22HLT01 QUMPHY

D6

Report containing guidelines to support the adoption of the benchmarking problems and the development of a framework for independent review of the proposed machine learning models by e.g., medical device, digital and health communities to assist them in getting regulatory approval

Organisation name of the lead participant for the deliverable: KTU

Due date of the deliverable: 31.05.2026 (M35)

Actual submission date of the deliverable: 29.05.2026

Confidentiality Status: PU - Public, fully open (remember to deposit public deliverables in a trusted repository)

Deliverable Cover Sheet

Funded by the European Union. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or EURAMET. Neither the European Union nor the granting authority can be held responsible for them.

The project has received funding from the European Partnership on Metrology, co-financed from the European Union's Horizon Europe Research and Innovation Programme and by the Participating States.

European Partnership Co-funded by the European Union

METROLOGY PARTNERSHIP EURAMET

CONTENTS

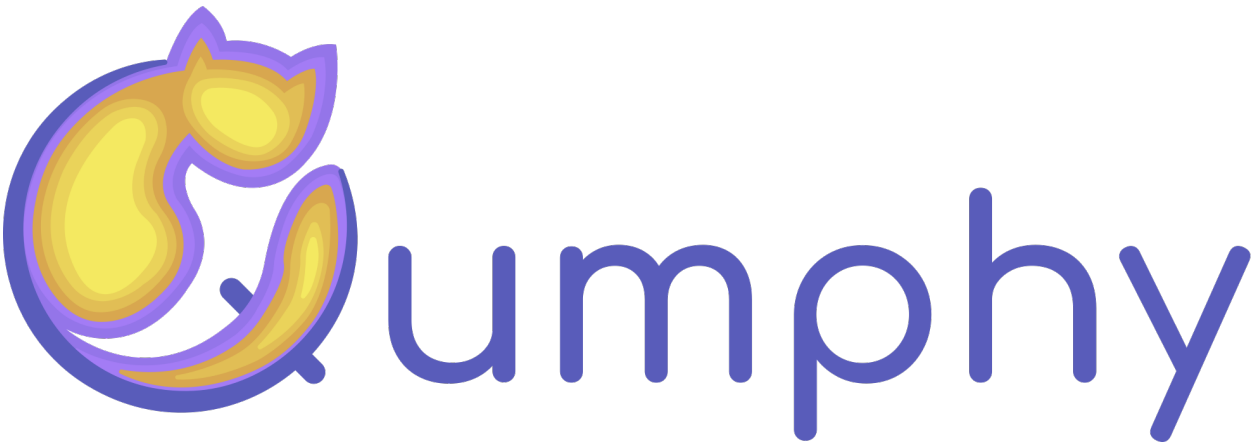
1 License **2**

2 Funding **3**

3 Contents **4**

3.1 Install . . . 4
3.2 Usage . . . 4
3.2.1 Using the Training Graphical User Interface . . . 4
3.2.2 Launching the GUI . . . 5
3.2.3 Window Layout . . . 5
3.2.4 1. Choose a Python Environment . . . 6
3.2.4.1 Creating a new environment from `requirements.txt` . . . 7
3.2.5 2. Pick a Config . . . 7
3.2.6 3. Edit the Config (optional) . . . 7
3.2.7 4. Run . . . 7
3.2.8 Troubleshooting . . . 8
3.3 Writing a Config File . . . 8
3.3.1 The core idea: `class_path` + `init_args` . . . 8
3.3.1.1 Nesting components via `classes` . . . 8
3.3.2 Top-level structure . . . 9
3.3.3 Walkthrough: a deep-ensemble regressor for PulseDB . . . 9
3.3.3.1 Header . . . 9
3.3.3.2 `model` . . . 10
3.3.3.3 `trainer` . . . 11
3.3.3.4 `data` . . . 12
3.3.3.5 `ensemble` . . . 12
3.3.4 Uncertainty-quantification variants . . . 12
3.3.4.1 1. Deep ensemble + Gaussian NLL (default) . . . 12
3.3.4.2 2. Monte-Carlo dropout (MCD) . . . 12
3.3.4.3 3. Quantile / pinball loss . . . 13
3.3.5 Common edits . . . 13
3.3.6 CLI parameter overrides . . . 13
3.3.7 Validating a config before a long run . . . 13
3.3.8 Where to put your own configs . . . 14
3.4 App package . . . 14
3.4.1 Submodules . . . 14
3.4.1.1 app.deepbeat_converter module . . . 14
3.4.1.2 app.deepbeat_data_visualization module . . . 14
3.4.1.3 app.gui module . . . 15
3.4.1.4 app.mimic3bp_converter module . . . 16

3.4.1.5 app.mimic3bp_data_visualization module . . . 17
3.4.1.6 app.pulsedb_statistics module . . . 17
3.4.1.7 app.train module . . . 18
3.4.1.8 app.tutorial_metrics module . . . 19
3.5 QUMPHY package . . . 20
3.5.1 Subpackages . . . 20
3.5.1.1 qumphy.callbacks package . . . 20
3.5.1.2 qumphy.data package . . . 23
3.5.1.3 qumphy.evaluate package . . . 32
3.5.1.4 qumphy.misc package . . . 35
3.5.1.5 qumphy.models package . . . 39
3.5.2 Submodules . . . 78
3.5.2.1 qumphy.metrics module . . . 78
3.5.2.2 qumphy.trainer module . . . 90
3.5.2.3 qumphy.uq module . . . 92
3.5.2.4 qumphy.uq_metrics module . . . 93



For more information, see the 22HLT01 QUMPHY Project Homepage and the QUMPHY Software Package Repository.

CHAPTER

ONE

LICENSE

This work is licensed under European Union Public License v1.2 or later.

```
SPDX-License-Identifier:  EUPL-1.2+
```

CHAPTER

TWO

FUNDING

The 22HLT01 QUMPHY Software Package is part of the EPM project 22HLT01 QUMPHY. The project (22HLT01 QUMPHY) has received funding from the European Partnership on Metrology, co-financed from the European Union's Horizon Europe Research and Innovation Programme and by the Participating States.

CHAPTER

THREE

CONTENTS

3.1 Install

First clone the repository to your local machine:
`git clone git@gitlab.com:qumphy/qumphy-software.git`

And `cd` int the repository: `cd qumphy-software`

Then create a new `pip` or `conda` environment and activate it:
Using **Conda**:

```
conda env create -f environment.yml -n qumphy
conda activate qumphy
conda-develop .
```

Using **pip**:

```
python3 -m venv .venv
echo "*" > .venv/.gitignore # To exclude the pip env from git tracking
source .venv/bin/activate
pip install -r requirements.txt
pip install -e .
```

Now qumphy is ready to be used!

3.2 Usage

Move the dataset files that you want to use to the `data` directory (e.g. in a subdirectory deepbeat). Write a config file for your model (use the templates as a basis).

Run the config file for training and testing:

```
python app/train.py --config app/configs/CONFIG_FILE.yml
```

3.2.1 Using the Training Graphical User Interface

The repository ships a small Tk-based Graphical User Interface (GUI) (`app/gui.py`) that wraps the command above. It lets you pick (or create) a Python environment, choose a YAML config from `app/configs/` (or anywhere on disk), optionally edit the config in place, and stream `train.py`'s output back to a panel.

3.2.2 Launching the GUI

Pick whichever method matches your platform — all three end up running `python app/gui.py`:

Platform	How to launch
Any	`python app/gui.py` from a shell with the project's Python env active
Linux	Double-click `app/QUMPHY-Train.desktop` in a file manager (Nautilus / Dolphin will ask you to mark it trusted the first time)
macOS	Double-click `app/launch_gui.command` in Finder
Windows	Double-click `app\launch_gui.bat`

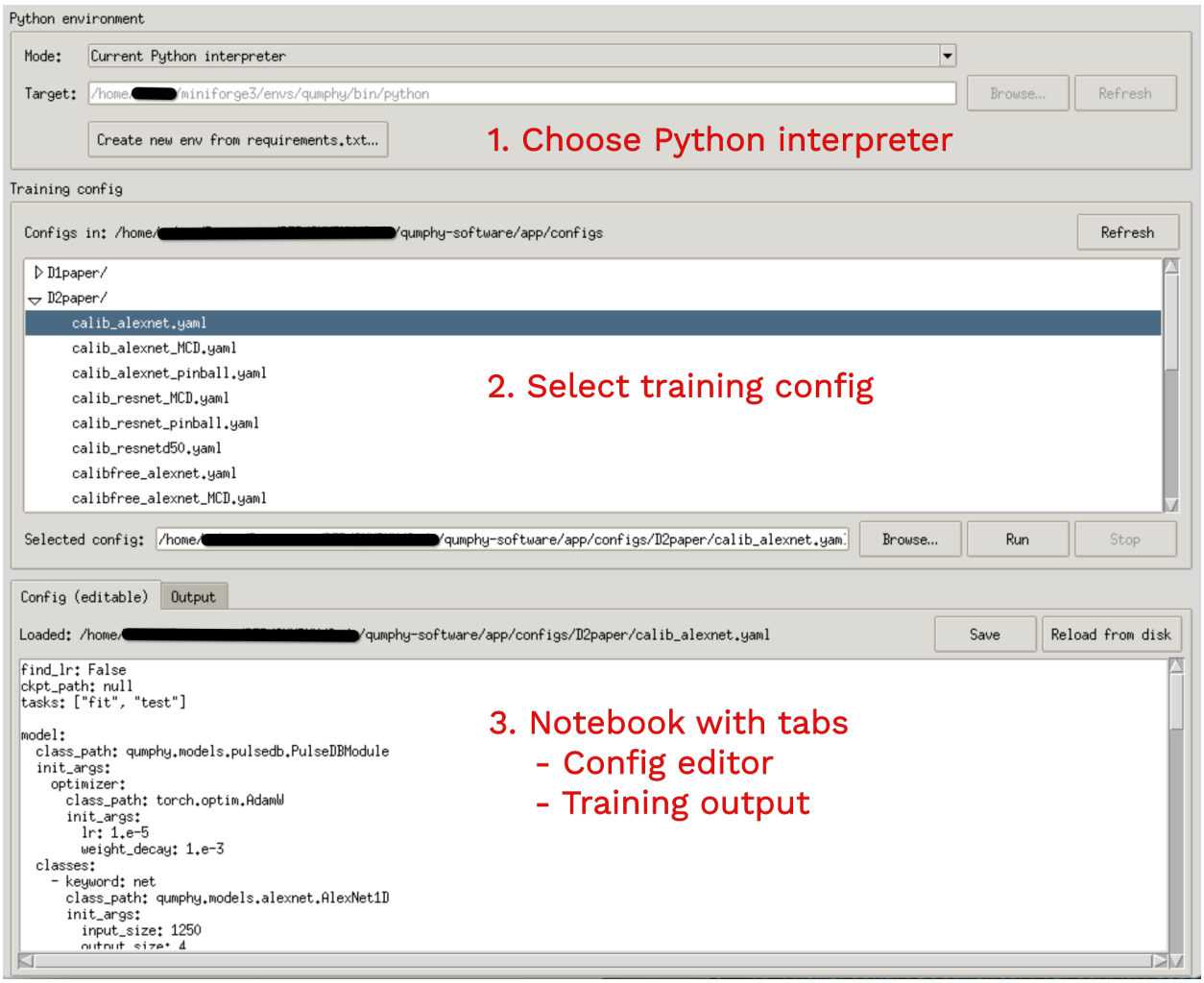
The GUI uses **tkinter** from the standard library, but tkinter links against the system Tcl/Tk shared libraries. If the GUI fails to start with an `ImportError: No module named '_tkinter'`, e.g.:

```
❯ python app/gui.py
Traceback (most recent call last):
  File "/home/[redacted]/qumphy-software/app/gui.py", line 41, in <module>
    from tkinter import (
    ...<18 lines>...
    )
  File "/usr/lib/python3.14/tkinter/__init__.py", line 38, in <module>
    import _tkinter # If this fails your Python may not be configured for Tk
    ^^^^^^^^^^^^^^^
ImportError: libtk8.6.so: cannot open shared object file: No such file or directory
```

then, install the Tk runtime for your OS (e.g. `sudo apt install python3-tk` on Debian/Ubuntu, `brew install python-tk` on macOS, or reinstall Python with the "tcl/tk and IDLE" component on Windows).

3.2.3 Window Layout

The window is divided into four sections, top to bottom:



1. **Python environment** — what interpreter `train.py` will be launched with.
2. **Training config** — the file tree of configs and the selection row.
3. **Notebook** — two tabs:
   - **Config (editable)** — YAML editor for the currently selected config.
   - **Output** — live stdout/stderr of the running subprocess.

3.2.4 1. Choose a Python Environment

The **Mode** dropdown selects how the interpreter is resolved:

- **Current Python interpreter** — uses `sys.executable`, i.e. whatever Python is running the GUI itself. Useful when you started the GUI from an already-activated venv or conda env.
- **Existing conda environment** — click **Refresh** to list environments reported by `conda env list --json`, then pick one. `conda` (or `mamba`) must be on `PATH`. The GUI invokes commands as `conda run --no-capture-output -n <name> python ...`.
- **Existing venv / virtualenv directory** — click **Browse…** and pick the venv directory (the one containing `bin/` or `Scripts/`).

3.2.4.1 Creating a new environment from `requirements.txt`

Click **Create new env from requirements.txt…**. The dialog asks for:

- **Type** — `venv` (a Python virtualenv directory) or `conda` (a named conda env).
- **Name / path** — directory path for venv, env name for conda.
- **Python version** — only used for `conda create -n <name> python=<ver> -y`.

The GUI then runs, in order:

- **venv**: `python -m venv <path>` → `pip install -U pip` → `pip install -r requirements.txt`
- **conda**: `conda create -n <name> python=<ver> -y` → `conda run -n <name> pip install -r requirements.txt`

Each step's output streams into the **Output** tab. On success the new environment is automatically selected in the mode dropdown.

3.2.5 2. Pick a Config

The tree on the left shows everything under `app/configs/` that ends in `.yaml` or `.yml`. Click a file to select it; its path appears in the **Selected config** field and its content is loaded into the editor tab.

- **Browse…** opens a file picker for configs outside `app/configs/`.
- **Refresh** rescans the directory (useful after creating a new config outside the GUI).
- **Double-clicking** a config in the tree launches training immediately.

3.2.6 3. Edit the Config (optional)

The **Config (editable)** tab is a YAML editor with undo/redo. The label above it shows the load status:

- `Loaded: <path>` — the editor matches what's on disk.
- `Modified: <path>` — there are unsaved changes.
- `Saved: <path>` — you've just persisted changes back to the file.

Buttons:

- **Save** writes the editor content back to the original file. If PyYAML is installed (it is part of `requirements.txt`), the content is parsed first and you'll be prompted if it has syntax errors.
- **Reload from disk** discards in-memory edits (with confirmation) and re-reads the file.

You don't have to save to run with edits — see below.

3.2.7 4. Run

Click **Run** (or double-click a config in the tree). The GUI invokes:

```
<env-python> app/train.py --config <CONFIG_FILE>
```

with `app/` as the working directory. The exact command is echoed into the **Output** tab before the run starts.

Behaviour with edited configs:

- If the editor matches disk → the selected path is passed through unchanged.
- If the editor has unsaved changes → the current text is written to a fresh temp file (preserving the original basename and `.yaml`/`.yml` suffix) and that temp path is passed to `train.py`. Your original file is left untouched. The Output tab logs: `[using edited config from temp file: ...]`.

This means you can iterate on a config without overwriting the on-disk version. Hit **Save** when you want to keep the edits.

While a run is in progress:

- **Run**, **Mode** and **Create new env…** are disabled.
- **Stop** terminates the subprocess (`SIGTERM` on POSIX, `TerminateProcess` on Windows).
- Closing the window asks for confirmation before killing the run.

3.2.8 Troubleshooting

- *"Could not find any conda environments"* — `conda` is not on `PATH`, or you have only the base env. Make sure `conda init <shell>` was run for the shell you launched the GUI from.
- *Invalid venv* — the path you selected does not contain `bin/python` (POSIX) or `Scripts\python.exe` (Windows).
- *Run starts but immediately exits* — check the **Output** tab; the most common cause is the chosen environment not having the project's dependencies installed. Use **Create new env from requirements.txt…** to bootstrap one.
- *Config edits not applied* — confirm the status above the editor reads `Modified:` and not `Loaded:`; only dirty editors get written to the temp file.

3.3 Writing a Config File

QUMPHY trains models from a single YAML config that is parsed by `app/train.py` and turned into a Lightning training run by *`qumphy.trainer.Trainer`*. The config describes **what to build** (model, data module, trainer, callbacks, …), not **how to run it** from the `tasks:` section of the config `fit`, `test`, `predict`.

The best starting points are the up-to-date templates under `app/configs/D2paper/`.

3.3.1 The core idea: `class_path` + `init_args`

Every component in a QUMPHY config is constructed from the same shape:

```
class_path: dotted.path.to.SomeClass
init_args:
  some_kwarg: value
  another_kwarg: 42
```

At runtime, *`qumphy.misc.misc.instantiate_class()`* imports the class named by `class_path` and instantiates it with the keyword arguments in `init_args`. This is how Lightning's `cli`-style configs work, and it applies uniformly to optimizers, loss functions, networks, callbacks, loggers, and the trainer itself.

3.3.1.1 Nesting components via `classes`

Many objects take other objects as constructor arguments — a model needs a network, a loss function, optionally an output activation; a trainer needs callbacks and a logger. Those nested objects are listed under a sibling `classes:` key:

```
class_path: qumphy.models.pulsedb.PulseDBModule
init_args:
  optimizer:
    class_path: torch.optim.AdamW
    init_args:
```

(continues on next page)

(continued from previous page)

```
      lr: 1.e-5
      weight_decay: 1.e-3
classes:
  - keyword: net
    class_path: qumphy.models.alexnet.AlexNet1D
    init_args:
      input_size: 1250
      output_size: 4
  - keyword: loss_fn
    class_path: qumphy.models.pulsedb.PulseDBGaussianLoss
    init_args:
      num_distributions: 2
```

Each entry under `classes` carries a `keyword:` field — that's the constructor argument name the instantiated object is passed to (so `net=AlexNet1D(...)`, `loss_fn=PulseDBGaussianLoss(...)`).

If the same `keyword` appears multiple times, the instances are collected into a list. This is how the trainer ends up with several callbacks (see the trainer example below).

`init_args: {}` is a valid empty value — use it when a class takes no arguments.

3.3.2 Top-level structure

A complete training config has these top-level keys:

Key	Required	What it is
`model`	yes	LightningModule definition (class + nested net, loss, etc.)
`data`	yes	LightningDataModule definition
`trainer`	yes	`lightning.Trainer` + its callbacks and logger
`ensemble`	no	Wraps the model + trainer in an N-member ensemble
`find_lr`	no	If `True`, run Lightning's LR finder before training
`ckpt_path`	no	Path to a checkpoint to resume from (or per-member list for ensembles)
`seed_everything`	no	Integer seed; defaults to `None` (no seeding)
`feature_extractor`	no	Optional pre-trained feature extractor applied to the data
`sweep_parameters`	no	Used by W&B sweeps to map top-level keys into nested locations

`tasks` is consumed by the trainer to decide what to do (`fit`, `test`, `predict`); CLI flags (`--fit`, `--test`, `--predict`) act as additional toggles in `app/train.py`.

3.3.3 Walkthrough: a deep-ensemble regressor for PulseDB

The file `app/configs/D2paper/calib_alexnet.yaml` is a good reference. We'll go through it block by block.

3.3.3.1 Header

```
find_lr: False
ckpt_path: null
```

`find_lr` runs Lightning's learning-rate finder when `True`. `ckpt_path: null` means start from scratch; provide a string path here to resume.

3.3.3.2 `model`

```yaml
model:
  class_path: qumphy.models.pulsedb.PulseDBModule
  init_args:
    optimizer:
      class_path: torch.optim.AdamW
      init_args:
        lr: 1.e-5
        weight_decay: 1.e-3
  classes:
    - keyword: net
      class_path: qumphy.models.alexnet.AlexNet1D
      init_args:
        input_size: 1250
        output_size: 4
    - keyword: dataset
      class_path: qumphy.data.pulsedb.PulseDBDataModule
      init_args:
        data_directory: /gpu-scratch/pfeffe01/pulsedb
        batch_size: 32
        num_workers: 1
        dataset: calib
        source: vital
        load_data: False
    - keyword: loss_fn
      class_path: qumphy.models.pulsedb.PulseDBGaussianLoss
      init_args:
        num_distributions: 2
```

Notes:

- The optimizer (and an optional `lr_scheduler:` block) sit *inside* `init_args`, not under `classes`, because PyTorch optimizer objects are passed as ordinary keyword arguments to the LightningModule.
- `net.output_size: 4` matches `loss_fn.num_distributions: 2` — the Gaussian loss expects two parameters (`mu`, `sigma`) per output target, and PulseDB calibration has two targets (SBP, DBP). For pinball loss the output size is `len(quantiles) * num_targets` instead.
- The `dataset:` sub-class inside `model` is the data module used for internal calibration / normalisation only; the *actual* training data module is configured separately under the top-level `data:` key (see below).

Adding a learning-rate scheduler

```yaml
init_args:
  optimizer: { ... }
  lr_scheduler:
    class_path: torch.optim.lr_scheduler.ReduceLROnPlateau
    init_args:
      mode: max
      factor: 0.5
      patience: 8
    config:
      monitor: val_auc
```

The extra `config:` block at the scheduler level holds the Lightning scheduler config (which metric to monitor, etc.), separate from the constructor arguments. See `app/configs/D2paper/deepbeat_alexnet.yaml`.

3.3.3.3 trainer

```yaml
trainer:
  class_path: lightning.Trainer
  init_args:
    accelerator: auto
    fast_dev_run: False
    max_epochs: 100
    overfit_batches: 0
    log_every_n_steps: 1
    precision: "32"
    default_root_dir: /.../logs/
  classes:
    - keyword: callbacks
      class_path: qumphy.callbacks.pulsedb.PulseDBLogging_Ensemble
      init_args:
        log_quantities: ["mae", "std"]
        log_pressure: "both"
        save_predictions: True
    - keyword: logger
      class_path: lightning.pytorch.loggers.WandbLogger
      init_args:
        save_dir: /.../logs/
        offline: True
        project: calib
        name: alexnet
    - keyword: callbacks
      class_path: lightning.pytorch.callbacks.early_stopping.EarlyStopping
      init_args:
        monitor: val_loss
        patience: 15
        mode: min
    - keyword: callbacks
      class_path: qumphy.callbacks.progressbar.EpochProgressBar
      init_args: {}
    - keyword: callbacks
      class_path: lightning.pytorch.callbacks.ModelCheckpoint
      init_args:
        monitor: val_loss
        mode: min
        save_last: True
        save_top_k: 1
```

The four entries with `keyword: callbacks` are collected into a list and passed as `callbacks=[...]` to `lightning.Trainer`. The single `logger:` entry is passed as `logger=WandbLogger(...)`. Replace it with e.g. `lightning.pytorch.loggers.TensorBoardLogger` if you don't use W&B.

`fast_dev_run: True` is the quickest way to smoke-test a new config — one batch through fit/validate/test, no checkpoints.

3.3.3.4 data

```yaml
data:
  class_path: qumphy.data.pulsedb.PulseDBDataModule
  init_args:
    data_directory: /gpu-scratch/pfeffe01/pulsedb
    batch_size: 32
    num_workers: 14
    dataset: calib
    source: vital
    load_data: True
```

Notes:

- `data_directory` should point at the local copy of the dataset (PulseDB / DeepBeat). Change this to match your machine.
- For PulseDB, valid `dataset:` values include `calib`, `calibfree`, and `mini` (the small dev split).
- For DeepBeat, see `deepbeat_alexnet.yaml`: `dataset: set_revised`, plus a `target_format` field (`class_index` for cross-entropy-style targets, `one_hot` for the KG-loss variants).
- `num_workers` should be increased on production runs; the value inside the model's nested `dataset` block can stay low (it's only used for instantiation, not the actual loader).

3.3.3.5 ensemble

```yaml
ensemble:
  size: 5
  class_path: qumphy.models.pulsedb.PulseDBEnsemble
  init_args: {}
```

When `ensemble:` is present, the trainer creates `size` Lightning trainers + models, fits each independently, then wraps them in the class given by `class_path` for inference. Remove this block to train a single model.

For DeepBeat ensembles, the wrapper takes an extra `noise_samples:` argument; see `deepbeat_alexnet.yaml`.

3.3.4 Uncertainty-quantification variants

Three approaches are represented in `D2paper/`, and each picks a different combination of network + loss:

3.3.4.1 1. Deep ensemble + Gaussian NLL (default)

- Network: `qumphy.models.alexnet.AlexNet1D` or `qumphy.models.xresnet1d.XResNet1d50`
- Loss: `qumphy.models.pulsedb.PulseDBGaussianLoss(num_distributions: 2)` → predicts `mu` and `sigma`; combined with `ensemble.size: 5` gives a deep ensemble.
- Output size: `2 × num_distributions = 4`.

Example: `calib_alexnet.yaml`.

3.3.4.2 2. Monte-Carlo dropout (MCD)

- Module: `qumphy.models.pulsedb.PulseDBModule_MCD` with `MCD_samples: 50` (number of stochastic forward passes at inference).
- Network: the `_MCD` variant (e.g. `AlexNet1D_MCD`) with `dropout_rate: 0.05` and `mcdropout: True`.
- Logging callback: `qumphy.callbacks.pulsedb.PulseDBLogging_MCD`.

- No `ensemble:` block — uncertainty comes from dropout sampling.

Example: `calib_alexnet_MCD.yaml`.

3.3.4.3 3. Quantile / pinball loss

- Loss: `qumphy.models.utils.pinballloss.PinballLoss`
- Loss args: `quantiles: [0.0228, 0.1587, 0.5, 0.8413, 0.9772]` and `num_targets: 2`.
- Network output size = `len(quantiles) * num_targets` (here, `10`).
- Logging callback: `qumphy.callbacks.pulsedb.PulseDBLogging_Pinballloss`.
- No `ensemble:` block — a single model emits the full quantile vector.

Example: `calib_alexnet_pinball.yaml`.

When you change loss family, **three things must agree**: the loss class, the network's `output_size`, and the logging callback under the trainer. Mismatches surface as cryptic shape errors at the first training step.

3.3.5 Common edits

Goal	Change
Try a quick run	Set `trainer.init_args.fast_dev_run: True`
Use a different network	Swap the `keyword: net` entry's `class_path` and `init_args` (mind `input_size` / `output_size`)
Use a different dataset split	Change `data.init_args.dataset` (`calib`, `calibfree`, `mini`, `set_revised`, …)
Disable W&B	Remove the `WandbLogger` entry or set `offline: True`
Stop earlier / later	Adjust the `EarlyStopping` `patience` and / or `Trainer.max_epochs`
Train a single model instead of an ensemble	Delete the `ensemble:` block
Resume from a checkpoint	Set `ckpt_path:` (string for single model, list of strings for an ensemble)
Reproducibility	Add `seed_everything: 42` (or any int) at the top level

3.3.6 CLI parameter overrides

`train.py` accepts `-p key.subkey:value` to overlay values onto the loaded config without editing the YAML. The same syntax can be passed multiple times:

```
python app/train.py --config app/configs/D2paper/calib_alexnet.yaml \
    -p trainer.init_args.max_epochs:5 \
    -p data.init_args.batch_size:16
```

Unknown `--key:value` flags are also accepted (see *qumphy.misc.misc.train_argument_parser()*). This is the same mechanism the W&B sweep agent uses.

3.3.7 Validating a config before a long run

The fastest sanity check is a single-batch fit:

```
trainer:
  init_args:
    fast_dev_run: True
```

…or override from the CLI:

```
python app/train.py --config <file> -p trainer.init_args.fast_dev_run:True
```

If you use the GUI (*Usage*), the **Config (editable)** tab lets you toggle that flag and hit Run without modifying the on-disk file.

3.3.8 Where to put your own configs

`app/configs/` is part of the repository, and most subdirectories (`D1paper/`, `D2paper/`, ...) are version-controlled curated sets. For ad-hoc experiments use `app/configs/personal_configs/` or `app/configs/working_configs/`, which are ignored by git via `app/configs/.gitignore`.

3.4 App package

File: app/__init__.py Project: 22HLT01 QUMPHY Contact: nando.hegemann@ptb.de Gitlab: https://gitlab.com/qumphy Description: init

3.4.1 Submodules

3.4.1.1 app.deepbeat_converter module

File: app/deepbeat_converter.py Project: 22HLT01 QUMPHY Contact: nando.hegemann@ptb.de Gitlab: https://gitlab.com/qumphy Description: Convert DeepBeat Database into PPG training chunks. SPDX-License-Identifier: EUPL-1.2

`app.deepbeat_converter.`**`convert`**(*file_name*, *save_path*, *folds=1*)

Chunk data into multiple npy files.

Return type
`str`

Parameters

- **`file_name`** (*str*) – Path to file.
- **`save_path`** (*str*) – Save directory.
- **`folds`** (*int, optional*) – Number of different files the file is split into.

Returns
Save location.

3.4.1.2 app.deepbeat_data_visualization module

File: app/deepbeat_data_visualization.py Project: 22HLT01 QUMPHY Contact: nando.hegemann@ptb.de Gitlab: https://gitlab.com/qumphy Description: Create images of samples from DeepBeat dataset.

`app.deepbeat_data_visualization.`**`main`**()

Render one PNG grid per DeepBeat split into `../img/deepbeat/`.

Return type
`None`

`app.deepbeat_data_visualization.`**`normalize_signals`**(*signals*)

Normalize signal data to [0, 1].

Return type
`ndarray`

Parameters
signals (*np.ndarray*) – Signal array.

Returns
Normalized signal array.

app.deepbeat_data_visualization.**plot_signal_grid**(*labels*, *signals*, *subject_ids*, *tag*, *save_path*, *dpi=300*)

Save a 6x4 grid of randomly sampled PPG signals colored by AF label.

Return type
None

Parameters

- **labels** (*np.ndarray*) – Binary AF labels (0/1) for each sample.
- **signals** (*np.ndarray*) – PPG time series.
- **subject_ids** (*np.ndarray*) – Subject IDs.
- **tag** (*str*) – Filename stem and figure title suffix.
- **save_path** (*str*) – Directory the <tag>.png image is written to.
- **dpi** (*int, optional*) – Resolution of image, by default 300.

3.4.1.3 app.gui module

File: app/gui.py Project: 22HLT01 QUMPHY Contact: oskar.pfeffer@ptb.de Gitlab: https://gitlab.com/qumphy Description: Graphical user interface for training the models.

Cross-platform Tk GUI that:

- **lets the user pick (or create) a Python environment to run in:**
  - current Python interpreter,
  - an existing conda environment,
  - an existing virtualenv / venv directory,
  - a new venv created from requirements.txt,
  - a new conda env created from requirements.txt;
- lets the user pick a YAML config from app/configs (or browse the filesystem) and runs:

  <env-python> train.py –config <CONFIG_FILE>

Standard-library only — runs on Windows, macOS and Linux.

class app.gui.**TrainGUI**(*root*)

Bases: object

Tk train-launcher window.

Wraps the full launcher workflow: pick (or create) a Python environment, pick or edit a YAML config from app/configs, and run *app.train* against the selected config in a background subprocess whose output is streamed back into the GUI.

Parameters
root (*tkinter.Tk*) – Top-level Tk window the GUI is attached to.

app.gui.**list_conda_envs**()

Return conda environment names. Empty list if conda is unavailable.

Return type
list[str]

app.gui.**main**()

Launch the train GUI and run the Tk main loop until the window closes.

Return type
int

Returns
Process exit code (always `0` from the launcher itself).

Return type
int

app.gui.**venv_python**(*venv_path*)

Return the python executable inside a venv directory.

Return type
Path

3.4.1.4 app.mimic3bp_converter module

File: app/mimic3bp_converter.py Project: 22HLT01 QUMPHY Contact: nando.hegemann@ptb.de Gitlab: https://gitlab.com/qumphy Description: Convert MIMIC-III Blood Pressure Database into PPG training chunks. SPDX-License-Identifier: EUPL-1.2

app.mimic3bp_converter.**convert**(*file_name*, *save_path*, *folds=1*)

Chunk data into multiple npy files.

Return type
str

Parameters

- **file_name** (*str*) – Path to file.
- **save_path** (*str*) – Save directory.
- **folds** (*int, optional*) – Number of different files the file is split into.

Returns
Save location.

app.mimic3bp_converter.**map_labels_to_classes**(*blood_pressure*)

Map systolic/diastolic blood pressure to hypertension classes.

The hypertension classes are given by the following table

Category	Value	Systolic (mmgG)	and/or	Diastolic (mmHg)
normal	0	< 120	and	< 80
elevated	1	120 - 129	and	< 80
Hypertension Stage 1	2	130 - 139	or	80 - 89
Hypertension Stage 2	3	> 139	or	> 89
Hypertensive Crisis	4	> 179	and/or	> 119

Classes separation is done according to Hypertension classes.

Parameters
blood_pressure (*np.ndarray*) – Array with systolic/diastolic blood pressure values.

Returns
Array containing respevtive hypertension classes.

Return type
ndarray

3.4.1.5 app.mimic3bp_data_visualization module

File: app/mimic3bp_data_visualization.py Project: 22HLT01 QUMPHY Contact: nando.hegemann@ptb.de Gitlab: https://gitlab.com/qumphy Description: Create images of samples from MIMIC III BP dataset.

app.mimic3bp_data_visualization.**main**()

Render one PNG grid per MIMIC III BP split into `../img/mimic3bp/`.

Return type
None

app.mimic3bp_data_visualization.**normalize_signals**(*signals*)

Normalize signal data to [0, 1].

Return type
ndarray

Parameters
signals (*np.ndarray*) – Signal array.

Returns
Normalized signal array.

app.mimic3bp_data_visualization.**plot_signal_grid**(*labels*, *signals*, *subject_ids*, *tag*, *save_path*, *dpi=300*)

Save a 6x6 grid of randomly sampled PPG signals colored by blood pressure.

Inner line color encodes systolic BP, outer line color encodes diastolic BP.

Return type
None

Parameters

- **labels** (*np.ndarray*) – Systolic/diastolic blood pressure values, shape `(n_samples, 2)`.
- **signals** (*np.ndarray*) – PPG time series.
- **subject_ids** (*np.ndarray*) – Subject IDs.
- **tag** (*str*) – Filename stem and figure title suffix.
- **save_path** (*str*) – Directory the `<tag>.png` image is written to.
- **dpi** (*int, optional*) – Resolution of image, by default 300.

3.4.1.6 app.pulsedb_statistics module

File: app/pulsedb_statistics.py Project: 22HLT01 QUMPHY Contact: oskar.pfeffer@ptb.de Gitlab: https://gitlab.com/qumphy Description: Script for generating statistics for PulseDB data.

app.pulsedb_statistics.**main**()

CLI entry point: compute PulseDB target statistics and write `stats.yaml`.

Parses `data_directory` from the command line and delegates to *qumphy.data.pulsedb.write_target_stats_yaml()*, which writes a `stats.yaml` containing the mean, median, std and baseline measures of the SBP and DBP targets into the given directory.

Return type
None

3.4.1.7 app.train module

File: app/train.py Project: 22HLT01 QUMPHY Contact: oskar.pfeffer@ptb.de Gitlab: https://gitlab.com/qumphy Description: Entry point for training, testing and predicting QUMPHY models.

app.train.**load_config**(*args*)

Merge YAML config files and command-line overrides into a single dict.

Reads each YAML file listed in `args.config` (later files override earlier ones) and then applies the parameter overrides in `args.parameters`.

Parameters
args (*argparse.Namespace*) – Parsed command-line arguments. Must expose `config` (list of YAML paths) and `parameters` (iterable of dotted-key/value overrides).

Returns
Combined configuration dictionary.

Return type
dict

app.train.**main**()

CLI entry point: parse arguments and dispatch to Optuna or plain training.

app.train.**objective**(*trial*, *args*, *config*)

Optuna objective: sample hyperparameters, train, return early-stop score.

For each parameter listed in `config["optuna"]["parameters"]`, the matching Optuna `trial.suggest_*` function is called with its arguments and the sampled value is written back into `config` before training.

Return type
float

Parameters

- **trial** (*optuna.trial.Trial*) – Trial object provided by the Optuna study.
- **args** (*argparse.Namespace*) – Parsed command-line arguments forwarded to *train()*.
- **config** (*dict*) – Training configuration; mutated in-place with the sampled parameters.

Returns
Best validation score recorded by the trainer's early-stopping callback.

Return type
float

app.train.**run_optuna**(*args*, *config*)

Create and run an Optuna study around *objective()*.

Parameters

- **args** (*argparse.Namespace*) – Parsed command-line arguments forwarded to *objective()*.
- **config** (*dict*) – Configuration with an "optuna" section providing sampler, direction, optional sampler_arguments, n_trials and timeout.

Returns
The completed study. Best trial, value and parameters are also printed.

Return type
optuna.study.Study

app.train.**train**(*args*, *config*)

Run the QUMPHY trainer for the tasks selected in config / args.

Executes fit, test and/or predict depending on whether each task is listed in config["tasks"] or enabled by the matching CLI flag.

Parameters

- **args** (*argparse.Namespace*) – Parsed command-line arguments providing the fit, test and predict boolean flags.
- **config** (*dict*) – Combined training configuration.

Returns
The trainer instance after the requested tasks have run.

Return type
qumphy.trainer.Trainer

3.4.1.8 app.tutorial_metrics module

File: app/tutorial_metrics.py Project: 22HLT01 QUMPHY Contact: nando.hegemann@ptb.de Gitlab: https://gitlab.com/qumphy Description: Tutorial for evaluation metrics for model performance.

app.tutorial_metrics.**mc2ml**(*arr*, *noise=0.0*)

Helper function to convert multi-class array to multi-lable array.

Converts a 1D array with n classes in a matrix with n columns. Each row is zero except for the column corresponding to the class lable, which is set to one. Noise add a perturbation to the values and subsequently normalizes the rows to mimic softmax values.

Return type
ndarray

Parameters

- **arr** (*np.ndarray*) – Array with two or more different values.
- **noise** (*float, optional*) – Noise level.

Returns
Multi-lable version of the array.

Return type
np.ndarray

3.5 QUMPHY package

File: qumphy/__init__.py Project: 22HLT01 QUMPHY Contact: nando.hegemann@ptb.de Gitlab: https://gitlab.com/qumphy Description: init

3.5.1 Subpackages

3.5.1.1 qumphy.callbacks package

File: qumphy/callbacks/__init__.py Project: 22HLT01 QUMPHY Contact: oskar.pfeffer@ptb.de Gitlab: https://gitlab.com/qumphy Description: init

Submodules

qumphy.callbacks.base_logging module

File: qumphy/models/lightning_callbacks.py Project: 22HLT01 QUMPHY Contact: oskar.pfeffer@ptb.de Gitlab: https://gitlab.com/qumphy Description: Pytorch lightning callbacks.

class qumphy.callbacks.base_logging.**BaseLoggingCallback**(**args: Any*, ***kwargs: Any*)

Bases: Callback

This is a base class for logging callbacks. Predictions and targets of each epoch for each stage are stored in dictionaries in self.predictions and self.targets as self.predictions[stage] and self.targets[stage], where stage is "train", "val" or "test". This allows to write new logging callbacks on top of this class, by overriding the log_epoch_end method.

log_epoch_end(*trainer*, *pl_module*, *stage*)

Override this method to log the metrics at the end of the epoch.

log_loss(*trainer*, *pl_module*, *stage*)

on_test_batch_end(*trainer*, *pl_module*, *outputs*, *batch*, *batch_idx*)

on_test_epoch_end(*trainer*, *pl_module*)

on_test_epoch_start(*trainer*, *pl_module*)

on_train_batch_end(*trainer*, *pl_module*, *outputs*, *batch*, *batch_idx*)

on_train_epoch_end(*trainer*, *pl_module*)

on_train_epoch_start(*trainer*, *pl_module*)

on_validation_batch_end(*trainer*, *pl_module*, *outputs*, *batch*, *batch_idx*)

on_validation_epoch_end(*trainer*, *pl_module*)

on_validation_epoch_start(*trainer*, *pl_module*)

qumphy.callbacks.deepbeat module

File: qumphy/models/lightning_callbacks.py Project: 22HLT01 QUMPHY Contact: oskar.pfeffer@ptb.de Gitlab: https://gitlab.com/qumphy Description: Pytorch lightning callbacks.

class qumphy.callbacks.deepbeat.**DeepBeatLogging**(**args: Any*, ***kwargs: Any*)

Bases: *BaseLoggingCallback*

Pytorch lightning callback for logging for the DeepBeat dataset.

log_epoch_end(*trainer*, *pl_module*, *stage*)

Override this method to log the metrics at the end of the epoch.

class qumphy.callbacks.deepbeat.**DeepBeatLogging_KGLoss**(**args: Any*, ***kwargs: Any*)

Bases: *DeepBeatLogging*

on_test_epoch_end(*trainer*, *pl_module*)

class qumphy.callbacks.deepbeat.**DeepBeatLogging_MCD**(**args: Any*, ***kwargs: Any*)

Bases: *DeepBeatLogging*

Pytorch lightning callback for logging for the DeepBeat dataset.

Initialize with a list of quantities to log.

Available quantities:

- loss
- qumphy

on_test_epoch_end(*trainer*, *pl_module*)

qumphy.callbacks.progressbar module

class qumphy.callbacks.progressbar.**EpochProgressBar**(**args: Any*, ***kwargs: Any*)

Bases: ProgressBar

disable()

Return type
None

enable()

Return type
None

on_train_end(*trainer*, *pl_module*)

Return type
None

on_train_epoch_end(*trainer*, *pl_module*)

Return type
None

on_train_epoch_start(*trainer*, *pl_module*)

Return type
None

on_train_start(*trainer*, *pl_module*)

Return type
None

qumphy.callbacks.pulsedb module

File: qumphy/models/lightning_callbacks.py Project: 22HLT01 QUMPHY Contact: oskar.pfeffer@ptb.de Gitlab: https://gitlab.com/qumphy Description: Pytorch lightning callbacks.

class qumphy.callbacks.pulsedb.**PulseDBLogging**(**args: Any*, ***kwargs: Any*)

Bases: *BaseLoggingCallback*

Pytorch lightning callback for logging for the PulseDB dataset.

Initialize with a list of quantities to log and a flag to specify which pressure to log. Loss is always logged.

Available quantities:

- mae
- rmse
- std

log_epoch_end(*trainer*, *pl_module*, *stage*)

Override this method to log the metrics at the end of the epoch.

log_ieee_metrics(*pl_module*, *values*, *stage*)

log_mae(*pl_module*, *values*, *stage*)

log_rmse(*pl_module*, *values*, *stage*)

log_std(*pl_module*, *values*, *stage*)

set_function_dictionary()

class qumphy.callbacks.pulsedb.**PulseDBLogging_Ensemble**(**args: Any*, ***kwargs: Any*)

Bases: *PulseDBLogging*

on_test_epoch_end(*trainer*, *pl_module*)

class qumphy.callbacks.pulsedb.**PulseDBLogging_MCD**(**args: Any*, ***kwargs: Any*)

Bases: *PulseDBLogging*

on_test_epoch_end(*trainer*, *pl_module*)

class qumphy.callbacks.pulsedb.**PulseDBLogging_Pinballloss**(**args: Any*, ***kwargs: Any*)

Bases: *BaseLoggingCallback*

on_test_epoch_end(*trainer*, *pl_module*)

qumphy.callbacks.sleepapnea module

Description: Pytorch lightning callbacks.

class qumphy.callbacks.sleepapnea.**SleepApneaLogging**(**args: Any*, ***kwargs: Any*)

Bases: *BaseLoggingCallback*

Pytorch lightning callback for logging for the sleepApnea dataset.

log_epoch_end(*trainer*, *pl_module*, *stage*)

Override this method to log the metrics at the end of the epoch.

3.5.1.2 qumphy.data package

File: qumphy/data/__init__.py Project: 22HLT01 QUMPHY Contact: nando.hegemann@ptb.de Gitlab: https://gitlab.com/qumphy Description: init

Subpackages

qumphy.data.signal_preprocessing package

File: qumphy/data/signal_preprocessing/__init__.py Project: 22HLT01 QUMPHY Contact: oskar.pfeffer@ptb.de Gitlab: https://gitlab.com/qumphy Description: init

Submodules

qumphy.data.signal_preprocessing.filters module

File: qumphy/data/signal_preprocessing/filters.py Project: 22HLT01 QUMPHY Contact: oskar.pfeffer@ptb.de Gitlab: https://gitlab.com/qumphy Description: Functions for filtering the signals.

qumphy.data.signal_preprocessing.filters.**apply_filter**(*signal*, *filter_params*)

Applies a filter to the input signal.

Return type
ndarray

Parameters

- **signal** (*np.ndarray*) – The input signal to be filtered.
- **filter_params** (*dict*) – A dictionary containing the filter parameters.

Returns
filtered_signal – The filtered signal.

Return type
np.ndarray

qumphy.data.signal_preprocessing.filters.**bandpass_filter**(*signal*, *lowcut*, *highcut*, *signal_frequency*, *order=4*)

Applies a Butterworth band-pass filter to the input signal.

Parameters

- **signal** (*np.ndarray*) – The input signal to be filtered.
- **lowcut** (*float*) – The lower bound of the frequency range to be filtered out.
- **highcut** (*float*) – The upper bound of the frequency range to be filtered out.
- **signal_frequency** (*float*) – The sampling frequency of the signal.
- **order** (*int, optional*) – The order of the filter. Defaults to 4.

Returns
filtered_signal – The filtered signal.

Return type
np.ndarray

qumphy.data.signal_preprocessing.filters.**highpass_filter**(*signal*, *cutoff*, *signal_frequency*, *order=4*)

Applies a Butterworth high-pass filter to the input signal.

Parameters

- **signal** (*np.ndarray*) – The input signal to be filtered.
- **cutoff** (*float*) – The frequency below which the signal should be filtered out.
- **signal_frequency** (*float*) – The sampling frequency of the input signal.
- **order** (*int, optional*) – The order of the filter. Defaults to 4.

Returns
filtered_signal – The filtered signal.

Return type
np.ndarray

qumphy.data.signal_preprocessing.filters.**lowpass_filter**(*signal*, *cutoff*, *signal_frequency*, *order=4*)

Applies a Butterworth low-pass filter to the input signal.

Parameters

- **signal** (*np.ndarray*) – The input signal to be filtered.
- **cutoff** (*float*) – The frequency above which the signal should be filtered out.
- **signal_frequency** (*float*) – The sampling frequency of the input signal.
- **order** (*int, optional*) – The order of the filter. Defaults to 4.

Returns
filtered_signal – The filtered signal.

Return type
np.ndarray

qumphy.data.signal_preprocessing.noise module

File: qumphy/data/signal_preprocessing/noise.py Project: 22HLT01 QUMPHY Contact: oskar.pfeffer@ptb.de Gitlab: https://gitlab.com/qumphy Description: Functions for adding noise to the signals.

qumphy.data.signal_preprocessing.noise.**add_noise**(*data*, *noise_params=None*)

Adds noise to the given data.

Parameters
data (*np.ndarray*) – Input data.

Returns
Data with added noise.

Return type
np.ndarray

qumphy.data.signal_preprocessing.noise.**baseline_wander_noise**(*signal*, *signal_frequency*, *snr*, *cutoff=0.5*, *seed=None*)

Add baseline wander noise to a signal.

Return type
ndarray

Parameters

- **signal** (*np.ndarray*) – Input array with shape (…, L), where the last dimension is the signal length and all preceding dimensions index different signals (e.g. (batch, channels, length))

- **snr** (*float*) – The signal-to-noise ratio (SNR) in decibels.
- **cutoff** (*float*) – The cutoff frequency of the low-pass filter.
- **seed** (*int or None, optional*) – Seed of the RNG.

Returns
The signal with added baseline wander noise.

Return type
np.ndarray

qumphy.data.signal_preprocessing.noise.**gaussian_noise**(*signal*, *snr*, *seed=None*)

Add Gaussian noise to a signal.

Return type
ndarray

Parameters

- **signal** (*np.ndarray*) – Input array with shape (..., L), where the last dimension is the signal length and all preceding dimensions index different signals (e.g. (batch, channels, length))
- **snr** (*float*) – The signal-to-noise ratio (SNR) in decibels.
- **seed** (*int or None, optional*) – Seed of the RNG.

Returns
The signal with added Gaussian noise.

Return type
np.ndarray

qumphy.data.signal_preprocessing.noise.**powerline_noise**(*signal*, *signal_frequency*, *snr*, *noise_frequency=50.0*)

Add powerline noise to a signal.

Typically, powerline noise is a sinusoidal signal with a frequency of 50 or 60 Hz.

Return type
ndarray

Parameters

- **signal** (*np.ndarray*) – Input array with shape (..., L), where the last dimension is the signal length and all preceding dimensions index different signals (e.g. (batch, channels, length))
- **signal_frequency** (*float*) – The sampling frequency (Hz) of the signal.
- **snr** (*float*) – The signal-to-noise ratio (SNR) in decibels.
- **noise_frequency** (*float*) – The frequency (Hz) of the powerline noise.

Returns
The signal with added powerline noise.

Return type
np.ndarray

qumphy.data.signal_preprocessing.resampling module

File: qumphy/data/signal_preprocessing/resampling.py Project: 22HLT01 QUMPHY Contact: oskar.pfeffer@ptb.de Gitlab: https://gitlab.com/qumphy Description: Functions for resampling the rate of PPG signals.

class qumphy.data.signal_preprocessing.resampling.**MatlabResampleConfig**(*n=10*, *beta=5.0*, *padtype='constant'*, *cval=0.0*)

Bases: `object`

Parameters chosen to mimic MATLAB's resample defaults:

- Kaiser beta default is 5
- n default is 10 (controls filter length in MATLAB)

beta: `float = 5.0`

cval: `float = 0.0`

n: `int = 10`

padtype: `str = 'constant'`

qumphy.data.signal_preprocessing.resampling.**resample_like_matlab**(*x*, *fs_in*, *fs_out*, ***, *axis=-1*, *cfg=MatlabResampleConfig(n=10, beta=5.0, padtype='constant', cval=0.0)*, *max_denominator=10000*)

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

MATLAB-like resampling:

- polyphase FIR lowpass
- Kaiser window (beta default 5)
- zero padding beyond edges (padtype='constant', cval=0)

Submodules

qumphy.data.attractor module

File: qumphy/data/attractor.py Project: 22HLT01 QUMPHY Contact: oskar.pfeffer@ptb.de Gitlab: https://gitlab.com/qumphy Description: Attractor image dataset and lightning data module .

class qumphy.data.attractor.**AttractorDataModule**(*data_directory*, *batch_size*, *num_workers*)

Bases: `LightningDataModule`

LightningDataModule implementation for the DeepBeat dataset.

setup(*stage*)

Called at the beginning of fit (train + validate), validate, test, or predict. This is a good hook when you need to build models dynamically or adjust something about them. This hook is called on every process when using DDP.

Parameters
stage – either `'fit'`, `'validate'`, `'test'`, or `'predict'`

Example:

```
class LitModel(...):
    def __init__(self):
        self.l1 = None

    def prepare_data(self):
        download_data()
        tokenize()

        # don't do this
        self.something = else

    def setup(self, stage):
        data = load_data(...)
        self.l1 = nn.Linear(28, data.num_classes)
```

test_dataloader()

Create the test dataloader.

Returns
Dataloader for the test dataset.

Return type
torch.utils.data.DataLoader

train_dataloader()

Create the training dataloader.

Returns
Dataloader for the training dataset.

Return type
torch.utils.data.DataLoader

val_dataloader()

Create the validation dataloader.

Returns
Dataloader for the validation dataset.

Return type
torch.utils.data.DataLoader

class qumphy.data.attractor.**AttractorDataset**(*data_directory*, *subset*)

Bases: Dataset

qumphy.data.deepbeat module

File: qumphy/data/pulsedb.py Project: 22HLT01 QUMPHY Contact: oskar.pfeffer@ptb.de Gitlab: https://gitlab.com/qumphy Description: Functions handling DeepBeat dataset.

class qumphy.data.deepbeat.**DeepBeatDataModule**(*sampling_rate*, *batch_size*, *num_workers*, *pin_memory=True*, *prefetch_factor=8*, ***dskwargs*)

Bases: LightningDataModule

LightningDataModule implementation for the DeepBeat dataset.

setup(*stage*)

test_dataloader()

Create the test dataloader.

Returns
Dataloader for the test dataset.

Return type
torch.utils.data.DataLoader

train_dataloader()

Create the training dataloader.

Returns
Dataloader for the training dataset.

Return type
torch.utils.data.DataLoader

val_dataloader()

Create the validation dataloader.

Returns
Dataloader for the validation dataset.

Return type
torch.utils.data.DataLoader

class qumphy.data.deepbeat.**DeepBeatDataset**(*data_directory*, *subset*, *dataset='set_revised'*, *target_format='binary'*, *normalize=False*, *dtype=torch.float32*, *load_data=True*, *data_fraction=1.0*, *filter_params=None*, *noise_params=None*, *input_sampling_rate=None*, *split_to_input_sampling_rate=None*, *target_sampling_rate=None*)

Bases: Dataset

DeepBeat dataset class.

get_data()

get_labels()

load_data(*data_directory*, *filter_params=None*, *noise_params=None*, *data_fraction=1.0*)

normalize_data(*data*)

Rescales the data to the range [-1, 1].

Parameters
(array) (*data*)

Returns
array: The normalized data in the range [-1, 1].

select_subset_indices(*metadata*, *data_fraction=1.0*)

Set an index mask of the memmapped dataset based on the selected subset.

Return type
Index

Parameters
metadata (*pd.core.frame.DataFrame*) – The metadata dataframe.

Returns
The index mask.

Return type
pd.core.indexes.base.Index

qumphy.data.pulsedb module

File: qumphy/data/pulsedb.py Project: 22HLT01 QUMPHY Contact: oskar.pfeffer@ptb.de Gitlab: https://gitlab.com/qumphy Description: Functions handling PulseDB data.

class qumphy.data.pulsedb.**PulseDBDataModule**(*data_directory*, *dataset*, *source*, *batch_size*, *num_workers*, *sampling_rate*, *prefetch_factor=8*, ***dskwargs*)

Bases: LightningDataModule

LightningDataModule implementation for the PulseDB dataset.

get_target_stats()

setup(*stage*)

test_dataloader()

train_dataloader()

val_dataloader()

class qumphy.data.pulsedb.**PulseDBDataset**(*data_directory*, *dataset*, *subset*, *source='all'*, *pressure='both'*, *normalize=False*, *dtype=torch.float32*, *load_data=True*, *data_fraction=1.0*, *filter_params=None*, *noise_params=None*, *input_sampling_rate=None*, *split_to_input_sampling_rate=None*, *target_sampling_rate=None*)

Bases: Dataset

Dataset Class for the PulseDB dataset.

To use the PulseDB Dataset, the data should be in a directory containing signals.npy and metadata.csv. Before using the dataset, run the *write_target_stats_yaml(data_directory)* function to create a stats.yaml file containint the statistics of the target, i.e., the mean, median, std, and baseline measures of the SBP and DBP.

calculate_baseline_measures()

calculate_target_stats()

get_data()

get_labels()

get_target_stats()
Returns the statistics of the target.

Returns
(BP_mean, BP_std, BP_median, MAE_baseline, RMSE_baseline)

Return type
tuple

load_data(*data_directory*, *filter_params=None*, *noise_params=None*)

load_target_stats(*data_directory*)

Reads the target statistics of the PulseDB training datasets from a YAML file.

Return type
None

Parameters
data_directory (*pathlib.Path*) – The directory where the data is located.

Return type
None

normalize_data(*data*)

Rescales the data to the range [-1, 1].

Parameters
(array) (*data*)

Returns
array: The normalized data in the range [-1, 1].

normalize_target(*target*, *denormalize=False*)

Rescales the target to 0 mean and 1 standard deviation, using the BP_mean and BP_std attributes.

Parameters

- **target** (*np.ndarray*) – The input target to be normalized.
- **denormalize** (*bool, optional*) – If True, the target will be denormalized back to its original scale.

Returns
The (de)normalized target.

Return type
np.ndarray

select_subset_indices(*metadata*)

Return an index mask of the memmapped dataset based on the selected subset.

Return type
Index

Parameters
metadata (*pd.core.frame.DataFrame*) – The metadata dataframe.

Returns
The index mask.

Return type
pd.core.indexes.base.Index

qumphy.data.pulsedb.**get_target_stats**(*data_directory*, *dataset*, *source*, *dtype=numpy.float16*)

Reads the target statistics of the PulseDB training datasets from a YAML file.

Return type
dict

Parameters

- **data_directory** (*string*) – Full path of the directory containing the data.

- **dataset** (*string*) – Choose between the “calibfree”, “calib”, “aami” or “mini” dataset.
- **source** (*string*) – Either “mimiciii” or “vital” or “all” (both).
- **dtype** (*np.dtype*) – The data type of the target statistics.

Returns
The target statistics as a dictionary.

Return type
dict

qumphy.data.pulsedb.**write_target_stats_yaml**(*data_directory*, *verbose=True*)

Writes the target statistics of the PulseDB training datasets to a YAML file.

Parameters
data_directory (*string*) – The directory where the data is located.

Returns
None

qumphy.data.sleepapnea module

File: qumphy/data/signal_preprocessing/noise.py Project: 22HLT01 QUMPHY Contact: oskar.pfeffer@ptb.de Gitlab: https://gitlab.com/qumphy Description: Functions handling SleepApnea data.

class qumphy.data.sleepapnea.**SleepApneaDataModule**(*sampling_rate*, *batch_size*, *num_workers*, *pin_memory=True*, *prefetch_factor=8*, ***dskwargs*)

Bases: LightningDataModule

LightningDataModule implementation for the SleepApnea dataset.

setup(*stage=None*)

test_dataloader()

train_dataloader()

val_dataloader()

class qumphy.data.sleepapnea.**SleepApneaDataset**(*data_directory*, *split*, *normalize=False*, *dtype=torch.float32*, *data_fraction=1.0*, *filter_params=None*, *noise_params=None*, *input_sampling_rate=None*, *split_to_input_sampling_rate=None*, *target_sampling_rate=None*)

Bases: Dataset

SleepApnea dataset class.

get_data()

get_labels()

normalize_data(*data*)

Rescales the data to the range [-1, 1].

Parameters
(array) (*data*)

Returns
array: The normalized data in the range [-1, 1].

```
split_to_data_key = {'test_ID': 'signalsTest_ID', 'test_OOD': 'signalsTest_OOD',
'train':  'signalsTrain', 'val':  'signalsValidation'}

split_to_label_key = {'test_ID': 'labelsTest_ID', 'test_OOD': 'labelsTest_OOD',
'train':  'labelsTrain', 'val':  'labelsValidation'}
```

qumphy.data.utils module

File: qumphy/data/utils.py Project: 22HLT01 QUMPHY Contact: nando.hegemann@ptb.de Gitlab: https://gitlab.com/qumphy Description: Loading functions for various datasets.

qumphy.data.utils.**calculate_regression_baseline**(*dataset*, *median*, *mean*)

Calculate regression baseline metrics using the median and mean of the dataset.

Return type
tuple[float, float, float, float]

Parameters

- **dataset** (*Class*) – Dataset object.
- **median** (*np.ndarray | float*) – Median of the dataset.
- **mean** (*np.ndarray | float*) – Mean of the dataset.

Returns
Baseline metrics.

Return type
tuple[float, float, float, float]

3.5.1.3 qumphy.evaluate package

File: qumphy/__init__.py Project: 22HLT01 QUMPHY Contact: nando.hegemann@ptb.de Gitlab: https://gitlab.com/qumphy Description: init

Submodules

qumphy.evaluate.deepensembles module

File: qumphy/uq/deepensembles.py Project: 22HLT01 QUMPHY Contact: oskar.pfeffer@ptb.de Gitlab: https://gitlab.com/qumphy Description: DeepEnsemble Trainer.

class qumphy.evaluate.deepensembles.**DeepBeatEvaluation**

Bases: object

Evaluation utilities for DeepBeat ensemble predictions.

evaluation_function(*target*, *predictions*, *ensemble_predictions*)

Evaluate individual DeepBeat models and ensemble predictions.

Parameters

- **target** (*np.ndarray*) – Ground truth target values.
- **predictions** (*np.ndarray*) – Predictions from individual ensemble members.
- **ensemble_predictions** (*np.ndarray*) – Aggregated ensemble predictions.

Returns
The function prints the calculated metrics.

Return type
None

print_metrics(*mean_metrics*, *max_metrics*, *min_metrics*, *ensemble_metrics*)

reduce(*predictions*)

mean of predictions

class qumphy.evaluate.deepensembles.**DeepEnsembleEvaluate**(*config*)

Bases: object

Evaluation pipeline for deep ensemble predictions.

calculate_metrics()

load_dataset()

Load the dataset from the configuration.

Returns
The function stores the dataset and target values as attributes.

Return type
None

load_evaluation_class()

Load the evaluation class from the configuration.

Returns
The function stores the evaluation class as an attribute.

Return type
None

load_predictions()

Load and process prediction files.

Returns
The function loads predictions, optionally applies extra processing and denormalization, and stores individual and ensemble predictions.

Return type
None

class qumphy.evaluate.deepensembles.**PulseDBEvaluation**

Bases: object

A class for evaluating PulseDB models and ensembles.

Contains methods for evaluating individual models and ensembles, as well as methods for printing and saving the results.

denormalize(*dataset*, *predictions*)

Denormalize PulseDB predictions.

Parameters

- **dataset** (*object*) – Dataset object containing BP_mean and BP_std attributes.
- **predictions** (*np.ndarray*) – Normalized predictions containing mean and standard deviation values.

Returns
Denormalized predictions.

Return type
np.ndarray

evaluation_function(*target*, *predictions*, *ensemble_predictions*)

Evaluate individual PulseDB models and ensemble predictions.

Parameters

- **target** (*np.ndarray*) – Ground truth blood pressure targets.
- **predictions** (*np.ndarray*) – Predictions from individual ensemble members.
- **ensemble_predictions** (*np.ndarray*) – Aggregated ensemble predictions.

Returns
The function prints the calculated metrics.

Return type
None

extra_function(*predictions*)

Convert predicted log-variances to standard deviations.

Parameters
predictions (*np.ndarray*) – Prediction array containing mean values and log-variance values.

Returns
Prediction array containing mean values and standard deviation values.

Return type
np.ndarray

print_metrics(*mean_metrics*, *max_metrics*, *min_metrics*, *ensemble_metrics*)

reduce(*predictions*)

GAUSSIAN MIXTURE AS IN LAKSMINARAYANAN PAPER The predictions are given as mean and std and returned the same way. :param predictions: Predictions from individual ensemble members. :type predictions: np.ndarray

Returns
Aggregated ensemble prediction.

Return type
np.ndarray

qumphy.evaluate.deepensembles.**append_nested_dicts**(*dicts_list*)

Take a list of nested dictionaries and create a single dictionary with the values replaced by arrays of the values of the individual dictionaries. The dictionaries must have the same structure.

qumphy.evaluate.deepensembles.**reduce_nested_dict_list**(*dicts_list*, *reduction_function*)

Take a list of nested dictionaries and reduce them to a single dictionary using the given reduction function. The dictionaries must have the same structure.

Parameters

- **dicts_list** (*list*) – List of nested dictionaries.
- **reduction_function** (*function, optional*) – Function to use for reduction.

Returns
Reduced dictionary.

Return type
dict

3.5.1.4 qumphy.misc package

File: qumphy/__init__.py Project: 22HLT01 QUMPHY Contact: nando.hegemann@ptb.de Gitlab: https://gitlab.com/qumphy Description: init

Submodules

qumphy.misc.misc module

File: qumphy/misc.py Project: 22HLT01 QUMPHY Contact: nando.hegemann@ptb.de Gitlab: https://gitlab.com/qumphy Description: Miscellaneous functions.

qumphy.misc.misc.**batch**(*iterable*, *n=1*)

Split iterable into different batches of batchsize n.

Return type
Iterator

Parameters

- **iterable** (*array_like*) – Iterable to split.
- **n** (*int, default=1*) – Batch size.

Yields
Iterable for different batches.

qumphy.misc.misc.**eval_argument_parser**()

qumphy.misc.misc.**eval_torch_model_by_numpy_ndarray**(*model*, *data*)

Evaluate a torch model (nn.Module) with a numpy ndarray.

Return type
ndarray

Parameters

- **model** (*nn.Module*) – Torch model.
- **data** (*np.ndarray*) – Input data.

Returns
Model output predictions.

Return type
np.ndarray

qumphy.misc.misc.**instantiate_class**(*config*)

Instantiate a class from a dictionary config.

The config dictionary should have a “class_path” key with the path to the class (e.g., ‘my_module.MyClass’). It should also have an “init_args” key with a dictionary of arguments to pass to the class constructor.

If the config dictionary has a “classes” key, the function will recursively instantiate the classes specified in the list and pass them as arguments to the class constructor. The “classes” key should be a list of dictionaries, each with a “class_path” key and a “keyword” key specifying the keyword argument to pass the class instance to. If the “class_list” key is present, a list of classes will be instantiated and passed to the same keyword argument.

Example: >>> config = { … “class_path”: “my_module.Class1”, … “init_args”: {“first_arg”: 1, “second_arg”: 2}, … “classes”: [… {“keyword”: “third_arg”, “class_path”: “my_module.Class2”, “init_args”: {“x”: 3}}, … {“keyword”: “fourth_arg”, “class_list”: [… {“class_path”: “my_module.Class3”, “init_args”: {“y”: 4}}, … {“class_path”: “my_module.Class4”, “init_args”: {“z”: 5}} …] …] … }

Return type
object

Parameters
config (*dict*) – A dictionary with the class path and arguments to pass to the class constructor.

Returns
An instance of the specified class.

Return type
object

qumphy.misc.misc.**instantiate_class_from_string**(*class_path*, **init_args*, ***init_kwargs*)

Instantiate a class from a given string path.

Parameters

- **class_path** (*str*) – The full path to the class (e.g., ‘my_module.MyClass’).
- ***init_args** – Arguments to pass to the class constructor.
- ****init_kwargs** – Keyword arguments to pass to the class constructor.

Returns
An instance of the specified class.

Return type
object

qumphy.misc.misc.**parse_value**(*value*)

Try parsing a string as a boolean, integer, or float. If parsing fails, return the original string.

Parameters
value (*str*) – The string to parse.

Returns
The parsed value, or the original string if parsing fails.

Return type
object

qumphy.misc.misc.**set_value_at_nested_key**(*d*, *path*, *value*)

qumphy.misc.misc.**str2dict**(*text*)

Convert a string of the format “a.b.c:value” or “a.b.c=value” into a nested dictionary.

The value part of the string is parsed as a boolean, integer, float, or string.

Parameters
text (*str*) – The string to convert.

Returns
A nested dictionary where the keys are the parts of the string separated by ‘.’, and the value is separated by ‘:’.

Return type
dict

Examples

```
>>> str2dict("a.b.c:1")
{'a': {'b': {'c': '1'}}}
>>> str2dict("x.y.z:foo")
{'x': {'y': {'z': 'foo'}}}
```

qumphy.misc.misc.**train_argument_parser**()

qumphy.misc.misc.**update_dictionary**(*d1*, *d2*)

qumphy.misc.output_conversions module

File: qumphy/output_conversions.py Project: 22HLT01 QUMPHY Contact: vivek.desai@npl.co.uk Gitlab: https://gitlab.com/qumphy Description: Model output conversion functions.

qumphy.misc.output_conversions.**convert_prediction_intervals**(*intervals*, *converter*, *confidence_level=0.95*)

Convert a list of prediction intervals to distributions, giving a list of distributions, predictions, and uncertainties as variances.

Return type
tuple

Parameters:

intervals (np.ndarray):
Prediction interval outputs from a model. Expected shape of (n_samples, 2) for [lower_bound, upper_bound] pairs.

converter (Callable):
Conversion function used to convert intervals, given as lower and upper bounds, to distributions. Means and variances for the distributions are also returned.

confidence_level (float, optional):
The confidence level at which conformal prediction was evaluated. Used in the sampling process for converting the interval to a distribution. Defaults to 0.95.

Returns:

: distributions (List[distribution objects]):

List of distribution objects.

predictions (List[float]):
List of predictions, which are the means of the distributions, and should be asymptotically equivalent to the midpoint of the prediction intervals.

uncertainties (List[float]):
List of uncertainties, given as variances of the distributions.

qumphy.misc.output_conversions.**intervals_to_probs**(*intervals*, *method='jaccard'*)

Convert prediction intervals to probabilities by taking the mean of lower and upper bounds.

Return type
ndarray

Parameters:

intervals (np.ndarray): Prediction intervals with shape (n_samples, n_classes, 2), where last dimension represents [lower_bound, upper_bound].

Returns:

:

probs (np.ndarray): Probabilities derived from interval means

qumphy.misc.output_conversions.**kde_convert_interval**(*lower_bound*, *upper_bound*, *num_samples=10000*, *confidence_level=0.95*, *integrate=False*)

Convert a prediction interval to a distribution object using Kernel Density Estimation (KDE).

Return type
`tuple`

References:

KDE documentation for statsmodel package can be found here: <https://www.statsmodels.org/dev/generated/statsmodels.nonparametric.kde.KDEUnivariate.html>.

Documentation for scipy quad integration method can be found here: <https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.quad.html>.

Parameters:

lower_bound (float):
Lower bound of the prediction interval.

upper_bound (float):
Upper bound of the prediction interval.

num_samples (int, optional):
Number of samples to generate between the interval bounds. Defaults to 10000.

confidence_level (float, optional):
Confidence level associated with prediction interval. Defaults to 0.95.

integrate (bool, optional):
Option to evaluate mean and standard deviation using numerical integration instead of discrete summation. Defaults to False.

Returns:

:

mean (float):
The mean of the output distribution.

var (float):
The variance of the output distribution - used as the uncertainty.

kde (statsmodel.nonparametric.KDEUnivariate object):
The output distribution approximated using KDE.

qumphy.misc.output_conversions.**norm_convert_interval**(*lower_bound*, *upper_bound*, *confidence_level=0.95*)

Assuming the quantiles are from a Gaussian distribution, convert to a full distribution object. Return the distribution object, as well as the mean and standard deviation.

Parameters:

lower_bound (float): Lower quantile. upper_bound (float): Upper quantile. confidence_level (float, optional): Confidence level associated with the quantiles. Defaults to 0.95.

Returns:

:

mean (float): The mean of the normal distribution. var (float): The variance of the normal distribution. distribution (scipy.stats.norm object): The full scipy.stats.norm distribution object, which has useful attributes e.g. CDF.

qumphy.misc.output_conversions.**probs_to_pred_sets**(*probabilities*, *alpha*)

Wrapper function to convert probabilities to prediction sets given an array of model confidences per class.

Return type
ndarray

Parameters:

probabilities (np.ndarray): Array of model outputs as probabilities for each class. alpha (float): Defines the confidence/coverage level at which to set threshold for top-k selection.

Returns:

:

np.ndarry: Array of lists, where each list is a prediction set determined through top-k selection at 1-alpha confidence level.

3.5.1.5 qumphy.models package

File: qumphy/models/__init__.py Project: 22HLT01 QUMPHY Contact: oskar.pfeffer@ptb.de Gitlab: https://gitlab.com/qumphy Description: init

Subpackages

qumphy.models.utils package

File: qumphy/models/utils/__init__.py Project: 22HLT01 QUMPHY Contact: oskar.pfeffer@ptb.de Gitlab: https://gitlab.com/qumphy Description: init

Submodules

qumphy.models.utils.conv_blocks module

class qumphy.models.utils.conv_blocks.**Inception_Block_V1**(**args: Any*, ***kwargs: Any*)

Bases: Module

forward(*x*)

class qumphy.models.utils.conv_blocks.**Inception_Block_V2**(**args: Any*, ***kwargs: Any*)

Bases: Module

forward(*x*)

qumphy.models.utils.embed module

class qumphy.models.utils.embed.**DataEmbedding**(**args: Any*, ***kwargs: Any*)

Bases: Module

forward(*x*, *x_mark*)

class qumphy.models.utils.embed.**DataEmbedding_inverted**(**args: Any*, ***kwargs: Any*)

Bases: Module

forward(*x*, *x_mark*)

class qumphy.models.utils.embed.**DataEmbedding_wo_pos**(**args: Any*, ***kwargs: Any*)

Bases: Module

forward(*x*, *x_mark*)

class qumphy.models.utils.embed.**FixedEmbedding**(**args: Any*, ***kwargs: Any*)

Bases: Module

forward(*x*)

class qumphy.models.utils.embed.**PatchEmbedding**(**args: Any*, ***kwargs: Any*)

Bases: Module

forward(*x*)

class qumphy.models.utils.embed.**PositionalEmbedding**(**args: Any*, ***kwargs: Any*)

Bases: Module

forward(*x*)

class qumphy.models.utils.embed.**TemporalEmbedding**(**args: Any*, ***kwargs: Any*)

Bases: Module

forward(*x*)

class qumphy.models.utils.embed.**TimeFeatureEmbedding**(**args: Any*, ***kwargs: Any*)

Bases: Module

forward(*x*)

class qumphy.models.utils.embed.**TokenEmbedding**(**args: Any*, ***kwargs: Any*)

Bases: Module

forward(*x*)

qumphy.models.utils.kgactivation module

class qumphy.models.utils.kgactivation.**KGActivation**(**args: Any*, ***kwargs: Any*)

Bases: Module

Activation neeed to use KGLoss Equation 12 from https://arxiv.org/abs/1703.04977 Used to model aleatoric uncertainty for classification tasks

forward(*x*)

qumphy.models.utils.kgloss module

class qumphy.models.utils.kgloss.**KGLoss**(**args: Any*, ***kwargs: Any*)

Bases: Module

Equation 12 from https://arxiv.org/abs/1703.04977 Models aleatoric uncertainty for classification tasks

forward(*logits*, *noise*, *target*)

Compute the loss for aleatoric uncertainty.

Parameters

- **logits** (*torch.Tensor*) – The output of the model before softmax
- **noise** (*torch.Tensor*) – The output of the model for the aleatoric uncertainty
- **target** (*torch.Tensor*) – The target of the model

Returns
loss – The loss of the model

Return type
torch.Tensor

class qumphy.models.utils.kgloss.**KGLoss_unified_prediction**(**args: Any*, ***kwargs: Any*)

Bases: *KGLoss*

Equation 12 from https://arxiv.org/abs/1703.04977 Models aleatoric uncertainty for classification tasks

forward(*prediction*, *target*)

Compute the loss for aleatoric uncertainty.

Parameters

- **logits** (*torch.Tensor*) – The output of the model before softmax
- **noise** (*torch.Tensor*) – The output of the model for the aleatoric uncertainty
- **target** (*torch.Tensor*) – The target of the model

Returns
loss – The loss of the model

Return type
torch.Tensor

qumphy.models.utils.masking module

class qumphy.models.utils.masking.**ProbMask**(*B*, *H*, *L*, *index*, *scores*, *device='cpu'*)

Bases: object

property mask

class qumphy.models.utils.masking.**TriangularCausalMask**(*B*, *L*, *device='cpu'*)

Bases: object

property mask

qumphy.models.utils.mcdropout module

class qumphy.models.utils.mcdropout.**MCDropout**(*p=0.5*, *mcdropout=False*, *inplace=False*)

Bases: Dropout

Implementation of MCDropout to the torch dropout layer. This adds a mcdropout flag, which turns dropout always on in training and evaluation mode.

Parameters

- **p** (*float*) – probability of an element to be zeroed. Default: 0.5
- **mcdropout** (*bool*) – if True, will always perform dropout
- **inplace** (*bool*) – if True, will perform dropout in-place

forward(*input*)

Return type
Tensor

qumphy.models.utils.pinballloss module

class qumphy.models.utils.pinballloss.**PinballLoss**(**args: Any*, ***kwargs: Any*)

Bases: Module

Calculates the quantile loss function.

self.**quantiles**

Type
torch.tensor

forward(*prediction*, *target*)

Computes the loss for the given prediction.

qumphy.models.utils.selfattention_family module

class qumphy.models.utils.selfattention_family.**AttentionLayer**(**args: Any*, ***kwargs: Any*)

Bases: Module

forward(*queries*, *keys*, *values*, *attn_mask*, *tau=None*, *delta=None*)

class qumphy.models.utils.selfattention_family.**FlashAttention**(**args: Any*, ***kwargs: Any*)

Bases: Module

flash_attention_forward(*Q*, *K*, *V*, *mask=None*)

forward(*queries*, *keys*, *values*, *attn_mask*, *tau=None*, *delta=None*)

class qumphy.models.utils.selfattention_family.**FlowAttention**(**args: Any*, ***kwargs: Any*)

Bases: Module

forward(*queries*, *keys*, *values*, *attn_mask*, *tau=None*, *delta=None*)

kernel_method(*x*)

class qumphy.models.utils.selfattention_family.**FullAttention**(**args: Any*, ***kwargs: Any*)

Bases: Module

forward(*queries*, *keys*, *values*, *attn_mask*, *tau=None*, *delta=None*)

class qumphy.models.utils.selfattention_family.**ProbAttention**(**args: Any*, ***kwargs: Any*)

Bases: Module

forward(*queries*, *keys*, *values*, *attn_mask*, *tau=None*, *delta=None*)

class qumphy.models.utils.selfattention_family.**ReformerLayer**(**args: Any*, ***kwargs: Any*)

Bases: Module

fit_length(*queries*)

forward(*queries*, *keys*, *values*, *attn_mask*, *tau*, *delta*)

qumphy.models.utils.transformer_encdec module

class qumphy.models.utils.transformer_encdec.**ConvLayer**(**args: Any*, ***kwargs: Any*)

Bases: Module

forward(*x*)

class qumphy.models.utils.transformer_encdec.**Decoder**(**args: Any*, ***kwargs: Any*)

Bases: Module

forward(*x*, *cross*, *x_mask=None*, *cross_mask=None*, *tau=None*, *delta=None*)

class qumphy.models.utils.transformer_encdec.**DecoderLayer**(**args: Any*, ***kwargs: Any*)

Bases: Module

forward(*x*, *cross*, *x_mask=None*, *cross_mask=None*, *tau=None*, *delta=None*)

class qumphy.models.utils.transformer_encdec.**Encoder**(**args: Any*, ***kwargs: Any*)

Bases: Module

forward(*x*, *attn_mask=None*, *tau=None*, *delta=None*)

class qumphy.models.utils.transformer_encdec.**EncoderLayer**(**args: Any*, ***kwargs: Any*)

Bases: Module

forward(*x*, *attn_mask=None*, *tau=None*, *delta=None*)

Submodules

qumphy.models.alexnet module

File: qumphy/models/alexnet.py Project: 22HLT01 QUMPHY Contact: oskar.pfeffer@ptb.de Gitlab: https://gitlab.com/qumphy Description: Alexnet pytorch lightning model.

class qumphy.models.alexnet.**AlexNet1D**(**args: Any*, ***kwargs: Any*)

Bases: Module

General 1D AlexNet model implementation.

The input_size and output size can be chosen freely. Minimum input_size is 67.

The input shape is (batch_size, 1, input_size) and the output shape is (batch_size, output_size).

Activation function for the last layer can be chosen freely.

forward(*x*)

Run a forward pass through the model.

Parameters
x (*torch.Tensor*) – Input tensor of shape (batch_size, 1, input_size).

Returns
Model output tensor of shape (batch_size, output_size).

Return type
torch.Tensor

class qumphy.models.alexnet.**AlexNet1D_MCD**(**args: Any*, ***kwargs: Any*)

Bases: Module

General 1D AlexNet model implementation with Monte Carlo dropout.

The input_size and output size can be chosen freely. Minimum input_size is 67.

The input shape is (batch_size, 1, input_size) and the output shape is (batch_size, output_size).

Activation function for the last layer can be chosen freely.

forward(*x*)

Run a forward pass through the model.

Parameters
x (*torch.Tensor*) – Input tensor of shape (batch_size, 1, input_size).

Returns
Model output tensor of shape (batch_size, output_size).

Return type
torch.Tensor

qumphy.models.apnea module

Description: Lightning model for SleepApnea data.

class qumphy.models.apnea.**SleepApneaModule**(**args: Any*, ***kwargs: Any*)

Bases: LightningModule

Lightning parent module for DeepBeat data. Takes a specific model architecture as input. (net)

configure_optimizers()

forward(*data*)

predict_step(*batch*, *batch_idx*)

set_lr_scheduler()

test_step(*batch*, *batch_idx*)

training_step(*batch*, *batch_idx*)

validation_step(*batch*, *batch_idx*)

qumphy.models.basic_conv1d module

File: qumphy/models/basic_conv1d.py Project: 22HLT01 QUMPHY Contact: oskar.pfeffer@ptb.de Gitlab: https://gitlab.com/qumphy Description: Convolutional Neural Network.

class qumphy.models.basic_conv1d.**AdaptiveConcatPool1d**(**args: Any*, ***kwargs: Any*)

Bases: Module

Concatenate adaptive average pooling and adaptive max pooling.

Parameters
sz (*int*) – Output size of each pooling layer. If None, the output size is 1.

forward(*x*)

Run a forward pass through the adaptive concatenated pooling layer.

Parameters
x (*torch.Tensor*) – Input tensor.

Returns
Tensor obtained by concatenating max-pooled and average-pooled features along the channel dimension.

Return type
torch.Tensor

class qumphy.models.basic_conv1d.**Basic_Conv1d**(**args: Any*, ***kwargs: Any*)

Bases: Sequential

Basic configurable 1D convolutional neural network.

Parameters

- **filters** (*list*) – List containing the number of filters for each convolutional layer.
- **kernel_size** (*int or list*) – Kernel size used for the convolutional layers.
- **stride** (*int*) – Stride used for the convolutional layers.
- **dilation** (*int*) – Dilation factor used for the convolutional layers.
- **pool** (*int*) – Max pooling kernel size. If 0, no max pooling is applied.
- **pool_stride** (*int*) – Stride used for max pooling.
- **squeeze_excite_reduction** (*int*) – Reduction factor for squeeze-and-excitation blocks. If 0, no squeeze-and-excitation block is applied.
- **num_classes** (*int*) – Number of output classes or output values.
- **input_channels** (*int*) – Number of input channels.
- **act** (*str*) – Activation function used in the convolutional blocks.
- **bn** (*bool*) – If True, use batch normalization in convolutional blocks.
- **headless** (*bool*) – If True, remove the final prediction head and return extracted features.
- **split_first_layer** (*bool*) – If True, split the first convolutional layer into two convolutional steps.
- **drop_p** (*float*) – Dropout probability used in convolutional blocks.
- **lin_ftrs_head** (*list*) – Hidden layer sizes used in the model head.

- **ps_head** (*float or Iterable*) – Dropout probability or probabilities used in the model head.
- **bn_final_head** (*bool*) – Indicates whether final batch normalization should be used in the head.
- **bn_head** (*bool*) – If True, use batch normalization in the model head.
- **act_head** (*str*) – Activation function used in the model head.
- **concat_pooling** (*bool*) – If True, use concatenated adaptive average and max pooling in the head.

get_layer_groups()

Get layer groups of the model.

Returns
Tuple containing selected convolutional layers and the model head.

Return type
tuple

get_output_layer()

Get the output layer of the model.

Returns
Output layer if the model is not headless. Otherwise, None.

Return type
nn.Module or None

set_output_layer(*x*)

Set the output layer of the model.

Parameters
x (*nn.Module*) – New output layer.

Returns
The function modifies the output layer in place.

Return type
None

class qumphy.models.basic_conv1d.**Flatten**(*full=False*)

Bases: Module

Flatten an input tensor to a single dimension.

Parameters
full (*bool*) – If True, flatten the complete tensor into a rank-1 tensor. If False, keep the batch dimension and flatten all remaining dimensions.

forward(*x*)

Run a forward pass through the flatten layer.

Parameters
x (*torch.Tensor*) – Input tensor to be flattened.

Returns
Flattened tensor. If full is True, the output has shape (-1,). Otherwise, the output has shape (batch_size, -1).

Return type
torch.Tensor

class qumphy.models.basic_conv1d.**LambdaLayer**(**args: Any*, ***kwargs: Any*)

Bases: Module

Layer that applies a given lambda function to the input.

Parameters
lambd (*callable*) – Function applied to the input tensor.

forward(*x*)

Run a forward pass through the lambda layer.

Parameters
x (*torch.Tensor*) – Input tensor.

Returns
Output tensor after applying the lambda function.

Return type
torch.Tensor

class qumphy.models.basic_conv1d.**SqueezeExcite1d**(**args: Any*, ***kwargs: Any*)

Bases: Module

Squeeze-and-excitation block for 1D inputs.

Parameters

- **channels** (*int*) – Number of input channels.
- **reduction** (*int*) – Reduction factor used to compute the number of hidden channels.

forward(*x*)

Run a forward pass through the squeeze-and-excitation block.

Parameters
x (*torch.Tensor*) – Input tensor of shape (batch_size, channels, sequence_length).

Returns
Reweighted tensor with the same shape as the input.

Return type
torch.Tensor

qumphy.models.basic_conv1d.**basic1d**(*filters=[128, 128, 128, 128, 128]*, *kernel_size=3*, *stride=2*, *dilation=1*, *pool=0*, *pool_stride=1*, *squeeze_excite_reduction=0*, *num_classes=2*, *input_channels=8*, *act='relu'*, *bn=True*, *headless=False*, *drop_p=0.0*, *lin_ftrs_head=None*, *ps_head=0.5*, *bn_final_head=False*, *bn_head=True*, *act_head='relu'*, *concat_pooling=True*, ***kwargs*)

Create a basic configurable 1D convolutional network.

Parameters

- **filters** (*list*) – List containing the number of filters for each convolutional layer.
- **kernel_size** (*int or list*) – Kernel size used for the convolutional layers.
- **stride** (*int*) – Stride used for the convolutional layers.
- **dilation** (*int*) – Dilation factor used for the convolutional layers.
- **pool** (*int*) – Max pooling kernel size. If 0, no max pooling is applied.

- **pool_stride** (*int*) – Stride used for max pooling.
- **squeeze_excite_reduction** (*int*) – Reduction factor for squeeze-and-excitation blocks. If 0, no squeeze-and-excitation block is applied.
- **num_classes** (*int*) – Number of output classes or output values.
- **input_channels** (*int*) – Number of input channels.
- **act** (*str*) – Activation function used in the convolutional blocks.
- **bn** (*bool*) – If True, use batch normalization in convolutional blocks.
- **headless** (*bool*) – If True, remove the final prediction head and return extracted features.
- **drop_p** (*float*) – Dropout probability used in convolutional blocks.
- **lin_ftrs_head** (*list*) – Hidden layer sizes used in the model head.
- **ps_head** (*float or Iterable*) – Dropout probability or probabilities used in the model head.
- **bn_final_head** (*bool*) – Indicates whether final batch normalization should be used in the head.
- **bn_head** (*bool*) – If True, use batch normalization in the model head.
- **act_head** (*str*) – Activation function used in the model head.
- **concat_pooling** (*bool*) – If True, use concatenated adaptive average and max pooling in the head.
- ****kwargs** – Additional keyword arguments.

Returns
Basic configurable 1D convolutional model.

Return type
Basic_Conv1d

qumphy.models.basic_conv1d.**bn_drop_lin**(*n_in*, *n_out*, *bn=True*, *p=0.0*, *actn=None*, *layer_norm=False*, *permute=False*)

Create batch normalization, dropout, linear, and activation layers.

Parameters

- **n_in** (*int*) – Number of input features.
- **n_out** (*int*) – Number of output features.
- **bn** (*bool*) – If True, add a normalization layer before the linear layer.
- **p** (*float*) – Dropout probability.
- **actn** (*nn.Module*) – Activation function added after the linear layer.
- **layer_norm** (*bool*) – If True, use layer normalization instead of batch normalization.
- **permute** (*bool*) – If True, permute the input dimensions before and after normalization. This is useful for inputs of shape (batch_size, sequence, features).

Returns
List of PyTorch layers.

Return type
list

qumphy.models.basic_conv1d.**create_head1d**(*nf*, *nc*, *lin_ftrs=None*, *ps=0.5*, *bn=True*, *act='relu'*, *concat_pooling=True*)

Create a 1D model classification or regression head.

Parameters

- **nf** (*int*) – Number of input features.
- **nc** (*int*) – Number of output classes or output values.
- **lin_ftrs** (*list*) – List of hidden layer sizes used in the head.
- **ps** (*float or Iterable*) – Dropout probability or list of dropout probabilities.
- **bn** (*bool*) – If True, use batch normalization in the linear blocks.
- **act** (*str*) – Activation function used between linear layers.
- **concat_pooling** (*bool*) – If True, use concatenated adaptive average and max pooling. If False, use adaptive average pooling only.

Returns
Sequential model head.

Return type
nn.Sequential

qumphy.models.basic_conv1d.**fcn**(*filters=[128, 128, 128, 128, 128]*, *num_classes=2*, *input_channels=8*, ***kwargs*)

Create a fully convolutional 1D network.

Parameters

- **filters** (*list*) – List containing the number of filters for the convolutional layers.
- **num_classes** (*int*) – Number of output classes or output values.
- **input_channels** (*int*) – Number of input channels.
- ****kwargs** – Additional keyword arguments.

Returns
Fully convolutional 1D model.

Return type
Basic_Conv1d

qumphy.models.basic_conv1d.**fcn_wang**(*num_classes=2*, *input_channels=8*, *lin_ftrs_head=None*, *ps_head=0.5*, *bn_final_head=False*, *bn_head=True*, *act_head='relu'*, *concat_pooling=True*, ***kwargs*)

Create a Wang-style fully convolutional 1D network.

Parameters

- **num_classes** (*int*) – Number of output classes or output values.
- **input_channels** (*int*) – Number of input channels.
- **lin_ftrs_head** (*list*) – Hidden layer sizes used in the model head.
- **ps_head** (*float or Iterable*) – Dropout probability or probabilities used in the model head.
- **bn_final_head** (*bool*) – Indicates whether final batch normalization should be used in the head.

- **bn_head** (*bool*) – If True, use batch normalization in the model head.
- **act_head** (*str*) – Activation function used in the model head.
- **concat_pooling** (*bool*) – If True, use concatenated adaptive average and max pooling in the head.
- ****kwargs** – Additional keyword arguments.

Returns
Wang-style fully convolutional 1D model.

Return type
Basic_Conv1d

qumphy.models.basic_conv1d.**schirrmeister**(*num_classes=2*, *input_channels=8*, *kernel_size=10*, *lin_ftrs_head=None*, *ps_head=0.5*, *bn_final_head=False*, *bn_head=True*, *act_head='relu'*, *concat_pooling=True*, ***kwargs*)

Create a Schirrmeister-style 1D convolutional network.

Parameters

- **num_classes** (*int*) – Number of output classes or output values.
- **input_channels** (*int*) – Number of input channels.
- **kernel_size** (*int*) – Kernel size used in the convolutional layers.
- **lin_ftrs_head** (*list*) – Hidden layer sizes used in the model head.
- **ps_head** (*float or Iterable*) – Dropout probability or probabilities used in the model head.
- **bn_final_head** (*bool*) – Indicates whether final batch normalization should be used in the head.
- **bn_head** (*bool*) – If True, use batch normalization in the model head.
- **act_head** (*str*) – Activation function used in the model head.
- **concat_pooling** (*bool*) – If True, use concatenated adaptive average and max pooling in the head.
- ****kwargs** – Additional keyword arguments.

Returns
Schirrmeister-style 1D convolutional model.

Return type
Basic_Conv1d

qumphy.models.basic_conv1d.**sen**(*filters=[128, 128, 128, 128, 128]*, *num_classes=2*, *input_channels=8*, *kernel_size=3*, *squeeze_excite_reduction=16*, *drop_p=0.0*, *lin_ftrs_head=None*, *ps_head=0.5*, *bn_final_head=False*, *bn_head=True*, *act_head='relu'*, *concat_pooling=True*, ***kwargs*)

Create a squeeze-and-excitation 1D convolutional network.

Parameters

- **filters** (*list*) – List containing the number of filters for each convolutional layer.
- **num_classes** (*int*) – Number of output classes or output values.
- **input_channels** (*int*) – Number of input channels.

- **kernel_size** (*int or list*) – Kernel size used in the convolutional layers.
- **squeeze_excite_reduction** (*int*) – Reduction factor used in squeeze-and-excitation blocks.
- **drop_p** (*float*) – Dropout probability used in convolutional blocks.
- **lin_ftrs_head** (*list*) – Hidden layer sizes used in the model head.
- **ps_head** (*float or Iterable*) – Dropout probability or probabilities used in the model head.
- **bn_final_head** (*bool*) – Indicates whether final batch normalization should be used in the head.
- **bn_head** (*bool*) – If True, use batch normalization in the model head.
- **act_head** (*str*) – Activation function used in the model head.
- **concat_pooling** (*bool*) – If True, use concatenated adaptive average and max pooling in the head.
- ****kwargs** – Additional keyword arguments.

Returns
Squeeze-and-excitation 1D convolutional model.

Return type
Basic_Conv1d

qumphy.models.basic_conv1d.**weight_init**(*m*)

Initialize weights of supported model layers.

Parameters
m (*nn.Module*) – Module whose weights should be initialized.

Returns
The function modifies the module weights in place.

Return type
None

qumphy.models.cnn-gru module

qumphy.models.deep_ensemble module

File: qumphy/models/deep_ensemble.py Project: 22HLT01 QUMPHY Contact: oskar.pfeffer@ptb.de Gitlab: https://gitlab.com/qumphy Description: Lightning deep ensemble net integration.

class qumphy.models.deep_ensemble.**DeepEnsemble**(*net_config*, *ensemble_size*)

Bases: Module

Deep ensemble network.

Parameters

- **net_config** (*dict*) – Configuration dictionary used to instantiate each network in the ensemble. It should contain the keys “class_path” and “init_args”.
- **ensemble_size** (*int*) – Number of networks in the ensemble.

forward(*x*)

Run a forward pass through all networks in the ensemble.

Parameters
x (*torch.Tensor*) – Input tensor passed to each network in the ensemble.

Returns
Stacked outputs from all ensemble members. The outputs are stacked along the last dimension.

Return type
torch.Tensor

qumphy.models.deepbeat module

File: qumphy/models/deepbeat.py Project: 22HLT01 QUMPHY Contact: oskar.pfeffer@ptb.de Gitlab: https://gitlab.com/qumphy Description: Lightning model for DeepBeat data.

class qumphy.models.deepbeat.**DeepBeatEnsemble**(*noise_samples=100*, ***kwargs*)

Bases: *DeepBeatModule*

DeepBeat ensemble module.

This module combines predictions from multiple models and estimates uncertainty using noise-corrupted logits.

Parameters

- **noise_samples** (*int*) – Number of Gaussian noise samples used to corrupt logits.
- ****kwargs** – Keyword arguments passed to DeepBeatModule.

forward(*x*)

Run a forward pass through all models in the ensemble.

Parameters
x (*torch.Tensor*) – Input tensor passed to each model.

Returns
Stacked predictions from all ensemble models.

Return type
torch.Tensor

load_models(*model_list*)

Load models into the ensemble.

Parameters
model_list (*list*) – List of PyTorch models used as ensemble members.

Returns
The function stores the models as a ModuleList.

Return type
None

test_step(*batch*, *batch_idx*)

Run one ensemble test step. prediction has shape [model, batchsize, logits+noise]

Parameters

- **batch** (*tuple*) – Batch containing input data and target values.
- **batch_idx** (*int*) – Index of the current batch.

Returns
Dictionary containing the loss, ensemble prediction, individual model predictions, target, corrupted entropy, aleatoric uncertainty, and non-noise-corrupted entropy.

Return type
dict

class qumphy.models.deepbeat.**DeepBeatModule**(**args: Any*, ***kwargs: Any*)

Bases: LightningModule

Lightning module for DeepBeat data.

This module wraps a specific model architecture and defines the common training, validation, testing, prediction, optimizer, and scheduler logic.

Parameters

- **net** (*torch.nn.Module*) – Model architecture used for prediction.
- **optimizer** (*dict*) – Optimizer configuration dictionary.
- **output_activation** (*torch.nn.Module*) – Activation function applied to the raw model output.
- **prediction_activation** (*torch.nn.Module*) – Activation function applied to predictions returned by evaluation steps.
- **loss_fn** (*torch.nn.Module*) – Loss function used for training, validation, and testing.
- **lr_scheduler** (*dict*) – Learning rate scheduler configuration dictionary.

configure_optimizers()

Configure the optimizer and optional learning rate scheduler.

Returns
Dictionary containing the optimizer and, if provided, the learning rate scheduler configuration.

Return type
dict

forward(*data*)

Run a forward pass through the model.

Parameters
data (*torch.Tensor*) – Input tensor passed to the model.

Returns
Model prediction after applying the output activation.

Return type
torch.Tensor

predict_step(*batch*, *batch_idx*)

Run one prediction step.

Parameters

- **batch** (*tuple*) – Batch containing input data and target values.
- **batch_idx** (*int*) – Index of the current batch.

Returns
Prediction after applying the prediction activation.

Return type
torch.Tensor

`set_lr_scheduler()`

Instantiate and set the learning rate scheduler.

Returns
The function modifies the learning rate scheduler configuration in place.

Return type
None

`test_step(batch, batch_idx)`

Run one test step.

Parameters

- **batch** (*tuple*) – Batch containing input data and target values.
- **batch_idx** (*int*) – Index of the current batch.

Returns
Dictionary containing the loss, prediction, and target.

Return type
dict

`training_step(batch, batch_idx)`

Run one training step.

Parameters

- **batch** (*tuple*) – Batch containing input data and target values.
- **batch_idx** (*int*) – Index of the current batch.

Returns
Dictionary containing the loss, prediction, and target.

Return type
dict

`validation_step(batch, batch_idx)`

Run one validation step.

Parameters

- **batch** (*tuple*) – Batch containing input data and target values.
- **batch_idx** (*int*) – Index of the current batch.

Returns
Dictionary containing the loss, prediction, and target.

Return type
dict

`class qumphy.models.deepbeat.DeepBeatModule_MCD(*args: Any, **kwargs: Any)`

Bases: *DeepBeatModule*

Lightning parent module for DeepBeat data. Takes a specific model architecture as inputand uses sampling softmax for evaluation from https://arxiv.org/abs/1703.04977 :param *args: Positional arguments passed to DeepBeatModule. :param MCD_samples: Number of Monte Carlo dropout samples. :type MCD_samples: int :param **kwargs: Keyword arguments passed to DeepBeatModule.

test_step(*batch*, *batch_idx*)

Run one Monte Carlo dropout test step.

Parameters

- **batch** (*tuple*) – Batch containing input data and target values.
- **batch_idx** (*int*) – Index of the current batch.

Returns

Dictionary containing the loss, prediction, target, corrupted entropy, aleatoric uncertainty, and non-noise-corrupted entropy.

Return type

dict

qumphy.models.dumbnet module

File: qumphy/models/dumbnet.py Project: 22HLT01 QUMPHY Contact: oskar.pfeffer@ptb.de Gitlab: https://gitlab.com/qumphy Description: TinyVGG-style convolutional neural network.

class qumphy.models.dumbnet.**DumbNet**(*input_shape*, *hidden_units*, *output_shape*)

Bases: Module

Model architecture that replicates the TinyVGG model from CNN explainer website.

Parameters

- **input_shape** (*int*) – Number of input channels.
- **hidden_units** (*int*) – Number of hidden channels used in the convolutional layers.
- **output_shape** (*int*) – Number of output classes or output values.

forward(*x*)

Run a forward pass through the model.

Parameters

x (*torch.Tensor*) – Input tensor of shape (batch_size, input_shape, height, width).

Returns

Model output tensor of shape (batch_size, output_shape).

Return type

torch.Tensor

qumphy.models.inception1d module

File: qumphy/models/inception1d.py Project: 22HLT01 QUMPHY Contact: oskar.pfeffer@ptb.de Gitlab: https://gitlab.com/qumphy Description: InceptionTime-based 1D convolutional neural network.

class qumphy.models.inception1d.**Inception1d**(**args: Any*, ***kwargs: Any*)

Bases: Module

inception time architecture

forward(*x*)

Run a forward pass through the model.

Parameters

x (*torch.Tensor*) – Input tensor of shape (batch_size, input_channels, sequence_length).

Returns
Model output tensor of shape (batch_size, num_classes).

Return type
torch.Tensor

`get_layer_groups()`

Get grouped layers for model training.

Returns
Layer groups containing part of the backbone and the model head. If the depth is less than or equal to 3, only the model head is returned.

Return type
tuple or nn.Module

`get_output_layer()`

Get the output layer of the model.

Returns
Final output layer of the model head.

Return type
nn.Module

`set_output_layer(x)`

Set the output layer of the model.

Parameters
x (*nn.Module*) – New output layer.

Returns
The function modifies the output layer in place.

Return type
None

`class qumphy.models.inception1d.InceptionBackbone(*args: Any, **kwargs: Any)`

Bases: `Module`

Backbone of the 1D Inception architecture.

The backbone stacks multiple Inception blocks and optionally applies residual shortcut connections every three blocks.

`forward(x)`

Run a forward pass through the Inception backbone.

Parameters
x (*torch.Tensor*) – Input tensor of shape (batch_size, input_channels, sequence_length).

Returns
Output features from the stacked Inception blocks.

Return type
torch.Tensor

`class qumphy.models.inception1d.InceptionBlock1d(*args: Any, **kwargs: Any)`

Bases: `Module`

One-dimensional Inception block.

The block applies several convolutions with different kernel sizes and concatenates their outputs with a max-pooling branch. :param ni: Number of input channels. :type ni: int :param nb_filters: Number of filters used in each convolution branch. :type nb_filters: int :param kss: List of kernel sizes used by the convolution branches. :type kss: list :param stride: Stride used in the bottleneck and pooling branches. :type stride: int :param act: Activation function name. This parameter is currently not used. :type act: str :param bottleneck_size: Number of filters used in the bottleneck convolution. If 0, no

bottleneck convolution is applied.

forward(*x*)

Run a forward pass through the Inception block.

Parameters
x (*torch.Tensor*) – Input tensor of shape (batch_size, channels, sequence_length).

Returns
Output tensor after concatenation, batch normalization, and ReLU.

Return type
torch.Tensor

class qumphy.models.inception1d.**Shortcut1d**(**args: Any*, ***kwargs: Any*)

Bases: Module

Residual shortcut connection for 1D convolutional features.

forward(*inp*, *out*)

Apply the residual shortcut connection.

Parameters

- **inp** (*torch.Tensor*) – Input tensor used for the shortcut branch.
- **out** (*torch.Tensor*) – Output tensor from the main branch.

Returns
Result of adding the shortcut branch to the main branch and applying ReLU activation.

Return type
torch.Tensor

qumphy.models.inception1d.**conv**(*in_planes*, *out_planes*, *kernel_size=3*, *stride=1*)

Create a 1D convolutional layer with padding.

Parameters

- **in_planes** (*int*) – Number of input channels.
- **out_planes** (*int*) – Number of output channels.
- **kernel_size** (*int*) – Size of the convolution kernel.
- **stride** (*int*) – Stride of the convolution.

Returns
One-dimensional convolutional layer.

Return type
nn.Conv1d

qumphy.models.inception1d.**inception1d**(***kwargs*)

Construct an Inception1d model.

Parameters
****kwargs** – Keyword arguments passed to Inception1d.

Returns
Initialized Inception1d model.

Return type
Inception1d

qumphy.models.inception1d.**noop**(*x*)

Return the input unchanged.

Parameters
x (*torch.Tensor*) – Input tensor.

Returns
Same tensor as the input.

Return type
torch.Tensor

qumphy.models.inception_mantas module

File: qumphy/models/inception_mantas.py Project: 22HLT01 QUMPHY Contact: oskar.pfeffer@ptb.de Gitlab: https://gitlab.com/qumphy Description: Inception model, adapted from Matlab code from KTU.

class qumphy.models.inception_mantas.**Inception1DBlock**(*in_channels*)

Bases: Module

Single Inception block

Branch 1: 1x1 conv Branch 2: 1x1 conv -> 3x1 conv Branch 3: 1x1 conv -> 5x1 conv Branch 4: 3x1 maxpool -> 1x1 conv

forward(*x*)

Define 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.

class qumphy.models.inception_mantas.**Inception1DNet**(*in_channels*, *num_classes*)

Bases: Module

forward(*x*)

x shape: (Batch_size, Channels, Length)

qumphy.models.itransformer module

File: qumphy/models/itransformer.py Project: 22HLT01 QUMPHY Contact: oskar.pfeffer@ptb.de Gitlab: https://gitlab.com/qumphy Description: code implementation from https://github.com/thuml/Time-Series-Library .

class qumphy.models.itransformer.**iTransformer**(**args: Any*, ***kwargs: Any*)

Bases: Module

Inverted Transformer model for time-series tasks. Paper link: https://arxiv.org/abs/2310.06625

anomaly_detection(*x_enc*)

Run anomaly detection with the iTransformer model.

Parameters
x_enc (*torch.Tensor*) – Encoder input tensor of shape (batch_size, seq_len, num_features).

Returns
Reconstructed output tensor of shape (batch_size, seq_len, num_features).

Return type
torch.Tensor

classification(*x_enc*)

Run classification with the iTransformer model.

Parameters
x_enc (*torch.Tensor*) – Input tensor of shape (batch_size, channels, seq_len).

Returns
Classification logits of shape (batch_size, num_class).

Return type
torch.Tensor

forecast(*x_enc*, *x_mark_enc*, *x_dec*, *x_mark_dec*)

Run forecasting with the iTransformer model.

Parameters

- **x_enc** (*torch.Tensor*) – Encoder input tensor of shape (batch_size, seq_len, num_features).
- **x_mark_enc** (*torch.Tensor*) – Encoder time feature tensor.
- **x_dec** (*torch.Tensor*) – Decoder input tensor. This parameter is kept for compatibility.
- **x_mark_dec** (*torch.Tensor*) – Decoder time feature tensor. This parameter is kept for compatibility.

Returns
Forecast output tensor of shape (batch_size, pred_len, num_features).

Return type
torch.Tensor

forward(*x_enc*, *x_mark_enc=None*, *x_dec=None*, *x_mark_dec=None*, *mask=None*)

Run a forward pass for the selected task.

Parameters

- **x_enc** (*torch.Tensor*) – Encoder input tensor.
- **x_mark_enc** (*torch.Tensor*) – Encoder time feature tensor.

- **x_dec** (*torch.Tensor*) – Decoder input tensor.
- **x_mark_dec** (*torch.Tensor*) – Decoder time feature tensor.
- **mask** (*torch.Tensor*) – Mask tensor used for imputation.

Returns
Output tensor for the selected task. Returns None if the task name is not supported.

Return type
torch.Tensor or None

imputation(*x_enc*, *x_mark_enc*, *x_dec*, *x_mark_dec*, *mask*)

Run imputation with the iTransformer model.

Parameters

- **x_enc** (*torch.Tensor*) – Encoder input tensor of shape (batch_size, seq_len, num_features).
- **x_mark_enc** (*torch.Tensor*) – Encoder time feature tensor.
- **x_dec** (*torch.Tensor*) – Decoder input tensor. This parameter is kept for compatibility.
- **x_mark_dec** (*torch.Tensor*) – Decoder time feature tensor. This parameter is kept for compatibility.
- **mask** (*torch.Tensor*) – Mask tensor indicating missing or observed values.

Returns
Imputed output tensor of shape (batch_size, seq_len, num_features).

Return type
torch.Tensor

qumphy.models.minirocket module

File: qumphy/models/minrocket.py Project: 22HLT01 QUMPHY Contact: oskar.pfeffer@ptb.de Gitlab: https://gitlab.com/qumphy Description: MiniRocKeT implementation as from [...].

class qumphy.models.minirocket.**MiniRocket**(**args: Any*, ***kwargs: Any*)

Bases: Module

MiniRocket model with feature extractor and prediction head.

forward(*x*)

Run a forward pass through the MiniRocket model.

Parameters
x (*torch.Tensor*) – Input tensor of shape (batch_size, c_in, seq_len).

Returns
Model output after feature extraction, prediction head, and output activation.

Return type
torch.Tensor

class qumphy.models.minirocket.**MiniRocketFeatures**(**args: Any*, ***kwargs: Any*)

Bases: Module

This is a Pytorch implementation of MiniRocket developed by Malcolm McLean and Ignacio Oguiza This module extracts MiniRocket features from time-series data using fixed convolutional kernels, multiple dilations, and proportion of positive values features.

MiniRocket paper citation: @article{dempster_etal_2020,

author = {Dempster, Angus and Schmidt, Daniel F and Webb, Geoffrey I}, title = {{MINIROCKET}: A Very Fast (Almost) Deterministic Transform for Time Series Classification}, year = {2020}, journal = {arXiv:2012.08791}

} Original paper: https://arxiv.org/abs/2012.08791 Original code: https://github.com/angus924/minirocket

extract_features(*data*)

Extract MiniRocket features from input data.

Parameters
data (*torch.Tensor*) – Input data of shape (batch_size, seq_len) or (batch_size, c_in, seq_len).

Returns
Extracted MiniRocket features.

Return type
torch.Tensor

fitting = False

forward(*x*)

Extract MiniRocket features from an input tensor.

Parameters
x (*torch.Tensor*) – Input tensor of shape (batch_size, c_in, seq_len).

Returns
Extracted MiniRocket features of shape (batch_size, num_features).

Return type
torch.Tensor

get_quantiles(*num_quantiles*)

Calculate quantile values using the golden ratio.

Parameters
num_quantiles (*int*) – Number of quantile values to calculate.

Returns
List containing the calculated quantile values.

Return type
list

kernel_size = 9

num_kernels = 84

class qumphy.models.minirocket.**MiniRocketHead**(**args: Any*, ***kwargs: Any*)

Bases: Sequential

qumphy.models.minirocket.**get_minirocket_features**(*o*, *model*, *chunksize=1024*, *use_cuda=None*, *to_np=True*)

Extract MiniRocket features from a large dataset in chunks.

Parameters

- **o** (*np.ndarray or torch.Tensor*) – Input dataset.
- **model** (*nn.Module*) – MiniRocket feature extraction model.

- **chunksize** (*int*) – Number of samples processed in each chunk.
- **use_cuda** (*bool*) – If True, use CUDA. If False, use CPU. If None, CUDA is used when available.
- **to_np** (*bool*) – If True, return the features as a NumPy array. If False, return them as a torch.Tensor.

Returns
Extracted MiniRocket features.

Return type
np.ndarray or torch.Tensor

qumphy.models.ppnet module

File: qumphy/models/ppnet.py Project: 22HLT01 QUMPHY Contact: oskar.pfeffer@ptb.de Gitlab: https://gitlab.com/qumphy Description: PPNet model implementation from 10.1109/JSEN.2020.2990864.

class qumphy.models.ppnet.**PPNet**(**args: Any*, ***kwargs: Any*)

Bases: Module

CNN-LSTM network for one-dimensional sequence classification.

forward(*x*)

Run a forward pass through the PPNet model.

Parameters
x (*torch.Tensor*) – Input tensor of shape (batch_size, 1, input_length).

Returns
Output logits of shape (batch_size, num_classes).

Return type
torch.Tensor

qumphy.models.pulsedb module

File: qumphy/models/pulsedb.py Project: 22HLT01 QUMPHY Contact: oskar.pfeffer@ptb.de Gitlab: https://gitlab.com/qumphy Description: Lightning model for PulseDB data.

class qumphy.models.pulsedb.**PulseDBEnsemble**(**args: Any*, ***kwargs: Any*)

Bases: *PulseDBModule*

PulseDB ensemble module for Gaussian prediction aggregation.

denormalize(*prediction*)

forward(*x*)

load_models(*model_list*)

test_step(*batch*, *batch_idx*)

Run one ensemble test step. prediction has shape [model, batchsize, logits+noise]

Parameters

- **batch** (*tuple*) – Batch containing input data and target values.
- **batch_idx** (*int*) – Index of the current batch.

Returns
Dictionary containing the loss, ensemble prediction, individual model predictions, and target.

Return type
dict

class qumphy.models.pulsedb.**PulseDBGaussianLoss**(*num_distributions=2*, ***kwargs*)

Bases: GaussianNLLLoss

Gaussian negative log likelihood loss for the PulseDB dataset.

forward(*input*, *target*)

Calculate the Gaussian negative log likelihood loss.

Return type
Tensor

Parameters

- **input** (*torch.Tensor*) – Model output tensor containing predicted means and log-variances.
- **target** (*torch.Tensor*) – Ground truth target tensor.

Returns
Gaussian negative log likelihood loss.

Return type
torch.Tensor

class qumphy.models.pulsedb.**PulseDBModule**(**args: Any*, ***kwargs: Any*)

Bases: LightningModule

Lightning parent module for PulseDB data. Takes a specific model architecture as input. (net)

configure_optimizers()

Configure the optimizer and optional learning rate scheduler.

Returns
Dictionary containing the optimizer and, if provided, the learning rate scheduler configuration.

Return type
dict

denormalize_std(*prediction_std*)

Rescales in-place the standard deviation of the normalized target of 0 mean and 1 standard deviation to the original values, using the BP_std attribute.

Parameters
prediction_std (*torch.tensor*) – The input standard deviation to be denormalized.

Returns
The denormalized standard deviation.

Return type
torch.tensor

denormalize_target(*target*)

Rescales in-place the normalized target of 0 mean and 1 standard deviation to the original BP values, using the BP_mean and BP_std attributes.

Parameters
target (*torch.tensor*) – The input target to be denormalized.

Returns
The denormalized target.

Return type
torch.tensor

forward(*x*)

predict_step(*batch*, *batch_idx*)

set_dataset_stats(*dataset*)

Set and register target statistics from the dataset.

Parameters
dataset (*qumphy.datasets.PulseDBDataset*) – PulseDB dataset instance that provides target statistics.

Returns
The function registers target statistics as buffers.

Return type
None

set_lr_scheduler()

Configure the optimizer and optional learning rate scheduler.

Returns
Dictionary containing the optimizer and, if provided, the learning rate scheduler configuration.

Return type
dict

test_step(*batch*, *batch_idx*)

training_step(*batch*, *batch_idx*)

validation_step(*batch*, *batch_idx*)

class qumphy.models.pulsedb.**PulseDBModule_MCD**(**args: Any*, ***kwargs: Any*)

Bases: *PulseDBModule*

PulseDB Lightning module with Monte Carlo dropout evaluation.

test_step(*batch*, *batch_idx*)

qumphy.models.s42 module

File: qumphy/models/s42.py Project: 22HLT01 QUMPHY Contact: oskar.pfeffer@ptb.de Gitlab: https://gitlab.com/qumphy Description: Standalone version of Structured (Sequence) State Space (S4) model.

qumphy.models.s42.**Activation**(*activation=None*, *dim=-1*)

Compute the Cauchy multiplication using PyKeOps.

Parameters

- **v** (*torch.Tensor*) – Numerator tensor.
- **z** (*torch.Tensor*) – Evaluation points.

- **w** (*torch.Tensor*) – Complex poles.

Returns
Result of the Cauchy multiplication.

Return type
torch.Tensor

class qumphy.models.s42.**HippoSSKernel**(**args: Any*, ***kwargs: Any*)

Bases: Module

Wrapper around SSKernel that generates A, B, C, dt according to HiPPO arguments.

The SSKernel is expected to support the interface forward() default_state() setup_step() step()

default_state(**args*, ***kwargs*)

forward(*L=None*, *rate=1.0*)

step(*u*, *state*, ***kwargs*)

qumphy.models.s42.**LinearActivation**(*d_input*, *d_output*, *bias=True*, *zero_bias_init=False*, *transposed=False*, *initializer=None*, *activation=None*, *activate=False*, *weight_norm=False*, ***kwargs*)

Create a linear layer with optional initialization and activation.

Parameters

- **d_input** (*int*) – Number of input features.
- **d_output** (*int*) – Number of output features.
- **bias** (*bool*) – If True, include a bias parameter.
- **zero_bias_init** (*bool*) – If True, initialize the bias with zeros.
- **transposed** (*bool*) – If True, use TransposedLinear instead of nn.Linear.
- **initializer** (*str*) – Name of the weight initializer.
- **activation** (*str*) – Name of the activation function.
- **activate** (*bool*) – If True, append the activation function to the linear layer.
- **weight_norm** (*bool*) – If True, apply weight normalization to the linear layer.
- ****kwargs** – Additional keyword arguments passed to the linear layer.

Returns
Linear module, optionally followed by an activation function.

Return type
nn.Module

class qumphy.models.s42.**S4**(**args: Any*, ***kwargs: Any*)

Bases: Module

Structured State Space Sequence layer.

property d_output

property d_state

default_state(**batch_shape*, *device=None*)

forward(*u*, *rate=1.0*, ***kwargs*)

Run a forward pass through the S4 layer.

Parameters

- **u** (*torch.Tensor*) – Input tensor. If transposed is True, the shape is (batch_size, d_model, sequence_length). Otherwise, the shape is (batch_size, sequence_length, d_model).
- **rate** (*float*) – Sampling rate factor.
- ****kwargs** – Additional keyword arguments kept for compatibility.

Returns
Tuple containing the output tensor and None for compatibility with recurrent interfaces.

Return type
tuple

property state_to_tensor

step(*u*, *state*)

Step one time step as a recurrent model. Intended to be used during validation.

u: (B H) state: (B H N) Returns: output (B H), state (B H N)

class qumphy.models.s42.**SSKernelNPLR**(**args: Any*, ***kwargs: Any*)

Bases: Module

Stores a representation of and computes the SSKernel function K_L(A^dt, B^dt, C) corresponding to a discretized state space, where A is Normal + Low Rank (NPLR)

The class name stands for 'State-Space SSKernel for Normal Plus Low-Rank'. The parameters of this function are as follows.

A: (... N N) the state matrix B: (... N) input matrix C: (... N) output matrix dt: (...) timescales / discretization step size p, q: (... P N) low-rank correction to A, such that Ap=A+pq^T is a normal matrix

The forward pass of this Module returns: (... L) that represents represents FFT SSKernel_L(A^dt, B^dt, C)

default_state(**batch_shape*)

Create an initial recurrent state.

Parameters
***batch_shape** (*int*) – Batch dimensions of the state.

Returns
Zero-initialized recurrent state.

Return type
torch.Tensor

double_length()

Double the internal kernel length.

Returns
The function updates the internal length and cached FFT nodes.

Return type
None

forward(*state=None*, *rate=1.0*, *L=None*)

state: (..., s, N) extra tensor that augments B rate: sampling rate factor

returns: (..., c+s, L)

register(*name*, *tensor*, *trainable=False*, *lr=None*, *wd=None*)

Register a tensor as a parameter or buffer.

Parameters

- **name** (*str*) – Name used to register the tensor.
- **tensor** (*torch.Tensor*) – Tensor to register.
- **trainable** (*bool*) – If True, register the tensor as a trainable parameter. Otherwise, register it as a buffer.
- **lr** (*float*) – Optional learning rate metadata.
- **wd** (*float*) – Optional weight decay metadata.

Returns
The function registers the tensor in the module.

Return type
None

setup_step(*mode='dense'*)

Set up dA, dB, dC discretized parameters for stepping

step(*u*, *state*)

Must have called self.setup_step() and created state with self.default_state() before calling this

class qumphy.models.s42.**TransposedLinear**(**args: Any*, ***kwargs: Any*)

Bases: Module

Linear module on the second-to-last dimension

forward(*x*)

Run a forward pass through the transposed linear layer.

Parameters
x (*torch.Tensor*) – Input tensor.

Returns
Output tensor after applying the linear transformation.

Return type
torch.Tensor

qumphy.models.s42.**bilinear**(*dt*, *A*, *B=None*)

Apply bilinear discretization to a continuous state-space system.

Parameters

- **dt** (*torch.Tensor*) – Time step or timescale tensor.
- **A** (*torch.Tensor*) – Continuous transition matrix.
- **B** (*torch.Tensor*) – Optional continuous input matrix.

Returns
Discretized transition matrix and discretized input matrix.

Return type
tuple

qumphy.models.s42.**cauchy_conj**(*v*, *z*, *w*)

Compute the Cauchy multiplication using PyKeOps.

Parameters

- **v** (*torch.Tensor*) – Numerator tensor.
- **z** (*torch.Tensor*) – Evaluation points.
- **w** (*torch.Tensor*) – Complex poles.

Returns
Result of the Cauchy multiplication.

Return type
torch.Tensor

qumphy.models.s42.**embed_c2r**(*A*)

qumphy.models.s42.**get_initializer**(*name*, *activation=None*)

Get a weight initialization function.

Parameters

- **name** (*str*) – Name of the initializer. Supported values are “uniform”, “normal”, “xavier”, “zero”, and “one”.
- **activation** (*str*) – Activation function used to determine the initialization gain.

Returns
Weight initialization function.

Return type
callable

qumphy.models.s42.**krylov**(*L*, *A*, *b*, *c=None*, *return_power=False*)

Compute a Krylov sequence.

Parameters

- **L** (*int*) – Length of the Krylov sequence.
- **A** (*torch.Tensor*) – Square transition matrix.
- **b** (*torch.Tensor*) – Initial vector.
- **c** (*torch.Tensor*) – Optional projection vector.
- **return_power** (*bool*) – If True, also return A raised to the power L - 1.

Returns
Krylov sequence, optionally together with A raised to the power L - 1.

Return type
torch.Tensor or tuple

qumphy.models.s42.**nplr**(*measure*, *N*, *rank=1*, *dtype=torch.float*)

Convert a HiPPO matrix into normal plus low-rank form.

Parameters

- **measure** (*str*) – Type of HiPPO measure.
- **N** (*int*) – State dimension.
- **rank** (*int*) – Rank of the low-rank correction.

- **dtype** (*torch.dtype*) – Floating point data type.

Returns
Tuple containing eigenvalues, low-rank correction, input vector, and eigenvector matrix.

Return type
tuple

qumphy.models.s42.**power**(*L*, *A*, *v=None*)

Compute a matrix power and optional scan reduction.

Parameters

- **L** (*int*) – Power to which the matrix is raised.
- **A** (*torch.Tensor*) – Square matrix of shape (..., N, N).
- **v** (*torch.Tensor*) – Optional tensor used to compute the scan sum over powers of A.

Returns
If v is None, returns A raised to the power L. Otherwise, returns the matrix power and the scan reduction.

Return type
torch.Tensor or tuple

qumphy.models.s42.**rank_correction**(*measure*, *N*, *rank=1*, *dtype=torch.float*)

Return low-rank matrix L such that A + L is normal

Parameters

- **measure** (*str*) – Type of HiPPO measure.
- **N** (*int*) – State dimension.
- **rank** (*int*) – Rank of the correction matrix.
- **dtype** (*torch.dtype*) – Data type of the returned tensor.

Returns
Low-rank correction matrix.

Return type
torch.Tensor

qumphy.models.s42.**transition**(*measure*, *N*, ***measure_args*)

A, B transition matrices for different measures

measure: the type of measure
legt - Legendre (translated) legs - Legendre (scaled) glagt - generalized Laguerre (translated) lagt, tlagt - previous versions of (tilted) Laguerre with slightly different normalization

Parameters

- **measure** (*str*) – Type of HiPPO measure. Supported values include “legt”, “legs”, “glagt”, “lagt”, “fourier”, “random”, and “diagonal”.
- **N** (*int*) – State dimension.
- ****measure_args** – Additional arguments for the selected measure.

Returns
Tuple containing the transition matrix A and input matrix B.

Return type
tuple

qumphy.models.s4_model module

File: qumphy/models/s4_model.py Project: 22HLT01 QUMPHY Contact: oskar.pfeffer@ptb.de Gitlab: https://gitlab.com/qumphy Description: adapted from https://github.com/HazyResearch/state-spaces/blob/main/example.py .

class qumphy.models.s4_model.**S4Model**(**args: Any*, ***kwargs: Any*)

Bases: Module

Stacked S4 model for sequence modeling.

This model uses an optional input encoder, multiple S4 residual blocks, optional pooling over the sequence dimension, and an optional output decoder.

forward(*x*, *rate=1.0*)

Run a forward pass through the S4 model.

Parameters

- **x** (*torch.Tensor*) – Input tensor. If transposed_input is True, the shape is (batch_size, d_input, sequence_length). Otherwise, the shape is (batch_size, sequence_length, d_input).
- **rate** (*float*) – Sampling rate factor passed to the S4 layers.

Returns
Model output. If pooling is True and the decoder is enabled, the output has shape (batch_size, d_output). If pooling is False, the output keeps the sequence dimension.

Return type
torch.Tensor

qumphy.models.timesnet module

File: qumphy/models/timesnet.py Project: 22HLT01 QUMPHY Contact: oskar.pfeffer@ptb.de Gitlab: https://gitlab.com/qumphy Description: Adapted from https://github.com/thuml/Time-Series-Library.

qumphy.models.timesnet.**FFT_for_Period**(*x*, *k=2*)

Find dominant periods in a time-series batch using the FFT.

Parameters

- **x** (*torch.Tensor*) – Input tensor of shape (batch_size, sequence_length, channels).
- **k** (*int*) – Number of dominant frequencies to select.

Returns
Tuple containing the selected periods and their corresponding frequency weights.

Return type
tuple

class qumphy.models.timesnet.**TimesBlock**(**args: Any*, ***kwargs: Any*)

Bases: Module

TimesNet block for multi-period temporal feature extraction.

forward(*x*)

class qumphy.models.timesnet.**TimesNet**(**args: Any*, ***kwargs: Any*)

Bases: Module

TimesNet model for time series forecasting and classification.

Implements the TimesNet architecture as described in: https://openreview.net/pdf?id=ju_Uqw384Oq

Parameters

- **seq_len** (*int*) – Length of the input sequence.
- **label_len** (*int*) – Length of the label/start token sequence (for forecasting).
- **pred_len** (*int, optional*) – Length of the prediction/output sequence.
- **e_layers** (*int, optional*) – Number of encoder layers. Default is 2.
- **d_model** (*int, optional*) – Model hidden dimension. Default is 16.
- **d_ff** (*int, optional*) – Dimension of the feed-forward network. Default is 32.
- **num_kernels** (*int, optional*) – Number of kernels for the Inception-like blocks. Default is 6.
- **top_k** (*int, optional*) – Top-k selection parameter for TimesBlock. Default is 5.
- **enc_in** (*int, optional*) – Number of input channels/features. Default is 1.
- **c_out** (*int, optional*) – Number of output channels/features or classes. Default is 1.
- **embed** (*str, optional*) – Type of time feature embedding. Options are 'timeF', 'fixed', or 'learned'. Default is 'fixed'.
- **freq** (*str, optional*) – Frequency string for time feature encoding (e.g., 'h' for hourly). Default is 's'.
- **dropout** (*float, optional*) – Dropout rate. Default is 0.1.
- **num_class** (*int, optional*) – Number of classes (for classification tasks).
- **task_name** (*str, optional*) – Task type, either 'classification' or 'forecasting'. Default is 'classification'.

References

anomaly_detection(*x_enc*)

Run anomaly detection with the TimesNet model.

Parameters
x_enc (*torch.Tensor*) – Encoder input tensor of shape (batch_size, seq_len, enc_in).

Returns
Reconstructed output tensor.

Return type
torch.Tensor

classification(*x_enc*, *x_mark_enc*)

Run classification with the TimesNet model.

Parameters

- **x_enc** (*torch.Tensor*) – Input tensor of shape (batch_size, enc_in, seq_len).
- **x_mark_enc** (*torch.Tensor*) – Encoder time feature tensor. This parameter is kept for compatibility.

Returns
Classification logits of shape (batch_size, num_class).

Return type
torch.Tensor

forecast(*x_enc*, *x_mark_enc*, *x_dec*, *x_mark_dec*)

Run forecasting with the TimesNet model.

Parameters

- **x_enc** (*torch.Tensor*) – Encoder input tensor of shape (batch_size, seq_len, enc_in).
- **x_mark_enc** (*torch.Tensor*) – Encoder time feature tensor.
- **x_dec** (*torch.Tensor*) – Decoder input tensor. This parameter is kept for compatibility.
- **x_mark_dec** (*torch.Tensor*) – Decoder time feature tensor. This parameter is kept for compatibility.

Returns
Forecast output tensor of shape (batch_size, seq_len + pred_len, c_out).

Return type
torch.Tensor

forward(*x_enc*, *x_mark_enc=None*, *x_dec=None*, *x_mark_dec=None*, *mask=None*)

imputation(*x_enc*, *x_mark_enc*, *x_dec*, *x_mark_dec*, *mask*)

Run imputation with the TimesNet model.

Parameters

- **x_enc** (*torch.Tensor*) – Encoder input tensor of shape (batch_size, seq_len, enc_in).
- **x_mark_enc** (*torch.Tensor*) – Encoder time feature tensor.
- **x_dec** (*torch.Tensor*) – Decoder input tensor. This parameter is kept for compatibility.
- **x_mark_dec** (*torch.Tensor*) – Decoder time feature tensor. This parameter is kept for compatibility.
- **mask** (*torch.Tensor*) – Mask tensor indicating observed and missing values.

Returns
Imputed output tensor.

Return type
torch.Tensor

qumphy.models.xresnet1d module

File: qumphy/models/xresnet1d.py Project: 22HLT01 QUMPHY Contact: oskar.pfeffer@ptb.de Gitlab: https://gitlab.com/qumphy Description: One-dimensional XResNet and XBotNet model architectures.

qumphy.models.xresnet1d.**BatchNorm**(*nf*, *norm_type=NormType.Batch*, ***kwargs*)

Create a batch normalization layer.

Parameters

- **nf** (*int*) – Number of input features.
- **norm_type** (NormType) – Type of batch normalization initialization.
- ****kwargs** – Additional keyword arguments passed to the batch normalization layer.

Returns
Initialized batch normalization layer.

Return type
nn.Module

class qumphy.models.xresnet1d.**ConvLayer**(**args: Any*, ***kwargs: Any*)

Bases: Sequential

One-dimensional convolutional layer with optional activation and normalization.

class qumphy.models.xresnet1d.**MHSA1d**(**args: Any*, ***kwargs: Any*)

Bases: Module

Multi-head self-attention block for one-dimensional inputs.

forward(*x*)

Run a forward pass through the attention block.

Parameters
x (*torch.Tensor*) – Input tensor of shape (batch_size, channels, sequence_length).

Returns
Output tensor with the same shape as the input.

Return type
torch.Tensor

class qumphy.models.xresnet1d.**NormType**(*value*)

Bases: Enum

An enumeration.

Batch = 1

BatchZero = 2

class qumphy.models.xresnet1d.**ResBlock**(**args: Any*, ***kwargs: Any*)

Bases: Module

Residual block for one-dimensional XResNet models.

forward(*x*)

Run a forward pass through the residual block.

Parameters
x (*torch.Tensor*) – Input tensor.

Returns
Output tensor after the residual connection and activation.

Return type
torch.Tensor

class qumphy.models.xresnet1d.**ResBlock_dropout**(**args: Any*, ***kwargs: Any*)

Bases: Module

Residual block with Monte Carlo dropout for one-dimensional XResNet models.

forward(*x*)

Run a forward pass through the dropout residual block.

Parameters
x (*torch.Tensor*) – Input tensor.

Returns
Output tensor after the residual connection and activation.

Return type
torch.Tensor

class qumphy.models.xresnet1d.**XResNet1d**(**args: Any*, ***kwargs: Any*)

Bases: Sequential

Configurable one-dimensional XResNet model.

get_layer_groups()

Get layer groups of the model.

Returns
Tuple containing selected feature layers and the model head.

Return type
tuple

get_output_layer()

Get the output layer of the model.

Returns
Final output layer of the model head.

Return type
nn.Module

set_output_layer(*x*)

Set the output layer of the model.

Parameters
x (*nn.Module*) – New output layer.

Returns
The function modifies the output layer in place.

Return type
None

class qumphy.models.xresnet1d.**XResNet1d101**(**args: Any*, ***kwargs: Any*)

Bases: *XResNet1d*

XResNet1d-101 model.

class qumphy.models.xresnet1d.**XResNet1d50**(**args: Any*, ***kwargs: Any*)

Bases: *XResNet1d*

XResNet1d-50 model.

qumphy.models.xresnet1d.**init_cnn**(*m*)

Initialize CNN module weights recursively.

Parameters
m (*nn.Module*) – Module whose children should be initialized.

Returns
The function modifies the module weights in place.

Return type
None

qumphy.models.xresnet1d.**init_default**(*m*, *func=torch.nn.init.kaiming_normal_*)

Initialize module weights and bias.

Parameters

- **m** (*nn.Module*) – Module whose weights and bias should be initialized.
- **func** (*callable*) – Initialization function applied to the module weights.

Returns
Initialized module.

Return type
nn.Module

qumphy.models.xresnet1d.**xbotnet1d101**(***kwargs*)

Create an XBotNet1d-101 model with multi-head self-attention.

Parameters
****kwargs** – Additional keyword arguments passed to XResNet1d.

Returns
Initialized XBotNet1d-101 model.

Return type
XResNet1d

qumphy.models.xresnet1d.**xbotnet1d152**(***kwargs*)

Create an XBotNet1d-152 model with multi-head self-attention.

Parameters
****kwargs** – Additional keyword arguments passed to XResNet1d.

Returns
Initialized XBotNet1d-152 model.

Return type
XResNet1d

qumphy.models.xresnet1d.**xbotnet1d50**(***kwargs*)

Create an XBotNet1d-50 model with multi-head self-attention.

Parameters
****kwargs** – Additional keyword arguments passed to XResNet1d.

Returns
Initialized XBotNet1d-50 model.

Return type
XResNet1d

qumphy.models.xresnet1d.**xresnet1d101**(***kwargs*)

Create an XResNet1d-101 model.

Parameters
****kwargs** – Additional keyword arguments passed to XResNet1d.

Returns
Initialized XResNet1d-101 model.

Return type
XResNet1d

qumphy.models.xresnet1d.**xresnet1d152**(***kwargs*)

Create an XResNet1d-152 model.

Parameters
****kwargs** – Additional keyword arguments passed to XResNet1d.

Returns
Initialized XResNet1d-152 model.

Return type
XResNet1d

qumphy.models.xresnet1d.**xresnet1d18**(***kwargs*)

Create an XResNet1d-18 model.

Parameters
****kwargs** – Additional keyword arguments passed to XResNet1d.

Returns
Initialized XResNet1d-18 model.

Return type
XResNet1d

qumphy.models.xresnet1d.**xresnet1d18_deep**(***kwargs*)

Create a deep XResNet1d-18 model.

Parameters
****kwargs** – Additional keyword arguments passed to XResNet1d.

Returns
Initialized deep XResNet1d-18 model.

Return type
XResNet1d

qumphy.models.xresnet1d.**xresnet1d18_deeper**(***kwargs*)

Create a deeper XResNet1d-18 model.

Parameters
****kwargs** – Additional keyword arguments passed to XResNet1d.

Returns
Initialized deeper XResNet1d-18 model.

Return type
XResNet1d

qumphy.models.xresnet1d.**xresnet1d34**(***kwargs*)

Create an XResNet1d-34 model.

Parameters
****kwargs** – Additional keyword arguments passed to XResNet1d.

Returns
Initialized XResNet1d-34 model.

Return type
XResNet1d

qumphy.models.xresnet1d.**xresnet1d34_deep**(***kwargs*)

Create a deep XResNet1d-34 model.

Parameters
****kwargs** – Additional keyword arguments passed to XResNet1d.

Returns
Initialized deep XResNet1d-34 model.

Return type
XResNet1d

qumphy.models.xresnet1d.**xresnet1d34_deeper**(***kwargs*)

Create a deeper XResNet1d-34 model.

Parameters
****kwargs** – Additional keyword arguments passed to XResNet1d.

Returns
Initialized deeper XResNet1d-34 model.

Return type
XResNet1d

qumphy.models.xresnet1d.**xresnet1d50**(***kwargs*)

Create an XResNet1d-50 model.

Parameters
****kwargs** – Additional keyword arguments passed to XResNet1d.

Returns
Initialized XResNet1d-50 model.

Return type
XResNet1d

qumphy.models.xresnet1d.**xresnet1d50_MCD**(***kwargs*)

Create an XResNet1d-50 model with Monte Carlo dropout.

Parameters
****kwargs** – Additional keyword arguments passed to XResNet1d.

Returns
Initialized XResNet1d-50 model with Monte Carlo dropout.

Return type
XResNet1d

qumphy.models.xresnet1d.**xresnet1d50_deep**(***kwargs*)

Create a deep XResNet1d-50 model.

Parameters
****kwargs** – Additional keyword arguments passed to XResNet1d.

Returns
Initialized deep XResNet1d-50 model.

Return type
XResNet1d

qumphy.models.xresnet1d.**xresnet1d50_deeper**(***kwargs*)

Create a deeper XResNet1d-50 model.

Parameters
****kwargs** – Additional keyword arguments passed to XResNet1d.

Returns
Initialized deeper XResNet1d-50 model.

Return type
XResNet1d

3.5.2 Submodules

3.5.2.1 qumphy.metrics module

File: qumphy/metrics.py Project: 22HLT01 QUMPHY Contact: nando.hegemann@ptb.de Gitlab: https://gitlab.com/qumphy Description: Evaluation metrics for model performance.

qumphy.metrics.**all_binary_metrics**(*target*, *prediction*)

Evaluate all binary classification metrics.

Given a target and a prediction array, this function computes all metrics as decided for the QUMPHY common evaluation framework.

The metrics are returned as a dictionary with the following keys:

- *auc*: Area under the curve calculated with raw probabilities
- *f1*: F1-score calculated with a classification threshold of 0.5
- *mcc_sens*: Matthews correlation coefficient calculated with a threshold achieving a sensitivity of 0.8
- *mcc_spec*: Matthews correlation coefficient calculated with a threshold achieving a specificity of 0.8
- *sens*: Sensitivity (with a threshold achieving a sensitivity of 0.8)
- *spec*: Specificity (with a threshold achieving a specificity of 0.8)

Parameters

- **target** (*np.ndarray*) – Ground truth values.
- **prediction** (*np.ndarray*) – Model output predictions (raw probability of positive class).

Returns
Dictionary with all metrics.

Return type
Dict[str, float]

qumphy.metrics.**all_regression_metrics**(*target*, *prediction*, *baseline_mae=None*)

Evaluate all regression classification metrics.

Return type
dict[str, float]

Parameters

- **target** (*np.ndarray*) – Ground truth values.
- **prediction** (*np.ndarray*) – Model output predictions.
- **baseline_mae** (*float*) – Baseline mean absolute error.

Returns
Dictionary with all metrics.

Return type
dict[str, float]

qumphy.metrics.**auc_score_binary**(*target*, *prediction*, *axis=0*)

Compute the area und curve (AUC) score for binary classification.

Return type
float | np.ndarray

Parameters

- **target** (*np.ndarray*) – Binary ground truth values for different samples.
- **prediction** (*np.ndarray*) – Binary model output predictions (raw prob.) associated with the positive class.
- **axis** (*int, optional*) – Axis to compute AUC over, by default 0.

Returns
Array of AUC values.

See also

multiclass_auc_score
AUC score for more then two classes.

Examples

```
>>> target = np.array([0, 1, 0, 1, 1, 0, 1, 0, 1, 0])
>>> prediction = np.array([.99, .8, .6, .63, .77, .23, .3, .78, .2, 0.01])
>>> auc_score_binary(target, prediction)
xx
```

```
>>> target = np.random.randint(0, 2, (100, 2))
>>> auc_score_binary(target, target, axis=0)
(1.0, 1.0)
```

```
>>> target = np.random.randint(0, 2, (50, 100, 2, 5))
>>> auc_score_binary(target, target, axis=1).shape
(50, 2, 5)
```

qumphy.metrics.**auc_score_multiclass**(*target*, *prediction*, *comparison_type='ovr'*)

Compute the area und curve (AUC) score.

Return type
float | np.ndarray

Parameters

- **target** (*np.ndarray*) – Multiclass ground truth values.
- **prediction** (*np.ndarray*) – Array of model output probabilities of different classes for different samples. If *target* shape is `(n_samples, ...)` with `n_classes` different class values, then *prediction* needs to have shape `(n_samples, n_classes, ...)`. Axis 1 (`n_classes`) needs to sum to one.

- **comparison_type** (*str*) – Comparison type for multiclasses, by default “ovr”.

  `ovr` : Stands for one-vs-rest. Computes the AUC for each class against the rest of the classes.

  `ovo` : Stands for one-vs-one. Computes the average AUC of all possible pairwise combinations of classes.

Returns

Array of AUC values.

See also

auc_score_binary
AUC score for exactly two classes.

Examples

```
>>> target = np.array([0, 1, 2, 1, 2, 0])
>>> prediction = np.array([[0.8, 0.1, 0.1],
>>>                        [0.2, 0.5, 0.3],
>>>                        [0.8, 0.1, 0.1],
>>>                        [0.7, 0.2, 0.1],
>>>                        [0.4, 0.3, 0.3],
>>>                        [0.5, 0.4, 0.1]])
>>> auc_score_multiclass(target, prediction, comparison_type="ovo")
0.6875
```

```
>>> target = np.random.randint(0, 3, (100, 10, 5, 2))
>>> prediction = np.random.uniform(0, 1, (100, 3, 10, 5, 2))
>>> prediction /= np.expand_dims(np.sum(prediction, axis=1), 1)
>>> auc_score_multiclass(target, prediction).shape
(10, 5, 2)
```

qumphy.metrics.**balanced_accuracy_score**(*target*, *prediction*)

Compute balanced accuracy score for binary or multi-class classification.

For the binary case the balanced accuracy score Acc_b is given by the arithmetic mean of sensitivity (Se) and specificity (Sp), i.e. :rtype: `float`

$$\mathrm{Acc}_b = \frac{1}{2}(\mathrm{Se}+\mathrm{Sp}) = \frac{1}{2}\Big(\frac{\mathrm{TP}}{\mathrm{TP}+\mathrm{FN}} + \frac{\mathrm{TN}}{\mathrm{TN}+\mathrm{FP}}\Big),$$

expressed by true positives (TP), true negatives (TN), false positives (FP) and false negatives (FN). In general, balanced accuracy is computed by

$$\mathrm{Acc}_b(y_{\mathrm{true}}, y_{\mathrm{pred}}) = \frac{\sum_{i=1}^{N} w_i\, \delta(y_{\mathrm{true},i} = y_{\mathrm{pred},i})}{\sum_{i=1}^{N} w_i}$$

with weights

$$w_i = \frac{1}{\sum_{j=1}^{N} \delta(y_{\mathrm{true},i} = y_{\mathrm{true},j})},$$

where $\delta(y_i = y_j)$ denotes the Kronecker delta function.

Parameters

- **target** (*np.ndarray*) – Ground truth values.
- **prediction** (*np.ndarray*) – Predicted values.

Returns
Balanced accuracy score for binary or multi-class classification.

Return type
float

Examples

Computation of balanced accuracy score for binary classification, i.e., a one dimensional array with only 2 classes

```
>>> target = np.array([0, 1, 1, 0, 1, 0, 0, 1, 0, 1])
>>> prediction = np.array([0, 1, 0, 1, 1, 1, 0, 1, 1, 1])
>>> balanced_accuracy_score(target, prediction)
0.6
```

Computation of balanced accuracy score for a multi-class scenario, i.e., a 1D array with more then two classes

```
>>> target = np.array([1, 2, 2, 2, 1, 2, 1, 0, 1, 1])
>>> prediction = np.array([1, 1, 2, 0, 0, 1, 1, 0, 0, 2])
>>> balanced_accuracy_score(target, prediction)
0.55
```

qumphy.metrics.**brier_score**(*target*, *prediction*)

Compute Brier score for binary classification.

Return type
`float`

Parameters

- **target** (*np.ndarray*) – Ground truth values.
- **prediction** (*np.ndarray*) – Predicted values.

Returns
Brier score for binary classification.

Return type
float

Examples

```
>>> target = np.array([0, 1, 1, 0, 1, 0, 0, 1, 0, 1])
>>> prediction = prediction = np.linspace(0,1,10)
>>> brier_score(target, prediction)
0.34074074074074073
```

qumphy.metrics.**f1_score**(*target*, *prediction*, *average=None*)

Compute F1-score of binary, multi-class or multi-label classification.

The F_1 score is computed using the true positives (TP), false positives (FP) and false negatives (FN) via :rtype: float | np.ndarray

$$F_1 = \frac{2\,\mathrm{TP}}{2\,\mathrm{TP} + \mathrm{FP} + \mathrm{FN}}.$$

Parameters

- **target** (*np.ndarray*) – Ground truth values.
- **prediction** (*np.ndarray*) – Predicted values.
- **average** (*str | None, optional*) – Averaging of the F1-scores (default None). For binary classification, `average=binary` is the default case. For multi-class and multi-lable classification, `average=None` is the default case, which results in F1-scores for each individual class. For more detail about averaging see the documentation of sklearn.metrics.f1_score.

Returns
F1 score(s) for binary, multi-class or multi-lable classification.

Return type
float | np.ndarray

Examples

Computation of F1-score for binary classification, i.e., a one dimensional array with only 2 classes

```
>>> target = np.array([0, 0, 1, 0, 1, 0, 0, 1, 0, 1])
>>> prediction = np.array([1, 1, 0, 1, 1, 1, 0, 1, 1, 1])
>>> f1_score(target, prediction)
0.5
```

Computation of F1-score for a multi-class scenario, i.e., a 1D array with more then two classes

```
>>> target = np.array([1, 2, 2, 2, 1, 2, 1, 0, 1, 1])
>>> prediction = np.array([1, 1, 2, 0, 0, 1, 1, 0, 0, 2])
>>> f1_score(target, prediction)
array([0.4, 0.44444444, 0.33333333])
```

Computation of F1-score for a multi-lable scenario, i.e., a 2D array with with columns representing different labels and values 0 or 1 as entries

```
>>> target = np.array([[0, 1, 0], [1, 0, 1], [1, 1, 0], [1, 1, 1]])
>>> prediction = np.array([[1, 0, 1], [1, 1, 1], [0, 0, 1], [1, 1, 0]])
>>> f1_score(target, prediction)
array([0.66666667, 0.4, 0.4])
```

Computation of F1-score for a multi-class scenario with averaging of F1-scores over the different classes

```
>>> target = np.array([1, 2, 2, 2, 1, 2, 1, 0, 1, 1])
>>> prediction = np.array([1, 1, 2, 0, 0, 1, 1, 0, 0, 2])
>>> f1_score(target, prediction, average='micro')
0.4
```

qumphy.metrics.**false_discovery_rate**(*target*, *prediction*, *average=None*)

Compute false discovery rate (FDR).

The false discovery rate (FDR) is given by $\mathrm{FDR} = 1 - \mathrm{PPV}$, where PPV is the precision (positive predicted value).

Return type
float | np.ndarray

Parameters

- **target** (*np.ndarray*) – Ground truth values.
- **prediction** (*np.ndarray*) – Predicted values.
- **average** (*str | None, optional*) – Averaging type (default None). For more detail about averaging see the documentation of sklearn.metrics.precision_score.

Returns
False discovery rate for binary, multi-class or multi-lable classification.

Return type
float | np.ndarray

See also

precision_score, *f1_score*

qumphy.metrics.**general_threshold**(*target*, *prediction*, *metric*, *metric_value*, *greater_than=True*)

Find the threshold that sets the given metric closest to *metric_value*.

Return type
float

Parameters

- **target** (*np.ndarray*) – Ground truth values.
- **prediction** (*np.ndarray*) – Model output predictions.
- **metric** (*Callable[[np.ndarray, np.ndarray], float]*) – A metric function that takes target and prediction arrays as input.
- **metric_value** (*float*) – The value of the metric to be achieved.
- **greater_than** (*bool*) – True if the metric is supposed to be higher than *metric_value*, false otherwise.

Returns
The threshold that achieves the metric.

Return type
float

qumphy.metrics.**ieee_grades**(*target*, *prediction*)

Compute the IEEE grades of the predicted values.

The grades are calculated by comparing the difference between the target and the prediction. Returned are the percentage of samples that fall within each grade. The grading scores follow the IEEE Std 1708a™-2019 scheme, where instead of a mean absolute difference of two measurements with the standard device, we use only one measurement.

Return type
ndarray

The grading for each sample is determined as follows

- Grade A for error <5 mmHg
- Grade B for error between 5-6 mmHg
- Grade C for error between 6-7 mmHg

- Grade D for error >7 mmHg

Parameters

- **target** (*np.ndarray*) – Ground truth values.
- **prediction** (*np.ndarray*) – Model output predictions.

Return type
np.ndarray

qumphy.metrics.**ieee_grades_torch**(*target*, *prediction*)

qumphy.metrics.**l1_norm**(*array*, *axis=0*)

Compute the L^1-norm of an array along an axis.

The L^1-norm of an array $x \in \mathbb{R}^N$ is given by $\|x\|_{L^1} = \frac{1}{N}\sum_{j=1}^{N} |x_j|$.

Return type
float | np.ndarray

Parameters

- **array** (*np.ndarray*) – Data array.
- **axis** (*int, optional*) – Axis, by default 0.

Returns
Array of L^1-norms over the specified axes.

See also

mean_absolute_error
Wrapper for `l1_norm(target - prediction)`.

l2_norm, *root_mean_square_error*

Examples

```
>>> l1_norm(np.array([1, 2, 3, 4]))
10
```

```
>>> array = np.random.normal(0, 1, (10, 5, 3, 2))
>>> l1_norm(array, axis=1).shape  # norm over second axis
(10, 3, 1)
```

qumphy.metrics.**l2_norm**(*array*, *axis=0*)

Compute the L^2-norm along an axis.

The L^2-norm of an array $x \in \mathbb{R}^N$ is given by $\|x\|_{L^2} = \sqrt{\frac{1}{N}\sum_{j=1}^{N} x_j^2}$.

Return type
float | np.ndarray

Parameters

- **array** (*np.ndarray*) – Data array.

- **axis** (*int, optional*) – Axis, by default 0.

Returns

Array of L^2-norms over the specified axes.

See also

root_mean_square_error
Wrapper for `l2_norm(target - prediction)`.

l1_norm, *mean_absolute_error*

Examples

```
>>> l2_norm(np.array([1, 2, 3, 4]))**2
30.0
```

```
>>> array = np.random.normal(0, 1, (10, 5, 3, 2))
>>> l2_norm(array, axis=1).shape  # norm over second axis
(10, 3, 1)
```

qumphy.metrics.**matthews_correlation_coefficient**(*target*, *prediction*)

Compute Matthews correlation coefficient (Mcc) of binary or multi-class task.

The Matthews correlation coefficient (Mcc) is computed using the true positives (TP), false positives (FP), true negatives (TN) and false negatives (FN) via :rtype: `float`

$$\text{Mcc} = \frac{\text{TP} \cdot \text{TN} - \text{FP} \cdot \text{FN}}{\sqrt{(\text{TP}+\text{FP})(\text{TP}+\text{FN})(\text{TN}+\text{FP})(\text{TN}+\text{FN})}}.$$

For the multi-class case, let C be the confusion matrix for K classes and define the number of times class k truly occurs $t_k = \sum_{i=1}^{K} C_{ik}$, the number of times class k was predicted $p_k = \sum_{i=1}^{K} C_{ki}$, the total number of samples correctly predicted $c = \sum_{k=1}^{K} C_{kk}$ and the total number of samples $s = \sum_{i,j=1}^{K} C_{ij}$. Then the multiclass Mcc is defined as

$$\text{Mcc} = \frac{c \cdot s - \sum_{k=1}^{K} p_k \cdot t_k}{\sqrt{(s^2 - \sum_{k=1}^{K} p_k^2)(s^2 - \sum_{k=1}^{K} t_k^2)}}.$$

Note

When there are more than two labels, the value of the MCC will no longer range between -1 and +1. Instead the minimum value will be somewhere between -1 and 0 depending on the number and distribution of ground true labels. The maximum value is always +1.

Parameters

- **target** (*np.ndarray*) – Ground truth values.
- **prediction** (*np.ndarray*) – Predicted values.

Returns

Mcc for binary and multi-class classification.

Return type

float | np.ndarray

Examples

Computation of Mcc for binary classification, i.e., a one dimensional array with only 2 classes

```
>>> target = np.array([0, 0, 1, 0, 1, 0, 0, 1, 0, 1])
>>> prediction = np.array([1, 1, 0, 1, 1, 1, 0, 1, 1, 1])
>>> matthews_correlation_coefficient(target, prediction)
-0.10206207261596577
```

Computation of Mcc for a multi-class scenario, i.e., a 1D array with more then two classes

```
>>> target = np.array([1, 2, 2, 2, 1, 2, 1, 0, 1, 1])
>>> prediction = np.array([1, 1, 2, 0, 0, 1, 1, 0, 0, 2])
>>> matthews_correlation_coefficient(target, prediction)
0.13130643285972254
```

qumphy.metrics.**mean_absolute_error**(*target*, *prediction*, *axis=0*)

Compute the mean absolute error (MAE) between target and prediction.

Return type
float | np.ndarray

Parameters

- **target** (*np.ndarray*) – Ground truth values.
- **prediction** (*np.ndarray*) – Model output predictions.
- **axis** (*int, optional*) – Axis to sum over, by default 0.

Returns
Array of MAE values (L^1-norms) over the specified axes.

See also

l1_norm
This is a wrapper for `l1_norm(target - prediction, axis=axis)`.

l2_norm, *root_mean_square_error*

Examples

```
>>> mean_absolute_error(np.array([1, 2, 3]), np.array([1, 2, 3]))
0.0
```

```
>>> target = np.random.normal(0, 1, (10, 5, 3, 2))
>>> prediction = np.random.normal(0, 1, (10, 5, 3, 2))
>>> mean_absolute_error(target, prediction, axis=1).shape  # norm over second axis
(10, 3, 1)
```

qumphy.metrics.**mean_absolute_scaled_error**(*baseline_mae*, *model_mae*)

Compute mean absolute scaled error (MASE).

The MASE is a measure of the magnitude of the error relative to a baseline error. It is defined as the mean absolute error divided by the baseline error.

Return type
`float`

Parameters

- **baseline_mae** (*float*) – Mean absolute scaled error of the baseline.
- **model_mae** (*float*) – Mean absolute error.

Returns
Mean absolute scaled error.

Return type
float

qumphy.metrics.**negative_predictive_value**(*target*, *prediction*)

Compute negative predictive value (NPV) of binary, multi-class or multi-label classification.

The negative predictive value is computed using the true positives (TN) and false positives (FN) via :rtype: float | np.ndarray

$$\mathrm{NPV} = \frac{\mathrm{TN}}{\mathrm{TN} + \mathrm{FN}}.$$

Parameters

- **target** (*np.ndarray*) – Ground truth values.
- **prediction** (*np.ndarray*) – Predicted values.

Returns
Precision scores for binary, multi-class or multi-lable classification.

Return type
float | np.ndarray

See also

f1_score, *false_discovery_rate*, *precision_score*

Examples

Computation of negative predictive value score for binary classification, i.e., a one dimensional array with only 2 classes

```
>>> target = np.array([0, 0, 1, 0, 1, 0, 0, 1, 0, 1])
>>> prediction = np.array([1, 1, 0, 1, 1, 1, 0, 1, 1, 1])
>>> negative_predictive_value(target, prediction)
0.5
```

qumphy.metrics.**precision_score**(*target*, *prediction*, *average=None*)

Compute precision (PPV) of binary, multi-class or multi-label classification.

The precision score (positive predictive value, PPV) is computed using the true positives (TP) and false positives (FP) via :rtype: float | np.ndarray

$$\mathrm{PPV} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP}}.$$

Parameters

- **target** (*np.ndarray*) – Ground truth values.
- **prediction** (*np.ndarray*) – Predicted values.
- **average** (*str | None, optional*) – Averaging type for score (default None). For more detail about averaging see the documentation of sklearn.metrics.precision_score.

Returns
Precision scores for binary, multi-class or multi-lable classification.

Return type
float | np.ndarray

See also

f1_score, *false_discovery_rate*, *negative_predictive_value*

Examples

Computation of precision score for binary classification, i.e., a one dimensional array with only 2 classes

```
>>> target = np.array([0, 0, 1, 0, 1, 0, 0, 1, 0, 1])
>>> prediction = np.array([1, 1, 0, 1, 1, 1, 0, 1, 1, 1])
>>> precision_score(target, prediction)
0.375
```

Computation of precision score for a multi-class scenario, i.e., a 1D array with more then two classes

```
>>> target = np.array([1, 2, 2, 2, 1, 2, 1, 0, 1, 1])
>>> prediction = np.array([1, 1, 2, 0, 0, 1, 1, 0, 0, 2])
>>> precision_score(target, prediction)
array([0.25, 0.5, 0.5])
```

Computation of precision score for a multi-lable scenario, i.e., a 2D array with with columns representing different labels and values 0 or 1 as entries

```
>>> target = np.array([[0, 1, 0], [1, 0, 1], [1, 1, 0], [1, 1, 1]])
>>> prediction = np.array([[1, 0, 1], [1, 1, 1], [0, 0, 1], [1, 1, 0]])
>>> precision_score(target, prediction)
array([0.33333333, 0.5, 0.66666667])
```

Computation of precision score for a multi-class scenario with averaging of F1-scores over the different classes

```
>>> target = np.array([1, 2, 2, 2, 1, 2, 1, 0, 1, 1])
>>> prediction = np.array([1, 1, 2, 0, 0, 1, 1, 0, 0, 2])
>>> precision_score(target, prediction, average='micro')
0.4
```

qumphy.metrics.**recall_score_threshold**(*target*, *prediction*, *recall_value*, *pos_label=1*, *greater_than=True*, *dtype=numpy.float32*)

Compute the classification threshold so that the recall score is closest to the specified value, but greater (or lower).

Default: The threshold is computed for the sensitivity score.

The threshold is set as the next floating point number after (before) the value of the prediction that needs to be classified positive (negative) to achieve the desired recall score.

Parameters

- **target** (*np.ndarray*) – Ground truth values.
- **prediction** (*np.ndarray*) – Model output predictions.
- **recall_value** (*float*) – The desired recall score.
- **pos_label** (*1 | 0, optional*) – 1 to compute the threshold for sensitivity, 0 for specificity.
- **greater_than** (*bool, optional*) – True to let the recall score be higher than recall_value, false to let the recall score be lower than recall_value.
- **dtype** (*np.dtype, optional*) – The data type of the threshold, by default np.float32

Returns
The classification threshold.

Return type
float

qumphy.metrics.**root_mean_square_error**(*target*, *prediction*, *axis=0*)

Compute the root mean square error (RMSE) between target and prediction.

Return type
float | np.ndarray

Parameters

- **target** (*np.ndarray*) – Ground truth values.
- **prediction** (*np.ndarray*) – Model output predictions.
- **axis** (*int, optional*) – Axis to sum over, by default 0.

Returns
Array of RMSE values (L^2-norms) over the specified axes.

See also

l2_norm
This is a wrapper for `l2_norm(target - prediction, axis=axis)`.

l1_norm, *mean_absolute_error*

Examples

```
>>> root_mean_square_error(np.array([1, 2, 3]), np.array([1, 2, 3]))
0.0
```

```
>>> target = np.random.normal(0, 1, (10, 5, 3, 2))
>>> prediction = np.random.normal(0, 1, (10, 5, 3, 2))
>>> root_mean_square_error(target, prediction, axis=1).shape  # norm over second␣
↪axis
(10, 3, 1)
```

qumphy.metrics.**sensitivity**(*target*, *prediction*)

Compute the sensitivity (or true positive rate). Sensitivity is also known as the recall score of the positive class.

Return type
float

Parameters

- **target** (*np.ndarray*) – Ground truth values.
- **prediction** (*np.ndarray*) – Model output predictions.

Returns
Sensitivity score.

Return type
float

See also

specificity

qumphy.metrics.**specificity**(*target*, *prediction*)

Compute the specificity (or true negative rate). Specificity is also known as the recall score of the negative class.

Return type
float

Parameters

- **target** (*np.ndarray*) – Ground truth values.
- **prediction** (*np.ndarray*) – Model output predictions.

Returns
Specificity score.

Return type
float

See also

sensitivity

3.5.2.2 qumphy.trainer module

File: qumphy/trainer.py Project: 22HLT01 QUMPHY Contact: oskar.pfeffer@ptb.de Gitlab: https://gitlab.com/qumphy Description: Lightning Trainer.

class qumphy.trainer.**Trainer**(*config*)

Bases: object

Training pipeline wrapper for QuMPhy experiments.

This class loads the trainer, data module, feature extractor, model, and optional ensemble configuration from a configuration dictionary.

add_parameters_to_experiment()

base_config()

best_model_path(*trainer*)

Get the best checkpoint path from a trainer.

Parameters
trainer (*lightning.Trainer*) – Lightning trainer whose callbacks are searched.

Returns
Best model checkpoint path if a ModelCheckpoint callback is found.

Return type
str or None

extract_features()

Extract features from the train, validation, and test datasets.

Returns
The function replaces each dataset's data with extracted features and keeps the corresponding labels.

Return type
None

find_lr()

fit()

Train the model or ensemble members.

Returns
The function starts the Lightning training loop.

Return type
None

load_data_module()

Load data module from config.

load_ensemble_model()

Load trained ensemble members into an ensemble model.

Returns
The function loads the best checkpoint for each ensemble member and stores the combined ensemble model.

Return type
None

load_feature_extractor()

load_model()

load_trainer()

load_tuner()

Load the Lightning tuner.

Returns
The function creates a tuner if it does not already exist.

Return type
None

predict()

Predict using the best model and save the predictions to “best_predictions.pt”.

seed_everything()

Set random seeds for reproducibility.

Returns
The function calls the Lightning seed utility.

Return type
None

set_save_dir()

The save directory and project name are passed to the config of the model_checkpoint and logger.

set_sweep_parameters()

Set sweep parameters inside the nested configuration.

Returns
The function modifies the configuration dictionary in place.

Return type
None

test()

3.5.2.3 qumphy.uq module

File: qumphy/uq.py Project: 22HLT01 QUMPHY Contact: oskar.pfeffer@ptb.de Gitlab: https://gitlab.com/qumphy Description: Uncertainty quantification utilities.

qumphy.uq.**deep_ensemble**(*models*, *data*, *weights=None*)

Compute deep ensemble of the data using the given models.

Return type
np.ndarray

Parameters

- **models** (*list*) – List of callable models. Expected to return type np.ndarray.
- **data** (*np.ndarray*) – Input data.
- **weights** (*np.ndarray, optional*) – Weights for each model.

Returns
Weighted model output predictions.

Return type
np.ndarray

Examples

Compute unweighted deep ensemble prediction of two models on given data.

```
>>> model0 = lambda x : np.dot(np.zeros((1, 2)), x.T).reshape(-1, 1)
>>> model1 = lambda x : np.dot(np.ones((1, 2)), x.T).reshape(-1, 1)
>>> data = np.array([[0.0, 0.0], [0.0, 1.0], [1.0, 0.0], [1.0, 1.0]])
>>> deep_ensemble([model0, model1], data)
[[0. ]
 [0.5]
```

(continues on next page)

(continued from previous page)

```
 [0.5]
 [1. ]]
```

Compute weighted deep ensemble prediction of two models on given data.

```
>>> model0 = lambda x : np.dot(np.zeros((1, 2)), x.T).reshape(-1, 1)
>>> model1 = lambda x : np.dot(np.ones((1, 2)), x.T).reshape(-1, 1)
>>> data = np.array([[0.0, 0.0], [0.0, 1.0], [1.0, 0.0], [1.0, 1.0]])
>>> weights=np.array([0.0, 1.0])
>>> deep_ensemble([model0, model1], data)
[[0.]
 [1.]
 [1.]
 [2.]]
```

qumphy.uq.**deep_ensemble_gaussian**(*prediction_mean*, *prediction_var*, *weights=None*)

Compute deep ensemble using the given predictions.

Return type
typing.Tuple[np.ndarray, np.ndarray]

Parameters

- **prediction_mean** (*np.ndarray*) – Mean of the predicted gaussian distribution.
- **prediction_std** (*np.ndarray*) – Variance of the predicted gaussian distribution.
- **weights** (*np.ndarray, optional*) – Weights for each model.

Returns
Weighted ensemble prediction mean and variance.

Return type
Tuple[np.ndarray, np.ndarray]

Examples

Compute deep ensemble using the given predictions. Shape of *prediction_mean* and *prediction_var* should be:

[#models, #samples, #outputs]

```
>>> prediction_mean_1 = np.array([[[0.0], [0.5]], [[0.5], [1.0]]])
>>> prediction_mean_2 = np.array([[[1.0], [0.5]], [[0.5], [1.0]]])
>>> prediction_var_1 = np.array([[[0.0], [1.0]], [[0.5], [1.0]]])
>>> prediction_var_2 = np.array([[[0.0], [0.0]], [[0.5], [1.0]]])
>>> prediction_mean = np.array([prediction_mean_1, prediction_mean_2])
>>> prediction_var = np.array([prediction_var_1, prediction_var_2])
>>> weights = np.array([0.25, 0.75])
>>> deep_ensemble_gaussian(prediction_mean, prediction_var, weights)
```

3.5.2.4 qumphy.uq_metrics module

File: qumphy/uq_metrics.py Project: QUMPHY Contact: vivek.desai@npl.co.uk Gitlab: https://gitlab.com/qumphy Description: Evaluation metrics for assessing model calibration for classification and regression tasks.

qumphy.uq_metrics.**adaptive_calibration_error**(*predictions*, *targets*, **args*, *range_number=15*, *threshold=0.0*)

Calculate the Adaptive Calibration Error (ACE) for the predicted class probabilities. This metric will work for both the binary and multiclass classification cases, with the option to apply thresholding as done in the original paper.

References:

The ACE is introduced in the following paper: <https://arxiv.org/pdf/1904.01685>. This implementation is heavily inspired by the more general calibration error functions in: <https://github.com/JeremyNixon/uncertainty-metrics-1/tree/master>. We present a simplified version here, focused on the ACE alone.

Parameters:

predictions: np.ndarray
Model output probabilities with shape (n, m) for classification. Note m >= 2 i.e. probabilities given for all classes.

targets: np.ndarray
Target class labels with shape (n, 1)

***args:**
Used to catch any un-used arguments in the main UQ evaluation notebook.

range_number: int, optional
Number of ranges to be used (note: ranges are equal frequency bins in the mean confidence axis), by default 15

threshold: float, optional
Threshold value below which probabilities are ignored, by default 0.0 for the ACE i.e. no thresholding.

Returns:

:

ACE: float
The ACE metric value.

qumphy.uq_metrics.**continuous_ranked_probability_score**(*predictions*, *targets*, *uncertainties*, *converter='Gaussian'*, *distributions=None*, *is_var=True*)

Calculate the Continuous Ranked Probability Score (CRPS) from the given model predictions, uncertainties, and corresponding ground truths. The option is given to calculate the CRPS either using the analytical expression given the model predictions parameterising a Gaussian, or using numerical integration. The numerical integration method for a distribution estimated with KDE yields errors of <1% compared to the analytical expression for a Gaussian.

Return type
`float`

References:

The CRPS and its analytical expression for a univariate Gaussian is given in Gneiting et al., 2007: <https://sites.stat.washington.edu/raftery/Research/PDF/Gneiting2007jasa.pdf>.

Parameters:

predictions (np.ndarray):
An array of the model predictions with shape (n, 1).

targets (np.ndarray):
Array of target values with shape (n, 1).

uncertainties (np.ndarray):
Array of the total uncertainties associated with the model predictions from a chosen UQ method, with shape (n, 1).

converter (str, optional):
Specify whether the converter used was "Gaussian" or "KDE"; evaluates CRPS using numerical integration of CDF if "KDE", otherwise uses analytical expression for Gaussian. Defaults to "Gaussian".

distributions (List[distribution objects], optional):
Option to include the list of distributions as a direct input - needed if input is from KDE conversion of prediction interval to distribution. Defaults to None.

is_var (bool, optional):
Boolean parameter that sets whether to convert the uncertainties from variances to standard deviations. Defaults to True.

Returns:

:

CRPS (float):
The average CRPS value across the dataset.

qumphy.uq_metrics.**coverage_calibration_error**(*predictions*, *targets*, *uncertainties*, *filepath=None*, *is_var=True*, *save_fig=False*, *number_of_intervals=10*)

Calculate the confidence interval-based metric for model predictions and their corresponding uncertainties. If uncertainties are given as variances, they are converted to standard deviations.

Parameters:

predictions: np.ndarray
Array of model predictions with shape (n, 1)

targets: np.ndarray
Array of target values with shape (n, 1)

uncertainties: np.ndarray
Array of total uncertainties (aleatoric + epistemic) with shape (n, 1)

filepath: str
Path to directory to save the coverage calibration curve.

is_var: bool, optional
Boolean parameter that determines whether to convert the uncertainties from variances to standard deviations, by default True

save_fig: bool, optional
Boolean parameter that sets whether to save the plot to the filepath, by default True.

number_of_intervals: int, optional
The number of confidence intervals to evaluate the calibration error on, by default 10 (range 0 to 1 in steps of 0.1)

Returns:

:

calibration_error: float
The confidence interval-based calibration error metric

qumphy.uq_metrics.**entropy**(*p*)

Calculate the normalised entropy of the probabilities array.

Return type
ndarray

Parameters:

p (np.ndarray):
Probabilities from classifier.

Returns:

:

ents (np.ndarray):
Entropy values for each sample in the probability array.

qumphy.uq_metrics.**expected_calibration_error**(*predictions*, *targets*, *filepath=None*, *save_fig=False*, *bin_number=15*)

Calculate the Expected Calibration Error (ECE) for the predicted class model probabilities for classification tasks.

References:

The script below is heavily based on the following implementations of the ECE: <https://medium.com/towards-data-science/expected-calibration-error-ece-a-step-by-step-visual-explanation-with-python-code-c3e9aa12937d>

and

<https://github.com/gpleiss/temperature_scaling/blob/master/temperature_scaling.py>.

There exists alternative packages that calculate an ECE value, such as the netcal package. Depending on the implementation of binning in each package, the ECE values differ by small amounts (~1e-3). In the QUMPHY evaluation framework, the quoted ECE value is determined using this function.

Parameters:

predictions: np.ndarray
Model output probabilities with shape (n, k) for k-class classification.

targets: np.ndarray
Target class labels with shape (n, 1).

filepath: str
The filepath for the directory to save the reliability diagram to.

save_fig: bool
Boolean to determine whether to save the reliability diagram to the given filepath.

bin_number: int, optional
Number of bins to be used (note: bins are equal width in the mean confidence axis), by default 15.

Returns:

:

ECE: float
ECE value for the input data

qumphy.uq_metrics.**expected_cumulative_calibration_errors**(*predictions*, *targets*, *filepath*, *save_fig=True*)

Calculating the ECCE-MAD (aka Kolomogorv-Smirnov statistic) and ECCE-R (aka Kuiper statistic) metrics. The saved plot shows the deviation from zero of the cumulative errors.

References:

The ECCE metrics are introduced in the following: <https://arxiv.org/pdf/2205.09680>. This implementation is from the cumulative function in: <https://github.com/facebookresearch/ecevecce/blob/main/codes/calibration.py>.

The alterations to the cumulative function simply involve editing the format of the inputs to the function to be consistent with the other metrics. In addition, the normalised ECCE_MAD and ECCE_R metrics are given, as this is easier to compare with the target limits of convergence under the null hypothesis of perfect calibration.

Parameters:

predictions: np.ndarray
Model output probabilities with shape (n, 2) for binary classification

targets: np.ndarray
Target class labels with shape (n, 1)

filepath: str
Path to directory to save the plot of the cumulative calibration errors.

save_fig: bool
Boolean to determine whether to save the reliability diagram to the given filepath.

Returns:

:

norm_ECCE-MAD: float
The maximum absolute deviation from zero statistic (Kolmogorov-Smirnov) for the cumulative differences between the confidences and outcomes

norm_ECCE-R: float
The range of deviation from zero statistic (Kuiper) for the cumulative differences between the confidences and outcomes

statistical_significance_scale: float
Normalisation factor that is used to assess statistically significant deviations from the asymptotic expected values (under large sample size n)

qumphy.uq_metrics.**expected_normalised_calibration_error**(*predictions*, *targets*, *uncertainties*, *filepath=None*, *is_var=False*, *save_fig=False*, *bin_number=15*)

Calculating the Expected Normaised Calibration Error (ENCE) for the model predictions and the corresponding total uncertainties.

Parameters:

predictions: np.ndarray
An array of the model predictions with shape (n, 1)

targets: np.ndarray
Array of target values with shape (n, 1)

uncertainties: np.ndarray
Array of the total uncertainties associated with the model predictions from a chosen UQ method, with shape (n, 1)

filepath: str
Path of directory to save the reliability diagram.

is_var: bool, optional
Boolean parameter that sets whether to convert the uncertainties to standard deviations, by default True

save_fig: bool, optional
Boolean parameter that sets whether to save the plot to the filepath, by default True.

bin_number: int, optional
Chosen number of bins for calculating ENCE (note: the bins are defined to have an equal number of samples per bin), by default 15

Returns:

:

ENCE: float
The value of the ENCE calculated for the given inputs

qumphy.uq_metrics.**flatten_list_of_batches**(*output_list*, *concat_axis=-1*)

Function to flatten model output arrays before they are used to calculate the calibration error metrics.

Parameters:

output_list: list
List to be flattened into 1D np.ndarray

concat_axis: int, optional
Axis on which to perform the flattening, by default -1 (last axis)

Returns:

: flattened_array: np.ndarry

Flattened array

qumphy.uq_metrics.**negative_log_likelihood**(*predictions*, *targets*, **args*)

Compute the negative log likelihood metric for the predicted probabilities for k-class classification. Values are clipped to avoid divide by zero errors when taking the base-2 logarithm.

Return type
`float`

Parameters:

predictions: np.ndarray
Model output probabilities with shape (n, k) for k-class classification.

targets: np.ndarray
Target class labels with shape (n, 1).

***args:**
Used to capture other arguments passed to the function that are not necessary e.g. filepath for reliability diagram.

Returns:

:
nll (float): Calculated NLL value.

qumphy.uq_metrics.**picp**(*predictions*, *targets*, *confidence_level*, *is_interval=True*, *uncertainties=None*, *is_var=True*)

Calculate the Prediction Interval Coverage Probability (PICP) for predictions given either as intervals or point-predictions, for a specified confidence level. If inputs are given as point-predictions, uncertainty (as variance/standard deviations) arrays must be given, from which intervals are derived, assuming a Gaussian distribution.

The Mean Prediction Interval Width (MPIW) is also returned. Note this is an unbounded non-negative metric.

Return type
`tuple`

Parameters:

predictions: np.ndarray
Array of model predictions with shape either (n, 2) for intervals or (n, 1) for point predictions

targets: np.ndarray
Array of target values with shape (n, 1)

confidence_level: float
The confidence level of the prediction intervals. If point-predictions are given, intervals will be created assuming Gaussian distribution at this confidence level.

is_interval: bool
Boolean flag to set whether predictions are given as intervals or point-predictions. Defaults to True.

uncertainties: np.ndarray
Array of model uncertainties with shape (n, 1). Defaults to None, must be present if is_interval is False.

is_var: bool
Bollean flag corresponding to if the uncertainties are given as variances (default) or standard deviations. Defaults to True.

Returns:

:

picp: float
The prediction interval coverage probability of the model predictions. Aim is for the PICP to match the confidence level.

MPIW: float
The Mean Predicted Interval Wdith of the model predictions.

qumphy.uq_metrics.**set_coverage**(*prediction_sets*, *targets*)

Calculate the prediction set coverage for model ouputs given from either basic conformal prediction, or sets created using top-k selection from model probabilities. Note: top-k selection from model probabilities has no coverage guarantee, in contrast to conformal methods.

Parameters:

prediction_sets (np.ndarray(List)): Array of lists containing the predicted classes. targets (np.ndarry): The ground truth class labels.

Returns:

:
float: The empirical frequency of the ground truth class labels in the set.

qumphy.uq_metrics.**smooth_expected_calibration_error**(*predictions*, *targets*, *filepath*, *save_fig=True*)

Calculate the smooth ECE value (smECE) for the predicted class probabilities for the classification task.

Return type
`float`

References:

Details on the relplot python package can be found in the following repository: <https://github.com/apple/ml-calibration/tree/main>. The accompanying paper for the smooth ECE metric, and the smooth reliability diagrams can be found here: <https://arxiv.org/pdf/2309.12236>.

Parameters:

predictions: np.ndarray
Model output probabilities with shape (n, k) for k class classification.

targets: np.ndarray
Target class labels with shape (n, 1).

filepath: str
Path to directory to save the reliability diagram.

save_fig: bool
Boolean to determine whether to save the reliability diagram to the given filepath.

Returns:

:

smECE: float
The calculated smECE value using the relplot python package.

qumphy.uq_metrics.**split_into_classes**(*predictions*, *targets*)

Helper function to split classification predictions into predictions conditioned on the ground truth class label.

Return type
tuple

Parameters:

predictions: np.ndarray
Model output probabilities with shape (n, k) for k-class classification.

targets: np.ndarray
Target class labels with shape (n, 1).

returns
Predictions per-class for k classes, with corresponding ground truth class labels for each class. Size of tuple is dependent on number of classes.

rtype
tuple

qumphy.uq_metrics.**uncertainty_calibration_error**(*predictions*, *targets*, *filepath=None*, *save_fig=False*, *prediction_entropies=None*, *bin_number=15*)

Calculate the Uncertainty Calibration Error (UCE) for model predictions and target labels for classification. There exists the optional argument to provide an array of entropies of shape (n, 1); if None, the entropies will be calculated from the probabilities.

References:

The UCE was introduced in the following paper: <https://arxiv.org/pdf/1909.13550>. This implementation is based on: <https://github.com/mlaves/bayesian-temperature-scaling/blob/master/uce.py>.

Parameters:

predictions: np.ndarray
Model output probabilities with shape (n, k) for k-class classification.

targets: np.ndarray
Target class labels with shape (n, 1).

filepath: str
Path to directory to save the reliability diagram.

prediction_entropies: np.ndarray, optional
The entropies for the predictions, by default None; if None, then the entropies are calculated from the prediction probabilities.

bin_number: int, optional
Number of bins to be used (note: bins are equal width in the normalised entropy axis), by default 15.

Returns:

:

UCE: float
The UCE for the model predictions and their corresponding entropies.

xs: np.ndarray
Array of the mean normalised predicted entropies, used for plotting the reliability diagram.

ys: np.ndarray
Array of the empirical inaccuracy per bin, used for the plotting the reliability diagram.

qumphy.uq_metrics.**variation_calibration_error**(*predictions*, *targets*, *filepath*, *save_fig=True*, *bin_number=15*)

Compute the Variation Calibration Error (VCE) metric. Extends the ECE to a measure of distributional calibration for multiclass problems. Measure of variation employed for this metric is the Shannon entropy. Bins can be defined with either equal-widths or equal-frequency.

Return type
Tuple[float, ndarray, ndarray]

Parameters:

predictions (np.ndarray):
Prediction array with shape (n, k) for n samples and k classes.

targets (np.ndarray):
Target class labels with shape (n, 1).

save_fig (bool, default = False):
Flag to save the reliability diagram.

filepath (str, default = os.getcwd()):
Filepath to which the reliability diagram is saved.

bin_number (int, default = 10):
Number of bins used to compute metric and plot reliability diagram.

Returns:

: VCE (float):

VCE metric value.

xs (np.ndarray):
Mean entropy of predictions array for plotting the reliability diagram.

ys (np.ndarray):
Entropy of the rank distribution array for plotting the reliability diagram.

qumphy.uq_metrics.**z_variance_error**(*predictions*, *targets*, *uncertainties*, *is_var=True*, *bin_number=15*)

Calculate the Z-variance error (ZVE) for the model predictions and corresponding total uncertainties.

Parameters:

predictions: np.ndarray
An array of the model predictions with shape (n, 1)

targets: np.ndarray
Array of target values with shape (n, 1)

uncertainties: np.ndarray
Array of the total uncertainties associated with the model predictions from a chosen UQ method, with shape (n, 1)

is_var: bool, optional
Boolean parameter that sets whether to convert the uncertainties from variances to standard deviations, by default True

bin_number: int, optional
Chosen number of bins for calculating ZVE (note: the bins are defined to have an equal number of samples per bin), by default 15

Returns:

:

ZVE: float
The ZVE value for the input predictions and uncertainties

PYTHON MODULE INDEX

a

app, 14
app.deepbeat_converter, 14
app.deepbeat_data_visualization, 14
app.gui, 15
app.mimic3bp_converter, 16
app.mimic3bp_data_visualization, 17
app.pulsedb_statistics, 17
app.train, 18
app.tutorial_metrics, 19

q

qumphy, 20
qumphy.callbacks, 20
qumphy.callbacks.base_logging, 20
qumphy.callbacks.deepbeat, 20
qumphy.callbacks.progressbar, 21
qumphy.callbacks.pulsedb, 22
qumphy.callbacks.sleepapnea, 22
qumphy.data, 23
qumphy.data.attractor, 26
qumphy.data.deepbeat, 27
qumphy.data.pulsedb, 29
qumphy.data.signal_preprocessing, 23
qumphy.data.signal_preprocessing.filters, 23
qumphy.data.signal_preprocessing.noise, 24
qumphy.data.signal_preprocessing.resampling, 26
qumphy.data.sleepapnea, 31
qumphy.data.utils, 32
qumphy.evaluate, 32
qumphy.evaluate.deepensembles, 32
qumphy.metrics, 78
qumphy.misc, 35
qumphy.misc.misc, 35
qumphy.misc.output_conversions, 37
qumphy.models, 39
qumphy.models.alexnet, 43
qumphy.models.apnea, 44
qumphy.models.basic_conv1d, 45
qumphy.models.deep_ensemble, 51

qumphy.models.deepbeat, 52
qumphy.models.dumbnet, 55
qumphy.models.inception1d, 55
qumphy.models.inception_mantas, 58
qumphy.models.itransformer, 59
qumphy.models.minirocket, 60
qumphy.models.ppnet, 62
qumphy.models.pulsedb, 62
qumphy.models.s42, 64
qumphy.models.s4_model, 70
qumphy.models.timesnet, 70
qumphy.models.utils, 39
qumphy.models.utils.conv_blocks, 39
qumphy.models.utils.embed, 40
qumphy.models.utils.kgactivation, 40
qumphy.models.utils.kgloss, 41
qumphy.models.utils.masking, 41
qumphy.models.utils.mcdropout, 42
qumphy.models.utils.pinballloss, 42
qumphy.models.utils.selfattention_family, 42
qumphy.models.utils.transformer_encdec, 43
qumphy.models.xresnet1d, 72
qumphy.trainer, 90
qumphy.uq, 92
qumphy.uq_metrics, 93

INDEX

A

Activation() (*in module qumphy.models.s42*), 64
adaptive_calibration_error() (*in module qumphy.uq_metrics*), 93
AdaptiveConcatPool1d (*class in qumphy.models.basic_conv1d*), 45
add_noise() (*in module qumphy.data.signal_preprocessing.noise*), 24
add_parameters_to_experiment() (*qumphy.trainer.Trainer method*), 90
AlexNet1D (*class in qumphy.models.alexnet*), 43
AlexNet1D_MCD (*class in qumphy.models.alexnet*), 44
all_binary_metrics() (*in module qumphy.metrics*), 78
all_regression_metrics() (*in module qumphy.metrics*), 78
anomaly_detection() (*qumphy.models.itransformer.iTransformer method*), 59
anomaly_detection() (*qumphy.models.timesnet.TimesNet method*), 71
app
    module, 14
app.deepbeat_converter
    module, 14
app.deepbeat_data_visualization
    module, 14
app.gui
    module, 15
app.mimic3bp_converter
    module, 16
app.mimic3bp_data_visualization
    module, 17
app.pulsedb_statistics
    module, 17
app.train
    module, 18
app.tutorial_metrics
    module, 19
append_nested_dicts() (*in module qumphy.evaluate.deepensembles*), 34
apply_filter() (*in module qumphy.data.signal_preprocessing.filters*), 23
AttentionLayer (*class in qumphy.models.utils.selfattention_family*), 42
AttractorDataModule (*class in qumphy.data.attractor*), 26
AttractorDataset (*class in qumphy.data.attractor*), 27
auc_score_binary() (*in module qumphy.metrics*), 79
auc_score_multiclass() (*in module qumphy.metrics*), 79

B

balanced_accuracy_score() (*in module qumphy.metrics*), 80

bandpass_filter() (*in module qumphy.data.signal_preprocessing.filters*), 23
base_config() (*qumphy.trainer.Trainer method*), 90
baseline_wander_noise() (*in module qumphy.data.signal_preprocessing.noise*), 24
BaseLoggingCallback (*class in qumphy.callbacks.base_logging*), 20
basic1d() (*in module qumphy.models.basic_conv1d*), 47
Basic_Conv1d (*class in qumphy.models.basic_conv1d*), 45
Batch (*qumphy.models.xresnet1d.NormType attribute*), 73
batch() (*in module qumphy.misc.misc*), 35
BatchNorm() (*in module qumphy.models.xresnet1d*), 72
BatchZero (*qumphy.models.xresnet1d.NormType attribute*), 73
best_model_path() (*qumphy.trainer.Trainer method*), 91
beta (*qumphy.data.signal_preprocessing.resampling.MatlabResampleConfig attribute*), 26
bilinear() (*in module qumphy.models.s42*), 67
bn_drop_lin() (*in module qumphy.models.basic_conv1d*), 48
brier_score() (*in module qumphy.metrics*), 81

C

calculate_baseline_measures() (*qumphy.data.pulsedb.PulseDBDataset method*), 29
calculate_metrics() (*qumphy.evaluate.deepensembles.DeepEnsembleEvaluate method*), 33
calculate_regression_baseline() (*in module qumphy.data.utils*), 32
calculate_target_stats() (*qumphy.data.pulsedb.PulseDBDataset method*), 29
cauchy_conj() (*in module qumphy.models.s42*), 67
classification() (*qumphy.models.itransformer.iTransformer method*), 59
classification() (*qumphy.models.timesnet.TimesNet method*), 71
configure_optimizers() (*qumphy.models.apnea.SleepApneaModule method*), 44
configure_optimizers() (*qumphy.models.deepbeat.DeepBeatModule method*), 53
configure_optimizers() (*qumphy.models.pulsedb.PulseDBModule method*), 63
continuous_ranked_probability_score() (*in module qumphy.uq_metrics*), 94
conv() (*in module qumphy.models.inception1d*), 57
convert() (*in module app.deepbeat_converter*), 14
convert() (*in module app.mimic3bp_converter*), 16
convert_prediction_intervals() (*in module qumphy.misc.output_conversions*), 37
ConvLayer (*class in qumphy.models.utils.transformer_encdec*), 43
ConvLayer (*class in qumphy.models.xresnet1d*), 73
coverage_calibration_error() (*in module qumphy.uq_metrics*), 95
create_head1d() (*in module qumphy.models.basic_conv1d*), 48
cval (*qumphy.data.signal_preprocessing.resampling.MatlabResampleConfig attribute*), 26

D

d_output (*qumphy.models.s42.S4 property*), 65
d_state (*qumphy.models.s42.S4 property*), 65
DataEmbedding (*class in qumphy.models.utils.embed*), 40
DataEmbedding_inverted (*class in qumphy.models.utils.embed*), 40
DataEmbedding_wo_pos (*class in qumphy.models.utils.embed*), 40
Decoder (*class in qumphy.models.utils.transformer_encdec*), 43
DecoderLayer (*class in qumphy.models.utils.transformer_encdec*), 43
deep_ensemble() (*in module qumphy.uq*), 92
deep_ensemble_gaussian() (*in module qumphy.uq*), 93
DeepBeatDataModule (*class in qumphy.data.deepbeat*), 27
DeepBeatDataset (*class in qumphy.data.deepbeat*), 28
DeepBeatEnsemble (*class in qumphy.models.deepbeat*), 52
DeepBeatEvaluation (*class in qumphy.evaluate.deepensembles*), 32
DeepBeatLogging (*class in qumphy.callbacks.deepbeat*), 20
DeepBeatLogging_KGLoss (*class in qumphy.callbacks.deepbeat*), 21

DeepBeatLogging_MCD (*class in qumphy.callbacks.deepbeat*), 21
DeepBeatModule (*class in qumphy.models.deepbeat*), 53
DeepBeatModule_MCD (*class in qumphy.models.deepbeat*), 54
DeepEnsemble (*class in qumphy.models.deep_ensemble*), 51
DeepEnsembleEvaluate (*class in qumphy.evaluate.deepensembles*), 33
default_state() (*qumphy.models.s42.HippoSSKernel method*), 65
default_state() (*qumphy.models.s42.S4 method*), 65
default_state() (*qumphy.models.s42.SSKernelNPLR method*), 66
denormalize() (*qumphy.evaluate.deepensembles.PulseDBEvaluation method*), 33
denormalize() (*qumphy.models.pulsedb.PulseDBEnsemble method*), 62
denormalize_std() (*qumphy.models.pulsedb.PulseDBModule method*), 63
denormalize_target() (*qumphy.models.pulsedb.PulseDBModule method*), 63
disable() (*qumphy.callbacks.progressbar.EpochProgressBar method*), 21
double_length() (*qumphy.models.s42.SSKernelNPLR method*), 66
DumbNet (*class in qumphy.models.dumbnet*), 55

E

embed_c2r() (*in module qumphy.models.s42*), 68
enable() (*qumphy.callbacks.progressbar.EpochProgressBar method*), 21
Encoder (*class in qumphy.models.utils.transformer_encdec*), 43
EncoderLayer (*class in qumphy.models.utils.transformer_encdec*), 43
entropy() (*in module qumphy.uq_metrics*), 96
EpochProgressBar (*class in qumphy.callbacks.progressbar*), 21
eval_argument_parser() (*in module qumphy.misc.misc*), 35
eval_torch_model_by_numpy_ndarray() (*in module qumphy.misc.misc*), 35
evaluation_function() (*qumphy.evaluate.deepensembles.DeepBeatEvaluation method*), 32
evaluation_function() (*qumphy.evaluate.deepensembles.PulseDBEvaluation method*), 34
expected_calibration_error() (*in module qumphy.uq_metrics*), 96
expected_cumulative_calibration_errors() (*in module qumphy.uq_metrics*), 97
expected_normalised_calibration_error() (*in module qumphy.uq_metrics*), 97
extra_function() (*qumphy.evaluate.deepensembles.PulseDBEvaluation method*), 34
extract_features() (*qumphy.models.minirocket.MiniRocketFeatures method*), 61
extract_features() (*qumphy.trainer.Trainer method*), 91

F

f1_score() (*in module qumphy.metrics*), 81
false_discovery_rate() (*in module qumphy.metrics*), 82
fcn() (*in module qumphy.models.basic_conv1d*), 49
fcn_wang() (*in module qumphy.models.basic_conv1d*), 49
FFT_for_Period() (*in module qumphy.models.timesnet*), 70
find_lr() (*qumphy.trainer.Trainer method*), 91
fit() (*qumphy.trainer.Trainer method*), 91
fit_length() (*qumphy.models.utils.selfattention_family.ReformerLayer method*), 43
fitting (*qumphy.models.minirocket.MiniRocketFeatures attribute*), 61
FixedEmbedding (*class in qumphy.models.utils.embed*), 40
flash_attention_forward() (*qumphy.models.utils.selfattention_family.FlashAttention method*), 42
FlashAttention (*class in qumphy.models.utils.selfattention_family*), 42
Flatten (*class in qumphy.models.basic_conv1d*), 46
flatten_list_of_batches() (*in module qumphy.uq_metrics*), 98
FlowAttention (*class in qumphy.models.utils.selfattention_family*), 42
forecast() (*qumphy.models.itransformer.iTransformer method*), 59
forecast() (*qumphy.models.timesnet.TimesNet method*), 72
forward() (*qumphy.models.alexnet.AlexNet1D method*), 43
forward() (*qumphy.models.alexnet.AlexNet1D_MCD method*), 44

forward() (*qumphy.models.apnea.SleepApneaModule method*), 44
forward() (*qumphy.models.basic_conv1d.AdaptiveConcatPool1d method*), 45
forward() (*qumphy.models.basic_conv1d.Flatten method*), 46
forward() (*qumphy.models.basic_conv1d.LambdaLayer method*), 47
forward() (*qumphy.models.basic_conv1d.SqueezeExcite1d method*), 47
forward() (*qumphy.models.deep_ensemble.DeepEnsemble method*), 51
forward() (*qumphy.models.deepbeat.DeepBeatEnsemble method*), 52
forward() (*qumphy.models.deepbeat.DeepBeatModule method*), 53
forward() (*qumphy.models.dumbnet.DumbNet method*), 55
forward() (*qumphy.models.inception1d.Inception1d method*), 55
forward() (*qumphy.models.inception1d.InceptionBackbone method*), 56
forward() (*qumphy.models.inception1d.InceptionBlock1d method*), 57
forward() (*qumphy.models.inception1d.Shortcut1d method*), 57
forward() (*qumphy.models.inception_mantas.Inception1DBlock method*), 58
forward() (*qumphy.models.inception_mantas.Inception1DNet method*), 58
forward() (*qumphy.models.itransformer.iTransformer method*), 59
forward() (*qumphy.models.minirocket.MiniRocket method*), 60
forward() (*qumphy.models.minirocket.MiniRocketFeatures method*), 61
forward() (*qumphy.models.ppnet.PPNet method*), 62
forward() (*qumphy.models.pulsedb.PulseDBEnsemble method*), 62
forward() (*qumphy.models.pulsedb.PulseDBGaussianLoss method*), 63
forward() (*qumphy.models.pulsedb.PulseDBModule method*), 64
forward() (*qumphy.models.s42.HippoSSKernel method*), 65
forward() (*qumphy.models.s42.S4 method*), 65
forward() (*qumphy.models.s42.SSKernelNPLR method*), 66
forward() (*qumphy.models.s42.TransposedLinear method*), 67
forward() (*qumphy.models.s4_model.S4Model method*), 70
forward() (*qumphy.models.timesnet.TimesBlock method*), 70
forward() (*qumphy.models.timesnet.TimesNet method*), 72
forward() (*qumphy.models.utils.conv_blocks.Inception_Block_V1 method*), 39
forward() (*qumphy.models.utils.conv_blocks.Inception_Block_V2 method*), 40
forward() (*qumphy.models.utils.embed.DataEmbedding method*), 40
forward() (*qumphy.models.utils.embed.DataEmbedding_inverted method*), 40
forward() (*qumphy.models.utils.embed.DataEmbedding_wo_pos method*), 40
forward() (*qumphy.models.utils.embed.FixedEmbedding method*), 40
forward() (*qumphy.models.utils.embed.PatchEmbedding method*), 40
forward() (*qumphy.models.utils.embed.PositionalEmbedding method*), 40
forward() (*qumphy.models.utils.embed.TemporalEmbedding method*), 40
forward() (*qumphy.models.utils.embed.TimeFeatureEmbedding method*), 40
forward() (*qumphy.models.utils.embed.TokenEmbedding method*), 40
forward() (*qumphy.models.utils.kgactivation.KGActivation method*), 40
forward() (*qumphy.models.utils.kgloss.KGLoss method*), 41
forward() (*qumphy.models.utils.kgloss.KGLoss_unified_prediction method*), 41
forward() (*qumphy.models.utils.mcdropout.MCDropout method*), 42
forward() (*qumphy.models.utils.pinballloss.PinballLoss method*), 42
forward() (*qumphy.models.utils.selfattention_family.AttentionLayer method*), 42
forward() (*qumphy.models.utils.selfattention_family.FlashAttention method*), 42
forward() (*qumphy.models.utils.selfattention_family.FlowAttention method*), 42
forward() (*qumphy.models.utils.selfattention_family.FullAttention method*), 42
forward() (*qumphy.models.utils.selfattention_family.ProbAttention method*), 43
forward() (*qumphy.models.utils.selfattention_family.ReformerLayer method*), 43
forward() (*qumphy.models.utils.transformer_encdec.ConvLayer method*), 43
forward() (*qumphy.models.utils.transformer_encdec.Decoder method*), 43
forward() (*qumphy.models.utils.transformer_encdec.DecoderLayer method*), 43

forward() (*qumphy.models.utils.transformer_encdec.Encoder method*), 43
forward() (*qumphy.models.utils.transformer_encdec.EncoderLayer method*), 43
forward() (*qumphy.models.xresnet1d.MHSA1d method*), 73
forward() (*qumphy.models.xresnet1d.ResBlock method*), 73
forward() (*qumphy.models.xresnet1d.ResBlock_dropout method*), 73
FullAttention (*class in qumphy.models.utils.selfattention_family*), 42

G

gaussian_noise() (*in module qumphy.data.signal_preprocessing.noise*), 25
general_threshold() (*in module qumphy.metrics*), 83
get_data() (*qumphy.data.deepbeat.DeepBeatDataset method*), 28
get_data() (*qumphy.data.pulsedb.PulseDBDataset method*), 29
get_data() (*qumphy.data.sleepapnea.SleepApneaDataset method*), 31
get_initializer() (*in module qumphy.models.s42*), 68
get_labels() (*qumphy.data.deepbeat.DeepBeatDataset method*), 28
get_labels() (*qumphy.data.pulsedb.PulseDBDataset method*), 29
get_labels() (*qumphy.data.sleepapnea.SleepApneaDataset method*), 31
get_layer_groups() (*qumphy.models.basic_conv1d.Basic_Conv1d method*), 46
get_layer_groups() (*qumphy.models.inception1d.Inception1d method*), 56
get_layer_groups() (*qumphy.models.xresnet1d.XResNet1d method*), 74
get_minirocket_features() (*in module qumphy.models.minirocket*), 61
get_output_layer() (*qumphy.models.basic_conv1d.Basic_Conv1d method*), 46
get_output_layer() (*qumphy.models.inception1d.Inception1d method*), 56
get_output_layer() (*qumphy.models.xresnet1d.XResNet1d method*), 74
get_quantiles() (*qumphy.models.minirocket.MiniRocketFeatures method*), 61
get_target_stats() (*in module qumphy.data.pulsedb*), 30
get_target_stats() (*qumphy.data.pulsedb.PulseDBDataModule method*), 29
get_target_stats() (*qumphy.data.pulsedb.PulseDBDataset method*), 29

H

highpass_filter() (*in module qumphy.data.signal_preprocessing.filters*), 23
HippoSSKernel (*class in qumphy.models.s42*), 65

I

ieee_grades() (*in module qumphy.metrics*), 83
ieee_grades_torch() (*in module qumphy.metrics*), 84
imputation() (*qumphy.models.itransformer.iTransformer method*), 60
imputation() (*qumphy.models.timesnet.TimesNet method*), 72
Inception1d (*class in qumphy.models.inception1d*), 55
inception1d() (*in module qumphy.models.inception1d*), 57
Inception1DBlock (*class in qumphy.models.inception_mantas*), 58
Inception1DNet (*class in qumphy.models.inception_mantas*), 58
Inception_Block_V1 (*class in qumphy.models.utils.conv_blocks*), 39
Inception_Block_V2 (*class in qumphy.models.utils.conv_blocks*), 39
InceptionBackbone (*class in qumphy.models.inception1d*), 56
InceptionBlock1d (*class in qumphy.models.inception1d*), 56
init_cnn() (*in module qumphy.models.xresnet1d*), 74
init_default() (*in module qumphy.models.xresnet1d*), 75
instantiate_class() (*in module qumphy.misc.misc*), 35
instantiate_class_from_string() (*in module qumphy.misc.misc*), 36
intervals_to_probs() (*in module qumphy.misc.output_conversions*), 37
iTransformer (*class in qumphy.models.itransformer*), 59

K

kde_convert_interval() (*in module qumphy.misc.output_conversions*), 38
kernel_method() (*qumphy.models.utils.selfattention_family.FlowAttention method*), 42
kernel_size (*qumphy.models.minirocket.MiniRocketFeatures attribute*), 61
KGActivation (*class in qumphy.models.utils.kgactivation*), 40
KGLoss (*class in qumphy.models.utils.kgloss*), 41
KGLoss_unified_prediction (*class in qumphy.models.utils.kgloss*), 41
krylov() (*in module qumphy.models.s42*), 68

L

l1_norm() (*in module qumphy.metrics*), 84
l2_norm() (*in module qumphy.metrics*), 84
LambdaLayer (*class in qumphy.models.basic_conv1d*), 47
LinearActivation() (*in module qumphy.models.s42*), 65
list_conda_envs() (*in module app.gui*), 15
load_config() (*in module app.train*), 18
load_data() (*qumphy.data.deepbeat.DeepBeatDataset method*), 28
load_data() (*qumphy.data.pulsedb.PulseDBDataset method*), 29
load_data_module() (*qumphy.trainer.Trainer method*), 91
load_dataset() (*qumphy.evaluate.deepensembles.DeepEnsembleEvaluate method*), 33
load_ensemble_model() (*qumphy.trainer.Trainer method*), 91
load_evaluation_class() (*qumphy.evaluate.deepensembles.DeepEnsembleEvaluate method*), 33
load_feature_extractor() (*qumphy.trainer.Trainer method*), 91
load_model() (*qumphy.trainer.Trainer method*), 91
load_models() (*qumphy.models.deepbeat.DeepBeatEnsemble method*), 52
load_models() (*qumphy.models.pulsedb.PulseDBEnsemble method*), 62
load_predictions() (*qumphy.evaluate.deepensembles.DeepEnsembleEvaluate method*), 33
load_target_stats() (*qumphy.data.pulsedb.PulseDBDataset method*), 30
load_trainer() (*qumphy.trainer.Trainer method*), 91
load_tuner() (*qumphy.trainer.Trainer method*), 91
log_epoch_end() (*qumphy.callbacks.base_logging.BaseLoggingCallback method*), 20
log_epoch_end() (*qumphy.callbacks.deepbeat.DeepBeatLogging method*), 20
log_epoch_end() (*qumphy.callbacks.pulsedb.PulseDBLogging method*), 22
log_epoch_end() (*qumphy.callbacks.sleepapnea.SleepApneaLogging method*), 22
log_ieee_metrics() (*qumphy.callbacks.pulsedb.PulseDBLogging method*), 22
log_loss() (*qumphy.callbacks.base_logging.BaseLoggingCallback method*), 20
log_mae() (*qumphy.callbacks.pulsedb.PulseDBLogging method*), 22
log_rmse() (*qumphy.callbacks.pulsedb.PulseDBLogging method*), 22
log_std() (*qumphy.callbacks.pulsedb.PulseDBLogging method*), 22
lowpass_filter() (*in module qumphy.data.signal_preprocessing.filters*), 24

M

main() (*in module app.deepbeat_data_visualization*), 14
main() (*in module app.gui*), 16
main() (*in module app.mimic3bp_data_visualization*), 17
main() (*in module app.pulsedb_statistics*), 17
main() (*in module app.train*), 18
map_labels_to_classes() (*in module app.mimic3bp_converter*), 16
mask (*qumphy.models.utils.masking.ProbMask property*), 41
mask (*qumphy.models.utils.masking.TriangularCausalMask property*), 41
MatlabResampleConfig (*class in qumphy.data.signal_preprocessing.resampling*), 26
matthews_correlation_coefficient() (*in module qumphy.metrics*), 85
mc2ml() (*in module app.tutorial_metrics*), 19
MCDropout (*class in qumphy.models.utils.mcdropout*), 42

mean_absolute_error() (*in module qumphy.metrics*), 86
mean_absolute_scaled_error() (*in module qumphy.metrics*), 86
MHSA1d (*class in qumphy.models.xresnet1d*), 73
MiniRocket (*class in qumphy.models.minirocket*), 60
MiniRocketFeatures (*class in qumphy.models.minirocket*), 60
MiniRocketHead (*class in qumphy.models.minirocket*), 61
module
    app, 14
    app.deepbeat_converter, 14
    app.deepbeat_data_visualization, 14
    app.gui, 15
    app.mimic3bp_converter, 16
    app.mimic3bp_data_visualization, 17
    app.pulsedb_statistics, 17
    app.train, 18
    app.tutorial_metrics, 19
    qumphy, 20
    qumphy.callbacks, 20
    qumphy.callbacks.base_logging, 20
    qumphy.callbacks.deepbeat, 20
    qumphy.callbacks.progressbar, 21
    qumphy.callbacks.pulsedb, 22
    qumphy.callbacks.sleepapnea, 22
    qumphy.data, 23
    qumphy.data.attractor, 26
    qumphy.data.deepbeat, 27
    qumphy.data.pulsedb, 29
    qumphy.data.signal_preprocessing, 23
    qumphy.data.signal_preprocessing.filters, 23
    qumphy.data.signal_preprocessing.noise, 24
    qumphy.data.signal_preprocessing.resampling, 26
    qumphy.data.sleepapnea, 31
    qumphy.data.utils, 32
    qumphy.evaluate, 32
    qumphy.evaluate.deepensembles, 32
    qumphy.metrics, 78
    qumphy.misc, 35
    qumphy.misc.misc, 35
    qumphy.misc.output_conversions, 37
    qumphy.models, 39
    qumphy.models.alexnet, 43
    qumphy.models.apnea, 44
    qumphy.models.basic_conv1d, 45
    qumphy.models.deep_ensemble, 51
    qumphy.models.deepbeat, 52
    qumphy.models.dumbnet, 55
    qumphy.models.inception1d, 55
    qumphy.models.inception_mantas, 58
    qumphy.models.itransformer, 59
    qumphy.models.minirocket, 60
    qumphy.models.ppnet, 62
    qumphy.models.pulsedb, 62
    qumphy.models.s42, 64
    qumphy.models.s4_model, 70

qumphy.models.timesnet, 70
qumphy.models.utils, 39
qumphy.models.utils.conv_blocks, 39
qumphy.models.utils.embed, 40
qumphy.models.utils.kgactivation, 40
qumphy.models.utils.kgloss, 41
qumphy.models.utils.masking, 41
qumphy.models.utils.mcdropout, 42
qumphy.models.utils.pinballloss, 42
qumphy.models.utils.selfattention_family, 42
qumphy.models.utils.transformer_encdec, 43
qumphy.models.xresnet1d, 72
qumphy.trainer, 90
qumphy.uq, 92
qumphy.uq_metrics, 93

N

n (*qumphy.data.signal_preprocessing.resampling.MatlabResampleConfig attribute*), 26
negative_log_likelihood() (*in module qumphy.uq_metrics*), 98
negative_predictive_value() (*in module qumphy.metrics*), 87
noop() (*in module qumphy.models.inception1d*), 58
norm_convert_interval() (*in module qumphy.misc.output_conversions*), 38
normalize_data() (*qumphy.data.deepbeat.DeepBeatDataset method*), 28
normalize_data() (*qumphy.data.pulsedb.PulseDBDataset method*), 30
normalize_data() (*qumphy.data.sleepapnea.SleepApneaDataset method*), 31
normalize_signals() (*in module app.deepbeat_data_visualization*), 14
normalize_signals() (*in module app.mimic3bp_data_visualization*), 17
normalize_target() (*qumphy.data.pulsedb.PulseDBDataset method*), 30
NormType (*class in qumphy.models.xresnet1d*), 73
nplr() (*in module qumphy.models.s42*), 68
num_kernels (*qumphy.models.minirocket.MiniRocketFeatures attribute*), 61

O

objective() (*in module app.train*), 18
on_test_batch_end() (*qumphy.callbacks.base_logging.BaseLoggingCallback method*), 20
on_test_epoch_end() (*qumphy.callbacks.base_logging.BaseLoggingCallback method*), 20
on_test_epoch_end() (*qumphy.callbacks.deepbeat.DeepBeatLogging_KGLoss method*), 21
on_test_epoch_end() (*qumphy.callbacks.deepbeat.DeepBeatLogging_MCD method*), 21
on_test_epoch_end() (*qumphy.callbacks.pulsedb.PulseDBLogging_Ensemble method*), 22
on_test_epoch_end() (*qumphy.callbacks.pulsedb.PulseDBLogging_MCD method*), 22
on_test_epoch_end() (*qumphy.callbacks.pulsedb.PulseDBLogging_Pinballloss method*), 22
on_test_epoch_start() (*qumphy.callbacks.base_logging.BaseLoggingCallback method*), 20
on_train_batch_end() (*qumphy.callbacks.base_logging.BaseLoggingCallback method*), 20
on_train_end() (*qumphy.callbacks.progressbar.EpochProgressBar method*), 21
on_train_epoch_end() (*qumphy.callbacks.base_logging.BaseLoggingCallback method*), 20
on_train_epoch_end() (*qumphy.callbacks.progressbar.EpochProgressBar method*), 21
on_train_epoch_start() (*qumphy.callbacks.base_logging.BaseLoggingCallback method*), 20
on_train_epoch_start() (*qumphy.callbacks.progressbar.EpochProgressBar method*), 21
on_train_start() (*qumphy.callbacks.progressbar.EpochProgressBar method*), 21
on_validation_batch_end() (*qumphy.callbacks.base_logging.BaseLoggingCallback method*), 20
on_validation_epoch_end() (*qumphy.callbacks.base_logging.BaseLoggingCallback method*), 20
on_validation_epoch_start() (*qumphy.callbacks.base_logging.BaseLoggingCallback method*), 20

P

padtype (*qumphy.data.signal_preprocessing.resampling.MatlabResampleConfig attribute*), 26
parse_value() (*in module qumphy.misc.misc*), 36
PatchEmbedding (*class in qumphy.models.utils.embed*), 40
picp() (*in module qumphy.uq_metrics*), 99
PinballLoss (*class in qumphy.models.utils.pinballloss*), 42
plot_signal_grid() (*in module app.deepbeat_data_visualization*), 15
plot_signal_grid() (*in module app.mimic3bp_data_visualization*), 17
PositionalEmbedding (*class in qumphy.models.utils.embed*), 40
power() (*in module qumphy.models.s42*), 69
powerline_noise() (*in module qumphy.data.signal_preprocessing.noise*), 25
PPNet (*class in qumphy.models.ppnet*), 62
precision_score() (*in module qumphy.metrics*), 87
predict() (*qumphy.trainer.Trainer method*), 91
predict_step() (*qumphy.models.apnea.SleepApneaModule method*), 44
predict_step() (*qumphy.models.deepbeat.DeepBeatModule method*), 53
predict_step() (*qumphy.models.pulsedb.PulseDBModule method*), 64
print_metrics() (*qumphy.evaluate.deepensembles.DeepBeatEvaluation method*), 33
print_metrics() (*qumphy.evaluate.deepensembles.PulseDBEvaluation method*), 34
ProbAttention (*class in qumphy.models.utils.selfattention_family*), 43
ProbMask (*class in qumphy.models.utils.masking*), 41
probs_to_pred_sets() (*in module qumphy.misc.output_conversions*), 39
PulseDBDataModule (*class in qumphy.data.pulsedb*), 29
PulseDBDataset (*class in qumphy.data.pulsedb*), 29
PulseDBEnsemble (*class in qumphy.models.pulsedb*), 62
PulseDBEvaluation (*class in qumphy.evaluate.deepensembles*), 33
PulseDBGaussianLoss (*class in qumphy.models.pulsedb*), 63
PulseDBLogging (*class in qumphy.callbacks.pulsedb*), 22
PulseDBLogging_Ensemble (*class in qumphy.callbacks.pulsedb*), 22
PulseDBLogging_MCD (*class in qumphy.callbacks.pulsedb*), 22
PulseDBLogging_Pinballloss (*class in qumphy.callbacks.pulsedb*), 22
PulseDBModule (*class in qumphy.models.pulsedb*), 63
PulseDBModule_MCD (*class in qumphy.models.pulsedb*), 64

Q

quantiles (*qumphy.models.utils.pinballloss.PinballLoss.self attribute*), 42
qumphy
    module, 20
qumphy.callbacks
    module, 20
qumphy.callbacks.base_logging
    module, 20
qumphy.callbacks.deepbeat
    module, 20
qumphy.callbacks.progressbar
    module, 21
qumphy.callbacks.pulsedb
    module, 22
qumphy.callbacks.sleepapnea
    module, 22
qumphy.data
    module, 23
qumphy.data.attractor
    module, 26

qumphy.data.deepbeat
    module, 27
qumphy.data.pulsedb
    module, 29
qumphy.data.signal_preprocessing
    module, 23
qumphy.data.signal_preprocessing.filters
    module, 23
qumphy.data.signal_preprocessing.noise
    module, 24
qumphy.data.signal_preprocessing.resampling
    module, 26
qumphy.data.sleepapnea
    module, 31
qumphy.data.utils
    module, 32
qumphy.evaluate
    module, 32
qumphy.evaluate.deepensembles
    module, 32
qumphy.metrics
    module, 78
qumphy.misc
    module, 35
qumphy.misc.misc
    module, 35
qumphy.misc.output_conversions
    module, 37
qumphy.models
    module, 39
qumphy.models.alexnet
    module, 43
qumphy.models.apnea
    module, 44
qumphy.models.basic_conv1d
    module, 45
qumphy.models.deep_ensemble
    module, 51
qumphy.models.deepbeat
    module, 52
qumphy.models.dumbnet
    module, 55
qumphy.models.inception1d
    module, 55
qumphy.models.inception_mantas
    module, 58
qumphy.models.itransformer
    module, 59
qumphy.models.minirocket
    module, 60
qumphy.models.ppnet
    module, 62
qumphy.models.pulsedb
    module, 62

qumphy.models.s42
    module, 64
qumphy.models.s4_model
    module, 70
qumphy.models.timesnet
    module, 70
qumphy.models.utils
    module, 39
qumphy.models.utils.conv_blocks
    module, 39
qumphy.models.utils.embed
    module, 40
qumphy.models.utils.kgactivation
    module, 40
qumphy.models.utils.kgloss
    module, 41
qumphy.models.utils.masking
    module, 41
qumphy.models.utils.mcdropout
    module, 42
qumphy.models.utils.pinballloss
    module, 42
qumphy.models.utils.selfattention_family
    module, 42
qumphy.models.utils.transformer_encdec
    module, 43
qumphy.models.xresnet1d
    module, 72
qumphy.trainer
    module, 90
qumphy.uq
    module, 92
qumphy.uq_metrics
    module, 93

R

rank_correction() (*in module qumphy.models.s42*), 69
recall_score_threshold() (*in module qumphy.metrics*), 88
reduce() (*qumphy.evaluate.deepensembles.DeepBeatEvaluation method*), 33
reduce() (*qumphy.evaluate.deepensembles.PulseDBEvaluation method*), 34
reduce_nested_dict_list() (*in module qumphy.evaluate.deepensembles*), 34
ReformerLayer (*class in qumphy.models.utils.selfattention_family*), 43
register() (*qumphy.models.s42.SSKernelNPLR method*), 66
resample_like_matlab() (*in module qumphy.data.signal_preprocessing.resampling*), 26
ResBlock (*class in qumphy.models.xresnet1d*), 73
ResBlock_dropout (*class in qumphy.models.xresnet1d*), 73
root_mean_square_error() (*in module qumphy.metrics*), 89
run_optuna() (*in module app.train*), 18

S

S4 (*class in qumphy.models.s42*), 65
S4Model (*class in qumphy.models.s4_model*), 70
schirrmeister() (*in module qumphy.models.basic_conv1d*), 50
seed_everything() (*qumphy.trainer.Trainer method*), 92

select_subset_indices() (*qumphy.data.deepbeat.DeepBeatDataset method*), 28
select_subset_indices() (*qumphy.data.pulsedb.PulseDBDataset method*), 30
sen() (*in module qumphy.models.basic_conv1d*), 50
sensitivity() (*in module qumphy.metrics*), 89
set_coverage() (*in module qumphy.uq_metrics*), 100
set_dataset_stats() (*qumphy.models.pulsedb.PulseDBModule method*), 64
set_function_dictionary() (*qumphy.callbacks.pulsedb.PulseDBLogging method*), 22
set_lr_scheduler() (*qumphy.models.apnea.SleepApneaModule method*), 44
set_lr_scheduler() (*qumphy.models.deepbeat.DeepBeatModule method*), 53
set_lr_scheduler() (*qumphy.models.pulsedb.PulseDBModule method*), 64
set_output_layer() (*qumphy.models.basic_conv1d.Basic_Conv1d method*), 46
set_output_layer() (*qumphy.models.inception1d.Inception1d method*), 56
set_output_layer() (*qumphy.models.xresnet1d.XResNet1d method*), 74
set_save_dir() (*qumphy.trainer.Trainer method*), 92
set_sweep_parameters() (*qumphy.trainer.Trainer method*), 92
set_value_at_nested_key() (*in module qumphy.misc.misc*), 36
setup() (*qumphy.data.attractor.AttractorDataModule method*), 26
setup() (*qumphy.data.deepbeat.DeepBeatDataModule method*), 27
setup() (*qumphy.data.pulsedb.PulseDBDataModule method*), 29
setup() (*qumphy.data.sleepapnea.SleepApneaDataModule method*), 31
setup_step() (*qumphy.models.s42.SSKernelNPLR method*), 67
Shortcut1d (*class in qumphy.models.inception1d*), 57
SleepApneaDataModule (*class in qumphy.data.sleepapnea*), 31
SleepApneaDataset (*class in qumphy.data.sleepapnea*), 31
SleepApneaLogging (*class in qumphy.callbacks.sleepapnea*), 22
SleepApneaModule (*class in qumphy.models.apnea*), 44
smooth_expected_calibration_error() (*in module qumphy.uq_metrics*), 100
specificity() (*in module qumphy.metrics*), 90
split_into_classes() (*in module qumphy.uq_metrics*), 100
split_to_data_key (*qumphy.data.sleepapnea.SleepApneaDataset attribute*), 32
split_to_label_key (*qumphy.data.sleepapnea.SleepApneaDataset attribute*), 32
SqueezeExcite1d (*class in qumphy.models.basic_conv1d*), 47
SSKernelNPLR (*class in qumphy.models.s42*), 66
state_to_tensor (*qumphy.models.s42.S4 property*), 66
step() (*qumphy.models.s42.HippoSSKernel method*), 65
step() (*qumphy.models.s42.S4 method*), 66
step() (*qumphy.models.s42.SSKernelNPLR method*), 67
str2dict() (*in module qumphy.misc.misc*), 36

T

TemporalEmbedding (*class in qumphy.models.utils.embed*), 40
test() (*qumphy.trainer.Trainer method*), 92
test_dataloader() (*qumphy.data.attractor.AttractorDataModule method*), 27
test_dataloader() (*qumphy.data.deepbeat.DeepBeatDataModule method*), 28
test_dataloader() (*qumphy.data.pulsedb.PulseDBDataModule method*), 29
test_dataloader() (*qumphy.data.sleepapnea.SleepApneaDataModule method*), 31
test_step() (*qumphy.models.apnea.SleepApneaModule method*), 44
test_step() (*qumphy.models.deepbeat.DeepBeatEnsemble method*), 52
test_step() (*qumphy.models.deepbeat.DeepBeatModule method*), 54
test_step() (*qumphy.models.deepbeat.DeepBeatModule_MCD method*), 54
test_step() (*qumphy.models.pulsedb.PulseDBEnsemble method*), 62
test_step() (*qumphy.models.pulsedb.PulseDBModule method*), 64
test_step() (*qumphy.models.pulsedb.PulseDBModule_MCD method*), 64
TimeFeatureEmbedding (*class in qumphy.models.utils.embed*), 40

TimesBlock (*class in qumphy.models.timesnet*), 70
TimesNet (*class in qumphy.models.timesnet*), 70
TokenEmbedding (*class in qumphy.models.utils.embed*), 40
train() (*in module app.train*), 19
train_argument_parser() (*in module qumphy.misc.misc*), 37
train_dataloader() (*qumphy.data.attractor.AttractorDataModule method*), 27
train_dataloader() (*qumphy.data.deepbeat.DeepBeatDataModule method*), 28
train_dataloader() (*qumphy.data.pulsedb.PulseDBDataModule method*), 29
train_dataloader() (*qumphy.data.sleepapnea.SleepApneaDataModule method*), 31
Trainer (*class in qumphy.trainer*), 90
TrainGUI (*class in app.gui*), 15
training_step() (*qumphy.models.apnea.SleepApneaModule method*), 44
training_step() (*qumphy.models.deepbeat.DeepBeatModule method*), 54
training_step() (*qumphy.models.pulsedb.PulseDBModule method*), 64
transition() (*in module qumphy.models.s42*), 69
TransposedLinear (*class in qumphy.models.s42*), 67
TriangularCausalMask (*class in qumphy.models.utils.masking*), 41

U

uncertainty_calibration_error() (*in module qumphy.uq_metrics*), 101
update_dictionary() (*in module qumphy.misc.misc*), 37

V

val_dataloader() (*qumphy.data.attractor.AttractorDataModule method*), 27
val_dataloader() (*qumphy.data.deepbeat.DeepBeatDataModule method*), 28
val_dataloader() (*qumphy.data.pulsedb.PulseDBDataModule method*), 29
val_dataloader() (*qumphy.data.sleepapnea.SleepApneaDataModule method*), 31
validation_step() (*qumphy.models.apnea.SleepApneaModule method*), 44
validation_step() (*qumphy.models.deepbeat.DeepBeatModule method*), 54
validation_step() (*qumphy.models.pulsedb.PulseDBModule method*), 64
variation_calibration_error() (*in module qumphy.uq_metrics*), 102
venv_python() (*in module app.gui*), 16

W

weight_init() (*in module qumphy.models.basic_conv1d*), 51
write_target_stats_yaml() (*in module qumphy.data.pulsedb*), 31

X

xbotnet1d101() (*in module qumphy.models.xresnet1d*), 75
xbotnet1d152() (*in module qumphy.models.xresnet1d*), 75
xbotnet1d50() (*in module qumphy.models.xresnet1d*), 75
XResNet1d (*class in qumphy.models.xresnet1d*), 74
XResNet1d101 (*class in qumphy.models.xresnet1d*), 74
xresnet1d101() (*in module qumphy.models.xresnet1d*), 75
xresnet1d152() (*in module qumphy.models.xresnet1d*), 76
xresnet1d18() (*in module qumphy.models.xresnet1d*), 76
xresnet1d18_deep() (*in module qumphy.models.xresnet1d*), 76
xresnet1d18_deeper() (*in module qumphy.models.xresnet1d*), 76
xresnet1d34() (*in module qumphy.models.xresnet1d*), 76
xresnet1d34_deep() (*in module qumphy.models.xresnet1d*), 76
xresnet1d34_deeper() (*in module qumphy.models.xresnet1d*), 77
XResNet1d50 (*class in qumphy.models.xresnet1d*), 74
xresnet1d50() (*in module qumphy.models.xresnet1d*), 77

xresnet1d50_deep() (*in module qumphy.models.xresnet1d*), 77
xresnet1d50_deeper() (*in module qumphy.models.xresnet1d*), 77
xresnet1d50_MCD() (*in module qumphy.models.xresnet1d*), 77

Z

z_variance_error() (*in module qumphy.uq_metrics*), 102