dingo-gw

Stephen Green

Apr 07, 2024

GETTING STARTED

Dingo (Deep Inference for Gravitational-wave Observations) is a Python program for analyzing gravitational wave data using neural posterior estimation. It dramatically speeds up inference of astrophysical source parameters from data measured at gravitational-wave observatories. Dingo aims to enable the routine use of the most advanced theoretical models in analysing data, to make rapid predictions for multi-messenger counterparts, and to do so in the context of sensitive detectors with high event rates.

The basic approach of Dingo is to *train a neural network to represent the Bayesian posterior*, conditioned on data. This enables **amortized inference**: when new data are observed, they can be plugged in and results obtained in a small amount of time. Tasks handled by Dingo include

- *building training datasets*;
- *training* normalizing flows to estimate the posterior density;
- *performing inference* on real or simulated data; and
- verifying and correcting model results using *importance sampling*.

As training a network from scratch can be expensive, we intend to also distribute trained networks that can be used directly for inference. These can be used with *dingo_pipe* to automate analysis of gravitational wave events.

CHAPTER

ONE

INSTALLATION

1.1 Standard

1.1.1 Pip

To install using pip, run the following within a suitable virtual environment:

```
pip install dingo-gw
```

This will install Dingo as well as all of its requirements, which are listed in pyproject.toml.

1.1.2 Conda

Dingo is also available from the conda-forge repository. To install using conda, first activate a conda environment, and then run

```
conda install -c conda-forge dingo-gw
```

1.2 Development

If you would like to make changes to Dingo, or to contribute to its development, you should install Dingo from source. To do so, first clone this repository:

```
git clone git@github.com:dingo-gw/dingo.git
```

Next create a virtual environment for Dingo, e.g.,

```
python3 -m venv dingo-venv
source dingo-venv/bin/activate
```

This creates and activates a venv for Dingo called `dingo-venv`. In this virtual environment, install Dingo:

```
cd dingo
pip install -e ."[dev]"
```

This command installs an editable version of Dingo, meaning that any changes to the Dingo source are reflected immediately in the installation. The inclusion of `dev` installs extra packages needed for development (code formatting, compiling documentation, etc.)

1.2.1 Documentation

To build the documentation, first generate the API documentation using autodoc:

```
cd docs
sphinx-apidoc -o source ../dingo
```

This will create `dingo.*.rst` and `modules.rst` files in `source/`. These correspond to the various modules and are constructed from docstrings.

To finally compile the documentation, run

```
make html
```

This creates a directory `build/` containing HTML documentation. The main index is at `build/html/index.html`.

To use the autodoc feature, which works for pycharm and numpy docstrings, insert in a .rst file, e.g.,

```
.. autofunction:: dingo.core.utils.trainutils.write_history`
```

This will render as

`dingo.core.utils.trainutils.`**`write_history`**(*log_dir*, *epoch*, *train_loss*, *test_loss*, *learning_rates*, *aux=None*, *filename='history.txt'*)

Writes losses and learning rate history to csv file.

Parameters

- **log_dir** (*str*) – directory containing the history file
- **epoch** (*int*) – epoch
- **train_loss** (*float*) – train_loss of epoch
- **test_loss** (*float*) – test_loss of epoch
- **learning_rates** (*list*) – list of learning rates in epoch
- **aux** (*list = []*) – list of auxiliary information to be logged
- **filename** (*str = 'history.txt'*) – name of history file

Cleanup

To remove generated docs, execute

```
make clean
rm source/dingo.* source/modules.rst
```

CHAPTER

TWO

OVERVIEW

Dingo performs gravitational-wave (GW) parameter estimation using ***neural posterior estimation***. The basic idea is to train a neural network (a normalizing flow) to represent the Bayesian posterior distribution $p(\theta|d)$ for GW parameters θ given observed data d. Training can take some time (typically, a week for a production-level model) but once trained, inference is very fast (just a few seconds).

2.1 Basic workflow

The basic workflow for using Dingo is as follows:

1. **Prepare training data.** This consists of pairs of intrinsic parameters and *waveform polarizations*, as well as *noise PSDs*. Training parameters are drawn from the prior distribution, and *waveforms are simulated* using a waveform model.
2. **Train a model.** *Build a neural network* and *simulate data sets* (noisy waveforms in detectors). *Train the model* to infer parameters based on the data.
3. ***Perform inference*** **on new data** using the trained model.

In many cases, a user may have downloaded a pre-trained model. If so, there is no need to carry out the first two steps, and one may instead skip to **step 3**.

2.2 Command-line interface

In most cases, we expect Dingo to be called from the command line. Dingo commands begin with the prefix `dingo_`. There can be a large number of configurations options for many tasks, so in such cases, rather than specify all settings as arguments, Dingo commands take a single YAML or INI file containing all settings. As described in the *quickstart tutorial*, it is best to begin with settings files provided in the examples/ folder, modifying them as necessary.

2.2.1 Summary of commands

Here we provide a list of key user commands along with brief descriptions. The commands for carrying out the main tasks above are

Command	Description
`dingo_generate_dataset`	Generate a training dataset of waveform polarizations.
`dingo_generate_ASD_dataset`	Generate a training dataset of detector noise ASDs.
`dingo_train`	Build and train a neural network.
`dingo_pipe`	Perform inference on data (real or simulated), starting from an INI file.

Building a training dataset and training a model can be very expensive tasks. We therefore expect these to be frequently run on clusters, and for this reason provided HTCondor versions of these commands (note that `dingo_pipe` is already HTCondor-compatible):

Command	Description
`dingo_generate_dataset_dag`	HTCondor version of `dingo_generate_dataset`.
`dingo_train_condor`	HTCondor version of `dingo_train`.

Finally, there are several utility commands that are useful for working with Dingo-produced files:

Command	Description
`dingo_ls`	Inspect a file produced by Dingo and print a summary.
`dingo_append_training_stage`	Modify the training plan of a model checkpoint.
`dingo_pt_to_hdf5`	Convert a trained Dingo model from a PyTorch pickle .pt file to HDF5.

Hint: The `dingo_ls` command is very useful for inspecting Dingo files. It will print all settings that went in to producing the file, as well as some derived quantities.

2.2.2 File types

As noted above, most Dingo commands take a YAML file to specify configuration options (except for `dingo_pipe`, which uses an INI file, as is standard for LVK parameter estimation). When run, these commands generate data, which is usually stored in HDF5 files. One exception is when training a neural network. This saves the network weights using the PyTorch `.pt` format. However, primarily for LVK use, `dingo_pt_to_hdf5` can convert the weights of a trained model to a HDF5 file.

Important: In all cases, Dingo will save the YAML file settings within the final output file. This is needed for downstream tasks and for maintaining reproducibility.

2.3 GNPE

A slightly more complicated workflow occurs when using *GNPE*. GNPE is an algorithm that combines physical symmetries with Gibbs sampling to significantly improve results. When using GNPE, however, it is necessary to train **two networks**—one main (conditional) network that will be repeatedly sampled during Gibbs sampling and one smaller network used to initialize the Gibbs sampler. At inference time, `dingo_pipe` must be pointed to **both** of these networks. See the section on *GNPE usage* for further details.

CHAPTER

THREE

QUICKSTART TUTORIAL

To learn to use Dingo, we recommend starting with the examples provided in the examples/ folder. The YAML files contained in this directory (and subdirectories) contain configuration settings for the various Dingo tasks (constructing training data, training networks, and performing inference). These files should be provided as input to the command-line scripts, which then run Dingo and save output files. These output files contain as metadata the settings in the YAML files, and they may usually be inspected by running `dingo_ls`.

After configuring the settings files, the scripts may be used as follows, assuming the Dingo `venv` is active.

3.1 Generate training data

3.1.1 Waveforms

To generate a waveform dataset for training, execute

```
dingo_generate_dataset --settings_file waveform_dataset_settings.yaml --num_processes N -
↪-out_file waveform_dataset.hdf5
```

where `N` is the number of processes you would like to use to generate the waveforms in parallel. This saves the dataset of waveform polarizations in the file `waveform_dataset.hdf5` (typically compressed using SVD, depending on configuration).

One can use `dingo_generate_dataset_dag` to set up a condor DAG for generating waveforms on a cluster. This is typically useful for slower waveform models.

3.1.2 Noise ASDs

Training also requires a dataset of noise ASDs, which are sampled randomly for each training sample. To generate this dataset based on noise observed during a run, execute

```
dingo_generate_ASD_dataset --data_dir data_dir --settings_file asd_dataset_settings.yaml
```

This will download data from GWOSC and create a `/tmp` directory, in which the estimated PSDs are stored. Subsequently, these are collected together into a final `.hdf5` ASD dataset. If no `settings_file` is passed, the script will attempt to use the default one `data_dir/asd_dataset_settings.yaml`.

3.2 Training

With a waveform dataset and ASD dataset(s), one can train a neural network. Configure the `train_settings.yaml` file to point to these datasets, and run

```
dingo_train --settings_file train_settings.yaml --train_dir train_dir
```

This will configure the network, train it, and store checkpoints, a record of the history, and the final network in the directory `train_dir`. Alternatively, to resume training from a checkpoint file, run

```
dingo_train --checkpoint model.pt --train_dir train_dir
```

If using CUDA on a machine with several GPUs, be sure to first select the desired GPU number using the `CUDA_VISIBLE_DEVICES` environment variable. If using a cluster, Dingo can be trained using `dingo_train_condor`.

Example training files can be found under `examples/training`. `train_settings_toy.yaml` and `train_settings_production.yaml` train a flow to estimate the full posterior of the event conditioned on the time of coalescence in the detectors. The "toy" label is to indicate this should NOT be used for production but rather to get a feel for the Dingo pipeline. The production settings contain tested settings. Note that depending on the waveform model and event, these may need to occasionally be tuned. `train_settings_init_toy.yaml` and `train_settings_init_production.yaml` train flows to estimate the time of coalescence in the individual detectors. These two networks are needed to use *GNPE*. This is the preferred and most tested way of using Dingo.

Alternatively, the `train_settings_no_gnpe_toy.yaml` and `train_settings_no_gnpe_production.yaml` contain settings to train a network without the GNPE step. Note the lack of a `data/gnpe_time_shifts` option. While this is not recommended for production, it is still pedagogically useful and is good for prototyping new ideas or doing a less expensive training.

3.3 Inference

Once a Dingo model is trained, inference for real events can be performed using *dingo_pipe*. There are 3 main inference steps, downloading the data, running Dingo on this data and finally running importance sampling. The basic idea is to create a .ini file which contains the filepaths of the Dingo networks trained above and the segment of data to analyze. An example .ini file can be found under `examples/pipe/GW150914.ini`.

To do inference, cd into the directory with the .ini file and run

```
dingo_pipe GW150914.ini
```

CHAPTER

FOUR

TOY EXAMPLE

The goal of the following tutorial is to take a user from start to finish analyzing GW150914 using dingo.

> **Caution:** This is only a toy example which is useful for testing on a local machine. This is NOT meant be used for production gravitational wave analyses.

There are 4 main steps:

1. Generate the waveform dataset
2. Generate the ASD dataset
3. Train the network
4. Do inference

In this tutorial as well as the *npe model* and *gnpe model* the following file structure will be employed

```
toy_npe_model/

    #  config files
    waveform_dataset_settings.yaml
    asd_dataset_settings.yaml
    train_settings.yaml
    GW150914.ini

    training_data/
        waveform_dataset.hdf5
        asd_dataset/ # Contains the asd_dataset.hdf5 and also temp files for asd␣
↪generation

    training/
        model_050.pt
        model_stage_0.pt
        model_latest.pt
        history.txt
        #  etc...

    outdir_GW150914/
        #  dingo_pipe output
```

The config files which are the only ones which need to be edited are contained in the top level directory. In the next few sections these config files will be explained. To download sample config files, please visit https://github.com/

dingo-gw/dingo/tree/main/examples. In this tutorial the `toy_npe_model` folder will be used.

4.1 Step 1 Generating a waveform dataset

After downloading the files for the tutorial first run

```
cd toy_npe_model/
mkdir training_data
mkdir training
```

to set up the file structure. Then run

```
dingo_generate_dataset --settings waveform_dataset_settings.yaml --out_file training_
↪data/waveform_dataset.hdf5
```

which will create a *dingo.gw.waveform_generator.waveform_generator.WaveformGenerator* object and store it at the location provided with `--out_file`. For convenience, here is the waveform dataset file

```
domain:
type: FrequencyDomain
f_min: 20.0
f_max: 1024.0
delta_f: 0.25  # Expressions like 1.0/8.0 would require eval and are not supported

waveform_generator:
approximant: IMRPhenomD
f_ref: 20.0
# f_start: 15.0  # Optional setting useful for EOB waveforms. Overrides f_min when␣
↪generating waveforms.

# Dataset only samples over intrinsic parameters. Extrinsic parameters are chosen at␣
↪train time.
intrinsic_prior:
mass_1: bilby.core.prior.Constraint(minimum=10.0, maximum=80.0)
mass_2: bilby.core.prior.Constraint(minimum=10.0, maximum=80.0)
chirp_mass: bilby.gw.prior.UniformInComponentsChirpMass(minimum=15.0, maximum=100.0)
mass_ratio: bilby.gw.prior.UniformInComponentsMassRatio(minimum=0.125, maximum=1.0)
phase: default
chi_1: bilby.gw.prior.AlignedSpin(name='chi_1', a_prior=Uniform(minimum=0, maximum=0.9))
chi_2: bilby.gw.prior.AlignedSpin(name='chi_2', a_prior=Uniform(minimum=0, maximum=0.9))
theta_jn: default
# Reference values for fixed (extrinsic) parameters. These are needed to generate a␣
↪waveform.
luminosity_distance: 100.0  # Mpc
geocent_time: 0.0  # s

# Dataset size
num_samples: 10000

compression: None
```

The file `waveform_dataset_settings.yaml` contains four sections: `domain`, `waveform_generator`, `intrinsic_prior`, and `compression`. The domain section defines the settings for storing the waveform.

Note the `type` attribute; this does not refer to the native domain of the waveform model, but rather to the internal *dingo.gw.domains.Domain* class. This allows the use of time domain waveform models, which are transformed into Fourier domain before being passed to the network. Currently, only the *dingo.gw.domains.FrequencyDomain* class is supported for training the network. It is sometimes advisable to generate waveforms with a higher `f_max` and then truncate them at a lower `f_max` for training due to issues with generating short waveforms for some of the waveform models implemented in LALSuite's LALSimulation package (https://lscsoft.docs.ligo.org/lalsuite/lalsimulation/).

The `waveform_generator` section specifies the `approximant` attribute. At present any waveform model, aka `approximant`, that is callable through LALSimulation's `SimInspiralFD` API can be used to generate waveforms for dingo via the *dingo.gw.waveform_generator.waveform_generator.WaveformGenerator* module (see *generating_waveforms*).

The `intrinsic_prior` section is based on Bilby's prior module. Default values can be found in `dingo.gw.prior`. Two priors to note are the `chirp_mass` and `mass_ratio`, whose minimum values are set to 15.0 and 0.125, respectively. Extending these priors towards lower chirp masses or more extreme mass-ratios may lead to poor performance of the embedding network and normalizing flow during training and would require changes to the network setup. Note that the `luminosity_distance` and `geocent_time` are defined as constants to generate the waveform at a fixed reference point.

The compression section can be set to None for testing purposes. For a practical example of how it is used, see the next tutorial.

4.2 Step 2 Generating the Amplitude Spectral Density (ASD) dataset

To generate an ASD dataset run

```
dingo_generate_asd_dataset --settings_file asd_dataset_settings.yaml --data_dir training_
↪data/asd_dataset
```

This command will generate an *dingo.gw.noise.asd_dataset.ASDDataset* object in the form of an .hdf5 file, which will be used later for training. The reason for specifying a folder instead of a file, as in the waveform dataset example, is because some temporary data is downloaded to create Welch estimates of the ASD. This data can be removed later, but it is sometimes useful for understanding how the ASDs were estimated. For convenience here is a copy of the `asd_dataset_settings.yaml` file.

```yaml
dataset_settings:
f_s: 4096
time_psd: 1024
T: 4
window:
    roll_off: 0.4
    type: tukey
time_gap: 0          # specifies the time skipped between to consecutive PSD estimates.␣
↪If set < 0, the time segments overlap
num_psds_max: 1  # if set > 0, only a subset of all available PSDs will be used
detectors:
    - H1
    - L1
observing_run: O1
```

The `asd_dataset_settings.yaml` file includes several attributes. `f_s` is the sampling frequency in Hz, `time_psd` is the length of time used for an ASD estimate, and `T` is the duration of each ASD segment. Thus, the value of `time_psd/T` gives the number of segments analyzed to estimate one ASD. To avoid spectral leakage, a window is applied to each segment. We use the standard window used in LVK analyses, a Tukey window with a roll off of $\alpha = 0.4$. The next attribute, num_psds_max=1, defines the number of ASDs stored in the ASD dataset. For now, we will use only one. See the next *tutorial* for a more advanced setup.

4.3 Step 3 Training the network

To train the network, first the paths to the correct datasets must be specfied

```
dingo_train --settings_file train_settings.yaml --train_dir training
```

While this file contains numerous settings that are discussed in *training*, we will cover the most significant ones here. Again here is the file.

```
data:
  waveform_dataset_path: training_data/waveform_dataset.hdf5  # Contains intrinsic␣
↪waveforms
  train_fraction: 0.95
  window:  # Needed to calculate window factor for simulated data
    type: tukey
    f_s: 4096
    T: 4.0
    roll_off: 0.4
  detectors:
    - H1
    - L1
  extrinsic_prior:  # Sampled at train time
    dec: default
    ra: default
    geocent_time: bilby.core.prior.Uniform(minimum=-0.10, maximum=0.10)
    psi: default
    luminosity_distance: bilby.core.prior.Uniform(minimum=100.0, maximum=1000.0)
  ref_time: 1126259462.391
  inference_parameters:
  - chirp_mass
  - mass_ratio
  - chi_1
  - chi_2
  - theta_jn
  - dec
  - ra
  - geocent_time
  - luminosity_distance
  - psi
  - phase

# Model architecture
model:
  type: nsf+embedding
  # kwargs for neural spline flow
  nsf_kwargs:
    num_flow_steps: 5
    base_transform_kwargs:
      hidden_dim: 64
```

(continues on next page)

(continued from previous page)

```
      num_transform_blocks: 5
      activation: elu
      dropout_probability: 0.0
      batch_norm: True
      num_bins: 8
      base_transform_type: rq-coupling
  # kwargs for embedding net
  embedding_net_kwargs:
    output_dim: 128
    hidden_dims: [1024, 512, 256, 128]
    activation: elu
    dropout: 0.0
    batch_norm: True
    svd:
      num_training_samples: 1000
      num_validation_samples: 100
      size: 50

# The first stage (and only) stage of training.
training:
  stage_0:
    epochs: 20
    asd_dataset_path: training_data/asd_dataset/asds_O1.hdf5  # this should just contain␣
↪a single fiducial ASD per detector for pretraining
    freeze_rb_layer: True
    optimizer:
      type: adam
      lr: 0.0001
    scheduler:
      type: cosine
      T_max: 20
    batch_size: 64

# Local settings for training that have no impact on the final trained network.
local:
  device: cpu  # Change this to 'cuda' for training on a GPU.
  num_workers: 6  # num_workers >0 does not work on Mac, see https://stackoverflow.com/
↪questions/64772335/pytorch-w-parallelnative-cpp206
  runtime_limits:
    max_time_per_run: 36000
    max_epochs_per_run: 30
  checkpoint_epochs: 15
```

For training, several `extrinsic_priors` are set, which project the waveforms generated in step 1 onto the detector network according to the specified priors. This is considerably cheaper than generating waveforms sampled from the full intrinsic plus extrinsic prior in step 1.

Another crucial setting is `inference_parameters`. By default all the parameters described in `dingo.gw.prior` are inferred. If a parameter needs to be marginalized over this parameter can be omitted from `inference_parameters`.

Essential settings for the model architecture (the neural spline flow and the embedding network) are as follows: `nsf_kwargs.num_flow_steps` describes the number of flow transforms from the base distribution to the final distribution, while `embedding_net_kwargs.hidden_dim` defines the dimensions of the neural network's hidden layer, which selects the most important data features. Finally, `embedding_net_kwargs.svd` describes the settings of the

SVD used as a pre-processing step before passing data vectors to the embedding network. For a production network, these values should be much higher than those used in this tutorial.

Next, we turn to the training section. Here we only employ a single stage of training with settings provided under the `stage_0` attribute. This stage uses the training dataset generated in step 1 for 30 epochs. We also specify the `asd_dataset_path` here, which was created in step 2.

Finally, the local settings section affects only parallelization during training and the device used. An important setting here is `num_workers`, which determines how many PyTorch dataloader processes are spawned during training. If training is too slow, a potential cause is a lack of workers to load data into the network. This can be identified if the dataloader times in the `dingo_train` output exceed 100ms. The solution is generally to increase the number of workers.

4.4 Step 4 Doing Inference

The final step is to do inference, for example on GW150914. To do this we will use *dingo_pipe*. For a local run execute:

```
dingo_pipe GW150914.ini
```

This calls `dingo_pipe` on an INI file that specifies the event to run on,

```
################################################################################
##  Job submission arguments
################################################################################

local = True
accounting = dingo
request-cpus-importance-sampling = 2

################################################################################
##  Sampler arguments
################################################################################

model = training/model_latest.pt
device = 'cpu'
num-samples = 5000
batch-size = 5000
recover-log-prob = false
importance-sample = false

################################################################################
## Data generation arguments
################################################################################

trigger-time = GW150914
label = GW150914
outdir = outdir_GW150914
channel-dict = {H1:GWOSC, L1:GWOSC}
psd-length = 128
# sampling-frequency = 2048.0
# importance-sampling-updates = {'duration': 4.0}

################################################################################
```

(continues on next page)

(continued from previous page)

```
## Plotting arguments
################################################################################

plot-corner = true
plot-weights = true
plot-log-probs = true
```

This will generate files which are described in *dingo_pipe*. To see the results, take a look in `outdir_GW150914`. We set the flag `importance-sample = False` in the INI file, which disables importance sampling for this simple example. Generally one would omit this (it defaults to True).

We can load and manipulate the data with the following code. For example, here we create a cornerplot

```
from dingo.gw.result import Result
result = Result(file_name="outdir_GW150914/result/GW150914_data0_1126259462-4_sampling.
↪hdf5")
result.plot_corner()
```

Notice the results don't look very promising, but this is expected as the settings used in this example are not enough to warrant convergence. Dingo should also automatically generate a cornerplot which will be displayed under outdir_GW150914.

NPE MODEL (PRODUCTION)

We will now do a tutorial with higher profile settings. Note these are not the full production settings used for runs since we are not using *GNPE*, but they should lead to decent results. Go to *this* tutorial for the full production network. The steps are the essentially same as *the toy example* but with higher level settings. It is recommended to run this on a cluster or GPU machine.

We can repeat the same first few steps from the previous tutorial with a couple differences. The file structure is mostly the same but now there is an additional `asd_dataset_fiducial` which will be explained below.

```
npe_model/

    #  config files
    waveform_dataset_settings.yaml
    asd_dataset_settings.yaml
    asd_dataset_settings_fiducial.yaml
    train_settings.yaml
    GW150914.ini

    training_data/
        waveform_dataset.hdf5
        asd_dataset_fiducial/ # Contains the asd_dataset.hdf5 and also temp files for␣
↪asd generation
        asd_dataset/ # Contains the asd_dataset.hdf5 and also temp files for asd␣
↪generation

    training/
        model_050.pt
        model_stage_0.pt
        model_latest.pt
        history.txt
        #  etc...

    outdir_GW150914/
        #  dingo_pipe output
```

5.1 Step 1 Generating a Waveform Dataset

Again the first step is to generate the necessary folders

```
cd npe_model
mkdir training_data
mkdir training
```

As before we run `dingo_generate_dataset`:

```
dingo_generate_dataset --settings waveform_dataset_settings.yaml --out_file training_
↪data/waveform_dataset.hdf5
```

The `waveform_dataset_settings.yaml` settings file now includes a new attribute `compression`. This creates a truncated singular value decomposition (SVD) of the waveform polarizations which is stored on disk as a compressed representation of the dataset. The `size` attribute refers to the number of basis vectors included in the expansion of the waveform. This can later be changed during training. When the compression phase is finished, the log will display the mismatch between the decompressed waveform and generated waveform. You can also access these mismatch settings by running `dingo_ls` on a generated `waveform_dataset.hdf5` file. It will show multiple mismatches corresponding to the number of basis vectors used to decompress the waveform. It is up to the user as to what type of mismatch is acceptable, typically a maximum mismatch of $10^{-3} - 10^{-4}$ is recommended.

We could also generate the waveform dataset using a condor DAG on a cluster. To do this run

```
dingo_generate_dataset_dag --settings_file waveform_dataset_settings.yaml --out_file␣
↪training_data/waveform_dataset.hdf5 --env_path $DINGO_VENV_PATH --num_jobs 4 --request_
↪cpus 64 --request_memory 128000 --request_memory_high 256000
```

and then submit the generated DAG

```
condor_submit_dag condor/submit/dingo_generate_dataset_dagman_DATE.submit
```

where `DATE` is specified in the filename of the `.submit` file that was generated.

5.2 Step 2 Generating an ASD dataset

To generate an ASD dataset we can run the same command as in the previous tutorial.

```
dingo_generate_asd_dataset --settings_file asd_dataset_settings_fiducial.yaml --data_dir␣
↪training_data/asd_dataset_fiducial -out_name training_data/asd_dataset_fiducial/asds_
↪O1_fiducial.hdf5
```

However, this time, during training we will need two sets of ASDs. The first one will be fixed during the initial training – this is the fiducial dataset generated above. This dataset will contain only a single ASD. The second ASDDataset will contain many ASDs and is used during the fine tuning stage. The reason to use just one ASD during the first stage is to allow the network to train in an easier inference setting. It should learn how to infer parameters in the presence of that one ASD. However, during inference the ASD will be variable. Thus, in the second stage many ASDs are used so that dingo learns the distribution of ASDs from the observing run. We find this split leads to an improvement in overall performance. To generate this second dataset run

```
dingo_generate_asd_dataset --settings_file asd_dataset_settings.yaml --data_dir training_
↪data/asd_dataset -out_name training_data/asd_dataset/asds_O1.hdf5
```

We can see that in asd_dataset_settings.yaml the num_psds_max attribute is set to 0 indicating that all possible ASDs will be downloaded. If you want to decrease this, make sure that there are enough ASDs in the training set to represent any possible data the dingo network will see. Typically this should be at least 1000, but of course more is better.

5.3 Step 3 Training the network

Now we are ready for training. The command is analogous to the previous tutorial but the settings are increased to production values. To run the training do

```
dingo_train --settings_file train_settings.yaml --train_dir training
```

Tip: If running on a machine with multiple GPUs make sure to specify the GPU by running export CUDA_VISIBILE_DEVICES=GPU_NUM before running dingo_train

The main difference from the toy example in the network architecture is the size of the embedding network which is described in model.embedding_net_kwargs.hidden_dims and the number of neural spline flow transforms described in model.nsf_kwargs.num_flow_steps. These increase the depth of the network and the number/size of the layers in the embedding network.

Notice, we are not inferring the phase parameter here as it is not listed below inference_parameters. However, we do recover the phase in post processing. To see why and how this is done see *synthetic phase*

Also notice there are now two training stages stage_0 and stage_1. In stage_0 a fixed ASD is used and the reduced basis layer is frozen. Then in stage_1 all ASDs are used and the reduced basis layer is unfrozen.

The main difference in the local settings is that the device is set to CUDA. Note if you have multiple GPUs on the machine, you can select which GPU to use by running

Important: It is recommended to have at least 40 GB of GPU memory on the device. If there is not enough memory on the machine, first try halving the batch_size. In this case one should also multiply the learning rate, lr, by $\frac{1}{\sqrt{2}}$. If there is still not enough memory, consider reducing the number of hidden dimensions.

5.4 Step 4 Doing Inference

We can run inference with the same command as before

```
dingo_pipe GW150914.ini
```

There is just one difference from the previous example. It is possible to reweight the posterior to a new prior. Note though, that the new prior must be a subset of the previous prior. Otherwise, the proposal distribution generated by dingo will include regions from the new prior where the network has not been trained which will result in a low effective sample size and lead to poor results. As an example see the prior-dict attribute in GW150914.ini.

CHAPTER

SIX

GNPE MODEL (PRODUCTION)

This tutorial has the highest profile settings and is the one typically used for production use. The main difference from the *NPE* tutorial is that here we are now using *GNPE* (group neural posterior estimation). The data generation is exactly the same as the *previous* tutorial, but we repeat it here, for completeness.

The file structure is similar to the NPE example, except now there are two training sub-directories and two `train_settings.yaml` files.

```
gnpe_model/

    #  config files
    waveform_dataset_settings.yaml
    asd_dataset_settings_fiducial.yaml
    asd_dataset_settings.yaml
    train_settings_main.yaml
    train_settings_init.yaml
    GW150914.ini

    training_data/
        waveform_dataset.hdf5
        asd_dataset.hdf5
        asd_dataset_fiducial.hdf5
        asd_dataset_fiducial/ # Contains the asd_dataset.hdf5 and also temp files for␣
↪asd generation
        asd_dataset/ # Contains the asd_dataset.hdf5 and also temp files for asd␣
↪generation

    training/
        main_train_dir/
            model_050.pt
            model_stage_0.pt
            model_latest.pt
            history.txt
            #  etc...
        init_train_dir/
            model_050.pt
            model_stage_0.pt
            model_latest.pt
            history.txt
            #  etc...

    outdir_GW150914/
```

(continues on next page)

(continued from previous page)

```
        #  dingo_pipe output
```

6.1 Step 1 Generating a Waveform Dataset

First generate the directory structure:

```
cd gnpe_model
mkdir training_data
mkdir training
mkdir training/main_train_dir
mkdir training/init_train_dir
```

Generate the waveform dataset:

```
dingo_generate_dataset --settings waveform_dataset_settings.yaml --out_file training_
↪data/waveform_dataset.hdf5
```

or using condor:

```
dingo_generate_dataset_dag --settings_file
waveform_dataset_settings.yaml --out_file
training_data/waveform_dataset.hdf5 --env_path $DINGO_VENV_PATH --num_jobs 4
--request_cpus 16 --request_memory 1280000 --request_memory_high 256000
```

6.2 Step 2 Generating an ASD dataset

As before we generate a fiducial ASD dataset containing a single ASD:

```
dingo_generate_asd_dataset --settings_file asd_dataset_settings_fiducial.yaml --data_dir
training_data/asd_dataset_fiducial -out_name training_data/asd_dataset_fiducial/asds_O1_
↪fiducial.hdf5
```

and a large ASD dataset:

```
dingo_generate_asd_dataset --settings_file asd_dataset_settings.yaml --data_dir
training_data/asd_dataset -out_name training_data/asd_dataset/asds_O1.hdf5
```

6.3 Step 3 Training the network

Now we are ready for training using GNPE. Here we need to train two networks, one which estimates the time of arrival in the detectors and one which does the full inference task. A natural question is why train two networks. The main idea is if one is able to align (and thus standardize) the times of arrival in the detectors, the inference task will become significantly easier. To do this we first need to train an initialization network which estimates the time of arrival in the detectors:

```
dingo_train --settings_file train_settings_init.yaml --train_dir training/init_network
```

Notice that the inference parameters are only the `H1_time` and `L1_time`. Also notice that the embedding_net is significantly smaller and the number of flow steps, `num_flow_steps` is reduced.

```
dingo_train --settings_file train_settings_main.yaml --train_dir training/main_network
```

Notice the `data.gnpe_time_shifts` section. The `kernel` describes how much to blur the GNPE proxies and is specified in seconds. To read more about this see *GNPE*.

6.4 Step 4 Doing Inference

Performing inference requires a few changes to the previous NPE setup. Most notably, since we are now using GNPE, we have to specify the file path to both the initialization network and the main network. Another difference is the new attribute under sampler arguments `num-gnpe-iterations` which indicates the number of GNPE steps to take. If the initialization network is not fully converged or if the length of the segment being analyzed is very long, it is recommended to increase this number.

```
dingo_pipe GW150914.ini
```

CHAPTER

SEVEN

INFERENCE ON AN INJECTION

A simple example is creating an injection consistent with what the network was trained on, and then running Dingo on it. First one can instantiate the *dingo.gw.injection.Injection* using the metadata from the *dingo.core.models.posterior_model.PosteriorModel* (the trained network). An ASD dataset also needs to be specified, one can take the fiducial asd dataset the network was trained on.

```
from dingo.core.models import PosteriorModel
import dingo.gw.injection as injection
from dingo.gw.ASD_dataset.noise_dataset import ASDDataset

main_pm = PosteriorModel(
    device="cuda",
    model_filename="/path/to/main_network",
    load_training_info=False
)

init_pm = PosteriorModel(
    device='cuda',
    model_filename="/path/to/init_network",
    load_training_info=False
)

injection_generator = injection.Injection.from_posterior_model_metadata(main_pm.metadata)
asd_fname = main_pm.metadata["train_settings"]["training"]["stage_0"]["asd_dataset_path"]
asd_dataset = ASDDataset(file_name=asd_fname)
injection_generator.asd = {k:v[0] for k,v in asd_dataset.asds.items()}

intrinsic_parameters = {
    "chirp_mass": 35,
    "mass_ratio": 0.5,
    "a_1": .3,
    "a_2": .5,
    "tilt_1": 0.,
    "tilt_2": 0.,
    "phi_jl": 0.,
    "phi_12": 0.
}

extrinsic_parameters = {
    'phase': 0.,
    'theta_jn': 2.3,
```

(continues on next page)

(continued from previous page)

```
    'geocent_time': 0.,
    'luminosity_distance': 400.,
    'ra': 0.,
    'dec': 0.,
    'psi': 0.,
}

theta = {**intrinsic_parameters, **extrinsic_parameters}
strain_data = injection_generator.injection(theta)
```

Then one can create a injections and do inference on them.

```
from dingo.gw.inference.gw_samplers import GWSamplerGNPE, GWSampler

init_sampler = GWSampler(model=init_pm)
sampler = GWSamplerGNPE(model=main_pm, init_sampler=init_sampler, num_iterations=30)
sampler.context = strain_data
sampler.run_sampler(num_samples=50_000, batch_size=10_000)
result = sampler.to_result()
result.plot_corner()
```

CHAPTER

EIGHT

INTRODUCTION TO NEURAL POSTERIOR ESTIMATION

In contrast to classical parameter estimation codes like Bilby and LALInference, Dingo uses simulation-based (or likelihood-free) inference. The basic idea is to train a neural network to represent the Bayesian posterior over source parameters given the observed data. Training is based on simulated data rather than likelihood evaluations. Neural posterior estimation (NPE) combines the ideas of simulation-based inference with conditional neural density estimators.

8.1 Normalizing flows

Normalizing flows provide a means to represent complicated probability distributions using neural networks, in a way that enables rapid sampling and density estimation. They represent the distribution in terms of a mapping (or flow) $f : u \to \theta$ on the sample space from a much simpler "base" distribution, which we take to be standard normal (of the same dimension as the parameter space). If f is allowed to depend on observed data d (denoted f_d) then the flow describes a conditional probability distribution $q(\theta|d)$. The PDF is given by the change of variables rule,

$$q(\theta|d) = \mathcal{N}(0,1)^D (f_d^{-1}(\theta)) \left| \det f_d^{-1} \right| , \tag{8.1}$$

where D is the dimensionality of the parameter space.

A normalizing flow must satisfy the following properties:

1. **Invertibility,** so that one can evaluate $f_d^{-1}(\theta)$ for any θ.
2. **Simple Jacobian determinant,** so that one can quickly evaluate $\det f_d^{-1}(\theta)$.

With these properties, one can quickly evaluate the right-hand side of (8.1) to obtain the density. Various types of normalizing flow have been constructed to satisfy these properties, typically as a composition of relatively simple transforms $f^{(j)}$. These relatively simple transforms are then parametrized by the output of a neural network. To sample $\theta \sim q(\theta|d)$, one samples $u \sim \mathcal{N}(0,1)^D$ and applies the flow in the forward direction.

For each flow step, Dingo uses a conditional coupling transform, meaning that half of the components are held fixed, and the other half transform elementwise, conditional on the untransformed components and the data,

$$f_{d,i}^{(j)}(u) = \begin{cases} u_i & \text{if } i \le D/2, \\ f_i^{(j)}(u_i; u_{1:D/2}, d) & \text{if } i > D/2. \end{cases} \tag{8.2}$$

if $i > D/2$.

If the elementwise functions $f_i^{(j)}$ are differentiable, then it follows automatically that we have a normalizing flow. We use a neural spline flow, meaning that the functions $f_i^{(j)}$ are splines, which in turn are parametrized by neural network outputs (taking as input $(u_{1:D/2}, d)$). Between each of these transforms, the parameters are randomly permuted, ensuring that the full flow is sufficiently flexible. Dingo uses the implementation of this entire structure provided by nflows.

8.2 Training

The conditional neural density estimator $q(\theta|d)$ is initialized randomly and must be trained to become a good approximation to the posterior $p(\theta|d)$. To achieve this, one must specify a target loss function to minimize. A reasonable starting point is to minimize the Kullback-Leibler (KL) divergence of p from q,

$$D_{\mathrm{KL}}(p\|q) = \int d\theta\, p(\theta|d) \log \frac{p(\theta|d)}{q(\theta|d)}.$$

This measures a deviation between the two distributions, and is notably not symmetric. (We take the so-called “forward” KL divergence, which is “mass-covering”.) Taking the expectation over data samples $d \sim p(d)$, and dropping the numerator from the $\log$ term (since it is independent of the network parameters), we arrive at the loss function

$$\begin{aligned} L &= \int dd\, p(d) \int d\theta\, p(\theta|d)\, [-\log q(\theta|d)] \\ &= \int d\theta\, p(\theta) \int dd\, p(d|\theta)\, [-\log q(\theta|d)] \end{aligned} \tag{8.3}$$

On the second line we used Bayes’ theorem $p(d)p(\theta|d) = p(\theta)p(d|\theta)$ to re-order the integrations. The loss may finally be approximated on a mini-batch of samples,

$$L \approx -\frac{1}{N}\sum_{i=1}^{N} \log q(\theta^{(i)}|d^{(i)}),$$

where the samples are drawn ancestrally in a two-step process:

1. **Sample from the prior,** $\theta^{(i)} \sim p(\theta)$,
2. **Simulate data,** $d^{(i)} \sim p(d|\theta^{(i)})$,

We then take the gradient of L with respect to network parameters and minimize using the Adam optimizer.

Importantly, the process to generate training samples incorporates the **same information** as a standard (likelihood-based) sampler would use. Namely, the prior is incorporated by sampling parameters from it, and the likelihood is incorporated by simulating data. Bayes’ theorem is incorporated in going from line 1 to line 2 in (8.3). For gravitational waves, the likelihood is taken to be the probability that the residual when subtracting a signal $h(\theta)$ from d is stationary Gaussian noise (with the measured PSD $S_{\mathrm{n}}(f)$ in the detector). Likewise, to simulate data we generate a waveform $h(\theta^{(i)})$ and add a random noise realization $n \sim \mathcal{N}(0, S_{\mathrm{n}}(f))$. Ultimately, however, the SBI approach is more flexible, since in principle one could add non-stationary or non-Gaussian noise, and train the network to reproduce the posterior, despite not having a tractable likelihood. See the section on training data for additional details of training for gravitational wave inference.

Intuitively, one way to understand NPE is simply that we are doing supervised deep learning—inferring parameter labels from examples—but allowing for the flexibility to produce a probabilistic answer. With this flexibility, the network learns to produce the Bayesian posterior.

CHAPTER

NINE

CODE DESIGN

9.1 Reproducibility

Generating reproducible results must be central to any deep learning code. Dingo attempts to achieve this in the following ways:

9.1.1 Settings

There are a large number of configuration options that must be selected when using Dingo. These include

- Waveform and noise dataset settings,
- Training settings, including pre-processing, neural network, and training strategy settings,
- Inference settings, including event time or injection data.

The Dingo approach is to save all of these settings as nested dictionaries together with the outputs of the various tasks. In practice, this means specifying the settings as a `.yaml` file and passing this to a command-line script that runs some code and produces an output file (`.hdf5` or `.pt`). The output file then contains the settings dictionary (possibly augmented by additional derived parameters). All output files can be inspected using the command-line script `dingo_ls`, which prints the stored settings and possibly additional information. The output from `dingo_ls` could (with a small amount of effort) be used to reproduce the exact results **(modulo random seeds, to be implemented)**.

In addition to saving the user-provided settings at each step, Dingo also saves the settings from precursor steps. For example, when training a model on data from a given waveform dataset, the waveform dataset settings are also saved along with the model settings. This can be very useful at a later point, when only the trained model is available, not the training data. Beyond ensuring reproducibility, having these precursor settings available is needed for certain downstream tasks (e.g., combining the intrinsic prior from a waveform dataset with the extrinsic prior specified for training).

9.1.2 Random seeds

To-do

Implement this.

9.1.3 Unique identifiers for models

To-do

Implement this.

9.2 Code re-use

9.2.1 `core` and `gw` packages

Although the only current use case for Dingo is to analyze LVK data, we hope that it can be extended to other GW or astrophysical (or more general scientific) applications. To facilitate this, we follow the Bilby approach of partitioning code into `core` and `gw` components: `gw` contains GW-specific code (relating to waveforms, interferometers, etc.) whereas `core` contains generic network architectures, data structures, samplers, etc., that we expect could be used in other applications. As we find ways to write elements of code in more generic ways, we hope to migrate additional components from `gw` to `core`. We could then envision future packages, e.g., for LISA inference, GW populations, or cosmology.

9.2.2 Data transforms

We follow the PyTorch guidelines of pre-processing data using a sequence of transforms. Dingo includes *transforms* for tasks such as sampling extrinsic parameters, projecting waveform polarizations to detectors, and adding noise. The same transforms are re-used at inference time, where a similar (but always identical) sequence is required. Some transforms also behave differently at inference time, and thus have a flag to specify the mode.

9.2.3 Data structures

Dingo uses several dataset classes, all of which inherit from *`dingo.core.dataset.DingoDataset`*. This provides a common IO (to save/load from HDF5 as well as dictionaries). It also stores the settings dictionary as an attribute.

9.3 Command-line scripts

In general, Dingo is constructed around libraries and classes that are used to carry out various data processing tasks. There are a large number of configuration options, which are often passed as dictionaries, enabling the addition of new settings without breaking old code.

For very high-level tasks, such as generating a training dataset or training a network, we believe it is most straightforward to use a command-line interface. This is because these are end-user tasks that might be called by separate programs, or on a cluster, or because some of these (dataset generation and training) can be quite expensive.

A Dingo command-line script begins with the prefix `dingo_` and is usually a thin wrapper around a function that could be called by other code if desired. It takes as input a `.yaml` file, passes it as a dictionary to the function, obtains a result, and saves it to disk. We hope that this balance between libraries and a command-line interface enables an extensible code going forward.

CHAPTER

TEN

GENERATING WAVEFORMS

Training data for Dingo consist of pairs of parameters θ and corresponding simulated strain data sets d_I, where I runs over the GW interferometers (L1, H1, V1, etc.). Additionally, when conditioning on detector noise properties, data also include noise context (the PSD $S_{\mathrm{n},I}$). Strain data sets are of the form

$$d_I = h_I(\theta) + n_I,$$

where $h_I(\theta)$ is a *signal waveform* (provided by a waveform model) and n_I is a *noise realization* (stationary and Gaussian, consistent with $S_{\mathrm{n},I}$).

10.1 Data domain

At present, Dingo works entirely with frequency domain data. Although NPE is very flexible and could in principle learn to interpret data in any representation, FD data are especially convenient because (1) stationary Gaussian noise is independent in each frequency bin, so noise generation is straightforward, (2) time shifts take a simple form, enabling improved data augmentation, and (3) the noise context is already in FD. Other domains could be useful in the future, however, so the code is written in a way that the domain could be adapted.

The domain is specified by instantiating a `FrequencyDomain`,

```
from dingo.gw.domains import FrequencyDomain
domain = FrequencyDomain(f_min=20.0, f_max=1024.0, delta_f=0.125)
```

```
/home/docs/checkouts/readthedocs.org/user_builds/dingo-gw/envs/latest/lib/python3.10/
↪site-packages/dingo/gw/__init__.py:3: UserWarning: Wswiglal-redir-stdio:

SWIGLAL standard output/error redirection is enabled in IPython.
This may lead to performance penalties. To disable locally, use:

with lal.no_swig_redirect_standard_output_error():
    ...

To disable globally, use:

lal.swig_redirect_standard_output_error(False)

Note however that this will likely lead to error messages from
LAL functions being either misdirected or lost when called from
Jupyter notebooks.

```

(continues on next page)

(continued from previous page)

```
To suppress this warning, use:

import warnings
warnings.filterwarnings("ignore", "Wswiglal-redir-stdio")
import lal

  import lal
```

Derived class properties include, e.g., the frequency grid. Frequency arrays run from 0 to `f_max`, as is standard for GW data analysis software.

```
domain.sample_frequencies
```

```
array([0.000000e+00, 1.250000e-01, 2.500000e-01, ..., 1.023750e+03,
       1.023875e+03, 1.024000e+03], dtype=float32)
```

Note: The window factor w used when FFTing from time domain data is also stored within the domain, in `domain.window_factor`. This enters into the standard deviation of white noise in each frequency bin, `domain.noise_std`. In frequency domain, this is given by $\sqrt{w/4\delta f}$.

Various class methods also act on data, to perform operations such as zeroing below `f_min`, truncating above `f_max`, or applying a time shift:

class dingo.gw.domains.FrequencyDomain(*f_min: float*, *f_max: float*, *delta_f: float*, *window_factor: float | None = None*)

Defines the physical domain on which the data of interest live.

The frequency bins are assumed to be uniform between [0, f_max] with spacing delta_f. Given a finite length of time domain data, the Fourier domain data starts at a frequency f_min and is zero below this frequency. window_kwargs specify windowing used for FFT to obtain FD data from TD data in practice.

static add_phase(*data*, *phase*)

Add a (frequency-dependent) phase to a frequency series. Allows for batching, as well as additional channels (such as detectors). Accounts for the fact that the data could be a complex frequency series or real and imaginary parts.

Convention: the phase phi(f) is defined via exp(- 1j * phi(f)).

Parameters

- **data** (*Union[np.array, torch.Tensor]*) –
- **phase** (*Union[np.array, torch.Tensor]*) –

Return type

New array or tensor of the same shape as data.

property delta_f: float

The frequency spacing of the uniform grid [Hz].

property domain_dict

Enables to rebuild the domain via calling build_domain(domain_dict).

property duration: float

Waveform duration in seconds.

property f_max: float

The maximum frequency [Hz] is typically set to half the sampling rate.

property f_min: float

The minimum frequency [Hz].

property frequency_mask: ndarray

Mask which selects frequency bins greater than or equal to the starting frequency

property frequency_mask_length: int

Number of samples in the subdomain domain[frequency_mask].

get_sample_frequencies_astype(*data*)

Returns a 1D frequency array compatible with the last index of data array.

Decides whether array is numpy or torch tensor (and cuda vs cpu), and whether it contains the leading zeros below f_min.

Parameters
`data` (*Union[np.array, torch.Tensor]*) – Sample data

Return type
frequency array compatible with last index

property noise_std: float

Standard deviation of the whitened noise distribution.

To have noise that comes from a multivariate *unit* normal distribution, you must divide by this factor. In practice, this means dividing the whitened waveforms by this.

TODO: This description makes some assumptions that need to be clarified. Windowing of TD data; tapering window has a slope -> reduces power only for noise, but not for the signal which is in the main part unaffected by the taper

property sampling_rate: float

The sampling rate of the data [Hz].

set_new_range(*f_min: float | None = None*, *f_max: float | None = None*)

Set a new range [f_min, f_max] for the domain. This is only allowed if the new range is contained within the old one.

time_translate_data(*data*, *dt*)

Time translate frequency-domain data by dt. Time translation corresponds (in frequency domain) to multiplication by

$$\exp(-2\pi i\, f\, dt).$$

This method allows for multiple batch dimensions. For torch.Tensor data, allow for either a complex or a (real, imag) representation.

Parameters

- `data` (*array-like (numpy, torch)*) – Shape (B, C, N), where
  - B corresponds to any dimension >= 0,
  - C is either absent (for complex data) or has dimension >= 2 (for data represented as real and imaginary parts), and
  - N is either len(self) or len(self)-self.min_idx (for truncated data).
- `dt` (*torch tensor, or scalar (if data is numpy)*) – Shape (B)

Return type
Array-like of the same form as data.

update(*new_settings: dict*)

Update the domain with new settings. This is only allowed if the new settings are "compatible" with the old ones. E.g., f_min should be larger than the existing f_min.

Parameters
new_settings (*dict*) – Settings dictionary. Must contain a subset of the keys contained in domain_dict.

update_data(*data: ndarray*, *axis: int = -1*, *low_value: float = 0.0*)

Adjusts data to be compatible with the domain:

- Below f_min, it sets the data to low_value (typically 0.0 for a waveform, but for a PSD this might be a large value).
- Above f_max, it truncates the data array.

Parameters

- **data** (*np.ndarray*) – Data array
- **axis** (*int*) – Which data axis to apply the adjustment along.
- **low_value** (*float*) – Below f_min, set the data to this value.

Returns
The new data array.

Return type
np.ndarray

10.2 Waveform generator

Waveforms are generated using the `WaveformGenerator` class (or its subclass `NewInterfaceWaveformGenerator`, for employing the new LIGO waveform interface, needed for some approximants). This depends on a `Domain` as well as a waveform approximant and a reference frequency `f_ref`. In the backend, the `WaveformGenerator` class calls LALSimulation functions (typically `SimInspiralFD`) via the SWIG-Python interface. For time domain waveforms, `SimInspiralFD` takes care of FFTing to frequency domain. The `NewInterfaceWaveformGenerator` class calls the gwsignal module, a Python interface recently implemented in LALSimulation, which is needed for employing some of the latest waveform approximants, as the `SEOBNRv5HM` and `SEOBNRv5PHM`.

```
from dingo.gw.waveform_generator import WaveformGenerator #,␣
↪NewInterfaceWaveformGenerator

wfg = WaveformGenerator(approximant='IMRPhenomXPHM', domain=domain, f_ref=20.0)
# wfg = NewInterfaceWaveformGenerator(approximant='SEOBNRv5PHM', domain=domain, f_ref=20.
↪0)
```

```
Setting spin_conversion_phase = None. Using phase parameter for conversion to cartesian␣
↪spins.
```

To generate a waveform we first need to choose parameters. Here we sample parameters from a `bilby.core.prior.PriorDict`. We use the default Dingo intrinsic prior.

```python
from bilby.core.prior import PriorDict
from dingo.gw.prior import default_intrinsic_dict

prior = PriorDict(default_intrinsic_dict)
prior
```

```
{'mass_1': Constraint(minimum=10.0, maximum=80.0, name=None, latex_label=None,␣
↪unit=None),
 'mass_2': Constraint(minimum=10.0, maximum=80.0, name=None, latex_label=None,␣
↪unit=None),
 'mass_ratio': bilby.gw.prior.UniformInComponentsMassRatio(minimum=0.125, maximum=1.0,␣
↪name='mass_ratio', latex_label='$q$', unit=None, boundary=None, equal_mass=False),
 'chirp_mass': bilby.gw.prior.UniformInComponentsChirpMass(minimum=25.0, maximum=100.0,␣
↪name='chirp_mass', latex_label='$\\mathcal{M}$', unit=None, boundary=None),
 'luminosity_distance': DeltaFunction(peak=1000.0, name=None, latex_label=None,␣
↪unit=None),
 'theta_jn': Sine(minimum=0.0, maximum=3.141592653589793, name=None, latex_label=None,␣
↪unit=None, boundary=None),
 'phase': Uniform(minimum=0.0, maximum=6.283185307179586, name=None, latex_label=None,␣
↪unit=None, boundary='periodic'),
 'a_1': Uniform(minimum=0.0, maximum=0.99, name=None, latex_label=None, unit=None,␣
↪boundary=None),
 'a_2': Uniform(minimum=0.0, maximum=0.99, name=None, latex_label=None, unit=None,␣
↪boundary=None),
 'tilt_1': Sine(minimum=0.0, maximum=3.141592653589793, name=None, latex_label=None,␣
↪unit=None, boundary=None),
 'tilt_2': Sine(minimum=0.0, maximum=3.141592653589793, name=None, latex_label=None,␣
↪unit=None, boundary=None),
 'phi_12': Uniform(minimum=0.0, maximum=6.283185307179586, name=None, latex_label=None,␣
↪unit=None, boundary='periodic'),
 'phi_jl': Uniform(minimum=0.0, maximum=6.283185307179586, name=None, latex_label=None,␣
↪unit=None, boundary='periodic'),
 'geocent_time': DeltaFunction(peak=0.0, name=None, latex_label=None, unit=None)}
```

```python
p = prior.sample()
p
```

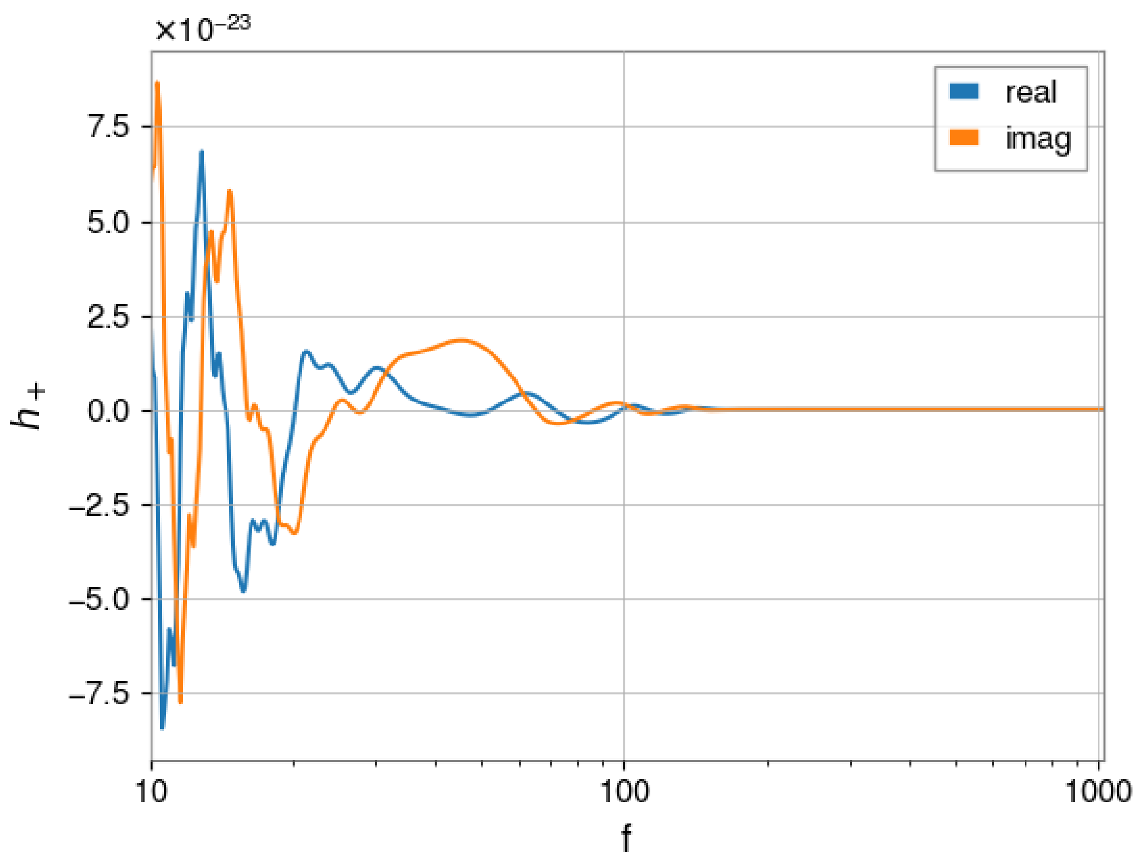
```
{'mass_ratio': 0.27516887747784635,
 'chirp_mass': 75.96284973482983,
 'luminosity_distance': 1000.0,
 'theta_jn': 1.4424160368867687,
 'phase': 3.5597919874340875,
 'a_1': 0.6803566132772145,
 'a_2': 0.1772403333232536,
 'tilt_1': 2.4084579981751792,
 'tilt_2': 1.5913639153680237,
 'phi_12': 0.17224461804836214,
 'phi_jl': 5.8646174013435814,
 'geocent_time': 0.0}
```

Finally, we generate the waveform. This is returned as a dictionary, with entries for each polarization. This way of representing a sample is used throughout Dingo, and will be very convenient when applying transforms (to apply extrinsic parameters, add noise, etc.).

```
h = wfg.generate_hplus_hcross(p)
h
```

```
{'h_plus': array([0.+0.j, 0.+0.j, 0.+0.j, ..., 0.+0.j, 0.+0.j, 0.+0.j]),
 'h_cross': array([0.+0.j, 0.+0.j, 0.+0.j, ..., 0.+0.j, 0.+0.j, 0.+0.j])}
```

```
import matplotlib.pyplot as plt
plt.plot(domain.sample_frequencies, h['h_plus'].real, label='real')
plt.plot(domain.sample_frequencies, h['h_plus'].imag, label='imag')
plt.xlim((10,1024))
plt.xscale('log')
plt.legend()
plt.xlabel('f')
plt.ylabel(r'$h_+$')
plt.show()
```



Note that the waveform is nonzero slightly below `f_min`. This simply arises from the model implementation in `LALSimulation`. When training networks, input data will be truncated below `f_min`.

The complete specification of the `WaveformGenerator` class is given as

class `dingo.gw.waveform_generator.`**`WaveformGenerator`**(*approximant: str*, *domain:* Domain, *f_ref: float*, *f_start: float | None = None*, *mode_list: List[Tuple] | None = None*, *transform=None*, *spin_conversion_phase=None*, ***kwargs*)

Generate polarizations using LALSimulation routines in the specified domain for a single GW coalescence given a set of waveform parameters.

Parameters

- **approximant** (*str*) – Waveform "approximant" string understood by lalsimulation This is defines which waveform model is used.
- **domain** (Domain) – Domain object that specifies on which physical domain the waveform polarizations will be generated, e.g. Fourier domain, time domain.
- **f_ref** (*float*) – Reference frequency for the waveforms
- **f_start** (*float*) – Starting frequency for waveform generation. This is optional, and if not included, the starting frequency will be set to f_min. This exists so that EOB waveforms can be generated starting from a lower frequency than f_min.
- **mode_list** (*List[Tuple]*) – A list of waveform (ell, m) modes to include when generating the polarizations.
- **spin_conversion_phase** (*float = None*) – Value for phiRef when computing cartesian spins from bilby spins via bilby_to_lalsimulation_spins. The common convention is to use the value of the phase parameter here, which is also used in the spherical harmonics when combining the different modes. If spin_conversion_phase = None, this default behavior is adapted. For dingo, this convention for the phase parameter makes it impossible to treat the phase as an extrinsic parameter, since we can only account for the change of phase in the spherical harmonics when changing the phase (in order to also change the cartesian spins – specifically, to rotate the spins by phase in the sx-sy plane – one would need to recompute the modes, which is expensive). By setting spin_conversion_phase != None, we impose the convention to always use phase = spin_conversion_phase when computing the cartesian spins.

generate_FD_modes_LO(*parameters*)

Generate FD modes in the L0 frame.

Parameters

parameters (*dict*) – Dictionary of parameters for the waveform. For details see see self.generate_hplus_hcross.

Returns

- **hlm_fd** (*dict*) – Dictionary with (l,m) as keys and the corresponding FD modes in lal format as values.
- **iota** (*float*)

generate_FD_waveform(*parameters_lal: Tuple*) → Dict[str, ndarray]

Generate Fourier domain GW polarizations (h_plus, h_cross).

Parameters

parameters_lal – A tuple of parameters for the lalsimulation waveform generator

Returns

A dictionary of generated waveform polarizations

Return type

pol_dict

generate_TD_modes_L0(*parameters*)

Generate TD modes in the L0 frame.

Parameters
`parameters` (*dict*) – Dictionary of parameters for the waveform. For details see see self.generate_hplus_hcross.

Returns

- **hlm_td** (*dict*) – Dictionary with (l,m) as keys and the corresponding TD modes in lal format as values.
- **iota** (*float*)

`generate_TD_waveform`(*parameters_lal: Tuple*) → Dict[str, ndarray]

Generate time domain GW polarizations (h_plus, h_cross)

Parameters
`parameters_lal` – A tuple of parameters for the lalsimulation waveform generator

Returns
A dictionary of generated waveform polarizations

Return type
pol_dict

`generate_hplus_hcross`(*parameters: Dict[str, float]*, *catch_waveform_errors=True*) → Dict[str, ndarray]

Generate GW polarizations (h_plus, h_cross).

If the generation of the lalsimulation waveform fails with an "Input domain error", we return NaN polarizations.

Use the domain, approximant, and mode_list specified in the constructor along with the waveform parameters to generate the waveform polarizations.

Parameters

- `parameters` (*Dict[str, float]*) – A dictionary of parameter names and scalar values. The parameter dictionary must include the following keys. For masses, spins, and distance there are multiple options.

  Mass: (mass_1, mass_2) or a pair of quantities from
  ((chirp_mass, total_mass), (mass_ratio, symmetric_mass_ratio))

  Spin:
  (a_1, a_2, tilt_1, tilt_2, phi_12, phi_jl) if precessing binary or (chi_1, chi_2) if the binary has aligned spins

  Reference frequency: f_ref at which spin vectors are defined Extrinsic:

  Distance: one of (luminosity_distance, redshift, comoving_distance) Inclination: theta_jn Reference phase: phase Geocentric time: geocent_time (GPS time)

  The following parameters are not required:
  Sky location: ra, dec, Polarization angle: psi

  Units:
  Masses should be given in units of solar masses. Distance should be given in megaparsecs (Mpc). Frequencies should be given in Hz and time in seconds. Spins should be dimensionless. Angles should be in radians.

- `catch_waveform_errors` (*bool*) – Whether to catch lalsimulation errors

Returns
A dictionary of generated waveform polarizations

Return type
wf_dict

generate_hplus_hcross_m(*parameters: Dict[str, float]*) → Dict[tuple, Dict[str, ndarray]]

Generate GW polarizations (h_plus, h_cross), separated into contributions from the different modes. This method is identical to self.generate_hplus_hcross, except that it generates the individual contributions of the modes to the polarizations and sorts these according to their transformation behavior (see below), instead of returning the overall sum.

This is useful in order to treat the phase as an extrinsic parameter. Instead of {"h_plus": hp, "h_cross": hc}, this method returns a dict in the form of {m: {"h_plus": hp_m, "h_cross": hc_m} for m in [-l_max,…,0,…,l_max]}. Each key m contains the contribution to the polarization that transforms according to exp(-1j * m * phase) under phase transformations (due to the spherical harmonics).

Note:

- pol_m[m] contains contributions of the m modes *and* and the -m modes. This is because the frequency domain (FD) modes have a positive frequency part which transforms as exp(-1j * m * phase), while the negative frequency part transforms as exp(+1j * m * phase). Typically, one of these dominates [e.g., the (2,2) mode is dominated by the negative frequency part and the (-2,2) mode is dominated by the positive frequency part] such that the sum of (l,|m|) and (l,-**|m|**) modes transforms approximately as exp(1j * **|m|** * phase), which is e.g. used for phase marginalization in bilby/lalinference. However, this is not exact. In this method we account for this effect, such that each contribution pol_m[m] transforms *exactly* as exp(-1j * m * phase).
- Phase shifts contribute in two ways: Firstly via the spherical harmonics, which we account for with the exp(-1j * m * phase) transformation. Secondly, the phase determines how the PE spins transform to cartesian spins, by rotating (sx,sy) by phase. This is *not* accounted for in this function. Instead, the phase for computing the cartesian spins is fixed to self.spin_conversion_phase (if not None). This effectively changes the PE parameters {phi_jl, phi_12} to parameters {phi_jl_prime, phi_12_prime}. For parameter estimation, a postprocessing operation can be applied to account for this, {phi_jl_prime, phi_12_prime} -> {phi_jl, phi_12}. See also documentation of __init__ method for more information on self.spin_conversion_phase.

Differences to self.generate_hplus_hcross: - We don't catch errors yet TODO - We don't apply transforms yet TODO

Parameters
parameters (*dict*) – Dictionary of parameters for the waveform. For details see see self.generate_hplus_hcross.

Returns
pol_m – Dictionary with contributions to h_plus and h_cross, sorted by their transformation behaviour under phase shifts: {m: {"h_plus": hp_m, "h_cross": hc_m} for m in [-l_max,…,0,…,l_max]} Each contribution h_m transforms as exp(-1j * m * phase) under phase shifts (for fixed self.spin_conversion_phase, see above).

Return type
dict

setup_mode_array(*mode_list: List[Tuple]*) → Dict

Define a mode array to select waveform modes to include in the polarizations from a list of modes.

Parameters
mode_list (*a list of (ell, m) modes*) –

Returns
A lal parameter dictionary

Return type
lal_params

10.2.1 Waveform modes

Add later.

CHAPTER

ELEVEN

BUILDING A WAVEFORM DATASET

For training neural networks, the more training samples the better. With too little training data, one runs the risk of overfitting. Waveforms, however, can be expensive to generate and take up significant storage. Dingo adopts several strategies to mitigate these problems:

- Dingo partitions parameters into two types—intrinsic and extrinsic—and builds a training set based only on the intrinsic parameters. This consists of waveform polarizations h_+ and $h_\times$. Extrinsic parameters are selected during training, and applied to generate the detector waveforms h_I. This augments the training set to provide unlimited samples from the extrinsic parameters.
- Saved waveforms are compressed using a singular value decomposition. Although this is lossy, waveform mismatches can monitored to ensure that they fall below the intrinsic error in the waveform model.

11.1 The `WaveformDataset` class

The `WaveformDataset` is a storage container for waveform polarizations and parameters, which can used to serve samples to a neural network during training:

class `dingo.gw.dataset.`**`WaveformDataset`**(*file_name=None*, *dictionary=None*, *transform=None*, *precision=None*, *domain_update=None*, *svd_size_update=None*)

Bases: *`DingoDataset`*, `Dataset`

This class stores a dataset of waveforms (polarizations) and corresponding parameters.

It can load the dataset either from an HDF5 file or suitable dictionary.

Once a waveform data set is in memory, the waveform data are consumed through a __getitem__() call, optionally applying a chain of transformations, which are classes that implement a __call__() method.

For constructing, provide either file_name, or dictionary containing data and settings entries, or neither.

Parameters

- **file_name** (*str*) – HDF5 file containing a dataset
- **dictionary** (*dict*) – Contains settings and data entries. The dictionary keys should be 'settings', 'parameters', and 'polarizations'.
- **transform** (*Transform*) – Transform to be applied to dataset samples when accessed through __getitem__
- **precision** (*str ('single', 'double')*) – If provided, changes precision of loaded dataset.
- **domain_update** (*dict*) – If provided, update domain from existing domain using new settings.

- **svd_size_update** (*int*) – If provided, reduces the SVD size when decompressing (for speed).

initialize_decompression(*svd_size_update: int | None = None*)

Sets up decompression transforms. These are applied to the raw dataset before self.transform. E.g., SVD decompression.

Parameters

svd_size_update (*int*) – If provided, reduces the SVD size when decompressing (for speed).

load_supplemental(*domain_update=None*, *svd_size_update=None*)

Method called immediately after loading a dataset.

Creates (and possibly updates) domain, updates dtypes, and initializes any decompression transform. Also zeros data below f_min, and truncates above f_max.

Parameters

- **domain_update** (*dict*) – If provided, update domain from existing domain using new settings.
- **svd_size_update** (*int*) – If provided, reduces the SVD size when decompressing (for speed).

update_domain(*domain_update: dict | None = None*)

Update the domain based on new configuration.

The waveform dataset provides waveform polarizations in a particular domain. In Frequency domain, this is [0, domain._f_max]. Furthermore, data is set to 0 below domain._f_min. In practice one may want to train a network based on slightly different domain settings, which corresponds to truncating the likelihood integral.

This method provides functionality for that. It truncates and/or zeroes the dataset to the range specified by the domain, by calling domain.update_data.

Parameters

domain_update (*dict*) – Settings dictionary. Must contain a subset of the keys contained in domain_dict.

`WaveformDataset` subclasses `dingo.core.dataset.DingoDataset` and `torch.utils.data.Dataset`. The former provides generic functionality for saving and loading datasets as HDF5 files and dictionaries, and is used in several components of Dingo. The latter allows the `WaveformDataset` to be used with a PyTorch `DataLoader`. In general, we follow the PyTorch design framework for training, including Datasets, DataLoaders, and Transforms.

11.2 Generating a simple dataset

As described above, the `WaveformDataset` class is just a container, and does not generate the contents itself. Dataset generation is instead carried out using functions in the `dingo.gw.dataset.generate_dataset` module. Although in practice, datasets are likely to be generated from a settings file using the command line interface, here we describe how to generate one interactively.

A dataset is based on an intrinsic prior and a waveform generator, so we build these as described *here*.

```
from dingo.gw.waveform_generator import WaveformGenerator
from bilby.core.prior import PriorDict
from dingo.gw.prior import default_intrinsic_dict
```

(continues on next page)

(continued from previous page)

```
from dingo.gw.domains import FrequencyDomain

domain = FrequencyDomain(f_min=20.0, f_max=1024.0, delta_f=0.125)
wfg = WaveformGenerator(approximant='IMRPhenomXPHM', domain=domain, f_ref=20.0)
prior = PriorDict(default_intrinsic_dict)
```

```
/home/docs/checkouts/readthedocs.org/user_builds/dingo-gw/envs/latest/lib/python3.10/
↪site-packages/dingo/gw/__init__.py:3: UserWarning: Wswiglal-redir-stdio:

SWIGLAL standard output/error redirection is enabled in IPython.
This may lead to performance penalties. To disable locally, use:

with lal.no_swig_redirect_standard_output_error():
    ...

To disable globally, use:

lal.swig_redirect_standard_output_error(False)

Note however that this will likely lead to error messages from
LAL functions being either misdirected or lost when called from
Jupyter notebooks.

To suppress this warning, use:

import warnings
warnings.filterwarnings("ignore", "Wswiglal-redir-stdio")
import lal

  import lal
```

```
Setting spin_conversion_phase = None. Using phase parameter for conversion to cartesian␣
↪spins.
```

We can use the following function to generate sets of parameters and associated waveforms:

```
from dingo.gw.dataset.generate_dataset import generate_parameters_and_polarizations

parameters, polarizations = generate_parameters_and_polarizations(wfg,
                                                                  prior,
                                                                  num_samples=100,
                                                                  num_processes=1)
```

```
Generating dataset of size 100
```

```
parameters
```

```
    mass_ratio  chirp_mass  luminosity_distance  theta_jn     phase       a_1  \
0     0.302823   99.704171               1000.0  1.045181  1.956625  0.624675
1     0.208840   62.787584               1000.0  1.147821  1.231156  0.523501
2     0.167807   75.470319               1000.0  2.248237  1.657892  0.084256
```

(continues on next page)

(continued from previous page)

```
3     0.885713   71.122514              1000.0  2.500974  3.249277  0.151447
4     0.602687   78.836732              1000.0  2.421702  3.511744  0.920802
..         ...         ...                 ...       ...       ...       ...
95    0.710777   87.396051              1000.0  2.460913  2.277778  0.310158
96    0.540317   50.685929              1000.0  1.736872  2.947742  0.105328
97    0.295335   56.577308              1000.0  2.340868  1.890975  0.720159
98    0.390020   94.408416              1000.0  1.719763  0.986305  0.968500
99    0.286339   69.640219              1000.0  2.093173  3.837138  0.229537

         a_2    tilt_1    tilt_2    phi_12    phi_jl  geocent_time
0   0.864143  2.053759  0.897406  4.995902  1.026707           0.0
1   0.457318  1.689699  2.109647  1.568618  0.603215           0.0
2   0.106429  1.648166  2.215986  0.030520  5.517168           0.0
3   0.338559  0.801100  1.155492  3.671084  3.230101           0.0
4   0.570441  2.646145  0.735851  3.358571  0.196351           0.0
..       ...       ...       ...       ...       ...           ...
95  0.221775  2.112918  0.499147  1.131839  3.146899           0.0
96  0.923134  1.103193  1.681197  2.727364  1.440938           0.0
97  0.634283  2.004960  1.563219  3.219694  2.781885           0.0
98  0.747774  1.329375  1.642715  3.340588  2.788269           0.0
99  0.666358  1.686112  0.907684  3.178193  5.533587           0.0

[100 rows x 12 columns]
```

```
polarizations
```

```
{'h_plus': array([[0.+0.j, 0.+0.j, 0.+0.j, ..., 0.+0.j, 0.+0.j, 0.+0.j],
        [0.+0.j, 0.+0.j, 0.+0.j, ..., 0.+0.j, 0.+0.j, 0.+0.j],
        [0.+0.j, 0.+0.j, 0.+0.j, ..., 0.+0.j, 0.+0.j, 0.+0.j],
        ...,
        [0.+0.j, 0.+0.j, 0.+0.j, ..., 0.+0.j, 0.+0.j, 0.+0.j],
        [0.+0.j, 0.+0.j, 0.+0.j, ..., 0.+0.j, 0.+0.j, 0.+0.j],
        [0.+0.j, 0.+0.j, 0.+0.j, ..., 0.+0.j, 0.+0.j, 0.+0.j]]),
 'h_cross': array([[0.+0.j, 0.+0.j, 0.+0.j, ..., 0.+0.j, 0.+0.j, 0.+0.j],
        [0.+0.j, 0.+0.j, 0.+0.j, ..., 0.+0.j, 0.+0.j, 0.+0.j],
        [0.+0.j, 0.+0.j, 0.+0.j, ..., 0.+0.j, 0.+0.j, 0.+0.j],
        ...,
        [0.+0.j, 0.+0.j, 0.+0.j, ..., 0.+0.j, 0.+0.j, 0.+0.j],
        [0.+0.j, 0.+0.j, 0.+0.j, ..., 0.+0.j, 0.+0.j, 0.+0.j],
        [0.+0.j, 0.+0.j, 0.+0.j, ..., 0.+0.j, 0.+0.j, 0.+0.j]])}
```

We can then put these in a `WaveformDataset`,

```
from dingo.gw.dataset import WaveformDataset

dataset_dict = {'parameters': parameters, 'polarizations':polarizations}
wfd = WaveformDataset(dictionary=dataset_dict)
```

Samples can then be easily indexed,

```
wfd[0]
```

```
{'parameters': {'mass_ratio': 0.3028225394903101,
  'chirp_mass': 99.70417093459808,
  'luminosity_distance': 1000.0,
  'theta_jn': 1.0451812800940419,
  'phase': 1.956625056491646,
  'a_1': 0.6246749028641134,
  'a_2': 0.8641430543194487,
  'tilt_1': 2.053758607319084,
  'tilt_2': 0.8974055366142486,
  'phi_12': 4.995902277732458,
  'phi_jl': 1.0267065217222517,
  'geocent_time': 0.0},
 'waveform': {'h_plus': array([0.+0.j, 0.+0.j, 0.+0.j, ..., 0.+0.j, 0.+0.j, 0.+0.j]),
  'h_cross': array([0.+0.j, 0.+0.j, 0.+0.j, ..., 0.+0.j, 0.+0.j, 0.+0.j])}}
```

Note: The sample is represented as a *nested dictionary*. This is a standard format for Dingo.

11.3 Automated dataset construction

The simple dataset constructed above is useful for illustrative purposes, but it lacks the several important features:

- Waveforms are not compressed. A dataset with many samples would therefore take up enormous storage space.
- Not reproducible. The dataset contains no metadata describing its construction (e.g., waveform approximant, domain, prior, . . .).

The `generate_dataset` function automates all of these advanced features:

`dingo.gw.dataset.generate_dataset.`**`generate_dataset`**(*settings: Dict*, *num_processes: int*) → *WaveformDataset*

Generate a waveform dataset.

Parameters

- **`settings`** (*`dict`*) – Dictionary of settings to configure the dataset
- **`num_processes`** (*`int`*) –

Return type

A WaveformDataset based on the settings.

This function is in turn wrapped by the command-line functions `dingo_generate_dataset` and `dingo_generate_dataset_dag`. These take a `.yaml` file with the same contents as the settings dictionary.

11.3.1 Configuration

A typical settings dictionary / `.yaml` config file takes the following form, described in detail below:

```yaml
domain:
  type: FrequencyDomain
  f_min: 20.0
  f_max: 1024.0
  delta_f: 0.125

waveform_generator:
  approximant: IMRPhenomXPHM
  f_ref: 20.0
  # f_start: 15.0  # Optional setting useful for EOB waveforms. Overrides f_min when␣
↪generating waveforms.
  # new_interface: true # Optional setting for employing new waveform interface. This is␣
↪needed for SEOBNRv5 approximants, and optional for standard LAL approximants.
  spin_conversion_phase: 0.0

# Dataset only samples over intrinsic parameters. Extrinsic parameters are chosen at␣
↪train time.
intrinsic_prior:
  mass_1: bilby.core.prior.Constraint(minimum=10.0, maximum=80.0)
  mass_2: bilby.core.prior.Constraint(minimum=10.0, maximum=80.0)
  chirp_mass: bilby.gw.prior.UniformInComponentsChirpMass(minimum=25.0, maximum=100.0)
  mass_ratio: bilby.gw.prior.UniformInComponentsMassRatio(minimum=0.125, maximum=1.0)
  phase: default
  a_1: bilby.core.prior.Uniform(minimum=0.0, maximum=0.99)
  a_2: bilby.core.prior.Uniform(minimum=0.0, maximum=0.99)
  tilt_1: default
  tilt_2: default
  phi_12: default
  phi_jl: default
  theta_jn: default
  # Reference values for fixed (extrinsic) parameters. These are needed to generate a␣
↪waveform.
  luminosity_distance: 100.0  # Mpc
  geocent_time: 0.0  # s

# Dataset size
num_samples: 5000000

# Save a compressed representation of the dataset
compression:
  svd:
    # Truncate the SVD basis at this size. No truncation if zero.
    size: 200
    num_training_samples: 50000
    num_validation_samples: 10000
  whitening: aLIGO_ZERO_DET_high_P_asd.txt
```

domain
Specifies the data domain. Currenly only `FrequencyDomain` is implemented.

waveform_generator

Choose the approximant and reference frequency. For EOB models that require time integration, it is usually necessary to specify a lower starting frequency. In this case, `f_ref` is ignored.

spin_conversion_phase (optional)
Value for `phiRef` when converting PE spins to Cartesian spins via `bilby_to_lalsimulation_spins`. When set to `None` (default), this uses the `phase` parameter. When set to 0.0, `phase` only refers to the azimuthal observation angle, allowing for it to be treated as an extrinsic parameter.

Important: It is necessary to set this to 0.0 if planning to train a `phase`-marginalized network, and then reconstruct the `phase` synthetically.

intrinsic_prior
Specify the prior over intrinsic parameters. Intrinsic parameters here refer to those parameters that are needed to generate waveform polarizations. Extrinsic parameters here refer to those parameters that can be sampled and applied rapidly during training. As shown in the example, it is also possible to specify `default` priors, which is convenient for certain parameters. These are listed in `dingo.gw.prior.default_intrinsic_dict`.

Intrinsic parameters obviously include masses and spins, but also inclination, reference phase, luminosity distance, and time of coalescense at geocenter. Although inclination and phase are often considered extrinsic parameters, they are needed to generate waveform polarizations and cannot be easily transformed.

Luminosity distance and time of coalescense are considered as *both* intrinsic and extrinsic. Indeed they are needed to generate polarizations, but they can also be easily transformed during training to augment the dataset. We therefore fix them to fiducial values for generating polarizations.

num_samples
The number of samples to include in the dataset. For a production model, we typically use 5×10^6 samples.

compression (optional)
How to compress the dataset.

svd (optional)
Construct an SVD basis based on a specified number of additional samples. Save the main dataset in terms of its SVD basis coefficients. The number of elements in the basis is specified by the `size` setting. The performance of the basis is also evaluated in terms of the mismatch against a number of validation samples. All of the validation information, as well as the basis itself, is saved along with the waveform dataset.

whitening (optional)
Whether to save whitened waveforms, and in particular, whether to construct the basis based on whitened waveforms. The basis will be more efficient if whitening is used to adapt it to the detector noise characteristics. To use whitening, simply specify the desired ASD do use, from the Bilby list of ASDs. Note that the whitening is used only for the internal storage of the dataset. When accessing samples from the dataset, they will be unwhitened.

Dataset compression is implemented internally by setting the `WaveformGenerator.transform` operator, so that elements are compressed immediately after generation (avoiding the need to store many uncompressed waveforms in memory). Likewise, decompression is implemented by setting the `WaveformDataset.decompression_transform` operator to apply the inverse transformation. This will act on samples to decompress them when accessed through `WaveformDataset.__getitem__()`.

Important: The automated dataset constructors store the configuration settings in `WaveformDataset.settings`. This is so that the settings can be accessed by more downstream tasks, and for reference.

11.3.2 Command-line interface

In most cases the command-line interface will be used to generate a dataset. Given a settings file, one can call

```
dingo_generate_dataset --settings_file settings.yaml
                       --num_processes N
                       --out_file waveform_dataset.hdf5
```

This will generate a dataset following the configuration in `settings.yaml` and save it as `waveform_dataset.hdf5`, using `N` processes.

To inspect the dataset (or any other Dingo-generated file) use

```
dingo_ls waveform_dataset.hdf5
```

This will print the configuration settings, as well as a summary of the SVD compression performance (if available).

For larger datasets, or those based on slower waveform models, Dingo includes a script that builds a condor DAG, `dingo_generate_dataset_dag`. This splits the generation of waveforms across several nodes, and then reconstitutes the final dataset.

CHAPTER TWELVE

DATA PRE-PROCESSING

A sample from a `WaveformDataset` consists of labeled waveform polarizations $(\theta_{\text{intrinsic}}, (h_+, h_\times))$, represented as a nested dictionary. This must be transformed into noisy detector data d_I (with additional noise context data) in a form suitable for input to a neural network. Dingo accomplishes this by applying a sequence of **transforms** to the sample.

A transform is simply a class with a `__call__()` method, which takes a sample as input and returns a transformed sample. A sequence of transforms can be then be composed to build a more complex transform in a modular way. Dingo's training transform sequence is stored as `WaveformDataset.transform`, and is applied automatically when elements are accessed through indexing.

12.1 GW transform sequence

For Dingo, the flowchart below indicates the sequence of transforms applied to a sample from a `WaveformDataset`.

Fig. 1: Flowchart for Dingo data-preprocessing pipeline for training, starting from a sample from a `WaveformDataset`. Transforms with rounded corners include an element of randomness, whereas trapezoidal items are deterministic.

Important: Some pre-processing transforms include an element of randomness. This serves to augment the training data and reduce overfitting.

12.1.1 Extrinsic parameters

The starting point for this chain of transforms is a sample `sample` with `parameters` and `polarizations` sub-dictionaries. The first transform samples the extrinsic parameters, and adds a new sub-dictionary `extrinsic_parameters` to `sample`. Extrinsic parameters include sky position (right ascension, declination), polarization, time of coalescense, and luminosity distance (the latter two of which are also considered intrinsic parameters).

class `dingo.gw.transforms.SampleExtrinsicParameters`(*extrinsic_prior_dict*)

Sample extrinsic parameters and add them to sample in a separate dictionary.

12.1.2 Detector waveforms

The next sequence of transforms applies the extrinsic parameters to `sample["polarizations"]` to produce detector waveforms in `sample["waveform"]`. First it calculates the arrival time t_I of the waveform in each detector, based on the time of coalescense at geocenter and the sky position, and stores this in `sample["extrinsic_parameters"]`,

`class dingo.gw.transforms.GetDetectorTimes(`*ifo_list*, *ref_time*`)`

Compute the time shifts in the individual detectors based on the sky position (ra, dec), the geocent_time and the ref_time.

Important: Dingo models are trained for a **fixed set of detectors.** This must be selected prior to training, and a new model must be trained if one wishes to analyze data in a different set of detectors. Thus, e.g., separate models must be trained for HL and HLV configurations.

Note: During training, Dingo **fixes the orientation of the Earth** (and corresponding interferometer positions and orientations) to that at a fixed reference time `ref_time`. This is so that the model does not have to learn about the rotation of the Earth. This is corrected in post-processing by shifting the inferred right ascension by the difference between the true and reference sidereal times.

Optionally, the times t_I are perturbed to give new "proxy times" as part of the *GNPE* algorithm.

`class dingo.gw.transforms.GNPECoalescenceTimes(`*ifo_list*, *kernel*, *exact_global_equivariance=True*, *inference=False*`)`

GNPE [1] Transformation for detector coalescence times.

For each of the detector coalescence times, a proxy is generated by adding a perturbation epsilon from the GNPE kernel to the true detector time. This proxy is subtracted from the detector time, such that the overall time shift only amounts to -epsilon in training. This standardizes the input data to the inference network, since the applied time shifts are always restricted to the range of the kernel.

To preserve information at inference time, conditioning of the inference network on the proxies is required. To that end, the proxies are stored in sample['gnpe_proxies'].

We can enforce an exact equivariance under global time translations, by subtracting one proxy (by convention: the first one, usually for H1 ifo) from all other proxies, and from the geocent time, see [1]. This is enabled with the flag exact_global_equivariance.

Note that this transform does not modify the data itself. It only determines the amount by which to time-shift the data.

[1]: arxiv.org/abs/2111.13139

Parameters

- **ifo_list** (*bilby.gw.detector.InterferometerList*) – List of interferometers.
- **kernel** (*str*) – Defines a Bilby prior, to be used for all interferometers.
- **exact_global_equivariance** (*bool = True*) – Whether to impose the exact global time translation symmetry.
- **inference** (*bool = False*) – Whether to use inference or training mode.

Finally, the detector waveforms h_I are calculated from the extrinsic parameters. (In the backend, these transforms use the Bilby interferometer libraries.) The contents of the `extrinsic_parameters` sub-dictionary are then moved into `sample["parameters"]`; this was essentially a holding place for parameters not yet applied to the waveform.

class dingo.gw.transforms.**ProjectOntoDetectors**(*ifo_list*, *domain*, *ref_time*)

Project the GW polarizations onto the detectors in ifo_list. This does not sample any new parameters, but relies on the parameters provided in sample['extrinsic_parameters']. Specifically, this transform applies the following operations:

(1) Rescale polarizations to account for sampled luminosity distance

(2) Project polarizations onto the antenna patterns using the ref_time and the extrinsic parameters (ra, dec, psi)

(3) Time shift the strains in the individual detectors according to the times <ifo.name>_time provided in the extrinsic parameters.

12.1.3 Noise

Once the detector waveforms have been obtained, noise n_I must be added to simulate realistic data. First, noise ASDs are selected randomly for each detector from an `ASDDataset` for the relevant observing run. This is stored in `sample["asds"]`. For details see *ASD dataset*.

class dingo.gw.transforms.**SampleNoiseASD**(*asd_dataset*)

Sample a random asds for each detector and add them to sample['asds'].

The waveform is then whitened based on the PSD, and furthermore scaled by the standard deviation of white noise. This is so that each input to the network will have unit variance, which is important for successful training.

class dingo.gw.transforms.**WhitenAndScaleStrain**(*scale_factor*)

Whiten the strain data by dividing w.r.t. the corresponding asds, and scale it with 1/scale_factor.

In uniform frequency domain the scale factor should be np.sqrt(window_factor) / np.sqrt(4.0 * delta_f). It has two purposes:

(*) the denominator accounts for frequency binning (*) dividing by window factor accounts for windowing of strain data

For whitened waveforms, noise is white, so finally this is randomly sampled and added to `sample["waveform"]`.

class dingo.gw.transforms.**AddWhiteNoiseComplex**

Adds white noise with a standard deviation determined by self.scale to the complex strain data.

12.1.4 Output

The final set of transforms prepares the sample for input to the neural network. First, the desired inference parameters are selected. By taking only a subset of `parameters`, one can train a marginalized posterior model. These parameters are also standardized to have zero mean and unit variance to improve training. (Standardization will be undone in post-processing after inference.) The parameters will then be repackaged into a `numpy.ndarray`, so that parameter labels are implicit based on ordering.

class dingo.gw.transforms.**SelectStandardizeRepackageParameters**(*parameters_dict*, *standardization_dict*, *inverse=False*, *as_type=None*, *device='cpu'*)

This transformation selects the parameters in standardization_dict, normalizes them by setting p = (p - mean) / std, and repackages the selected parameters to a numpy array.

as_type: str = None
only applies, if self.inverse == True * if None, data type is kept * if 'dict', dict with * if 'pandas', use pandas.DataFrame

The `waveform` and `asds` dictionaries are also repackaged into a single array of shape suitable for input to the network. In particular, the complex frequency domain strain data are decomposed into real and imaginary parts.

class `dingo.gw.transforms.`**`RepackageStrainsAndASDS`**(*ifos*, *first_index=0*)

Repackage the strains and the asds into an [num_ifos, 3, num_bins] dimensional tensor. Order of ifos is provided by self.ifos. By convention, [:,i,:] is used for:

i = 0: strain.real i = 1: strain.imag i = 2: 1 / (asd * 1e23)

Finally, the `samples` dictionary of arrays is unpacked to a tuple of arrays for parameters and data.

class `dingo.gw.transforms.`**`UnpackDict`**(*selected_keys*)

Unpacks the dictionary to prepare it for final output of the dataloader. Only returns elements specified in selected_keys.

When used with a torch `DataLoader`, the final numpy arrays are automatically transformed into torch tensors.

12.2 Building the transforms

The following function will set the `transform` property of a `WaveformDataset` to the above transform sequence:

`dingo.gw.training.`**`set_train_transforms`**(*wfd*, *data_settings*, *asd_dataset_path*, *omit_transforms=None*)

Set the transform attribute of a waveform dataset based on a settings dictionary. The transform takes waveform polarizations, samples random extrinsic parameters, projects to detectors, adds noise, and formats the data for input to the neural network. It also implements optional GNPE transformations.

Note that the WaveformDataset is modified in-place, so this function returns nothing.

Parameters

- **wfd** (`WaveformDataset`) –
- **data_settings** (*dict*) –
- **asd_dataset_path** (*str*) – Path corresponding to the ASD dataset used to generate noise.
- **omit_transforms** – List of sub-transforms to omit from the full composition.

The various options are specified by passing an appropriate `data_settings` dictionary. In practice, these settings will be specified along with other *training settings*.

Listing 1: Sample `data_settings` dictionary for configuring a sequence of training transforms. This dictionary includes several options not needed for `set_train_transforms`, but which are needed as part of other training settings.

```
waveform_dataset_path: /path/to/waveform_dataset.hdf5  # Contains intrinsic waveforms
train_fraction: 0.95
window:  # Needed to calculate window factor for simulated data
  type: tukey
  f_s: 4096
  T: 8.0
  roll_off: 0.4
domain_update:
  f_min: 20.0
  f_max: 1024.0
svd_size_update: 200  # Optionally, reduce the SVD size when decompressing (for␣
↪performance)
```

(continues on next page)

(continued from previous page)

```
detectors:
  - H1
  - L1
extrinsic_prior:  # Sampled at train time
  dec: default
  ra: default
  geocent_time: bilby.core.prior.Uniform(minimum=-0.10, maximum=0.10)
  psi: default
  luminosity_distance: bilby.core.prior.Uniform(minimum=100.0, maximum=1000.0)
ref_time: 1126259462.391
gnpe_time_shifts:
  kernel: bilby.core.prior.Uniform(minimum=-0.001, maximum=0.001)
  exact_equiv: True
inference_parameters: default
```

waveform_dataset_path
Points to the waveform dataset.

train_fraction
Fraction of waveform dataset to be used for training. The remainder are used to compute the test loss.

window
Specifies the window function to use when FFTing the time-domain data. It is used here to calculate a window factor for simulating data. See the discussion *here*.

domain_update (optional)
Optionally specify new domain properties. These will update the domain associated to the `WaveformDataset`. They must necessarily describe a domain contained within the original.

svd_size_update (optional)
If the `WaveformDataset` uses SVD compression, optionally use a smaller number of basis elements than stored in the dataset. Decompression of the waveforms is the slowest preprocessing operation, so using this option can improve training speed at the expense of accuracy.

detectors
Set the desired GW interferometers for the Dingo model.

extrinsic_prior
Specify the extrinsic prior. Default options are available.

ref_time
Reference time for the interferometer locations and orientations. See the *important note* above.

gnpe_time_shifts (optional)
GNPE kernel and additional options. See *GNPE*.

inference_parameters
Parameters to infer with the model. At present they must be a subset of `sample["parameters"]`. By specifying a strict subset, this can be used to marginalize over parameters. The `default` setting points to `dingo.gw.prior.default_inference_parameters`:

```
from dingo.gw.prior import default_inference_parameters
default_inference_parameters
```

```
/home/docs/checkouts/readthedocs.org/user_builds/dingo-gw/envs/latest/lib/python3.10/
↪site-packages/dingo/gw/__init__.py:3: UserWarning: Wswiglal-redir-stdio:
```

(continues on next page)

(continued from previous page)

```
SWIGLAL standard output/error redirection is enabled in IPython.
This may lead to performance penalties. To disable locally, use:

with lal.no_swig_redirect_standard_output_error():
    ...

To disable globally, use:

lal.swig_redirect_standard_output_error(False)

Note however that this will likely lead to error messages from
LAL functions being either misdirected or lost when called from
Jupyter notebooks.

To suppress this warning, use:

import warnings
warnings.filterwarnings("ignore", "Wswiglal-redir-stdio")
import lal

  import lal
```

```
['chirp_mass',
 'mass_ratio',
 'phase',
 'a_1',
 'a_2',
 'tilt_1',
 'tilt_2',
 'phi_12',
 'phi_jl',
 'theta_jn',
 'luminosity_distance',
 'geocent_time',
 'ra',
 'dec',
 'psi']
```

CHAPTER THIRTEEN

DETECTOR NOISE

During training, simulated noise n_I is added to waveforms $h_I(\theta)$ measured in detectors to produce realistic simulated data,

$$d_I = h_I(\theta) + n_I.$$

Dingo assumes this noise to be stationary and Gaussian, thus it is independent in each frequency bin, with variance given by some power spectral density (PSD).

Important: Similar to extrinsic parameters, detector noise is repeatedly sampled **during training** and added to the simulated signal. This augments the training set with new noise realizations for each epoch, reducing overfitting.

Although noise is *mostly* stationary and Guassian during an LVK observing run, the PSD in each detector does tend to drift from event to event. In a usual likelihood-based PE run, this is taken into account by estimating the PSD at the time of the event (either using Welch's method on signal-free data surrounding the event, or at the same time as the event using BayesWave), and using this in the likelihood integral.

Dingo also estimates the PSD just prior to an event and uses this at inference time in two ways:

1. It whitens the data with respect to this PSD.
2. It provides the PSD (or rather, the inverse ASD) as context to the neural network.

A suitably trained model can therefore make use of the PSD as needed to generate the posterior.

13.1 ASD dataset

To train a model to perform inference conditioned on the noise PSD, it is necessary to not just sample random noise realizations for a given PSD, but also **sample the PSD** from a distribution for a given observing run. Training in this way is necessary to perform fully amortized inference and account for the variation of PSDs from event to event.

The `ASDDataset` class stores a set of ASD samples for several detectors, allowing for sampling during training.

As with the noise realizations, a random ASD is chosen from the dataset when preparing each sample during training. This augments the training set compared to fixing the noise ASD for each sample prior to training.

Similarly to the `WaveformDataset`, the `ASDDataset` is just a container. Dingo includes routines for building such a dataset from observational data.

13.2 Generating an ASDDataset

13.2.1 dingo_generate_asd_dataset

The basic approach is as follows:

1. Identify stretches of data within an observing run meeting certain criteria (sufficiently long, without events, and sufficiently high quality, . . .) or take-in user-specified stretches.
2. Fetch data corresponding to these stretches using either
   - GWOSC
   - channels, optionally specified in the settings file.
3. Estimate ASDs using Welch's method on these stretches.
4. Save the collection of ASDs.

```
usage: dingo_generate_asd_dataset [-h] --data_dir DATA_DIR [--settings_file SETTINGS_
↪FILE] [--time_segments_file TIME_SEGMENTS_FILE] [--out_name OUT_NAME] [--verbose]

Generate an ASD dataset based on a settings file.

optional arguments:
  -h, --help            show this help message and exit
  --data_dir DATA_DIR   Path where the PSD data is to be stored. Must contain a
↪'settings.yaml' file.
  --settings_file SETTINGS_FILE
                        Path to a settings file in case two different datasets are␣
↪generated in the same directory
  --time_segments_file TIME_SEGMENTS_FILE
                        Optional file containing a dictionary of a list of time segments␣
↪that should be used for estimating PSDs.This has to be a pickle file.
  --out_name OUT_NAME   Path to resulting ASD dataset
  --verbose
```

where the settings file is of the form

```yaml
dataset_settings:
  f_min: 0
  f_max: 2048
  f_s: 4096
  time_psd: 1024
  T: 8
  time_gap: 0
  window:
    roll_off: 0.4
    type: tukey
  num_psds_max: 20
  channels:
   H1: H1:DCS-CALIB_STRAIN_C02
   L1: L1:DCS-CALIB_STRAIN_C02
  detectors:
    - H1
    - L1
```

(continues on next page)

(continued from previous page)

```
  observing_run: O2
condor:
  env_path: path/to/environment
  num_jobs: 2    # per detector
  num_cpus: 16
  memory_cpus: 16000
```

Options correspond to the following:

f_min, f_max (optional)
Lower and upper frequency range of the ASDs. Defaults to 0 and `f_s/2`, respectively.

Sampling rate f_s (Hz)
This should be at least twice the value of `f_max` expected to be used.

Data length time_psd (s)
The entire length of data from which to estimate a PSD using Welch's method. Periodigrams are calculated on segments of this, and then averaged using the `median` method.

Segment length T (s)
The length of each segment on which to take the DFT and calculate a periodigram.

Gap time_gap (s)
Gap between duration-T segments. E.g., if `time_psd=1024`, `T=8`, `time_gap=8`, then for each PSD, 64 periodigrams are computed, each using data stretches 8 s long, with gaps of 8 s between segments. Segments would then be $[0\text{ s}, 8\text{ s}]$, $[16\text{ s}, 24\text{ s}]$, $\ldots$.

Window function
Parameters of the window function used before taking DFT of data segments.

num_psds_max (optional)
If set, stop building the dataset after this number of PSDs have been estimated. This setting is useful for building a single-PSD dataset for pretraining a network.

Channels (optional)
If set, data will be fetched from these channels, instead of using GWOSC.

Detectors
Which detectors (H1, L1, V1, . . .) to include in the dataset.

Observing run
Which observing run to use when estimating PSDs.

Condor (optional)
Settings for HTCondor useful for parallelizing the ASD estimation across condor jobs.

13.2.2 dingo_generate_synthetic_asd_dataset

This method generates a dataset of synthetic ASDs from a dataset of existing ASDs to enhance robustness against ASD distribution shifts. In particular, this allows to generate a dataset of synthetic ASDs that are *scaled* by a fiducial ASD in order to adapt to a new observing run. This is particularly useful for training Dingo networks at the beginning of an observing run, when the number of training ASDs is limited. It also allows to generate smoother synthetic ASDs that more closely resemble those from BayesWave. The implementation follows the steps explained in this paper.

```
usage: dingo_generate_synthetic_asd_dataset [-h] --asd_dataset ASD_DATASET --settings_
↪file SETTINGS_FILE [--num_processes NUM_PROCESSES] [--out_file OUT_FILE] [--verbose]
```

(continues on next page)

(continued from previous page)

```
Generate a synthetic noise ASD dataset from an existing dataset of real ASDs.

optional arguments:
  -h, --help            show this help message and exit
  --asd_dataset ASD_DATASET
                        Path to existing ASD dataset to be parameterized and re-sampled
  --settings_file SETTINGS_FILE
                        YAML file containing database settings
  --num_processes NUM_PROCESSES
                        Number of processes to use in pool for parallel parameterization
  --out_file OUT_FILE   Name of file for storing dataset.
  --verbose
```

with a settings file of the form

```yaml
parameterization_settings:
  num_spline_positions: 30
  num_spectral_segments: 400
  sigma: 0.14
  delta_f: -1
  smoothen: True
sampling_settings:
   bandwidth_spectral: 0.5
   bandwidth_spline: 0.25
   num_samples: 500
   split_frequencies:
     - 30
     - 100
   rescaling_psd_paths:
     H1: /path/to/rescaling_asd_H1.hdf5
     L1: /path/to/rescaling_asd_L1.hdf5
```

Options correspond to the following:

num_spline_positions
: Number of nodes to use for the cubic spline interpolating the broad-band noise PSD.

num_spectral_segments
: Maximum number of spectral lines to model.

sigma
: Standard deviation of the Normal distribution parameterizing $p(\log S_n|z)$.

delta_f
: If > 0, truncates each spectral line.

smoothen
: Whether to save the smooth ASDs (True) or the noisy ASDs (False). The noisy synthetic ASDs resemble real ASDs estimated with Welch's method more closely, while the smooth ASDs are more similar to ASDs generated with BayesWave. (Default: False)

bandwidth_spectral, bandwidth_spline
: Bandwidths for the KDEs modeling the distribution over spectral lines and broad-band noise, respectively. These determine the width of the resulting distribution.

num_samples

Number of synthetic ASDs to generate.

`split_frequencies`
: (Set of) frequencies at dividing the broad-band noise into independent segments, e.g. due to different dominant noise sources (shot noise, seismic noise, etc.).

`rescaling_psd_paths`
: Paths to ASD datasets for each detector to which the synthetic ASDs should be rescaled, e.g. the PSDs from the target observing run. If the dataset contains multiple ASDs, we use the first one. (Optional; if not provided, no rescaling will be done.)

13.3 Data conditioning

Importantly, the variance of *white* noise in each frequency bin is not 1, but rather

$$\sigma^2_{\text{white}} = \frac{w}{4\delta f}$$

where δf is the frequency resolution and w is a "window factor".

The denominator in the noise variance is seen to arise most easily in the noise-weighted inner product,

$$(a|b) = 4\text{Re} \int_{f_{\text{min}}}^{f_{\text{max}}} df \, \frac{a^*(f)b(f)}{S_{\text{n}}(f)}$$

The window factor comes in because a window must be applied to time series data prior to taking the FFT. The windowing is assumed to reduce the power in the noise, but not affect the signal (which is localized away from the edge of the data segment). To simulate this, we add noise with variance scaled by the window factor.

The noise standard deviation is stored in the property `FrequencyDomain.noise_std`. The window factor is calculated from the data conditioning settings specified in the train settings file.

CHAPTER

FOURTEEN

NEURAL NETWORK ARCHITECTURE

Dingo is based on a method called Neural posterior estimation, see *here* for an introduction. A central object is the conditional neural density estimator, a deep neural network trained to represent the Bayesian posterior. This section describes the neural network architecture developed in [3], and subsequently used in [4], [5] and [6]. Note that Dingo can easily be extended to different architectures.

14.1 Neural spline flow with SVD compression

The architecture consists of two compenents, the embedding network which compresses the high-dimensionl data to a lower dimensional feature vector, and the conditional normalizing flow which estimates the Bayesian posterior based on this feature vector. Both components are trained jointly and end-to-end with the objective descriped *here*. The network can be build with

```
from dingo.core.nn.nsf import create_nsf_with_rb_projection_embedding_net
```

14.1.1 Embedding network

The embedding network compresses the high-dimensional conditioning information (consisting of frequency domain strain and PSD data). The first layer of this network is initialized with an SVD matrix from a reduced basis built with non-noisy waveforms. This projection filters out the noise that is orthogonal to the signal manifold, and significantly simplifies the task for the neural network.

The initial compression layer is followed by a sequence of residual blocks consisting of dense layers for further compression. Example kwargs:

```
embedding_net_kwargs = {
    "input_dims": (2, 3, 8033),
    "output_dim": 128,
    "hidden_dims": [
        1024, 1024, 1024, 1024, 1024, 1024, \
        512, 512, 512, 512, 512, 512, \
        256, 256, 256, 256, 256, 256, \
        128, 128, 128, 128, 128, 128
    ],
    "activation": "elu",
    "dropout": 0.0,
    "batch_norm": True,
    "svd": {
        "num_training_samples": 50000,
```

(continues on next page)

(continued from previous page)

```
        "num_validation_samples": 5000,
        "size": 200,
    }
}
```

Here, `input_dims=(2, 3, 8033)` refers to the input dimension, for frequency domain data with `8033` frequency bins and `3` channels (real part, complex part, ASD) in `2` detectors. The embedding network compresses this to `output_dim=128` components. The SVD initialization is controlled with the `svd` argument, and the residual blocks are specified with `hidden_dims`.

Note: Not all of these arguments have to be set in the configuration file when training dingo. For example, the `input_dims` argument is automatically filled in based on the specified domain information and number of detectors. Similarly, the `context_dim` of the flow (see below) is filled in based on the `output_dim` of the embedding network and the number of *GNPE* proxies. See the Dingo examples for the corresponding configuration files and training commands.

14.1.2 Flow

We use the neural spline flow as a density estimator. This takes the output of the embedding network as context information and estimates the Bayesian posterior distribution. Example kwargs:

```
nsf_kwargs = {
    "input_dim": 15,
    "context_dim": 129,
    "num_flow_steps": 30,
    "base_transform_kwargs": {
        "hidden_dim": 512,
        "num_transform_blocks": 5,
        "activation": "elu",
        "dropout_probability": 0.0,
        "batch_norm": True,
        "num_bins": 8,
        "base_transform_type": "rq-coupling",
    },
}
```

This creates a neural spline flow with `input_dim=15` parameters, conditioned on a `129` dimensional context vector, corresponding to the `128` dimensional output of the embedding network and one *GNPE* proxy variable. The neural spline flow consists of `num_flow_steps=30` layers, for which the transformation is specified with `base_transform_kwargs`.

```
nde = create_nsf_with_rb_projection_embedding_net(nsf_kwargs, embedding_net_kwargs)
```

CHAPTER

FIFTEEN

TRAINING

Training a network can require a significant amount of time (for production models, typically a week with a fast GPU). We therefore expect that this will almost always be done non-interactively using a command-line script. Dingo offers two options, `dingo_train` and `dingo_train_condor`, depending on whether your GPU is local or cluster-based.

Both of these scripts take as main argument a settings file, which specifies options relating to *Data pre-processing*, training strategy, *Neural network architecture*, hardware, and checkpointing. They produce a trained model in PyTorch `.pt` format, and they save checkpoints and the training history. The settings file is furthermore saved within the model files for reproducibility and to be able to resume training from a checkpoint. Finally, all *precursor* settings files (for the waveform or noise datasets) are also saved with the model.

15.1 Settings file

Listing 1: Example `train_settings.yaml` file. This is also available in the examples/ folder. The specific settings listed will train a production-size network, taking about a week on an NVIDIA A100. Consider reducing some model hyperparameters for experimentation.

```
data:
  waveform_dataset_path: /path/to/waveform_dataset.hdf5  # Contains intrinsic waveforms
  train_fraction: 0.95
  window:
    type: tukey
    f_s: 4096
    T: 8.0
    roll_off: 0.4
  domain_update:
    f_min: 20.0
    f_max: 1024.0
  svd_size_update: 200
  detectors:
    - H1
    - L1
  extrinsic_prior:
    dec: default
    ra: default
    geocent_time: bilby.core.prior.Uniform(minimum=-0.10, maximum=0.10)
    psi: default
    luminosity_distance: bilby.core.prior.Uniform(minimum=100.0, maximum=1000.0)
  ref_time: 1126259462.391
```

(continues on next page)

(continued from previous page)

```yaml
  gnpe_time_shifts:
    kernel: bilby.core.prior.Uniform(minimum=-0.001, maximum=0.001)
    exact_equiv: True
  inference_parameters: default

model:
  type: nsf+embedding
  nsf_kwargs:
    num_flow_steps: 30
    base_transform_kwargs:
      hidden_dim: 512
      num_transform_blocks: 5
      activation: elu
      dropout_probability: 0.0
      batch_norm: True
      num_bins: 8
      base_transform_type: rq-coupling
  embedding_net_kwargs:
    output_dim: 128
    hidden_dims: [1024, 1024, 1024, 1024, 1024, 1024,
                  512, 512, 512, 512, 512, 512,
                  256, 256, 256, 256, 256, 256,
                  128, 128, 128, 128, 128, 128]
    activation: elu
    dropout: 0.0
    batch_norm: True
    svd:
      num_training_samples: 20000
      num_validation_samples: 5000
      size: 200

# Training is divided in stages. They each require all settings as indicated below.
training:
  stage_0:
    epochs: 300
    asd_dataset_path: /path/to/asds_fiducial.hdf5
    freeze_rb_layer: True
    optimizer:
      type: adam
      lr: 0.0001
    scheduler:
      type: cosine
      T_max: 300
    batch_size: 64

  stage_1:
    epochs: 150
    asd_dataset_path: /path/to/asds.hdf5
    freeze_rb_layer: False
    optimizer:
      type: adam
      lr: 0.00001
```

(continues on next page)

(continued from previous page)

```yaml
    scheduler:
      type: cosine
      T_max: 150
    batch_size: 64

# Local settings that have no impact on the final trained network.
local:
  device: cpu  # Change this to 'cuda' for training on a GPU.
  num_workers: 6
#  wandb:
#    project: dingo
#    group: O4
  runtime_limits:
    max_time_per_run: 36000
    max_epochs_per_run: 500
  checkpoint_epochs: 10
#   condor:
#     bid: 100
#     num_cpus: 16
#     memory_cpus: 128000
#     num_gpus: 1
#     memory_gpus: 8000
```

The train settings file is grouped into **four sections:**

15.1.1 data_settings

These settings point to a saved dataset of waveform polarizations and describe the transforms to obtain detector waveforms. A detailed description of these settings is available *here*.

15.1.2 model

This describes the model architecture, including network type and hyperparameters. All of these settings are described in the section on *Neural network architecture*.

15.1.3 training

This describes the training strategy. Training is divided into **stages**, each of which can differ to some extent. Stages are numbered (`stage_0`, `stage_1`, . . .) and executed in this order. Each stage is defined by the following settings:

epochs
: Total number of training epochs for the stage. The network sees the entire training set once per epoch.

asd_dataset_path
: Points to an `ASDDataset` file. Each stage can have its own ASD dataset, which is useful for implementing a pre-training stage with fixed ASD and a fine-tuning stage with variable ASD.

freeze_rb_layer
: Whether to freeze the first layer of the embedding network in `nsf+embedding` models. This layer is seeded with reduced (SVD) basis vectors, so freezing this layer during pre-training simply projects data onto the basis coefficients. In the fine-tuning stage, when other weights are more stable, unfreezing this can be useful.

optimizer
Specify optimizer type and parameters such as initial learning rate.

scheduler
Use a learning rate scheduler to reduce the learning rate over time. This can improve overall optimization.

batch_size
Number of training samples per mini-batch. For a training dataset of size N, then each epoch will consist of $N/$

Important: The stage-training framework allows for separate pre-training and fine-tuning stages. We found that having a pre-training stage where we freeze certain network weights and fix the noise ASD improves overall training results.

15.1.4 `local`

The `local` settings are the only group that have no impact on the final trained network. Indeed, they are not even saved in the `.pt` files; rather they are split off and saved in a new file `local_settings.yaml`.

device
`cpu` or `cuda`. Training on a GPU with CUDA is highly recommended.

num_workers
Number of CPU worker processes to use for pre-processing training data before copying to the GPU. Data pre-processing (inluding decompression, projection to detectors, and noise generation) is quite expensive, so using 16 or 32 processes is recommended, otherwise this can become a bottleneck. We recommend monitoring the GPU utilization percentage as well as time spent on pre-processing (output during training) to fine-tune this number.

wandb
Settings for Weights & Biases. If you have an account, you can use this to track your training progress and compare different runs.

runtime_limits
Maximum time (in seconds) or maximum number of epochs per run. Using this could make sense in a cluster environment.

checkpoint_epochs
Dingo saves a temporary checkpoint in `model_latest.py` after every epoch, but this is later overwritten by the next checkpoint. This setting saves a permanent checkpoint after the specified number of epochs. Having these checkpoints can help in recovering from training failures that do not result in program termination.

condor
Settings for HTCondor. The condor script will (re)submit itself according to these options.

15.2 Command-line scripts

15.2.1 `dingo_train`

On a local machine, simply pass the settings file (or checkpoint) and an output directory to `dingo_train`. It will train until complete, or until a runtime limit is reached.

```
usage: dingo_train [-h] [--settings_file SETTINGS_FILE] --train_dir TRAIN_DIR [--
↪checkpoint CHECKPOINT]
```

(continues on next page)

(continued from previous page)

```
Train a neural network for gravitational-wave single-event inference.

This program can be called in one of two ways:
    a) with a settings file. This will create a new network based on the
    contents of the settings file.
    b) with a checkpoint file. This will resume training from the checkpoint.

optional arguments:
  -h, --help            show this help message and exit
  --settings_file SETTINGS_FILE
                        YAML file containing training settings.
  --train_dir TRAIN_DIR
                        Directory for Dingo training output.
  --checkpoint CHECKPOINT
                        Checkpoint file from which to resume training.
```

15.2.2 dingo_train_condor

On a cluster using HTCondor, use `dingo_train_condor`. This calls itself recursively as follows:

1. The first time you call it, use the flag `--start-submission`. This creates a condor submission file `submission_file.sub` that again calls the executable `dingo_train_condor` (now without the flag) and submits it. This will run `dingo_train_condor` directly on the cluster node that is assigned.
2. On the cluster node, `dingo_train_condor` first trains the network until done or a runtime limit is reached (be careful to set this shorter than the condor time limit). Then it creates a *new* submission file that once again calls `dingo_train_condor`, and submits it. This will resume the run on a new node, and repeat.

```
usage: dingo_train_condor [-h] --train_dir TRAIN_DIR [--checkpoint CHECKPOINT] [--start_
↪submission]

optional arguments:
  -h, --help            show this help message and exit
  --train_dir TRAIN_DIR
                        Directory for Dingo training output.
  --checkpoint CHECKPOINT
  --start_submission
```

15.3 Output

Output from training is stored in the `TRAIN_DIR` folder passed to the training scripts. This consists of the following:

- `model_latest.pt` checkpoints every epoch (overwritten);
- `model_XXX.pt` checkpoints where `XXX` is the epoch number, every `checkpoint_epochs` epochs;
- `model_stage_X.pt` at the end of training stage `X`;
- `history.txt` with columns (epoch number, train loss, test loss, learning rate);
- `svd_L1.hdf5`, . . . , storing SVD basis information used for seeding the embedding network;
- `local_settings.yaml` with local settings for the run (not stored with checkpoints).

The .pt and .hdf5 files may all be inspected using dingo_ls. This prints all the settings, as well as diagnostic information for SVD bases. The saved settings include all the settings provided in the settings file, as well as several derived quantities, such as parameter standardizations, additional context parameters (for GNPE), etc.

15.3.1 Modifying a checkpoint

Occasionally it may be necessary to change a setting of a partially trained model. For example, a model may have been successfully pre-trained, but the fine-tuning failed, and one may wish to change the fine-tuning settings without starting from scratch. Since the model setting are all stored with the checkpoint, they just need to be changed.

The script dingo_append_training_stage allows for appending a model stage or replacing an existing planned stage. It will fail if the stage has already begun training, so be sure to use it on a sufficiently early checkpoint.

```
usage: dingo_append_training_stage [-h] --checkpoint CHECKPOINT --stage_settings_file␣
↪STAGE_SETTINGS_FILE --out_file OUT_FILE [--replace REPLACE]

optional arguments:
  -h, --help            show this help message and exit
  --checkpoint CHECKPOINT
  --stage_settings_file STAGE_SETTINGS_FILE
  --out_file OUT_FILE
  --replace REPLACE
```

For more detailed adjustments to the training settings the script one can use the script compatibility/update_model_metadata.py.

```
usage: update_model_metadata.py [-h] --checkpoint CHECKPOINT --key KEY [KEY ...] --value␣
↪VALUE

optional arguments:
  -h, --help            show this help message and exit
  --checkpoint CHECKPOINT
  --key KEY [KEY ...]
  --value VALUE
```

> **Warning:** Modifications to model metadata can easily break things. Do not use this unless completely sure what you are doing!

CHAPTER

SIXTEEN

INFERENCE

With a trained network, inference can be performed on real data by executing following on the command line:

```
dingo_analyze_event
  --model model.pt
  --gps_time_event gps_time_event
  --num_samples num_samples
  --batch_size batch_size
```

This will download data from GWOSC at the specified time, apply the data conditioning consistent with the trained Dingo model and transform to frequency domain, and generate the requested number of posterior samples. It will save them in a file `dingo_samples-gps_time_event.hdf5`, along with *all* settings used in upstream components of Dingo (the waveform dataset, noise dataset, and model training) and the data analyzed.

The `dingo_analyze_event` script can also be used to analyze an *injection*.

16.1 The `Sampler` class

Under the hood, the inference script uses the `Sampler` class, or more specifically, the `GWSampler` class, which inherits from it.

class `dingo.gw.inference.gw_samplers.`**`GWSampler`**`(**kwargs)`

Bases: *`GWSamplerMixin`*, *`Sampler`*

Sampler for gravitational-wave inference using neural posterior estimation. Augments the base class by defining transform_pre and transform_post to prepare data for the inference network.

transform_pre :

- Whitens strain.
- Repackages strain data and the inverse ASDs (suitably scaled) into a torch tensor.

transform_post :

- Extract the desired inference parameters from the network output (array-like), de-standardize them, and repackage as a dict.

Also mixes in GW functionality for building the domain and correcting the reference time.

Allows for conditional and unconditional models, and draws samples from the model based on (optional) context data.

This is intended for use either as a standalone sampler, or as a sampler producing initial sample points for a GNPE sampler.

Parameters
`kwargs` – Keyword arguments that are forwarded to the superclass.

`property context`

Data on which to condition the sampler. For injections, there should be a 'parameters' key with truth values.

`property event_metadata`

Metadata for data analyzed. Can in principle influence any post-sampling parameter transformations (e.g., sky position correction), as well as the likelihood detector positions.

`log_prob`(*samples: DataFrame*) → ndarray

Calculate the model log probability at specific sample points.

Parameters
`samples` (*pd.DataFrame*) – Sample points at which to calculate the log probability.

Return type
np.array of log probabilities.

`run_sampler`(*num_samples: int*, *batch_size: int | None = None*)

Generates samples and stores them in self.samples. Conditions the model on self.context if appropriate (i.e., if the model is not unconditional).

If possible, it also calculates the log_prob and saves it as a column in self.samples. When using GNPE it is not possible to obtain the log_prob due to the many Gibbs iterations. However, in the case of just one iteration, and when starting from a sampler for the proxy, the GNPESampler does calculate the log_prob.

Allows for batched sampling, e.g., if limited by GPU memory. Actual sampling for each batch is performed by _run_sampler(), which will differ for Sampler and GNPESampler.

Parameters

- `num_samples` (*int*) – Number of samples requested.
- `batch_size` (*int, optional*) – Batch size for sampler.

`to_result`() → *Result*

Export samples, metadata, and context information to a Result instance, which can be used for saving or, e.g., importance sampling, training an unconditional flow, etc.

Return type
Result

This is instantiated based on a `PosteriorModel`. To draw samples, the `context` property must first be set to the data to be analyzed. For gravitational waves this should be a dictionary with the following keys:

waveform
(unwhitened) strain data in each detector

asds
noise ASDs estimated in each detector at the time of the event

parameters (optional)
for injections, the true parameters of the signal (for saving; ignored for sampling)

Once this is set, the `run_sampler()` method draws the requested samples from the posterior conditioned on the context. It applies some post-processing (to de-standardize the data, and to correct for the rotation of the Earth between the network reference time and the event time), and then stores the result as a DataFrame in `GWSampler.samples`. The DataFrame contains columns for each inference parameter, as well as the log probability of the sample under the posterior model.

The `GWSampler.metadata` attribute contains all settings that went into producing the samples, including training datasets, network training settings, event metadata (for real events) and possible injection parameters. Finally, the `to_samples_dataset()` method returns a `SamplesDataset` containing all results, including the samples, settings, and context. This can be saved easily as HDF5.

16.2 Injections

Injections (i.e., simulated data) are produced using the `Injection` class. It includes options for fixed or random parameters (drawn from a prior), and it returns injections in a format that can be directly set as `GWSampler.context`.

class `dingo.gw.injection.`**`Injection`**(*prior*, ***gwsignal_kwargs*)

Bases: *`GWSignal`*

Produces injections of signals (with random or specified parameters) into stationary Gaussian noise. Output is not whitened.

Parameters

- **prior** (*PriorDict*) – Prior used for sampling random parameters.
- **gwsignal_kwargs** – Arguments to be passed to GWSignal base class.

classmethod from_posterior_model_metadata(*metadata*)

Instantiate an Injection based on a posterior model. The prior, waveform settings, etc., will all be consistent with what the model was trained with.

Parameters

metadata (*dict*) – Dict which you can get via PosteriorModel.metadata

injection(*theta*)

Generate an injection based on specified parameters.

This is a signal + noise consistent with the amplitude spectral density in self.asd. If self.asd is an ASDDataset, then it uses a random ASD from this dataset.

Data are not whitened.

Parameters

theta (*dict*) – Parameters used for injection.

Returns

keys:

waveform: data (signal + noise) in each detector extrinsic_parameters: {} parameters: waveform parameters asd (if set): amplitude spectral density for each detector

Return type

dict

random_injection()

Generate a random injection.

This is a signal + noise consistent with the amplitude spectral density in self.asd. If self.asd is an ASDDataset, then it uses a random ASD from this dataset.

Data are not whitened.

Returns

keys:
waveform: data (signal + noise) in each detector extrinsic_parameters: {} parameters: waveform parameters asd (if set): amplitude spectral density for each detector

Return type
dict

Hint: The convenience class method `from_posterior_model_metadata()` instantiates an `Injection` with all of the settings that went into the posterior model. To this class pass the PosteriorModel.metadata dictionary. It should produce injections that perfectly match the characteristics of the training data (waveform approximant, data conditioning, noise characteristics, etc.). This can be very useful for testing a trained model.

Important: Repeated calls to `Injection.injection()`, even with the same parameters, will produce injections with different noise realizations (which therefore lead to different posteriors). For repeated analyses of the *exact same* injection (e.g., with different models or codes) it is necessary to either save the injection for re-use or fix a random seed.

CHAPTER SEVENTEEN

GNPE

GNPE (Gibbs- or Group-Equivariant Neural Posterior Estimation) is an algorithm that can generate significantly improved results by incorporating known physical symmetries into NPE.[1] The aim is to simplify the data seen by the network by using the symmetries to transform certain parameters to "standardized" values. This simplifies the learning task of the network. At inference time, the standardizing transform is initially unknown, so we use Gibbs sampling to simultaneously learn the transform (along with the rest of the parameters) *and* apply it to simplify the data.

For gravitational waves, we use GNPE to standardize the times of arrival of the signal in the individual interferometers. (This corresponds to translations of the time of arrival at geocenter, and approximate sky rotations.) In frequency domain, time translations correspond to multiplication of the data by $e^{-2\pi i f \Delta t}$, and a standard NPE network would have to learn to interpret such transformations consistent with the prior from the data. We found this to be a challenging learning task, which limited inference performance on the other parameters. Instead, GNPE leverages our knowledge of the time translations to build a network that is only required to interpret a much narrower window of arrival times.

We now provide a brief description of the GNPE method. Readers more interested in getting started with GNPE may skip to *Usage* below.

17.1 Description of method

GNPE allows us to incorporate knowledge of **joint symmetries of data and parameters**. That is, if a parameter (e.g., coalescence time) is transformed by a certain amount (Δt), then there is a corresponding transformation of the data (multiplication by $e^{-2\pi i f \Delta t}$) such that the transformed data is equally likely to occur under the transformed parameter,

$$p(t_c|d) = p(t_c + \Delta t | d \cdot e^{-2\pi i f \Delta t}).$$

It is based on two ideas:

17.1.1 Gibbs + NPE

Gibbs sampling is an algorithm for obtaining samples from a joint distribution $p(x, y)$ if we are able to sample directly from each of the conditionals, $p(x|y)$ and $p(y|x)$. Starting from some point y_0, we construct a Markov chain $\{(x_i, y_i)\}$ by sampling

1. $x_i \sim p(x_i|y_{i-1})$,
2. $y_i \sim p(y_i|x_i)$,

and repeating until the chain is converged. The stationary distribution of the Markov chain is then $p(x, y)$.

Gibbs sampling can be combined with NPE by first introducing blurred "proxy" versions of a subset of parameters, which we denote $\hat{\theta}$ i.e., $\hat{\theta} \sim p(\hat{\theta}|\theta)$ where $p(\hat{\theta}|\theta)$ is defined by a blurring kernel. For example, for GWs we take

[1] Maximilian Dax, Stephen R. Green, Jonathan Gair, Michael Deistler, Bernhard Schölkopf, and Jakob H. Macke. Group equivariant neural posterior estimation. *International Conference on Learning Representations*, 2022. arXiv:2111.13139.

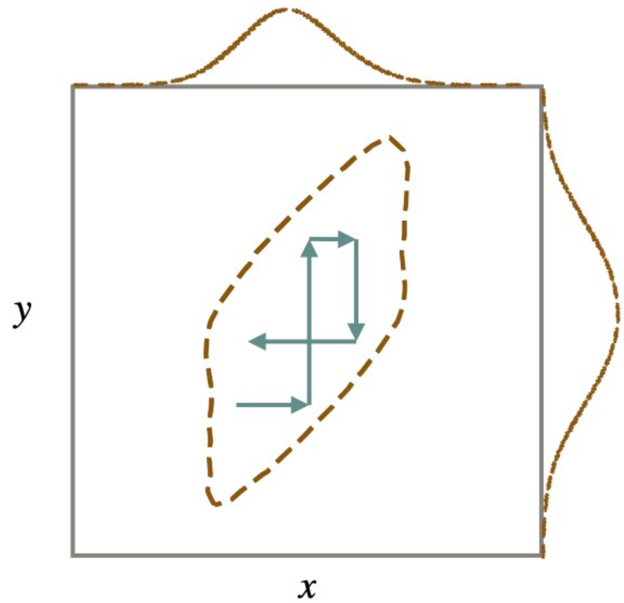


Fig. 1: Illustration of Gibbs sampling for a distribution $p(x, y)$.

$\hat{t}_I = t_I + \epsilon_I$, where $\epsilon_I \sim \text{Unif}(-1\text{ ms}, 1\text{ ms})$. We then train a network to model the posterior, but now conditioned also on $\hat{\theta}$, i.e., $p(\theta|d, \hat{\theta})$. We can then apply Gibbs sampling to obtain samples from the joint distribution $p(\theta, \hat{\theta}|d)$, since we are able to sample individually from the conditional distributions:

- We can sample from $p(\hat{\theta}|\theta)$ since we defined the blurring kernel.
- We can sample from $p(\theta|d, \hat{\theta})$ since we are modeling it using NPE.

Finally, we can drop $\hat{\theta}$ from the samples to obtain the desired posterior samples.

The trick now is that since $p(\theta|d, \hat{\theta})$ is conditional on $\hat{\theta}$, we can apply any $\hat{\theta}$-dependent transformation to d. Returning to the time translations, $\hat{t}_I$ is a good approximation to t_I, so we apply the inverse time shift $d_I \to d_I \cdot e^{2\pi i f \hat{t}_I}$, which brings d_I into a close approximation to having coalescence time 0 in each detector. This means that the network never sees any data with merger time further than 1 ms from 0, greatly simplifying the learning task.

In practice, we generate many Monte Carlo chains in parallel—one for each desired sample and with different starting points—and keep only the final sample from each chain—rather than generating one long chain. Each individual chain in this ensemble is unlikely to converge, but if the individual chains are initialized from a distribution sufficiently close to $p(\hat{\theta}|d)$ then the collection of final samples from each chain should be a good approximation to samples from $p(\theta, \hat{\theta}|d)$.

17.1.2 Group-equivariant NPE

So far we have described how Gibbs sampling together with NPE can simplify data by allowing any $\hat{\theta}$-dependent transformation of d, simplifying the data distribution. If we know the data and parameters to be equivariant under a particular transformation, however, we can go a step further and enforce this exactly. To do so, we simply drop the dependence of the neural density estimator on $\hat{\theta}$.

For gravitational waves, the overall time translation symmetry (in each detector) of the time of coalescence at geocenter is an exact symmetry, so we fully enforce this. The sky rotation, however, corresponds to an approximate symmetry: it shifts the time of coalescence in each detector, but a subleading effect is to change angle of incidence on a detector and hence the combination of polarizations observed. For this latter symmetry, we simply do not drop the proxy dependence.

Tip: GNPE is a generic method to incorporate symmetries into NPE:

- **Any** symmetry (exact or approximate) connecting data and parameters
- **Any** architecture, as it just requires (at most) conditioning on the proxy variables

As far as we are aware, GNPE is the only way to incorporate symmetries connecting data and parameters into architectures such as normalizing flows.

17.2 Usage

17.2.1 Training

To use GNPE for GW inference one must train **two** Dingo models:

1. An **initialization network** modeling $p(t_I|d)$. This gives the initial guess of the proxy variables for the staring point of the Gibbs sampler. Since this is only modeling two or three parameters and it does not need to give perfect results, this network can also be much smaller than typical Dingo networks.

   For an HL detector network, to infer *just* the detector coalescence times, set this in the train configuration.

   ```
   data:
     inference_parameters: [H1_time, L1_time]
   ```

2. A **main "GNPE" network**, conditional on the proxy variables, $p(\theta|d, \hat{t}_I)$. Implicitly in this expression, the data are transformed by the proxies, and the exact time-translation symmetry is enforced.

   To condition this network on the correct proxies, we configure it to use GNPE in the settings file.

   ```
   data:
     gnpe_time_shifts:
       kernel: bilby.core.prior.Uniform(minimum=-0.001, maximum=0.001)
       exact_equiv: True
   ```

   This sets the blurring kernel to be $\mathrm{Unif}(-1\ \mathrm{ms}, 1\ \mathrm{ms})$ for all $\hat{t}_I$, and it specifies to enforce the overall time of coalescence symmetry exactly. Dingo will determine automatically from the `detectors` setting which proxy variables to condition on.

Complete example config files for both networks are provided in the /examples folder.

17.2.2 Inference

The inference script must be pointed to both trained networks in order to sample using GNPE.

```
dingo_analyze_event
  --model model
  --model_init model_init
  --gps_time_event gps_time_event
  --num_samples num_samples
  --num_gnpe_iterations num_gnpe_iterations
  --batch_size batch_size
```

The number of Gibbs iterations is also specified here (typically 30 is appropriate). This script will save the final samples from each Gibbs chain.

17.3 The `GNPESampler` class

The inference script above uses the `GWSamplerGNPE` class, which is based on `GNPESampler`,

class `dingo.core.samplers.`**`GNPESampler`**(*model: PosteriorModel*, *init_sampler: Sampler*, *num_iterations: int = 1*)

Bases: *`Sampler`*

Base class for GNPE sampler. It wraps a PosteriorModel *and* a standard Sampler for initialization. The former is used to generate initial samples for Gibbs sampling.

A GNPE network is conditioned on additional "proxy" context theta^, i.e.,

p(theta | theta^, d)

The theta^ depend on theta via a fixed kernel p(theta^ | theta). Combining these known distributions, this class uses Gibbs sampling to draw samples from the joint distribution,

p(theta, theta^ | d)

The advantage of this approach is that we are allowed to perform any transformation of d that depends on theta^. In particular, we can use this freedom to simplify the data, e.g., by aligning data to have merger times = 0 in each detector. The merger times are unknown quantities that must be inferred jointly with all other parameters, and GNPE provides a means to do this iteratively. See https://arxiv.org/abs/2111.13139 for additional details.

Gibbs sampling breaks access to the probability density, so this must be recovered through other means. One way is to train an unconditional flow to represent p(theta^ | d) for fixed d based on the samples produced through the GNPE Gibbs sampling. Starting from these, a single Gibbs iteration gives theta from the GNPE network, along with the probability density in the joint space. This is implemented in GNPESampler provided the init_sampler provides proxies directly and num_iterations = 1.

17.3.1 Attributes (beyond those of Sampler)

init_sampler
[Sampler] Used for providing initial samples for Gibbs sampling.

num_iterations
[int] Number of Gibbs iterations to perform.

iteration_tracker : IterationTracker **not set up** remove_init_outliers : float **not set up**

param model
type model
PosteriorModel

param init_sampler
Used for generating initial samples

type init_sampler
Sampler

param num_iterations
Number of GNPE iterations to be performed by sampler.

type num_iterations
int

`property context`

Data on which to condition the sampler. For injections, there should be a 'parameters' key with truth values.

`property event_metadata`

Metadata for data analyzed. Can in principle influence any post-sampling parameter transformations (e.g., sky position correction), as well as the likelihood detector positions.

`log_prob`(*samples: DataFrame*) → ndarray

Calculate the model log probability at specific sample points.

Parameters
`samples` (*pd.DataFrame*) – Sample points at which to calculate the log probability.

Return type
np.array of log probabilities.

`property num_iterations`

The number of GNPE iterations to perform when sampling.

`run_sampler`(*num_samples: int*, *batch_size: int | None = None*)

Generates samples and stores them in self.samples. Conditions the model on self.context if appropriate (i.e., if the model is not unconditional).

If possible, it also calculates the log_prob and saves it as a column in self.samples. When using GNPE it is not possible to obtain the log_prob due to the many Gibbs iterations. However, in the case of just one iteration, and when starting from a sampler for the proxy, the GNPESampler does calculate the log_prob.

Allows for batched sampling, e.g., if limited by GPU memory. Actual sampling for each batch is performed by _run_sampler(), which will differ for Sampler and GNPESampler.

Parameters

- **num_samples** (*int*) – Number of samples requested.
- **batch_size** (*int, optional*) – Batch size for sampler.

`to_result`() → *Result*

Export samples, metadata, and context information to a Result instance, which can be used for saving or, e.g., importance sampling, training an unconditional flow, etc.

Return type
Result

In addition to storing a `PosteriorModel`, a `GNPESampler` also stores a second `Sampler` instance, which is based on the initialization network. When `run_sampler()` is called, it first generates samples from the initialization network, perturbs them with the kernel to obtain proxy samples, and then performs `num_iterations` Gibbs steps to obtain the final samples.

CHAPTER

EIGHTEEN

THE RESULT CLASS

The `Result` class stores the output of a `Sampler` run, namely a collection of samples. It contains several methods for operating on the samples, including for **importance sampling**, **plotting**, and **density recovery**:

class dingo.gw.result.**Result**(***kwargs*)

Bases: *Result*

A dataset class to hold a collection of gravitational-wave parameter samples and perform various operations on them.

Compared to the base class, this class implements the domain, prior, and likelihood. It also includes a method for sampling the binary reference phase parameter based on the other parameters and the likelihood.

Attributes:

samples
[pd.Dataframe] Contains parameter samples, as well as (possibly) log_prob, log_likelihood, weights, log_prior, delta_log_prob_target.

domain
[Domain] The domain of the data (e.g., FrequencyDomain), needed for calculating likelihoods.

prior
[PriorDict] The prior distribution, used for importance sampling.

likelihood
[Likelihood] The Likelihood object, needed for importance sampling.

context
[dict] Context data from which the samples were produced (e.g., strain data, ASDs).

metadata
[dict] Metadata inherited from the Sampler object. This describes the neural networks and sampling settings used.

event_metadata
[dict] Metadata for the event analyzed, including time, data conditioning, channel, and detector information.

log_evidence
[float] Calculated log(evidence) after importance sampling.

log_evidence_std
[float (property)] Standard deviation of the log(evidence)

effective_sample_size, n_eff
[float (property)] Number of effective samples, (sum_i w_i)^2 / sum_i w_i^2

sample_efficiency
[float (property)] Number of effective samples / Number of samples

synthetic_phase_kwargs
[dict] kwargs describing the synthetic phase sampling.

For constructing, provide either file_name, or dictionary containing data and settings entries, or neither.

Parameters

- `file_name` (*str*) – HDF5 file containing a dataset
- `dictionary` (*dict*) – Contains settings and data entries. The data keys should be the same as save_keys
- `data_keys` (*list*) – Variables that should be saved / loaded. This allows for class to store additional variables beyond those that are saved. Typically, this list would be provided by any subclass.

`get_samples_bilby_phase()`

Convert the spin angles phi_jl and theta_jn to account for a difference in phase definition compared to Bilby.

Returns
Samples

Return type
pd.DataFrame

`importance_sample`(*num_processes: int = 1*, ***likelihood_kwargs*)

Calculate importance weights for samples.

Importance sampling starts with samples have been generated from a proposal distribution q(theta), in this case a neural network model. Certain networks (i.e., non-GNPE) also provide the log probability of each sample, which is required for importance sampling.

Given the proposal, we re-weight samples according to the (un-normalized) target distribution, which we take to be the likelihood L(theta) times the prior pi(theta). This gives sample weights

w(theta) ~ pi(theta) L(theta) / q(theta),

where the overall normalization does not matter (and we take to have mean 1). Since q(theta) enters this expression, importance sampling is only possible when we know the log probability of each sample.

As byproducts, this method also estimates the evidence and effective sample size of the importance sampled points.

This method modifies the samples pd.DataFrame in-place, adding new columns for log_likelihood, log_prior, and weights. It also stores the log_evidence as an attribute.

Parameters

- `num_processes` (*int*) – Number of parallel processes to use when calculating likelihoods. (This is the most expensive task.)
- `likelihood_kwargs` (*dict*) – kwargs that are forwarded to the likelihood constructor. E.g., options for marginalization.

`classmethod merge`(*parts*)

Merge several Result instances into one. Check that they are compatible, in the sense of having the same metadata. Finally, calculate a new log evidence for the combined result.

This is useful when recombining separate importance sampling jobs.

Parameters
`parts` (*list[Result]*) – List of sub-Results to be combined.

Return type
Combined Result.

`parameter_subset`(*parameters*)

Return a new object of the same type, with only a subset of parameters. Drops all other columns in samples DataFrame as well (e.g., log_prob, weights).

Parameters
`parameters` (*list*) – List of parameters to keep.

Return type
Result

`property pesummary_prior`

The prior in a form suitable for PESummary.

By convention, Dingo stores all times *relative* to a reference time, typically the trigger time for an event. The prior returned here corrects for that offset to be consistent with other codes.

`property pesummary_samples`

Samples in a form suitable for PESummary.

These samples are adjusted to undo certain conventions used internally by Dingo:

- Times are corrected by the reference time t_ref.
- Samples are unweighted, using a fixed random seed for sampling importance

resampling. * The spin angles phi_jl and theta_jn are transformed to account for a difference in phase definition. * Some columns are dropped: delta_log_prob_target, log_prob

`plot_corner`(*parameters=None*, *filename='corner.pdf'*)

Generate a corner plot of the samples.

Parameters

- `parameters` (*list[str]*) – List of parameters to include. If None, include all parameters. (Default: None)
- `filename` (*str*) – Where to save samples.

`plot_log_probs`(*filename='log_probs.png'*)

Make a scatter plot of the target versus proposal log probabilities. For the target, subtract off the log evidence.

`plot_weights`(*filename='weights.png'*)

Make a scatter plot of samples weights vs log proposal.

`print_summary`()

Display the number of samples, and (if importance sampling is complete) the log evidence and number of effective samples.

`reset_event`(*event_dataset*)

Set the Result context and event_metadata based on an EventDataset.

If these attributes already exist, perform a comparison to check for changes. Update relevant objects appropriately. Note that setting context and event_metadata attributes directly would not perform these additional checks and updates.

Parameters
`event_dataset` (`EventDataset`) – New event to be used for importance sampling.

`sample_synthetic_phase`(*synthetic_phase_kwargs*, *inverse: bool = False*)

Sample a synthetic phase for the waveform. This is a post-processing step applicable to samples theta in the full parameter space, except for the phase parameter (i.e., 14D samples). This step adds a phase parameter to the samples based on likelihood evaluations.

A synthetic phase is sampled as follows.

- Compute and cache the modes for the waveform mu(theta, phase=0) for phase 0, organize them such that each contribution m transforms as exp(-i * m * phase).
- Compute the likelihood on a phase grid, by computing mu(theta, phase) from the cached modes. In principle this likelihood is exact, however, it can deviate slightly from the likelihood computed without cached modes for various technical reasons (e.g., slightly different windowing of cached modes compared to full waveform when transforming TD waveform to FD). These small deviations can be fully accounted for by importance sampling. *Note*: when approximation_22_mode=True, the entire waveform is assumed to transform as exp(2i*phase), in which case the likelihood is only exact if the waveform is fully dominated by the (2, 2) mode.
- Build a synthetic conditional phase distribution based on this grid. We use an interpolated prior distribution bilby.core.prior.Interped, such that we can sample and also evaluate the log_prob. We add a constant background with weight self.synthetic_phase_kwargs to the kde to make sure that we keep a mass-covering property. With this, the importance sampling will yield exact results even when the synthetic phase conditional is just an approximation.

Besides adding phase samples to self.samples['phase'], this method also modifies self.samples['log_prob'] by adding the log_prob of the synthetic phase conditional.

This method modifies self.samples in place.

Parameters

- `synthetic_phase_kwargs` (*dict*) –

  This should consist of the kwargs
  approximation_22_mode (optional) num_processes (optional) n_grid uniform_weight (optional)

- `inverse` (*bool, default False*) – Whether to apply instead the inverse transformation. This is used prior to calculating the log_prob. In inverse mode, the posterior probability over phase is calculated for given samples. It is stored in self.samples['log_prob'].

`sampling_importance_resampling`(*num_samples=None*, *random_state=None*)

Generate unweighted posterior samples from weighted ones. New samples are sampled with probability proportional to the sample weight. Resampling is done with replacement, until the desired number of unweighted samples is obtained.

Parameters

- `num_samples` (*int*) – Number of samples to resample.
- `random_state` (*int or None*) – Sampling seed.

Returns
Unweighted samples

Return type
pd.Dataframe

split(*num_parts*)

Split the Result into a set of smaller results. The samples are evenly divided among the sub-results. Additional information (metadata, context, etc.) are copied into each.

This is useful for splitting expensive tasks such as importance sampling across multiple jobs.

Parameters
`num_parts` (*int*) – The number of parts to split the Result across.

Return type
list of sub-Results.

train_unconditional_flow(*parameters*, *nde_settings: dict*, *train_dir: str | None = None*, *threshold_std: float | None = inf*)

Train an unconditional flow to represent the distribution of self.samples.

Parameters

- `parameters` (*list*) – List of parameters over which to train the flow. Can be a subset of the existing parameters.
- `nde_settings` (*dict*) – Configuration settings for the neural density estimator.
- `train_dir` (*Optional[str]*) – Where to save the output of network training, e.g., logs, checkpoints. If not provide, a temporary directory is used.
- `threshold_std` (*Optional[float]*) – Drop samples more than threshold_std standard deviations away from the mean (in any parameter) before training the flow. This is meant to remove outlier samples.

Return type
PosteriorModel

update_prior(*prior_update*)

Update the prior based on a new dict of priors. Use the existing prior for parameters not included in the new dict.

If class samples have not been importance sampled, then save new sample weights based on the new prior. If class samples have been importance sampled, then update the weights.

Parameters
`prior_update` (*dict*) – Priors to update. This should be of the form {key : prior_str}, where str is a string that can instantiate a prior via PriorDict(prior_update). The prior_update is provided in this form so that it can be properly saved with the Result and later instantiated.

Following a sampler run, a `Result` can be obtained using `Sampler.to_result()`. Since `Result` inherits from `DingoDataset` it also possesses `to_file()` and `to_dictionary()` methods for saving samples and associated metadata (including context data, namely event data and ASDs).

18.1 Density recovery

When sampling with GNPE, there is no direct access to the probability density $q(\theta|d)$. This is because of the Gibbs iterations: one only has access to the probability density of the entire chain, not just the final samples. The probability density is, however, needed for importance sampling, since this is the proposal distribution.

The `Result` class contains methods to enable *recovery* of the probability density for a collection of samples. The approach is as follows:

1. Start from the samples $\{(\theta_i, \hat{\theta}_i)\}_{i=1}^N$ from the final Gibbs iteration, including parameters θ and proxy parameters $\hat{\theta}$. By default these are included in the `samples` attribute generated by the `Sampler`.
2. Train an *unconditional* density estimator $q(\hat{\theta})$ to model the proxy parameters. This is done by (1) using `parameter_subset()` to produce a new `Result` containing just the proxies, and (2) using `train_unconditional_flow()` on this subset.
3. Generate new samples $(\theta, \hat{\theta}) \sim q(\theta, \hat{\theta}|d) = q(\theta|d, \hat{\theta})q(\hat{\theta})$. This can be accomplished using `GNPESampler.sample()` with `num_iterations = 1` and setting the initial sampler to be the unconditional flow trained in the previous step. Since this does not involve multiple iterations, the density is obtained as well, so importance sampling can be performed.

Note: Density recovery can also be achieved using an unconditional density estimator for θ (trained on samples $\{\theta_i\}_{i=1}^N$ from GNPE). Since θ typically comprises 14 parameters (versus 2 or 3 for $\hat{\theta}$) it is usually more straightforward to learn the proxies.

18.2 Synthetic phase

It is often challenging for Dingo to learn to model the `phase` parameter ϕ_c. For this reason, we usually marginalize over it in training by excluding it from the list of `inference_parameters`. The phase is, however, required for importance sampling unless using also a phase-marginalized likelihood (which is approximate except under special circumstances).

The Dingo `gw.Result` class includes a method `sample_synthetic_phase()` which produces a ϕ_c sample from a ϕ_c-marginalized sample. It does so by evaluating the likelihood on a ϕ_c-grid and then sampling from the associated 1D distribution. The `log_prob` value for the sample is also corrected to reflect the sampled ϕ_c. Speed is ensured by caching waveform modes and evaluating the polarizations for different ϕ_c. For further details, see the Supplemental Material of [5].

This method should be run *after* recovering the density, since in particular it applies a correction to the density.

18.2.1 Configuration

The method `sample_synthetic_phase()` takes a kwargs argument. An example configuration is

```yaml
approximation_22_mode: false
n_grid: 5001
uniform_weight: 0.01
num_processes: 100
```

approximation_22_mode
Whether to make the approximation that only the $(l, m) = (2, 2)$ mode is present, i.e., waveforms transform as $\exp 2\pi i\phi_c$. This simplifies computations since it does not require caching of waveform modes.

n_grid
Specifies the phase grid on which the likelihoods are evaluated.

uniform_weight
Base probability level to add to ensure mass coverage.

num_processes
For parallelization of synthetic phase sampling. This is usually the most expensive part of importance sampling, so it is advantageous to perform calculations in parallel.

18.3 Importance sampling

Once samples are in the right form—including all relevant parameters *and* the log probability—importance sampling is carried out using the `importance_sample()` method. It allows to specify options for using a marginalized likelihood. (Time and phase marginalization are separately supported; see the documentation of `dingo.gw.likelihood.StationaryGaussianGWLikelihood`.)

As with the synthetic phase, importance sampling allows for parallelization.

18.4 Plotting

The plotting methods included here are intended for quick plots for evaluating results. They include

- **corner plots** comparing importance sampled and proposal results;
- **weights plots** to evaluate performance of importance sampling; and
- **log probability plots** comparing target and proposal log probability.

CHAPTER

NINETEEN

DINGO_PIPE

Dingo includes a command-line tool **dingo_pipe** for automating inference tasks. This is based *very closely* on the bilby_pipe package, with suitable modifications. The basic usage is to pass a `.ini` file containing event information and run configuration settings, e.g.,

```
dingo_pipe GW150914.ini
```

dingo_pipe then executes various commands for *preparing data*, *sampling from networks*, *importance sampling*, and *plotting*. It can execute commands locally or on a cluster using a DAG. This documentation will only describe the relevant differences compared to bilby_pipe, and we refer the reader to the bilby_pipe documentation for additional information.

Listing 1: Example `GW150914.ini` file. This is also available in the examples/ directory.

```
################################################################################
##  Job submission arguments
################################################################################

local = True
accounting = dingo
request-cpus-importance-sampling = 16
n-parallel = 4
sampling-requirements = [TARGET.CUDAGlobalMemoryMb>40000]
extra-lines = [+WantsGPUNode = True]
simple-submission = false

################################################################################
##  Sampler arguments
################################################################################

model-init = /data/sgreen/dingo-experiments/XPHM/O1_init/model_stage_1.pt
model = /data/sgreen/dingo-experiments/XPHM/testing_inference/model.pt
device = 'cuda'
num-gnpe-iterations = 30
num-samples = 50000
batch-size = 50000
recover-log-prob = true
importance-sample = true
prior-dict = {
luminosity_distance = bilby.gw.prior.UniformComovingVolume(minimum=100, maximum=2000,␣
↪name='luminosity_distance'),
```

(continues on next page)

(continued from previous page)

```
}

################################################################################
## Data generation arguments
################################################################################

trigger-time = GW150914
label = GW150914
outdir = outdir_GW150914
channel-dict = {H1:GWOSC, L1:GWOSC}
psd-length = 128
sampling-frequency = 2048.0
importance-sampling-updates = {'duration': 4.0}

################################################################################
## Calibration marginalization arguments
################################################################################

calibration-model = CubicSpline
spline-calibration-envelope-dict = {H1: GWTC1_GW150914_H_CalEnv.txt, L1: GWTC1_GW150914_
↪L_CalEnv.txt}
spline-calibration-nodes = 10
spline-calibration-curves = 1000

################################################################################
## Plotting arguments
################################################################################

plot-corner = true
plot-weights = true
plot-log-probs = true
```

The main difference compared to a bilby_pipe `.ini` file is that one specifies trained Dingo models rather than data conditioning and prior settings. The reason for this is that such settings have already been incorporated into training of the model. It is therefore not possible to change them when sampling from the Dingo model. Understandably, this could cause inconvenience if one is interested in a different prior or data conditioning settings. As a solution, Dingo enables the changing of such settings during importance sampling, which applies the new settings for likelihood evaluations.

Important: For dingo_pipe it is necessary to specify a trained Dingo model *instead* of sampler settings such as prior and data conditioning.

19.1 Data generation

The first step is to download and prepare gravitational-wave data. In the example, dingo_pipe (using bilby_pipe routines) downloads the event and PSD data at the time of GW150914. It then prepares the data based on conditioning settings in the specified Dingo model. If other conflicting conditioning settings are provided (e.g., `sampling_frequency = 2048.0`), dingo_pipe stores these in the dictionary `importance_sampling_updates` (which can also be specified explicitly). These settings are ignored for now, and only applied later for calculating the likelihood in importance sampling.

The prepared event data and ASD are stored in a *dingo.gw.data.event_dataset.EventDataset*, which is then saved to disk in HDF5 format.

Note: Dingo models are typically trained using Welch PSDs. For this reason we do not recommend using a BayesWave PSD for initial sampling. Rather, a BayesWave PSD should be specified within the `importance_sampling_updates` dictionary, so that it will be used during importance sampling.

19.2 Sampling

The next step is sampling from the Dingo model. The model is loaded into a *GWSampler* or *GWSamplerGNPE* object. (If using *GNPE* it is necessary to specify a `model-init`.) The Sampler `context` is then set from the EventDataset prepared in the previous step. `num-samples` samples are then generated in batches of size `batch-size`. The samples (and context) are stored in a *Result* object and saved in HDF5 format.

If using GNPE, one can optionally specify `num-gnpe-iterations` (it defaults to 30). Importantly, obtaining the log probability when using GNPE requires an *extra step of training an unconditional flow*. This is done using the `recover-log-prob` flag, which defaults to `True`. The default density recovery settings can be overwritten by providing a `density-recovery-settings` dictionary in the `.ini` file.

Since sampling uses GPU hardware, there is an additional key `sampling-requirements` for HTCondor requirements during the sampling stage. This is intended for specifying GPU requirements such as memory or CUDA version.

19.3 Importance sampling

For importance sampling, the Result saved in the previous step is loaded. Since this contains the strain data and ASDs, as well as all settings used for training the network, the likelihood and prior can be evaluated for each sample point. If it is necessary to change data conditioning or PSD for importance sampling (i.e., if the `importance-sampling-updates` dictionary is non-empty), then a second *data generation* step is first carried using the new settings, and used as importance sampling context. The importance sampled result is finally saved as HDF5, including the estimated Bayesian evidence.

If a `prior-dict` is specified in the `.ini` file, then this will be used for the importance sampling prior. One example where this is useful is for the luminosity distance prior. Indeed, Dingo tends to train better using a uniform prior over luminosity distance, but physically one would prefer a uniform in volume prior. By specifying a `prior-dict` this change can be made in importance sampling.

Caution: If extending the prior support during importance sampling, be sure that the posterior does not rail up against the prior boundary being extended.

By default, dingo_pipe assumes that it is necessary to sample the phase synthetically, so it will do so before importance sampling. This can be turned off by passing an empty dictionary to `importance-sampling-settings`. Note that importance sampling itself can be switched off by setting the `importance-sample` flag to False (it defaults to True).

Importance sampling (including synthetic phase sampling) is an expensive step, so dingo_pipe allows for parallelization: this step is split over `n-parallel` jobs, each of which uses `request-cpus-importance-sampling` processes. In the backend, this makes use of the Result *split()* and *merge()* methods.

19.3.1 Calibration marginalization

Settings related to calibration are used to **marginalize** over calibration uncertainty during importance sampling.

calibration-model
: None or "CubicSpline". If "CubicSpline", perform calibration marginalization using a cubic spline calibration model. If None do not perform calibration marginalization. (Default: None)

spline-calibration-envelope-dict
: Dictionary pointing to the spline calibration envelope files. This is required if calibration-model is "CubicSpline".

spline-calibration-nodes
: Number of calibration nodes. (Default: 10)

spline-calibration-curves
: Number of calibration curves to use for marginalization. (Default: 1000)

19.4 Plotting

The standard Result *plots* are turned on using the `plot-corner`, `plot-weights`, and `plot-log-probs` flags.

19.5 Additional options

extra-lines
: Additional lines for all submission scripts. This could be useful for particular cluster configurations.

simple-submission
: Strip the keys `accounting_tag`, `getenv`, `priority`, and `universe` from submission scripts. Again useful for particular cluster configurations.

CHAPTER

TWENTY

DINGO

20.1 dingo package

20.1.1 Subpackages

dingo.asimov package

Submodules

dingo.asimov.asimov module

Module contents

dingo.core package

Subpackages

dingo.core.density package

Submodules

dingo.core.density.interpolation module

dingo.core.density.interpolation.**interpolated_log_prob**(*sample_points*, *values*, *evaluation_point*)

Given a distribution discretized on a grid, return a sample and the log prob from an interpolated distribution. Wraps the bilby.core.prior.Interped class.

Parameters

- **sample_points** (*np.ndarray*) – x values for samples
- **values** (*np.ndarray*) – y values for samples. The distribution does not have to be initially normalized, although the final log_prob will be.
- **evaluation_point** (*float*) – x value at which to evaluate log_prob.

Returns
float

Return type
log_prob

`dingo.core.density.interpolation.`**`interpolated_log_prob_multi`**(*sample_points*, *values*, *evaluation_points*, *num_processes: int = 1*)

Given a distribution discretized on a grid, the log prob at a specific point using an interpolated distribution. Wraps the bilby.core.prior.Interped class. Works with multiprocessing.

Parameters

- **sample_points** (*np.ndarray, shape (N)*) – x values for samples
- **values** (*np.ndarray, shape (B, N)*) – y values for samples. The distributions do not have to be initially normalized, although the final log_probs will be. B = batch dimension.
- **evaluation_points** (*np.ndarray, shape (B)*) – x values at which to evaluate log_prob.
- **num_processes** (*int*) – Number of parallel processes to use.

Returns
(np.ndarray, np.ndarray)

Return type
sample and log_prob arrays, each of length B

`dingo.core.density.interpolation.`**`interpolated_sample_and_log_prob`**(*sample_points*, *values*)

Given a distribution discretized on a grid, return a sample and the log prob from an interpolated distribution. Wraps the bilby.core.prior.Interped class.

Parameters

- **sample_points** (*np.ndarray*) – x values for samples
- **values** (*np.ndarray*) – y values for samples. The distribution does not have to be initially normalized, although the final log_prob will be.

Returns
(float, float)

Return type
sample and log_prob

`dingo.core.density.interpolation.`**`interpolated_sample_and_log_prob_multi`**(*sample_points*, *values*, *num_processes: int = 1*)

Given a distribution discretized on a grid, return a sample and the log prob from an interpolated distribution. Wraps the bilby.core.prior.Interped class. Works with multiprocessing.

Parameters

- **sample_points** (*np.ndarray, shape (N)*) – x values for samples
- **values** (*np.ndarray, shape (B, N)*) – y values for samples. The distributions do not have to be initially normalized, although the final log_probs will be. B = batch dimension.
- **num_processes** (*int*) – Number of parallel processes to use.

Returns
(np.ndarray, np.ndarray)

Return type
sample and log_prob arrays, each of length B

dingo.core.density.nde_settings module

Default settings for unconditional density estimation

`dingo.core.density.nde_settings.`**`get_default_nde_settings_3d`**(*device='cpu'*, *num_workers=0*, *inference_parameters=None*)

dingo.core.density.unconditional_density_estimation module

class `dingo.core.density.unconditional_density_estimation.`**`SampleDataset`**(*data*)

Bases: `Dataset`

Dataset class for unconditional density estimation. This is required, since the training method of dingo.core.models.PosteriorModel expects a tuple of (theta, *context) as output of the DataLoader, but here we have no context, so len(context) = 0. This SampleDataset therefore returns a tuple (theta,) instead of just theta.

`dingo.core.density.unconditional_density_estimation.`**`parse_args`**()

`dingo.core.density.unconditional_density_estimation.`**`train_unconditional_density_estimator`**(*result*, *settings: dict*, *train_dir: str*)

Train unconditional density estimator for a given set of samples.

Parameters

- **samples** (*pd.DataFrame*) – DataFrame containing the samples to train the density estimator on.
- **settings** (*dict*) – Dictionary containing the settings for the density estimator.
- **train_dir** (*str*) – Path to the directory where the trained model should be saved.

Returns

model – trained density estimator

Return type

PosteriorModel

Module contents

This submodule contains tools for density estimation from samples. This is required for instance to recover the posterior density from GNPE samples, since the density is intractable with GNPE.

dingo.core.models package

Submodules

dingo.core.models.posterior_model module

TODO: Docstring

`class dingo.core.models.posterior_model.PosteriorModel(model_filename: str | None = None, metadata: dict | None = None, initial_weights: dict | None = None, device: str = 'cuda', load_training_info: bool = True)`

Bases: `object`

TODO: Docstring

initialize_model:
initialize the NDE (including embedding net) as posterior model

initialize_training:
initialize for training, that includes storing the epoch, building an optimizer and a learning rate scheduler

save_model:
save the model, including all information required to rebuild it, except for the builder function

load_model:
load and build a model from a file

train_model:
train the model

inference:
perform inference

Parameters

- **model_builder** (*Callable*) – builder function for the model, self.model = model_builder(**model_kwargs)
- **model_kwargs** (*dict = None*) – kwargs for for the model, self.model = model_builder(**model_kwargs)
- **model_filename** (*str = None*) – path to filename of loaded model
- **optimizer_kwargs** (*dict = None*) – kwargs for optimizer
- **scheduler_kwargs** (*dict = None*) – kwargs for scheduler
- **init_for_training** (*bool = False*) – flag whether initialization for training (e.g., optimizer) required
- **metadata** (*dict = None*) – dict with metadata, used to save dataset_settings and train_settings

initialize_model()

Initialize a model for the posterior by calling the self.model_builder with self.model_kwargs.

initialize_optimizer_and_scheduler()

Initializes the optimizer and scheduler with self.optimizer_kwargs and self.scheduler_kwargs, respectively.

load_model(*model_filename: str*, *load_training_info: bool = True*, *device: str = 'cuda'*)

Load a posterior model from the disk.

Parameters

- **model_filename** (*str*) – path to saved model
- **load_training_info** (*bool #TODO: load information for training*) – specifies whether information required to proceed with training is loaded, e.g. optimizer state dict

model_to_device(*device*)

Put model to device, and set self.device accordingly.

sample(**x*, *batch_size=None*, *get_log_prob=False*)

Sample from posterior model, conditioned on context x. x is expected to have a batch dimension, i.e., to obtain N samples with additional context requires x = **x_**.expand(N, ***x_**.shape).

This method takes care of the batching, makes sure that self.model is in evaluation mode and disables gradient computation.

Parameters

- ***x** – input context to the neural network; has potentially multiple elements for, e.g., gnpe proxies
- **batch_size** (*int = None*) – batch size for sampling
- **get_log_prob** (*bool = False*) – if True, also return log probability along with the samples

Returns
samples – samples from posterior model

Return type
torch.Tensor

save_model(*model_filename: str*, *save_training_info: bool = True*)

Save the posterior model to the disk.

Parameters

- **model_filename** (*str*) – filename for saving the model
- **save_training_info** (*bool*) – specifies whether information required to proceed with training is saved, e.g. optimizer state dict

train(*train_loader: DataLoader*, *test_loader: DataLoader*, *train_dir: str*, *runtime_limits: object | None = None*, *checkpoint_epochs: int | None = None*, *use_wandb=False*, *test_only=False*)

Parameters

- **train_loader** –
- **test_loader** –
- **train_dir** –
- **runtime_limits** –
- **checkpoint_epochs** –

- `use_wandb` –
- `test_only` (*bool = False*) – if True, training is skipped

`dingo.core.models.posterior_model.`**`get_model_callable`**(*model_type: str*)

`dingo.core.models.posterior_model.`**`test_epoch`**(*pm*, *dataloader*)

`dingo.core.models.posterior_model.`**`train_epoch`**(*pm*, *dataloader*)

Module contents

dingo.core.nn package

Submodules

dingo.core.nn.enets module

Implementation of embedding networks.

class `dingo.core.nn.enets.`**`DenseResidualNet`**(*input_dim: int*, *output_dim: int*, *hidden_dims: ~typing.Tuple*, *activation: ~typing.Callable = <function elu>*, *dropout: float = 0.0*, *batch_norm: bool = True*)

Bases: `Module`

A nn.Module consisting of a sequence of dense residual blocks. This is used to embed high dimensional input to a compressed output. Linear resizing layers are used for resizing the input and output to match the first and last hidden dimension, respectively.

Module specs

input dimension: (batch_size, input_dim) output dimension: (batch_size, output_dim)

param input_dim
dimension of the input to this module

type input_dim
int

param output_dim
output dimension of this module

type output_dim
int

param hidden_dims
tuple with dimensions of hidden layers of this module

type hidden_dims
tuple

param activation
activation function used in residual blocks

type activation
callable

param dropout
dropout probability for residual blocks used for reqularization

type dropout
float

param batch_norm
flag that specifies whether to use batch normalization

type batch_norm
bool

`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 dingo.core.nn.enets.LinearProjectionRB`(*input_dims: List[int]*, *n_rb: int*, *V_rb_list: Tuple* | *None*)

Bases: `Module`

A compression layer that reduces the input dimensionality via projection onto a reduced basis. The input data is of shape (batch_size, num_blocks, num_channels, num_bins). Each of the num_blocks blocks (for GW use case: block=detector) is treated independently.

A single block consists of 1D data with num_bins bins (e.g. GW use case: num_bins=number of frequency bins). It has num_channels>=2 different channels, channel 0 and 1 store the real and imaginary part of the signal. Channels with index >=2 are used for auxiliary signals (such as PSD for GW use case).

This layer compresses the complex signal in channels 0 and 1 to n_rb reduced-basis (rb) components. This is achieved by initializing the weights of this layer with the rb matrix V, such that the (2*n_rb) dimensional output of each block is the concatenation of the real and imaginary part of the reduced basis projection of the complex signal in channel 0 and 1. The projection of the auxiliary channels with index >=2 onto these components is initialized with 0.

Module specs

input dimension: (batch_size, num_blocks, num_channels, num_bins) output dimension: (batch_size, 2 * n_rb * num_blocks)

param input_dims
dimensions of input batch, omitting batch dimension input_dims = [num_blocks, num_channels, num_bins]

type input_dims
list

param n_rb
number of reduced basis elements used for projection the output dimension of the layer is 2 * n_rb * num_blocks

type n_rb
int

param V_rb_list
tuple with V matrices of the reduced basis SVD projection, convention for SVD matrix decomposition: U @ s @ V^h; if None, layer is not initialized with reduced basis projection, this is useful when loading a saved model

type V_rb_list
tuple of np.arrays, or None

`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.

`init_layers(V_rb_list)`

Loop through layers and initialize them individually with the corresponding rb projection. V_rb_list is a list that contains the rb matrix V for each block. Each matrix V in V_rb_list is represented with a numpy array of shape (self.num_bins, num_el), where num_el >= self.n_rb.

`property input_dim`

`property output_dim`

`test_dimensions(V_rb_list)`

Test if input dimensions to this layer are consistent with each other, and the reduced basis matrices V.

`class dingo.core.nn.enets.ModuleMerger(module_list: Tuple)`

Bases: `Module`

This is a wrapper used to process multiple different kinds of context information collected in x = (x_0, x_1, …). For each kind of context information x_i, an individual embedding network is provided in enets = (enet_0, enet_1, …). The embedded output of the forward method is the concatenation of the individual embeddings enet_i(x_i).

In the GW use case, this wrapper can be used to embed the high-dimensional signal input into a lower dimensional feature vector with a large embedding network, while applying an identity embedding to the time shifts.

Module specs

input dimension: (batch_size, …), (batch_size, …), … output dimension: (batch_size, ?)

param module_list
nn.Modules for embedding networks, use torch.nn.Identity for identity mappings

type module_list
tuple

`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.

`dingo.core.nn.enets.`**`create_enet_with_projection_layer_and_dense_resnet`**(*input_dims: List[int]*, *V_rb_list: Tuple | None*, *output_dim: int*, *hidden_dims: Tuple*, *svd: dict*, *activation: str = 'elu'*, *dropout: float = 0.0*, *batch_norm: bool = True*, *added_context: bool = False*)

Builder function for 2-stage embedding network for 1D data with multiple blocks and channels. Module 1 is a linear layer initialized as the projection of the complex signal onto reduced basis components via the LinearProjectionRB, where the blocks are kept separate. See docstring of LinearProjectionRB for details. Module 2 is a sequence of dense residual layers, that is used to further reduce the dimensionality.

The projection requires the complex signal to be represented via the real part in channel 0 and the imaginary part in channel 1. Auxiliary signals may be contained in channels with indices => 2. In GW use case a block corresponds to a detector and channel 2 is used for ASD information.

If added_context = True, the 2-stage embedding network described above is merged with an identity mapping via ModuleMerger. Then, the expected input is not x with x.shape = (batch_size, num_blocks, num_channels, num_bins), but rather the tuple *(x, z), where z is additional context information. The output of the full module is then the concatenation of enet(x) and z. In GW use case, this is used to concatenate the applied time shifts z to the embedded feature vector of the strain data enet(x).

Module specs

For added_context == False:
input dimension: (batch_size, num_blocks, num_channels, num_bins) output dimension: (batch_size, output_dim)

For added_context == True:

input dimension: (batch_size, num_blocks, num_channels, num_bins),
(batch_size, N)

output dimension: (batch_size, output_dim + N)

param input_dims
list dimensions of input batch, omitting batch dimension input_dims = (num_blocks, num_channels, num_bins)

param n_rb
int number of reduced basis elements used for projection the output dimension of the layer is 2 * n_rb * num_blocks

param V_rb_list
tuple of np.arrays, or None tuple with V matrices of the reduced basis SVD projection, convention for SVD matrix decomposition: U @ s @ V^h; if None, layer is not initialized with reduced basis projection, this is useful when loading a saved model

param output_dim
int output dimension of the full module

param hidden_dims
tuple tuple with dimensions of hidden layers of module 2

param activation
str str that specifies activation function used in residual blocks

param dropout
float dropout probability for residual blocks used for reqularization

param batch_norm
bool flag that specifies whether to use batch normalization

param added_context
bool if set to True, additional context z is concatenated to the embedded feature vector enet(x); note that in this case, the expected input is a tuple with 2 elements, input = (x, z) rather than just the tensor x.

return
nn.Module

dingo.core.nn.nsf module

Implementation of the neural spline flow (NSF). Most of this code is adapted from the uci.py example from https://github.com/bayesiains/nsf.

class `dingo.core.nn.nsf.`**`FlowWrapper`**(*flow: Flow*, *embedding_net: Module | None = None*)

Bases: `Module`

This class wraps the neural spline flow. It is required for multiple reasons. (i) some embedding networks take tuples as input, which is not supported by the nflows package. (ii) paralellization across multiple GPUs requires a forward method, but the relevant flow method for training is log_prob.

Parameters

- **`flow`** – flows.base.Flow
- **`embedding_net`** – nn.Module

`forward`(*y*, **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.

`log_prob`(*y*, **x*)

`sample`(**x*, *num_samples=1*)

`sample_and_log_prob`(**x*, *num_samples=1*) dingo.core.nn.nsf.**autocomplete_model_kwargs_nsf**(*model_kwargs*, *data_sample*)

Autocomplete the model kwargs from train_settings and data_sample from the dataloader: (*) set input dimension of embedding net to shape of data_sample[1] (*) set dimension of nsf parameter space to len(data_sample[0]) (*) set added_context flag of embedding net if required for gnpe proxies (*) set context dim of nsf to output dim of embedding net + gnpe proxy dim

Parameters

- **train_settings** – dict train settings as loaded from .yaml file
- **data_sample** – list Sample from dataloader (e.g., wfd[0]) used for autocomplection. Should be of format [parameters, GW data, gnpe_proxies], where the last element is only there is gnpe proxies are required.

Returns

model_kwargs: dict updated, autocompleted model_kwargs

dingo.core.nn.nsf.**create_base_transform**(*i: int*, *param_dim: int*, *context_dim: int | None = None*, *hidden_dim: int = 512*, *num_transform_blocks: int = 2*, *activation: str = 'relu'*, *dropout_probability: float = 0.0*, *batch_norm: bool = False*, *num_bins: int = 8*, *tail_bound: float = 1.0*, *apply_unconditional_transform: bool = False*, *base_transform_type: str = 'rq-coupling'*)

Build a base NSF transform of y, conditioned on x.

This uses the PiecewiseRationalQuadraticCoupling transform or the MaskedPiecewiseRationalQuadraticAutoregressiveTransform, as described in the Neural Spline Flow paper (https://arxiv.org/abs/1906.04032).

Code is adapted from the uci.py example from https://github.com/bayesiains/nsf.

A coupling flow fixes half the components of y, and applies a transform to the remaining components, conditioned on the fixed components. This is a restricted form of an autoregressive transform, with a single split into fixed/transformed components.

The transform here is a neural spline flow, where the flow is parametrized by a residual neural network that depends on y_fixed and x. The residual network consists of a sequence of two-layer fully-connected blocks.

Parameters

- **i** – int index of transform in sequence
- **param_dim** – int dimensionality of y
- **context_dim** – int = None dimensionality of x
- **hidden_dim** – int = 512 number of hidden units per layer
- **num_transform_blocks** – int = 2 number of transform blocks comprising the transform
- **activation** – str = 'relu' activation function
- **dropout_probability** – float = 0.0 dropout probability for regularization
- **batch_norm** – bool = False whether to use batch normalization
- **num_bins** – int = 8 number of bins for the spline
- **tail_bound** – float = 1.
- **apply_unconditional_transform** – bool = False whether to apply an unconditional transform to fixed components
- **base_transform_type** – str = 'rq-coupling' type of base transform, one of {rq-coupling, rq-autoregressive}

Returns
Transform the NSF transform

dingo.core.nn.nsf.**create_linear_transform**(*param_dim: int*)

Create the composite linear transform PLU.

Parameters
param_dim – int dimension of the parameter space

Returns
nde.Transform the linear transform PLU

dingo.core.nn.nsf.**create_nsf_model**(*input_dim: int*, *context_dim: int*, *num_flow_steps: int*, *base_transform_kwargs: dict*, *embedding_net_builder: Callable | str | None = None*, *embedding_net_kwargs: dict | None = None*)

Build NSF model. This models the posterior distribution p(y|x).

The model consists of

- a base distribution (StandardNormal, dim(y))
- a sequence of transforms, each conditioned on x

Parameters

- **input_dim** – int, dimensionality of y
- **context_dim** – int, dimensionality of the (embedded) context
- **num_flow_steps** – int, number of sequential transforms
- **base_transform_kwargs** – dict, hyperparameters for transform steps
- **embedding_net_builder** – Callable=None, build function for embedding network TODO
- **embedding_net_kwargs** – dict=None, hyperparameters for embedding network

Returns
Flow the NSF (posterior model)

dingo.core.nn.nsf.**create_nsf_with_rb_projection_embedding_net**(*nsf_kwargs: dict*, *embedding_net_kwargs: dict*, *initial_weights: dict | None = None*)

Builds a neural spline flow with an embedding network that consists of a reduced basis projection followed by a residual network. Optionally initializes the embedding network weights.

Parameters

- **nsf_kwargs** (*dict*) – kwargs for neural spline flow
- **embedding_net_kwargs** (*dict*) – kwargs for emebedding network
- **initial_weights** (*dict*) – Dictionary containing the initial weights for the SVD projection. This should have one key 'V_rb_list', with value a list of SVD V matrices (one for each detector).

Returns
Neural spline flow model

Return type
nn.Module

dingo.core.nn.nsf.**create_nsf_wrapped**(***kwargs*)

Wraps the NSF model in a FlowWrapper. This is required for parallel training, and wraps the log_prob method as a forward method.

dingo.core.nn.nsf.**create_transform**(*num_flow_steps: int*, *param_dim: int*, *context_dim: int*, *base_transform_kwargs: dict*)

Build a sequence of NSF transforms, which maps parameters y into the base distribution u (noise). Transforms are conditioned on context data x.

Note that the forward map is f^{-1}(y, x).

Each step in the sequence consists of

- A linear transform of y, which in particular permutes components
- A NSF transform of y, conditioned on x.

There is one final linear transform at the end.

Parameters

- **num_flow_steps** – int, number of transforms in sequence
- **param_dim** – int, dimensionality of parameter space (y)
- **context_dim** – int, dimensionality of context (x)
- **base_transform_kwargs** – int hyperparameters for NSF step

Returns

Transform the NSF transform sequence

Module contents

dingo.core.utils package

Submodules

dingo.core.utils.condor_utils module

dingo.core.utils.condor_utils.**copy_logfiles**(*log_dir*, *epoch*, *name='info'*, *suffixes=('.err', '.log', '.out')*)

dingo.core.utils.condor_utils.**copyfile**(*src*, *dst*)

dingo.core.utils.condor_utils.**create_submission_file**(*train_dir*, *filename='submission_file.sub'*)

TODO: documentation :param train_dir: :param filename: :return:

dingo.core.utils.condor_utils.**create_submission_file_and_submit_job**(*train_dir*, *filename='submission_file.sub'*)

TODO: documentation :param train_dir: :param filename: :return:

dingo.core.utils.condor_utils.**resubmit_condor_job**(*train_dir*, *train_settings*, *epoch*)

TODO: documentation :param train_dir: :param train_settings: :param epoch: :return:

dingo.core.utils.gnpeutils module

class dingo.core.utils.gnpeutils.**IterationTracker**(*data=None*, *store_data=False*)

Bases: object

property pvalue_min

update(*new_data*)

Append new_data to self.data.

Parameters

new_data (*dict*) – dict with numpy arrays to append to data

dingo.core.utils.logging_utils module

dingo.core.utils.logging_utils.**check_directory_exists_and_if_not_mkdir**(*directory*, *logger*)

Checks if the given directory exists and creates it if it does not exist

Parameters

- **directory** (*str*) – Name of the directory
- **bilby-pipe** (*Borrowed from*) –

dingo.core.utils.logging_utils.**setup_logger**(*outdir=None*, *label=None*, *log_level='INFO'*)

Setup logging output: call at the start of the script to use

Parameters

- **outdir** (*str*) – If supplied, write the logging output to outdir/label.log
- **label** (*str*) – If supplied, write the logging output to outdir/label.log
- **log_level** (*str, optional*) – ['debug', 'info', 'warning'] Either a string from the list above, or an integer as specified in https://docs.python.org/2/library/logging.html#logging-levels
- **bilby-pipe** (*Borrowed from*) –

dingo.core.utils.misc module

dingo.core.utils.misc.**get_version**()

dingo.core.utils.misc.**recursive_check_dicts_are_equal**(*dict_a*, *dict_b*)

dingo.core.utils.plotting module

dingo.core.utils.plotting.**plot_corner_multi**(*samples*, *weights=None*, *labels=None*, *filename='corner.pdf'*, ***kwargs*)

Generate a corner plot for multiple posteriors.

Parameters

- **samples** (*list[pd.DataFrame]*) – List of sample sets. The DataFrame column names are used as parameter labels.

- **weights** (*list[np.ndarray or None] or None*) – List of weights sets. The length of each array should be the same as the length of the corresponding samples.
- **labels** (*list[str or None] or None*) – Labels for the posteriors.
- **filename** (*str*) – Where to save samples.
- ****kwargs** – Forwarded to corner.corner.

dingo.core.utils.pt_to_hdf5 module

dingo.core.utils.pt_to_hdf5.**main**()

dingo.core.utils.pt_to_hdf5.**parse_args**()

dingo.core.utils.torchutils module

dingo.core.utils.torchutils.**build_train_and_test_loaders**(*dataset: Dataset*, *train_fraction: float*, *batch_size: int*, *num_workers: int*)

Split the dataset into train and test sets, and build corresponding DataLoaders. The random split uses a fixed seed for reproducibility.

Parameters

- **dataset** (*torch.utils.data.Dataset*) –
- **train_fraction** (*float*) – Fraction of dataset to use for training. The remainder is used for testing. Should lie between 0 and 1.
- **batch_size** (*int*) –
- **num_workers** (*int*) –

Return type

(train_loader, test_loader)

dingo.core.utils.torchutils.**fix_random_seeds**(_)

Utility function to set random seeds when using multiple workers for DataLoader.

dingo.core.utils.torchutils.**forward_pass_with_unpacked_tuple**(*model: Module*, *x: Tuple | Tensor*)

Performs forward pass of model with input x. If x is a tuple, it return y = model(*x), else it returns y = model(x). :param model: nn.Module

model for forward pass

Parameters

x – Union[Tuple, torch.Tensor] input for forward pass

Returns

torch.Tensor output of the forward pass, either model(*x) or model(x)

dingo.core.utils.torchutils.**get_activation_function_from_string**(*activation_name: str*)

Returns an activation function, based on the name provided.

Parameters

activation_name – str name of the activation function, one of {'elu', 'relu', 'leaky_rely'}

Returns

function corresponding activation function `dingo.core.utils.torchutils.`**`get_lr`**`(`*optimizer*`)`

Returns a list with the learning rates of the optimizer.

`dingo.core.utils.torchutils.`**`get_number_of_model_parameters`**`(`*model: Module*, *requires_grad_flags: tuple = (True, False)*`)`

Counts parameters of the module. The list requires_grad_flag can be used to specify whether all parameters should be counted, or only those with requires_grad = True or False. :param model: nn.Module

model

Parameters

`requires_grad_flags` – tuple tuple of bools, for requested requires_grad flags

Returns

number of parameters of the model with requested required_grad flags

`dingo.core.utils.torchutils.`**`get_optimizer_from_kwargs`**`(`*model_parameters: Iterable*, ***optimizer_kwargs*`)`

Builds and returns an optimizer for model_parameters. The type of the optimizer is determined by kwarg type, the remaining kwargs are passed to the optimizer.

Parameters

- **`model_parameters`** (*`Iterable`*) – iterable of parameters to optimize or dicts defining parameter groups
- **`optimizer_kwargs`** – kwargs for optimizer; type needs to be one of [adagrad, adam, adamw, lbfgs, RMSprop, sgd], the remaining kwargs are used for specific optimizer kwargs, such as learning rate and momentum

Return type

optimizer

`dingo.core.utils.torchutils.`**`get_scheduler_from_kwargs`**`(`*optimizer: Optimizer*, ***scheduler_kwargs*`)`

Builds and returns an scheduler for optimizer. The type of the scheduler is determined by kwarg type, the remaining kwargs are passed to the scheduler.

Parameters

- **`optimizer`** (*`torch.optim.optimizer.Optimizer`*) – optimizer for which the scheduler is used
- **`scheduler_kwargs`** – kwargs for scheduler; type needs to be one of [step, cosine, reduce_on_plateau], the remaining kwargs are used for specific scheduler kwargs, such as learning rate and momentum

Return type

scheduler

`dingo.core.utils.torchutils.`**`perform_scheduler_step`**`(`*scheduler*, *loss=None*`)`

Wrapper for scheduler.step(). If scheduler is ReduceLROnPlateau, then scheduler.step(loss) is called, if not, scheduler.step().

Parameters

- **`scheduler`** – scheduler for learning rate
- **`loss`** – validation loss

dingo.core.utils.torchutils.**set_requires_grad_flag**(*model*, *name_startswith=None*, *name_contains=None*, *requires_grad=True*)

Set param.requires_grad of all model parameters with a name starting with name_startswith, or name containing name_contains, to requires_grad.

dingo.core.utils.torchutils.**split_dataset_into_train_and_test**(*dataset*, *train_fraction*)

Splits dataset into a trainset of size int(train_fraction * len(dataset)), and a testset with the remainder. Uses fixed random seed for reproducibility.

Parameters

- **dataset** (*torch.utils.data.Datset*) – dataset to be split
- **train_fraction** (*float*) – fraction of the dataset to be used for trainset

Return type

trainset, testset

dingo.core.utils.torchutils.**torch_detach_to_cpu**(*x*)

dingo.core.utils.trainutils module

class dingo.core.utils.trainutils.**AvgTracker**

Bases: object

get_avg()

update(*x*, *n=1*)

class dingo.core.utils.trainutils.**LossInfo**(*epoch*, *len_dataset*, *batch_size*, *mode='Train'*, *print_freq=1*)

Bases: object

get_avg()

print_info(*batch_idx*)

update(*loss*, *n*)

update_timer(*timer_mode='Dataloader'*)

class dingo.core.utils.trainutils.**RuntimeLimits**(*max_time_per_run: float | None = None*, *max_epochs_per_run: int | None = None*, *max_epochs_total: int | None = None*, *epoch_start: int | None = None*)

Bases: object

Keeps track of the runtime limits (time limit, epoch limit, max. number of epochs for model).

Parameters

- **max_time_per_run** (*float = None*) – maximum time for run, in seconds [soft limit, break only after full epoch]
- **max_epochs_per_run** (*int = None*) – maximum number of epochs for run
- **max_epochs_total** (*int = None*) – maximum total number of epochs for model
- **epoch_start** (*int = None*) – start epoch of run

limits_exceeded(*epoch: int | None = None*)

Check whether any of the runtime limits are exceeded.

Parameters
epoch (*int = None*) –

Returns
limits_exceeded – flag whether runtime limits are exceeded and run should be stopped; if limits_exceeded = True, this prints a message for the reason

Return type
bool

local_limits_exceeded(*epoch: int | None = None*)

Check whether any of the local runtime limits are exceeded. Local runtime limits include max_epochs_per_run and max_time_per_run, but not max_epochs_total.

Parameters
epoch (*int = None*) –

Returns
limits_exceeded – flag whether local runtime limits are exceeded

Return type
bool

dingo.core.utils.trainutils.**copyfile**(*src*, *dst*)

copy src to dst. :param src: :param dst: :return:

dingo.core.utils.trainutils.**save_model**(*pm*, *log_dir*, *model_prefix='model'*, *checkpoint_epochs=None*)

Save model to <model_prefix>_latest.pt in log_dir. Additionally, all checkpoint_epochs a permanent checkpoint is saved.

Parameters

- **pm** – model to be saved
- **log_dir** (*str*) – log directory, where model is saved
- **model_prefix** (*str = 'model'*) – prefix for name of save model
- **checkpoint_epochs** (*int = None*) – number of steps between two consecutive model checkpoints

dingo.core.utils.trainutils.**write_history**(*log_dir*, *epoch*, *train_loss*, *test_loss*, *learning_rates*, *aux=None*, *filename='history.txt'*)

Writes losses and learning rate history to csv file.

Parameters

- **log_dir** (*str*) – directory containing the history file
- **epoch** (*int*) – epoch
- **train_loss** (*float*) – train_loss of epoch
- **test_loss** (*float*) – test_loss of epoch
- **learning_rates** (*list*) – list of learning rates in epoch
- **aux** (*list = []*) – list of auxiliary information to be logged
- **filename** (*str = 'history.txt'*) – name of history file

Module contents

Submodules

dingo.core.dataset module

class dingo.core.dataset.**DingoDataset**(*file_name=None*, *dictionary=None*, *data_keys=None*)

Bases: object

This is a generic dataset class with save / load methods.

A common use case is to inherit multiply from DingoDataset and torch.utils.data.Dataset, in which case the subclass picks up these I/O methods, and DingoDataset is acting as a Mixin class.

Alternatively, if the torch Dataset is not needed, then DingoDataset can be subclassed directly.

For constructing, provide either file_name, or dictionary containing data and settings entries, or neither.

Parameters

- **file_name** (*str*) – HDF5 file containing a dataset
- **dictionary** (*dict*) – Contains settings and data entries. The data keys should be the same as save_keys
- **data_keys** (*list*) – Variables that should be saved / loaded. This allows for class to store additional variables beyond those that are saved. Typically, this list would be provided by any subclass.

dataset_type = 'dingo_dataset'

from_dictionary(*dictionary: dict*)

from_file(*file_name*)

to_dictionary()

to_file(*file_name*, *mode='w'*)

dingo.core.dataset.**recursive_hdf5_load**(*group*, *keys=None*)

dingo.core.dataset.**recursive_hdf5_save**(*group*, *d*)

dingo.core.likelihood module

class dingo.core.likelihood.**Likelihood**

Bases: object

log_likelihood(*theta*)

log_likelihood_multi(*theta: DataFrame*, *num_processes: int = 1*) → ndarray

Calculate the log likelihood at multiple points in parameter space. Works with multiprocessing.

This wraps the log_likelihood() method.

Parameters

- **theta** (*pd.DataFrame*) – Parameters values at which to evaluate likelihood.
- **num_processes** (*int*) – Number of processes to use.

Return type
np.array of log likelihoods

dingo.core.multiprocessing module

dingo.core.multiprocessing.**apply_func_with_multiprocessing**(*func: callable*, *theta: DataFrame*, *num_processes: int = 1*) → ndarray

Call func(theta.iloc[idx].to_dict()) with multiprocessing.

Parameters

- **func** (*callable*) –
- **theta** (*pd.DataFrame*) – Parameters with multiple rows, evaluate func for each row.
- **num_processes** (*int*) – Number of parallel processes to use.

Returns
result – Output array, where result[idx] = func(theta.iloc[idx].to_dict())

Return type
np.ndarray

dingo.core.result module

class dingo.core.result.**Result**(*file_name=None*, *dictionary=None*)

Bases: *DingoDataset*

A dataset class to hold a collection of samples, implementing I/O, importance sampling, and unconditional flow training.

Attributes:

samples
[pd.Dataframe] Contains parameter samples, as well as (possibly) log_prob, log_likelihood, weights, log_prior, delta_log_prob_target.

domain
[Domain] Should be implemented in a subclass.

prior
[PriorDict] Should be implemented in a subclass.

likelihood
[Likelihood] Should be implemented in a subclass.

context
[dict] Context data from which the samples were produced (e.g., strain data, ASDs).

metadata : dict event_metadata : dict log_evidence : float log_evidence_std : float (property) effective_sample_size, n_eff : float (property) sample_efficiency : float (property)

For constructing, provide either file_name, or dictionary containing data and settings entries, or neither.

Parameters

- **file_name** (*str*) – HDF5 file containing a dataset
- **dictionary** (*dict*) – Contains settings and data entries. The data keys should be the same as save_keys

- **data_keys** (*list*) – Variables that should be saved / loaded. This allows for class to store additional variables beyond those that are saved. Typically, this list would be provided by any subclass.

property base_metadata

property constraint_parameter_keys

dataset_type = 'core_result'

property effective_sample_size

property fixed_parameter_keys

importance_sample(*num_processes: int = 1*, ***likelihood_kwargs*)

Calculate importance weights for samples.

Importance sampling starts with samples have been generated from a proposal distribution q(theta), in this case a neural network model. Certain networks (i.e., non-GNPE) also provide the log probability of each sample, which is required for importance sampling.

Given the proposal, we re-weight samples according to the (un-normalized) target distribution, which we take to be the likelihood L(theta) times the prior pi(theta). This gives sample weights

w(theta) ~ pi(theta) L(theta) / q(theta),

where the overall normalization does not matter (and we take to have mean 1). Since q(theta) enters this expression, importance sampling is only possible when we know the log probability of each sample.

As byproducts, this method also estimates the evidence and effective sample size of the importance sampled points.

This method modifies the samples pd.DataFrame in-place, adding new columns for log_likelihood, log_prior, and weights. It also stores the log_evidence as an attribute.

Parameters

- **num_processes** (*int*) – Number of parallel processes to use when calculating likelihoods. (This is the most expensive task.)
- **likelihood_kwargs** (*dict*) – kwargs that are forwarded to the likelihood constructor. E.g., options for marginalization.

property injection_parameters

property log_bayes_factor

property log_evidence_std

classmethod merge(*parts*)

Merge several Result instances into one. Check that they are compatible, in the sense of having the same metadata. Finally, calculate a new log evidence for the combined result.

This is useful when recombining separate importance sampling jobs.

Parameters

parts (*list[Result]*) – List of sub-Results to be combined.

Return type

Combined Result.

property metadata

property n_eff

property num_samples

parameter_subset(*parameters*)

Return a new object of the same type, with only a subset of parameters. Drops all other columns in samples DataFrame as well (e.g., log_prob, weights).

Parameters
parameters (*list*) – List of parameters to keep.

Return type
Result

plot_corner(*parameters=None*, *filename='corner.pdf'*)

Generate a corner plot of the samples.

Parameters

- **parameters** (*list[str]*) – List of parameters to include. If None, include all parameters. (Default: None)
- **filename** (*str*) – Where to save samples.

plot_log_probs(*filename='log_probs.png'*)

Make a scatter plot of the target versus proposal log probabilities. For the target, subtract off the log evidence.

plot_weights(*filename='weights.png'*)

Make a scatter plot of samples weights vs log proposal.

print_summary()

Display the number of samples, and (if importance sampling is complete) the log evidence and number of effective samples.

reset_event(*event_dataset*)

Set the Result context and event_metadata based on an EventDataset.

If these attributes already exist, perform a comparison to check for changes. Update relevant objects appropriately. Note that setting context and event_metadata attributes directly would not perform these additional checks and updates.

Parameters
event_dataset (*EventDataset*) – New event to be used for importance sampling.

property sample_efficiency

sampling_importance_resampling(*num_samples=None*, *random_state=None*)

Generate unweighted posterior samples from weighted ones. New samples are sampled with probability proportional to the sample weight. Resampling is done with replacement, until the desired number of unweighted samples is obtained.

Parameters

- **num_samples** (*int*) – Number of samples to resample.
- **random_state** (*int or None*) – Sampling seed.

Returns
Unweighted samples

Return type
pd.Dataframe

`property search_parameter_keys`

`split`(*num_parts*)

Split the Result into a set of smaller results. The samples are evenly divided among the sub-results. Additional information (metadata, context, etc.) are copied into each.

This is useful for splitting expensive tasks such as importance sampling across multiple jobs.

Parameters
num_parts (*int*) – The number of parts to split the Result across.

Return type
list of sub-Results.

`train_unconditional_flow`(*parameters*, *nde_settings: dict*, *train_dir: str | None = None*, *threshold_std: float | None = inf*)

Train an unconditional flow to represent the distribution of self.samples.

Parameters

- **parameters** (*list*) – List of parameters over which to train the flow. Can be a subset of the existing parameters.
- **nde_settings** (*dict*) – Configuration settings for the neural density estimator.
- **train_dir** (*Optional[str]*) – Where to save the output of network training, e.g., logs, checkpoints. If not provide, a temporary directory is used.
- **threshold_std** (*Optional[float]*) – Drop samples more than threshold_std standard deviations away from the mean (in any parameter) before training the flow. This is meant to remove outlier samples.

Return type
PosteriorModel

`dingo.core.result.check_equal_dict_of_arrays`(*a*, *b*)

`dingo.core.result.freeze`(*d*)

dingo.core.samplers module

`class dingo.core.samplers.GNPESampler`(*model:* PosteriorModel, *init_sampler:* Sampler, *num_iterations: int = 1*)

Bases: `Sampler`

Base class for GNPE sampler. It wraps a PosteriorModel *and* a standard Sampler for initialization. The former is used to generate initial samples for Gibbs sampling.

A GNPE network is conditioned on additional "proxy" context theta^, i.e.,

p(theta | theta^, d)

The theta^ depend on theta via a fixed kernel p(theta^ | theta). Combining these known distributions, this class uses Gibbs sampling to draw samples from the joint distribution,

p(theta, theta^ | d)

The advantage of this approach is that we are allowed to perform any transformation of d that depends on theta^. In particular, we can use this freedom to simplify the data, e.g., by aligning data to have merger times = 0 in each detector. The merger times are unknown quantities that must be inferred jointly with all other parameters, and GNPE provides a means to do this iteratively. See https://arxiv.org/abs/2111.13139 for additional details.

Gibbs sampling breaks access to the probability density, so this must be recovered through other means. One way is to train an unconditional flow to represent p(theta^ | d) for fixed d based on the samples produced through the GNPE Gibbs sampling. Starting from these, a single Gibbs iteration gives theta from the GNPE network, along with the probability density in the joint space. This is implemented in GNPESampler provided the init_sampler provides proxies directly and num_iterations = 1.

Attributes (beyond those of Sampler)

init_sampler
: [Sampler] Used for providing initial samples for Gibbs sampling.

num_iterations
: [int] Number of Gibbs iterations to perform.

iteration_tracker : IterationTracker **not set up** remove_init_outliers : float **not set up**

param model
type model
: PosteriorModel

param init_sampler
: Used for generating initial samples

type init_sampler
: Sampler

param num_iterations
: Number of GNPE iterations to be performed by sampler.

type num_iterations
: int

property gnpe_proxy_parameters

property init_sampler

property num_iterations

The number of GNPE iterations to perform when sampling.

class dingo.core.samplers.Sampler(*model: PosteriorModel*)

Bases: `object`

Sampler class that wraps a PosteriorModel. Allows for conditional and unconditional models.

Draws samples from the model based on (optional) context data.

This is intended for use either as a standalone sampler, or as a sampler producing initial sample points for a GNPE sampler.

run_sampler()

log_prob()

to_result()

`to_hdf5()`

`model`

Type
PosteriorModel

`inference_parameters`

Type
list

`samples`
Samples produced from the model by run_sampler().

Type
DataFrame

`context`

Type
dict

`metadata`

Type
dict

`event_metadata`

Type
dict

`unconditional_model`
Whether the model is unconditional, in which case it is not provided context information.

Type
bool

`transform_pre, transform_post`
Transforms to be applied to data and parameters during inference. These are typically implemented in a subclass.

Type
Transform

Parameters
model (`PosteriorModel`) –

`property context`
Data on which to condition the sampler. For injections, there should be a 'parameters' key with truth values.

`property event_metadata`
Metadata for data analyzed. Can in principle influence any post-sampling parameter transformations (e.g., sky position correction), as well as the likelihood detector positions.

`log_prob`(*samples: DataFrame*) → ndarray
Calculate the model log probability at specific sample points.

Parameters
samples (*pd.DataFrame*) – Sample points at which to calculate the log probability.

Return type
np.array of log probabilities.

run_sampler(*num_samples: int*, *batch_size: int | None = None*)

Generates samples and stores them in self.samples. Conditions the model on self.context if appropriate (i.e., if the model is not unconditional).

If possible, it also calculates the log_prob and saves it as a column in self.samples. When using GNPE it is not possible to obtain the log_prob due to the many Gibbs iterations. However, in the case of just one iteration, and when starting from a sampler for the proxy, the GNPESampler does calculate the log_prob.

Allows for batched sampling, e.g., if limited by GPU memory. Actual sampling for each batch is performed by _run_sampler(), which will differ for Sampler and GNPESampler.

Parameters

- **num_samples** (*int*) – Number of samples requested.
- **batch_size** (*int, optional*) – Batch size for sampler.

to_hdf5(*label='result'*, *outdir='.'*)

to_result() → *Result*

Export samples, metadata, and context information to a Result instance, which can be used for saving or, e.g., importance sampling, training an unconditional flow, etc.

Return type
Result

write_pesummary(*filename*)

dingo.core.transforms module

class dingo.core.transforms.**GetItem**(*key*)

Bases: object

class dingo.core.transforms.**RenameKey**(*old*, *new*)

Bases: object

Module contents

dingo.gw package

Subpackages

dingo.gw.conversion package

Submodules

dingo.gw.conversion.spin_conversion module

dingo.gw.conversion.spin_conversion.**cartesian_spins**(*p*, *f_ref*)

Transform PE spins to cartesian spins.

Parameters

- **p** (*dict*) – contains parameters, including PE spins
- **f_ref** (*float*) – reference frequency for definition of spins

Returns
  result – parameters, including cartesian spins

Return type
  dict

`dingo.gw.conversion.spin_conversion.`**`change_spin_conversion_phase`**(*samples*, *f_ref*, *sc_phase_old*, *sc_phase_new*)

Change the phase used to convert cartesian spins to PE spins. The cartesian spins are independent of the spin conversion phase. When converting from cartesian spins to PE spins, the phase value has an impact on theta_jn and phi_jl.

The usual convention for the PE spins is to use the phase parameter for the conversion (cart. spins <–> PE spins), but for dingo-IS with the synthetic phase extension we need to use another convention, where the PE spins are defined with spin conversion phase 0. This function transforms between the different conventions.

Parameters

- **samples** (*pd.Dataframe*) – Parameters.
- **f_ref** (*float*) – Reference frequency for definition of spins.
- **sc_phase_old** (*float or None*) – Spin conversion phase used for input parameters. If None, use the phase parameter.
- **sc_phase_new** (*float or None*) – Spin conversion phase used for output parameters. If None, use the phase parameter.

Returns
  parameters with changed spin conversion phase

Return type
  p_new

`dingo.gw.conversion.spin_conversion.`**`component_masses`**(*p*)

`dingo.gw.conversion.spin_conversion.`**`pe_spins`**(*p*, *f_ref*)

Transform cartesian spins to PE spins.

Parameters

- **p** (*dict*) – contains parameters, including cartesian spins
- **f_ref** (*float*) – reference frequency for definition of spins

Returns
  result – parameters, including PE spins

Return type
  dict

Module contents

dingo.gw.data package

Submodules

dingo.gw.data.data_download module

`dingo.gw.data.data_download.`**`download_psd`**(*det*, *time_start*, *time_psd*, *window*, *f_s*)

Download strain data and generate a PSD based on these. Use num_segments of length time_segment, starting at GPS time time_start.

Parameters

- **det** (*str*) – detector
- **time_start** (*float*) – start GPS time for PSD estimation
- **time_psd** (*float = 1024*) – time in seconds for strain used for PSD generation
- **window** (*Union(np.ndarray, dict)*) – Window used for PSD generation, needs to be the same as used for Fourier transform of event strain data. Provided as dict, window is generated by window = dingo.gw.gwutils.get_window(**window).
- **f_s** (*float*) – sampling rate of strain data

Returns
psd – array of psd

Return type
np.array

`dingo.gw.data.data_download.`**`download_raw_data`**(*time_event*, *time_segment*, *time_psd*, *time_buffer*, *detectors*, *window*, *f_s*)

dingo.gw.data.data_preparation module

`dingo.gw.data.data_preparation.`**`data_to_domain`**(*raw_data*, *settings_raw_data*, *domain*, ***kwargs*)

Parameters

- **raw_data** –
- **settings_raw_data** –
- **model_metadata** –

Returns
data – dict with domain_data

Return type
dict

`dingo.gw.data.data_preparation.`**`get_event_data_and_domain`**(*model_metadata*, *time_event*, *time_psd*, *time_buffer*, *event_dataset=None*) `dingo.gw.data.data_preparation.`**`load_raw_data`**(*time_event*, *settings*, *event_dataset=None*)

Load raw event data.

- If event_dataset is provided and event data is saved in it, load and return the data
- Else, event data is downloaded. If event_dataset is provided, the event data is additionally saved to the file.

Parameters

- **time_event** (*float*) – gps time of the events
- **settings** (*dict*) – dict with the settings
- **event_dataset** (*str*) – name of the event dataset file

`dingo.gw.data.data_preparation.`**`parse_settings_for_raw_data`**(*model_metadata*, *time_psd*, *time_buffer*)

dingo.gw.data.event_dataset module

class `dingo.gw.data.event_dataset.`**`EventDataset`**(*file_name=None*, *dictionary=None*)

Bases: *DingoDataset*

Dataset class for storing single event.

For constructing, provide either file_name, or dictionary containing data and settings entries, or neither.

Parameters

- **file_name** (*str*) – HDF5 file containing a dataset
- **dictionary** (*dict*) – Contains settings and data entries. The data keys should be the same as save_keys
- **data_keys** (*list*) – Variables that should be saved / loaded. This allows for class to store additional variables beyond those that are saved. Typically, this list would be provided by any subclass.

dataset_type = 'event_dataset'

Module contents

dingo.gw.dataset package

Submodules

dingo.gw.dataset.generate_dataset module

`dingo.gw.dataset.generate_dataset.`**`generate_dataset`**(*settings: Dict*, *num_processes: int*) → *WaveformDataset*

Generate a waveform dataset.

Parameters

- **settings** (*dict*) – Dictionary of settings to configure the dataset
- **num_processes** (*int*) –

Return type
A WaveformDataset based on the settings.

dingo.gw.dataset.generate_dataset.**generate_parameters_and_polarizations**(*waveform_generator:* WaveformGenerator, *prior: BBHPriorDict*, *num_samples: int*, *num_processes: int*) → Tuple[DataFrame, Dict[str, ndarray]]

Generate a dataset of waveforms based on parameters drawn from the prior.

Parameters

- **waveform_generator** (WaveformGenerator) –
- **prior** (*Prior*) –
- **num_samples** (*int*) –
- **num_processes** (*int*) –

Returns

- *pandas DataFrame of parameters*
- *dictionary of numpy arrays corresponding to waveform polarizations*

dingo.gw.dataset.generate_dataset.**main**()

dingo.gw.dataset.generate_dataset.**parse_args**()

dingo.gw.dataset.generate_dataset.**train_svd_basis**(*dataset:* WaveformDataset, *size: int*, *n_train: int*)

Train (and optionally validate) an SVD basis.

Parameters

- **dataset** (WaveformDataset) – Contains waveforms to be used for building SVD.
- **size** (*int*) – Number of elements to keep for the SVD basis.
- **n_train** (*int*) – Number of training waveforms to use. Remaining are used for validation. Note that the actual number of training waveforms is n_train * len(polarizations), since there is one waveform used for each polarization.

Returns
Since EOB waveforms can fail to generate, provide also the number used in training and validation.

Return type
SVDBasis, n_train, n_test

dingo.gw.dataset.generate_dataset_dag module

dingo.gw.dataset.generate_dataset_dag.**configure_runs**(*settings*, *num_jobs*, *temp_dir*)

Prepare and save settings .yaml files for generating subsets of the dataset. Generally this will produce two .yaml files, one for generating the main dataset, one for the SVD training.

Parameters

- **settings** (*dict*) – Settings for full dataset configuration.
- **num_jobs** (*int*) – Number of jobs over which to split the run.
- **temp_dir** (*str*) – Name of (temporary) directory in which to place temporary output files.

dingo.gw.dataset.generate_dataset_dag.**create_args_string**(*args_dict: Dict*)

Generate argument string from dictionary of argument names and arguments.

dingo.gw.dataset.generate_dataset_dag.**create_dag**(*args*, *settings*)

Create a Condor DAG from command line arguments to carry out the five steps in the workflow.

dingo.gw.dataset.generate_dataset_dag.**main**()

dingo.gw.dataset.generate_dataset_dag.**modulus_check**(*a: int*, *b: int*, *a_label: str*, *b_label: str*)

Raise error if a % b != 0.

dingo.gw.dataset.generate_dataset_dag.**parse_args**()

dingo.gw.dataset.utils module

dingo.gw.dataset.utils.**build_svd_cli**()

Command-line function to build an SVD based on an uncompressed dataset file.

dingo.gw.dataset.utils.**merge_datasets**(*dataset_list: List[WaveformDataset]*) → *WaveformDataset*

Merge a collection of datasets into one.

Parameters

dataset_list (*list[WaveformDataset]*) – A list of WaveformDatasets. Each item should be a dictionary containing parameters and polarizations.

Return type

WaveformDataset containing the merged data.

dingo.gw.dataset.utils.**merge_datasets_cli**()

Command-line function to combine a collection of datasets into one. Used for parallelized waveform generation.

dingo.gw.dataset.waveform_dataset module

class dingo.gw.dataset.waveform_dataset.**WaveformDataset**(*file_name=None*, *dictionary=None*, *transform=None*, *precision=None*, *domain_update=None*, *svd_size_update=None*)

Bases: *DingoDataset*, Dataset

This class stores a dataset of waveforms (polarizations) and corresponding parameters.

It can load the dataset either from an HDF5 file or suitable dictionary.

Once a waveform data set is in memory, the waveform data are consumed through a __getitem__() call, optionally applying a chain of transformations, which are classes that implement a __call__() method.

For constructing, provide either file_name, or dictionary containing data and settings entries, or neither.

Parameters

- **file_name** (*str*) – HDF5 file containing a dataset
- **dictionary** (*dict*) – Contains settings and data entries. The dictionary keys should be 'settings', 'parameters', and 'polarizations'.
- **transform** (*Transform*) – Transform to be applied to dataset samples when accessed through __getitem__
- **precision** (*str ('single', 'double')*) – If provided, changes precision of loaded dataset.
- **domain_update** (*dict*) – If provided, update domain from existing domain using new settings.
- **svd_size_update** (*int*) – If provided, reduces the SVD size when decompressing (for speed).

`dataset_type = 'waveform_dataset'`

`initialize_decompression`(*svd_size_update: int | None = None*)

Sets up decompression transforms. These are applied to the raw dataset before self.transform. E.g., SVD decompression.

Parameters
svd_size_update (*int*) – If provided, reduces the SVD size when decompressing (for speed).

`load_supplemental`(*domain_update=None*, *svd_size_update=None*)

Method called immediately after loading a dataset.

Creates (and possibly updates) domain, updates dtypes, and initializes any decompression transform. Also zeros data below f_min, and truncates above f_max.

Parameters

- **domain_update** (*dict*) – If provided, update domain from existing domain using new settings.
- **svd_size_update** (*int*) – If provided, reduces the SVD size when decompressing (for speed).

`parameter_mean_std()`

`update_domain`(*domain_update: dict | None = None*)

Update the domain based on new configuration.

The waveform dataset provides waveform polarizations in a particular domain. In Frequency domain, this is [0, domain._f_max]. Furthermore, data is set to 0 below domain._f_min. In practice one may want to train a network based on slightly different domain settings, which corresponds to truncating the likelihood integral.

This method provides functionality for that. It truncates and/or zeroes the dataset to the range specified by the domain, by calling domain.update_data.

Parameters
`domain_update` (*dict*) – Settings dictionary. Must contain a subset of the keys contained in domain_dict.

Module contents

dingo.gw.importance_sampling package

Submodules

dingo.gw.importance_sampling.diagnostics module

`dingo.gw.importance_sampling.diagnostics.`**`plot_diagnostics`**`(`*result: Result, outdir, num_processes=1, num_slice_plots=0, n_grid_slice1d=200, n_grid_slice2d=100, params_slice2d=None*`)`

`dingo.gw.importance_sampling.diagnostics.`**`plot_posterior_slice`**`(`*sampler, theta, theta_range, outname=None, num_processes=1, n_grid=200, parameters=None, normalize_conditionals=False*`)`

`dingo.gw.importance_sampling.diagnostics.`**`plot_posterior_slice2d`**`(`*sampler, theta, theta_range, n_grid=100, num_processes=1, outname=None*`)`

dingo.gw.importance_sampling.importance_weights module

Step 1: Train unconditional nde Step 2: Set up likelihood and prior

`dingo.gw.importance_sampling.importance_weights.`**`main`**`()`

`dingo.gw.importance_sampling.importance_weights.`**`parse_args`**`()`

Module contents

Implements sampling-importance-resampling (sir) for GW posteriors.

dingo.gw.inference package

Submodules

dingo.gw.inference.gw_samplers module

class dingo.gw.inference.gw_samplers.**GWSampler**(***kwargs*)

Bases: *GWSamplerMixin*, *Sampler*

Sampler for gravitational-wave inference using neural posterior estimation. Augments the base class by defining transform_pre and transform_post to prepare data for the inference network.

transform_pre :

- Whitens strain.
- Repackages strain data and the inverse ASDs (suitably scaled) into a torch tensor.

transform_post :

- Extract the desired inference parameters from the network output (array-like), de-standardize them, and repackage as a dict.

Also mixes in GW functionality for building the domain and correcting the reference time.

Allows for conditional and unconditional models, and draws samples from the model based on (optional) context data.

This is intended for use either as a standalone sampler, or as a sampler producing initial sample points for a GNPE sampler.

Parameters

kwargs – Keyword arguments that are forwarded to the superclass.

class dingo.gw.inference.gw_samplers.**GWSamplerGNPE**(***kwargs*)

Bases: *GWSamplerMixin*, *GNPESampler*

Gravitational-wave GNPE sampler. It wraps a PosteriorModel and a standard Sampler for initialization. The former is used to generate initial samples for Gibbs sampling.

Compared to the base class, this class implements the required transforms for preparing data and parameters for the network. This includes GNPE transforms, data processing transforms, and standardization/de-standardization of parameters.

A GNPE network is conditioned on additional "proxy" context theta^, i.e.,

p(theta | theta^, d)

The theta^ depend on theta via a fixed kernel p(theta^ | theta). Combining these known distributions, this class uses Gibbs sampling to draw samples from the joint distribution,

p(theta, theta^ | d)

The advantage of this approach is that we are allowed to perform any transformation of d that depends on theta^. In particular, we can use this freedom to simplify the data, e.g., by aligning data to have merger times = 0 in each detector. The merger times are unknown quantities that must be inferred jointly with all other parameters, and GNPE provides a means to do this iteratively. See https://arxiv.org/abs/2111.13139 for additional details.

Gibbs sampling breaks access to the probability density, so this must be recovered through other means. One way is to train an unconditional flow to represent p(theta^ | d) for fixed d based on the samples produced through the GNPE Gibbs sampling. Starting from these, a single Gibbs iteration gives theta from the GNPE network, along with the probability density in the joint space. This is implemented in GNPESampler provided the init_sampler provides proxies directly and num_iterations = 1.

Attributes (beyond those of Sampler)

init_sampler
[Sampler] Used for providing initial samples for Gibbs sampling.

num_iterations
[int] Number of Gibbs iterations to perform.

iteration_tracker
[IterationTracker] **not set up**

remove_init_outliers
[float] **not set up**

param kwargs
Keyword arguments that are forwarded to the superclass.

`class dingo.gw.inference.gw_samplers.`**`GWSamplerMixin`**`(**kwargs)`

Bases: `object`

Mixin class designed to add gravitational wave functionality to Sampler classes:

- builder for data domain
- correction for fixed detector locations during training (t_ref)

Parameters
`kwargs` – Keyword arguments that are forwarded to the superclass.

dingo.gw.inference.inference_pipeline module

`dingo.gw.inference.inference_pipeline.`**`analyze_event`**`()`

`dingo.gw.inference.inference_pipeline.`**`get_event_data`**`(event, args, model, ref=None)`

`dingo.gw.inference.inference_pipeline.`**`parse_args`**`()`

`dingo.gw.inference.inference_pipeline.`**`prepare_log_prob`**`(sampler, num_samples: int, nde_settings: dict, batch_size: int | None = None, threshold_std: float | None = inf, remove_init_outliers: float | None = 0.0, low_latency_label: str | None = None, outdir: str | None = None)`

Prepare gnpe sampling with log_prob. This is required, since in its vanilla form gnpe does not provide the density for its samples.

Specifically, we train an unconditional neural density estimator (nde) for the gnpe proxies. This requires running the gnpe sampler till convergence, and extracting the gnpe proxies after the final gnpe iteration. The nde is trained to match the distribution over gnpe proxies, which provides a way of rapidly sampling (converged!) gnpe proxies *and* evaluating the log_prob.

After this preparation step, self.run_sampler can leverage self.gnpe_proxy_sampler (which is based on the aforementioned trained nde) to sample gnpe proxies, such that one gnpe iteration is sufficient. The log_prob of the samples in the *joint* space (inference parameters + gnpe proxies) is then simply given by the sum of the corresponding log_probs (from self.model and self.gnpe_proxy_sampler.model).

Parameters

- **num_samples** (*int*) – number of samples for training of nde
- **batch_size** (*int = None*) – batch size for sampler
- **threshold_std** (*float = np.inf*) – gnpe proxies deviating by more then threshold_std standard deviations from the proxy mean (along any axis) are discarded.
- **low_latency_label** (*str = None*) – File label for low latency samples (= samples used for training nde). If None, these samples are not saved.
- **outdir** (*str = None*) – Directory in which low latency samples are saved. Needs to be set if low_latency_label is not None.

dingo.gw.inference.visualization module

dingo.gw.inference.visualization.**generate_cornerplot**(**sample_sets*, *filename=None*)

dingo.gw.inference.visualization.**load_ref_samples**(*ref_samples_file*, *drop_geocent_time=True*)

Module contents

dingo.gw.noise package

Subpackages

dingo.gw.noise.synthetic package

Submodules

dingo.gw.noise.synthetic.asd_parameterization module

dingo.gw.noise.synthetic.asd_parameterization.**curve_fit**(*data*, *std*, *delta_f=None*)

Fit a Lorentzian to the PSD.

Parameters

- **data** (*dict*) – Dictionary containing the PSD, broadband noise, and frequency grid.
- **std** (*float*) – Standard deviation of the Gaussian noise.
- **delta_f** (*float*) – Truncation parameter for Lorentzians. Set to None if non-positive value is passed.

dingo.gw.noise.synthetic.asd_parameterization.**fit_broadband_noise**(*domain*, *psd*, *num_spline_positions*, *sigma*, *f_min=20*)

Fit a spline to the broadband noise of a PSD.

Parameters

- **domain** (Domain) – Domain object containing the frequency grid.
- **psd** (*array_like*) – PSD to be parameterized.
- **num_spline_positions** (*int*) – Number of spline positions.
- **sigma** (*float*) – Standard deviation of the Gaussian noise used for the spline fit.

- **f_min** (*float, optional*) – position of the first node for the spline fi

dingo.gw.noise.synthetic.asd_parameterization.**fit_spectral**(*frequencies*, *psd*, *broadband_noise*, *num_spectral_segments*, *sigma*, *delta_f*)

Fit Lorentzians to the spectral features of a PSD.

Parameters

- **frequencies** (*array_like*) – Frequency grid.
- **psd** (*array_like*) – PSD to be parameterized.
- **broadband_noise** (*array_like*) – Broadband noise of the PSD.
- **num_spectral_segments** (*int*) – Number of spectral segments.
- **sigma** (*float*) – Standard deviation of the Gaussian noise used for the spline fit.
- **delta_f** (*float*) – Truncation parameter for Lorentzians. Set to None if non-positive value is passed.

dingo.gw.noise.synthetic.asd_parameterization.**parameterize_asd_dataset**(*real_dataset*, *parameterization_settings*, *num_processes*, *verbose*)

Parameterize a dataset of ASDs using a spline fit to the broadband noise and Lorentzians for the spectral features.

Parameters

- **real_dataset** (ASDDataset) – Dataset containing the ASDs to be parameterized.
- **parameterization_settings** (*dict*) – Dictionary containing the settings for the parameterization.
- **num_processes** (*int*) – Number of processes to use for parallelization.
- **verbose** (*bool*) – If True, print progress bars.

dingo.gw.noise.synthetic.asd_parameterization.**parameterize_asds_parallel**(*asds*, *domain*, *parameterization_settings*, *pool=None*, *verbose=False*)

Helper function to be called for parallel ASD parameterization.

Parameters

- **asds** (*array_like*) – Array containing the ASDs to be parameterized.
- **domain** (Domain) – Domain object containing the frequency grid.
- **parameterization_settings** (*dict*) – Dictionary containing the settings for the parameterization.
- **pool** (*Pool, optional*) – Pool object for parallelization. If None, the function is not parallelized.
- **verbose** (*bool*) – If True, print progress bars.

dingo.gw.noise.synthetic.asd_parameterization.**parameterize_single_psd**(*real_psd*, *domain*, *parameterization_settings*)

Parameterize a single ASD using a spline fit to the broadband noise and Lorentzians for the spectral features.

Parameters

- **real_psd** (*array_like*) – PSD to be parameterized.
- **domain** (Domain) – Domain object containing the frequency grid.
- **parameterization_settings** (*dict*) – Dictionary containing the settings for the parameterization.

dingo.gw.noise.synthetic.asd_sampling module

class dingo.gw.noise.synthetic.asd_sampling.**KDE**(*parameter_dict*, *sampling_settings*)

Bases: object

Kernel Density Estimation (KDE) class for sampling ASDs.

Parameters

- **parameter_dict** (*dict*) – Dictionary containing the parameters of the ASDs used for fitting the synthetic distribution.
- **sampling_settings** (*dict*) – Dictionary containing the settings for the sampling.

fit(*weights=None*)

Fit the KDEs to the parameters saved in 'self.parameter_dict'. :param weights: Weights for the KDEs. If None, all weights are set to 1. :type weights: array_like, optional

sample(*num_samples*, *rescaling_ys=None*)

Sample a synthetic ASD dataset from the fitted KDEs

Parameters: num_samples (int): Number of samples to draw. rescaling_ys (dict): Optional dictionary of spline y-values used for rescaling the base noise.

dingo.gw.noise.synthetic.asd_sampling.**get_rescaling_params**(*filenames*, *parameterization_settings*)

Get the parameters of the ASDs that are used for rescaling. :param filenames: Dictionary containing the paths to the ASD files. :type filenames: dict :param parameterization_settings: Dictionary containing the settings for the parameterization. :type parameterization_settings: dict

dingo.gw.noise.synthetic.generate_dataset module

dingo.gw.noise.synthetic.generate_dataset.**generate_dataset**(*real_dataset*, *settings: Dict*, *num_samples*, *num_processes: int*, *verbose: bool*)

Generate a synthetic ASD dataset from an existing dataset of real ASDs.

Parameters

- **real_dataset** (ASDDataset) – Existing dataset of real ASDs.
- **settings** (*dict*) – Dictionary containing the settings for the parameterization and sampling.
- **num_processes** (*int*) – Number of processes to use in pool for parallel parameterization.
- **verbose** (*bool*) – Whether to print progress information.

dingo.gw.noise.synthetic.generate_dataset.**main**()

dingo.gw.noise.synthetic.generate_dataset.**parse_args**()

dingo.gw.noise.synthetic.utils module

dingo.gw.noise.synthetic.utils.**get_index_for_elem**(*arr*, *elem*)

dingo.gw.noise.synthetic.utils.**lorentzian_eval**(*x*, *f0*, *A*, *Q*, *delta_f=None*)

Evaluates a Lorentzian function at the given frequencies. :param x: Frequencies at which the Lorentzian is evaluated. :type x: array_like :param f0: Center frequency of the Lorentzian. :type f0: float :param A: Amplitude of the Lorentzian. :type A: float :param Q: Parameter determining the width of the Lorentzian :type Q: float :param delta_f: If given, the Lorentzian is truncated :type delta_f: float, optional

Return type
array_like

dingo.gw.noise.synthetic.utils.**reconstruct_psds_from_parameters**(*parameters_dict*, *domain*, *parameterization_settings*)

Reconstructs the PSDs from the parameters. :param parameters_dict: Dictionary containing the parameters of the PSDs. :type parameters_dict: dict :param domain: Domain object containing the frequencies at which the PSDs are evaluated. :type domain: dingo.gw.noise.domain.Domain :param parameterization_settings: Dictionary containing the settings for the parameterization. :type parameterization_settings: dict

Return type
array_like

Module contents

Submodules

dingo.gw.noise.asd_dataset module

class dingo.gw.noise.asd_dataset.**ASDDataset**(*file_name=None*, *dictionary=None*, *ifos=None*, *precision=None*, *domain_update=None*)

Bases: `DingoDataset`

Dataset of amplitude spectral densities (ASDs). The ASDs are typically used for whitening strain data, and additionally passed as context to the neural density estimator.

Parameters

- **file_name** (*str*) – HDF5 file containing a dataset
- **dictionary** (*dict*) – Contains settings and data entries. The dictionary keys should be 'settings', 'asds', and 'gps_times'.
- **ifos** (*List[str]*) – List of detectors used for dataset, e.g. ['H1', 'L1']. If not set, all available ones in the dataset are used.
- **precision** (*str ('single', 'double')*) – If provided, changes precision of loaded dataset.
- **domain_update** (*dict*) – If provided, update domain from existing domain using new settings.

dataset_type = 'asd_dataset'

property gps_info

Min/Max GPS time for each detector.

property length_info

The number of asd samples per detector.

sample_random_asds()

Sample a random asd for each detector. :rtype: Dict with a random asd from the dataset for each detector.

update_domain(*domain_update*)

Update the domain based on new configuration. Also adjust data arrays to match the new domain.

The ASD dataset provides ASDs in a particular domain. In Frequency domain, this is [0, domain._f_max]. In practice one may want to train a network based on slightly different domain settings, which corresponds to truncating the likelihood integral.

This method provides functionality for that. It truncates the data below a new f_max, and sets the ASD below f_min to a large but finite value.

Parameters

domain_update (*dict*) – Settings dictionary. Must contain a subset of the keys contained in domain_dict.

dingo.gw.noise.asd_estimation module

dingo.gw.noise.asd_estimation.**download_and_estimate_cli**()

Command-line function to download strain data and estimate PSDs based on the data. Used for parallelized ASD dataset generation.

dingo.gw.noise.asd_estimation.**download_and_estimate_psds**(*data_dir: str*, *settings: dict*, *time_segments: dict*, *verbose=False*)

Downloads strain data for the specified time segments and estimates PSDs based on these

Parameters

- **data_dir** (*str*) – Path to the directory where the PSD dataset will be stored
- **settings** (*dict*) – Settings that determine the segments
- **time_segments** (*dict*) – specifying the time segments used for downloading the data
- **verbose** (*bool*) – optional parameter determining if progress should be printed

Return type

A dictionary containing the paths to the dataset files

dingo.gw.noise.generate_dataset module

dingo.gw.noise.generate_dataset.**generate_dataset**()

Creates and saves an ASD dataset

dingo.gw.noise.generate_dataset.**parse_args**()

dingo.gw.noise.generate_dataset_dag module

dingo.gw.noise.generate_dataset_dag.**create_args_string**(*args_dict: Dict*)

Generate argument string from dictionary of argument names and arguments.

dingo.gw.noise.generate_dataset_dag.**create_dag**(*data_dir*, *settings_file*, *time_segments*, *out_name*)

Create a Condor DAG to (a) download, estimate, individual PSDs and (b) merge them into one dataset

Parameters

- **data_dir** (*str*) – Path to the directory where the PSD dataset will be stored
- **settings_file** (*str*) – Settings : Path to settings file relevant for PSD generation
- **time_segments** (*dict*) – contains all time segments used for estimating PSDs
- **out_name** (*str*) – path where the resulting ASD dataset should be stored

Return type

Condor DAG

dingo.gw.noise.generate_dataset_dag.**split_time_segments**(*time_segments*, *condor_dir*, *num_jobs*)

Split up all time segments used for estimating PSDs into num_jobs-many segments and save them into a condor directory

Parameters

- **time_segments** (*dict*) – contains all time segments used for estimating PSDs
- **condor_dir** (*str*) – path to a directory where condr-related files are stored
- **num_jobs** (*int*) – number of jobs that should be used per detector to parallelize the PSD estimation

Return type

List of paths where the files including the subsets of all time segments are stored

dingo.gw.noise.utils module

dingo.gw.noise.utils.**CATALOGS = ['GWTC-1-confident', 'GWTC-2.1-confident', 'GWTC-3-confident']**

Contains links for PSD segment lists with quality label BURST_CAT2 from the Gravitational Wave Open Science Center. Some events are split up into multiple chunks such that there are multiple URLs for one observing run

dingo.gw.noise.utils.**get_event_gps_times**()

dingo.gw.noise.utils.**get_time_segments**(*settings*)

Creates a dictionary storing time segments used for estimating PSDs :param settings: Settings that determine the segments :type settings: dict

Return type

Dictionary containing the time segments for each detector

dingo.gw.noise.utils.**merge_datasets**(*asd_dataset_list*)

Merges a list of asd datasets into ont :param asd_dataset_list: :type asd_dataset_list: List of ASDDatasets to be merged

Return type

A single combined ASDDataset object

dingo.gw.noise.utils.**merge_datasets_cli**()

Command-line function to combine a collection of datasets into one. Used for parallelized ASD dataset generation.

dingo.gw.noise.utils.**psd_data_path**(*data_dir*, *run*, *detector*)

Return the directory where the PSD data is to be stored :param data_dir: Path to the directory where the PSD dataset will be stored :type data_dir: str :param run: Observing run that is used for the PSD dataset generation :type run: str :param detector: Detector that is used for the PSD dataset generation :type detector: str

Return type

the path where the data is stored

Module contents

dingo.gw.training package

Submodules

dingo.gw.training.train_builders module

dingo.gw.training.train_builders.**build_dataset**(*data_settings*)

Build a dataset based on a settings dictionary. This should contain the path of a saved waveform dataset.

This function also truncates the dataset as necessary.

Parameters

data_settings (*dict*) –

Return type

WaveformDataset

dingo.gw.training.train_builders.**build_svd_for_embedding_network**(*wfd: WaveformDataset*, *data_settings: dict*, *asd_dataset_path: str*, *size: int*, *num_training_samples: int*, *num_validation_samples: int*, *num_workers: int = 0*, *batch_size: int = 1000*, *out_dir=None*)

Construct SVD matrices V based on clean waveforms in each interferometer. These will be used to seed the weights of the initial projection part of the embedding network.

It first generates a number of training waveforms, and then produces the SVD.

Parameters

- **wfd** (WaveformDataset) –
- **data_settings** (*dict*) –
- **asd_dataset_path** (*str*) – Training waveforms will be whitened with respect to these ASDs.
- **size** (*int*) – Number of basis elements to include in the SVD projection.
- **num_training_samples** (*int*) –
- **num_validation_samples** (*int*) –

- **num_workers** (*int*) –
- **batch_size** (*int*) –
- **out_dir** (*str*) – SVD performance diagnostics are saved here.

Returns
The V matrices for each interferometer. They are ordered as in data_settings['detectors'].

Return type
list of numpy arrays

dingo.gw.training.train_builders.**set_train_transforms**(*wfd*, *data_settings*, *asd_dataset_path*, *omit_transforms=None*)

Set the transform attribute of a waveform dataset based on a settings dictionary. The transform takes waveform polarizations, samples random extrinsic parameters, projects to detectors, adds noise, and formats the data for input to the neural network. It also implements optional GNPE transformations.

Note that the WaveformDataset is modified in-place, so this function returns nothing.

Parameters

- **wfd** (WaveformDataset) –
- **data_settings** (*dict*) –
- **asd_dataset_path** (*str*) – Path corresponding to the ASD dataset used to generate noise.
- **omit_transforms** – List of sub-transforms to omit from the full composition.

dingo.gw.training.train_pipeline module

dingo.gw.training.train_pipeline.**initialize_stage**(*pm*, *wfd*, *stage*, *num_workers*, *resume=False*)

Initializes training based on PosteriorModel metadata and current stage:

- Builds transforms (based on noise settings for current stage);
- Builds DataLoaders;
- At the beginning of a stage (i.e., if not resuming mid-stage), initializes

a new optimizer and scheduler; * Freezes / unfreezes SVD layer of embedding network

Parameters

- **pm** (PosteriorModel) –
- **wfd** (WaveformDataset) –
- **stage** (*dict*) – Settings specific to current stage of training
- **num_workers** (*int*) –
- **resume** (*bool*) – Whether training is resuming mid-stage. This controls whether the optimizer and scheduler should be re-initialized based on contents of stage dict.

Return type
(train_loader, test_loader)

dingo.gw.training.train_pipeline.**parse_args**()

`dingo.gw.training.train_pipeline.`**`prepare_training_new`**(*train_settings: dict*, *train_dir: str*, *local_settings: dict*)

Based on a settings dictionary, initialize a WaveformDataset and PosteriorModel.

For model type 'nsf+embedding' (the only acceptable type at this point) this also initializes the embedding network projection stage with SVD V matrices based on clean detector waveforms.

Parameters

- **`train_settings`** (*`dict`*) – Settings which ultimately come from train_settings.yaml file.
- **`train_dir`** (*`str`*) – This is only used to save diagnostics from the SVD.
- **`local_settings`** (*`dict`*) – Local settings (e.g., num_workers, device)

Return type

(*WaveformDataset*, *PosteriorModel*)

`dingo.gw.training.train_pipeline.`**`prepare_training_resume`**(*checkpoint_name*, *local_settings*, *train_dir*)

Loads a PosteriorModel from a checkpoint, as well as the corresponding WaveformDataset, in order to continue training. It initializes the saved optimizer and scheduler from the checkpoint.

Parameters

- **`checkpoint_name`** (*`str`*) – File name containing the checkpoint (.pt format).
- **`device`** (*`str`*) – 'cuda' or 'cpu'

Return type

(*PosteriorModel*, *WaveformDataset*)

`dingo.gw.training.train_pipeline.`**`train_local`**()

`dingo.gw.training.train_pipeline.`**`train_stages`**(*pm*, *wfd*, *train_dir*, *local_settings*)

Train the network, iterating through the sequence of stages. Stages can change certain settings such as the noise characteristics, optimizer, and scheduler settings.

Parameters

- **`pm`** (`PosteriorModel`) –
- **`wfd`** (`WaveformDataset`) –
- **`train_dir`** (*`str`*) – Directory for saving checkpoints and train history.
- **`local_settings`** (*`dict`*) –

Returns

True if all stages are complete False otherwise

Return type

bool

dingo.gw.training.train_pipeline_condor module

`dingo.gw.training.train_pipeline_condor.copy_logfiles`(*log_dir*, *epoch*, *name='info'*, *suffixes=('.err', '.log', '.out')*)

`dingo.gw.training.train_pipeline_condor.copyfile`(*src*, *dst*)

`dingo.gw.training.train_pipeline_condor.create_submission_file`(*train_dir*, *condor_settings*, *filename='submission_file.sub'*)

TODO: documentation :param train_dir: :param filename: :return:

`dingo.gw.training.train_pipeline_condor.train_condor`()

dingo.gw.training.utils module

`dingo.gw.training.utils.append_stage`()

Module contents

dingo.gw.transforms package

Submodules

dingo.gw.transforms.detector_transforms module

class `dingo.gw.transforms.detector_transforms.ApplyCalibrationUncertainty`(*ifo_list*, *data_domain*, *calibration_envelope*, *num_calibration_curves*, *num_calibration_nodes*)

Bases: `object`

Expand out a waveform using several detector calibration draws. These multiple draws are intended to be used for marginalizing over calibration uncertainty.

Detector calibration uncertainty is modeled as described in https://dcc.ligo.org/LIGO-T1400682/public

Gravitational wave data d is assumed to be of the form

$$d(f) = h_{obs}(f) + n(f),$$

where h_{obs} is the observed waveform and n is the noise. Since the detector is not perfectly calibrated, the observed waveform is not identical to the true waveform $h(f)$. Rather, it is assumed to have corrections of the form

$$h_{obs}(f) = h(f) * (1 + \delta A(f)) * \exp(i\delta\phi(f)),$$

where $\delta A(f)$ and $\delta\phi(f)$ are frequency-dependent amplitude and phase errors. Under the calibration model, these are parametrized with cubic splines, defined in terms of calibration parameters A_i and ϕ_i, defined at log-spaced frequency nodes,

$$\delta A(f) = \mathrm{spline}(f; f_i, \delta A_i),$$
$$\delta\phi(f) = \mathrm{spline}(f; f_i, \delta\phi_i).$$

The calibration parameters are not known precisely, rather they are assumed to be normally distributed, with mean 0 and standard deviation determined by the "calibration envelope", which varies from event to event.

For each detector waveform, this transform draws a collection of N calibration curves $\{(\delta A^n(f), \delta\phi^n(f))\}_{n=1}^N$ according to a calibration envelope, and applies them to generate N observed waveforms $\{h_{obs}^n(f)\}$. This is intended to be used for marginalizing over the calibration uncertainty when evaluating the likelihood for importance sampling.

Parameters

- **ifo_list** (*InterferometerList*) – List of Interferometers present in the analysis.
- **data_domain** (Domain) – Domain on which data is defined.
- **calibration_envelope** (*dict*) – Dictionary of the form `{"H1": filepath, "L1": filepath}`, where the filepaths are strings pointing to ".txt" files containing calibration envelopes. The calibration envelope depends on the event analyzed, and therefore remains fixed for all applications of the transform. The calibration envelope is used to define the variances $(\sigma_{\delta A_i}, \sigma_{\delta\phi_i})$ of the calibration paramters.
- **num_calibration_curves** (*int*) – Number of calibration curves N to produce and apply to the waveform. Ultimately, this will translate to the number of samples in the Monte Carlo estimate of the marginalized likelihood integral.
- **num_calibration_nodes** (*int*) – Number of log-spaced frequency nodes f_i to use in defining the spline.

class dingo.gw.transforms.detector_transforms.**GetDetectorTimes**(*ifo_list*, *ref_time*)

Bases: `object`

Compute the time shifts in the individual detectors based on the sky position (ra, dec), the geocent_time and the ref_time.

class dingo.gw.transforms.detector_transforms.**ProjectOntoDetectors**(*ifo_list*, *domain*, *ref_time*)

Bases: `object`

Project the GW polarizations onto the detectors in ifo_list. This does not sample any new parameters, but relies on the parameters provided in sample['extrinsic_parameters']. Specifically, this transform applies the following operations:

(1) Rescale polarizations to account for sampled luminosity distance

(2) Project polarizations onto the antenna patterns using the ref_time and the extrinsic parameters (ra, dec, psi)

(3) Time shift the strains in the individual detectors according to the times <ifo.name>_time provided in the extrinsic parameters.

class dingo.gw.transforms.detector_transforms.**TimeShiftStrain**(*ifo_list*, *domain*)

Bases: `object`

Time shift the strains in the individual detectors according to the times <ifo.name>_time provided in the extrinsic parameters.

dingo.gw.transforms.detector_transforms.**time_delay_from_geocenter**(*ifo: Interferometer*, *ra: float | ndarray | Tensor*, *dec: float | ndarray | Tensor*, *time: float*)

Calculate time delay between ifo and geocenter. Identical to method ifo.time_delay_from_geocenter(ra, dec, time), but the present implementation allows for batched computation, i.e., it also accepts arrays and tensors for ra and dec.

Implementation analogous to bilby-cython implementation https://git.ligo.org/colm.talbot/bilby-cython/-/blob/main/bilby_cython/geometry.pyx, which is in turn based on XLALArrivaTimeDiff in TimeDelay.c.

Parameters

- **ifo** (*bilby.gw.detector.interferometer.Interferometer*) – bilby interferometer object.
- **ra** (*Union[float, np.array, torch.Tensor]*) – Right ascension of the source in radians. Either float, or float array/tensor.
- **dec** (*Union[float, np.array, torch.Tensor]*) – Declination of the source in radians. Either float, or float array/tensor.
- **time** (*float*) – GPS time in the geocentric frame.

Returns
float

Return type
Time delay between the two detectors in the geocentric frame

dingo.gw.transforms.general_transforms module

class dingo.gw.transforms.general_transforms.**UnpackDict**(*selected_keys*)

Bases: `object`

Unpacks the dictionary to prepare it for final output of the dataloader. Only returns elements specified in selected_keys.

dingo.gw.transforms.gnpe_transforms module

class dingo.gw.transforms.gnpe_transforms.**GNPEBase**(*kernel_dict*, *operators*)

Bases: `ABC`

A base class for Group Equivariant Neural Posterior Estimation [1].

This implements GNPE for *approximate* equivariances. For exact equivariances, additional processing should be implemented within a subclass.

[1]: https://arxiv.org/abs/2111.13139

inverse(*a*, *k*)

multiply(*a*, *b*, *k*)

perturb(*g*, *k*)

Generate proxy variables based on initial parameter values.

Parameters

- **g** (*Union[np.float64, float, torch.Tensor]*) – Initial parameter values
- **k** (*str*) – Parameter name. This is used to identify the group binary operator.

Return type
Proxy variables in the same format as g.

`sample_proxies(input_parameters)`

Given input parameters, perturbs based on the kernel to produce "proxy" ("hatted") parameters, i.e., samples

hat g ~ p(hat g | g).

Typically the GNPE NDE will be conditioned on hat g. Furthermore, these proxy parameters will be used to transform the data to simplify it.

Parameters:

input_parameters
[dict] Initial parameter values to be perturbed. dict values can be either floats (for training) or torch Tensors (for inference).

rtype
A dict of proxy parameters.

`class dingo.gw.transforms.gnpe_transforms.GNPECoalescenceTimes(ifo_list, kernel, exact_global_equivariance=True, inference=False)`

Bases: *GNPEBase*

GNPE [1] Transformation for detector coalescence times.

For each of the detector coalescence times, a proxy is generated by adding a perturbation epsilon from the GNPE kernel to the true detector time. This proxy is subtracted from the detector time, such that the overall time shift only amounts to -epsilon in training. This standardizes the input data to the inference network, since the applied time shifts are always restricted to the range of the kernel.

To preserve information at inference time, conditioning of the inference network on the proxies is required. To that end, the proxies are stored in sample['gnpe_proxies'].

We can enforce an exact equivariance under global time translations, by subtracting one proxy (by convention: the first one, usually for H1 ifo) from all other proxies, and from the geocent time, see [1]. This is enabled with the flag exact_global_equivariance.

Note that this transform does not modify the data itself. It only determines the amount by which to time-shift the data.

[1]: arxiv.org/abs/2111.13139

Parameters

- **ifo_list** (*bilby.gw.detector.InterferometerList*) – List of interferometers.
- **kernel** (*str*) – Defines a Bilby prior, to be used for all interferometers.
- **exact_global_equivariance** (*bool = True*) – Whether to impose the exact global time translation symmetry.
- **inference** (*bool = False*) – Whether to use inference or training mode.

dingo.gw.transforms.inference_transforms module

class dingo.gw.transforms.inference_transforms.**CopyToExtrinsicParameters**(**parameter_list*)

Bases: object

Copy parameters specified in self.parameter_list from sample["parameters"] to sample["extrinsic_parameters"].

class dingo.gw.transforms.inference_transforms.**ExpandStrain**(*num_samples*)

Bases: object

Expand the waveform of sample by adding a batch axis and copying the waveform num_samples times along this new axis. This is useful for generating num_samples samples at inference time.

class dingo.gw.transforms.inference_transforms.**PostCorrectGeocentTime**(*inverse=False*)

Bases: object

Post correction for geocent time: add GNPE proxy (only necessary if exact equivariance is enforced)

class dingo.gw.transforms.inference_transforms.**ResetSample**(*extrinsic_parameters_keys=None*)

Bases: object

Resets sample:

- waveform was potentially modified by gnpe transforms, so reset to **waveform_**
- optionally remove all non-required extrinsic parameters

class dingo.gw.transforms.inference_transforms.**ToTorch**(*device='cpu'*)

Bases: object

Convert all numpy arrays sample to torch tensors and push them to the specified device. All items of sample that are not numpy arrays (e.g., dicts of arrays) remain unchanged.

dingo.gw.transforms.noise_transforms module

class dingo.gw.transforms.noise_transforms.**AddWhiteNoiseComplex**

Bases: object

Adds white noise with a standard deviation determined by self.scale to the complex strain data.

class dingo.gw.transforms.noise_transforms.**RepackageStrainsAndASDS**(*ifos*, *first_index=0*)

Bases: object

Repackage the strains and the asds into an [num_ifos, 3, num_bins] dimensional tensor. Order of ifos is provided by self.ifos. By convention, [:,i,:] is used for:

i = 0: strain.real i = 1: strain.imag i = 2: 1 / (asd * 1e23)

class dingo.gw.transforms.noise_transforms.**SampleNoiseASD**(*asd_dataset*)

Bases: object

Sample a random asds for each detector and add them to sample['asds'].

class dingo.gw.transforms.noise_transforms.**WhitenAndScaleStrain**(*scale_factor*)

Bases: object

Whiten the strain data by dividing w.r.t. the corresponding asds, and scale it with 1/scale_factor.

In uniform frequency domain the scale factor should be np.sqrt(window_factor) / np.sqrt(4.0 * delta_f). It has two purposes:

(*) the denominator accounts for frequency binning (*) dividing by window factor accounts for windowing of strain data

class dingo.gw.transforms.noise_transforms.**WhitenFixedASD**(*domain: FrequencyDomain*, *asd_file: str | None = None*, *inverse: bool = False*, *precision=None*)

Bases: `object`

Whiten frequency-series data according to an ASD specified in a file. This uses the ASD files contained in Bilby.

Parameters

- **domain** (`FrequencyDomain`) – ASD is interpolated to the associated frequency grid.
- **asd_file** (*str*) – Name of the ASD file. If None, use the aligo ASD. [Default: None]
- **inverse** (*bool*) – Whether to apply the inverse whitening transform, to un-whiten data. [Default: False]
- **precision** (*str ("single", "double")*) – If not None, sets precision of ASD to specified precision.

class dingo.gw.transforms.noise_transforms.**WhitenStrain**

Bases: `object`

Whiten the strain data by dividing w.r.t. the corresponding asds.

dingo.gw.transforms.parameter_transforms module

class dingo.gw.transforms.parameter_transforms.**SampleExtrinsicParameters**(*extrinsic_prior_dict*)

Bases: `object`

Sample extrinsic parameters and add them to sample in a separate dictionary.

property reproduction_dict

class dingo.gw.transforms.parameter_transforms.**SelectStandardizeRepackageParameters**(*parameters_dict*, *standardization_dict*, *inverse=False*, *as_type=None*, *device='cpu'*)

Bases: `object`

This transformation selects the parameters in standardization_dict, normalizes them by setting p = (p - mean) / std, and repackages the selected parameters to a numpy array.

as_type: str = None

only applies, if self.inverse == True * if None, data type is kept * if 'dict', dict with * if 'pandas', use pandas.DataFrame

class dingo.gw.transforms.parameter_transforms.**StandardizeParameters**(*mu*, *std*)

Bases: `object`

Standardize parameters according to the transform (x - mu) / std.

Initialize the standardization transform with means and standard deviations for each parameter

Parameters

- **mu** (*Dict[str, float]*) – The (estimated) means
- **std** (*Dict[str, float]*) – The (estimated) standard deviations

inverse(*samples*)

De-standardize the parameter array according to the specified means and standard deviations.

Parameters

- **samples** (*Dict[Dict, Dict]*) – A nested dictionary with keys 'parameters', 'waveform', 'noise_summary'.
- **mu** (*Only parameters included in*) –
- **transformed.** (*std get*) –

Module contents

dingo.gw.waveform_generator package

Submodules

dingo.gw.waveform_generator.frame_utils module

These functions are used for transforming between J and L0 frames.

dingo.gw.waveform_generator.frame_utils.**convert_J_to_L0_frame**(*hlm_J, p, wfg, spin_conversion_phase=None*)

dingo.gw.waveform_generator.frame_utils.**get_JL0_euler_angles**(*p, wfg, spin_conversion_phase=None*)

dingo.gw.waveform_generator.frame_utils.**rotate_y**(*angle, vx, vy, vz*)

dingo.gw.waveform_generator.frame_utils.**rotate_z**(*angle, vx, vy, vz*)

dingo.gw.waveform_generator.waveform_generator module

class dingo.gw.waveform_generator.waveform_generator.**NewInterfaceWaveformGenerator**(***kwargs*)

Bases: *WaveformGenerator*

Generate polarizations using GWSignal routines in the specified domain for a single GW coalescence given a set of waveform parameters.

Parameters

- **approximant** (*str*) – Waveform "approximant" string understood by lalsimulation This is defines which waveform model is used.
- **domain** (*Domain*) – Domain object that specifies on which physical domain the waveform polarizations will be generated, e.g. Fourier domain, time domain.
- **f_ref** (*float*) – Reference frequency for the waveforms

- `f_start` (*float*) – Starting frequency for waveform generation. This is optional, and if not included, the starting frequency will be set to f_min. This exists so that EOB waveforms can be generated starting from a lower frequency than f_min.
- `mode_list` (*List[Tuple]*) – A list of waveform (ell, m) modes to include when generating the polarizations.
- `spin_conversion_phase` (*float = None*) – Value for phiRef when computing cartesian spins from bilby spins via bilby_to_lalsimulation_spins. The common convention is to use the value of the phase parameter here, which is also used in the spherical harmonics when combining the different modes. If spin_conversion_phase = None, this default behavior is adapted. For dingo, this convention for the phase parameter makes it impossible to treat the phase as an extrinsic parameter, since we can only account for the change of phase in the spherical harmonics when changing the phase (in order to also change the cartesian spins – specifically, to rotate the spins by phase in the sx-sy plane – one would need to recompute the modes, which is expensive). By setting spin_conversion_phase != None, we impose the convention to always use phase = spin_conversion_phase when computing the cartesian spins.

`generate_FD_modes_LO`(*parameters*)

Generate FD modes in the L0 frame.

Parameters

`parameters` (*dict*) – Dictionary of parameters for the waveform. For details see see self.generate_hplus_hcross.

Returns

- **hlm_fd** (*dict*) – Dictionary with (l,m) as keys and the corresponding FD modes in lal format as values.
- **iota** (*float*)

`generate_FD_waveform`(*parameters_gwsignal: Dict*) → Dict[str, ndarray]

Generate Fourier domain GW polarizations (h_plus, h_cross).

Parameters

`parameters_lal` – A tuple of parameters for the lalsimulation waveform generator

Returns

A dictionary of generated waveform polarizations

Return type

pol_dict

`generate_TD_modes_L0`(*parameters*)

Generate TD modes in the L0 frame.

Parameters

`parameters` (*dict*) – Dictionary of parameters for the waveform. For details see see self.generate_hplus_hcross.

Returns

- **hlm_td** (*dict*) – Dictionary with (l,m) as keys and the corresponding TD modes in lal format as values.
- **iota** (*float*)

generate_TD_modes_L0_conditioned_extra_time(*parameters*)

Generate TD modes in the L0 frame applying a conditioning routine which mimics the behaviour of the standard LALSimulation conditioning (https://lscsoft.docs.ligo.org/lalsuite/lalsimulation/_l_a_l_sim_inspiral_generator_conditioning_8c.html#ac78b5fcdabf8922a3ac479da20185c85)

Essentially, a new starting frequency is computed to have some extra cycles that will be tapered. Some extra buffer time is also added to ensure that the waveform at the requested starting frequency is not modified, while still having a tapered timeseries suited for clean FFT.

Parameters
parameters (*dict*) – Dictionary of parameters for the waveform. For details see self.generate_hplus_hcross.

Returns

- **hlm_td** (*dict*) – Dictionary with (l,m) as keys and the corresponding TD modes in lal format as values.
- **iota** (*float*)

generate_TD_waveform(*parameters_gwsignal: Dict*) → Dict[str, ndarray]

Generate time domain GW polarizations (h_plus, h_cross)

Parameters
parameters_gwsignal – A dict of parameters for the gwsignal waveform generator

Returns
A dictionary of generated waveform polarizations

Return type
pol_dict

generate_hplus_hcross_m(*parameters: Dict[str, float]*) → Dict[tuple, Dict[str, ndarray]]

Generate GW polarizations (h_plus, h_cross), separated into contributions from the different modes. This method is identical to self.generate_hplus_hcross, except that it generates the individual contributions of the modes to the polarizations and sorts these according to their transformation behavior (see below), instead of returning the overall sum.

This is useful in order to treat the phase as an extrinsic parameter. Instead of {"h_plus": hp, "h_cross": hc}, this method returns a dict in the form of {m: {"h_plus": hp_m, "h_cross": hc_m} for m in [-l_max,…,0,…,l_max]}. Each key m contains the contribution to the polarization that transforms according to exp(-1j * m * phase) under phase transformations (due to the spherical harmonics).

Note:

- pol_m[m] contains contributions of the m modes *and* and the -m modes. This is because the frequency domain (FD) modes have a positive frequency part which transforms as exp(-1j * m * phase), while the negative frequency part transforms as exp(+1j * m * phase). Typically, one of these dominates [e.g., the (2,2) mode is dominated by the negative frequency part and the (-2,2) mode is dominated by the positive frequency part] such that the sum of (l,|m|) and (l,-**|m|**) modes transforms approximately as exp(1j * **|m|** * phase), which is e.g. used for phase marginalization in bilby/lalinference. However, this is not exact. In this method we account for this effect, such that each contribution pol_m[m] transforms *exactly* as exp(-1j * m * phase).
- Phase shifts contribute in two ways: Firstly via the spherical harmonics, which we account for with the exp(-1j * m * phase) transformation. Secondly, the phase determines how the PE spins transform to cartesian spins, by rotating (sx,sy) by phase. This is *not* accounted for in this function. Instead, the phase for computing the cartesian spins is fixed to self.spin_conversion_phase (if not None). This effectively changes the PE parameters {phi_jl, phi_12} to parameters {phi_jl_prime, phi_12_prime}. For parameter estimation, a postprocessing operation can be applied to account

for this, {phi_jl_prime, phi_12_prime} -> {phi_jl, phi_12}. See also documentation of __init__ method for more information on self.spin_conversion_phase.

Differences to self.generate_hplus_hcross: - We don't catch errors yet TODO - We don't apply transforms yet TODO

Parameters
`parameters` (*dict*) – Dictionary of parameters for the waveform. For details see see self.generate_hplus_hcross.

Returns
pol_m – Dictionary with contributions to h_plus and h_cross, sorted by their transformation behaviour under phase shifts: {m: {"h_plus": hp_m, "h_cross": hc_m} for m in [-l_max,…,0,…,l_max]} Each contribution h_m transforms as exp(-1j * m * phase) under phase shifts (for fixed self.spin_conversion_phase, see above).

Return type
dict

`dingo.gw.waveform_generator.waveform_generator.`**`SEOBNRv4PHM_maximum_starting_frequency`**(*total_mass: float*, *fudge: float = 0.99*) → float

Given a total mass return the largest possible starting frequency allowed for SEOBNRv4PHM and similar effective-one-body models.

The intended use for this function is at the stage of designing a data set: after choosing a mass prior one can use it to figure out which prior samples would run into an issue when generating an EOB waveform, and tweak the parameters to reduce the number of failing configurations.

Parameters

- `total_mass` – Total mass in units of solar masses
- `fudge` – A fudge factor

Returns
The largest possible starting frequency in Hz

Return type
f_max_Hz

`class dingo.gw.waveform_generator.waveform_generator.`**`WaveformGenerator`**(*approximant: str*, *domain: Domain*, *f_ref: float*, *f_start: float | None = None*, *mode_list: List[Tuple] | None = None*, *transform=None*, *spin_conversion_phase=None*, ***kwargs*)

Bases: `object`

Generate polarizations using LALSimulation routines in the specified domain for a single GW coalescence given a set of waveform parameters.

Parameters

- **approximant** (*str*) – Waveform "approximant" string understood by lalsimulation This is defines which waveform model is used.
- **domain** (Domain) – Domain object that specifies on which physical domain the waveform polarizations will be generated, e.g. Fourier domain, time domain.
- **f_ref** (*float*) – Reference frequency for the waveforms
- **f_start** (*float*) – Starting frequency for waveform generation. This is optional, and if not included, the starting frequency will be set to f_min. This exists so that EOB waveforms can be generated starting from a lower frequency than f_min.
- **mode_list** (*List[Tuple]*) – A list of waveform (ell, m) modes to include when generating the polarizations.
- **spin_conversion_phase** (*float = None*) – Value for phiRef when computing cartesian spins from bilby spins via bilby_to_lalsimulation_spins. The common convention is to use the value of the phase parameter here, which is also used in the spherical harmonics when combining the different modes. If spin_conversion_phase = None, this default behavior is adapted. For dingo, this convention for the phase parameter makes it impossible to treat the phase as an extrinsic parameter, since we can only account for the change of phase in the spherical harmonics when changing the phase (in order to also change the cartesian spins – specifically, to rotate the spins by phase in the sx-sy plane – one would need to recompute the modes, which is expensive). By setting spin_conversion_phase != None, we impose the convention to always use phase = spin_conversion_phase when computing the cartesian spins.

generate_FD_modes_LO(*parameters*)

Generate FD modes in the L0 frame.

Parameters

parameters (*dict*) – Dictionary of parameters for the waveform. For details see see self.generate_hplus_hcross.

Returns

- **hlm_fd** (*dict*) – Dictionary with (l,m) as keys and the corresponding FD modes in lal format as values.
- **iota** (*float*)

generate_FD_waveform(*parameters_lal: Tuple*) → Dict[str, ndarray]

Generate Fourier domain GW polarizations (h_plus, h_cross).

Parameters

parameters_lal – A tuple of parameters for the lalsimulation waveform generator

Returns

A dictionary of generated waveform polarizations

Return type

pol_dict

generate_TD_modes_L0(*parameters*)

Generate TD modes in the L0 frame.

Parameters

parameters (*dict*) – Dictionary of parameters for the waveform. For details see see self.generate_hplus_hcross.

Returns

- **hlm_td** (*dict*) – Dictionary with (l,m) as keys and the corresponding TD modes in lal format as values.
- **iota** (*float*)

`generate_TD_waveform`(*parameters_lal: Tuple*) → Dict[str, ndarray]

Generate time domain GW polarizations (h_plus, h_cross)

Parameters
`parameters_lal` – A tuple of parameters for the lalsimulation waveform generator

Returns
A dictionary of generated waveform polarizations

Return type
pol_dict

`generate_hplus_hcross`(*parameters: Dict[str, float]*, *catch_waveform_errors=True*) → Dict[str, ndarray]

Generate GW polarizations (h_plus, h_cross).

If the generation of the lalsimulation waveform fails with an "Input domain error", we return NaN polarizations.

Use the domain, approximant, and mode_list specified in the constructor along with the waveform parameters to generate the waveform polarizations.

Parameters

- `parameters` (`Dict[str, float]`) – A dictionary of parameter names and scalar values. The parameter dictionary must include the following keys. For masses, spins, and distance there are multiple options.

  Mass: (mass_1, mass_2) or a pair of quantities from
  ((chirp_mass, total_mass), (mass_ratio, symmetric_mass_ratio))

  Spin:
  (a_1, a_2, tilt_1, tilt_2, phi_12, phi_jl) if precessing binary or (chi_1, chi_2) if the binary has aligned spins

  Reference frequency: f_ref at which spin vectors are defined Extrinsic:

  Distance: one of (luminosity_distance, redshift, comoving_distance) Inclination: theta_jn Reference phase: phase Geocentric time: geocent_time (GPS time)

  The following parameters are not required:
  Sky location: ra, dec, Polarization angle: psi

  Units:
  Masses should be given in units of solar masses. Distance should be given in megaparsecs (Mpc). Frequencies should be given in Hz and time in seconds. Spins should be dimensionless. Angles should be in radians.

- `catch_waveform_errors` (`bool`) – Whether to catch lalsimulation errors

Returns
A dictionary of generated waveform polarizations

Return type
wf_dict

generate_hplus_hcross_m(*parameters: Dict[str, float]*) → Dict[tuple, Dict[str, ndarray]]

Generate GW polarizations (h_plus, h_cross), separated into contributions from the different modes. This method is identical to self.generate_hplus_hcross, except that it generates the individual contributions of the modes to the polarizations and sorts these according to their transformation behavior (see below), instead of returning the overall sum.

This is useful in order to treat the phase as an extrinsic parameter. Instead of {"h_plus": hp, "h_cross": hc}, this method returns a dict in the form of {m: {"h_plus": hp_m, "h_cross": hc_m} for m in [-l_max,...,0,...,l_max]}. Each key m contains the contribution to the polarization that transforms according to exp(-1j * m * phase) under phase transformations (due to the spherical harmonics).

Note:

- pol_m[m] contains contributions of the m modes *and* and the -m modes. This is because the frequency domain (FD) modes have a positive frequency part which transforms as exp(-1j * m * phase), while the negative frequency part transforms as exp(+1j * m * phase). Typically, one of these dominates [e.g., the (2,2) mode is dominated by the negative frequency part and the (-2,2) mode is dominated by the positive frequency part] such that the sum of (l,|m|) and (l,-**|m|**) modes transforms approximately as exp(1j * **|m|** * phase), which is e.g. used for phase marginalization in bilby/lalinference. However, this is not exact. In this method we account for this effect, such that each contribution pol_m[m] transforms *exactly* as exp(-1j * m * phase).
- Phase shifts contribute in two ways: Firstly via the spherical harmonics, which we account for with the exp(-1j * m * phase) transformation. Secondly, the phase determines how the PE spins transform to cartesian spins, by rotating (sx,sy) by phase. This is *not* accounted for in this function. Instead, the phase for computing the cartesian spins is fixed to self.spin_conversion_phase (if not None). This effectively changes the PE parameters {phi_jl, phi_12} to parameters {phi_jl_prime, phi_12_prime}. For parameter estimation, a postprocessing operation can be applied to account for this, {phi_jl_prime, phi_12_prime} -> {phi_jl, phi_12}. See also documentation of __init__ method for more information on self.spin_conversion_phase.

Differences to self.generate_hplus_hcross: - We don't catch errors yet TODO - We don't apply transforms yet TODO

Parameters

parameters (*dict*) – Dictionary of parameters for the waveform. For details see see self.generate_hplus_hcross.

Returns

pol_m – Dictionary with contributions to h_plus and h_cross, sorted by their transformation behaviour under phase shifts: {m: {"h_plus": hp_m, "h_cross": hc_m} for m in [-l_max,...,0,...,l_max]} Each contribution h_m transforms as exp(-1j * m * phase) under phase shifts (for fixed self.spin_conversion_phase, see above).

Return type

dict

setup_mode_array(*mode_list: List[Tuple]*) → Dict

Define a mode array to select waveform modes to include in the polarizations from a list of modes.

Parameters

mode_list (*a list of (ell, m) modes*) –

Returns

A lal parameter dictionary

Return type

lal_params

property spin_conversion_phase

dingo.gw.waveform_generator.waveform_generator.**generate_waveforms_parallel**(*waveform_generator: WaveformGenerator*, *parameter_samples: DataFrame*, *pool: Pool | None = None*) → Dict[str, ndarray]

Generate a waveform dataset, optionally in parallel.

Parameters

- **waveform_generator** (WaveformGenerator) – A WaveformGenerator instance
- **parameter_samples** (*pd.DataFrame*) – Intrinsic parameter samples
- **pool** (*multiprocessing.Pool*) – Optional pool of workers for parallel generation

Returns

A dictionary of all generated polarizations stacked together

Return type

polarizations

dingo.gw.waveform_generator.waveform_generator.**generate_waveforms_task_func**(*args: Tuple*, *waveform_generator: WaveformGenerator*) → Dict[str, ndarray]

Picklable wrapper function for parallel waveform generation.

Parameters

- **args** – A tuple of (index, pandas.core.series.Series)
- **waveform_generator** – A WaveformGenerator instance

Return type

The generated waveform polarization dictionary

dingo.gw.waveform_generator.waveform_generator.**sum_contributions_m**(*x_m*, *phase_shift=0.0*)

Sum the contributions over m-components, optionally introducing a phase shift.

dingo.gw.waveform_generator.wfg_utils module

dingo.gw.waveform_generator.wfg_utils.**get_polarizations_from_fd_modes_m**(*hlm_fd*, *iota*, *phase*)

dingo.gw.waveform_generator.wfg_utils.**get_starting_frequency_for_SEOBRNRv5_conditioning**(*parameters*)

Compute starting frequency needed for having 3 extra cycles for tapering the TD modes. It returns the needed quantities to apply the standard LALSimulation conditioning routines to the TD modes.

Parameters

parameters (*dict*) – Dictionary of parameters suited for GWSignal (obtained with NewInterfaceWaveformGenerator._convert_parameters)

Returns

- **f_min** (*float*) – Waveform starting frequency
- **f_start** (*float*) – New waveform starting frequency
- **extra_time** (*float*) – Extra time to take care of situations where the frequency is close to merger
- **original_f_min** (*float*) – Initial waveform starting frequency
- **f_isco** (*float*) – ISCO frequency

`dingo.gw.waveform_generator.wfg_utils.`**`get_tapering_window_for_complex_time_series`**`(`*h*, *tapering_flag: int = 1*`)`

Get window for tapering of a complex time series from the lal backend. This is done by tapering the time series with lal, and dividing tapered output by untapered input. lal does not support tapering of complex time series objects, so as a workaround we taper only the real part of the array and extract the window based on this.

Parameters

- **`h`** – complex lal time series object
- **`tapering_flag`** (`int = 1`) –

  Flag for tapering. See e.g. lines 2773-2777 in
  https://lscsoft.docs.ligo.org/lalsuite/lalsimulation/ _l_a_l_sim_inspiral_waveform_taper_8c_source.html#l00222

  tapering_flag = 1 corresponds to LAL_SIM_INSPIRAL_TAPER_START

Returns
window – Array of length h.data.length, with the window used for tapering.

Return type
np.ndarray

`dingo.gw.waveform_generator.wfg_utils.`**`linked_list_modes_to_dict_modes`**`(`*hlm_ll*`)`

Convert linked list of modes into dictionary with keys (l,m).

`dingo.gw.waveform_generator.wfg_utils.`**`taper_td_modes_for_SEOBRNRv5_extra_time`**`(`*h*, *extra_time*, *f_min*, *original_f_min*, *f_isco*`)`

Apply standard tapering procedure mimicking LALSimulation routine (https://lscsoft.docs.ligo.org/lalsuite/lalsimulation/_l_a_l_sim_inspiral_generator_conditioning_8c.html#ac78b5fcdabf8922a3ac479da20185c85)

Parameters

- **`h`** – complex gwpy TimeSeries object
- **`extra_time`** (`float`) – Extra time to take care of situations where the frequency is close to merger
- **`f_min`** (`float`) – Starting frequency employed in waveform generation
- **`original_f_min`** (`float`) – Initial starting frequency requested by the user
- **`f_isco`** – ISCO frequency

Returns
complex lal timeseries object

Return type
h_return

dingo.gw.waveform_generator.wfg_utils.**taper_td_modes_in_place**(*hlm_td*, *tapering_flag: int = 1*)

Taper the time domain modes in place.

Parameters

- **hlm_td** (*dict*) – Dictionary with (l,m) keys and the complex lal time series objects for the corresponding modes.
- **tapering_flag** (*int = 1*) –

  Flag for tapering. See e.g. lines 2773-2777 in
  https://lscsoft.docs.ligo.org/lalsuite/lalsimulation/ _l_a_l_sim_inspiral_waveform_taper_8c_source.html#l00222

  tapering_flag = 1 corresponds to LAL_SIM_INSPIRAL_TAPER_START

dingo.gw.waveform_generator.wfg_utils.**td_modes_to_fd_modes**(*hlm_td*, *domain*)

Transform dict of td modes to dict of fd modes via FFT. The td modes are expected to be tapered.

Parameters

- **hlm_td** (*dict*) – Dictionary with (l,m) keys and the complex lal time series objects for the corresponding tapered modes.
- **domain** (dingo.gw.domains.FrequencyDomain) – Target domain after FFT.

Returns

hlm_fd – Dictionary with (l,m) keys and numpy arrays with the corresponding modes as values.

Return type

dict

Module contents

Submodules

dingo.gw.SVD module

class dingo.gw.SVD.**ApplySVD**(*svd_basis: SVDBasis*, *inverse: bool = False*)

Bases: object

Transform operator for applying an SVD compression / decompression.

Parameters

- **svd_basis** (SVDBasis) –
- **inverse** (*bool*) – Whether to apply for the forward (compression) or inverse (decompression) transform. Default: False.

class dingo.gw.SVD.**SVDBasis**(*file_name=None*, *dictionary=None*)

Bases: *DingoDataset*

For constructing, provide either file_name, or dictionary containing data and settings entries, or neither.

Parameters

- **file_name** (*str*) – HDF5 file containing a dataset
- **dictionary** (*dict*) – Contains settings and data entries. The data keys should be the same as save_keys

- **data_keys** (*list*) – Variables that should be saved / loaded. This allows for class to store additional variables beyond those that are saved. Typically, this list would be provided by any subclass.

compress(*data: ndarray*)

Convert from data (e.g., frequency series) to compressed representation in terms of basis coefficients.

Parameters
data (*np.ndarray*) –

Return type
array of basis coefficients

compute_test_mismatches(*data: ndarray*, *parameters: DataFrame | None = None*, *increment: int = 50*, *verbose: bool = False*)

Test SVD basis by computing mismatches of compressed / decompressed data against original data. Results are saved as a DataFrame.

Parameters

- **data** (*np.ndarray*) – Array of data sets to validate against.
- **parameters** (*pd.DataFrame*) – Optional labels for the data sets. This is useful for checking performance on particular regions of the parameter space.
- **increment** (*int*) – Specifies SVD truncations for computing mismatches. E.g., increment = 50 means that the SVD will be truncated at size [50, 100, 150, …, len(data)].
- **verbose** (*bool*) – Whether to print summary statistics.

dataset_type = 'svd_basis'

decompress(*coefficients: ndarray*)

Convert from basis coefficients back to raw data representation.

Parameters
coefficients (*np.ndarray*) – Array of basis coefficients

Return type
array of decompressed data

from_dictionary(*dictionary: dict*)

Load the SVD basis from a dictionary.

Parameters
dictionary (*dict*) – The dictionary should contain at least a 'V' key, and optionally an 's' key.

from_file(*filename*)

Load the SVD basis from a HDF5 file.

Parameters
filename (*str*) –

generate_basis(*training_data: ndarray*, *n: int*, *method: str = 'random'*)

Generate the SVD basis from training data and store it.

The SVD decomposition takes

training_data = U @ diag(s) @ Vh

where U and Vh are unitary.

Parameters

- **training_data** (*np.ndarray*) – Array of waveform data on the physical domain
- **n** (*int*) – Number of basis elements to keep. n=0 keeps all basis elements.
- **method** (*str*) – Select SVD method, 'random' or 'scipy'

print_validation_summary()

Print a summary of the validation mismatches.

dingo.gw.domains module

class dingo.gw.domains.**Domain**

Bases: ABC

Defines the physical domain on which the data of interest live.

This includes a specification of the bins or points, and a few additional properties associated with the data.

abstract property domain_dict

Enables to rebuild the domain via calling build_domain(domain_dict).

abstract property duration: float

Waveform duration in seconds.

abstract property f_max: float

The maximum frequency [Hz] is set to half the sampling rate.

abstract property max_idx: int

abstract property min_idx: int

abstract property noise_std: float

Standard deviation of the whitened noise distribution

abstract property sampling_rate: float

The sampling rate of the data [Hz].

abstract time_translate_data(*data*, *dt*) → ndarray

Time translate strain data by dt seconds.

abstract update(*new_settings: dict*)

class dingo.gw.domains.**FrequencyDomain**(*f_min: float*, *f_max: float*, *delta_f: float*, *window_factor: float | None = None*)

Bases: *Domain*

Defines the physical domain on which the data of interest live.

The frequency bins are assumed to be uniform between [0, f_max] with spacing delta_f. Given a finite length of time domain data, the Fourier domain data starts at a frequency f_min and is zero below this frequency. window_kwargs specify windowing used for FFT to obtain FD data from TD data in practice.

static add_phase(*data*, *phase*)

Add a (frequency-dependent) phase to a frequency series. Allows for batching, as well as additional channels (such as detectors). Accounts for the fact that the data could be a complex frequency series or real and imaginary parts.

Convention: the phase phi(f) is defined via exp(- 1j * phi(f)).

Parameters

- **data** (*Union[np.array, torch.Tensor]*) –
- **phase** (*Union[np.array, torch.Tensor]*) –

Return type

New array or tensor of the same shape as data.

property delta_f: float

The frequency spacing of the uniform grid [Hz].

property domain_dict

Enables to rebuild the domain via calling build_domain(domain_dict).

property duration: float

Waveform duration in seconds.

property f_max: float

The maximum frequency [Hz] is typically set to half the sampling rate.

property f_min: float

The minimum frequency [Hz].

property frequency_mask: ndarray

Mask which selects frequency bins greater than or equal to the starting frequency

property frequency_mask_length: int

Number of samples in the subdomain domain[frequency_mask].

get_sample_frequencies_astype(*data*)

Returns a 1D frequency array compatible with the last index of data array.

Decides whether array is numpy or torch tensor (and cuda vs cpu), and whether it contains the leading zeros below f_min.

Parameters

data (*Union[np.array, torch.Tensor]*) – Sample data

Return type

frequency array compatible with last index

property max_idx

property min_idx

property noise_std: float

Standard deviation of the whitened noise distribution.

To have noise that comes from a multivariate *unit* normal distribution, you must divide by this factor. In practice, this means dividing the whitened waveforms by this.

TODO: This description makes some assumptions that need to be clarified. Windowing of TD data; tapering window has a slope -> reduces power only for noise, but not for the signal which is in the main part unaffected by the taper

property sample_frequencies

property sample_frequencies_torch

property sample_frequencies_torch_cuda

property sampling_rate: float

The sampling rate of the data [Hz].

set_new_range(*f_min: float | None = None*, *f_max: float | None = None*)

Set a new range [f_min, f_max] for the domain. This is only allowed if the new range is contained within the old one.

time_translate_data(*data*, *dt*)

Time translate frequency-domain data by dt. Time translation corresponds (in frequency domain) to multiplication by

$$\exp(-2\pi i\, f\, dt).$$

This method allows for multiple batch dimensions. For torch.Tensor data, allow for either a complex or a (real, imag) representation.

Parameters

- **data** (*array-like (numpy, torch)*) – Shape (B, C, N), where
  - B corresponds to any dimension >= 0,
  - C is either absent (for complex data) or has dimension >= 2 (for data represented as real and imaginary parts), and
  - N is either len(self) or len(self)-self.min_idx (for truncated data).
- **dt** (*torch tensor, or scalar (if data is numpy)*) – Shape (B)

Return type

Array-like of the same form as data.

update(*new_settings: dict*)

Update the domain with new settings. This is only allowed if the new settings are "compatible" with the old ones. E.g., f_min should be larger than the existing f_min.

Parameters

new_settings (*dict*) – Settings dictionary. Must contain a subset of the keys contained in domain_dict.

update_data(*data: ndarray*, *axis: int = -1*, *low_value: float = 0.0*)

Adjusts data to be compatible with the domain:

- Below f_min, it sets the data to low_value (typically 0.0 for a waveform, but for a PSD this might be a large value).
- Above f_max, it truncates the data array.

Parameters

- **data** (*np.ndarray*) – Data array
- **axis** (*int*) – Which data axis to apply the adjustment along.
- **low_value** (*float*) – Below f_min, set the data to this value.

Returns

The new data array.

Return type

np.ndarray

property window_factor

`class dingo.gw.domains.`**PCADomain**

Bases: *Domain*

TODO

property noise_std: float

Standard deviation of the whitened noise distribution.

To have noise that comes from a multivariate *unit* normal distribution, you must divide by this factor. In practice, this means dividing the whitened waveforms by this.

In the continuum limit in time domain, the standard deviation of white noise would at each point go to infinity, hence the delta_t factor.

`class dingo.gw.domains.`**TimeDomain**(*time_duration: float*, *sampling_rate: float*)

Bases: *Domain*

Defines the physical time domain on which the data of interest live.

The time bins are assumed to be uniform between [0, duration] with spacing 1 / sampling_rate. window_factor is used to compute noise_std().

property delta_t: float

The size of the time bins

property domain_dict

Enables to rebuild the domain via calling build_domain(domain_dict).

property duration: float

Waveform duration in seconds.

property f_max: float

The maximum frequency [Hz] is typically set to half the sampling rate.

property max_idx: int

property min_idx: int

property noise_std: float

Standard deviation of the whitened noise distribution.

To have noise that comes from a multivariate *unit* normal distribution, you must divide by this factor. In practice, this means dividing the whitened waveforms by this.

In the continuum limit in time domain, the standard deviation of white noise would at each point go to infinity, hence the delta_t factor.

property sampling_rate: float

The sampling rate of the data [Hz].

time_translate_data(*data*, *dt*) → ndarray

Time translate strain data by dt seconds.

`dingo.gw.domains.`**build_domain**(*settings: Dict*) → *Domain*

Instantiate a domain class from settings.

Parameters

settings (*dict*) – Dicionary with 'type' key denoting the type of domain, and keys corresponding to the kwargs needed to construct the Domain.

Return type
A Domain instance of the correct type.

dingo.gw.domains.**build_domain_from_model_metadata**(*model_metadata*) → *Domain*

Instantiate a domain class from settings of model.

Parameters
model_metadata (*dict*) – model metadata containing information to build the domain typically obtained from the model.metadata attribute

Return type
A Domain instance of the correct type.

dingo.gw.download_strain_data module

dingo.gw.download_strain_data.**download_event_data_in_FD**(*detectors*, *time_event*, *time_segment*, *time_buffer*, *window*, *num_segments_psd=128*)

Download event data in frequency domain. This includes strain data for the event at GPS time t_event as well as the correcponding ASD.

Parameters

- **detectors** (*list*) – list of detectors specified via strings
- **time_event** (*float*) – GPS time of the event
- **time_segment** (*float*) – length of the strain segment, in seconds
- **time_buffer** (*float*) – specifies buffer time after time_event, in seconds
- **window** (*Union(np.ndarray, dict)*) – Window used for Fourier transforming the event strain data. Either provided directly as np.ndarray, or as dict in which case the window is generated by window = dingo.gw.gwutils.get_window(**window).
- **num_segments_psd** (*int = 128*) – number of segments used for PSD generation

dingo.gw.download_strain_data.**download_strain_data_in_FD**(*det*, *time_event*, *time_segment*, *time_buffer*, *window*)

Download strain data for a GW event at GPS time time_event. The segment is time_segment seconds long, including time_buffer seconds after the event. The strain is Fourier transformed, the frequency domain strain is then time shifted by time_buffer, such that the event occurs at t=0.

Parameters

- **det** (*str*) – detector
- **time_event** (*float*) – GPS time of the event
- **time_segment** (*float*) – length of the strain segment, in seconds
- **time_buffer** (*float*) – specifies buffer time after time_event, in seconds
- **window** (*Union(np.ndarray, dict)*) – Window used for Fourier transforming the event strain data. Either provided directly as np.ndarray, or as dict in which case the window is generated by window = dingo.gw.gwutils.get_window(**window).

Returns
event_strain – array with the frequency domain strain

Return type
np.array

`dingo.gw.download_strain_data.`**`estimate_single_psd`**`(`*time_start*, *time_segment*, *window*, *f_s=4096*, *num_segments: int = 128*, *det=None*, *channel=None*`)`

Download strain data and generate a PSD based on these. Use num_segments of length time_segment, starting at GPS time time_start. If no channel is specified, GWOSC is used to download the data.

Parameters

- **time_start** (*float*) – start GPS time for PSD estimation
- **time_segment** (*float*) – time for a single segment for PSD information, in seconds
- **window** (*Union(np.ndarray, dict)*) – Window used for PSD generation, needs to be the same as used for Fourier transform of event strain data. Either provided directly as np.ndarray, or as dict in which case the window is generated by window = dingo.gw.gwutils.get_window(**window).
- **num_segments** (*int = 256*) – number of segments used for PSD generation
- **det** (*str*) – If provided, will download data from GWOSC using TimeSeries.fetch_open_data(). Mutually exclusive with 'channel'.
- **channel** (*str*) – If provided, will download the data from the channel instead of gwosc using TimeSeries.get()

Returns
psd – array of psd

Return type
np.array

dingo.gw.gwutils module

`dingo.gw.gwutils.`**`get_extrinsic_prior_dict`**`(`*extrinsic_prior*`)`

Build dict for extrinsic prior by starting with default_extrinsic_dict, and overwriting every element for which extrinsic_prior is not default. TODO: Move to dingo.gw.prior.py?

`dingo.gw.gwutils.`**`get_mismatch`**`(`*a*, *b*, *domain*, *asd_file=None*`)`

Mistmatch is 1 - overlap, where overlap is defined by inner(a, b) / sqrt(inner(a, a) * inner(b, b)). See e.g. Eq. (44) in https://arxiv.org/pdf/1106.1021.pdf.

Parameters

- **a** –
- **b** –
- **domain** –
- **asd_file** –

`dingo.gw.gwutils.`**`get_standardization_dict`**`(`*extrinsic_prior_dict*, *wfd*, *selected_parameters*, *transform=None*`)`

Calculates the mean and standard deviation of parameters. This is needed for standardizing neural-network input and output.

Parameters

- **extrinsic_prior_dict** (*dict*) –
- **wfd** (*WaveformDataset*) –

- **selected_parameters** (*list[str]*) – List of parameters for which to estimate standardization factors.
- **transform** (*Transform*) – Operator that will generate samples for parameters contained in selected_parameters that are not contained in the intrinsic or extrinsic prior. (E.g., H1_time, L1_time_proxy)

dingo.gw.gwutils.**get_window**(*window_kwargs*)

Compute window from window_kwargs.

dingo.gw.gwutils.**get_window_factor**(*window*)

Compute window factor. If window is not provided as array or tensor but as window_kwargs, first build the window.

dingo.gw.injection module

class dingo.gw.injection.**GWSignal**(*wfg_kwargs: dict*, *wfg_domain:* FrequencyDomain, *data_domain:* FrequencyDomain, *ifo_list: list*, *t_ref: float*)

Bases: object

Base class for generating gravitational wave signals in interferometers. Generates waveform polarizations based on provided parameters, and then projects to detectors.

Includes option for whitening the signal based on a provided ASD.

Parameters

- **wfg_kwargs** (*dict*) – Waveform generator parameters [approximant, f_ref, and (optionally) f_start].
- **wfg_domain** (FrequencyDomain) – Domain used for waveform generation. This can potentially deviate from the final domain, having a wider frequency range needed for waveform generation.
- **data_domain** (FrequencyDomain) – Domain object for final signal.
- **ifo_list** (*list*) – Names of interferometers for projection.
- **t_ref** (*float*) – Reference time that specifies ifo locations.

property **asd**

Amplitude spectral density.

Either a single array, a dict (for individual interferometers), or an ASDDataset, from which random ASDs are drawn.

property **calibration_marginalization_kwargs**

Dictionary with the following keys:

calibration_envelope
Dictionary of the form {“H1”: filepath, “L1”: filepath, . . . } with locations of lookup tables for the calibration uncertainty curves.

num_calibration_nodes
Number of nodes for the calibration model.

num_calibration_curves
Number of calibration curves to use in marginalization.

signal(*theta*)

Compute the GW signal for parameters theta.

Step 1: Generate polarizations Step 2: Project polarizations onto detectors; optionally (depending on self.whiten) whiten and scale.

Parameters

theta (*dict*) – Signal parameters. Includes intrinsic parameters to be passed to waveform generator, and extrinsic parameters for detector projection.

Returns

keys:

waveform: GW strain signal for each detector. extrinsic_parameters: {} parameters: waveform parameters asd (if set): amplitude spectral density for each detector

Return type

dict

signal_m(*theta*)

Compute the GW signal for parameters theta. Same as self.signal(theta) method, but it does not sum the contributions of the individual modes, and instead returns a dict {m: pol_m for m in [-l_max,…,0,…,l_max]} where each contribution pol_m transforms as exp(-1j * m * phase_shift) under phase shifts.

Step 1: Generate polarizations Step 2: Project polarizations onto detectors;

optionally (depending on self.whiten) whiten and scale.

Parameters

theta (*dict*) – Signal parameters. Includes intrinsic parameters to be passed to waveform generator, and extrinsic parameters for detector projection.

Returns

keys:

waveform:

GW strain signal for each detector, with individual contributions {m: pol_m for m in [-l_max,…,0,…,l_max]}

extrinsic_parameters: {} parameters: waveform parameters asd (if set): amplitude spectral density for each detector

Return type

dict

property whiten

Bool specifying whether to whiten (and scale) generated signals.

class dingo.gw.injection.**Injection**(*prior*, ***gwsignal_kwargs*)

Bases: *GWSignal*

Produces injections of signals (with random or specified parameters) into stationary Gaussian noise. Output is not whitened.

Parameters

- **prior** (*PriorDict*) – Prior used for sampling random parameters.
- **gwsignal_kwargs** – Arguments to be passed to GWSignal base class.

classmethod from_posterior_model_metadata(*metadata*)

Instantiate an Injection based on a posterior model. The prior, waveform settings, etc., will all be consistent with what the model was trained with.

Parameters
metadata (*dict*) – Dict which you can get via PosteriorModel.metadata

injection(*theta*)

Generate an injection based on specified parameters.

This is a signal + noise consistent with the amplitude spectral density in self.asd. If self.asd is an ASDDataset, then it uses a random ASD from this dataset.

Data are not whitened.

Parameters
theta (*dict*) – Parameters used for injection.

Returns

keys:
waveform: data (signal + noise) in each detector extrinsic_parameters: {} parameters: waveform parameters asd (if set): amplitude spectral density for each detector

Return type
dict

random_injection()

Generate a random injection.

This is a signal + noise consistent with the amplitude spectral density in self.asd. If self.asd is an ASDDataset, then it uses a random ASD from this dataset.

Data are not whitened.

Returns

keys:
waveform: data (signal + noise) in each detector extrinsic_parameters: {} parameters: waveform parameters asd (if set): amplitude spectral density for each detector

Return type
dict

dingo.gw.likelihood module

class dingo.gw.likelihood.**StationaryGaussianGWLikelihood**(*wfg_kwargs*, *wfg_domain*, *data_domain*, *event_data*, *t_ref=None*, *time_marginalization_kwargs=None*, *phase_marginalization_kwargs=None*, *calibration_marginalization_kwargs=None*, *phase_grid=None*)

Bases: *GWSignal*, *Likelihood*

Implements GW likelihood for stationary, Gaussian noise.

Parameters

- **wfg_kwargs** (*dict*) – Waveform generator parameters (at least approximant and f_ref).

- **wfg_domain** (`dingo.gw.domains.Domain`) – Domain used for waveform generation. This can potentially deviate from the final domain, having a wider frequency range needed for waveform generation.
- **data_domain** (`dingo.gw.domains.Domain`) – Domain object for event data.
- **event_data** (*dict*) – GW data. Contains strain data in event_data["waveforms"] and asds in event_data["asds"].
- **t_ref** (*float*) – Reference time; true geocent time for GW is t_ref + theta["geocent_time"].
- **time_marginalization_kwargs** (*dict*) – Time marginalization parameters. If None, no time marginalization is used.
- **calibration_marginalization_kwargs** (*dict*) – Calibration marginalization parameters. If None, no calibration marginalization is used.
- **phase_marginalization_kwargs** (*dict*) – Phase marginalization parameters. If None, no phase marginalization is used.

d_inner_h_complex(*theta*)

Calculate the complex inner product (d | h(theta)) between the stored data d and a simulated waveform with given parameters theta.

Parameters
theta (*dict*) – Parameters at which to evaluate h.

Returns
complex

Return type
Inner product

d_inner_h_complex_multi(*theta: DataFrame*, *num_processes: int = 1*) → ndarray

Calculate the complex inner product (d | h(theta)) between the stored data d and a simulated waveform with given parameters theta. Works with multiprocessing.

Parameters

- **theta** (*pd.DataFrame*) – Parameters at which to evaluate h.
- **num_processes** (*int*) – Number of parallel processes to use.

Returns
complex

Return type
Inner product

initialize_time_marginalization(*t_lower*, *t_upper*, *n_fft=1*)

Initialize time marginalization. Time marginalization can be performed via FFT, which is super fast. However, this limits the time resolution to delta_t = 1/self.data_domain.f_max. In order to allow for a finer time resolution we compute the time marginalized likelihood n_fft via FFT on a grid of n_fft different time shifts [0, delta_t, 2*delta_t, . . . , (n_fft-1)*delta_t] and average over the time shifts. The effective time resolution is thus

delta_t_eff = delta_t / n_fft = 1 / (f_max * n_fft).

Note: Time marginalization in only implemented for uniform time priors.

Parameters

- **t_lower** (*float*) – Lower time bound of the uniform time prior.

- **t_upper** (*float*) – Upper time bound of the uniform time prior.
- **n_fft** (*int = 1*) – Size of grid for FFT for time marginalization.

log_likelihood(*theta*)

log_likelihood_phase_grid(*theta*, *phases=None*)

dingo.gw.likelihood.**build_stationary_gaussian_likelihood**(*metadata*, *event_dataset=None*, *time_marginalization_kwargs=None*)

Build a StationaryGaussianLikelihoodBBH object from the metadata.

Parameters

- **metadata** (*dict*) – Metadata from stored dingo parameter samples file. Typially accessed via pd.read_pickle(/path/to/dingo-output.pkl).metadata.
- **event_dataset** (*str = None*) – Path to event dataset for caching. If None, don't cache.
- **time_marginalization_kwargs** (*dict = None*) – Forwarded to the likelihood.

Returns
likelihood – likelihood object

Return type
StationaryGaussianGWLikelihood

dingo.gw.likelihood.**get_wfg**(*wfg_kwargs*, *data_domain*, *frequency_range=None*)

Set up waveform generator from wfg_kwargs. The domain of the wfg is primarily determined by the data domain, but a new (larger) frequency range can be specified if this is necessary for the waveforms to be generated successfully (e.g., for EOB waveforms which require a sufficiently small f_min and sufficiently large f_max).

Parameters

- **wfg_kwargs** (*dict*) – Waveform generator parameters.
- **data_domain** (dingo.gw.domains.Domain) – Domain of event data, with bounds determined by likelihood integral.
- **frequency_range** (*dict = None*) – Frequency range for waveform generator. If None, that of data domain is used, which corresponds to the bounds of the likelihood integral. Possible keys:

  'f_start': float
  Frequency at which to start the waveform generation. Overrides f_start in metadata["model"]["dataset_settings"]["waveform_generator"].

  'f_end': float
  Frequency at which to start the waveform generation.

Returns
wfg – Waveform generator object.

Return type
dingo.gw.waveform_generator.WaveformGenerator

dingo.gw.likelihood.**inner_product**(*a*, *b*, *min_idx=0*, *delta_f=None*, *psd=None*)

Compute the inner product between two complex arrays. There are two modes: either, the data a and b are not whitened, in which case delta_f and the psd must be provided. Alternatively, if delta_f and psd are not provided, the data a and b are assumed to be whitened already (i.e., whitened as d -> d * sqrt(4 delta_f / psd)).

Note: sum is only taken along axis 0 (which is assumed to be the frequency axis), while other axes are preserved. This is e.g. useful when evaluating kappa2 on a phase grid.

Parameters

- **a** (*np.ndaarray*) – First array with frequency domain data.
- **b** (*np.ndaarray*) – Second array with frequency domain data.
- **min_idx** (*int = 0*) – Truncation of likelihood integral, index of lowest frequency bin to consider.
- **delta_f** (*float*) – Frequency resolution of the data. If None, a and b are assumed to be whitened and the inner product is computed without further whitening.
- **psd** (*np.ndarray = None*) – PSD of the data. If None, a and b are assumed to be whitened and the inner product is computed without further whitening.

Returns
inner_product

Return type
float

`dingo.gw.likelihood.`**`inner_product_complex`**`(`*a*, *b*, *min_idx=0*, *delta_f=None*, *psd=None*`)`

Same as inner product, but without taking the real part. Retaining phase information is useful for the phase-marginalized likelihood. For further documentation see inner_product function.

`dingo.gw.likelihood.`**`main`**`()`

dingo.gw.ls_cli module

`dingo.gw.ls_cli.`**`determine_dataset_type`**`(`*file_name*`)`

`dingo.gw.ls_cli.`**`ls`**`()`

dingo.gw.prior module

`class dingo.gw.prior.`**`BBHExtrinsicPriorDict`**`(`*dictionary=None*, *filename=None*, *aligned_spin=False*, *conversion_function=None*`)`

Bases: `BBHPriorDict`

This class is the same as BBHPriorDict except that it does not require mass parameters.

It also contains a method for estimating the standardization parameters.

TODO:

- Add support for zenith/azimuth
- Defaults?

Initialises a Prior set for Binary Black holes

Parameters

- **dictionary** (*dict, optional*) – See superclass
- **filename** (*str, optional*) – See superclass
- **conversion_function** (*func*) – Function to convert between sampled parameters and constraints. By default this generates many additional parameters, see BBHPriorDict.default_conversion_function

`default_conversion_function`(*sample*)

Default parameter conversion function for BBH signals.

This generates: - the parameters passed to source.lal_binary_black_hole - all mass parameters

It does not generate: - component spins - source-frame parameters

Parameters
`sample` (*dict*) – Dictionary to convert

Returns
sample – Same as input

Return type
dict

`mean_std`(*keys=[]*, *sample_size=50000*, *force_numerical=False*)

Calculate the mean and standard deviation over the prior.

Parameters

- `keys` (*list(str)*) – A list of desired parameter names
- `sample_size` (*int*) – For nonanalytic priors, number of samples to use to estimate the result.
- `force_numerical` (*bool (False)*) – Whether to force a numerical estimation of result, even when analytic results are available (useful for testing)
- `deviations.` (*Returns dictionaries for the means and standard*) –
- `TODO` (*Fix for constrained priors. Shouldn't be an issue for extrinsic parameters.*) –

`dingo.gw.prior.build_prior_with_defaults`(*prior_settings: Dict[str, str]*)

Generate BBHPriorDict based on dictionary of prior settings, allowing for default values.

Parameters

- `prior_settings` (*Dict*) – A dictionary containing prior definitions for intrinsic parameters Allowed values for each parameter are:
  - 'default' to use a default prior
  - **a string for a custom prior, e.g.,**
    "Uniform(minimum=10.0, maximum=80.0, name=None, latex_label=None, unit=None, boundary=None)"
- `a` (*Depending on the particular prior choices the dimensionality of*) –
- `vary.` (*parameter sample obtained from the returned GWPriorDict will*) –

`dingo.gw.prior.split_off_extrinsic_parameters`(*theta*)

Split theta into intrinsic and extrinsic parameters.

Parameters
`theta` (*dict*) – BBH parameters. Includes intrinsic parameters to be passed to waveform generator, and extrinsic parameters for detector projection.

Returns

- **theta_intrinsic** (*dict*) – BBH intrinsic parameters.
- **theta_extrinsic** (*dict*) – BBH extrinsic parameters.

dingo.gw.result module

`class dingo.gw.result.Result(**kwargs)`

Bases: `Result`

A dataset class to hold a collection of gravitational-wave parameter samples and perform various operations on them.

Compared to the base class, this class implements the domain, prior, and likelihood. It also includes a method for sampling the binary reference phase parameter based on the other parameters and the likelihood.

Attributes:

samples
[pd.Dataframe] Contains parameter samples, as well as (possibly) log_prob, log_likelihood, weights, log_prior, delta_log_prob_target.

domain
[Domain] The domain of the data (e.g., FrequencyDomain), needed for calculating likelihoods.

prior
[PriorDict] The prior distribution, used for importance sampling.

likelihood
[Likelihood] The Likelihood object, needed for importance sampling.

context
[dict] Context data from which the samples were produced (e.g., strain data, ASDs).

metadata
[dict] Metadata inherited from the Sampler object. This describes the neural networks and sampling settings used.

event_metadata
[dict] Metadata for the event analyzed, including time, data conditioning, channel, and detector information.

log_evidence
[float] Calculated log(evidence) after importance sampling.

log_evidence_std
[float (property)] Standard deviation of the log(evidence)

effective_sample_size, n_eff
[float (property)] Number of effective samples, (sum_i w_i)^2 / sum_i w_i^2

sample_efficiency
[float (property)] Number of effective samples / Number of samples

synthetic_phase_kwargs
[dict] kwargs describing the synthetic phase sampling.

For constructing, provide either file_name, or dictionary containing data and settings entries, or neither.

Parameters

- **file_name** (*str*) – HDF5 file containing a dataset
- **dictionary** (*dict*) – Contains settings and data entries. The data keys should be the same as save_keys

- `data_keys` (*list*) – Variables that should be saved / loaded. This allows for class to store additional variables beyond those that are saved. Typically, this list would be provided by any subclass.

`property approximant`

`property calibration_marginalization_kwargs`

`dataset_type = 'gw_result'`

`property f_ref`

`get_samples_bilby_phase()`

Convert the spin angles phi_jl and theta_jn to account for a difference in phase definition compared to Bilby.

Returns
Samples

Return type
pd.DataFrame

`property interferometers`

`property pesummary_prior`

The prior in a form suitable for PESummary.

By convention, Dingo stores all times *relative* to a reference time, typically the trigger time for an event. The prior returned here corrects for that offset to be consistent with other codes.

`property pesummary_samples`

Samples in a form suitable for PESummary.

These samples are adjusted to undo certain conventions used internally by Dingo:

- Times are corrected by the reference time t_ref.
- Samples are unweighted, using a fixed random seed for sampling importance

resampling. * The spin angles phi_jl and theta_jn are transformed to account for a difference in phase definition. * Some columns are dropped: delta_log_prob_target, log_prob

`property phase_marginalization_kwargs`

`sample_synthetic_phase`(*synthetic_phase_kwargs*, *inverse: bool = False*)

Sample a synthetic phase for the waveform. This is a post-processing step applicable to samples theta in the full parameter space, except for the phase parameter (i.e., 14D samples). This step adds a phase parameter to the samples based on likelihood evaluations.

A synthetic phase is sampled as follows.

- Compute and cache the modes for the waveform mu(theta, phase=0) for phase 0, organize them such that each contribution m transforms as exp(-i * m * phase).
- Compute the likelihood on a phase grid, by computing mu(theta, phase) from the cached modes. In principle this likelihood is exact, however, it can deviate slightly from the likelihood computed without cached modes for various technical reasons (e.g., slightly different windowing of cached modes compared to full waveform when transforming TD waveform to FD). These small deviations can be fully accounted for by importance sampling. *Note*: when approximation_22_mode=True, the entire waveform is assumed to transform as exp(2i*phase), in which case the likelihood is only exact if the waveform is fully dominated by the (2, 2) mode.

- Build a synthetic conditional phase distribution based on this grid. We use an interpolated prior distribution bilby.core.prior.Interped, such that we can sample and also evaluate the log_prob. We add a constant background with weight self.synthetic_phase_kwargs to the kde to make sure that we keep a mass-covering property. With this, the importance sampling will yield exact results even when the synthetic phase conditional is just an approximation.

Besides adding phase samples to self.samples['phase'], this method also modifies self.samples['log_prob'] by adding the log_prob of the synthetic phase conditional.

This method modifies self.samples in place.

Parameters

- `synthetic_phase_kwargs` (*dict*) –

  This should consist of the kwargs
  approximation_22_mode (optional) num_processes (optional) n_grid uniform_weight (optional)

- `inverse` (*bool, default False*) – Whether to apply instead the inverse transformation. This is used prior to calculating the log_prob. In inverse mode, the posterior probability over phase is calculated for given samples. It is stored in self.samples['log_prob'].

property synthetic_phase_kwargs

property t_ref

property time_marginalization_kwargs

update_prior(*prior_update*)

Update the prior based on a new dict of priors. Use the existing prior for parameters not included in the new dict.

If class samples have not been importance sampled, then save new sample weights based on the new prior. If class samples have been importance sampled, then update the weights.

Parameters
`prior_update` (*dict*) – Priors to update. This should be of the form {key : prior_str}, where str is a string that can instantiate a prior via PriorDict(prior_update). The prior_update is provided in this form so that it can be properly saved with the Result and later instantiated.

dingo.gw.temporary_debug_utils module

dingo.gw.temporary_debug_utils.**save_training_injection**(*outname*, *pm*, *data*, *idx=0*)

For debugging: extract a training injection. To be used inside train or test loop.

Module contents

dingo.pipe package

Subpackages

dingo.pipe.nodes package

Submodules

dingo.pipe.nodes.generation_node module

class dingo.pipe.nodes.generation_node.**GenerationNode**(*inputs*, *importance_sampling=False*, ***kwargs*)

Bases: GenerationNode

Node for data generation jobs

Parameters:

inputs: bilby_pipe.main.MainInput
The user-defined inputs

trigger_time: float
The trigger time to use in generating analysis data

idx: int
The index of the data-generation job, used to label data products

dag: bilby_pipe.dag.Dag
The dag structure

parent: bilby_pipe.job_creation.node.Node (optional)
Any job to set as the parent to this job - used to enforce dependencies

property event_data_file

property executable

property job_name

setup_arguments(***kwargs*)

dingo.pipe.nodes.importance_sampling_node module

class dingo.pipe.nodes.importance_sampling_node.**ImportanceSamplingNode**(*inputs*, *sampling_node*, *generation_node*, *parallel_idx*, *dag*)

Bases: AnalysisNode

property executable

property result_file

dingo.pipe.nodes.merge_node module

class dingo.pipe.nodes.merge_node.**MergeNode**(***kwargs*)

Bases: MergeNode

property executable

property result_file

dingo.pipe.nodes.pe_summary_node module

class `dingo.pipe.nodes.pe_summary_node.`**`PESummaryNode`**(*inputs*, *merged_node_list*, *generation_node_list*, *dag*)

Bases: `PESummaryNode`

dingo.pipe.nodes.plot_node module

class `dingo.pipe.nodes.plot_node.`**`PlotNode`**(*inputs*, *merged_node*, *dag*)

Bases: `PlotNode`

property executable

dingo.pipe.nodes.sampling_node module

class `dingo.pipe.nodes.sampling_node.`**`SamplingNode`**(*inputs*, *generation_node*, *dag*)

Bases: `AnalysisNode`

property executable

property result_file

property samples_file

Module contents

Submodules

dingo.pipe.dag_creator module

`dingo.pipe.dag_creator.`**`generate_dag`**(*inputs*, *model_args*)

`dingo.pipe.dag_creator.`**`get_parallel_list`**(*inputs*)

`dingo.pipe.dag_creator.`**`get_trigger_time_list`**(*inputs*)

Returns a list of GPS trigger times for each data segment

dingo.pipe.data_generation module

class `dingo.pipe.data_generation.`**`DataGenerationInput`**(*args*, *unknown_args*, *create_data=True*)

Bases: `DataGenerationInput`

property event_data_file

property importance_sampling_updates

`save_hdf5`()

Save frequency-domain strain and ASDs as DingoDataset HDF5 format.

dingo.pipe.data_generation.**create_generation_parser**()

Data generation parser creation

dingo.pipe.data_generation.**main**()

Data generation main logic

dingo.pipe.default_settings module

dingo.pipe.dingo_result module

dingo.pipe.dingo_result.**main**()

dingo.pipe.importance_sampling module

Script to importance sample based on Dingo samples. Based on bilby_pipe data analysis script.

class dingo.pipe.importance_sampling.**ImportanceSamplingInput**(*args*, *unknown_args*)

Bases: `Input`

property calibration_marginalization_kwargs

property importance_sampling_settings

property priors

Read in and compose the prior at run-time

run_sampler()

dingo.pipe.importance_sampling.**create_sampling_parser**()

Data analysis parser creation

dingo.pipe.importance_sampling.**main**()

Data analysis main logic

dingo.pipe.main module

class dingo.pipe.main.**MainInput**(*args*, *unknown_args*, *importance_sampling_updates*)

Bases: `MainInput`

property priors

Read in and compose the prior at run-time

property request_cpus_importance_sampling

dingo.pipe.main.**fill_in_arguments_from_model**(*args*)

dingo.pipe.main.**main**()

dingo.pipe.main.**write_complete_config_file**(*parser*, *args*, *inputs*, *input_cls=<class 'dingo.pipe.main.MainInput'>*)

dingo.pipe.parser module

class dingo.pipe.parser.**StoreBoolean**(*option_strings*, *dest*, *nargs=None*, *const=None*, *default=None*, *type=None*, *choices=None*, *required=False*, *help=None*, *metavar=None*)

Bases: `Action`

argparse class for robust handling of booleans with configargparse

When using configargparse, if the argument is setup with action="store_true", but the default is set to True, then there is no way, in the config file to switch the parameter off. To resolve this, this class handles the boolean properly.

dingo.pipe.parser.**create_parser**(*top_level=True*)

Creates the BilbyArgParser for dingo_pipe

Parameters
top_level – If true, parser is to be used at the top-level with requirement checking etc., else it is an internal call and will be ignored.

Returns
parser – Argument parser

Return type
BilbyArgParser instance

dingo.pipe.plot module

dingo.pipe.plot.**create_parser**()

Generate a parser for the plot script

Returns
parser – A parser with all the default options already added

Return type
BilbyArgParser

dingo.pipe.plot.**main**()

dingo.pipe.sampling module

Script to sample from a Dingo model. Based on bilby_pipe data analysis script.

class dingo.pipe.sampling.**SamplingInput**(*args*, *unknown_args*)

Bases: `Input`

property density_recovery_settings

run_sampler()

dingo.pipe.sampling.**create_sampling_parser**()

Data analysis parser creation

dingo.pipe.sampling.**main**()

Data analysis main logic

dingo.pipe.utils module

Module contents

20.1.2 Module contents

CHAPTER

TWENTYONE

REFERENCES

Dingo is based on a series of papers developing neural posterior estimation for gravitational waves, starting from proof of concept [1], to inclusion of all 15 parameters and analysis of real data [2], noise conditioning and full amortization [3], and group-equivariant NPE [4]. Dingo results are augmented with importance sampling in [5]. Finally, training with forecasted noise (needed for training *prior* to an observing run) is described in [6].

If you use Dingo in your work, we ask that you please cite at least [3].

Contributors to the code are listed in AUTHORS.md. We thank Vivien Raymond and Rory Smith for acting as LIGO-Virgo-KAGRA (LVK) code reviewers. Dingo makes use of many LVK software tools, including Bilby, bilby_pipe, and LALSimulation, as well as third party tools such as PyTorch and nflows.

CHAPTER TWENTYTWO

CONTACT

For questions or comments please contact Maximilian Dax or Stephen Green.

CHAPTER

TWENTYTHREE

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[1] Stephen R. Green, Christine Simpson, and Jonathan Gair. Gravitational-wave parameter estimation with autoregressive neural network flows. *Phys. Rev. D*, 102:104057, 2020. arXiv:2002.07656, doi:10.1103/PhysRevD.102.104057.

[2] Stephen R. Green and Jonathan Gair. Complete parameter inference for GW150914 using deep learning. *Mach. Learn. Sci. Tech.*, 2(3):03LT01, 2021. arXiv:2008.03312, doi:10.1088/2632-2153/abfaed.

[3] Maximilian Dax, Stephen R. Green, Jonathan Gair, Jakob H. Macke, Alessandra Buonanno, and Bernhard Schölkopf. Real-Time Gravitational Wave Science with Neural Posterior Estimation. *Phys. Rev. Lett.*, 127(24):241103, 2021. arXiv:2106.12594, doi:10.1103/PhysRevLett.127.241103.

[4] Maximilian Dax, Stephen R. Green, Jonathan Gair, Michael Deistler, Bernhard Schölkopf, and Jakob H. Macke. Group equivariant neural posterior estimation. *International Conference on Learning Representations*, 2022. arXiv:2111.13139.

[5] Maximilian Dax, Stephen R. Green, Jonathan Gair, Michael Pürrer, Jonas Wildberger, Jakob H. Macke, Alessandra Buonanno, and Bernhard Schölkopf. Neural Importance Sampling for Rapid and Reliable Gravitational-Wave Inference. 10 2022. arXiv:2210.05686.

[6] Jonas Wildberger, Maximilian Dax, Stephen R. Green, Jonathan Gair, Michael Pürrer, Jakob H. Macke, Alessandra Buonanno, and Bernhard Schölkopf. Adapting to noise distribution shifts in flow-based gravitational-wave inference. 11 2022. arXiv:2211.08801.

PYTHON MODULE INDEX

d

INDEX

A

B

C

D

E

F

G

M

N

O

P

T