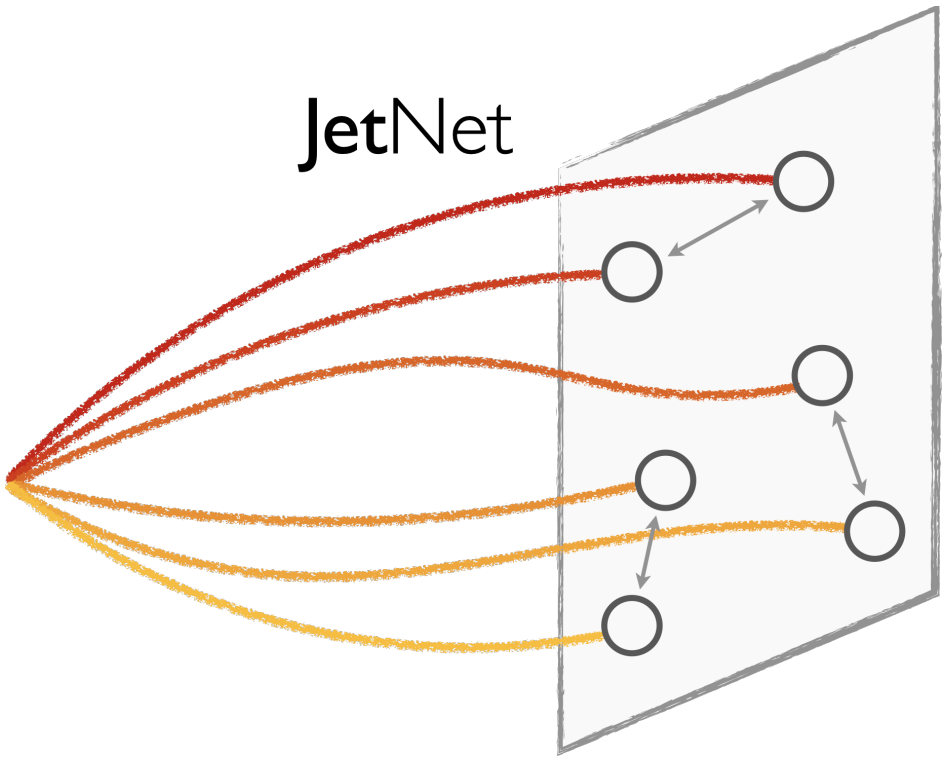
JetNet

Release 0.2.0a

Raghav Kansal

Feb 09, 2024

CONTENTS


JetNet

CHAPTER

ONE

WELCOME TO JETNET!

1.1 JetNet

JetNet is an effort to increase accessibility and reproducibility in jet-based machine learning.

Currently we provide:

- Easy-to-access and standardised interfaces for the following datasets:
  - JetNet
  - TopTagging
  - QuarkGluon
- Standard implementations of generative evaluation metrics (Ref. [1, 2]), including:
  - Fréchet physics distance (FPD)
  - Kernel physics distance (KPD)
  - Wasserstein-1 (W1)
  - Fréchet ParticleNet Distance (FPND)
  - coverage and minimum matching distance (MMD)
- Loss functions:
  - Differentiable implementation of the energy mover's distance [3]
- And more general jet utilities.

Additional functionality is under development, and please reach out if you're interested in contributing!

1.2 Installation

JetNet can be installed with pip:

```
pip install jetnet
```

To use the differentiable EMD loss `jetnet.losses.EMDLoss`, additional libraries must be installed via

```
pip install "jetnet[emdloss]"
```

Finally, PyTorch Geometric must be installed independently for the Fréchet ParticleNet Distance metric `jetnet.evaluation.fpnd` (Installation instructions).

1.3 Quickstart

Datasets can be downloaded and accessed quickly, for example:

```
from jetnet.datasets import JetNet, TopTagging

# as numpy arrays:
particle_data, jet_data = JetNet.getData(
    jet_type=["g", "q"], data_dir="./datasets/jetnet/", download=True
)
# or as a PyTorch dataset:
dataset = TopTagging(
    jet_type="all", data_dir="./datasets/toptagging/", split="train", download=True
)
```

Evaluation metrics can be used as such:

```
generated_jets = np.random.rand(50000, 30, 3)
fpnd_score = jetnet.evaluation.fpnd(generated_jets, jet_type="g")
```

Loss functions can be initialized and used similarly to standard PyTorch in-built losses such as MSE:

```
emd_loss = jetnet.losses.EMDLoss(num_particles=30)
loss = emd_loss(real_jets, generated_jets)
loss.backward()
```

1.4 Documentation

The full API reference and tutorials are available at jetnet.readthedocs.io. Tutorial notebooks are in the tutorials folder, with more to come.

1.5 Contributing

We welcome feedback and contributions! Please feel free to create an issue for bugs or functionality requests, or open pull requests from your forked repo to solve them.

1.5.1 Building and testing locally

Perform an editable installation of the package from inside your forked repo and install the `pytest` package for unit testing:

```
pip install -e .
pip install pytest
```

Run the test suite to ensure everything is working as expected:

```
pytest tests                    # tests all datasets
pytest tests -m "not slow"      # tests only on the JetNet dataset for convenience
```

1.6 Citation

If you use this library for your research, please cite our article in the Journal of Open Source Software:

```
@article{Kansal_JetNet_2023,
  author = {Kansal, Raghav and Pareja, Carlos and Hao, Zichun and Duarte, Javier},
  doi = {10.21105/joss.05789},
  journal = {Journal of Open Source Software},
  number = {90},
  pages = {5789},
  title = {{JetNet: A Python package for accessing open datasets and benchmarking␣
↪machine learning methods in high energy physics}},
  url = {https://joss.theoj.org/papers/10.21105/joss.05789},
  volume = {8},
  year = {2023}
}
```

Please further cite the following if you use these components of the library.

1.6.1 JetNet dataset or FPND

```
@inproceedings{Kansal_MPGAN_2021,
  author = {Kansal, Raghav and Duarte, Javier and Su, Hao and Orzari, Breno and Tomei,␣
↪Thiago and Pierini, Maurizio and Touranakou, Mary and Vlimant, Jean-Roch and Gunopulos,
↪ Dimitrios},
  booktitle = "{Advances in Neural Information Processing Systems}",
  editor = {M. Ranzato and A. Beygelzimer and Y. Dauphin and P.S. Liang and J. Wortman␣
↪Vaughan},
  pages = {23858--23871},
  publisher = {Curran Associates, Inc.},
  title = {Particle Cloud Generation with Message Passing Generative Adversarial␣
```

(continues on next page)

(continued from previous page)

```
↪Networks},
  url = {https://proceedings.neurips.cc/paper_files/paper/2021/file/
↪c8512d142a2d849725f31a9a7a361ab9-Paper.pdf},
  volume = {34},
  year = {2021},
  eprint = {2106.11535},
  archivePrefix = {arXiv},
}
```

1.6.2 FPD or KPD

```
@article{Kansal_Evaluating_2023,
  author = {Kansal, Raghav and Li, Anni and Duarte, Javier and Chernyavskaya, Nadezda␣
↪and Pierini, Maurizio and Orzari, Breno and Tomei, Thiago},
  title = {Evaluating generative models in high energy physics},
  reportNumber = "FERMILAB-PUB-22-872-CMS-PPD",
  doi = "10.1103/PhysRevD.107.076017",
  journal = "{Phys. Rev. D}",
  volume = "107",
  number = "7",
  pages = "076017",
  year = "2023",
  eprint = "2211.10295",
  archivePrefix = "arXiv",
}
```

1.6.3 EMD Loss

Please cite the respective qpth or cvxpy libraries, depending on the method used (`qpth` by default), as well as the original EMD paper *[3]*.

1.7 References

[1] R. Kansal et al., *Particle Cloud Generation with Message Passing Generative Adversarial Networks*, NeurIPS 2021 [2106.11535].

[2] R. Kansal et al., *Evaluating Generative Models in High Energy Physics*, Phys. Rev. D **107** (2023) 076017 [2211.10295].

[3] P. T. Komiske, E. M. Metodiev, and J. Thaler, *The Metric Space of Collider Events*, Phys. Rev. Lett. **123** (2019) 041801 [1902.02346].

CHAPTER

TWO

DATASETS

2.1 JetNet

class jetnet.datasets.**JetNet**(**args: Any*, ***kwargs: Any*)

PyTorch `torch.unit.data.Dataset` class for the JetNet dataset.

If hdf5 files are not found in the `data_dir` directory then dataset will be downloaded from Zenodo (https://zenodo.org/record/6975118 or https://zenodo.org/record/6975117).

Parameters

- **jet_type** (*Union[str, Set[str]], optional*) – individual type or set of types out of 'g' (gluon), 'q' (light quarks), 't' (top quarks), 'w' (W bosons), or 'z' (Z bosons). "all" will get all types. Defaults to "all".
- **data_dir** (*str, optional*) – directory in which data is (to be) stored. Defaults to "./".
- **particle_features** (*List[str], optional*) – list of particle features to retrieve. If empty or None, gets no particle features. Defaults to `["etarel", "phirel", "ptrel", "mask"]`.
- **jet_features** (*List[str], optional*) – list of jet features to retrieve. If empty or None, gets no jet features. Defaults to `["type", "pt", "eta", "mass", "num_particles"]`.
- **particle_normalisation** (*NormaliseABC, optional*) – optional normalisation to apply to particle data. Defaults to None.
- **jet_normalisation** (*NormaliseABC, optional*) – optional normalisation to apply to jet data. Defaults to None.
- **particle_transform** (*callable, optional*) – A function/transform that takes in the particle data tensor and transforms it. Defaults to None.
- **jet_transform** (*callable, optional*) – A function/transform that takes in the jet data tensor and transforms it. Defaults to None.
- **num_particles** (*int, optional*) – number of particles to retain per jet, max of 150. Defaults to 30.
- **split** (*str, optional*) – dataset split, out of {"train", "valid", "test", "all"}. Defaults to "train".
- **split_fraction** (*List[float], optional*) – splitting fraction of training, validation, testing data respectively. Defaults to [0.7, 0.15, 0.15].
- **seed** (*int, optional*) – PyTorch manual seed - important to use the same seed for all dataset splittings. Defaults to 42.

- **download** (*bool, optional*) – If True, downloads the dataset from the internet and puts it in the `data_dir` directory. If dataset is already downloaded, it is not downloaded again. Defaults to False.

Methods:

`getData`([jet_type, data_dir, ...])	Downloads, if needed, and loads and returns JetNet data.

classmethod getData(*jet_type: str | set[str] = 'all', data_dir: str = './', particle_features: list[str] | None = 'all', jet_features: list[str] | None = 'all', num_particles: int = 30, split: str = 'all', split_fraction: list[float] | None = None, seed: int = 42, download: bool = False*) → tuple[numpy.ndarray | None, numpy.ndarray | None]

Downloads, if needed, and loads and returns JetNet data.

Parameters

- **jet_type** (*Union[str, Set[str]], optional*) – individual type or set of types out of 'g' (gluon), 't' (top quarks), 'q' (light quarks), 'w' (W bosons), or 'z' (Z bosons). "all" will get all types. Defaults to "all".
- **data_dir** (*str, optional*) – directory in which data is (to be) stored. Defaults to "./".
- **particle_features** (*List[str], optional*) – list of particle features to retrieve. If empty or None, gets no particle features. Defaults to `["etarel", "phirel", "ptrel", "mask"]`.
- **jet_features** (*List[str], optional*) – list of jet features to retrieve. If empty or None, gets no jet features. Defaults to `["type", "pt", "eta", "mass", "num_particles"]`.
- **num_particles** (*int, optional*) – number of particles to retain per jet, max of 150. Defaults to 30.
- **split** (*str, optional*) – dataset split, out of {"train", "valid", "test", "all"}. Defaults to "train".
- **split_fraction** (*List[float], optional*) – splitting fraction of training, validation, testing data respectively. Defaults to [0.7, 0.15, 0.15].
- **seed** (*int, optional*) – PyTorch manual seed - important to use the same seed for all dataset splittings. Defaults to 42.
- **download** (*bool, optional*) – If True, downloads the dataset from the internet and puts it in the `data_dir` directory. If dataset is already downloaded, it is not downloaded again. Defaults to False.

Returns

particle data, jet data

Return type

tuple[np.ndarray | None, np.ndarray | None]

2.2 TopTagging

class jetnet.datasets.**TopTagging**(**args: Any*, ***kwargs: Any*)

PyTorch `torch.unit.data.Dataset` class for the Top Quark Tagging Reference dataset.

If hdf5 files are not found in the `data_dir` directory then dataset will be downloaded from Zenodo (https://zenodo.org/record/2603256).

Parameters

- **jet_type** (*Union[str, Set[str]], optional*) – individual type or set of types out of 'qcd' and 'top'. Defaults to "all".
- **data_dir** (*str, optional*) – directory in which data is (to be) stored. Defaults to "./".
- **particle_features** (*List[str], optional*) – list of particle features to retrieve. If empty or None, gets no particle features. Defaults to `["E", "px", "py", "pz"]`.
- **jet_features** (*List[str], optional*) – list of jet features to retrieve. If empty or None, gets no jet features. Defaults to `["type", "E", "px", "py", "pz"]`.
- **particle_normalisation** (*NormaliseABC, optional*) – optional normalisation to apply to particle data. Defaults to None.
- **jet_normalisation** (*NormaliseABC, optional*) – optional normalisation to apply to jet data. Defaults to None.
- **particle_transform** (*callable, optional*) – A function/transform that takes in the particle data tensor and transforms it. Defaults to None.
- **jet_transform** (*callable, optional*) – A function/transform that takes in the jet data tensor and transforms it. Defaults to None.
- **num_particles** (*int, optional*) – number of particles to retain per jet, max of 200. Defaults to 200.
- **split** (*str, optional*) – dataset split, out of {"train", "valid", "test", "all"}. Defaults to "train".
- **download** (*bool, optional*) – If True, downloads the dataset from the internet and puts it in the `data_dir` directory. If dataset is already downloaded, it is not downloaded again. Defaults to False.

Methods:

`getData`([jet_type, data_dir, ...])	Downloads, if needed, and loads and returns Top Quark Tagging data.

classmethod getData(*jet_type: str | set[str] = 'all'*, *data_dir: str = './'*, *particle_features: list[str] | None = 'all'*, *jet_features: list[str] | None = 'all'*, *num_particles: int = 200*, *split: str = 'all'*, *download: bool = False*) → tuple[numpy.ndarray | None, numpy.ndarray | None]

Downloads, if needed, and loads and returns Top Quark Tagging data.

Parameters

- **jet_type** (*Union[str, Set[str]], optional*) – individual type or set of types out of 'qcd' and 'top'. Defaults to "all".
- **data_dir** (*str, optional*) – directory in which data is (to be) stored. Defaults to "./".
- **particle_features** (*List[str], optional*) – list of particle features to retrieve. If empty or None, gets no particle features. Defaults to `["E", "px", "py", "pz"]`.

- **jet_features** (*List[str], optional*) – list of jet features to retrieve. If empty or None, gets no jet features. Defaults to `["type", "E", "px", "py", "pz"]`.
- **num_particles** (*int, optional*) – number of particles to retain per jet, max of 200. Defaults to 200.
- **split** (*str, optional*) – dataset split, out of {"train", "valid", "test", "all"}. Defaults to "all".
- **download** (*bool, optional*) – If True, downloads the dataset from the internet and puts it in the `data_dir` directory. If dataset is already downloaded, it is not downloaded again. Defaults to False.

Returns
particle data, jet data

Return type
(tuple[np.ndarray | None, np.ndarray | None])

2.3 QuarkGluon

class `jetnet.datasets.`**`QuarkGluon`**(**args: Any, **kwargs: Any*)

PyTorch `torch.unit.data.Dataset` class for the Quark Gluon Jets dataset. Either jets with or without bottom and charm quark jets can be selected (`with_bc` flag).

If npz files are not found in the `data_dir` directory then dataset will be automatically downloaded from Zenodo (https://zenodo.org/record/3164691).

Parameters

- **jet_type** (*Union[str, Set[str]], optional*) – individual type or set of types out of 'g' (gluon) and 'q' (light quarks). Defaults to "all".
- **data_dir** (*str, optional*) – directory in which data is (to be) stored. Defaults to "./".
- **with_bc** (*bool, optional*) – with or without bottom and charm quark jets. Defaults to True.
- **particle_features** (*List[str], optional*) – list of particle features to retrieve. If empty or None, gets no particle features. Defaults to `["pt", "eta", "phi", "pdgid"]`.
- **jet_features** (*List[str], optional*) – list of jet features to retrieve. If empty or None, gets no jet features. Defaults to `["type"]`.
- **particle_normalisation** (*NormaliseABC, optional*) – optional normalisation to apply to particle data. Defaults to None.
- **jet_normalisation** (*NormaliseABC, optional*) – optional normalisation to apply to jet data. Defaults to None.
- **particle_transform** (*callable, optional*) – A function/transform that takes in the particle data tensor and transforms it. Defaults to None.
- **jet_transform** (*callable, optional*) – A function/transform that takes in the jet data tensor and transforms it. Defaults to None.
- **num_particles** (*int, optional*) – number of particles to retain per jet, max of 153. Defaults to 153.
- **split** (*str, optional*) – dataset split, out of {"train", "valid", "test", "all"}. Defaults to "train".

- **split_fraction** (*List[float], optional*) – splitting fraction of training, validation, testing data respectively. Defaults to [0.7, 0.15, 0.15].
- **seed** (*int, optional*) – PyTorch manual seed - important to use the same seed for all dataset splittings. Defaults to 42.
- **file_list** (*List[str], optional*) – list of files to load, if full dataset is not required. Defaults to None (will load all files).
- **download** (*bool, optional*) – If True, downloads the dataset from the internet and puts it in the `data_dir` directory. If dataset is already downloaded, it is not downloaded again. Defaults to False.

Methods:

`getData`([jet_type, data_dir, with_bc, ...])	Downloads, if needed, and loads and returns Quark Gluon data.

classmethod getData(*jet_type: str | set[str] = 'all', data_dir: str = './', with_bc: bool = True, particle_features: list[str] | None = 'all', jet_features: list[str] | None = 'all', num_particles: int = 153, split: str = 'all', split_fraction: list[float] | None = None, seed: int = 42, file_list: list[str] | None = None, download: bool = False*) → tuple[numpy.ndarray | None, numpy.ndarray | None]

Downloads, if needed, and loads and returns Quark Gluon data.

Parameters

- **jet_type** (*Union[str, Set[str]], optional*) – individual type or set of types out of 'g' (gluon) and 'q' (light quarks). Defaults to "all".
- **data_dir** (*str, optional*) – directory in which data is (to be) stored. Defaults to "./".
- **with_bc** (*bool, optional*) – with or without bottom and charm quark jets. Defaults to True.
- **particle_features** (*List[str], optional*) – list of particle features to retrieve. If empty or None, gets no particle features. Defaults to `["pt", "eta", "phi", "pdgid"]`.
- **jet_features** (*List[str], optional*) – list of jet features to retrieve. If empty or None, gets no jet features. Defaults to `["type"]`.
- **num_particles** (*int, optional*) – number of particles to retain per jet, max of 153. Defaults to 153.
- **split** (*str, optional*) – dataset split, out of {"train", "valid", "test", "all"}. Defaults to "train".
- **split_fraction** (*List[float], optional*) – splitting fraction of training, validation, testing data respectively. Defaults to [0.7, 0.15, 0.15].
- **seed** (*int, optional*) – PyTorch manual seed - important to use the same seed for all dataset splittings. Defaults to 42.
- **file_list** (*List[str], optional*) – list of files to load, if full dataset is not required. Defaults to None (will load all files).
- **download** (*bool, optional*) – If True, downloads the dataset from the internet and puts it in the `data_dir` directory. If dataset is already downloaded, it is not downloaded again. Defaults to False.

Returns
particle data, jet data

Return type
tuple[np.ndarray | None, np.ndarray | None]

2.4 Normalisations

Suite of common ways to normalise data.

Classes:

`FeaturewiseLinear`([feature_shifts, ...])	Shifts features by `feature_shifts` then multiplies by `feature_scales`.
`FeaturewiseLinearBounded`([feature_norms, ...])	Normalizes dataset features by scaling each to an (absolute) max of `feature_norms` and shifting by `feature_shifts`.
`NormaliseABC`()	ABC for generalised normalisation class.

class `jetnet.datasets.normalisations.`**`FeaturewiseLinear`**(*feature_shifts: float | list[float] = 0.0, feature_scales: float | list[float] = 1.0, normalise_features: list[bool] | None = None, normal: bool = False*)

Shifts features by `feature_shifts` then multiplies by `feature_scales`.

If using the `normal` option, `feature_shifts` and `feature_scales` can be derived from the dataset (by calling `derive_dataset_features`) to normalise the data to have 0 mean and unit standard deviation per feature.

Parameters

- **`feature_shifts`** (*`Union[float, List[float]], optional`*) – value to shift features by. Can either be a single float for all features, or a list of length `num_features`. Defaults to 0.0.
- **`feature_scales`** (*`Union[float, List[float]], optional`*) – after shifting, value to multiply features by. Can either be a single float for all features, or a list of length `num_features`. Defaults to 1.0.
- **`normalise_features`** (*`Optional[List[bool]], optional`*) – if only some features need to be normalised, can input here a list of booleans of length `num_features` with `True` meaning normalise and `False` meaning to ignore. Defaults to None i.e. normalise all.
- **`normal`** (*`bool, optional`*) – derive `feature_shifts` and `feature_scales` to have 0 mean and unit standard deviation per feature after normalisation (`derive_dataset_features` method must be called before normalising).

Methods:

`derive_dataset_features`(x)	If using the `normal` option, this will derive the means and standard deviations per feature, and save and return them.
`features_need_deriving`()	Checks if any dataset values or features need to be derived

derive_dataset_features(*x: ArrayLike*) → tuple[numpy.ndarray, numpy.ndarray] | None

If using the `normal` option, this will derive the means and standard deviations per feature, and save and return them. If not, will do nothing.

Parameters
x (*ArrayLike*) – dataset of shape [..., `num_features`].

Returns

if `normal` option, means and stds of each
feature.

Return type
(Optional[Tuple[np.ndarray, np.ndarray]])

features_need_deriving() → bool

Checks if any dataset values or features need to be derived

class jetnet.datasets.normalisations.**FeaturewiseLinearBounded**(*feature_norms: float | list[float] = 1.0, feature_shifts: float | list[float] = 0.0, feature_maxes: list[float] | None = None, normalise_features: list[bool] | None = None*)

Normalizes dataset features by scaling each to an (absolute) max of `feature_norms` and shifting by `feature_shifts`.

If the value in the list for a feature is None, it won't be scaled or shifted.

Parameters

- **feature_norms** (*Union[float, List[float]], optional*) – max value to scale each feature to. Can either be a single float for all features, or a list of length `num_features`. Defaults to 1.0.
- **feature_shifts** (*Union[float, List[float]], optional*) – after scaling, value to shift feature by. Can either be a single float for all features, or a list of length `num_features`. Defaults to 0.0.
- **feature_maxes** (*List[float], optional*) – max pre-scaling absolute value of each feature, used for scaling to the norm and inverting.
- **normalise_features** (*Optional[List[bool]], optional*) – if only some features need to be normalised, can input here a list of booleans of length `num_features` with `True` meaning normalise and `False` meaning to ignore. Defaults to None i.e. normalise all.

Methods:

derive_dataset_features(x)	Derives, saves, and returns absolute feature maxes of dataset `x`.
features_need_deriving()	Checks if any dataset values or features need to be derived

derive_dataset_features(*x: ArrayLike*) → ndarray

Derives, saves, and returns absolute feature maxes of dataset `x`.

Parameters
x (*ArrayLike*) – dataset of shape [..., `num_features`].

Returns
feature maxes

Return type
np.ndarray

features_need_deriving() → bool

Checks if any dataset values or features need to be derived

class jetnet.datasets.normalisations.**NormaliseABC**

ABC for generalised normalisation class.

Methods:

derive_dataset_features(x)	Derive features from dataset needed for normalisation if needed
features_need_deriving()	Checks if any dataset values or features need to be derived

derive_dataset_features(*x: ArrayLike*)

Derive features from dataset needed for normalisation if needed

features_need_deriving() → bool

Checks if any dataset values or features need to be derived

2.5 Utility Functions

Utility methods for datasets.

Functions:

checkConvertElements(elem, valid_types[, ntype])	Checks if elem(s) are valid and if needed converts into a list
checkDownloadZenodoDataset(data_dir, ...)	Checks if dataset exists and md5 hash matches; if not and download = True, downloads it from Zenodo, and returns the file path.
checkListNotEmpty(*inputs)	Checks that list inputs are not None or empty
checkStrToList(*inputs[, to_set])	Converts str inputs to a list or set
download_progress_bar(file_url, file_dest)	Download while outputting a progress bar.
firstNotNoneElement(*inputs)	Returns the first element out of all inputs which isn't None
getOrderedFeatures(data, features, ...)	Returns data with features in the order specified by `features`.
getSplitting(length, split, splits, ...)	Returns starting and ending index for splitting a dataset of length `length` according to the input `split` out of the total possible `splits` and a given `split_fraction`.

jetnet.datasets.utils.**checkConvertElements**(*elem: str | list[str], valid_types: list[str], ntype: str = 'element'*)

Checks if elem(s) are valid and if needed converts into a list

jetnet.datasets.utils.**checkDownloadZenodoDataset**(*data_dir: str, dataset_name: str, record_id: int, key: str, download: bool*) → str

Checks if dataset exists and md5 hash matches; if not and download = True, downloads it from Zenodo, and returns the file path. or if not and download = False, raises an error.

`jetnet.datasets.utils.checkListNotEmpty`(**inputs: list[list]*) → list[bool]

Checks that list inputs are not None or empty

`jetnet.datasets.utils.checkStrToList`(**inputs: list[str | list[str] | set[str]], to_set: bool = False*) → list[list[str]] | list[set[str]] | list

Converts str inputs to a list or set

`jetnet.datasets.utils.download_progress_bar`(*file_url: str, file_dest: str*)

Download while outputting a progress bar. Modified from https://sumit-ghosh.com/articles/python-download-progress-bar/

Parameters

- **file_url** (*str*) – url to download from
- **file_dest** (*str*) – path at which to save downloaded file

`jetnet.datasets.utils.firstNotNoneElement`(**inputs: list[Any]*) → Any

Returns the first element out of all inputs which isn't None

`jetnet.datasets.utils.getOrderedFeatures`(*data: ArrayLike, features: list[str], features_order: list[str]*) → ndarray

Returns data with features in the order specified by `features`.

Parameters

- **data** (*ArrayLike*) – input data
- **features** (*List[str]*) – desired features in order
- **features_order** (*List[str]*) – name and ordering of features in input data

Returns
data with features in specified order

Return type
(np.ndarray)

`jetnet.datasets.utils.getSplitting`(*length: int, split: str, splits: list[str], split_fraction: list[float]*) → tuple[int, int]

Returns starting and ending index for splitting a dataset of length `length` according to the input `split` out of the total possible `splits` and a given `split_fraction`.

"all" is considered a special keyword to mean the entire dataset - it cannot be used to define a normal splitting, and if it is a possible splitting it must be the last entry in `splits`.

e.g. for `length = 100`, `split = "valid"`, `splits = ["train", "valid", "test"]`, `split_fraction = [0.7, 0.15, 0.15]`

This will return `(70, 85)`.

CHAPTER

THREE

METRICS

Functions:

cov_mmd(real_jets, gen_jets[, ...])	Calculate coverage and MMD between real and generated jets, using the Energy Mover's Distance as the distance metric.
fpd(real_features, gen_features[, ...])	Calculates the value and error of the Fréchet physics distance (FPD) between a set of real and generated features, as defined in https://arxiv.org/abs/2211.10295.
fpnd(jets, jet_type[, dataset_name, device, ...])	Calculates the Frechet ParticleNet Distance, as defined in https://arxiv.org/abs/2106.11535, for input `jets` of type `jet_type`.
get_fpd_kpd_jet_features(jets[, efp_jobs])	Get recommended jet features (36 EFPs) for the FPD and KPD metrics from an input sample of jets.
kpd(real_features, gen_features[, ...])	Calculates the median and error of the kernel physics distance (KPD) between a set of real and generated features, as defined in https://arxiv.org/abs/2211.10295.
w1efp(jets1, jets2[, use_particle_masses, ...])	Get 1-Wasserstein distances between Energy Flow Polynomials (Komiske et al. 2017 https://arxiv.org/abs/1712.07124) of `jets1` and `jets2`.
w1m(jets1, jets2[, num_eval_samples, ...])	Get 1-Wasserstein distance between masses of `jets1` and `jets2`.
w1p(jets1, jets2[, mask1, mask2, ...])	Get 1-Wasserstein distances between particle features of `jets1` and `jets2`.

`jetnet.evaluation.`**`cov_mmd`**(*real_jets: Tensor | np.ndarray*, *gen_jets: Tensor | np.ndarray*, *num_eval_samples: int = 100*, *num_batches: int = 10*, *use_tqdm: bool = True*) → tuple[float, float]

Calculate coverage and MMD between real and generated jets, using the Energy Mover's Distance as the distance metric.

Parameters

- **real_jets** (*Tensor | np.ndarray*) – Tensor or array of jets, of shape `[num_jets, num_particles, num_features]` with features in order `[eta, phi, pt]`
- **gen_jets** (*Tensor | np.ndarray*) – tensor or array of generated jets, same format as real_jets.
- **num_eval_samples** (*int*) – number of jets out of the real and gen jets each between which to evaluate COV and MMD. Defaults to 100.
- **num_batches** (*int*) – number of different batches to calculate COV and MMD and average over. Defaults to 100.

- **use_tqdm** (*bool*) – use tqdm bar while calculating over `num_batches` batches. Defaults to True.

Returns

- **float**: coverage, averaged over `num_batches`.
- **float**: MMD, averaged over `num_batches`.

Return type
Tuple[float, float]

`jetnet.evaluation.fpd`(*real_features: Tensor | np.ndarray*, *gen_features: Tensor | np.ndarray*, *min_samples: int = 20000*, *max_samples: int = 50000*, *num_batches: int = 20*, *num_points: int = 10*, *normalise: bool = True*, *seed: int = 42*) → tuple[float, float]

Calculates the value and error of the Fréchet physics distance (FPD) between a set of real and generated features, as defined in https://arxiv.org/abs/2211.10295.

It is recommended to use input sample sizes of at least 50,000, and the default values for other input parameters for consistency with other measurements.

Similarly, for jets, it is recommended to use the set of EFPs as provided by the `get_fpd_kpd_jet_features` method.

Parameters

- **real_features** (*Tensor | np.ndarray*) – set of real features of shape `[num_samples, num_features]`.
- **gen_features** (*Tensor | np.ndarray*) – set of generated features of shape `[num_samples, num_features]`.
- **min_samples** (*int, optional*) – min batch size to measure FPD for. Defaults to 20,000.
- **max_samples** (*int, optional*) – max batch size to measure FPD for. Defaults to 50,000.
- **num_batches** (*int, optional*) – # of batches to average over for each batch size. Defaults to 20.
- **num_points** (*int, optional*) – # of points to sample between the min and max samples. Defaults to 10.
- **normalise** (*bool, optional*) – normalise the individual features over the full sample to have the same scaling. Defaults to True.
- **seed** (*int, optional*) – random seed. Defaults to 42.

Returns
value and error of FPD.

Return type
Tuple[float, float]

`jetnet.evaluation.fpnd`(*jets: Tensor | np.ndarray*, *jet_type: str*, *dataset_name: str = 'jetnet'*, *device: str | None = None*, *batch_size: int = 16*, *use_tqdm: bool = True*) → float

Calculates the Frechet ParticleNet Distance, as defined in https://arxiv.org/abs/2106.11535, for input `jets` of type `jet_type`.

`jets` are passed through our pretrained ParticleNet module and activations are compared with the cached activations from real jets. The recommended and max number of jets is 50,000.

torch_geometric must be installed separately for running inference with ParticleNet.

Currently FPND only supported for the JetNet dataset with 30 particles, but functionality for other datasets + ability for users to use their own version is in development.

Parameters

- **jets** (*Tensor | np.ndarray*) – Tensor or array of jets, of shape `[num_jets, num_particles, num_features]` with features in order `[eta, phi, pt, (optional) mask]`
- **jet_type** (*str*) – jet type, out of `['g', 't', 'q']`.
- **dataset_name** (*str*) – Dataset to use. Currently only JetNet is supported. Defaults to "jetnet".
- **device** (*str*) – 'cpu' or 'cuda'. If not specified, defaults to cuda if available else cpu.
- **batch_size** (*int*) – Batch size for ParticleNet inference. Defaults to 16.
- **use_tqdm** (*bool*) – use tqdm bar while getting ParticleNet activations. Defaults to True.

Returns
the measured FPND.

Return type
float

`jetnet.evaluation.`**`get_fpd_kpd_jet_features`**(*jets: Tensor | np.ndarray, efp_jobs: int | None = None*) → np.ndarray

Get recommended jet features (36 EFPs) for the FPD and KPD metrics from an input sample of jets.

Parameters

- **jets** (*Tensor | np.ndarray*) – Tensor or array of jets, of shape `[num_jets, num_particles, num_features]` with features in order `[eta, phi, pt]`.
- **efp_jobs** (*int, optional*) – number of jobs to use for energyflow's EFP batch computation. None means as many processes as there are CPUs.

Returns
array of EFPs of shape `[num_jets, 36]`.

Return type
np.ndarray

`jetnet.evaluation.`**`kpd`**(*real_features: Tensor | np.ndarray, gen_features: Tensor | np.ndarray, num_batches: int = 10, batch_size: int = 5000, normalise: bool = True, seed: int = 42, num_threads: int | None = None*) → tuple[float, float]

Calculates the median and error of the kernel physics distance (KPD) between a set of real and generated features, as defined in https://arxiv.org/abs/2211.10295.

It is recommended to use input sample sizes of at least 50,000, and the default values for other input parameters for consistency with other measurements.

Similarly, for jets, it is recommended to use the set of EFPs as provided by the `get_fpd_kpd_jet_features` method.

Parameters

- **real_features** (*Tensor | np.ndarray*) – set of real features of shape `[num_samples, num_features]`.
- **gen_features** (*Tensor | np.ndarray*) – set of generated features of shape `[num_samples, num_features]`.

- **num_batches** (*int, optional*) – number of batches to average over. Defaults to 10.
- **batch_size** (*int, optional*) – size of each batch for which MMD is measured. Defaults to 5,000.
- **normalise** (*bool, optional*) – normalise the individual features over the full sample to have the same scaling. Defaults to True.
- **seed** (*int, optional*) – random seed. Defaults to 42.
- **num_threads** (*int, optional*) – parallelize KPD through numba using this many threads. 0 means numba's default number of threads, based on # of cores available. Defaults to None, i.e. no parallelization.

Returns
median and error of KPD.

Return type
Tuple[float, float]

`jetnet.evaluation.w1efp`(*jets1: Tensor | np.ndarray, jets2: Tensor | np.ndarray, use_particle_masses: bool = False, efpset_args: list | None = None, num_eval_samples: int = 50000, num_batches: int = 5, return_std: bool = True, efp_jobs: int | None = None*)

Get 1-Wasserstein distances between Energy Flow Polynomials (Komiske et al. 2017 https://arxiv.org/abs/1712.07124) of `jets1` and `jets2`.

Parameters

- **jets1** (*Tensor | np.ndarray*) – Tensor or array of jets of shape `[num_jets, num_particles, num_features]`, with features in order `[eta, phi, pt, (optional) mass]`. If no particle masses given (`particle_masses` should be False), they are assumed to be 0.
- **jets2** (*Tensor | np.ndarray*) – Tensor or array of jets, of same format as `jets1`.
- **use_particle_masses** (*bool*) – Whether `jets1` and `jets2` have particle masses as their 4th particle features. Defaults to False.
- **efpset_args** (*List*) – Args for the energyflow.efpset function to specify which EFPs to use, as defined here https://energyflow.network/docs/efp/#efpset. Defaults to the n=4, d=5, prime EFPs.
- **num_eval_samples** (*int*) – Number of jets out of the total to use for W1 measurement. Defaults to 50,000.
- **num_batches** (*int*) – Number of different batches to average W1 scores over. Defaults to 5.
- **average_over_efps** (*bool*) – Average over the EFPs to return a single W1-EFP score. Defaults to True.
- **return_std** (*bool*) – Return the standard deviation as well of the W1 scores over the `num_batches` batches. Defaults to True.
- **efp_jobs** (*int*) – number of jobs to use for energyflow's EFP batch computation. None means as many processes as there are CPUs.

Returns

- **np.ndarray**: array of average W1 scores for each EFP.
- **np.ndarray** *(optional, only if return_std is True)*: array of std of W1 scores for each feature.

Return type
Tuple[np.ndarray, np.ndarray]

`jetnet.evaluation.w1m`(*jets1: Tensor | np.ndarray*, *jets2: Tensor | np.ndarray*, *num_eval_samples: int = 50000*, *num_batches: int = 5*, *return_std: bool = True*)

Get 1-Wasserstein distance between masses of `jets1` and `jets2`.

Parameters

- **jets1** (*Tensor | np.ndarray*) – Tensor or array of jets, of shape `[num_jets, num_particles, num_features]` with features in order `[eta, phi, pt, (optional) mass]`
- **jets2** (*Tensor | np.ndarray*) – Tensor or array of jets, of same format as `jets1`.
- **num_eval_samples** (*int*) – Number of jets out of the total to use for W1 measurement. Defaults to 50,000.
- **num_batches** (*int*) – Number of different batches to average W1 scores over. Defaults to 5.
- **return_std** (*bool*) – Return the standard deviation as well of the W1 scores over the `num_batches` batches. Defaults to True.

Returns

- **float**: W1 mass score, averaged over `num_batches`.
- **float** *(optional, only if ``return_std`` is True)`*: standard deviation of W1 mass scores over `num_batches`.

Return type
Tuple[float, float]

`jetnet.evaluation.w1p`(*jets1: Tensor | np.ndarray*, *jets2: Tensor | np.ndarray*, *mask1: Tensor | np.ndarray = None*, *mask2: Tensor | np.ndarray = None*, *exclude_zeros: bool = True*, *num_particle_features: int = 0*, *num_eval_samples: int = 50000*, *num_batches: int = 5*, *return_std: bool = True*)

Get 1-Wasserstein distances between particle features of `jets1` and `jets2`.

Parameters

- **jets1** (*Tensor | np.ndarray*) – Tensor or array of jets, of shape `[num_jets, num_particles_per_jet, num_features_per_particle]`.
- **jets2** (*Tensor | np.ndarray*) – Tensor or array of jets, of same format as `jets1`.
- **mask1** (*Tensor | np.ndarray*) – Optional tensor or array of binary particle masks, of shape `[num_jets, num_particles_per_jet]` or `[num_jets, num_particles_per_jet, 1]`. If given, 0-masked particles will be excluded from w1 calculation.
- **mask2** (*Tensor | np.ndarray*) – Optional tensor or array of same format as `masks2`.
- **exclude_zeros** (*bool*) – Ignore zero-padded particles i.e. those whose whose feature norms are exactly 0. Defaults to True.
- **num_particle_features** (*int*) – Will return W1 scores of the first `num_particle_features` particle features. If 0, will calculate for all.
- **num_eval_samples** (*int*) – Number of jets out of the total to use for W1 measurement. Defaults to 50,000.

- **num_batches** (*int*) – Number of different batches to average W1 scores over. Defaults to 5.
- **return_std** (*bool*) – Return the standard deviation as well of the W1 scores over the num_batches batches. Defaults to True.

Returns

- **Union[float, np.ndarray]**: array of length num_particle_features containing average W1 scores for each feature.
- **Union[float, np.ndarray]** *(optional, only if ``return_std`` is True)`*: array of length num_particle_features containing standard deviation W1 scores for each feature.

Return type

Tuple[Union[float, np.ndarray], Union[float, np.ndarray]]

CHAPTER

FOUR

LOSS FUNCTIONS

Classes:

EMDLoss(*args, **kwargs)	Calculates the energy mover's distance between two batches of jets differentiably as a convex optimization problem either through the linear programming library `cvxpy` or by converting it to a quadratic programming problem and using the `qpth` library.

class `jetnet.losses.`**`EMDLoss`**(**args: Any, **kwargs: Any*)

Calculates the energy mover's distance between two batches of jets differentiably as a convex optimization problem either through the linear programming library `cvxpy` or by converting it to a quadratic programming problem and using the `qpth` library. `cvxpy` is marginally more accurate but `qpth` is significantly faster so defaults to `qpth`.

JetNet must be installed with the extra option `pip install jetnet[emdloss]` to use this.

Note: PyTorch <= 1.9 has a bug which will cause this to fail for >= 32 particles. This PR should fix this from 1.10 onwards https://github.com/pytorch/pytorch/pull/61815.

Parameters

- **method** (*str*) – 'cvxpy' or 'qpth'. Defaults to 'qpth'.
- **num_particles** (*int*) – number of particles per jet - only needs to be specified if method is 'cvxpy'.
- **qpth_form** (*str*) – 'L2' or 'QP'. Defaults to 'L2'.
- **qpth_l2_strength** (*float*) – regularization parameter for 'L2' qp form. Defaults to 0.0001.
- **device** (*str*) – 'cpu' or 'cuda'. Defaults to 'cpu'.

Methods:

forward(jets1, jets2[, return_flows])	Calculate EMD between `jets1` and `jets2`.

forward(*jets1: Tensor, jets2: Tensor, return_flows: bool = False*) → Tensor | tuple[Tensor, Tensor]

Calculate EMD between `jets1` and `jets2`.

Parameters

- **jets1** (*Tensor*) – tensor of shape `[num_jets, num_particles, num_features]`, with features in order `[eta, phi, pt]`.
- **jets2** (*Tensor*) – tensor of same format as `jets1`.

- **return_flows** (*bool*) – return energy flows between particles in each jet. Defaults to False.

Returns

- **Tensor**: EMD scores tensor of shape [num_jets].
- **Tensor** *Optional*, if `return_flows` is True: tensor of flows between particles of shape `[num_jets, num_particles, num_particles]`.

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

CHAPTER

FIVE

UTILITY FUNCTIONS

Functions:

EtaPhiPtE_to_cartesian(p4)	Transform 4-momenta from polar coordinates to Cartesian coordinates for massless particles.
EtaPhiPtE_to_relEtaPhiPt(p4)	Get particle features in relative polar coordinates from 4-momenta in polar coordinates.
YPhiPtE_to_cartesian(p4)	Transform 4-momenta from polar coordinates to Cartesian coordinates.
cartesian_to_EtaPhiPtE(p4)	Transform 4-momenta from Cartesian coordinates to polar coordinates for massless particles.
cartesian_to_YPhiPtE(p4)	Transform 4-momenta from Cartesian coordinates to polar coordinates.
cartesian_to_relEtaPhiPt(p4)	Get particle features in relative polar coordinates from 4-momenta in Cartesian coordinates.
efps(jets[, use_particle_masses, ...])	Utility for calculating EFPs for jets in JetNet format using the energyflow library.
gen_jet_corrections(jets[, ...])	Zero's masked particles and negative pTs.
jet_features(jets)	Calculates jet features by summing over particle Lorentz 4-vectors.
relEtaPhiPt_to_EtaPhiPt(p_polarrel, jet_features)	Get particle features in absolute polar coordinates from relative polar coordinates and jet features.
relEtaPhiPt_to_cartesian(p_polarrel, ...[, ...])	Get particle features in absolute Cartesian coordinates from relative polar coordinates
to_image(jets, im_size[, mask, maxR])	Convert jet(s) into 2D `im_size` x `im_size` or 3D `num_jets` x `im_size` x `im_size` image arrays.

`jetnet.utils.`**`EtaPhiPtE_to_cartesian`**(*p4: np.ndarray | torch.Tensor*) → np.ndarray | torch.Tensor

Transform 4-momenta from polar coordinates to Cartesian coordinates for massless particles.

Parameters

p4 (*np.ndarray or torch.Tensor*) – array of 4-momenta in polar coordinates, of shape `[. .., 4]`. The last axis should be in order $(\eta, \phi, p_\mathrm{T}, E/c)$, where η is the pseudorapidity.

Returns

array of 4-momenta in polar coordinates, arranged in order $(E/c, p_x, p_y, p_z)$.

Return type

np.ndarray or torch.Tensor

`jetnet.utils.`**`EtaPhiPtE_to_relEtaPhiPt`**(*p4: np.ndarray | torch.Tensor*) → np.ndarray | torch.Tensor

Get particle features in relative polar coordinates from 4-momenta in polar coordinates.

Parameters

p4 (*np.ndarray or torch.Tensor*) – array of 4-momenta in polar coordinates, of shape `[... , 4]`. The last axis should be in order $(\eta, \phi, p_{\mathrm{T}}, E/c)$, where η is the pseudorapidity.

Returns

array of features in relative polar coordinates, arranged in order $(\eta^{\mathrm{rel}}, \phi^{\mathrm{rel}}, p_{\mathrm{T}}^{\mathrm{rel}})$.

Return type

np.ndarray or torch.Tensor

`jetnet.utils.YPhiPtE_to_cartesian`(*p4: np.ndarray | torch.Tensor*) → np.ndarray | torch.Tensor

Transform 4-momenta from polar coordinates to Cartesian coordinates.

Parameters

p4 (*np.ndarray or torch.Tensor*) – array of 4-momenta in Cartesian coordinates, of shape `[... , 4]`. The last axis should be in order $(y, \phi, E/c, p_{\mathrm{T}})$, where y is the rapidity.

Returns

array of 4-momenta in polar coordinates, arranged in order $(E/c, p_x, p_y, p_z)$.

Return type

np.ndarray or torch.Tensor

`jetnet.utils.cartesian_to_EtaPhiPtE`(*p4: np.ndarray | torch.Tensor*) → np.ndarray | torch.Tensor

Transform 4-momenta from Cartesian coordinates to polar coordinates for massless particles.

Parameters

p4 (*np.ndarray or torch.Tensor*) – array of 4-momenta in Cartesian coordinates, of shape `[... , 4]`. The last axis should be in order $(E/c, p_x, p_y, p_z)$.

Returns

array of 4-momenta in polar coordinates, arranged in order $(\eta, \phi, p_{\mathrm{T}}, E/c)$, where η is the pseudorapidity.

Return type

np.ndarray or torch.Tensor

`jetnet.utils.cartesian_to_YPhiPtE`(*p4: np.ndarray | torch.Tensor*) → np.ndarray | torch.Tensor

Transform 4-momenta from Cartesian coordinates to polar coordinates.

Parameters

p4 (*np.ndarray or torch.Tensor*) – array of 4-momenta in Cartesian coordinates, of shape `[... , 4]`. The last axis should be in order $(E/c, p_x, p_y, p_z)$.

Returns

array of 4-momenta in polar coordinates, arranged in order $(y, \phi, E/c, p_{\mathrm{T}})$, where y is the rapidity.

Return type

np.ndarray or torch.Tensor

`jetnet.utils.cartesian_to_relEtaPhiPt`(*p4: np.ndarray | torch.Tensor*) → np.ndarray | torch.Tensor

Get particle features in relative polar coordinates from 4-momenta in Cartesian coordinates.

Parameters

p4 (*np.ndarray or torch.Tensor*) – array of 4-momenta in Cartesian coordinates, of shape `[... , 4]`. The last axis should be in order $(E/c, p_x, p_y, p_z)$.

Returns

array of features in relative polar coordinates, arranged in order $(\eta^{\mathrm{rel}}, \phi^{\mathrm{rel}}, p_{\mathrm{T}}^{\mathrm{rel}})$.

Return type
np.ndarray or torch.Tensor

`jetnet.utils.efps`(*jets: ndarray*, *use_particle_masses: bool = False*, *efpset_args: list | None = None*, *efp_jobs: int | None = None*) → ndarray

Utility for calculating EFPs for jets in JetNet format using the energyflow library.

Parameters

- **jets** (*np.ndarray*) – array of either a single or multiple jets, of shape either `[num_particles, num_features]` or `[num_jets, num_particles, num_features]`, with features in order `[eta, phi, pt, (optional) mass]`. If no particle masses given, they are assumed to be 0.
- **efpset_args** (*List*) – Args for the energyflow.efpset function to specify which EFPs to use, as defined here https://energyflow.network/docs/efp/#efpset. Defaults to the n=4, d=5, prime EFPs.
- **efp_jobs** (*int*) – number of jobs to use for energyflow's EFP batch computation. None means as many processes as there are CPUs.

Returns
1D (if inputted single jet) or 2D array of shape `[num_jets, num_efps]` of EFPs per jet

Return type
np.ndarray

`jetnet.utils.gen_jet_corrections`(*jets: ArrayLike*, *ret_mask_separate: bool = True*, *zero_mask_particles: bool = True*, *zero_neg_pt: bool = True*, *pt_index: int = 2*) → ArrayLike | tuple[ArrayLike, ArrayLike]

Zero's masked particles and negative pTs.

Parameters

- **jets** (*ArrayLike*) – jets to recorrect.
- **ret_mask_separate** (*bool, optional*) – return the jet and mask separately. Defaults to True.
- **zero_mask_particles** (*bool, optional*) – set features of zero-masked particles to 0. Defaults to True.
- **zero_neg_pt** (*bool, optional*) – set pT to 0 for particles with negative pt. Defaults to True.
- **pt_index** (*int, optional*) – index of the pT feature. Defaults to 2.

Returns
Jets of same type as input, of shape `[num_jets, num_particles, num_features (including mask)]` if `ret_mask_separate` is False, else a tuple with a tensor/array of shape `[num_jets, num_particles, num_features (excluding mask)]` and another binary mask tensor/array of shape `[num_jets, num_particles, 1]`.

`jetnet.utils.jet_features`(*jets: ndarray*) → dict[str, float | numpy.ndarray]

Calculates jet features by summing over particle Lorentz 4-vectors.

Parameters
jets (*np.ndarray*) – array of either a single or multiple jets, of shape either `[num_particles, num_features]` or `[num_jets, num_particles, num_features]`, with features in order `[eta, phi, pt, (optional) mass]`. If no particle masses given, they are assumed to be 0.

Returns
dict of float (if inputted single jet) or 1D arrays of length `num_jets` (if inputted multiple jets) with 'mass', 'pt', and 'eta' keys.

Return type
Dict[str, Union[float, np.ndarray]]

`jetnet.utils.`**`relEtaPhiPt_to_EtaPhiPt`**(*p_polarrel: np.ndarray | torch.Tensor*, *jet_features: np.ndarray | torch.Tensor*, *jet_coord: str = 'cartesian'*) → np.ndarray | torch.Tensor

Get particle features in absolute polar coordinates from relative polar coordinates and jet features.

Parameters

- **`p_polarrel`** (*`np.ndarray or torch.Tensor`*) – array of particle features in relative polar coordinates of shape `[..., 3]`. The last axis should be in order $(\eta^{\mathrm{rel}}, \phi^{\mathrm{rel}}, p_{\mathrm{T}}^{\mathrm{rel}})$, where η is the pseudorapidity.
- **`jet_features`** (*`np.ndarray or torch.Tensor`*) – array of jet features in polar coordinates, of shape `[..., 4]`. The coordinates are specified by `jet_coord`.
- **`jet_coord`** (*`str`*) – coordinate system of jet features. Can be either "cartesian" or "polar". Defaults to "cartesian". If "cartesian", the last axis of `jet_features` should be in order $(E/c, p_x, p_y, p_z)$. If "polar", the last axis of `jet_features` should be in order $(\eta, \phi, p_{\mathrm{T}}, E/c)$.

Returns
array of particle features in absolute polar coordinates, arranged in order $(\eta, \phi, p_{\mathrm{T}}, E/c)$.

Return type
np.ndarray or torch.Tensor

`jetnet.utils.`**`relEtaPhiPt_to_cartesian`**(*p_polarrel: np.ndarray | torch.Tensor*, *jet_features: np.ndarray | torch.Tensor*, *jet_coord: str = 'cartesian'*) → np.ndarray | torch.Tensor

Get particle features in absolute Cartesian coordinates from relative polar coordinates
and jet features.

Parameters

- **`p_polarrel`** (*`np.ndarray or torch.Tensor`*) – array of particle features in relative polar coordinates of shape `[..., 3]`. The last axis should be in order $(\eta^{\mathrm{rel}}, \phi^{\mathrm{rel}}, p_{\mathrm{T}}^{\mathrm{rel}})$, where η is the pseudorapidity.
- **`jet_features`** (*`np.ndarray or torch.Tensor`*) – array of jet features in polar coordinates, of shape `[..., 4]`. The coordinates are specified by `jet_coord`.
- **`jet_coord`** (*`str`*) – coordinate system of jet features. Can be either "cartesian" or "polar". Defaults to "cartesian". If "cartesian", the last axis of `jet_features` should be in order $(E/c, p_x, p_y, p_z)$. If "polar", the last axis of `jet_features` should be in order $(\eta, \phi, p_{\mathrm{T}}, E/c)$.

Returns
array of particle features in absolute polar coordinates, arranged in order $(E/c, p_x, p_y, p_z)$.

Return type
np.ndarray or torch.Tensor

`jetnet.utils.`**`to_image`**(*jets: ndarray*, *im_size: int*, *mask: ndarray = None*, *maxR: float = 1.0*) → ndarray

Convert jet(s) into 2D `im_size` x `im_size` or 3D `num_jets` x `im_size` x `im_size` image arrays.

Parameters

- **`jets`** (*`np.ndarray`*) – array of jet(s) of shape `[num_particles, num_features]` or `[num_jets, num_particles, num_features]` with features in order `[eta, phi, pt]`.
- **`im_size`** (*`int`*) – number of pixels per row and column.
- **`mask`** (*`np.ndarray`*) – optional binary array of masks of shape `[num_particles]` or `[num_jets, num_particles]`.
- **`maxR`** (*`float`*) – max radius of the jet. Defaults to 1.0.

Returns

2D or 3D array of shape `[im_size, im_size]` or `[num_jets, im_size, im_size]`.

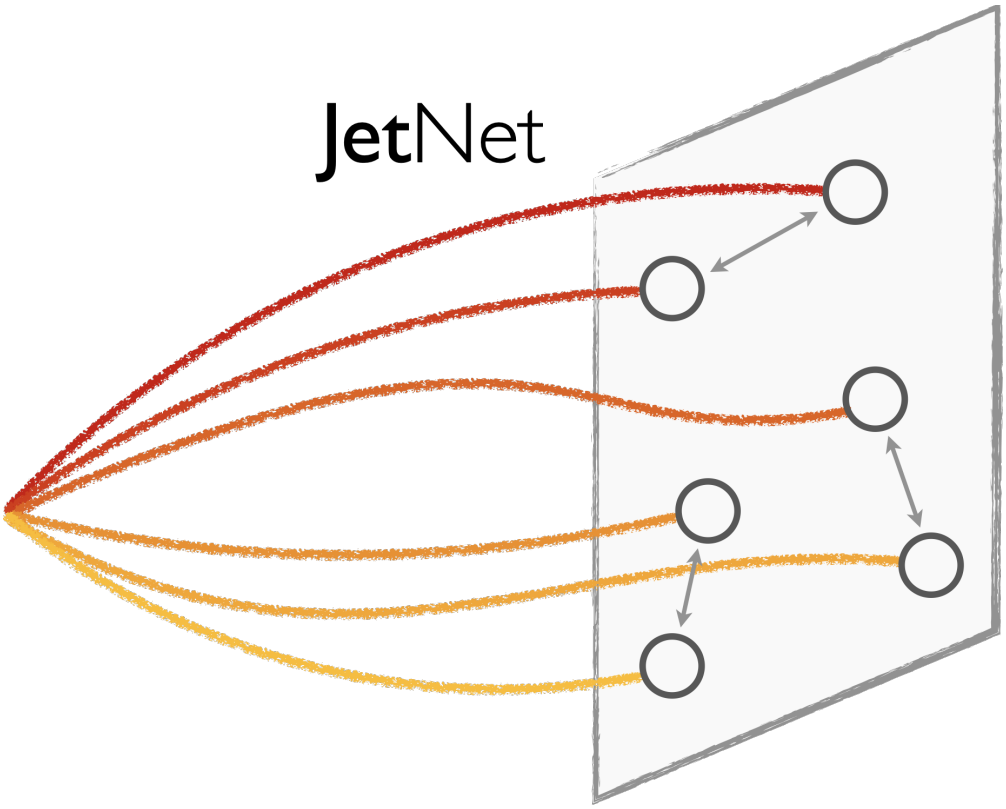
Return type

np.ndarray

CHAPTER

SIX

JETNET DEMO



Raghav KansalUC San Diego

PyHEP 2022 WorkshopOnline, 12-16 September 2022

JetNet: For developing and reproducing ML + HEP projects.

Repo: github.com/jet-net/JetNet

Docs: jetnet.readthedocs.io

Paper: 2106.11535

6.1 Introduction

6.1.1 Problems:

- How do I **get started** with machine learning in high energy physics?
- How do I **evaluate** my results?
- How do we **reproduce** and **compare** results?

6.1.2 Solution:

JetNet: Python package with easy-to-access datasets, standardised evaluation metrics, and more utilities for improving accessibility and reproducibility in ML + HEP.

Note: Still under development, with currently a limited number of datasets and metrics. Feedback and contributions welcome!

6.2 Today

- Loading and looking at the JetNet dataset
- Preparing the dataset for training a model

6.3 Data loading

We'll use the `jetnet.datasets.JetNet.getData` function to download and directly access the dataset.

First, we can check which particle and jet features are available in this dataset:

[1]:
```
from jetnet.datasets import JetNet

print(f"Particle features: {JetNet.ALL_PARTICLE_FEATURES}")
print(f"Jet features: {JetNet.ALL_JET_FEATURES}")
```

```
Particle features: ['etarel', 'phirel', 'ptrel', 'mask']
Jet features: ['type', 'pt', 'eta', 'mass', 'num_particles']
```

Next, let's load the data:

[2]:
```
data_args = {
    "jet_type": ["g", "t", "w"],  # gluon, top quark, and W boson jets
    "data_dir": "datasets/jetnet",
    # only selecting the kinematic features
    "particle_features": ["etarel", "phirel", "ptrel"],
    "num_particles": 30,
    "jet_features": ["type", "pt", "eta", "mass"],
    "download": True,
}

particle_data, jet_data = JetNet.getData(**data_args)
```

Let's look at some of the data:

[3]:
```
print(
    f"Particle features of the 10 highest pT particles in the first jet\n{data_args[
↪'particle_features']}\n{particle_data[0, :10]}"
)
print(f"\nJet features of first jet\n{data_args['jet_features']}\n{jet_data[0]}")
```

```
Particle features of the 10 highest pT particles in the first jet
['etarel', 'phirel', 'ptrel']
[[-0.04361616 -0.00706771  0.29305124]
 [-0.04611618 -0.00956919  0.06966697]
 [-0.04163383 -0.00890653  0.05733829]
 [ 0.13638385 -0.00706771  0.04643776]
 [-0.04111616 -0.0045667   0.04290354]
 [-0.04223531  0.00299934  0.03603047]
 [ 0.10638386  0.01294228  0.03550573]
 [-0.0461162  -0.01457169  0.03525265]
 [-0.04251299 -0.00919492  0.02895915]
 [-0.04227024 -0.01043073  0.02826967]]

Jet features of first jet
['type', 'pt', 'eta', 'mass']
[3.00000000e+00 1.13473572e+03 6.48616195e-01 8.08584366e+01]
```

We can also visualise these jets as images:

[4]:
```
from jetnet.utils import to_image
import matplotlib.pyplot as plt

num_images = 5
num_types = len(data_args["jet_type"])
im_size = 25  # number of pixels in height and width
maxR = 0.4  # max radius in (eta, phi) away from the jet axis

cm = plt.cm.jet.copy()
cm.set_under(color="white")
plt.rcParams.update({"font.size": 16})

fig, axes = plt.subplots(
    nrows=num_types,
    ncols=num_images,
    figsize=(40, 8 * num_types),
    gridspec_kw={"wspace": 0.25},
)

# get the index of each jet type using the JetNet.JET_TYPES array
type_indices = {jet_type: JetNet.JET_TYPES.index(jet_type) for jet_type in data_args[
↪"jet_type"]}

for j in range(num_types):
    jet_type = data_args["jet_type"][j]
    type_selector = jet_data[:, 0] == type_indices[jet_type]  # select jets based on jet_
↪type feat
```

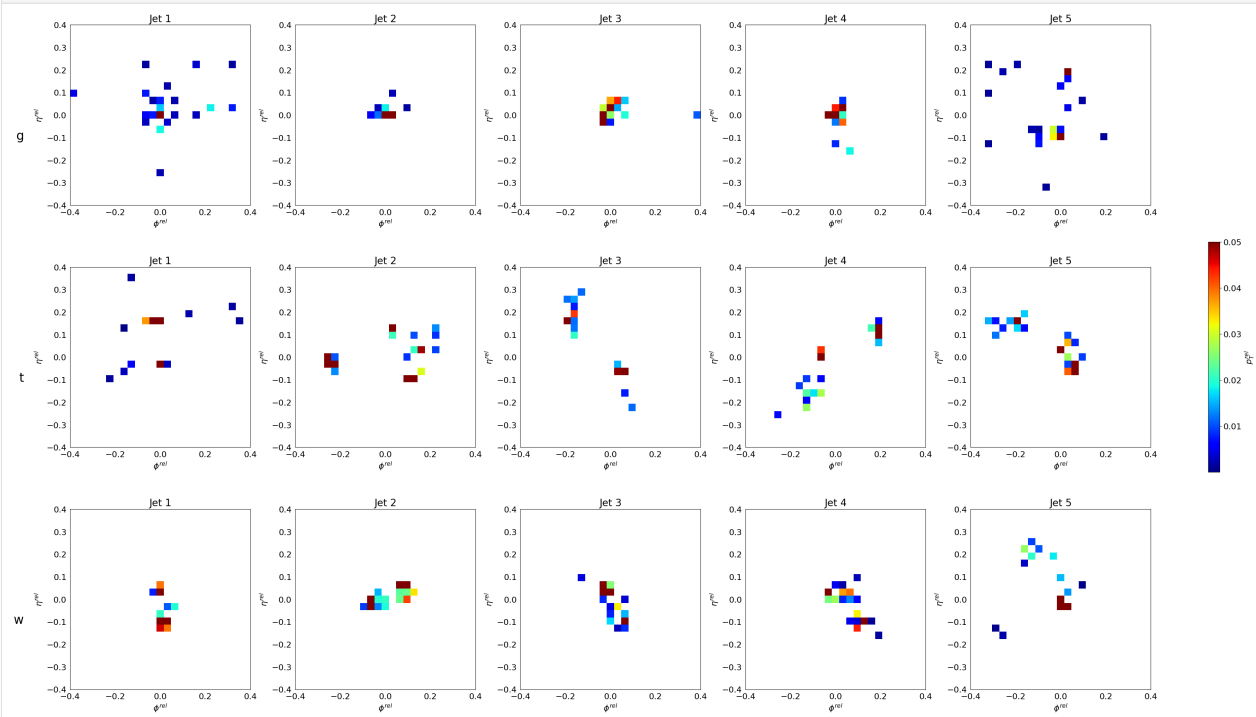
(continues on next page)

(continued from previous page)

```
    axes[j][0].annotate(
        jet_type,
        xy=(0, -1),
        xytext=(-axes[j][0].yaxis.labelpad - 15, 0),
        xycoords=axes[j][0].yaxis.label,
        textcoords="offset points",
        ha="right",
        va="center",
        fontsize=24,
    )

    for i in range(num_images):
        im = axes[j][i].imshow(
            to_image(particle_data[type_selector][i], im_size, maxR=maxR),
            cmap=cm,
            interpolation="nearest",
            vmin=1e-8,
            extent=[-maxR, maxR, -maxR, maxR],
            vmax=0.05,
        )
        axes[j][i].tick_params(which="both", bottom=False, top=False, left=False,␣
↪right=False)
        axes[j][i].set_xlabel("$\phi^{rel}$")
        axes[j][i].set_ylabel("$\eta^{rel}$")
        axes[j][i].set_title(f"Jet {i + 1}")

cbar = fig.colorbar(im, ax=axes.ravel().tolist(), fraction=0.01)
cbar.set_label("$p_T^{rel}$")
```



And calculate and plot their overall features:

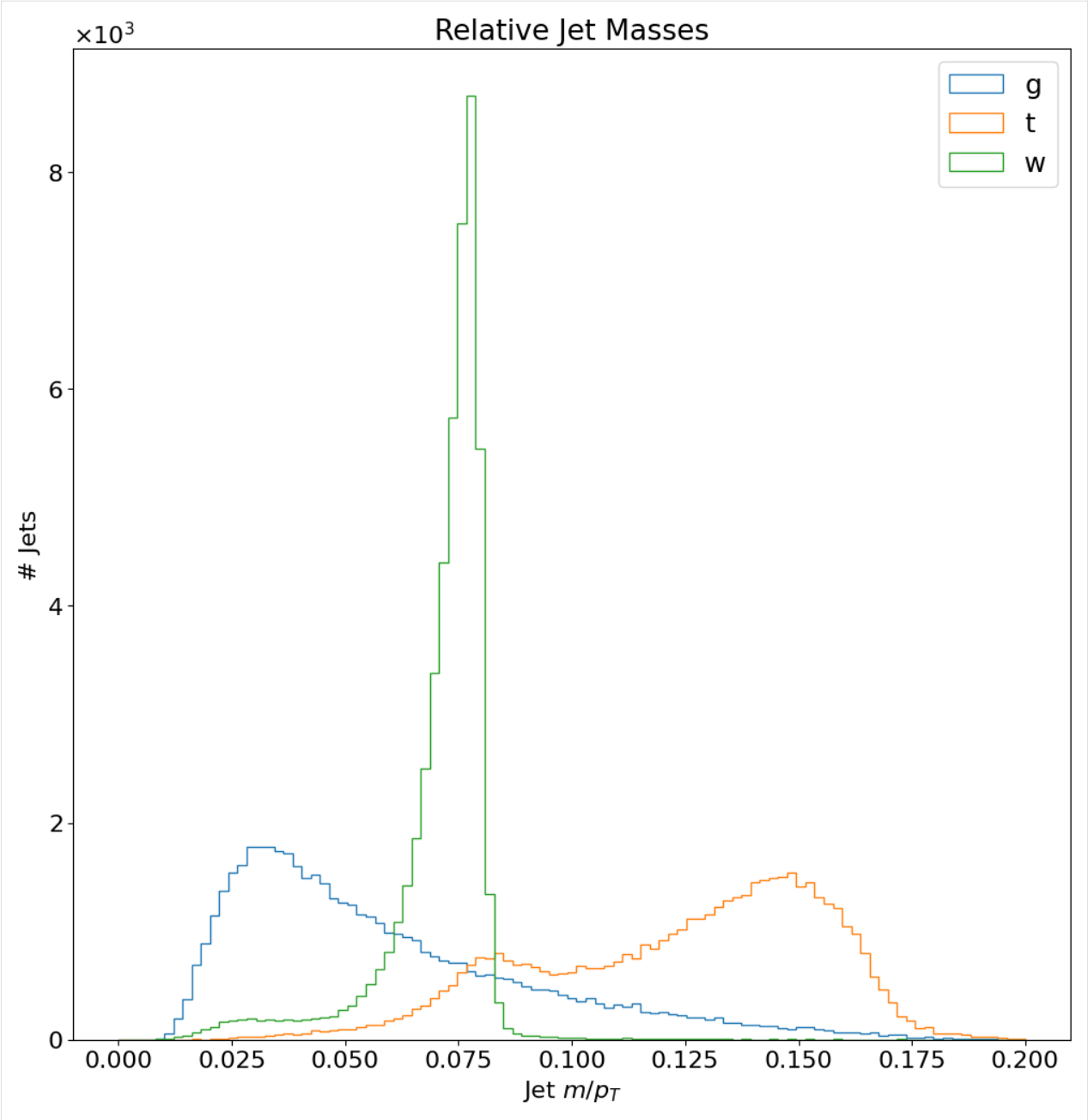
[5]:
```
from jetnet.utils import jet_features
import numpy as np

fig = plt.figure(figsize=(12, 12))
plt.ticklabel_format(axis="y", scilimits=(0, 0), useMathText=True)

for j in range(num_types):
    jet_type = data_args["jet_type"][j]
    type_selector = jet_data[:, 0] == type_indices[jet_type]  # select jets based on jet_
↪type feat

    jet_masses = jet_features(particle_data[type_selector][:50000])["mass"]
    _ = plt.hist(jet_masses, bins=np.linspace(0, 0.2, 100), histtype="step", label=jet_
↪type)

plt.xlabel("Jet $m/p_{T}$")
plt.ylabel("# Jets")
plt.legend(loc=1, prop={"size": 18})
plt.title("Relative Jet Masses")
plt.show()
```



6.4 Dataset preparation

To prepare the dataset for machine learning applications, we can use the `jetnet.datasets.JetNet` class itself, which inherits the `pytorch.data.utils.Dataset` class.

We'll also use the class to **normalise** the features to have zero means and unit standard deviations, and **transform** the jet type feature to be one-hot-encoded.

[6]:
```
from jetnet.datasets import JetNet
from jetnet.datasets.normalisations import FeaturewiseLinear
```

(continues on next page)

(continued from previous page)

```
import numpy as np
from sklearn.preprocessing import OneHotEncoder


# function to one hot encode the jet type and leave the rest of the features as is
def OneHotEncodeType(x: np.ndarray):
    enc = OneHotEncoder(categories=[[0, 1]])
    type_encoded = enc.fit_transform(x[..., 0].reshape(-1, 1)).toarray()
    other_features = x[..., 1:].reshape(-1, 3)
    return np.concatenate((type_encoded, other_features), axis=-1).reshape(*x.shape[:-1],
↪ -1)


data_args = {
    "jet_type": ["g", "t"],  # gluon and top quark jets
    "data_dir": "datasets/jetnet",
    # these are the default particle features, written here to be explicit
    "particle_features": ["etarel", "phirel", "ptrel", "mask"],
    "num_particles": 10,  # we retain only the 10 highest pT particles for this demo
    "jet_features": ["type", "pt", "eta", "mass"],
    # we don't want to normalise the 'mask' feature so we set that to False
    "particle_normalisation": FeaturewiseLinear(
        normal=True, normalise_features=[True, True, True, False]
    ),
    # pass our function as a transform to be applied to the jet features
    "jet_transform": OneHotEncodeType,
    "download": True,
}

jets_train = JetNet(**data_args, split="train")
jets_valid = JetNet(**data_args, split="valid")
```

We can look at one of our datasets to confirm everything is as we expect:

[7]:
```
jets_train
```

[7]:
```
Dataset JetNet
    Number of datapoints: 248637
    Data location: datasets/jetnet
    Including ['g', 't'] jets
    Split into train data out of ['train', 'valid', 'test', 'all'] possible splits, with␣
↪splitting fractions [0.7, 0.15, 0.15]
    Particle features: ['etarel', 'phirel', 'ptrel', 'mask'], max 10 particles per jet
    Jet features: ['type', 'pt', 'eta', 'mass']
    Particle normalisation: Normalising features to zero mean and unit standard␣
↪deviation, normalising features: [True, True, True, False]
    Jet transform: <function OneHotEncodeType at 0x163cd32e0>
```

And also directly at the data itself - note that the features have been **normalised** and the jet type has been **one-hot-encoded**):

[8]:
```
particle_features, jet_features = jets_train[0]
```

(continues on next page)

(continued from previous page)

```
print(f"Particle features ({data_args['particle_features']}):\n\t{particle_features}")
print(f"\nJet features ({data_args['jet_features']}):\n\t{jet_features}")
```

```
Particle features (['etarel', 'phirel', 'ptrel', 'mask']):
        tensor([[-1.5952e-03, -9.4181e-04,  6.7592e-01,  1.0000e+00],
        [ 1.3819e-03,  6.9232e-03,  9.6110e-02,  1.0000e+00],
        [ 5.9048e-03, -3.4432e-03,  9.1700e-02,  1.0000e+00],
        [ 1.4783e-02, -1.0506e-02,  2.8433e-02,  1.0000e+00],
        [ 1.3316e-03, -8.1813e-03,  2.6264e-02,  1.0000e+00],
        [ 9.0482e-04,  1.1564e-02,  1.6956e-02,  1.0000e+00],
        [-1.4095e-02,  1.1564e-02,  1.3759e-02,  1.0000e+00],
        [ 2.5905e-02, -3.4432e-03,  1.1798e-02,  1.0000e+00],
        [-4.0952e-03,  6.5619e-03,  9.9370e-03,  1.0000e+00],
        [-9.0952e-03,  3.1575e-02,  8.2691e-03,  1.0000e+00]])

Jet features (['type', 'pt', 'eta', 'mass']):
        tensor([ 1.0000e+00,  0.0000e+00,  1.2301e+03, -1.7340e-01,  2.2097e+01])
```

We can now feed this into a PyTorch DataLoader and start training!

Next things you can try are: - Repeat this with the Top Quark Tagging (`jetnet.datasets.TopTagging`) and Quark Gluon datasets (`jetnet.datasets.QuarkGluon`) - Training an ML model (tutorial coming soon…) - Evaluating generative models (`jetnet.evaluation`)

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

j

INDEX