Ride Documentation

Release 0.7.3

Lukas Hedegaard

Apr 17, 2023

GETTING STARTED

CHAPTER

ONE

INTRODUCTION

Training wheels, side rails, and helicopter parent for your Deep Learning projects in PyTorch.

```
pip install ride
```

1.1 ZERO-boilerplate AI research

`Ride` provides a feature-rich, battle-tested boilerplate, so that you can focus on the model-building and research.

Out of the box, `Ride` gives you:

- **Training and testing methods**
- **Checkpointing**
- **Metrics**
- **Finetuning schemes**
- **Feature extraction**
- **Visualisations**
- **Hyperparameter search**
- **Logging**
- **Command-line interface**
- **Multi-gpu, multi-node handling via**
- *... and more*

1.2 Boilerplate inheritance

With `Ride`, we inject functionality by means of *inheritance*. The same way, your network would usually inherit from `torch.nn.Module`, we can *mix in* a plethora of functionality by inheriting from the `RideModule` (which also includes the `torch.nn.Module`). In addition, boiler-plate for wiring up optimisers, metrics and datasets can be also *mixed in* as seen below.

1.2.1 Complete project definition

```python
# simple_classifier.py
import torch
import ride
import numpy as np
from .examples import MnistDataset


class SimpleClassifier(
    ride.RideModule,
    ride.SgdOneCycleOptimizer,
    ride.TopKAccuracyMetric(1,3),
    MnistDataset,
):
    def __init__(self, hparams):
        # `self.input_shape` and `self.output_shape` were injected via `MnistDataset`
        self.l1 = torch.nn.Linear(np.prod(self.input_shape), self.hparams.hidden_dim)
        self.l2 = torch.nn.Linear(self.hparams.hidden_dim, self.output_shape)

    def forward(self, x):
        x = x.view(x.size(0), -1)
        x = torch.relu(self.l1(x))
        x = torch.relu(self.l2(x))
        return x

    @staticmethod
    def configs():
        c = ride.Configs()
        c.add(
            name="hidden_dim",
            type=int,
            default=128,
            strategy="choice",
            choices=[128, 256, 512, 1024],
            description="Number of hidden units.",
        )
        return c


if __name__ == "__main__":
    ride.Main(SimpleClassifier).argparse()
```

The above is the **complete** code for a simple classifier on the MNIST dataset.

All of the usual boiler-plate code has been *mixed in* using multiple inheritance:

- `RideModule` is a base-module which includes `pl.LightningModule` and makes some behind-the-scenes python-magic work. For instance, it modifies your `__init__` function to automatically initiate all the mixins correctly. Moreover, it mixes in `training_step`, `validation_step`, and `test_step`.
- `SgdOneCycleOptimizer` mixes in a `configure_optimizers` functionality with SGD and OneCycleLR scheduler.
- `TopKAccuracyMetric` adds top1acc and top3acc metrics, which can be used for checkpointing and benchmarking.

- `MnistDataset` mixes in `train_dataloader`, `val_dataloader`, and `test_dataloader` functions for the MNIST dataset. Dataset mixins always provide `input_shape` and `output_shape` attributes, which are handy for defining the networking structure as seen in `__init__`.

1.3 Configs

In addition to inheriting lifecycle functions etc., the mixins also add `configs` to your module (powered by co-rider). These define all of the configurable (hyper)parameters including their

- *type*
- *default* value
- *description* in plain text (reflected in command-line interface),
- *choices* defines accepted input range
- *strategy* specifies how hyperparameter-search tackles the parameter.

Configs specific to the SimpleClassifier can be added by overloading the `configs` methods as shown in the example.

The final piece of sorcery is the `Main` class, which adds a complete command-line interface.

1.4 Command-line interface

1.4.1 Train and test

```
$ python simple_classifier.py --train --test --learning_rate 0.01 --hidden_dim 256 --max_
↪epochs 1
```

- *Example output:*

```
lightning: Global seed set to 123
ride: Running on host HostName
ride:  View project repository at https://github.com/UserName/project_name/tree/
↪commit_hash
ride: Run data is saved locally at /Users/UserName/project_name/logs/run_logs/your_
↪id/version_1
ride: Logging using Tensorboard
ride:  Saving /Users/au478108/Projects/ride/logs/run_logs/your_id/version_1/hparams.
↪yaml
ride:  Running training
ride:  Checkpointing on val/loss with optimisation direction min
lightning: GPU available: False, used: False
lightning: TPU available: False, using: 0 TPU cores
lightning:
  | Name | Type   | Params
---------------------------------
0 | l1   | Linear | 200 K
1 | l2   | Linear | 2.6 K
---------------------------------
203 K     Trainable params
0         Non-trainable params
```

(continues on next page)

(continued from previous page)

```
203 K     Total params
0.814     Total estimated model params size (MB)
lightning: Global seed set to 123

Epoch 0: 100%|| 3751/3751 [00:20<00:00, 184.89it/s, loss=0.785, v_num=9, step_train/
↪loss=0.762]
lightning: Epoch 0, global step 3437: val/loss reached 0.77671 (best 0.77671),␣
↪saving model to "/Users/UserName/project_name/logs/run_logs/your_id/version_1/
↪checkpoints/epoch=0-step=3437.ckpt" as top 1
lightning: Saving latest checkpoint...
Epoch 0: 100%|| 3751/3751 [00:20<00:00, 184.65it/s, loss=0.785, v_num=9, step_train/
↪loss=0.762]
ride:  Running evaluation on test set
Testing: 100%|| 625/625 [00:01<00:00, 358.86it/s]
--------------------------------------
DATALOADER:0 TEST RESULTS
{'loss': 0.7508705258369446,
'test/loss': 0.7508705258369446,
'test/top1acc': 0.7986000180244446,
'test/top3acc': 0.8528000116348267}
--------------------------------------

ride:  Saving /Users/UserName/project_name/logs/run_logs/your_id/version_1/test_
↪results.yaml
```

1.4.2 Feature extraction and visualisation

Extract features after layer `l1` and visualise them with UMAP.

```
$ python simple_classifier.py --train --test --extract_features_after_layer = "l1" --
↪visualise_features = "umap"
```

- *Example output:*

1.4.3 Confusion matrix visualisation

Plot the confution matrix for the test set.

```
$ python simple_classifier.py --train --test --test_confusion_matrix 1
```

- *Example output:*

1.4.4 Advanced model finetuning

Load model and finetune with gradual unfreeze and discriminative learning rates

```
$ python simple_classifier.py --train --finetune_from_weights your/path.ckpt --unfreeze_
↪layers_initial 1 --unfreeze_epoch_step 1 --unfreeze_from_epoch 0 --discriminative_lr_
↪fraction 0.1
```

1.4.5 Hyperparameter optimization

If we want to perform **hyperparameter optimisation** across four gpus, we can run:

```
$ python simple_classifier.py --hparamsearch --gpus 4
```

Curretly, we use Ray Tune and the ASHA algorithm under the hood.

1.4.6 Profile model

You can check the **timing** and **FLOPs** of the model with:

```
$ python simple_classifier.py --profile_model
```

- *Example output:*

```
Results:
  flops: 203530
  machine:
    cpu:
      architecture: x86_64
      cores:
        physical: 6
        total: 12
      frequency: 2.60 GHz
      model: Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz
    gpus: null
    memory:
      available: 5.17 GB
      total: 16.00 GB
      used: 8.04 GB
    system:
      node: d40049
      release: 19.6.0
      system: Darwin
  params: 203530
  timing:
    batch_size: 16
    num_runs: 10000
    on_gpu: false
    samples_per_second: 88194.303 +/- 17581.377 [20177.049, 113551.377]
    time_per_sample: 12.031us +/- 3.736us [8.807us, 49.561us]
```

1.4.7 Additional options

For additional configuration options, check out the help:

```
$ python simple_classifier.py --help
```

- *Truncated output:*

```
Flow:
  Commands that control the top-level flow of the programme.

  --hparamsearch        Run hyperparameter search. The best hyperparameters
                        will be used for subsequent lifecycle methods
  --train               Run model training
  --validate            Run model evaluation on validation set
  --test                Run model evaluation on test set
  --profile_model       Profile the model

General:
  Settings that apply to the programme in general.

  --id ID               Identifier for the run. If not specified, the current
                        timestamp will be used (Default: 202101011337)
  --seed SEED           Global random seed (Default: 123)
  --logging_backend {tensorboard,wandb}
                        Type of experiment logger (Default: tensorboard)
  ...

Pytorch Lightning:
  Settings inherited from the pytorch_lightning.Trainer
  ...
  --gpus GPUS           number of gpus to train on (int) or which GPUs to
                        train on (list or str) applied per node
  ...

Hparamsearch:
  Settings associated with hyperparameter optimisation
  ...

Module:
  Settings associated with the Module
  --loss {mse_loss,l1_loss,nll_loss,cross_entropy,binary_cross_entropy,...}
                        Loss function used during optimisation.
                        (Default: cross_entropy)
  --batch_size BATCH_SIZE
                        Dataloader batch size. (Default: 64)
  --num_workers NUM_WORKERS
                        Number of CPU workers to use for dataloading.
                        (Default: 10)
  --learning_rate LEARNING_RATE
                        Learning rate. (Default: 0.1)
  --weight_decay WEIGHT_DECAY
                        Weight decay. (Default: 1e-05)
  --momentum MOMENTUM   Momentum. (Default: 0.9)
```

(continues on next page)

(continued from previous page)

```
  --hidden_dim HIDDEN_DIM {128, 256, 512, 1024}
                        Number of hidden units. (Defualt: 128)
  --extract_features_after_layer EXTRACT_FEATURES_AFTER_LAYER
                        Layer name after which to extract features. Nested
                        layers may be selected using dot-notation, e.g.
                        `block.subblock.layer1` (Default: )
  --visualise_features {,umap,tsne,pca}
                        Visualise extracted features using selected
                        dimensionality reduction method. Visualisations are
                        created only during evaluation. (Default: )
  --finetune_from_weights FINETUNE_FROM_WEIGHTS
                        Path to weights to finetune from. Allowed extension
                        include {'.ckpt', '.pyth', '.pth', '.pkl',
                        '.pickle'}. (Default: )
  --unfreeze_from_epoch UNFREEZE_FROM_EPOCH
                        Number of epochs to wait before starting gradual
                        unfreeze. If -1, unfreeze is omitted. (Default: -1)
  --test_confusion_matrix {0,1}
                        Create and save confusion matrix for test data.
                        (Default: 0)
  ...
```

Though the above `--help` printout was truncated for readibility, there's still a lot going on! The general structure is a follows: First, there are flags for controlling the programme flow (e.g. whether to run hparamsearch or training), then some general parameters (id, seed, etc.), all the parameters from Pytorch Lightning, hparamsearch-related arguments, and finally the Module-specific arguments, which we either in the `SimpleClassifier` or inherited from the RideModule and mixins.

1.5 Environment

Per default, `Ride` projects are oriented around the current working directory and will save logs in the `~/logs folders`, and cache to `~/.cache`.

This behaviour can be overloaded by changing of the following environment variables (defaults noted):

```
ROOT_PATH="~/"
CACHE_PATH=".cache"
DATASETS_PATH="datasets"  # Dir relative to ROOT_PATH
LOGS_PATH="logs"          # Dir relative to ROOT_PATH
RUN_LOGS_PATH="run_logs"  # Dir relative to LOGS_PATH
TUNE_LOGS_PATH="tune_logs"# Dir relative to LOGS_PATH
LOG_LEVEL="INFO"          # One of "DEBUG", "INFO", "WARNING", "ERROR", "CRITICAL"
```

1.6 Examples

1.6.1 Library Examples

- SimpleClassifier
- MNIST Dataloader

1.6.2 Community Examples

Video-based human action recognition:

- I3D
- R(2+1)D
- SlowFast
- CoSlow
- X3D
- CoX3D

Skeleton-based human action recognition:

- ST-GCN
- CoST-GCN
- A-GCN
- CoA-GCN
- S-Tr
- CoS-Tr

1.7 Citation

1.7.1 BibTeX

If you use `Ride` for your research and feel like citing it, here's a BibTex:

```
@article{hedegaard2021ride,
  title={Ride},
  author={Lukas Hedegaard},
  journal={GitHub. Note: https://github.com/LukasHedegaard/ride},
  year={2021}
}
```

1.7.2 Badge

.MD

```
[![Ride](https://img.shields.io/badge/Built_to-Ride-643DD9.svg)](https://github.com/
↪LukasHedegaard/ride)
```

.HTML

```
<a href="https://github.com/LukasHedegaard/ride">
  <img src="https://img.shields.io/badge/Built_to-Ride-643DD9.svg" height="20">
</a>
```

CHAPTER

TWO

RIDEMODULE

The *RideModule* works in conjunction with the `LightningModule`, to add functionality to a plain `Module`. While `LightningModule` adds a bunch of structural code, that integrates with the `Trainer`, the *RideModule* provides good defaults for

- Train loop - `training_step()`
- Validation loop - `validation_step()`
- Test loop - `test_step()`
- Optimizers - `configure_optimizers()`

The only things left to be defined are

- Initialisation - `__init__()`.
- Network forward pass - `forward()`.
- *Dataset*

The following thus constitutes a fully functional Neural Network module, which (when integrated with *ride.Main*) provides full functionality for training, testing, hyperparameters search, profiling , etc., via a command line interface.

```
from ride import RideModule
from .examples.mnist_dataset import MnistDataset

class MyRideModule(RideModule, MnistDataset):
    def __init__(self, hparams):
        hidden_dim = 128
        # `self.input_shape` and `self.output_shape` were injected via `MnistDataset`
        self.l1 = torch.nn.Linear(np.prod(self.input_shape), hidden_dim)
        self.l2 = torch.nn.Linear(hidden_dim, self.output_shape)

    def forward(self, x):
        x = x.view(x.size(0), -1)
        x = torch.relu(self.l1(x))
        x = torch.relu(self.l2(x))
        return x
```

2.1 Configs

Out of the box, a wide selection parameters are integrated into *self.hparams* through *ride.Main*. These include all the `pytorch_lightning.Trainer` options, as well as configs in *ride.lifecycle.Lifecycle.configs()*, the selected optimizer (default: *ride.optimizers.SgdOptimizer.configs()*).

User-defined hyperparameters, which are reflected *self.hparams*, the command line interface, and hyperparameter serach space (by selection of *choices* and *strategy*), are easily defined by defining a *configs* method `MyRideModule`:

```
@staticmethod
def configs() -> ride.Configs:
    c = ride.Configs()
    c.add(
        name="hidden_dim",
        type=int,
        default=128,
        strategy="choice",
        choices=[128, 256, 512, 1024],
        description="Number of hidden units.",
    )
    return c
```

The configs package is also available seperately in the Co-Rider package.

2.2 Advanced behavior overloading

2.2.1 Lifecycle methods

Naturally, the `training_step()`, `validation_step()`, and `test_step()` can still be overloaded if complex computational schemes are required. In that case, ending the function with `common_step()` will ensure that loss computation and collection of metrics still works as expected:

```
def training_step(self, batch, batch_idx=None):
    x, target = batch
    pred = self.forward(x)  # replace with complex interaction
    return self.common_step(pred, target, prefix="train/", log=True)
```

2.2.2 Loss

By default, *RideModule* automatically integrates the loss functions in `torch.nn.functional` (set by command line using the "–loss" flag). If other options are needed, one can define the `self.loss()` in the module.

```
def loss(self, pred, target):
    return my_exotic_loss(pred, target)
```

2.2.3 Optimizer

The *SgdOptimizer* is added automatically if no other `Optimizer` is found and `configure_optimizers()` is not manually defined. Other optimizers can thus be specified by using either Mixins:

```
class MyModel(
    ride.RideModule,
    ride.AdamWOneCycleOptimizer
):
    def __init__(self, hparams):
        ...
```

or function overloading:

```
def configure_optimizers(self):
    optimizer = torch.optim.Adam(self.parameters(), lr=1e-3)
    return optimizer
```

While the specifying parent Mixins automatically adds *ride.AdamWOneCycleOptimizer.configs()* and hparams, the function overloading approach must be supplemented with a `configs()` methods in order to reflect the parameter in the command line tool and hyperparameter search space.

```
@staticmethod
def configs() -> ride.Configs:
    c = ride.Configs()
    c.add(
        name="learning_rate",
        type=float,
        default=0.1,
        choices=(1e-6, 1),
        strategy="loguniform",
        description="Learning rate.",
    )

def configure_optimizers(self):
    optimizer = torch.optim.Adam(self.parameters(), lr=self.hparams.learning_rate)
    return optimizer
```

Next, we'll see how to specify dataset.

CHAPTER

THREE

DATASETS

In PyTorch Lightning, datasets can be integrated by overloading dataloader functions in the `LightningModule`:

- `train_dataloader()`
- `val_dataloader()`
- `test_dataloader()`

This is exactly what a `RideDataset` does. In addition, it adds `num_workers` and `batch_size` `configs` as well as `self.input_shape` and `self.output_shape` tuples (which are very handy for computing layer shapes).

For classification dataset, the `RideClassificationDataset` expects a list of class-names defined in `self.classes` and provides a `self.num_classes` attribute. `self.classes` are then used plotting, e.g. if "–test_confusion_matrix True" is specified in the CLI.

In order to define a `RideDataset`, one can either define the `train_dataloader()`, `val_dataloader()`, `test_dataloader()` and functions or assign a `LightningDataModule` to `self.datamodule` as seen here:

```
from ride.core import AttributeDict, RideClassificationDataset, Configs
from ride.utils.env import DATASETS_PATH
import pl_bolts

class MnistDataset(RideClassificationDataset):

    @staticmethod
    def configs():
        c = Configs.collect(MnistDataset)
        c.add(
            name="val_split",
            type=int,
            default=5000,
            strategy="constant",
            description="Number samples from train dataset used for val split.",
        )
        c.add(
            name="normalize",
            type=int,
            default=1,
            choices=[0, 1],
            strategy="constant",
            description="Whether to normalize dataset.",
        )
        return c
```

(continues on next page)

(continued from previous page)

```
    def __init__(self, hparams: AttributeDict):
        self.datamodule = pl_bolts.datamodules.MNISTDataModule(
            data_dir=DATASETS_PATH,
            val_split=self.hparams.val_split,
            num_workers=self.hparams.num_workers,
            normalize=self.hparams.normalize,
            batch_size=self.hparams.batch_size,
            seed=42,
            shuffle=True,
            pin_memory=self.hparams.num_workers > 1,
            drop_last=False,
        )
        self.output_shape = 10
        self.classes = list(range(10))
        self.input_shape = self.datamodule.dims
```

3.1 Changing dataset

Though the dataset is specified at module definition, we can change the dataset using `with_dataset()`. This is especially handy for experiments using a single module over multiple datasets:

```
MyRideModuleWithMnistDataset = MyRideModule.with_dataset(MnistDataset)
MyRideModuleWithCifar10Dataset = MyRideModule.with_dataset(Cifar10Dataset)
...
```

Next, we'll cover how the *RideModule* integrates with *Main*.

CHAPTER

FOUR

MAIN

The *Main* class wraps a *RideModule* to supply a fully functional command-line interface which includes

- Training (”–train”)
- Evaluation on validation set (”–validate”)
- Evaluation on test set (”–test”)
- Logger integration (”–logging_backend”)
- Hyperparameter search (”–hparamsearch”)
- Hyperparameter file loading (”–from_hparams_file”)
- Profiling of model timing, flops, and params (”–profile_model”)
- Checkpointing
- Checkpoint loading (”–resume_from_checkpoint”)

4.1 Example

All it takes to get a working CLI is to add the following to the bottom of a file:

```
# my_ride_module.py

import numpy as np
from ride import RideModule, TopKAccuracyMetric
from .examples.mnist_dataset import MnistDataset

class MyRideModule(RideModule, TopKAccuracyMetric(1,3), MnistDataset):
    def __init__(self, hparams):
        # `self.input_shape` and `self.output_shape` were injected via `MnistDataset`
        self.lin = torch.nn.Linear(np.prod(self.input_shape), self.output_shape)

    def forward(self, x):
        x = x.view(x.size(0), -1)
        x = torch.relu(self.lin(x))
        return x

ride.Main(MyRideModule).argparse()  # <-- Add this
```

and executing from the command line:

```
>> python my_ride_module.py --train --test --max_epochs 1 --id my_first_run

lightning: Global seed set to 123
ride: Running on host d40049
ride:  View project repository at https://github.com/username/ride/tree/hash
ride: Run data is saved locally at /Users/username/project_folder/logs/run_logs/my_first_
↪run/version_0
ride: Logging using Tensorboard
ride:  Running training
ride: Checkpointing on val/loss with optimisation direction min
lightning: GPU available: False, used: False
lightning: TPU available: None, using: 0 TPU cores
lightning:
| Name | Type   | Params
---------------------------------
0 | l1   | Linear | 100 K
1 | l2   | Linear | 1.3 K
---------------------------------
101 K     Trainable params
0         Non-trainable params
101 K     Total params
0.407     Total estimated model params size (MB)
Epoch 0: 100%|| 3751/3751 [00:16<00:00, 225.44it/s, loss=0.762, v_num=0, step_train/
↪loss=0.899]
lightning: Epoch 0, global step 3437: val/loss reached 0.90666 (best 0.90666), saving␣
↪model to "/Users/username/project_folder/logs/run_logs/my_first_run/version_0/
↪checkpoints/epoch=0-step=3437.ckpt" as top 1
Epoch 1: 100%|| 3751/3751 [00:17<00:00, 210.52it/s, loss=0.581, v_num=1, step_train/
↪loss=0.0221]
lightning: Epoch 1, global step 3437: val/loss reached 0.61922 (best 0.61922), saving␣
↪model to "/Users/username/project_folder/logs/run_logs/my_first_run/version_0/
↪checkpoints/epoch=1-step=6875.ckpt" as top 1
lightning: Saving latest checkpoint...
ride:  Running evaluation on test set
Testing: 100%|| 625/625 [00:01<00:00, 432.69it/s]
--------------------------------------------------------------------------------
ride: Results:
test/epoch: 0.000000000
test/loss: 0.889312625
test/top1acc: 0.739199996
test/top3acc: 0.883000016

ride: Saving /Users/username/project_folder/ride/logs/my_first_run/version_0/evaluation/
↪test_results.yaml
```

4.2 Help

The best way to explore all the options available is to run the "–help"

```
>> python my_ride_module.py --help

...

Flow:
Commands that control the top-level flow of the programme.

--hparamsearch        Run hyperparameter search. The best hyperparameters
                        will be used for subsequent lifecycle methods
--train               Run model training
--validate            Run model evaluation on validation set
--test                Run model evaluation on test set
--profile_model       Profile the model

General:
Settings that apply to the programme in general.

--id ID               Identifier for the run. If not specified, the current
                        timestamp will be used (Default: 202101011337)
--seed SEED           Global random seed (Default: 123)
--logging_backend {tensorboard,wandb}
                        Type of experiment logger (Default: tensorboard)
...

Pytorch Lightning:
Settings inherited from the pytorch_lightning.Trainer
...
--gpus GPUS           number of gpus to train on (int) or which GPUs to
                        train on (list or str) applied per node
...

Hparamsearch:
Settings associated with hyperparameter optimisation
...

Module:
Settings associated with the Module
--loss {mse_loss,l1_loss,nll_loss,cross_entropy,binary_cross_entropy,...}
                        Loss function used during optimisation.
                        (Default: cross_entropy)
--batch_size BATCH_SIZE
                        Dataloader batch size. (Default: 64)
--num_workers NUM_WORKERS
                        Number of CPU workers to use for dataloading.
                        (Default: 10)
--learning_rate LEARNING_RATE
                        Learning rate. (Default: 0.1)
--weight_decay WEIGHT_DECAY
                        Weight decay. (Default: 1e-05)
```

(continues on next page)

(continued from previous page)

```
--momentum MOMENTUM   Momentum. (Default: 0.9)
...
```

CHAPTER

FIVE

API REFERENCE

This page contains auto-generated API reference documentation[1].

5.1 ride

5.1.1 Subpackages

ride.utils

Submodules

ride.utils.checkpoints

Module Contents

Functions

latest_file_in(→ pathlib.Path)
get_latest_checkpoint(→ pathlib.Path)
find_checkpoint(→ str)

ride.utils.checkpoints.**latest_file_in**(*path: pathlib.Path*) → pathlib.Path

ride.utils.checkpoints.**get_latest_checkpoint**(*log_dir: str*) → pathlib.Path

ride.utils.checkpoints.**find_checkpoint**(*path: str*) → str

[1] Created with sphinx-autoapi

ride.utils.discriminative_lr

Module Contents

Classes

PrePostInitMeta	A metaclass that calls optional *__pre_init__* and *__post_init__* methods
Module	Same as *nn.Module*, but no need for subclasses to call *super().__init__*
ParameterModule	Register a lone parameter *p* in a module.

Functions

children(m)	Get children of *m*.
num_children(m)	Get number of children modules in *m*.
children_and_parameters(m)	Return the children of *m* and its direct parameters not registered in modules.
even_mults(→ numpy.ndarray)	Build log-stepped array from *start* to *stop* in *n* steps.
lr_range(→ numpy.ndarray)	Build differential learning rates from *lr*.
unfreeze_layers(→ None)	Unfreeze or freeze all layers
build_param_dicts(→ Union[int, list])	Either return the number of layers with requires_grad is True
discriminative_lr(→ Union[list, numpy.ndarray, ...)	Flatten our model and generate a list of dictionnaries to be passed to the

Attributes

logger	Developped by the Fastai team for the Fastai library
flatten_model	Modified version of lr_range from fastai

ride.utils.discriminative_lr.**logger**

Developped by the Fastai team for the Fastai library From the fastai library https://www.fast.ai and https://github.com/fastai/fastai

class ride.utils.discriminative_lr.**PrePostInitMeta**

Bases: type

A metaclass that calls optional *__pre_init__* and *__post_init__* methods

class ride.utils.discriminative_lr.**Module**

Bases: torch.nn.Module

Same as *nn.Module*, but no need for subclasses to call *super().__init__*

__pre_init__()

class ride.utils.discriminative_lr.**ParameterModule**(*p: torch.nn.Parameter*)

Bases: *Module*

Register a lone parameter *p* in a module.

forward(*x*)

ride.utils.discriminative_lr.**children**(*m: torch.nn.Module*)

Get children of *m*.

ride.utils.discriminative_lr.**num_children**(*m: torch.nn.Module*)

Get number of children modules in *m*.

ride.utils.discriminative_lr.**children_and_parameters**(*m: torch.nn.Module*)

Return the children of *m* and its direct parameters not registered in modules.

ride.utils.discriminative_lr.**even_mults**(*start: float*, *stop: float*, *n: int*) → numpy.ndarray

Build log-stepped array from *start* to *stop* in *n* steps.

ride.utils.discriminative_lr.**flatten_model**

Modified version of lr_range from fastai https://github.com/fastai/fastai/blob/master/fastai/basic_train.py#L185

ride.utils.discriminative_lr.**lr_range**(*net: torch.nn.Module*, *lr: slice*, *model_len: int*) → numpy.ndarray

Build differential learning rates from *lr*.

ride.utils.discriminative_lr.**unfreeze_layers**(*model: torch.nn.Sequential*, *unfreeze: bool = True*) → None

Unfreeze or freeze all layers

ride.utils.discriminative_lr.**build_param_dicts**(*layers: torch.nn.Sequential*, *lr: list = [0]*, *return_len: bool = False*) → Union[int, list]

Either return the number of layers with requires_grad is True or return a list of dictionnaries containing each layers on its associated LR" Both weight and bias are check for requires_grad is True

ride.utils.discriminative_lr.**discriminative_lr**(*net: torch.nn.Module*, *lr: slice*, *unfreeze: bool = False*) → Union[list, numpy.ndarray, torch.nn.Sequential]

Flatten our model and generate a list of dictionnaries to be passed to the optimizer. - If only one learning rate is passed as a slice the last layer will have the corresponding learning rate and all other ones will have lr/10 - If two learning rates are passed such as slice(min_lr, max_lr) the last layer will have max_lr as a learning rate and the first one will have min_lr. All middle layers will have learning rates logarithmically interpolated ranging from min_lr to max_lr

ride.utils.env

Module Contents

ride.utils.env.**DATASETS_PATH**

ride.utils.env.**LOGS_PATH**

ride.utils.env.**RUN_LOGS_PATH**

ride.utils.env.**TUNE_LOGS_PATH**

ride.utils.env.**CACHE_PATH**

ride.utils.env.**LOG_LEVEL**

ride.utils.env.**NUM_CPU**

ride.utils.gpus

Module Contents

Functions

parse_gpus(→ List[int])	
parse_num_gpus(→ int)	

ride.utils.gpus.**parse_gpus**(*args_gpus: Optional[Union[int, str, List[int]]]*) → List[int]

ride.utils.gpus.**parse_num_gpus**(*args_gpus: Optional[Union[int, str, List[int]]]*) → int

ride.utils.io

Module Contents

Classes

NpJsonEncoder	Extensible JSON <http://json.org> encoder for Python data structures.

Functions

is_nonempty_file(→ bool)	
bump_version(→ pathlib.Path)	Bumps the version number for a path if it already exists
load_structured_data(path)	
dump_yaml(path, data)	
load_yaml(→ Any)	
dump_json(path, data)	
load_json(→ Any)	
float_representer(dumper, value)	
tensor_representer(dumper, data)	

ride.utils.io.**is_nonempty_file**(*path: Union[str, pathlib.Path]*) → bool

ride.utils.io.**bump_version**(*path: Union[str, pathlib.Path]*) → pathlib.Path

Bumps the version number for a path if it already exists

Example:

```
bump_version("folder/new_file.json") == Path("folder/new_file.json)
bump_version("folder/old_file.json") == Path("folder/old_file_1.json)
bump_version("folder/old_file_1.json") == Path("folder/old_file_2.json)
```

ride.utils.io.**load_structured_data**(*path: pathlib.Path*)

ride.utils.io.**dump_yaml**(*path: pathlib.Path*, *data: Any*)

ride.utils.io.**load_yaml**(*path: pathlib.Path*) → Any

ride.utils.io.**dump_json**(*path: pathlib.Path*, *data: Any*)

ride.utils.io.**load_json**(*path: pathlib.Path*) → Any

class ride.utils.io.**NpJsonEncoder**(**, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, sort_keys=False, indent=None, separators=None, default=None*)

Bases: `json.JSONEncoder`

Extensible JSON <http://json.org> encoder for Python data structures.

Supports the following objects and types by default:

Python	JSON
dict	object
list, tuple	array
str	string
int, float	number
True	true
False	false
None	null

To extend this to recognize other objects, subclass and implement a `.default()` method with another method that returns a serializable object for `o` if possible, otherwise it should call the superclass implementation (to raise `TypeError`).

default(*obj*)

Implement this method in a subclass such that it returns a serializable object for `o`, or calls the base implementation (to raise a `TypeError`).

For example, to support arbitrary iterators, you could implement default like this:

```
def default(self, o):
    try:
        iterable = iter(o)
    except TypeError:
        pass
    else:
        return list(iterable)
```

(continues on next page)

(continued from previous page)

```
    # Let the base class default method raise the TypeError
    return JSONEncoder.default(self, o)
```

ride.utils.io.**float_representer**(*dumper: yaml.Dumper*, *value: float*)

ride.utils.io.**tensor_representer**(*dumper: yaml.Dumper*, *data: torch.Tensor*)

ride.utils.logging

Module Contents

Functions

_process_rank()	
if_rank_zero(fn)	
getLogger(name[, log_once])	
style(text[, fg, bg, bold, dim, underline, blink, ...])	Styles a text with ANSI styles and returns the new string. By
style_logging()	
init_logging([logdir, logging_backend])	

Attributes

LOG_LEVELS	
process_rank	
logger	
_ansi_colors	
_ansi_reset_all	

ride.utils.logging.**LOG_LEVELS**

ride.utils.logging.**_process_rank**()

ride.utils.logging.**process_rank**

ride.utils.logging.**if_rank_zero**(*fn*)

ride.utils.logging.**getLogger**(*name*, *log_once=False*)

ride.utils.logging.**logger**

ride.utils.logging.**_ansi_colors**

ride.utils.logging.**_ansi_reset_all = '\x1b[0m'**

ride.utils.logging.**style**(*text*, *fg=None*, *bg=None*, *bold=None*, *dim=None*, *underline=None*, *blink=None*, *reverse=None*, *reset=True*)

Styles a text with ANSI styles and returns the new string. By default the styling is self contained which means that at the end of the string a reset code is issued. This can be prevented by passing `reset=False`.

This is a modified version of the one found in *click* https://click.palletsprojects.com/en/7.x/

Examples:

```
logger.info(style('Hello World!', fg='green'))
logger.info(style('ATTENTION!', blink=True))
logger.info(style('Some things', reverse=True, fg='cyan'))
```

Supported color names:

- `black` (might be a gray)
- `red`
- `green`
- `yellow` (might be an orange)
- `blue`
- `magenta`
- `cyan`
- `white` (might be light gray)
- `bright_black`
- `bright_red`
- `bright_green`
- `bright_yellow`
- `bright_blue`
- `bright_magenta`
- `bright_cyan`
- `bright_white`
- `reset` (reset the color code only)

Parameters

- **text** – the string to style with ansi codes.
- **fg** – if provided this will become the foreground color.
- **bg** – if provided this will become the background color.
- **bold** – if provided this will enable or disable bold mode.
- **dim** – if provided this will enable or disable dim mode. This is badly supported.

- **underline** – if provided this will enable or disable underline.
- **blink** – if provided this will enable or disable blinking.
- **reverse** – if provided this will enable or disable inverse rendering (foreground becomes background and the other way round).
- **reset** – by default a reset-all code is added at the end of the string which means that styles do not carry over. This can be disabled to compose styles.

ride.utils.logging.**style_logging**()

ride.utils.logging.**init_logging**(*logdir: str = None*, *logging_backend: str = 'tensorboard'*)

ride.utils.utils

Module Contents

Functions

is_shape(x)	Tests whether *x* is a shape, i.e. one of
once(fn)	
rsetattr(obj, attr, val)	
rgetattr(obj, attr, *args)	
attributedict(...)	If given a dict, it is converted it to an argparse.AttributeDict. Otherwise, no change is made
to_dict(d)	
merge_dicts(*args)	
merge_attributedicts(*args)	
some(self, attr)	
some_callable(self, attr[, min_num_args, max_num_args])	
get(self, attr)	
differ_and_exist(a, b)	
missing(→ Set[str])	
missing_or_not_in_other(→ Set[str])	
name(thing)	
prefix_keys(→ Dict)	
camel_to_snake(→ str)	Convert from camel-case to snake-case
temporary_parameter(obj, attr, val)	
flatten_dict(d[, parent_key, sep])	

Attributes

DictLike

ride.utils.utils.**DictLike**

ride.utils.utils.**is_shape**(*x: Any*)

Tests whether *x* is a shape, i.e. one of - int - List[int] - Tuple[int] - Namedtuple[int]

Parameters

x (*Any*) – instance to check

ride.utils.utils.**once**(*fn: Callable*)

ride.utils.utils.**rsetattr**(*obj*, *attr*, *val*)

ride.utils.utils.**rgetattr**(*obj*, *attr*, **args*)

ride.utils.utils.**attributedict**(*dict_like: DictLike*) → pytorch_lightning.utilities.parsing.AttributeDict

If given a dict, it is converted it to an argparse.AttributeDict. Otherwise, no change is made

ride.utils.utils.**to_dict**(*d*)

ride.utils.utils.**merge_dicts**(**args*)

ride.utils.utils.**merge_attributedicts**(**args*)

ride.utils.utils.**some**(*self*, *attr: str*)

ride.utils.utils.**some_callable**(*self*, *attr: str*, *min_num_args=0*, *max_num_args=math.inf*)

ride.utils.utils.**get**(*self*, *attr: str*)

ride.utils.utils.**differ_and_exist**(*a*, *b*)

ride.utils.utils.**missing**(*self*, *attrs: Collection[str]*) → Set[str]

ride.utils.utils.**missing_or_not_in_other**(*first*, *other*, *attrs: Collection[str]*, *must_be_callable=False*) → Set[str]

ride.utils.utils.**name**(*thing*)

ride.utils.utils.**prefix_keys**(*prefix: str*, *dictionary: Dict*) → Dict

ride.utils.utils.**camel_to_snake**(*s: str*) → str

Convert from camel-case to snake-case Source: https://stackoverflow.com/questions/1175208/elegant-python-function-to-convert-camelcase-to-snake-case

ride.utils.utils.**temporary_parameter**(*obj*, *attr*, *val*)

ride.utils.utils.**flatten_dict**(*d*, *parent_key=''*, *sep='_'*)

Package Contents

Functions

attributedict(...)	If given a dict, it is converted it to an argparse.AttributeDict. Otherwise, no change is made
flatten_dict(d[, parent_key, sep])	
name(thing)	
some(self, attr)	

ride.utils.**attributedict**(*dict_like: DictLike*) → pytorch_lightning.utilities.parsing.AttributeDict

If given a dict, it is converted it to an argparse.AttributeDict. Otherwise, no change is made

ride.utils.**flatten_dict**(*d*, *parent_key=''*, *sep='_'*)

ride.utils.**name**(*thing*)

ride.utils.**some**(*self*, *attr: str*)

5.1.2 Submodules

ride.core

Module Contents

Classes

Configs	Configs module for holding project configurations.
RideModule	Base-class for modules using the Ride ecosystem.
RideMixin	Abstract base-class for Ride mixins
DefaultMethods	Abstract base-class for Ride mixins
OptimizerMixin	Abstract base-class for Optimizer mixins
RideDataset	Base-class for Ride datasets.
RideClassificationDataset	Base-class for Ride classification datasets.

Functions

_init_subclass(cls)	
apply_init_args(fn, self, hparams, *args, **kwargs)	

Attributes

logger	
DataShape	

ride.core.**logger**

ride.core.**DataShape**

class ride.core.**Configs**

Bases: corider.Configs

Configs module for holding project configurations.

This is a wrapper of the Configs found as a stand-alone package in https://github.com/LukasHedegaard/co-rider

static collect(*cls:* RideModule) → *Configs*

Collect the configs from all class bases

Returns
Aggregated configurations

Return type
Configs

default_values()

ride.core.**_init_subclass**(*cls*)

ride.core.**apply_init_args**(*fn*, *self*, *hparams*, **args*, ***kwargs*)

class ride.core.**RideModule**

Base-class for modules using the Ride ecosystem.

This module should be inherited as the highest-priority parent (first in sequence).

Example:

```
class MyModule(ride.RideModule, ride.SgdOneCycleOptimizer):
    def __init__(self, hparams):
        ...
```

It handles proper initialisation of *RideMixin* parents and adds automatic attribute validation.

If *pytorch_lightning.LightningModule* is omitted as lowest-priority parent, *RideModule* will automatically add it.

If *training_step*, *validation_step*, and *test_step* methods are not found, the *ride.Lifecycle* will be automatically mixed in by this module.

property hparams: pytorch_lightning.utilities.parsing.AttributeDict

classmethod __init_subclass__()

classmethod with_dataset(*ds:* RideDataset)

class ride.core.**RideMixin**(*hparams: pytorch_lightning.utilities.parsing.AttributeDict*, **args*, ***kwargs*)

Bases: abc.ABC

Abstract base-class for Ride mixins

on_init_end(*hparams: pytorch_lightning.utilities.parsing.AttributeDict*, **args*, ***kwargs*)

validate_attributes()

class ride.core.**DefaultMethods**(*hparams: pytorch_lightning.utilities.parsing.AttributeDict*, **args*, ***kwargs*)

Bases: *RideMixin*

Abstract base-class for Ride mixins

warm_up(*input_shape: Sequence[int]*, **args*, ***kwargs*)

Warms up the model state with a dummy input of shape *input_shape*. This method is called prior to model profiling.

Parameters
input_shape (*Sequence[int]*) – input shape with which to warm the model up, including batch size.

class ride.core.**OptimizerMixin**(*hparams: pytorch_lightning.utilities.parsing.AttributeDict*, **args*, ***kwargs*)

Bases: *RideMixin*

Abstract base-class for Optimizer mixins

class ride.core.**RideDataset**(*hparams: pytorch_lightning.utilities.parsing.AttributeDict*, **args*, ***kwargs*)

Bases: *RideMixin*

Base-class for Ride datasets.

If no dataset is specified otherwise, this mixin is automatically add as a base of RideModule childen.

User-specified datasets must inherit from this class, and specify the following: - *self.input_shape*: Union[int, Sequence[int], Sequence[Sequence[int]]] - *self.output_shape*: Union[int, Sequence[int], Sequence[Sequence[int]]]

and either the functions: - *train_dataloader*: Callable[[Any], DataLoader] - *val_dataloader*: Callable[[Any], DataLoader] - *test_dataloader*: Callable[[Any], DataLoader]

or: - *self.datamodule*, which has *train_dataloader*, *val_dataloader*, and *test_dataloader* attributes.

input_shape: DataShape

output_shape: DataShape

validate_attributes()

static configs() → *Configs*

train_dataloader(**args: Any*, ***kwargs: Any*) → torch.utils.data.DataLoader

The train dataloader

val_dataloader(**args: Any*, ***kwargs: Any*) → Union[torch.utils.data.DataLoader, List[torch.utils.data.DataLoader]]

The val dataloader

test_dataloader(**args: Any*, ***kwargs: Any*) → Union[torch.utils.data.DataLoader, List[torch.utils.data.DataLoader]]

The test dataloader

class ride.core.**RideClassificationDataset**(*hparams: pytorch_lightning.utilities.parsing.AttributeDict*, **args*, ***kwargs*)

Bases: *RideDataset*

Base-class for Ride classification datasets.

If no dataset is specified otherwise, this mixin is automatically add as a base of RideModule childen.

User-specified datasets must inherit from this class, and specify the following: - *self.input_shape*: Union[int, Sequence[int], Sequence[Sequence[int]]] - *self.output_shape*: Union[int, Sequence[int], Sequence[Sequence[int]]] - *self.classes*: List[str]

and either the functions: - *train_dataloader*: Callable[[Any], DataLoader] - *val_dataloader*: Callable[[Any], DataLoader] - *test_dataloader*: Callable[[Any], DataLoader]

or: - *self.datamodule*, which has *train_dataloader*, *val_dataloader*, and *test_dataloader* attributes.

property num_classes: int

classes: List[str]

static configs() → *Configs*

validate_attributes()

metrics_epoch(*preds: torch.Tensor*, *targets: torch.Tensor*, *prefix: str = None*, **args*, ***kwargs*)

ride.feature_extraction

Module Contents

Classes

FeatureExtractable	Adds feature extraction capabilities to model

Attributes

logger

ride.feature_extraction.**logger**

class ride.feature_extraction.**FeatureExtractable**(*hparams: pytorch_lightning.utilities.parsing.AttributeDict*, **args*, ***kwargs*)

Bases: *ride.core.RideMixin*

Adds feature extraction capabilities to model

hparams: Ellipsis

static configs() → *ride.core.Configs*

validate_attributes()

on_init_end(*hparams*, **args*, ***kwargs*)

metrics_epoch(*preds: torch.Tensor*, *targets: torch.Tensor*, *prefix: str = None*, *clear_extracted_features=True*, **args*, ***kwargs*) → ride.metrics.MetricDict

ride.feature_visualisation

Module Contents

Classes

FeatureVisualisable	Adds feature visualisation capabilities to model

Functions

scatter_plot(features[, labels, classes])	

Attributes

logger	

ride.feature_visualisation.**logger**

class ride.feature_visualisation.**FeatureVisualisable**(*hparams*, **args*, ***kwargs*)

Bases: *ride.feature_extraction.FeatureExtractable*

Adds feature visualisation capabilities to model

hparams: Ellipsis

static configs() → *ride.core.Configs*

validate_attributes()

metrics_epoch(*preds: torch.Tensor*, *targets: torch.Tensor*, *prefix: str = None*, **args*, ***kwargs*) → ride.metrics.FigureDict

ride.feature_visualisation.**scatter_plot**(*features: numpy.array*, *labels: numpy.array = None*, *classes: List[str] = None*)

ride.finetune

Module Contents

Classes

Finetunable	Adds finetune capabilities to model

Functions

load_model_weights(file, hparams_passed, model_state_key)	
try_pyth_load(file, model_state_key)	
try_pickle_load(file)	

Attributes

logger

ride.finetune.**logger**

class ride.finetune.**Finetunable**(*hparams: pytorch_lightning.utilities.parsing.AttributeDict*, **args*, ***kwargs*)

Bases: *ride.unfreeze.Unfreezable*

Adds finetune capabilities to model

This module is automatically added when RideModule is inherited

hparams: Ellipsis

static configs() → *ride.core.Configs*

validate_attributes()

map_loaded_weights(*file*, *loaded_state_dict*)

on_init_end(*hparams*, **args*, ***kwargs*)

ride.finetune.**load_model_weights**(*file: str*, *hparams_passed*, *model_state_key*)

ride.finetune.**try_pyth_load**(*file*, *model_state_key*)

ride.finetune.**try_pickle_load**(*file*)

ride.hparamsearch

Module Contents

Classes

Hparamsearch

Attributes

logger

ride.hparamsearch.**logger**

class ride.hparamsearch.**Hparamsearch**(*Module: Type[*ride.core.RideModule*]*)

configs() → *ride.core.Configs*

__call__(*args: pytorch_lightning.utilities.parsing.AttributeDict*)

run(*args: pytorch_lightning.utilities.parsing.AttributeDict*)

Run hyperparameter search using the *tune.schedulers.ASHAScheduler*

Parameters
args (*AttributeDict*) – Arguments

Side-effects:
Saves logs to *TUNE_LOGS_PATH / args.id*

static dump(*hparams: dict*, *identifier: str*, *extention='yaml'*) → str

Dumps haparams to TUNE_LOGS_PATH / identifier / "best_hparams.json"

static load(*path: Union[pathlib.Path, str]*, *old_args=AttributeDict()*, *Cls: Type[*ride.core.RideModule*] = None*, *auto_scale_lr=False*) → pytorch_lightning.utilities.parsing.AttributeDict

Loads hparams from path

Parameters

- **path** (*Union[Path, str]*) – Path to jsonfile containing hparams
- **old_args** (*Optional[AttributeDict]*) – The AttributeDict to be updated with the new hparams
- **cls** (*Optional[RideModule]*) – A class whole hyperparameters can be used to select the relevant hparams to take

Returns
AttributeDict with updated hyperparameters

Return type
AttributeDict

ride.info

Module Contents

ride.info.**__version__** = **'0.7.3'**

ride.info.**__author__** = **'Lukas Hedegaard'**

ride.info.**__author_email__** = **'lukasxhedegaard@gmail.com'**

ride.info.**__license__** = **'Apache-2.0'**

ride.info.**__copyright__**

ride.info.**__homepage__** = **'https://github.com/LukasHedegaard/ride'**

ride.info.**__docs__** = **'Training wheels, side rails, and helicopter parent for your Deep Learning projects using Pytorch'**

ride.lifecycle

Module Contents

Classes

Lifecycle	Adds train, val, and test lifecycle methods with cross_entropy loss

Functions

prefix_keys(→ Dict)
detach_to_cpu(x)
cat_steps(steps)

Attributes

loss_names
logger

ride.lifecycle.**loss_names**

ride.lifecycle.**logger**

class ride.lifecycle.**Lifecycle**(*hparams=None*, **args*, ***kwargs*)

Bases: *ride.metrics.MetricMixin*

Adds train, val, and test lifecycle methods with cross_entropy loss

During its *traning_epoch_end(epoch)* lifecycle method, it will call *on_traning_epoch_end* for all superclasses of its child class

hparams: Ellipsis

forward: Callable[[torch.Tensor], torch.Tensor]

_epoch: int

classmethod _metrics()

validate_attributes()

static configs() → *ride.core.Configs*

metrics_step(*preds: torch.Tensor*, *targets: torch.Tensor*, ***kwargs*) → ride.metrics.MetricDict

common_step(*pred*, *target*, *prefix='train/'*, *log=False*)

common_epoch_end(*step_outputs*, *prefix='train/'*, *exclude_keys={'pred', 'target'}*)

preprocess_batch(*batch*)

training_step(*batch*, *batch_idx=None*)

training_epoch_end(*step_outputs*)

validation_step(*batch*, *batch_idx=None*)

validation_epoch_end(*step_outputs*)

test_step(*batch*, *batch_idx=None*)

test_epoch_end(*step_outputs*)

ride.lifecycle.**prefix_keys**(*prefix: str*, *dictionary: Dict*) → Dict

ride.lifecycle.**detach_to_cpu**(*x: Union[torch.Tensor, Sequence[torch.Tensor], Dict[Any, torch.Tensor]]*)

ride.lifecycle.**cat_steps**(*steps: Sequence[Union[torch.Tensor, Sequence[torch.Tensor], Dict[Any, torch.Tensor]]]*)

ride.logging

Module Contents

Classes

ResultsLogger	

Functions

singleton_experiment_logger(→ ExperimentLoggerCreator)	
fig2img(fig)	Convert a Matplotlib figure to a PIL Image and return it
add_experiment_logger(...)	
get_log_dir(module)	
log_figures(module, d)	

Attributes

logger
ExperimentLogger
ExperimentLoggerCreator
experiment_logger
StepOutputs

ride.logging.**logger**

ride.logging.**ExperimentLogger**

ride.logging.**ExperimentLoggerCreator**

ride.logging.**singleton_experiment_logger**() → ExperimentLoggerCreator

ride.logging.**experiment_logger**

ride.logging.**fig2img**(*fig*)

Convert a Matplotlib figure to a PIL Image and return it

ride.logging.**add_experiment_logger**(*prev_logger: pytorch_lightning.loggers.LightningLoggerBase*, *new_logger: pytorch_lightning.loggers.LightningLoggerBase*) → pytorch_lightning.loggers.LoggerCollection

ride.logging.**get_log_dir**(*module: pytorch_lightning.LightningModule*)

ride.logging.**log_figures**(*module: pytorch_lightning.LightningModule*, *d: ride.metrics.FigureDict*)

class ride.logging.**ResultsLogger**(*prefix='test'*, *save_to: str = None*)

Bases: pytorch_lightning.loggers.LightningLoggerBase

property experiment

property save_dir: Optional[str]

property name

property version

_fix_name_perfix(*s: str*, *replace='test/'*) → str

log_hyperparams(*params*)

log_metrics(*metrics: Dict*, *step*)

log_figure(*tag: str*, *fig: matplotlib.figure.Figure*)

finalize(*status*)

ride.logging.**StepOutputs**

ride.main

main.py
Main entry-point for the Ride main wrapper. For logging to be formatted consistently, this file should be imported prior to other libraries

isort:skip_file

Module Contents

Classes

Main	Complete main programme for the lifecycle of a machine learning project

Functions

patched_getLogger([name])	
hprint(msg)	Message header print
dprint(d)	
make_save_results(→ Callable[[str, Any], None])	

Attributes

original_getLogger
logger

ride.main.**original_getLogger**

ride.main.**patched_getLogger**(*name: str = None*)

ride.main.**logger**

class ride.main.**Main**(*Module: Type[*ride.core.RideModule*]*)
Complete main programme for the lifecycle of a machine learning project

Usage:
Main(YourRideModule).argparse()

argparse(*args: List[str] = None*, *run=True*)

main(*args: pytorch_lightning.utilities.parsing.AttributeDict*)

ride.main.**hprint**(*msg: str*)

Message header print

Parameters
msg (*str*) – Message to be printed

ride.main.**dprint**(*d: dict*)

ride.main.**make_save_results**(*root_path: str*, *verbose=True*) → Callable[[str, Any], None]

ride.metrics

Module Contents

Classes

OptimisationDirection	Generic enumeration.
MetricMixin	Abstract base class for Ride modules
MeanAveragePrecisionMetric	Mean Average Precision (mAP) metric
FlopsMetric	Computes Floating Point Operations (FLOPs) for the model and adds it as metric
FlopsWeightedAccuracyMetric	Computes acc * (flops / target_gflops) ** (-0.07)

Functions

sort_out_figures(→ Tuple[MetricDict, FigureDict])	
MetricSelector(→ MetricMixin)	
TopKAccuracyMetric(→ MetricMixin)	
topks_correct(→ List[torch.Tensor])	Given the predictions, labels, and a list of top-k values, compute the
topk_errors(preds, labels, ks)	Computes the top-k error for each k.
topk_accuracies(preds, labels, ks)	Computes the top-k accuracy for each k.
flops(model)	Compute the Floating Point Operations per Second for the model
params_count(model)	Compute the number of parameters.
make_confusion_matrix(→ matplotlib.figure.Figure)	

Attributes

ExtendedMetricDict
MetricDict
FigureDict
StepOutputs
logger

ride.metrics.**ExtendedMetricDict**

ride.metrics.**MetricDict**

ride.metrics.**FigureDict**

ride.metrics.**StepOutputs**

ride.metrics.**logger**

ride.metrics.**sort_out_figures**(*d: ExtendedMetricDict*) → Tuple[MetricDict, FigureDict]

class ride.metrics.**OptimisationDirection**

Bases: enum.Enum

Generic enumeration.

Derive from this class to define new enumerations.

MIN = 'min'

MAX = 'max'

class ride.metrics.**MetricMixin**(*hparams: pytorch_lightning.utilities.parsing.AttributeDict*, **args*, ***kwargs*)

Bases: *ride.core.RideMixin*

Abstract base class for Ride modules

classmethod __init_subclass__()

classmethod metrics() → Dict[str, str]

classmethod metric_names() → List[str]

metrics_step(**args*, ***kwargs*) → MetricDict

metrics_epoch(*preds: torch.Tensor*, *targets: torch.Tensor*, *prefix: str = ''*, **args*, ***kwargs*) → MetricDict

collect_metrics(*preds: torch.Tensor*, *targets: torch.Tensor*) → MetricDict

collect_epoch_metrics(*preds: torch.Tensor*, *targets: torch.Tensor*, *prefix: str = None*) → ExtendedMetricDict

ride.metrics.**MetricSelector**(*mapping: Dict[str, Union[MetricMixin, Iterable[MetricMixin]]] = None*, *default_config: str = ''*, ***kwargs: Union[MetricMixin, Iterable[MetricMixin]]*) → *MetricMixin*

class ride.metrics.**MeanAveragePrecisionMetric**(*hparams: pytorch_lightning.utilities.parsing.AttributeDict*, **args*, ***kwargs*)

Bases: *MetricMixin*

Mean Average Precision (mAP) metric

validate_attributes()

_compute_mean_average_precision(*preds*, *targets*)

classmethod _metrics()

metrics_step(*preds: torch.Tensor*, *targets: torch.Tensor*, **args*, ***kwargs*) → MetricDict

metrics_epoch(*preds: torch.Tensor*, *targets: torch.Tensor*, **args*, ***kwargs*) → MetricDict

ride.metrics.**TopKAccuracyMetric**(**Ks*) → *MetricMixin*

class ride.metrics.**FlopsMetric**(*hparams: pytorch_lightning.utilities.parsing.AttributeDict*, **args*, ***kwargs*)

Bases: *MetricMixin*

Computes Floating Point Operations (FLOPs) for the model and adds it as metric

classmethod _metrics()

on_init_end(**args*, ***kwargs*)

metrics_step(*preds: torch.Tensor*, *targets: torch.Tensor*, ***kwargs*) → MetricDict

class ride.metrics.**FlopsWeightedAccuracyMetric**(*hparams: pytorch_lightning.utilities.parsing.AttributeDict*, **args*, ***kwargs*)

Bases: *FlopsMetric*

Computes acc * (flops / target_gflops) ** (-0.07)

classmethod _metrics()

validate_attributes()

static configs() → *ride.core.Configs*

metrics_step(*preds: torch.Tensor*, *targets: torch.Tensor*, ***kwargs*) → MetricDict

ride.metrics.**topks_correct**(*preds: torch.Tensor*, *labels: torch.Tensor*, *ks: List[int]*) → List[torch.Tensor]

Given the predictions, labels, and a list of top-k values, compute the number of correct predictions for each top-k value.

Parameters

- **preds** (*array*) – array of predictions. Dimension is batchsize N x ClassNum.
- **labels** (*array*) – array of labels. Dimension is batchsize N.
- **ks** (*list*) – list of top-k values. For example, ks = [1, 5] correspods to top-1 and top-5.

Returns

list of numbers, where the *i*-th entry
corresponds to the number of top-*ks[i]* correct predictions.

Return type
topks_correct (list)

ride.metrics.**topk_errors**(*preds: torch.Tensor*, *labels: torch.Tensor*, *ks: List[int]*)

Computes the top-k error for each k. :param preds: array of predictions. Dimension is N. :type preds: array :param labels: array of labels. Dimension is N. :type labels: array :param ks: list of ks to calculate the top accuracies. :type ks: list

ride.metrics.**topk_accuracies**(*preds: torch.Tensor*, *labels: torch.Tensor*, *ks: List[int]*)

Computes the top-k accuracy for each k. :param preds: array of predictions. Dimension is N. :type preds: array :param labels: array of labels. Dimension is N. :type labels: array :param ks: list of ks to calculate the top accuracies. :type ks: list

ride.metrics.**flops**(*model: torch.nn.Module*)

Compute the Floating Point Operations per Second for the model

ride.metrics.**params_count**(*model: torch.nn.Module*)

Compute the number of parameters. :param model: model to count the number of parameters. :type model: model

ride.metrics.**make_confusion_matrix**(*preds: torch.Tensor*, *targets: torch.Tensor*, *classes: List[str]*) → matplotlib.figure.Figure

ride.optimizers

Modules adding optimizers

Module Contents

Classes

SgdOptimizer	Abstract base-class for Optimizer mixins
AdamWOptimizer	Abstract base-class for Optimizer mixins
SgdReduceLrOnPlateauOptimizer	Abstract base-class for Optimizer mixins
AdamWReduceLrOnPlateauOptimizer	Abstract base-class for Optimizer mixins
SgdCyclicLrOptimizer	Abstract base-class for Optimizer mixins
AdamWCyclicLrOptimizer	Abstract base-class for Optimizer mixins
SgdOneCycleOptimizer	Abstract base-class for Optimizer mixins
AdamWOneCycleOptimizer	Abstract base-class for Optimizer mixins
SgdMultiStepLR	Abstract base-class for Optimizer mixins
AdamWMultiStepLR	Abstract base-class for Optimizer mixins

Functions

discounted_steps_per_epoch(base_steps, num_gpus, ...)	
discriminative_lr_and_params(model, lr, ...)	

ride.optimizers.**discounted_steps_per_epoch**(*base_steps: int*, *num_gpus: int*, *accumulate_grad_batches: int*)

class ride.optimizers.**SgdOptimizer**(*hparams: pytorch_lightning.utilities.parsing.AttributeDict*, **args*, ***kwargs*)

Bases: *ride.core.OptimizerMixin*

Abstract base-class for Optimizer mixins

hparams: Ellipsis

parameters: Callable

validate_attributes()

static configs() → *ride.core.Configs*

configure_optimizers()

class ride.optimizers.**AdamWOptimizer**(*hparams: pytorch_lightning.utilities.parsing.AttributeDict*, **args*, ***kwargs*)

Bases: *ride.core.OptimizerMixin*

Abstract base-class for Optimizer mixins

hparams: Ellipsis

parameters: Callable

validate_attributes()

static configs() → *ride.core.Configs*

configure_optimizers()

class ride.optimizers.**SgdReduceLrOnPlateauOptimizer**(*hparams: pytorch_lightning.utilities.parsing.AttributeDict*, **args*, ***kwargs*)

Bases: *ride.core.OptimizerMixin*

Abstract base-class for Optimizer mixins

hparams: Ellipsis

parameters: Callable

validate_attributes()

static configs() → *ride.core.Configs*

configure_optimizers()

class ride.optimizers.**AdamWReduceLrOnPlateauOptimizer**(*hparams: pytorch_lightning.utilities.parsing.AttributeDict*, **args*, ***kwargs*)

Bases: *ride.core.OptimizerMixin*

Abstract base-class for Optimizer mixins

hparams: Ellipsis

parameters: Callable

validate_attributes()

static configs() → *ride.core.Configs*

configure_optimizers()

class ride.optimizers.**SgdCyclicLrOptimizer**(*hparams: pytorch_lightning.utilities.parsing.AttributeDict*, **args*, ***kwargs*)

Bases: *ride.core.OptimizerMixin*

Abstract base-class for Optimizer mixins

hparams: Ellipsis

parameters: Callable

train_dataloader: Callable

validate_attributes()

static configs() → *ride.core.Configs*

configure_optimizers()

class ride.optimizers.**AdamWCyclicLrOptimizer**(*hparams: pytorch_lightning.utilities.parsing.AttributeDict*, **args*, ***kwargs*)

Bases: *ride.core.OptimizerMixin*

Abstract base-class for Optimizer mixins

hparams: Ellipsis

parameters: Callable

train_dataloader: Callable

validate_attributes()

static configs() → *ride.core.Configs*

configure_optimizers()

class ride.optimizers.**SgdOneCycleOptimizer**(*hparams: pytorch_lightning.utilities.parsing.AttributeDict*, **args*, ***kwargs*)

Bases: *ride.core.OptimizerMixin*

Abstract base-class for Optimizer mixins

hparams: Ellipsis

parameters: Callable

train_dataloader: Callable

validate_attributes()

static configs() → *ride.core.Configs*

configure_optimizers()

class ride.optimizers.**AdamWOneCycleOptimizer**(*hparams: pytorch_lightning.utilities.parsing.AttributeDict*, **args*, ***kwargs*)

Bases: *ride.core.OptimizerMixin*

Abstract base-class for Optimizer mixins

hparams: Ellipsis

parameters: Callable

train_dataloader: Callable

validate_attributes()

static configs() → *ride.core.Configs*

configure_optimizers()

class ride.optimizers.**SgdMultiStepLR**(*hparams: pytorch_lightning.utilities.parsing.AttributeDict*, **args*, ***kwargs*)

Bases: *ride.core.OptimizerMixin*

Abstract base-class for Optimizer mixins

hparams: Ellipsis

parameters: Callable

train_dataloader: Callable

validate_attributes()

static configs() → *ride.core.Configs*

configure_optimizers()

class ride.optimizers.**AdamWMultiStepLR**(*hparams: pytorch_lightning.utilities.parsing.AttributeDict*, **args*, ***kwargs*)

Bases: *ride.core.OptimizerMixin*

Abstract base-class for Optimizer mixins

hparams: Ellipsis

parameters: Callable

train_dataloader: Callable

validate_attributes()

static configs() → *ride.core.Configs*

configure_optimizers()

ride.optimizers.**discriminative_lr_and_params**(*model: torch.nn.Module*, *lr: float*, *discriminative_lr_fraction: float*)

ride.runner

Module Contents

Classes

Runner

Functions

is_runnable(cls)

Attributes

EvalutationResults
logger

ride.runner.**EvalutationResults**

ride.runner.**logger**

ride.runner.**is_runnable**(*cls*)

class ride.runner.**Runner**(*Module: Type[*ride.core.RideModule*]*)

trained_model: *ride.core.RideModule*

train(*args: pytorch_lightning.utilities.parsing.AttributeDict*, *trainer_callbacks: List[Callable] = []*, *tune_checkpoint_dir: str = None*, *experiment_logger: ride.logging.ExperimentLoggerCreator = experiment_logger*) → *ride.core.RideModule*

evaluate(*args: pytorch_lightning.utilities.parsing.AttributeDict*, *mode='val'*) → EvalutationResults

validate(*args: pytorch_lightning.utilities.parsing.AttributeDict*) → EvalutationResults

test(*args: pytorch_lightning.utilities.parsing.AttributeDict*) → EvalutationResults

train_and_val(*args: pytorch_lightning.utilities.parsing.AttributeDict*, *trainer_callbacks: List[Callable] = []*, *tune_checkpoint_dir: str = None*, *experiment_logger: ride.logging.ExperimentLoggerCreator = experiment_logger*) → EvalutationResults

static static_train_and_val(*Module: Type[ride.core.RideModule]*, *args: pytorch_lightning.utilities.parsing.AttributeDict*, *trainer_callbacks: List[Callable] = []*, *tune_checkpoint_dir: str = None*, *experiment_logger: ride.logging.ExperimentLoggerCreator = experiment_logger*) → EvalutationResults

profile_model(*args: pytorch_lightning.utilities.parsing.AttributeDict*, *num_runs: int = 100*) → Dict[str, Any]

abstract find_learning_rate()

abstract find_batch_size()

ride.unfreeze

Module Contents

Classes

Unfreezable	Abstract base-class for Ride mixins

Functions

freeze_layers_except_names(parent_module, ...)	
get_modules_to_unfreeze(→ Sequence[Tuple[str, ...])	
unfreeze_from_end(layers, num_layers_from_end[, ...])	
linear_unfreeze_schedule(→ Dict[int, int])	

Attributes

logger	

ride.unfreeze.**logger**

class ride.unfreeze.**Unfreezable**(*hparams: pytorch_lightning.utilities.parsing.AttributeDict*, **args*, ***kwargs*)

Bases: *ride.core.RideMixin*

Abstract base-class for Ride mixins

hparams: Ellipsis

validate_attributes()

static configs() → *ride.core.Configs*

on_init_end(*hparams*, *layers_to_unfreeze: Sequence[Tuple[str, torch.nn.Module]] = None*, *names_to_unfreeze: Sequence[str] = None*, **args*, ***kwargs*)

on_traning_epoch_start(*epoch: int*)

ride.unfreeze.**freeze_layers_except_names**(*parent_module: torch.nn.Module*, *names_to_unfreeze: Sequence[str]*)

ride.unfreeze.**get_modules_to_unfreeze**(*parent_module: torch.nn.Module*, *name_must_include=''*) → Sequence[Tuple[str, torch.nn.Module]]

ride.unfreeze.**unfreeze_from_end**(*layers: Sequence[Tuple[str, torch.nn.Module]]*, *num_layers_from_end: int*, *freeze_others=False*)

ride.unfreeze.**linear_unfreeze_schedule**(*initial_epoch: int*, *total_layers: int*, *step_size: int = 1*, *init_layers: int = 0*, *max_layers: int = -1*, *epoch_step: int = 1*) → Dict[int, int]

5.1.3 Package Contents

Classes

Main	Complete main programme for the lifecycle of a machine learning project
Configs	Configs module for holding project configurations.
RideClassificationDataset	Base-class for Ride classification datasets.
RideDataset	Base-class for Ride datasets.
RideModule	Base-class for modules using the Ride ecosystem.
Finetunable	Adds finetune capabilities to model
Hparamsearch	
Lifecycle	Adds train, val, and test lifecycle methods with cross_entropy loss
FlopsMetric	Computes Floating Point Operations (FLOPs) for the model and adds it as metric
FlopsWeightedAccuracyMetric	Computes acc * (flops / target_gflops) ** (-0.07)
MeanAveragePrecisionMetric	Mean Average Precision (mAP) metric
AdamWOneCycleOptimizer	Abstract base-class for Optimizer mixins
AdamWOptimizer	Abstract base-class for Optimizer mixins
SgdOneCycleOptimizer	Abstract base-class for Optimizer mixins
SgdOptimizer	Abstract base-class for Optimizer mixins

Functions

getLogger(name[, log_once])	
MetricSelector(→ MetricMixin)	
TopKAccuracyMetric(→ MetricMixin)	

class ride.**Main**(*Module: Type[*ride.core.RideModule*]*)

Complete main programme for the lifecycle of a machine learning project

Usage:
Main(YourRideModule).argparse()

argparse(*args: List[str] = None*, *run=True*)

main(*args: pytorch_lightning.utilities.parsing.AttributeDict*)

class ride.**Configs**

Bases: `corider.Configs`

Configs module for holding project configurations.

This is a wrapper of the Configs found as a stand-alone package in https://github.com/LukasHedegaard/co-rider

static collect(*cls:* RideModule) → *Configs*

Collect the configs from all class bases

Returns
Aggregated configurations

Return type
Configs

default_values()

class ride.**RideClassificationDataset**(*hparams: pytorch_lightning.utilities.parsing.AttributeDict*, **args*, ***kwargs*)

Bases: *RideDataset*

Base-class for Ride classification datasets.

If no dataset is specified otherwise, this mixin is automatically add as a base of RideModule childen.

User-specified datasets must inherit from this class, and specify the following: - *self.input_shape*: Union[int, Sequence[int], Sequence[Sequence[int]]] - *self.output_shape*: Union[int, Sequence[int], Sequence[Sequence[int]]] - *self.classes*: List[str]

and either the functions: - *train_dataloader*: Callable[[Any], DataLoader] - *val_dataloader*: Callable[[Any], DataLoader] - *test_dataloader*: Callable[[Any], DataLoader]

or: - *self.datamodule*, which has *train_dataloader*, *val_dataloader*, and *test_dataloader* attributes.

property num_classes: int

classes: List[str]

static configs() → *Configs*

validate_attributes()

metrics_epoch(*preds: torch.Tensor*, *targets: torch.Tensor*, *prefix: str = None*, **args*, ***kwargs*)

class ride.**RideDataset**(*hparams: pytorch_lightning.utilities.parsing.AttributeDict*, **args*, ***kwargs*)

Bases: RideMixin

Base-class for Ride datasets.

If no dataset is specified otherwise, this mixin is automatically add as a base of RideModule childen.

User-specified datasets must inherit from this class, and specify the following: - *self.input_shape*: Union[int, Sequence[int], Sequence[Sequence[int]]] - *self.output_shape*: Union[int, Sequence[int], Sequence[Sequence[int]]]

and either the functions: - *train_dataloader*: Callable[[Any], DataLoader] - *val_dataloader*: Callable[[Any], DataLoader] - *test_dataloader*: Callable[[Any], DataLoader]

or: - *self.datamodule*, which has *train_dataloader*, *val_dataloader*, and *test_dataloader* attributes.

input_shape: DataShape

output_shape: DataShape

validate_attributes()

static configs() → *Configs*

train_dataloader(**args: Any*, ***kwargs: Any*) → torch.utils.data.DataLoader

The train dataloader

val_dataloader(**args: Any*, ***kwargs: Any*) → Union[torch.utils.data.DataLoader, List[torch.utils.data.DataLoader]]

The val dataloader

test_dataloader(**args: Any*, ***kwargs: Any*) → Union[torch.utils.data.DataLoader, List[torch.utils.data.DataLoader]]

The test dataloader

class ride.**RideModule**

Base-class for modules using the Ride ecosystem.

This module should be inherited as the highest-priority parent (first in sequence).

Example:

```
class MyModule(ride.RideModule, ride.SgdOneCycleOptimizer):
    def __init__(self, hparams):
        ...
```

It handles proper initialisation of *RideMixin* parents and adds automatic attribute validation.

If *pytorch_lightning.LightningModule* is omitted as lowest-priority parent, *RideModule* will automatically add it.

If *training_step*, *validation_step*, and *test_step* methods are not found, the *ride.Lifecycle* will be automatically mixed in by this module.

property hparams: pytorch_lightning.utilities.parsing.AttributeDict

classmethod __init_subclass__()

classmethod with_dataset(*ds:* RideDataset)

ride.**getLogger**(*name*, *log_once=False*)

class ride.**Finetunable**(*hparams: pytorch_lightning.utilities.parsing.AttributeDict*, **args*, ***kwargs*)

Bases: *ride.unfreeze.Unfreezable*

Adds finetune capabilities to model

This module is automatically added when RideModule is inherited

hparams: Ellipsis

static configs() → *ride.core.Configs*

validate_attributes()

map_loaded_weights(*file*, *loaded_state_dict*)

on_init_end(*hparams*, **args*, ***kwargs*)

class ride.**Hparamsearch**(*Module: Type[*ride.core.RideModule*]*)

configs() → *ride.core.Configs*

__call__(*args: pytorch_lightning.utilities.parsing.AttributeDict*)

run(*args: pytorch_lightning.utilities.parsing.AttributeDict*)

Run hyperparameter search using the *tune.schedulers.ASHAScheduler*

Parameters
args (*AttributeDict*) – Arguments

Side-effects:
Saves logs to *TUNE_LOGS_PATH / args.id*

static dump(*hparams: dict*, *identifier: str*, *extention='yaml'*) → str

Dumps haparams to TUNE_LOGS_PATH / identifier / "best_hparams.json"

static load(*path: Union[pathlib.Path, str]*, *old_args=AttributeDict()*, *Cls: Type[*ride.core.RideModule*] = None*, *auto_scale_lr=False*) → pytorch_lightning.utilities.parsing.AttributeDict

Loads hparams from path

Parameters

- **path** (*Union[Path, str]*) – Path to jsonfile containing hparams
- **old_args** (*Optional[AttributeDict]*) – The AttributeDict to be updated with the new hparams
- **cls** (*Optional[*RideModule*]*) – A class whole hyperparameters can be used to select the relevant hparams to take

Returns
AttributeDict with updated hyperparameters

Return type
AttributeDict

class ride.**Lifecycle**(*hparams=None*, **args*, ***kwargs*)

Bases: *ride.metrics.MetricMixin*

Adds train, val, and test lifecycle methods with cross_entropy loss

During its *traning_epoch_end(epoch)* lifecycle method, it will call *on_traning_epoch_end* for all superclasses of its child class

hparams: Ellipsis

forward: Callable[[torch.Tensor], torch.Tensor]

_epoch: int

classmethod _metrics()

validate_attributes()

static configs() → *ride.core.Configs*

metrics_step(*preds: torch.Tensor*, *targets: torch.Tensor*, ***kwargs*) → ride.metrics.MetricDict

common_step(*pred*, *target*, *prefix='train/'*, *log=False*)

common_epoch_end(*step_outputs*, *prefix='train/'*, *exclude_keys={'pred', 'target'}*)

preprocess_batch(*batch*)

training_step(*batch*, *batch_idx=None*)

training_epoch_end(*step_outputs*)

validation_step(*batch*, *batch_idx=None*)

validation_epoch_end(*step_outputs*)

test_step(*batch*, *batch_idx=None*)

test_epoch_end(*step_outputs*)

class ride.**FlopsMetric**(*hparams: pytorch_lightning.utilities.parsing.AttributeDict*, **args*, ***kwargs*)

Bases: MetricMixin

Computes Floating Point Operations (FLOPs) for the model and adds it as metric

classmethod _metrics()

on_init_end(**args*, ***kwargs*)

metrics_step(*preds: torch.Tensor*, *targets: torch.Tensor*, ***kwargs*) → MetricDict

class ride.**FlopsWeightedAccuracyMetric**(*hparams: pytorch_lightning.utilities.parsing.AttributeDict*, **args*, ***kwargs*)

Bases: *FlopsMetric*

Computes acc * (flops / target_gflops) ** (-0.07)

classmethod _metrics()

validate_attributes()

static configs() → *ride.core.Configs*

metrics_step(*preds: torch.Tensor*, *targets: torch.Tensor*, ***kwargs*) → MetricDict

class ride.**MeanAveragePrecisionMetric**(*hparams: pytorch_lightning.utilities.parsing.AttributeDict*, **args*, ***kwargs*)

Bases: `MetricMixin`

Mean Average Precision (mAP) metric

validate_attributes()

_compute_mean_average_precision(*preds*, *targets*)

classmethod _metrics()

metrics_step(*preds: torch.Tensor*, *targets: torch.Tensor*, **args*, ***kwargs*) → MetricDict

metrics_epoch(*preds: torch.Tensor*, *targets: torch.Tensor*, **args*, ***kwargs*) → MetricDict

ride.**MetricSelector**(*mapping: Dict[str, Union[*MetricMixin*, Iterable[*MetricMixin*]]] = None*, *default_config: str = ''*, ***kwargs: Union[*MetricMixin*, Iterable[*MetricMixin*]]*) → *MetricMixin*

ride.**TopKAccuracyMetric**(**Ks*) → *MetricMixin*

class ride.**AdamWOneCycleOptimizer**(*hparams: pytorch_lightning.utilities.parsing.AttributeDict*, **args*, ***kwargs*)

Bases: *ride.core.OptimizerMixin*

Abstract base-class for Optimizer mixins

hparams: Ellipsis

parameters: Callable

train_dataloader: Callable

validate_attributes()

static configs() → *ride.core.Configs*

configure_optimizers()

class ride.**AdamWOptimizer**(*hparams: pytorch_lightning.utilities.parsing.AttributeDict*, **args*, ***kwargs*)

Bases: *ride.core.OptimizerMixin*

Abstract base-class for Optimizer mixins

hparams: Ellipsis

parameters: Callable

validate_attributes()

static configs() → *ride.core.Configs*

configure_optimizers()

class ride.**SgdOneCycleOptimizer**(*hparams: pytorch_lightning.utilities.parsing.AttributeDict*, **args*, ***kwargs*)

Bases: *ride.core.OptimizerMixin*

Abstract base-class for Optimizer mixins

hparams: Ellipsis

parameters: Callable

train_dataloader: Callable

validate_attributes()

static configs() → *ride.core.Configs*

configure_optimizers()

class ride.**SgdOptimizer**(*hparams: pytorch_lightning.utilities.parsing.AttributeDict*, **args*, ***kwargs*)

Bases: *ride.core.OptimizerMixin*

Abstract base-class for Optimizer mixins

hparams: Ellipsis

parameters: Callable

validate_attributes()

static configs() → *ride.core.Configs*

configure_optimizers()

CHAPTER

SIX

DEVELOPMENT SETUP

Clone repository:

```
git clone https://github.com/LukasHedegaard/ride.git
cd ride
```

Install extended dependencies:

```
pip install -e .[build,dev,docs]
```

Run tests:

```
make test
```

Build docs

```
cd docs
make html
```

Build and publish to TestPyPI:

```
make clean
make testbuild
make testpublish
```

Build and publish to PyPI:

```
make clean
make build
make publish
```

CHAPTER

SEVEN

CHANGELOG

All notable changes to this project will be documented in this file.

The format is based on Keep a Changelog, and this project adheres to Semantic Versioning.

7.1 [Unreleased]

7.2 [0.7.3] - 2023-05-17

7.2.1 [0.7.3] - Fixed

- Compatibility with newer PyTorch Benchmark version.

7.3 [0.7.2] - 2022-06-03

7.3.1 [0.7.2] - Added

- Version for protobuf during build.
- Conditional install of redis on win platforms

7.4 [0.7.1] - 2022-03-18

7.4.1 [0.7.1] - Fixed

- Device transfer in benchmark.

7.5 [0.7.0] - 2022-03-18

7.5.1 [0.7.0] - Added

- Defensive fallback for FLOPs measurement.
- Add MultiStepLR optimizers.

7.5.2 [0.7.0] - Changed

- Profiling to use `pytorch_benchmark` package.

7.5.3 [0.7.0] - Fixed

- WandB logger log_dir extraction.

7.6 [0.6.1] - 2022-02-07

7.6.1 [0.6.1] - Changed

- Profile only warms up on first inference.

7.7 [0.6.0] - 2022-01-27

7.7.1 [0.6.0] - Added

- Memory profiling.

7.7.2 [0.6.0] - Fixed

- Tune DeprecationWarning.

7.8 [0.5.1] - 2021-11-16

7.8.1 [0.5.1] - Added

- Add pred and target dict support in Lifecycle.

7.8.2 [0.5.1] - Fixed

- Avoid detaching loss in step.

7.9 [0.5.0] - 2021-11-12

7.9.1 [0.5.0] - Added

- Add preprocess_batch method to Lifecycle.
- Add option for string type in utils.name.
- Add Metric Selector.

7.9.2 [0.5.0] - Fixed

- Weight freezing during model loading.
- Fix discriminative_lr param selection for NoneType parameters.
- Fix wandb project naming during hparamsearch.
- Optimizer Schedulers take `accumulate_grad_batches` into account.

7.9.3 [0.5.0] - Changed

- Key debug statements while loading models to include both missing and unexpected keys.
- Bumped PL to version 1.4. Holding back on 1.5 due to Tune integration issues.
- Bumped Tune to version 1.8.

7.10 [0.4.6] - 2021-09-21

7.10.1 [0.4.6] - Fixed

- Update profile to use model.**call**. This enable non-`forward` executions during profiling.
- Add DefaultMethods Mixin with `warm_up` to make `warm_up` overloadable by Mixins.

7.11 [0.4.5] - 2021-09-08

7.11.1 [0.4.5] - Fixed

- Fix `warm_up` function signature.
- Requirement versions.

7.12 [0.4.4] - 2021-09-08

7.12.1 [0.4.4] - Added

- `warm_up` function that is called prior to profil .

7.12.2 [0.4.4] - Fixed

- Learning rate schedulers discounted steps.

7.13 [0.4.3] - 2021-06-03

7.13.1 [0.4.3] - Added

- Logging of layers that are unfrozen.

7.13.2 [0.4.3] - Fixed

- Cyclic learning rate schedulers now update on step.

7.14 [0.4.2] - 2021-06-02

7.14.1 [0.4.2] - Added

- Added explicit logging of model profiling results.
- Automatic assignment of hparams.num_gpus.

7.14.2 [0.4.2] - Fixed

- Finetune weight loading checks.
- Cyclic learning rate schedulers account for batch size.

7.15 [0.4.1] - 2021-05-27

7.15.1 [0.4.1] - Fixed

- Feature extraction on GPU.

7.15.2 [0.4.1] - Added

- Added explicit logging of hparams.

7.16 [0.4.0] - 2021-05-17

7.16.1 [0.4.0] - Fixed

- Pass args correctly to trainer during testing.

7.16.2 [0.4.0] - Changed

- CheckpointEveryNSteps now included in ModelCheckpoint c.f. pl==1.3.
- Import from torchmetrics instead of pl.metrics .
- Moved confusion matrix to RideClassificationDataset and updated plot.

7.16.3 [0.4.0] - Added

- Feature extraction and visualisation.
- Lifecycle and Finetuneable mixins always included via RideModule.
- Support for pytorch-lightning==1.3.
- Additional tests: Coverage is now at 92%.

7.16.4 [0.4.0] - Removed

- Support for nested inheritance of RideModule.
- Support for pytorch-lightning==1.2.

7.17 [0.3.2] - 2021-04-15

7.17.1 [0.3.2] - Fixed

- Project dependencies: removed click and added psutil to requirements.
- Logging: Save stdout and stderr to run.log.

7.17.2 [0.3.2] - Changed

- Logged results names. Flattened folder structure and streamlines names.

7.17.3 [0.3.2] - Added

- Docstrings to remaining core classes.
- Tests that logged results exists.

7.18 [0.3.1] - 2021-03-24

7.18.1 [0.3.1] - Added

- Add support for namedtuples in dataset `input_shape` and `output_shape`.
- Add tests for test_enemble.
- Expose more classes via `from ride import XXX`.
- Fix import-error in hparamsearch.
- Fix issues in metrics and add tests.
- Remove unused cache module.

7.18.2 [0.3.1] - Change

- Renamed `Dataset` to `RideDataset`.

7.19 [0.3.0] - 2021-03-24

7.19.1 [0.3.0] - Added

- Documentation for getting started, the Ride API, and a general API reference.
- Automatic import of `SgdOptimizer`.

7.19.2 [0.3.0] - Change

- Renamed `Dataset` to `RideDataset`.

7.20 [0.2.0] - 2021-03-23

7.20.1 [0.2.0] - Added

- Initial publicly available implementation of the library.

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- search

PYTHON MODULE INDEX

r

INDEX

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W