The engineering owner of the proposed change will create a branch of the current codebase. This branch is separate from the released (main) branch to not affect any current functionality. In this branch, the engineer will carry out the changes as proposed in the design document.

The engineer will also add additional software tests at different levels (unit tests, integration tests) as appropriate for the design. These tests ensure that the proposed functionality will be maintained in the future.

The deliverable of this phase is a branch in the version control system that contains all proposed changes and a set of software tests.

31.2.3 Finalization

At this point, the engineering owner of the proposed change has carried out all necessary changes in a branch of the codebase. They will now initiate a Pull Request process that consists of the following steps:

- The code will be reviewed by at least 2 engineers. Both need to provide their explicit approval before the proposed change can be integrated in the "main" branch.
- All unit and integration tests will start.
- All automatic code checks will start. These checks will verify the following:
  - Consistency with static typing rules.
  - Ensure that no passwords or other types of credentials are revealed.
  - Ensure that none of the used third-party libraries contains high severity software vulnerabilities that could affect the proposed functionality.

For code to be accepted into the "main" branch, the following criteria need to be satisfied:

- All unit and integration tests pass.
- The code has been reviewed by at least 2 engineers who are members of the core development team. This review will also assess non-quantifiable aspects of the proposed change, such as clarity and readability of the code and quality of the documentation.
- Any comments that have been added throughout the review process need to be resolved.
- All automated checks pass.

Once all the above criteria are satisfied, the branch will be merged into "main".

31.3 Software Configuration Management

Third party libraries (sometimes called Software of Unknown Provenance, SOUP) are consumed via two package management systems:

- Conda
- PyPi

Both of those package management systems maintain strict versioning: once a version of a package is published, it cannot be modified in place. Rather, a new version needs to be released.

Our training and deployment code uses Conda environment files that specify an explicit version of a dependent package to use (for example, `lightning_bolts==0.4.0`). The Conda environment files are also version controlled in GitHub. Any change to a version of a third party library will need to be carried out via the same change management process as a code change, with Pull Request, review, and all tests passing.

The list of third party software is maintained in GitHub in the Conda configuration file, that is `environment.yml` for Linux environments and `environment_win.yml` for Windows environments. For example, this is the latest version of the environment file for the InnerEye-DeepLearning repository.

31.4 Defect Handling

The handling of any bugs or defects discovered is done via GitHub Issues.

When an Issue is created, the team members will get notified. Open Issues are kept in a list sorted by priority. For example, this is the list for all of the InnerEye-related Issues. Priorities are re-assesed regularly, at least once a week.

The Issue(s) with highest priority are assigned to an engineer. The engineer will then analyze the problem, and possibly request more information to reproduce the problem. Requesting more information is also handled in the GitHub Issue. Once the problem is clearly reproduced, the engineer will start to write a fix for the Issue as well as a test that ensures that the fix does indeed solve the reported problem.

Writing the fix and test will follow the process outlined above in the *Prototyping* and *Finalization* sections. In particular, the fix and test will undergo the Pull Request review process before they are merged into the main branch.

31.5 Updates to the Development Process

The development process described here is subject to change. Changes will undergo the review process described in this very document, and will be published on GitHub upon completion.

CHAPTER

THIRTYTWO

DEBUGGING AND PROFILING

While using the hi-ml toolbox modules, you might encounter some errors that require running the code step by step in order to track down the source of these errors. Here are some guidelines to help you debug and/or profile hi-ml deep learning training pipelines.

32.1 Debugging within VS Code

VS code has a great Debugging Support for many programming languages. Make sure to install and enable the Python Extension for VS Code to debug hi-ml toolbox modules built in Python.

32.1.1 Debugging configs

The hi-ml repository is organised in Multi-root workspaces to account for environment differences among the modules and offer flexibility to configure each module seperatly.

We provide a set of example debugging configs for each of hi-ml module:

- launch.json in hi-ml
- launch.json in hi-ml-azure
- launch.json in hi-ml-cpath

VS Code restricts debugging to user-written code only by default. If you want to step through external code and standard libraries functions, set `"justMyCode": false` inside the debugging config block in the `launch.json` file.

In particular, if you would like to debug the current file while breaking through external libraries, navigate to `himl-projects.code-workspace` in the repo root and edit the "launch" block as follows:

```
"launch": {
    "configurations": [
      {
        "name": "Python: Current File",
        "type": "python",
        "request": "launch",
        "program": "${file}",
        "console": "integratedTerminal",
        "justMyCode": false
      },
    ],
}
```

32.1.2 PyTorch Lightning flags for debugging and quick runs

The hi-ml toolbox is built upon PyTorch Lightning (PL) to help you build scalable deep learning models for healthcare and life sciences. Refer to *Running ML Experiments with hi-ml* for detailed instructions on how to build scalable pipelines within hi-ml.

Whether you're building a brand new model, or extending an existing one, you might want to make sure that your code runs as expected locally before submitting a job to AzureML. hi-ml supports a set of debugging flags that triggers PyTorch Lightning Trainer arguments to help you detect any potential errors or bugs at early stage.

These are available as part of the TrainerParams and can be used as extra command line arguments with the hi-ml runner.

- `pl_fast_dev_run`: If set to `n`, runs the pipeline for only `n` batch(es) of train, val and test for only a single epoch. Additionally this flag disables all callbacks and hyperparameters serialization which makes the debugging process very quick. This must be used for debugging purposes only.
- `pl_limit_train_batches`: Limits the training dataset to the given number of batches `n`.
- `pl_limit_val_batches`: Limits the validation dataset to the given number of batches `n`.
- `pl_limit_train_batches`: Limits the test dataset to the given number of batches `n`.

In general, it is very useful to run the following two steps as part of the developement cycle:

1. Make sure all training, validation and test loops complete properly by running the pipeline with a smaller batch size and `pl_fast_dev_run` argument. Add the following to the hi-ml runner command line:

```
--bach-size=2 --pl-fast-dev-run=4
```

1. Make sure the whole pipeline runs properly end to end, including checkpoints callbacks and hyperparameter serialization by running it with a smaller batch size once again while limiting train/val/test batches for few epochs. Add the following arguments to the hi-ml runner command line:

```
--bach-size=2 --pl-limit-train-batches=4 --pl-limit-val-batches=4 --pl-limit-test-
↪batches=4 --max_epochs=4
```

Note: Under the hood, setting `pl-fast-dev-run=n` overrides `pl-limit-train-batches=n`, `pl-limit-val-batches=n`, `pl-limit-train-batches=n`, `max_epochs=1` and disables all callbacks. Please keep in mind that all the above is useful for efficient and quick debugging purposes only.

32.2 Profiling Machine Learning Pipelines

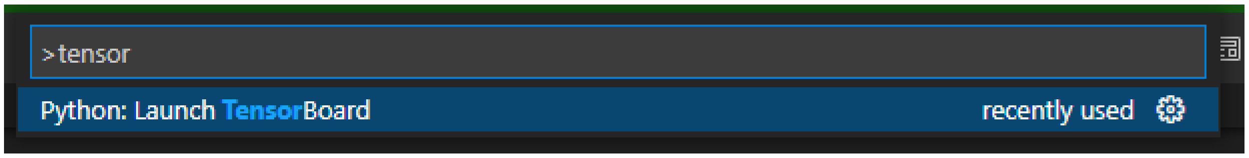
PyTorch Lightning supports a set of built-in profilers that help you identify bottlenecks in your code during training, testing and inference. You can trigger code profiling through the command line argument `--pl_profiler` that you can set to either `simple`, `advanced`, or `pytorch`.

The profiler outputs will be saved in a subfolder `profiler` inside the outputs folder of the run. Give it a try by adding the following arguments to the hi-ml runner:

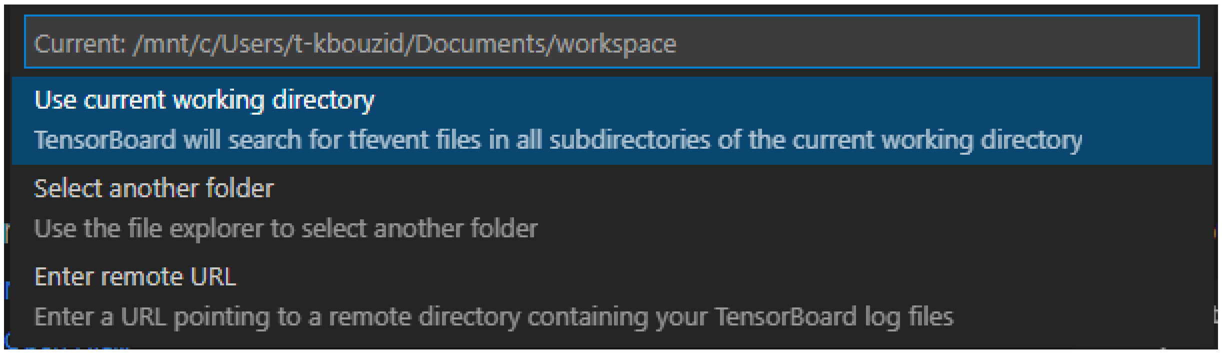
```
--max_epochs=4 --pl-profiler=pytorch
```

32.2.1 Interpret PyTorch Profiling outputs via Tensorboard

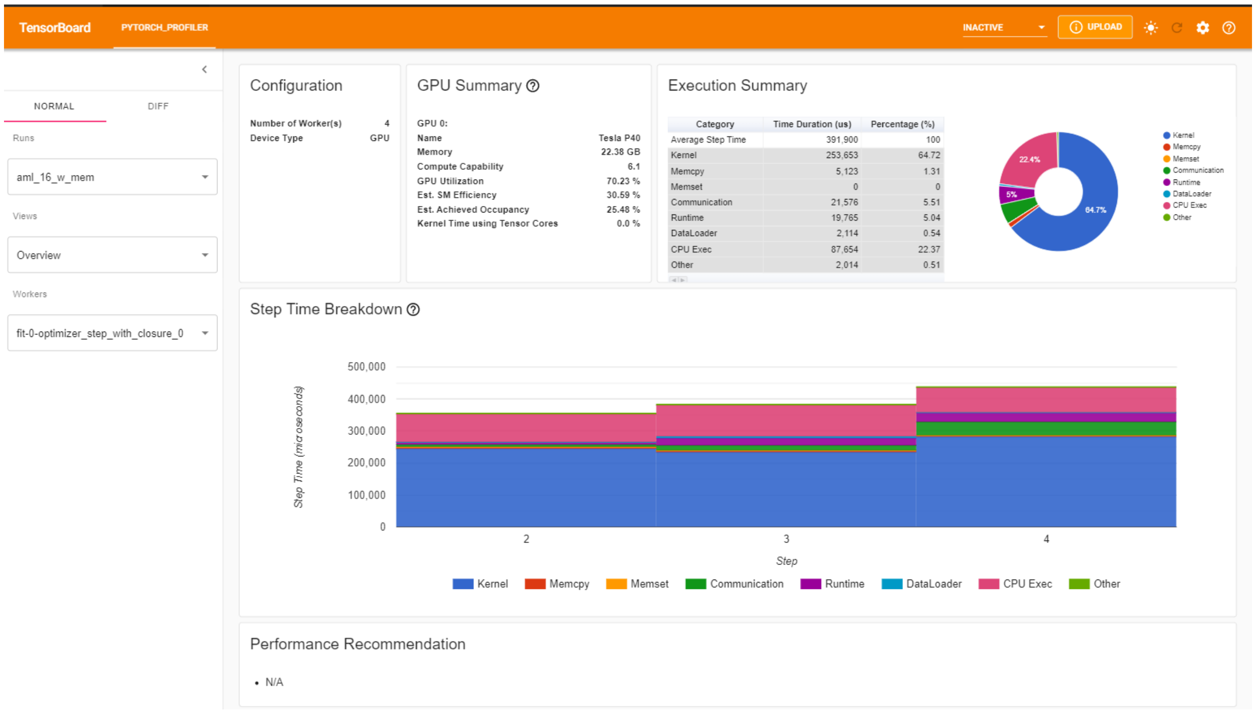
PyTorch Profiler can effectively be interpreted via TensorBoard dashbord interface that is integrated in VS Code as part of the Python extension. Once you have the outputs of the PyTorch Profiler in `outputs/YYYY-MM-DDTHHmmssZ_YourCustomContainer/pytorch_profiler`, you can open the TensorBoard Profiler plugin by launching the Command Palette using the keyboard shortcut CTRL + SHIFT + P (CMD + SHIFT + P on a Mac) and typing the "Launch TensorBoard" command.



Next, you will be asked to select the path where the profiler traces are saved. Select another folder and navigate to `outputs/YYYY-MM-DDTHHmmssZ_YourCustomContainer/pytorch_profiler`.


Current: /mnt/c/Users/t-kbouzid/Documents/workspace
Use current working directory
TensorBoard will search for tfevent files in all subdirectories of the current working directory
Select another folder
Use the file explorer to select another folder
Enter remote URL
Enter a URL pointing to a remote directory containing your TensorBoard log files

You can see Profiler plugin page as shown below. The overview shows a high-level summary of model performance.



Configuration

Number of Worker(s)	4
Device Type	GPU

GPU Summary

GPU 0:	
Name	Tesla P40
Memory	22.38 GB
Compute Capability	6.1
GPU Utilization	70.23 %
Est. SM Efficiency	30.59 %
Est. Achieved Occupancy	25.48 %
Kernel Time using Tensor Cores	0.0 %

Execution Summary

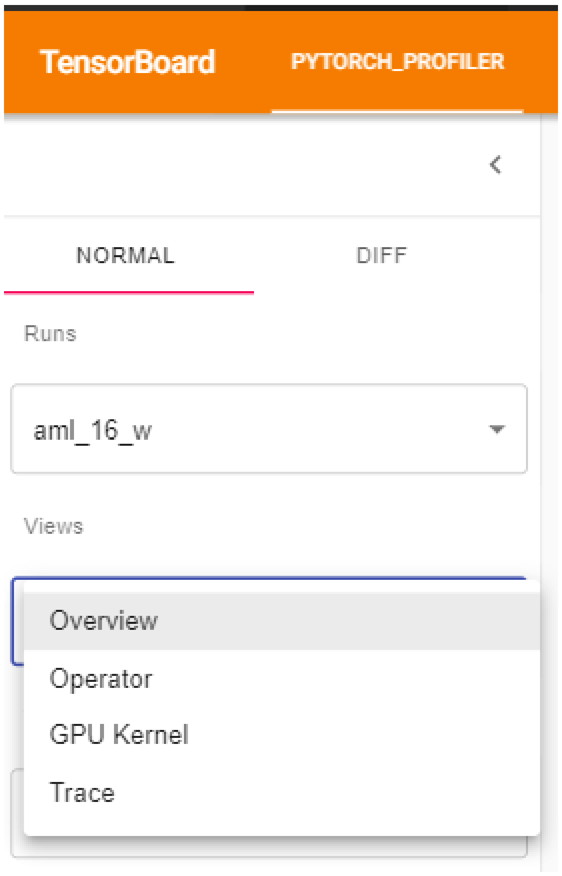
Category	Time Duration (us)	Percentage (%)
Average Step Time	391,900	100
Kernel	253,653	64.72
Memcpy	5,123	1.31
Memset	0	0
Communication	21,576	5.51
Runtime	19,765	5.04
DataLoader	2,114	0.54
CPU Exec	87,654	22.37
Other	2,014	0.51

Step Time Breakdown

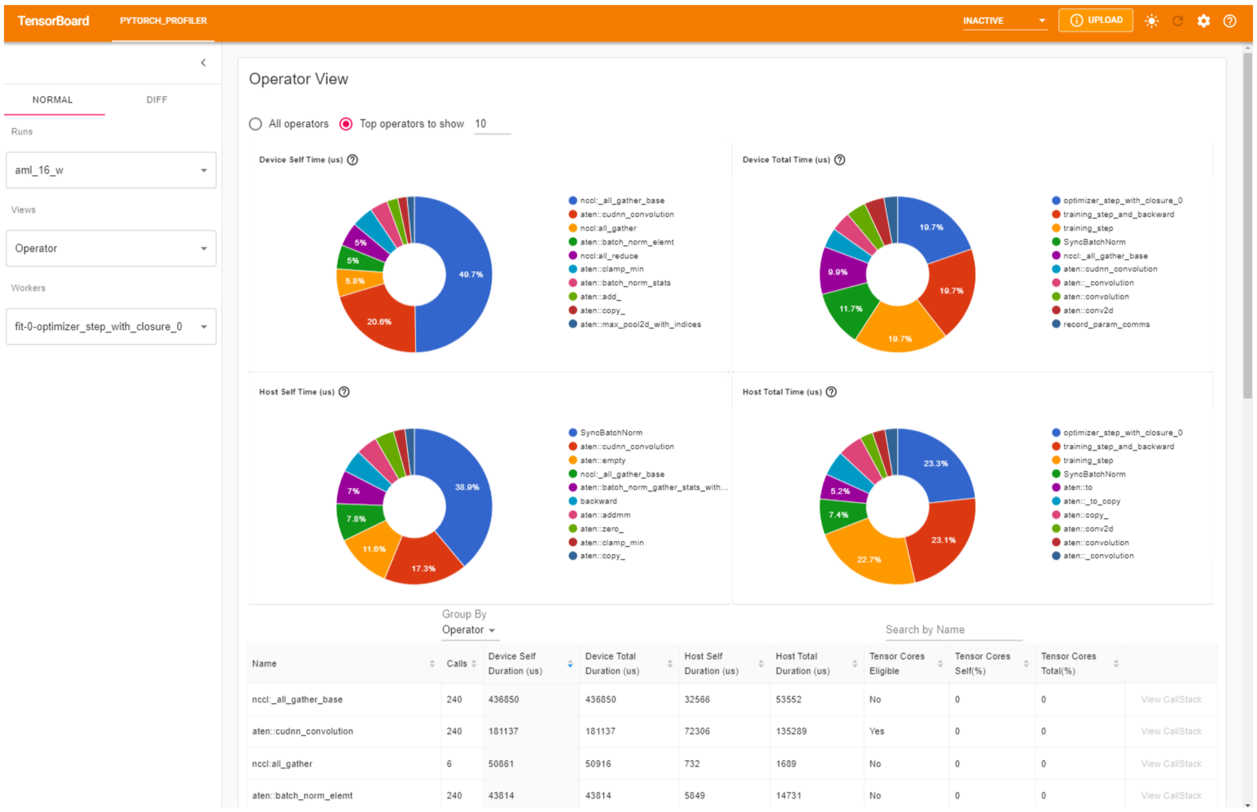
Performance Recommendation

- N/A

You can change the view page in the left dropdown list.



The operator view displays the performance of every PyTorch operator that is executed either on the host or device.



Runs: aml_16_w

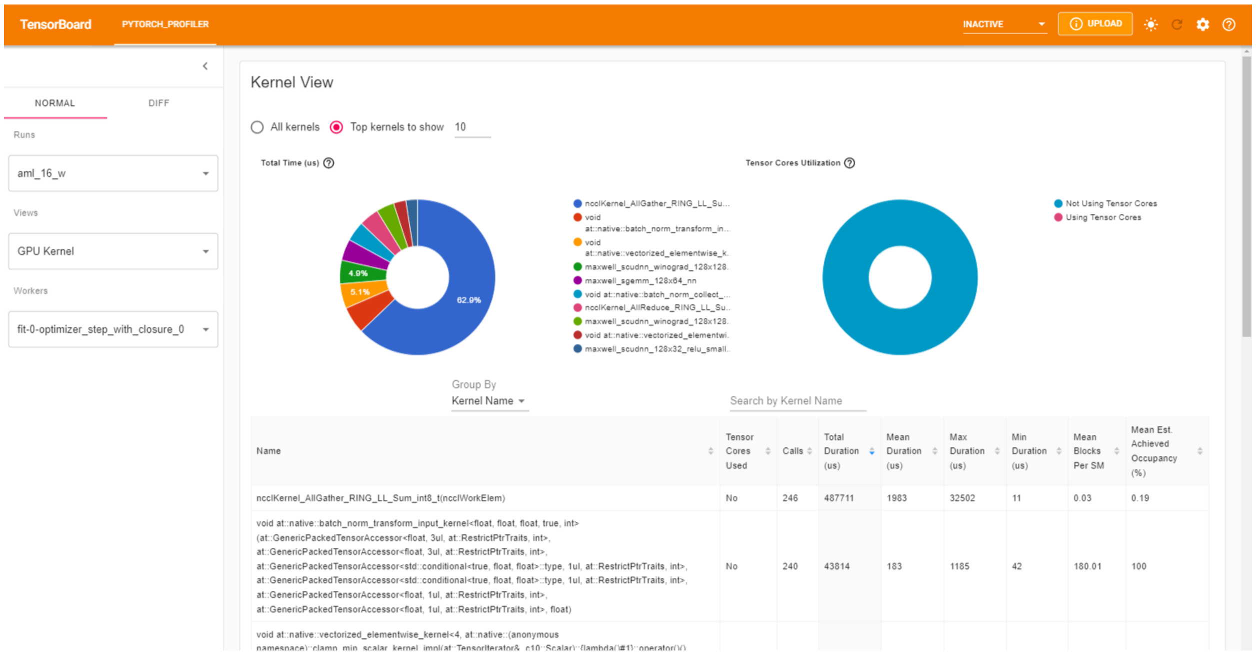
Views: Operator

Workers: fit-0-optimizer_step_with_closure_0

Group By: Operator

Name	Calls	Device Self Duration (us)	Device Total Duration (us)	Host Self Duration (us)	Host Total Duration (us)	Tensor Cores Eligible	Tensor Cores Self(%)	Tensor Cores Total(%)	
nccl:_all_gather_base	240	436850	436850	32566	53552	No	0	0	View CallStack
aten::cudnn_convolution	240	181137	181137	72306	135289	Yes	0	0	View CallStack
nccl:all_gather	6	50861	50916	732	1689	No	0	0	View CallStack
aten::batch_norm_elemt	240	43814	43814	5849	14731	No	0	0	View CallStack

The GPU Kernel panel shows the GPU configuration, GPU usage and Tensor Cores usage. We can see below all kernels' time spent on GPU.


TensorBoard
PYTORCH_PROFILER
INACTIVE
UPLOAD
NORMAL
DIFF
Kernel View
All kernels
Top kernels to show 10
Total Time (us)
Tensor Cores Utilization
ncclKernel_AllGather_RING_LL_Su...
void at::native::batch_norm_transform_in...
void at::native::vectorized_elementwise_k...
maxwell_scudnn_winograd_128x128...
maxwell_sgemm_128x64_nn
void at::native::batch_norm_collect_...
ncclKernel_AllReduce_RING_LL_Su...
maxwell_scudnn_winograd_128x128...
void at::native::vectorized_elementwi...
maxwell_scudnn_128x32_relu_small...
Not Using Tensor Cores
Using Tensor Cores
Search by Kernel Name

Runs: aml_16_w

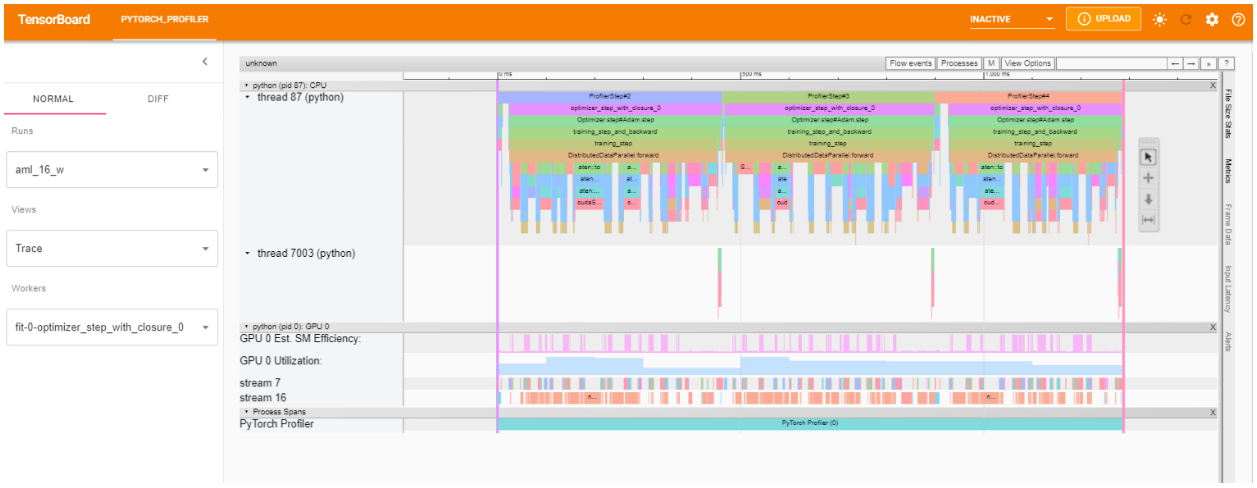
Views: GPU Kernel

Workers: fit-0-optimizer_step_with_closure_0

Group By: Kernel Name

Name	Tensor Cores Used	Calls	Total Duration (us)	Mean Duration (us)	Max Duration (us)	Min Duration (us)	Mean Blocks Per SM	Mean Est. Achieved Occupancy (%)
ncclKernel_AllGather_RING_LL_Sum_int8_t(ncclWorkElem)	No	246	487711	1983	32502	11	0.03	0.19
void at::native::batch_norm_transform_input_kernel<float, float, float, true, int>(at::GenericPackedTensorAccessor<float, 3ul, at::RestrictPtrTraits, int>, at::GenericPackedTensorAccessor<float, 3ul, at::RestrictPtrTraits, int>, at::GenericPackedTensorAccessor<std::conditional<true, float, float>::type, 1ul, at::RestrictPtrTraits, int>, at::GenericPackedTensorAccessor<std::conditional<true, float, float>::type, 1ul, at::RestrictPtrTraits, int>, at::GenericPackedTensorAccessor<float, 1ul, at::RestrictPtrTraits, int>, at::GenericPackedTensorAccessor<float, 1ul, at::RestrictPtrTraits, int>, float)	No	240	43814	183	1185	42	180.01	100
void at::native::vectorized_elementwise_kernel<4, at::native::(anonymous namespace)::clamp_min_scalar_kernel_impl(at::TensorIterator&, c10::Scalar)::{lambda()#1}::operator()()								

The trace view shows timeline of profiled operators and GPU kernels. You can select it to see details as below.



For more details on how to interpret and analyze these views, refer to the pytorch official documentation

32.2.2 Advanced profiling arguments

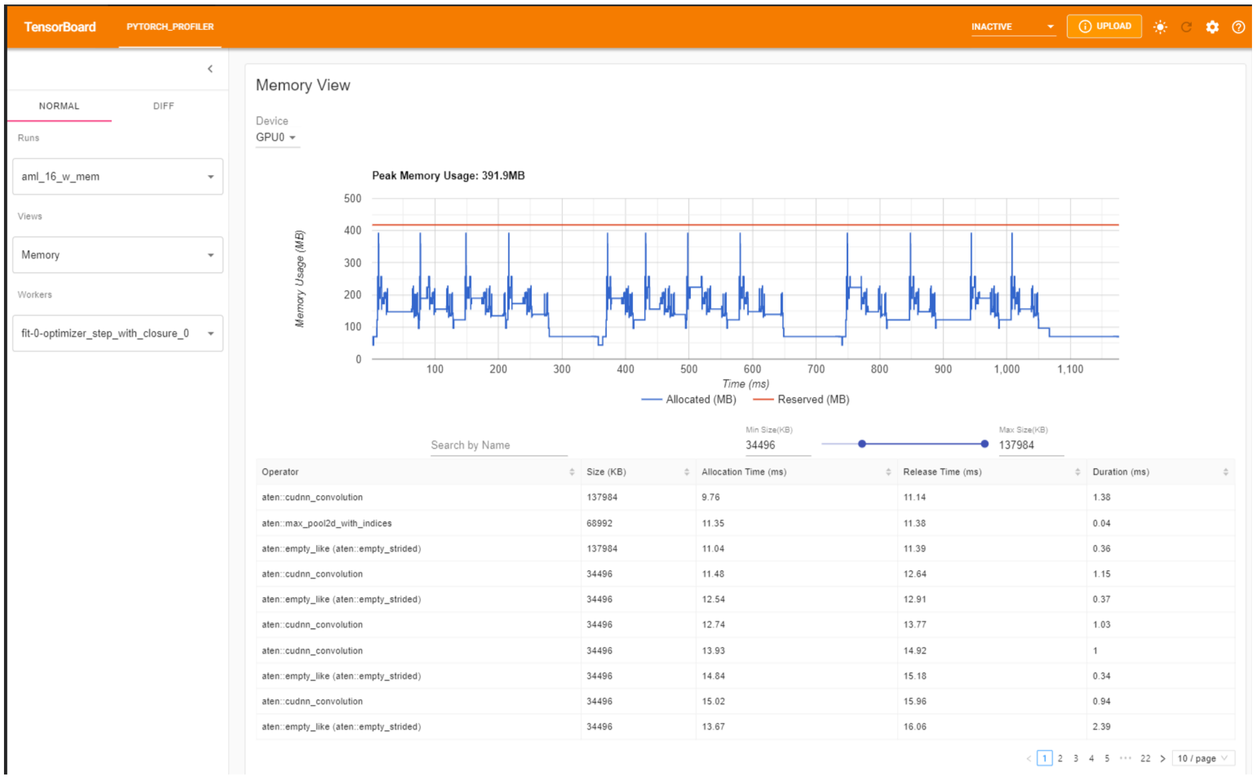
In some scenarios, you might be interested in profiling the memory usage by setting `profile_memory=True` or any of these additional arguments. You can specify additional profiling arguments by overriding get_trainer_arguments in your LightningContainer as shown below. Please make sure to specify your custom profiler under the `profiler` key and properly set `dirpath=self.outputs_folder/"profiler"` so that the profiler's outputs are saved in the right output folder.

```
class YourCustomContainer(LightningContainer):

  def __init__(custom_param: Any) -> None:
    self.custom_param = custom_param

  def get_trainer_arguments(self) -> Dict[str, Any]:
    return {"profiler": PyTorchProfiler(dirpath=self.outputs_folder/"profiler", with_
↪memory=True, with_stack=True)}
```

The profiler will record all memory allocation/release events and allocator's internal state during profiling. The memory view consists of three components as shown in the following.



Memory View

Device: GPU0

Runs: aml_16_w_mem

Views: Memory

Workers: fit-0-optimizer_step_with_closure_0

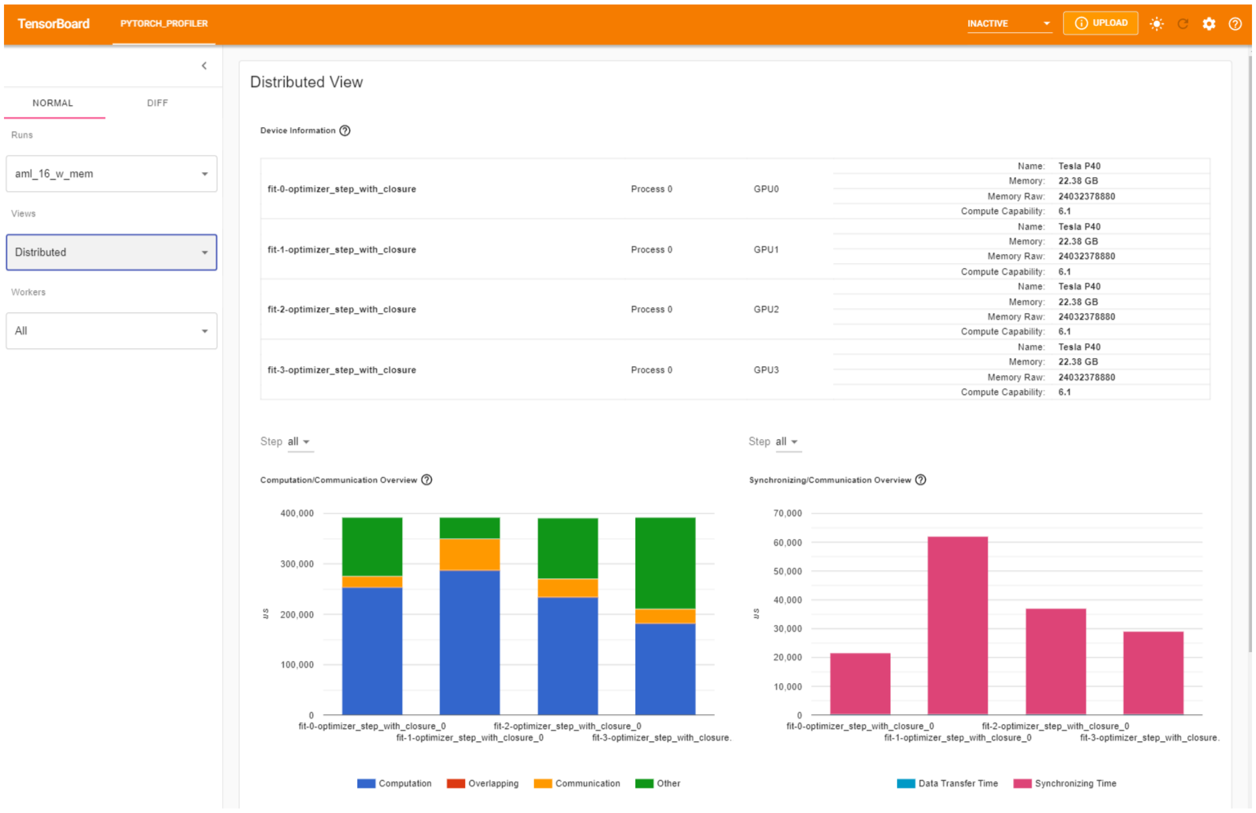
Min Size(KB): 34496

Max Size(KB): 137984

Operator	Size (KB)	Allocation Time (ms)	Release Time (ms)	Duration (ms)
aten::cudnn_convolution	137984	9.76	11.14	1.38
aten::max_pool2d_with_indices	68992	11.35	11.38	0.04
aten::empty_like (aten::empty_strided)	137984	11.04	11.39	0.36
aten::cudnn_convolution	34496	11.48	12.64	1.15
aten::empty_like (aten::empty_strided)	34496	12.54	12.91	0.37
aten::cudnn_convolution	34496	12.74	13.77	1.03
aten::cudnn_convolution	34496	13.93	14.92	1
aten::empty_like (aten::empty_strided)	34496	14.84	15.18	0.34
aten::cudnn_convolution	34496	15.02	15.96	0.94
aten::empty_like (aten::empty_strided)	34496	13.67	16.06	2.39

< 1 2 3 4 5 ··· 22 > 10 / page

Finally, the plugin also supports distributed view on profiling DDP with NCCL/GLOO as backend.


TensorBoard
PYTORCH_PROFILER
INACTIVE
UPLOAD
NORMAL
DIFF
Runs
Views
Workers
Computation/Communication Overview
Synchronizing/Communication Overview
us
fit-0-optimizer_step_with_closure_0
fit-1-optimizer_step_with_closure_0
fit-2-optimizer_step_with_closure_0
fit-3-optimizer_step_with_closure
Computation
Overlapping
Communication
Other
Data Transfer Time
Synchronizing Time

Distributed View

Runs: aml_16_w_mem

Views: Distributed

Workers: All

Device Information

fit-0-optimizer_step_with_closure	Process 0	GPU0	Name:	Tesla P40
			Memory:	22.38 GB
			Memory Raw:	24032378880
			Compute Capability:	6.1
fit-1-optimizer_step_with_closure	Process 0	GPU1	Name:	Tesla P40
			Memory:	22.38 GB
			Memory Raw:	24032378880
			Compute Capability:	6.1
fit-2-optimizer_step_with_closure	Process 0	GPU2	Name:	Tesla P40
			Memory:	22.38 GB
			Memory Raw:	24032378880
			Compute Capability:	6.1
fit-3-optimizer_step_with_closure	Process 0	GPU3	Name:	Tesla P40
			Memory:	22.38 GB
			Memory Raw:	24032378880
			Compute Capability:	6.1

Step: all

Step: all

32.3 Learn More

- PyTorch Profiler with Tensorboard
- PyTorch TensorBoard Profiler github
- Torch profiler API

CHAPTER

THIRTYTHREE

LOADING IMAGES AS TORCH TENSORS

There are many libraries available that can load png images. Simple examples were made using most of them and time of execution was compared. The goal was to load a png file, either RGB or greyscale, into a torch.Tensor.

The tensor specification was:

- shape [3, Height, Width] (for RGB images, in RGB order) or [1, Height, Width] (for greyscale images);
- dtype float32;
- scaled to between 0.0 and 1.0.

33.1 matplotlib

Two methods using matplotlib were compared. The first manipulates the numpy array from the image read before creating the torch tensor, the second uses torchvision to do the transformation.

```python
from pathlib import Path

import matplotlib.image as mpimg
import numpy as np
import torch
import torchvision.transforms.functional as TF

def read_image_matplotlib(input_filename: Path) -> torch.Tensor:
    """
    Read an image file with matplotlib and return a torch.Tensor.

    :param input_filename: Source image file path.
    :return: torch.Tensor of shape (C, H, W).
    """
    # https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.imread.html
    # numpy_array is a numpy.array of shape: (H, W), (H, W, 3), or (H, W, 4)
    # where H = height, W = width
    numpy_array = mpimg.imread(input_filename)
    if len(numpy_array.shape) == 2:
        # if loaded a greyscale image, then it is of shape (H, W) so add in an extra axis
        numpy_array = np.expand_dims(numpy_array, 2)
    # transpose to shape (C, H, W)
    numpy_array = np.transpose(numpy_array, (2, 0, 1))
    torch_tensor = torch.from_numpy(numpy_array)
    return torch_tensor
```

(continues on next page)

(continued from previous page)

```python
def read_image_matplotlib2(input_filename: Path) -> torch.Tensor:
    """
    Read an image file with matplotlib and return a torch.Tensor.

    :param input_filename: Source image file path.
    :return: torch.Tensor of shape (C, H, W).
    """
    # https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.imread.html
    # numpy_array is a numpy.array of shape: (H, W), (H, W, 3), or (H, W, 4)
    # where H = height, W = width
    numpy_array = mpimg.imread(input_filename)
    torch_tensor = TF.to_tensor(numpy_array)
    return torch_tensor
```

33.2 OpenCV

Two methods using the Python interface to OpenCV were compared. The first manipulates the numpy array from the image read before creating the torch tensor, the second uses torchvision to do the transformation. Note that OpenCV loads images in BGR format, so they need to be transformed.

```python
from pathlib import Path

import cv2
import numpy as np
import torch
import torchvision.transforms.functional as TF

def read_image_opencv(input_filename: Path) -> torch.Tensor:
    """
    Read an image file with OpenCV and return a torch.Tensor.

    :param input_filename: Source image file path.
    :return: torch.Tensor of shape (C, H, W).
    """
    # https://docs.opencv.org/4.5.3/d4/da8/group__imgcodecs.html
↪#ga288b8b3da0892bd651fce07b3bbd3a56
    # numpy_array is a numpy.ndarray, in BGR format.
    numpy_array = cv2.imread(str(input_filename))
    numpy_array = cv2.cvtColor(numpy_array, cv2.COLOR_BGR2RGB)
    is_greyscale = False not in \
        ((numpy_array[:, :, 0] == numpy_array[:, :, 1]) == (numpy_array[:, :, 1] ==␣
↪numpy_array[:, :, 2]))
    if is_greyscale:
        numpy_array = numpy_array[:, :, 0]
    if len(numpy_array.shape) == 2:
        # if loaded a greyscale image, then it is of shape (H, W) so add in an extra axis
        numpy_array = np.expand_dims(numpy_array, 2)
    numpy_array = np.float32(numpy_array) / 255.0
```

(continues on next page)

(continued from previous page)

```
    # transpose to shape (C, H, W)
    numpy_array = np.transpose(numpy_array, (2, 0, 1))
    torch_tensor = torch.from_numpy(numpy_array)
    return torch_tensor


def read_image_opencv2(input_filename: Path) -> torch.Tensor:
    """
    Read an image file with OpenCV and return a torch.Tensor.

    :param input_filename: Source image file path.
    :return: torch.Tensor of shape (C, H, W).
    """
    # https://docs.opencv.org/4.5.3/d4/da8/group__imgcodecs.html
↪#ga288b8b3da0892bd651fce07b3bbd3a56
    # numpy_array is a numpy.ndarray, in BGR format.
    numpy_array = cv2.imread(str(input_filename))
    numpy_array = cv2.cvtColor(numpy_array, cv2.COLOR_BGR2RGB)
    is_greyscale = False not in \
        ((numpy_array[:, :, 0] == numpy_array[:, :, 1]) == (numpy_array[:, :, 1] ==␣
↪numpy_array[:, :, 2]))
    if is_greyscale:
        numpy_array = numpy_array[:, :, 0]
    torch_tensor = TF.to_tensor(numpy_array)
    return torch_tensor
```

33.3 Pillow

Pillow is one of the easiest libraries to use because torchvision has a function to convert directly from Pillow images.

```
from pathlib import Path

from PIL import Image
import torch
import torchvision.transforms.functional as TF

def read_image_pillow(input_filename: Path) -> torch.Tensor:
    """
    Read an image file with pillow and return a torch.Tensor.

    :param input_filename: Source image file path.
    :return: torch.Tensor of shape (C, H, W).
    """
    pil_image = Image.open(input_filename)
    torch_tensor = TF.to_tensor(pil_image)
    return torch_tensor
```

33.4 SciPy

SciPy is also easy to use because it loads images into a numpy array of the expected shape so that it can easily be transformed into a torch tensor.

```python
from pathlib import Path

import imageio
import torch
import torchvision.transforms.functional as TF

def read_image_scipy(input_filename: Path) -> torch.Tensor:
    """
    Read an image file with scipy and return a torch.Tensor.

    :param input_filename: Source image file path.
    :return: torch.Tensor of shape (C, H, W).
    """
    numpy_array = imageio.imread(input_filename)
    torch_tensor = TF.to_tensor(numpy_array)
    return torch_tensor
```

33.5 SimpleITK

SimpleITK requires a two step process to load an image and extract the data as a numpy array, but it is then in the correct format.

```python
from pathlib import Path

import SimpleITK as sitk
import torch
import torchvision.transforms.functional as TF

def read_image_sitk(input_filename: Path) -> torch.Tensor:
    """
    Read an image file with SimpleITK and return a torch.Tensor.

    :param input_filename: Source image file path.
    :return: torch.Tensor of shape (C, H, W).
    """
    itk_image = sitk.ReadImage(str(input_filename))
    numpy_array = sitk.GetArrayFromImage(itk_image)
    torch_tensor = TF.to_tensor(numpy_array)
    return torch_tensor
```

33.6 scikit-image

scikit-image is also very simple to use, since it loads the image as a numpy array in the correct format.

```
from pathlib import Path

from skimage import io
import torch
import torchvision.transforms.functional as TF

def read_image_skimage(input_filename: Path) -> torch.Tensor:
    """
    Read an image file with scikit-image and return a torch.Tensor.

    :param input_filename: Source image file path.
    :return: torch.Tensor of shape (C, H, W).
    """
    numpy_array = io.imread(input_filename)
    torch_tensor = TF.to_tensor(numpy_array)
    return torch_tensor
```

33.7 numpy

For comparison, the png image data was saved in the numpy native data format and then reloaded.

```
from pathlib import Path

import numpy as np
import torch
import torchvision.transforms.functional as TF

def read_image_numpy(input_filename: Path) -> torch.Tensor:
    """
    Read an Numpy file with Torch and return a torch.Tensor.

    :param input_filename: Source image file path.
    :return: torch.Tensor of shape (C, H, W).
    """
    numpy_array = np.load(input_filename)
    torch_tensor = torch.from_numpy(numpy_array)
    return torch_tensor
```

33.8 torch

Again, for comparison, the png image data was saved in the torch tensor native data format and then reloaded.

```python
from pathlib import Path

import torch

def read_image_torch2(input_filename: Path) -> torch.Tensor:
    """
    Read a Torch file with Torch and return a torch.Tensor.

    :param input_filename: Source image file path.
    :return: torch.Tensor of shape (C, H, W).
    """
    torch_tensor = torch.load(input_filename)
    return torch_tensor
```

33.9 Results

All the above methods were ran against 122 small test images, repeated 10 times. So in total there were 1220 calls to each of the functions.

33.9.1 RGB Images

For 61 RGB images of size 224 x 224 pixels and 61 of size 180 x 224 pixels, repeated 10 times, there are the following timings:

Function	Total time (s)
read_image_matplotlib	**9.81336**
read_image_matplotlib2	9.96016
read_image_opencv	12.4301
read_image_opencv2	12.6227
read_image_pillow	16.2288
read_image_scipy	17.9958
read_image_sitk	63.6669
read_image_skimage	18.273
read_image_numpy	7.29741
read_image_torch2	**7.07304**

33.9.2 Greyscale Images

Similarly, with greyscale versions of the RGB images:

Function	Total time (s)
read_image_matplotlib	8.32523
read_image_matplotlib2	**8.26399**
read_image_opencv	11.6838
read_image_opencv2	11.7935
read_image_pillow	15.7406
read_image_scipy	17.9061
read_image_sitk	71.8732
read_image_skimage	18.0698
read_image_numpy	7.94197
read_image_torch2	**7.73153**

The recommendation therefore is to use matplotlib `mpimg.imread` to load the image and `TF.to_tensor` to transform the numpy array to a torch tensor. This is almost as fast as loading the data directly in a native numpy or torch format.

33.10 Loading Images as Numpy Arrays

Alternatively, a numpy array may be required with an equivalent form to PIL:

- shape [Height, Width, 3] (for RGB images), in RGB order or [Height, Width] (for greyscale images);
- dtype float;
- range between 0.0 and 255.0.

33.10.1 Pillow

If the image is known to be a png then a shortcut can be taken, which is quicker:

```python
from pathlib import Path

import numpy as np
from PIL import PngImagePlugin
from PIL import Image


def read_image_pillow2(input_filename: Path) -> np.array:  # type: ignore
    """
    Read an image file with pillow and return a numpy array.

    :param input_filename: Source image file path.
    :return: numpy array of shape (H, W), (H, W, 3).
    """
    with Image.open(input_filename) as pil_png:
        return np.asarray(pil_png, np.float)
```

(continues on next page)

(continued from previous page)

```python
def read_image_pillow3(input_filename: Path) -> np.array:  # type: ignore
    """
    Read an image file with pillow and return a numpy array.

    :param input_filename: Source image file path.
    :return: numpy array of shape (H, W), (H, W, 3).
    """
    with PngImagePlugin.PngImageFile(input_filename) as pil_png:
        return np.asarray(pil_png, np.float)
```

33.10.2 SciPy

Similarly, using SciPy:

```python
from pathlib import Path

import imageio
import numpy as np


def read_image_scipy2(input_filename: Path) -> np.array:  # type: ignore
    """
    Read an image file with scipy and return a numpy array.

    :param input_filename: Source image file path.
    :return: numpy array of shape (H, W), (H, W, 3).
    """
    numpy_array = imageio.imread(input_filename).astype(np.float)
    return numpy_array
```

33.11 Results

The three above methods were tested against the same images as above.

33.11.1 RGB Images

For 61 RGB images of size 224 x 224 pixels and 61 of size 180 x 224 pixels, repeated 10 times, there are the following timings:

Function	Total time (s)
read_image_pillow2	44.8641
read_image_pillow3	18.1665
read_image_scipy2	51.8801

33.11.2 Greyscale Images

Similarly, with greyscale versions of the RGB images:

Function	Total time (s)
read_image_pillow2	38.3468
read_image_pillow3	14.664
read_image_scipy2	39.6123

CHAPTER

THIRTYFOUR

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.

34.1 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
```

34.2 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.

34.3 Performance

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

34.3.1 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.

34.3.2 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

THIRTYFIVE

CHANGELOG

Early versions of this toolbox used a manually created changelog. As of March 2022, we have switched to using Github's auto-generated changelog. If you would like to view the changelog for a particular release, you can do so on the Releases page: Each release contains a link for "Full Changelog"

35.1 Changelog for Versions before March 2022

35.2 0.1.14

35.2.1 Added

- (#227) Add TransformerPooling.
- (#179) Add GaussianBlur and RotationByMultiplesOf90 augmentations. Added torchvision and opencv to the environment file since it is necessary for the augmentations.
- (#193) Add transformation adaptor to hi-ml-histopathology.
- (#178) Add runner script for running ML experiments.
- (#181) Add computational pathology tools in hi-ml-histopathology folder.
- (#187) Add mean pooling layer for MIL.
- (#186) Add inference to hi-ml runner.
- (#198) Add cross-validation to hi-ml runner.
- (#198) Improved editor setup for VSCode.

35.2.2 Changed

- (#227) Pooling constructor is outside of DeepMIL and inside of BaseMIL now.
- (#198) Model config loader is now more flexible, can accept fully qualified class name or just top-level module name and class (like histopathology.DeepSMILECrck)
- (#198) Runner raises an error when Conda environment file contains a pip include (-r) statement
- (#196) Show current workspace name in error message.

35.2.3 Fixed

- ([#267]https://github.com/microsoft/hi-ml/pull/267)) Correct PYTHONPATH for Windows in VS Code settings
- ([#266]https://github.com/microsoft/hi-ml/pull/266)) Pin jinja2 package to avoid 'No attribute Markup' bug in version 3.1.0
- (#246) Added tolerance to `test_attentionlayers.py`.
- (#198) Dependencies for histopathology folder are no longer specified in `test_requirements.txt`, but correctly in the histopathology Conda environment.
- (#188) Updated DeepSMILES models. Now they are uptodate with innereye-dl.
- (#179) HEDJitter was jittering the D channel as well. StainNormalization was relying on skimage.
- (#195) Fix DeepMIL metrics bug whereby hard labels were used instead of probabilities.

35.2.4 Removed

35.2.5 Deprecated

35.3 0.1.13

35.3.1 Added

- (#170) Add utils including bag sampling, bounding boxes, HEDJitter, StainNormalisation and add attention layers

35.3.2 Changed

- (#173) Improve report tool: allow lists of tables, option for zipping report folder, option for base64 encoding images

35.3.3 Fixed

- (#169) Fix a test that was failing occasionally

35.3.4 Removed

35.3.5 Deprecated

35.4 0.1.12

35.4.1 Added

- (#159) Add profiling for loading png image files as numpy arrays.
- (#152) Add a custom HTML reporting tool
- (#167) Ability to log to an AzureML run when outside of AzureML

35.4.2 Changed

- (164) Look in more locations for std out from AzureML run.
- (#167) The AzureMLLogger has one mandatory argument now, that controls whether it should log to AzureML also when running on a VM.

35.4.3 Fixed

- (#161) Empty string as target folder for a dataset creates an invalid mounting path for the dataset in AzureML (fixes #160)
- (#167) Fix bugs in logging hyperparameters: logging as name/value table, rather than one column per hyperparameter. Use string logging for all hyperparameters
- (#174) Fix bugs in returned local_checkpoint_path when downloading checkpoints from AML run

35.4.4 Removed

35.4.5 Deprecated

35.5 0.1.11

35.5.1 Added

- (#145) Add ability to mount datasets when running locally.
- (#149) Add a k-fold cross validation wrapper around HyperDrive
- (#132) Profile methods for loading png image files.

35.5.2 Changed

35.5.3 Fixed

- (#156 AzureML Runs should use registered environment after retrieval)

35.5.4 Removed

35.5.5 Deprecated

35.6 0.1.10

35.6.1 Added

- (#142) Adding AzureML progress bar and diagnostics for batch loading
- (#138) Guidelines and profiling for whole slide images.

35.6.2 Changed

- ([#129])https://github.com/microsoft/hi-ml/pull/129)) Refactor command line tools' arguments. Refactor health_azure.utils' various get_run functions. Replace argparsing with parametrized classes.

35.6.3 Fixed

35.6.4 Removed

35.6.5 Deprecated

35.7 0.1.9 (2021-10-20)

35.7.1 Added

- (#133) PyTorch Lightning logger for AzureML. Helper functions for consistent logging
- (#136) Documentation for using low priority nodes

35.7.2 Changed

- (#133) Made ***large breaking changes*** to module names, from `health.azure` to `health_azure`.
- ([#141])(https://github.com/microsoft/hi-ml/pull/141)) Update changelog for release and increase scope of test_register_environment to ensure that by default environments are registered with a version number

35.7.3 Fixed

- (#134) Fixed repo references and added pyright to enforce global checking
- (#139 Fix register_environment, which was ignoring existing environemnts previously. Also ensure that the environment is given version 1 by default instead of "autosave")

35.8 0.1.8 (2021-10-06)

35.8.1 Added

- (#123) Add helper function to download checkpoint files
- (#128) When downloading files in a distributed PyTorch job, a barrier is used to synchronize the processes.

35.8.2 Changed

- (#127) The field `is_running_in_azure` of `AzureRunInfo` has been renamed to `is_running_in_azure_ml`

35.8.3 Fixed

- (#127) Fixing bug #126: get_workspace was assuming it runs in AzureML, when it was running on a plain Azure build agent.

35.9 0.1.7 (2021-10-04)

35.9.1 Added

- (#111) Adding changelog. Displaying changelog in sphinx docu. Ensure changelog is updated.

35.9.2 Changed

- (#112) Update himl_tensorboard to work with files not in 'logs' directory
- (#106) Split into two packages. Most of existing package renamed to hi-ml-azure, remained remains hi-ml.
- (#113) Add helper function to download files from AML Run, tidied up some command line args, and moved some functions from himl.py to azure_util.py
- (#122) Add helper functions to upload to and download from AML Datastores

35.9.3 Fixed

- (#117) Bug fix: Config.json file was expected to be present, even if workspace was provided explicitly.
- (#119) Bug fix: Code coverage wasn't formatted correctly.

35.10 0.1.4 (2021-09-15)

- This is the baseline release.

CHAPTER

THIRTYSIX

HEALTH_AZURE PACKAGE

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

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.

input_datasets: List[Optional[pathlib.Path]]
A list of folders that contain all the datasets that the script uses as inputs. Input datasets must be specified when calling *submit_to_azure_if_needed*. Here, they are made available as Path objects. If no input datasets are specified, the list is empty.

is_running_in_azure_ml: bool
If True, the present script is executing inside AzureML. If False, outside AzureML.

logs_folder: Optional[pathlib.Path]
The folder into which all log files (for example, tensorboard) should be written. All files written to this folder will be uploaded to blob storage regularly during the script run.

mount_contexts: List[azureml.dataprep.fuse.daemon.MountContext]
A list of mount contexts for input datasets when running outside AzureML. There will be a mount context for each input dataset where there is no local_folder, there is a workspace, and use_mounting is set. This list is maintained only to prevent exit from these contexts until the RunInfo object is deleted.

output_datasets: List[Optional[pathlib.Path]]
A list of folders that contain all the datasets that the script uses as outputs. Output datasets must be specified when calling *submit_to_azure_if_needed*. Here, they are made available as Path objects. If no output datasets are specified, the list is empty.

output_folder: Optional[pathlib.Path]
The output folder into which all script outputs should be written, if they should be later available in the AzureML portal. Files written to this folder will be uploaded to blob storage at the end of the script run.

run: Optional[azureml.core.run.Run]
An AzureML Run object if the present script is executing inside AzureML, or None if outside of AzureML. The Run object has methods to log metrics, upload files, etc.

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

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.
- **overwrite_existing** (bool) – Only applies to uploading datasets. If True, the dataset will be overwritten if it already exists. If False, the dataset creation will fail if the dataset already exists.
- **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** (Union[Path, str, None]) – The folder into which the dataset should be downloaded or mounted. If left empty, a random folder on /tmp will be chosen. Do NOT use "." as the target_folder.
- **local_folder** (Union[Path, str, None]) – The folder on the local machine at which the dataset is available. This is used only for runs outside of AzureML. If this is empty then the target_folder will be used to mount or download the dataset.

to_input_dataset(*dataset_index*, *workspace*, *strictly_aml_v1*, *ml_client=None*)

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}
- **strictly_aml_v1** (bool) – If True, use Azure ML SDK v1. Otherwise, attempt to use Azure ML SDK v2.
- **ml_client** (Optional[MLClient]) – An Azure MLClient object for interacting with Azure resources.

Return type Optional[DatasetConsumptionConfig]

to_input_dataset_local(*workspace*)

Return a local path to the dataset when outside of an AzureML run. If local_folder is supplied, then this is assumed to be a local dataset, and this is returned. Otherwise the dataset is mounted or downloaded to either the target folder or a temporary folder and that is returned. If self.name refers to a v2 dataset, it is not possible to mount the data here, therefore a tuple of Nones will be returned.

Parameters **workspace** (Workspace) – The AzureML workspace to read from.

Return type Tuple[Path, Optional[MountContext]]

Returns Tuple of (path to dataset, optional mountcontext)

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 An AzureML OutputFileDatasetConfig object, representing the output dataset.

health_azure.**aggregate_hyperdrive_metrics**(*child_run_arg_name*, *run_id=None*, *run=None*, *keep_metrics=None*, *aml_workspace=None*, *workspace_config_path=None*)

For a given HyperDriveRun object, or id of a HyperDriveRun, retrieves the metrics from each of its children and then aggregates it. Optionally filters the metrics logged in the Run, by providing a list of metrics to keep. Returns a DataFrame where each column is one child run, and each row is a metric logged by that child run. For example, for a HyperDrive run with 2 children, where each logs epoch, accuracy and loss, the result would look like:

```
|              | 0               | 1                  |
|--------------|-----------------|--------------------|
| epoch        | [1, 2, 3]       | [1, 2, 3]          |
| accuracy     | [0.7, 0.8, 0.9] | [0.71, 0.82, 0.91] |
| loss         | [0.5, 0.4, 0.3] | [0.45, 0.37, 0.29] |
```

here each column is one of the splits/ child runs, and each row is one of the metrics you have logged to the run.

It is possible to log rows and tables in Azure ML by calling run.log_table and run.log_row respectively. In this case, the DataFrame will contain a Dictionary entry instead of a list, where the keys are the table columns (or keywords provided to log_row), and the values are the table values. E.g.:

```
|                | 0                                        | 1                       ␣
↪                   |
|----------------|-----------------------------------------|-----------------------
↪--------------------|
| accuracy_table |{'epoch': [1, 2], 'accuracy': [0.7, 0.8]} | {'epoch': [1, 2],
↪'accuracy': [0.8, 0.9]} |
```

It is also possible to log plots in Azure ML by calling run.log_image and passing in a matplotlib plot. In this case, the DataFrame will contain a string representing the path to the artifact that is generated by AML (the saved plot in the Logs & Outputs pane of your run on the AML portal). E.g.:

```
|                | 0                                       | 1                        ␣
↪              |
|----------------|----------------------------------------|------------------------
↪---------------|
| accuracy_plot  | aml://artifactId/ExperimentRun/dcid.... | aml://artifactId/
↪ExperimentRun/dcid...|
```

Parameters

- **child_run_arg_name** (str) – the name of the argument given to each child run to denote its position relative to other child runs (e.g. this arg could equal 'child_run_index' - then each of your child runs should expect to receive the arg '–child_run_index' with a value <= the total number of child runs)
- **run** (Optional[Run]) – An Azure ML HyperDriveRun object to aggregate the metrics from. Either this or run_id must be provided

- **run_id** (Optional[str]) – The id (type: str) of a parent/ HyperDrive run. Either this or run must be provided.
- **keep_metrics** (Optional[List[str]]) – An optional list of metric names to filter the returned metrics by
- **aml_workspace** (Optional[Workspace]) – If run_id is provided, this is an optional AML Workspace object to retrieve the Run from
- **workspace_config_path** (Optional[Path]) – If run_id is provided, this is an optional path to a config containing details of the AML Workspace object to retrieve the Run from.

Return type DataFrame

Returns A Pandas DataFrame containing the aggregated metrics from each child run

health_azure.**create_aml_run_object**(*experiment_name*, *run_name=None*, *workspace=None*, *workspace_config_path=None*, *snapshot_directory=None*)

Creates an AzureML Run object in the given workspace, or in the workspace given by the AzureML config file. This Run object can be used to write metrics to AzureML, upload files, etc, when the code is not running in AzureML. After finishing all operations, use *run.flush()* to write metrics to the cloud, and *run.complete()* or *run.fail()*.

Example: >>>run = create_aml_run_object(experiment_name="run_on_my_vm", run_name="try1") >>>run.log("foo", 1.23) >>>run.flush() >>>run.complete()

Parameters

- **experiment_name** (str) – The AzureML experiment that should hold the run that will be created.
- **run_name** (Optional[str]) – An optional name for the run (this will be used as the display name in the AzureML UI)
- **workspace** (Optional[Workspace]) – If provided, use this workspace to create the run in. If not provided, use the workspace specified by the *config.json* file in the folder or its parent folder(s).
- **workspace_config_path** (Optional[Path]) – If not provided with an AzureML workspace, then load one given the information in this config file.
- **snapshot_directory** (Union[Path, str, None]) – The folder that should be included as the code snapshot. By default, no snapshot is created (snapshot_directory=None or snapshot_directory=""). Set this to the folder that contains all the code your experiment uses. You can use a file .amlignore to skip specific files or folders, akin to .gitignore

Return type Run

Returns An AzureML Run object.

health_azure.**create_crossval_hyperdrive_config**(*num_splits*, *cross_val_index_arg_name='crossval_index'*, *metric_name='val/loss'*)

Creates an Azure ML HyperDriveConfig object for running cross validation. Note: this config expects a metric named <metric_name> to be logged in your training script([see here](https://docs.microsoft.com/en-us/azure/machine-learning/how-to-tune-hyperparameters#log-metrics-for-hyperparameter-tuning))

Parameters

- **num_splits** (int) – The number of splits for k-fold cross validation
- **cross_val_index_arg_name** (str) – The name of the commandline argument that each of the child runs gets, to indicate which split they should work on.

- **metric_name** (str) – The name of the metric that the HyperDriveConfig will compare runs by. Please note that it is your responsibility to make sure a metric with this name is logged to the Run in your training script

Return type HyperDriveConfig

Returns an Azure ML HyperDriveConfig object

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

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

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

Parameters

- **`script_params`** (`List[str]`) – A list of parameter to pass on to the script as it runs in AzureML. Required arg. Script parameters can be generated using the `_get_script_params()` function.
- **`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.
- **`entry_script`** (`Union[Path, str, None]`) – The script that should be run in AzureML. If None, the current main Python file will be executed.

Return type `ScriptRunConfig`

Returns

`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`

`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`

`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`

`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.

`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

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

Retrieve an Azure ML Workspace by going through the following steps:

1. If the function has been called from inside a run in AzureML, it returns the current AzureML workspace.
2. If a Workspace object has been provided in the *aml_workspace* argument, return that.
3. If a path to a Workspace config file has been provided, load the workspace according to that config file.
4. If a Workspace config file is present in the current working directory or one of its parents, load the

   workspace according to that config file.
5. If 3 environment variables are found, use them to identify the workspace (*HIML_RESOURCE_GROUP*,

   HIML_SUBSCRIPTION_ID, *HIML_WORKSPACE_NAME*)

If none of the above succeeds, an exception is raised.

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.

Raises

- **`ValueError`** – If none of the available options for accessing the workspace succeeds.
- **`FileNotFoundError`** – If the workspace config file is given in *workspace_config_path*, but is not present.

`health_azure.`**`health_azure_package_setup`**`()`

Set up the Python packages where needed. In particular, reduce the logging level for some of the used libraries, which are particularly talkative in DEBUG mode. Usually when running in DEBUG mode, we want diagnostics about the model building itself, but not for the underlying libraries.

Return type `None`

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.

health_azure.**object_to_yaml**(*o*)

Converts an object to a YAML string representation. This is done by recursively traversing all attributes and writing them out to YAML if they are basic datatypes.

Parameters **o** (Any) – The object to inspect.

Return type str

Returns A string in YAML format.

health_azure.**set_environment_variables_for_multi_node**()

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

Return type None

health_azure.**set_logging_levels**(*levels*)

Sets the logging levels for the given module-level loggers.

Parameters **levels** (Dict[str, int]) – A mapping from module name to desired logging level.

Return type None

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

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

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.
- **display_name** (`Optional[str]`) – The name for the run that will be displayed in the AML UI. If not provided, a random display name will be generated by AzureML.

Return type `Run`

Returns An AzureML Run object.

`health_azure.`**`submit_to_azure_if_needed`**(*compute_cluster_name=''*, *entry_script=None*, *aml_workspace=None*, *workspace_config_file=None*, *ml_client=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='mcr.microsoft.com/azureml/openmpi4.1.0-cuda11.3-cudnn8-ubuntu20.04:20230509.v1'*, *docker_shm_size='100g'*, *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*, *hyperparam_args=None*, *strictly_aml_v1=False*, *identity_based_auth=False*, *pytorch_processes_per_node_v2=None*, *use_mpi_run_for_single_node_jobs=True*, *display_name=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** (`Union[Callable[[Run], None], Callable[[Job, MLClient], None], None]`) – A function that will be called directly after submitting the job to AzureML. Use this to, for example, add additional tags or print information about the run. When using AzureML SDK V1, the only argument to this function is the Run object that was just submitted. When using AzureML SDK V2, the arguments are (Job, MLClient).
- **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.
- **ml_client** (`Optional[MLClient]`) – An Azure MLClient object for interacting with Azure resources.
- **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 parameters to pass on to the script as it runs in AzureML. If *None* (the default), these will be copied over from *sys.argv* (excluding the *–azureml* flag, if found).
- **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. The list of available images can be found here: https://github.com/Azure/AzureML-Containers The default image is *mcr.microsoft.com/azureml/openmpi3.1.2-cuda10.2-cudnn8-ubuntu18.04*
- **docker_shm_size** (`str`) – The Docker shared memory size that should be used when creating a new Docker image. Default value is '100g'.
- **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. When using a value > 1, multiple nodes in AzureML will be started. If *pytorch_processes_per_node_v2=None*, the job will be submitted as a multi-node MPI job, with 1 process per node. This is suitable for PyTorch Lightning jobs. If *pytorch_processes_per_node_v2* is not None, a job with framework "PyTorch" and communication backend "nccl" will be started. *pytorch_processes_per_node_v2* will guide the num-

ber of processes per node. This is suitable for plain PyTorch training jobs without the use of frameworks like PyTorch Lightning.

- **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).
- **strictly_aml_v1** (bool) – If True, use Azure ML SDK v1. Otherwise, attempt to use Azure ML SDK v2.
- **pytorch_processes_per_node_v2** (Optional[int]) – For plain PyTorch multi-GPU processing: The number of processes per node. This is only supported with AML SDK v2, and ignored in v1. If supplied, the job will be submitted as using the "pytorch" framework (rather than "Python"), and using "nccl" as the communication backend.
- **use_mpi_run_for_single_node_jobs** (bool) – If True, even single node jobs with SDK v2 will be run as distributed MPI jobs. This is required for Kubernetes compute. If False, single node jobs will not be run as distributed jobs. This setting only affects jobs submitted with SDK v2 (when *strictly_aml_v1=False*)
- **display_name** (Optional[str]) – The name for the run that will be displayed in the AML UI. If not provided, a random display name will be generated by AzureML.

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.

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

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

health_azure.**write_yaml_to_object**(*o*, *yaml_string*, *strict=False*)

Writes a serialized object in YAML format back into an object, assuming that the attributes of the object and the YAML field names are in sync.

Parameters strict (bool) – If True, any mismatch of field names will raise a ValueError. If False, only a warning will be

printed. Note that the object may have been modified even if an error is raised. :type o: `Any` :param o: The object to write to. :type yaml_string: `str` :param yaml_string: A YAML formatted string with attribute names and values.

Return type None

36.1 Functions

create_aml_run_object(experiment_name[, …])	Creates an AzureML Run object in the given workspace, or in the workspace given by the AzureML config file.
create_run_configuration(workspace, …[, …])	Creates an AzureML run configuration, that contains information about environment, multi node execution, and Docker.
create_script_run(script_params[, …])	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 by going through the following steps:
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.

continues on next page

Table 1 – continued from previous page

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.
create_crossval_hyperdrive_config(num_splits)	Creates an Azure ML HyperDriveConfig object for running cross validation.
aggregate_hyperdrive_metrics(child_run_arg_name)	For a given HyperDriveRun object, or id of a HyperDriveRun, retrieves the metrics from each of its children and then aggregates it. Optionally filters the metrics logged in the Run, by providing a list of metrics to keep. Returns a DataFrame where each column is one child run, and each row is a metric logged by that child run. For example, for a HyperDrive run with 2 children, where each logs epoch, accuracy and loss, the result would look like::.
object_to_yaml(o)	Converts an object to a YAML string representation.
write_yaml_to_object(o, yaml_string[, strict])	Writes a serialized object in YAML format back into an object, assuming that the attributes of the object and the YAML field names are in sync.
health_azure_package_setup()	Set up the Python packages where needed.
set_logging_levels(levels)	Sets the logging levels for the given module-level loggers.

36.1.1 create_aml_run_object

health_azure.**create_aml_run_object**(*experiment_name*, *run_name=None*, *workspace=None*, *workspace_config_path=None*, *snapshot_directory=None*)

Creates an AzureML Run object in the given workspace, or in the workspace given by the AzureML config file. This Run object can be used to write metrics to AzureML, upload files, etc, when the code is not running in AzureML. After finishing all operations, use *run.flush()* to write metrics to the cloud, and *run.complete()* or *run.fail()*.

Example: >>>run = create_aml_run_object(experiment_name="run_on_my_vm", run_name="try1") >>>run.log("foo", 1.23) >>>run.flush() >>>run.complete()

Parameters

- **experiment_name** (str) – The AzureML experiment that should hold the run that will be created.
- **run_name** (Optional[str]) – An optional name for the run (this will be used as the display name in the AzureML UI)
- **workspace** (Optional[Workspace]) – If provided, use this workspace to create the run in. If not provided, use the workspace specified by the *config.json* file in the folder or its parent folder(s).

- **workspace_config_path** (`Optional[Path]`) – If not provided with an AzureML workspace, then load one given the information in this config file.
- **snapshot_directory** (`Union[Path, str, None]`) – The folder that should be included as the code snapshot. By default, no snapshot is created (snapshot_directory=None or snapshot_directory=""). Set this to the folder that contains all the code your experiment uses. You can use a file .amlignore to skip specific files or folders, akin to .gitignore

Return type `Run`

Returns An AzureML Run object.

36.1.2 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

36.1.3 create_script_run

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

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

Parameters

- **script_params** (`List[str]`) – A list of parameter to pass on to the script as it runs in AzureML. Required arg. Script parameters can be generated using the `_get_script_params()` function.
- **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.
- **entry_script** (`Union[Path, str, None]`) – The script that should be run in AzureML. If None, the current main Python file will be executed.

Return type `ScriptRunConfig`

Returns

36.1.4 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

36.1.5 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

36.1.6 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

36.1.7 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.

36.1.8 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

36.1.9 get_workspace

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

Retrieve an Azure ML Workspace by going through the following steps:

1. If the function has been called from inside a run in AzureML, it returns the current AzureML workspace.
2. If a Workspace object has been provided in the *aml_workspace* argument, return that.
3. If a path to a Workspace config file has been provided, load the workspace according to that config file.
4. If a Workspace config file is present in the current working directory or one of its parents, load the

workspace according to that config file.

5. If 3 environment variables are found, use them to identify the workspace (*HIML_RESOURCE_GROUP*,

   HIML_SUBSCRIPTION_ID, *HIML_WORKSPACE_NAME*)

If none of the above succeeds, an exception is raised.

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.

Raises

- **ValueError** – If none of the available options for accessing the workspace succeeds.
- **FileNotFoundError** – If the workspace config file is given in *workspace_config_path*, but is not present.

36.1.10 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.

36.1.11 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`

36.1.12 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

36.1.13 submit_run

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

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.
- **`display_name`** (`Optional[str]`) – The name for the run that will be displayed in the AML UI. If not provided, a random display name will be generated by AzureML.

Return type `Run`

Returns An AzureML Run object.

36.1.14 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*, *ml_client=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='mcr.microsoft.com/azureml/openmpi4.1.0-cuda11.3-cudnn8-ubuntu20.04:20230509.v1'*, *docker_shm_size='100g'*, *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*, *hyperparam_args=None*, *strictly_aml_v1=False*, *identity_based_auth=False*, *pytorch_processes_per_node_v2=None*, *use_mpi_run_for_single_node_jobs=True*, *display_name=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`** (`Union[Callable[[Run], None], Callable[[Job, MLClient], None], None]`) – A function that will be called directly after submitting the job to AzureML. Use this to, for example, add additional tags or print information about the run. When using AzureML SDK V1, the only argument to this function is the Run object that was just submitted. When using AzureML SDK V2, the arguments are (Job, MLClient).
- **`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.
- **ml_client** (Optional[MLClient]) – An Azure MLClient object for interacting with Azure resources.
- **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 parameters to pass on to the script as it runs in AzureML. If *None* (the default), these will be copied over from *sys.argv* (excluding the *–azureml* flag, if found).
- **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. The list of available images can be found here: https://github.com/Azure/AzureML-Containers The default image is *mcr.microsoft.com/azureml/openmpi3.1.2-cuda10.2-cudnn8-ubuntu18.04*
- **docker_shm_size** (str) – The Docker shared memory size that should be used when creating a new Docker image. Default value is '100g'.
- **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. When using a value > 1, multiple nodes in AzureML will be started. If *pytorch_processes_per_node_v2=None*, the job will be submitted as a multi-node MPI job, with 1 process per node. This is suitable for PyTorch Lightning jobs. If *pytorch_processes_per_node_v2* is not None, a job with framework "PyTorch" and communication backend "nccl" will be started. *pytorch_processes_per_node_v2* will guide the number of processes per node. This is suitable for plain PyTorch training jobs without the use of frameworks like PyTorch Lightning.
- **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).
- **strictly_aml_v1** (bool) – If True, use Azure ML SDK v1. Otherwise, attempt to use Azure ML SDK v2.
- **pytorch_processes_per_node_v2** (Optional[int]) – For plain PyTorch multi-GPU processing: The number of processes per node. This is only supported with AML SDK v2, and ignored in v1. If supplied, the job will be submitted as using the "pytorch" framework (rather than "Python"), and using "nccl" as the communication backend.
- **use_mpi_run_for_single_node_jobs** (bool) – If True, even single node jobs with SDK v2 will be run as distributed MPI jobs. This is required for Kubernetes compute. If False, single node jobs will not be run as distributed jobs. This setting only affects jobs submitted with SDK v2 (when *strictly_aml_v1=False*)
- **display_name** (Optional[str]) – The name for the run that will be displayed in the AML UI. If not provided, a random display name will be generated by AzureML.

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.

36.1.15 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

36.1.16 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`

36.1.17 create_crossval_hyperdrive_config

`health_azure.`**`create_crossval_hyperdrive_config`**`(`*num_splits*, *cross_val_index_arg_name='crossval_index'*, *metric_name='val/loss'*`)`

Creates an Azure ML HyperDriveConfig object for running cross validation. Note: this config expects a metric named <metric_name> to be logged in your training script([see here](https://docs.microsoft.com/en-us/azure/machine-learning/how-to-tune-hyperparameters#log-metrics-for-hyperparameter-tuning))

Parameters

- **num_splits** (`int`) – The number of splits for k-fold cross validation
- **cross_val_index_arg_name** (`str`) – The name of the commandline argument that each of the child runs gets, to indicate which split they should work on.
- **metric_name** (`str`) – The name of the metric that the HyperDriveConfig will compare runs by. Please note that it is your responsibility to make sure a metric with this name is logged to the Run in your training script

Return type `HyperDriveConfig`

Returns an Azure ML HyperDriveConfig object

36.1.18 aggregate_hyperdrive_metrics

`health_azure.`**`aggregate_hyperdrive_metrics`**`(`*child_run_arg_name*, *run_id=None*, *run=None*, *keep_metrics=None*, *aml_workspace=None*, *workspace_config_path=None*`)`

For a given HyperDriveRun object, or id of a HyperDriveRun, retrieves the metrics from each of its children and then aggregates it. Optionally filters the metrics logged in the Run, by providing a list of metrics to keep. Returns a DataFrame where each column is one child run, and each row is a metric logged by that child run. For example, for a HyperDrive run with 2 children, where each logs epoch, accuracy and loss, the result would look like:

```
|              | 0               | 1                  |
|--------------|-----------------|--------------------|
| epoch        | [1, 2, 3]       | [1, 2, 3]          |
| accuracy     | [0.7, 0.8, 0.9] | [0.71, 0.82, 0.91] |
| loss         | [0.5, 0.4, 0.3] | [0.45, 0.37, 0.29] |
```

here each column is one of the splits/ child runs, and each row is one of the metrics you have logged to the run.

It is possible to log rows and tables in Azure ML by calling run.log_table and run.log_row respectively. In this case, the DataFrame will contain a Dictionary entry instead of a list, where the keys are the table columns (or keywords provided to log_row), and the values are the table values. E.g.:

```
|                 | 0                                         | 1                      ␣
↪                    |
|-----------------|-------------------------------------------|-----------------------
↪--------------------|
| accuracy_table |{'epoch': [1, 2], 'accuracy': [0.7, 0.8]} | {'epoch': [1, 2],
↪'accuracy': [0.8, 0.9]} |
```

It is also possible to log plots in Azure ML by calling run.log_image and passing in a matplotlib plot. In this case, the DataFrame will contain a string representing the path to the artifact that is generated by AML (the saved plot in the Logs & Outputs pane of your run on the AML portal). E.g.:

```
|                 | 0                                        | 1                       ␣
↪               |
|-----------------|------------------------------------------|------------------------
↪---------------|
| accuracy_plot   | aml://artifactId/ExperimentRun/dcid.... | aml://artifactId/
↪ExperimentRun/dcid...|
```

Parameters

- **child_run_arg_name** (str) – the name of the argument given to each child run to denote its position relative to other child runs (e.g. this arg could equal 'child_run_index' - then each of your child runs should expect to receive the arg '–child_run_index' with a value <= the total number of child runs)
- **run** (Optional[Run]) – An Azure ML HyperDriveRun object to aggregate the metrics from. Either this or run_id must be provided
- **run_id** (Optional[str]) – The id (type: str) of a parent/ HyperDrive run. Either this or run must be provided.
- **keep_metrics** (Optional[List[str]]) – An optional list of metric names to filter the returned metrics by
- **aml_workspace** (Optional[Workspace]) – If run_id is provided, this is an optional AML Workspace object to retrieve the Run from
- **workspace_config_path** (Optional[Path]) – If run_id is provided, this is an optional path to a config containing details of the AML Workspace object to retrieve the Run from.

Return type DataFrame

Returns A Pandas DataFrame containing the aggregated metrics from each child run

36.1.19 object_to_yaml

health_azure.**object_to_yaml**(*o*)

Converts an object to a YAML string representation. This is done by recursively traversing all attributes and writing them out to YAML if they are basic datatypes.

Parameters **o** (Any) – The object to inspect.

Return type str

Returns A string in YAML format.

36.1.20 write_yaml_to_object

health_azure.**write_yaml_to_object**(*o*, *yaml_string*, *strict=False*)

Writes a serialized object in YAML format back into an object, assuming that the attributes of the object and the YAML field names are in sync.

Parameters **strict** (bool) – If True, any mismatch of field names will raise a ValueError. If False, only a warning will be

printed. Note that the object may have been modified even if an error is raised. :type o: `Any` :param o: The object to write to. :type yaml_string: `str` :param yaml_string: A YAML formatted string with attribute names and values.

Return type None

36.1.21 health_azure_package_setup

health_azure.**health_azure_package_setup**()

Set up the Python packages where needed. In particular, reduce the logging level for some of the used libraries, which are particularly talkative in DEBUG mode. Usually when running in DEBUG mode, we want diagnostics about the model building itself, but not for the underlying libraries.

Return type None

36.1.22 set_logging_levels

health_azure.**set_logging_levels**(*levels*)

Sets the logging levels for the given module-level loggers.

Parameters **levels** (Dict[str, int]) – A mapping from module name to desired logging level.

Return type None

36.2 Classes

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

36.2.1 AzureRunInfo

class health_azure.**AzureRunInfo**(*input_datasets*, *output_datasets*, *mount_contexts*, *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.

input_datasets: List[Optional[pathlib.Path]]

A list of folders that contain all the datasets that the script uses as inputs. Input datasets must be specified when calling *submit_to_azure_if_needed*. Here, they are made available as Path objects. If no input datasets are specified, the list is empty.

is_running_in_azure_ml: bool
If True, the present script is executing inside AzureML. If False, outside AzureML.

logs_folder: Optional[pathlib.Path]
The folder into which all log files (for example, tensorboard) should be written. All files written to this folder will be uploaded to blob storage regularly during the script run.

mount_contexts: List[azureml.dataprep.fuse.daemon.MountContext]
A list of mount contexts for input datasets when running outside AzureML. There will be a mount context for each input dataset where there is no local_folder, there is a workspace, and use_mounting is set. This list is maintained only to prevent exit from these contexts until the RunInfo object is deleted.

output_datasets: List[Optional[pathlib.Path]]
A list of folders that contain all the datasets that the script uses as outputs. Output datasets must be specified when calling *submit_to_azure_if_needed*. Here, they are made available as Path objects. If no output datasets are specified, the list is empty.

output_folder: Optional[pathlib.Path]
The output folder into which all script outputs should be written, if they should be later available in the AzureML portal. Files written to this folder will be uploaded to blob storage at the end of the script run.

run: Optional[azureml.core.run.Run]
An AzureML Run object if the present script is executing inside AzureML, or None if outside of AzureML. The Run object has methods to log metrics, upload files, etc.

36.2.2 DatasetConfig

class health_azure.**DatasetConfig**(*name*, *datastore=''*, *overwrite_existing=True*, *version=None*, *use_mounting=None*, *target_folder=None*, *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.
- **overwrite_existing** (`bool`) – Only applies to uploading datasets. If True, the dataset will be overwritten if it already exists. If False, the dataset creation will fail if the dataset already exists.
- **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** (Union[Path, str, None]) – The folder into which the dataset should be downloaded or mounted. If left empty, a random folder on /tmp will be chosen. Do NOT use "." as the target_folder.
- **local_folder** (Union[Path, str, None]) – The folder on the local machine at which the dataset is available. This is used only for runs outside of AzureML. If this is empty then the target_folder will be used to mount or download the dataset.

Methods Summary

to_input_dataset(dataset_index, workspace, ...)	Creates a configuration for using an AzureML dataset inside of an AzureML run.
to_input_dataset_local(workspace)	Return a local path to the dataset when outside 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(*dataset_index*, *workspace*, *strictly_aml_v1*, *ml_client=None*)

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}
- **strictly_aml_v1** (bool) – If True, use Azure ML SDK v1. Otherwise, attempt to use Azure ML SDK v2.
- **ml_client** (Optional[MLClient]) – An Azure MLClient object for interacting with Azure resources.

Return type Optional[DatasetConsumptionConfig]

to_input_dataset_local(*workspace*)

Return a local path to the dataset when outside of an AzureML run. If local_folder is supplied, then this is assumed to be a local dataset, and this is returned. Otherwise the dataset is mounted or downloaded to either the target folder or a temporary folder and that is returned. If self.name refers to a v2 dataset, it is not possible to mount the data here, therefore a tuple of Nones will be returned.

Parameters **workspace** (Workspace) – The AzureML workspace to read from.

Return type Tuple[Path, Optional[MountContext]]

Returns Tuple of (path to dataset, optional mountcontext)

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 An AzureML OutputFileDatasetConfig object, representing the output dataset.

`to_input_dataset`(*dataset_index*, *workspace*, *strictly_aml_v1*, *ml_client=None*)

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}
- **`strictly_aml_v1`** (`bool`) – If True, use Azure ML SDK v1. Otherwise, attempt to use Azure ML SDK v2.
- **`ml_client`** (`Optional[MLClient]`) – An Azure MLClient object for interacting with Azure resources.

Return type `Optional[DatasetConsumptionConfig]`

`to_input_dataset_local`(*workspace*)

Return a local path to the dataset when outside of an AzureML run. If local_folder is supplied, then this is assumed to be a local dataset, and this is returned. Otherwise the dataset is mounted or downloaded to either the target folder or a temporary folder and that is returned. If self.name refers to a v2 dataset, it is not possible to mount the data here, therefore a tuple of Nones will be returned.

Parameters **`workspace`** (`Workspace`) – The AzureML workspace to read from.

Return type `Tuple[Path, Optional[MountContext]]`

Returns Tuple of (path to dataset, optional mountcontext)

`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 An AzureML OutputFileDatasetConfig object, representing the output dataset.

CHAPTER

THIRTYSEVEN

HEALTH_ML PACKAGE

class `health_ml.Runner`(*project_root*)

This class contains the high-level logic to start a training run: choose a model configuration by name, submit to AzureML if needed, or otherwise start the actual training and test loop.

Parameters **`project_root`** (`Path`) – The root folder that contains all of the source code that should be executed.

`additional_run_tags`(*script_params*)

Gets the set of tags that will be added to the AzureML run as metadata.

Parameters **`script_params`** (`List[str]`) – The commandline arguments used to invoke the present script.

Return type `Dict[str, str]`

`parse_and_load_model`()

Parses the command line arguments, and creates configuration objects for the model itself, and for the Azure-related parameters. Sets self.experiment_config to its proper values. Returns the parser output from parsing the model commandline arguments.

Return type `ParserResult`

Returns ParserResult object containing args, overrides and settings

`run`()

The main entry point for training and testing models from the commandline. This chooses a model to train via a commandline argument, runs training or testing, and writes all required info to disk and logs.

Return type `Tuple[LightningContainer, AzureRunInfo]`

Returns a tuple of the LightningContainer object and an AzureRunInfo containing all information about the present run (whether running in AzureML or not)

`run_in_situ`(*azure_run_info*)

Actually run the AzureML job; this method will typically run on an Azure VM.

Parameters **`azure_run_info`** (`AzureRunInfo`) – Contains all information about the present run in AzureML, in particular where the

datasets are mounted.

Return type `None`

`submit_to_azureml_if_needed`()

Submit a job to AzureML, returning the resulting Run object, or exiting if we were asked to wait for completion and the Run did not succeed.

Return type `AzureRunInfo`

Returns an AzureRunInfo object containing all of the details of the present run. If AzureML is not specified, the attribute 'run' will None, but the object still contains helpful information about datasets etc

`validate()`
Runs sanity checks on the whole experiment.

Return type `None`

class `health_ml.TrainingRunner`(*experiment_config*, *container*, *project_root=None*)
Driver class to run an ML experiment. Note that the project root argument MUST be supplied when using hi-ml as a package!

Parameters

- `experiment_config` (`ExperimentConfig`) – The ExperimentConfig object to use for training.
- `container` (`LightningContainer`) – The LightningContainer object to use for training.
- `project_root` (`Optional[Path]`) – Project root. This should only be omitted if calling run_ml from the test suite. Supplying it is crucial when using hi-ml as a package or submodule!

`after_ddp_cleanup`(*environ_before_training*)
Run processes cleanup after ddp context to prepare for single device inference. Kill all processes in DDP besides rank 0.

Return type `None`

`end_training`(*environ_before_training*)
Cleanup after training is done. This is called after the trainer has finished fitting the data. This is called to update the checkpoint handler state and remove redundant checkpoint files. If running inference on a single device, this is also called to kill all processes besides rank 0.

Return type `None`

`get_data_module()`
Reads the datamodule that should be used for training or valuation from the container. This must be overridden in subclasses.

Return type `LightningDataModule`

`init_inference()`
Prepare the trainer for running inference on the validation and test set. This chooses a checkpoint, initializes the PL Trainer object, and chooses the right data module. The hook for running inference on the validation set is run (*LightningContainer.on_run_extra_validation_epoch*) is first called to reflect any changes to the model or datamodule states before running inference.

Return type `None`

`init_training()`
Execute some bookkeeping tasks only once if running distributed and initialize the runner's trainer object.

Return type `None`

`is_crossval_disabled_or_child_0()`
Returns True if the present run is a non-cross-validation run, or child run 0 of a cross-validation run.

Return type `bool`

`run()`
Driver function to run a ML experiment

Return type `None`

`run_training()`

The main training loop. It creates the Pytorch model based on the configuration options passed in, creates a Pytorch Lightning trainer, and trains the model. If a checkpoint was specified, then it loads the checkpoint before resuming training. The cwd is changed to the outputs folder so that the model can write to current working directory, and still everything is put into the right place in AzureML (only the contents of the "outputs" folder is treated as a result file).

Return type `None`

`run_validation()`

Run validation on the validation set for all models to save time/memory consuming outputs. This is done in inference only mode or when the user has requested an extra validation epoch. The cwd is changed to the outputs folder

Return type `None`

37.1 Classes

`TrainingRunner`(experiment_config, container)	Driver class to run an ML experiment.
`Runner`(project_root)	This class contains the high-level logic to start a training run: choose a model configuration by name, submit to AzureML if needed, or otherwise start the actual training and test loop.

37.1.1 TrainingRunner

`class` `health_ml.`**`TrainingRunner`**(*experiment_config*, *container*, *project_root=None*)

Bases: `health_ml.runner_base.RunnerBase`

Driver class to run an ML experiment. Note that the project root argument MUST be supplied when using hi-ml as a package!

Parameters

- **`experiment_config`** (`ExperimentConfig`) – The ExperimentConfig object to use for training.
- **`container`** (`LightningContainer`) – The LightningContainer object to use for training.
- **`project_root`** (`Optional[Path]`) – Project root. This should only be omitted if calling run_ml from the test suite. Supplying it is crucial when using hi-ml as a package or submodule!

Methods Summary

after_ddp_cleanup(environ_before_training)	Run processes cleanup after ddp context to prepare for single device inference.
end_training(environ_before_training)	Cleanup after training is done.
get_data_module()	Reads the datamodule that should be used for training or valuation from the container.
get_multiple_trainloader_mode()	**rtype** `str`
init_inference()	Prepare the trainer for running inference on the validation and test set.
init_training()	Execute some bookkeeping tasks only once if running distributed and initialize the runner's trainer object.
is_crossval_disabled_or_child_0()	Returns True if the present run is a non-cross-validation run, or child run 0 of a cross-validation run.
run()	Driver function to run a ML experiment
run_regression_test()	**rtype** `None`
run_training()	The main training loop.
run_validation()	Run validation on the validation set for all models to save time/memory consuming outputs.

Methods Documentation

after_ddp_cleanup(*environ_before_training*)

Run processes cleanup after ddp context to prepare for single device inference. Kill all processes in DDP besides rank 0.

Return type `None`

end_training(*environ_before_training*)

Cleanup after training is done. This is called after the trainer has finished fitting the data. This is called to update the checkpoint handler state and remove redundant checkpoint files. If running inference on a single device, this is also called to kill all processes besides rank 0.

Return type `None`

get_data_module()

Reads the datamodule that should be used for training or valuation from the container. This must be overridden in subclasses.

Return type `LightningDataModule`

get_multiple_trainloader_mode()

Return type `str`

init_inference()

Prepare the trainer for running inference on the validation and test set. This chooses a checkpoint, initializes the PL Trainer object, and chooses the right data module. The hook for running inference on the validation set is run (*LightningContainer.on_run_extra_validation_epoch*) is first called to reflect any changes to the model or datamodule states before running inference.

Return type `None`

`init_training()`
Execute some bookkeeping tasks only once if running distributed and initialize the runner's trainer object.

Return type `None`

`is_crossval_disabled_or_child_0()`
Returns True if the present run is a non-cross-validation run, or child run 0 of a cross-validation run.

Return type `bool`

`run()`
Driver function to run a ML experiment

Return type `None`

`run_regression_test()`

Return type `None`

`run_training()`
The main training loop. It creates the Pytorch model based on the configuration options passed in, creates a Pytorch Lightning trainer, and trains the model. If a checkpoint was specified, then it loads the checkpoint before resuming training. The cwd is changed to the outputs folder so that the model can write to current working directory, and still everything is put into the right place in AzureML (only the contents of the "outputs" folder is treated as a result file).

Return type `None`

`run_validation()`
Run validation on the validation set for all models to save time/memory consuming outputs. This is done in inference only mode or when the user has requested an extra validation epoch. The cwd is changed to the outputs folder

Return type `None`

`after_ddp_cleanup(`*environ_before_training*`)`
Run processes cleanup after ddp context to prepare for single device inference. Kill all processes in DDP besides rank 0.

Return type `None`

`end_training(`*environ_before_training*`)`
Cleanup after training is done. This is called after the trainer has finished fitting the data. This is called to update the checkpoint handler state and remove redundant checkpoint files. If running inference on a single device, this is also called to kill all processes besides rank 0.

Return type `None`

`get_data_module()`
Reads the datamodule that should be used for training or valuation from the container. This must be overridden in subclasses.

Return type `LightningDataModule`

`init_inference()`
Prepare the trainer for running inference on the validation and test set. This chooses a checkpoint, initializes the PL Trainer object, and chooses the right data module. The hook for running inference on the validation

set is run (*LightningContainer.on_run_extra_validation_epoch*) is first called to reflect any changes to the model or datamodule states before running inference.

Return type None

init_training()

Execute some bookkeeping tasks only once if running distributed and initialize the runner's trainer object.

Return type None

is_crossval_disabled_or_child_0()

Returns True if the present run is a non-cross-validation run, or child run 0 of a cross-validation run.

Return type bool

run()

Driver function to run a ML experiment

Return type None

run_training()

The main training loop. It creates the Pytorch model based on the configuration options passed in, creates a Pytorch Lightning trainer, and trains the model. If a checkpoint was specified, then it loads the checkpoint before resuming training. The cwd is changed to the outputs folder so that the model can write to current working directory, and still everything is put into the right place in AzureML (only the contents of the "outputs" folder is treated as a result file).

Return type None

run_validation()

Run validation on the validation set for all models to save time/memory consuming outputs. This is done in inference only mode or when the user has requested an extra validation epoch. The cwd is changed to the outputs folder

Return type None

37.1.2 Runner

class health_ml.**Runner**(*project_root*)

Bases: object

This class contains the high-level logic to start a training run: choose a model configuration by name, submit to AzureML if needed, or otherwise start the actual training and test loop.

Parameters **project_root** (Path) – The root folder that contains all of the source code that should be executed.

Methods Summary

additional_environment_variables()	**rtype** Dict[str, str]
additional_run_tags(script_params)	Gets the set of tags that will be added to the AzureML run as metadata.
parse_and_load_model()	Parses the command line arguments, and creates configuration objects for the model itself, and for the Azure-related parameters.

continues on next page

Table 3 – continued from previous page

run()	The main entry point for training and testing models from the commandline.
run_in_situ(azure_run_info)	Actually run the AzureML job; this method will typically run on an Azure VM.
submit_to_azureml_if_needed()	Submit a job to AzureML, returning the resulting Run object, or exiting if we were asked to wait for completion and the Run did not succeed.
validate()	Runs sanity checks on the whole experiment.

Methods Documentation

additional_environment_variables()

Return type `Dict[str, str]`

additional_run_tags(*script_params*)

Gets the set of tags that will be added to the AzureML run as metadata.

Parameters **script_params** (`List[str]`) – The commandline arguments used to invoke the present script.

Return type `Dict[str, str]`

parse_and_load_model()

Parses the command line arguments, and creates configuration objects for the model itself, and for the Azure-related parameters. Sets self.experiment_config to its proper values. Returns the parser output from parsing the model commandline arguments.

Return type `ParserResult`

Returns ParserResult object containing args, overrides and settings

run()

The main entry point for training and testing models from the commandline. This chooses a model to train via a commandline argument, runs training or testing, and writes all required info to disk and logs.

Return type `Tuple[LightningContainer, AzureRunInfo]`

Returns a tuple of the LightningContainer object and an AzureRunInfo containing all information about the present run (whether running in AzureML or not)

run_in_situ(*azure_run_info*)

Actually run the AzureML job; this method will typically run on an Azure VM.

Parameters **azure_run_info** (*AzureRunInfo*) – Contains all information about the present run in AzureML, in particular where the

datasets are mounted.

Return type `None`

submit_to_azureml_if_needed()

Submit a job to AzureML, returning the resulting Run object, or exiting if we were asked to wait for completion and the Run did not succeed.

Return type *AzureRunInfo*

Returns an AzureRunInfo object containing all of the details of the present run. If AzureML is not specified, the attribute 'run' will None, but the object still contains helpful information about datasets etc

`validate()`
Runs sanity checks on the whole experiment.

Return type `None`

`additional_run_tags`(*script_params*)
Gets the set of tags that will be added to the AzureML run as metadata.

Parameters **`script_params`** (`List[str]`) – The commandline arguments used to invoke the present script.

Return type `Dict[str, str]`

`parse_and_load_model()`
Parses the command line arguments, and creates configuration objects for the model itself, and for the Azure-related parameters. Sets self.experiment_config to its proper values. Returns the parser output from parsing the model commandline arguments.

Return type `ParserResult`

Returns ParserResult object containing args, overrides and settings

`run()`
The main entry point for training and testing models from the commandline. This chooses a model to train via a commandline argument, runs training or testing, and writes all required info to disk and logs.

Return type `Tuple[LightningContainer, AzureRunInfo]`

Returns a tuple of the LightningContainer object and an AzureRunInfo containing all information about the present run (whether running in AzureML or not)

`run_in_situ`(*azure_run_info*)
Actually run the AzureML job; this method will typically run on an Azure VM.

Parameters **`azure_run_info`** (`AzureRunInfo`) – Contains all information about the present run in AzureML, in particular where the

datasets are mounted.

Return type `None`

`submit_to_azureml_if_needed()`
Submit a job to AzureML, returning the resulting Run object, or exiting if we were asked to wait for completion and the Run did not succeed.

Return type `AzureRunInfo`

Returns an AzureRunInfo object containing all of the details of the present run. If AzureML is not specified, the attribute 'run' will None, but the object still contains helpful information about datasets etc

`validate()`
Runs sanity checks on the whole experiment.

Return type `None`

CHAPTER THIRTYEIGHT

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX