neuralxc Documentation

neuralxc

Jul 05, 2021

Contents:

1 Installation **3**

2 Quickstart **5**

3 Filetypes **9**

4 Self-consistent training **11**

5 CLI **13**
5.1 Data . . . 13
5.2 Model . . . 14
5.3 Other . . . 15

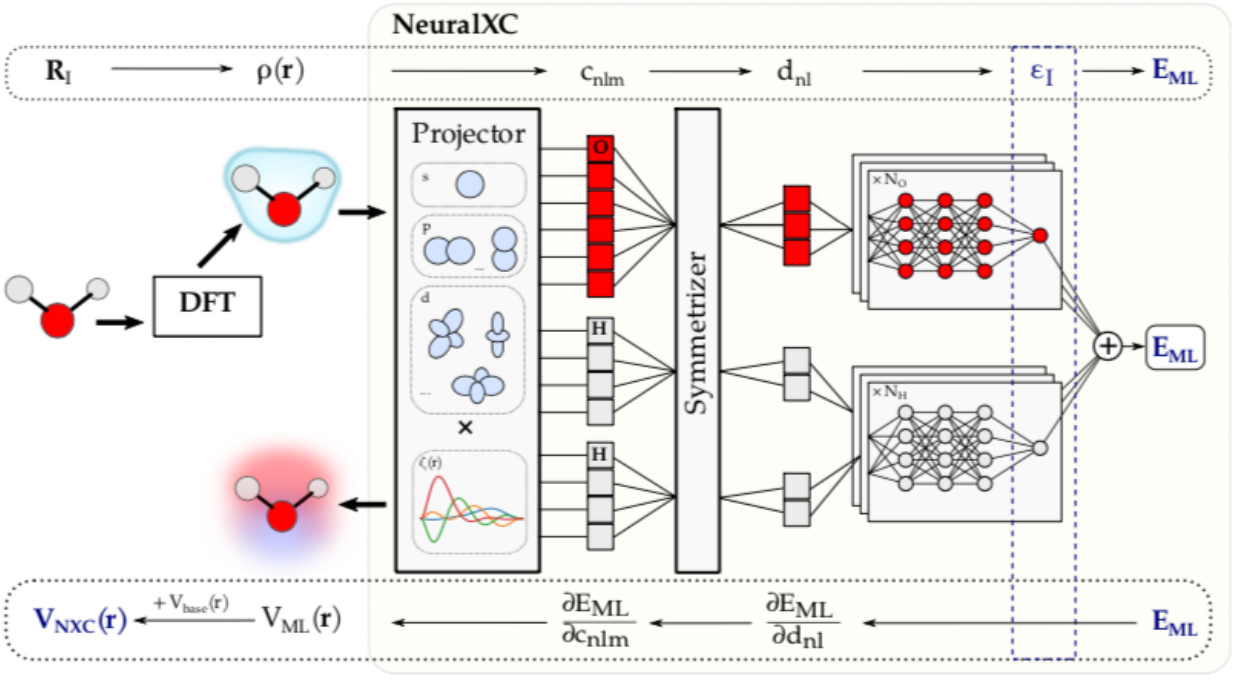
6 Projector **17**
6.1 Base classes and inheritance . . . 18
6.2 Radial basis . . . 19
6.3 PySCF projector . . . 20

7 Symmetrizer **21**

8 Configuration Files **23**
8.1 config.json . . . 23
8.2 hyper.json . . . 24

9 Indices and tables **27**

Index **29**



Implementation of a machine learned density functional as presented in Machine learning accurate exchange and correlation functionals of the electronic density. Nat Commun 11, 3509 (2020)

NeuralXC only includes routines to fit and test neural network based density functionals. To use these functionals for self-consistent calculations within electronic structure codes please refer to Libnxc.

The basic premise of NeuralXC can be summarized as follows:

1. The electron density on a real space grid is projected onto a set of atom-centered atomic orbitals
2. The projection coefficients are symmetrized to ensure that systems that only differ by a global rotation have the same energy
3. The symmetrized coefficients are fed through a neural network architecture that is invariant under atom permutations, similar to Behler-Parrinello networks.
4. The output of this neural network is the exchange-correlation (XC) energy (correction) for a given system. The XC-potential is then obtained by backpropagating through steps 1-3.

The very nature of this approach lends itself to a modular implementation. Hence, we have separated NeuralXC into three main modules, **Projector**, **Symmetrizer**, and **ML**, each of which can be individually customized.

CHAPTER 1

Installation

NeuralXC can be obtained free-of-charge from our Github repository here.

To install NeuralXC using pip, after cloning the repository to your local machine, navigate into the root directory of the repository and run:

```
sh install.sh
```

So far, NeuralXC has only been tested on Linux and Mac OS X.

To check the integrity of the installation, unit tests can be run with:

```
pytest -v
```

in the root directory. The installation succeeded if the pytest summary shows no failed test:

```
=========  25 passed, 19 warnings in 47.56s =========
```

In case of reported failures, try re-downloading and re-installing the library. If problem persists, please raise an issue on the github repository.

CHAPTER 2

Quickstart

The new version of NeuralXC only implements the neural network architecture along with routines to train and test functionals. As neural networks are trained self-consistently, an electronic structure code to drive these calculations is needed. For this purpose, we have developed Libnxc, which allows for easy interfacing with electronic structure codes such as SIESTA and CP2K. Its python version, pylibnxc is installed automatically together with this package and works with PySCF out-of-the-box.

To get accustomed with NeuralXC, we recommend that PySCF is used as the driver code. Examples on how to train and deploy a machine learned functional can be found in `examples/example_scripts/`.

In this tutorial we use files contained in `examples/quickstart/`. To begin, navigate into this directory.

To train/fit a functional a set of structures and their associated reference energies is required. These structures need to be provided in an ASE formatted `.xyz` or `.traj` file (in this example training_structures.xyz).

Self-consistent training can then be performed by running:

```
neuralxc sc training_structures.xyz config.json hyperparameters.json --hyperopt
```

- **config.json** contains information regarding the basis set as well as the 'driver' program (PySCF), other examples can be found in examples/inputs/ml_basis/.
- **hyperparameters.json** contains the machine learning hyperparameters, other examples can be found in examples/inputs/hyper.

A minimal input file structure would look something like this:

config.json

```
{
  "preprocessor":
  {
        "basis": {
                "file": "quickstart-basis"
        },
        "projector": "gaussian",
        "grid": "analytical",
```

(continues on next page)

(continued from previous page)

```
        "extension": "chkpt"
  },
  "n_workers" : 1,
  "engine": {"xc": "PBE",
             "application": "pyscf",
             "basis" : "def2-TZVP"}

}
```

hyperparameters.json

```
{
 "hyperparameters": {
     "var_selector__threshold": 1e-10,
     "estimator__n_nodes": 4,
     "estimator__n_layers": 0,
     "estimator__b": [0, 0.1, 0.001],
     "estimator__alpha": 0.001,
     "estimator__max_steps": 20001,
     "estimator__valid_size": 0,
     "estimator__batch_size": 0,
     "estimator__activation": "GeLU"
 },
    "cv": 4
}
```

A detailed explanation of these files is given in *Configuration Files*.

NeuralXC will train a model self-consistently on the provided structures. This means an initial model is fitted to the reference energies. This model is then used to run self-consistent calculations on the dataset producing updated baseline energies. Another model is fitted on the difference between the reference and updated baseline energies and self-consistent calculations are run with the new model. This is done iteratively until the model error converges within a given tolerance. This tolerance can be set with the `--tol` flag, the default is 0.5 meV.

At the end of the self-consistent training process a `final_model.jit` is produced that can be used by Libnxc. If either `testing.traj` or `testing.xyz` is found in the work directory self-consistent calculations are run for these structures using the final model and the error on the test set is reported. In our example, the final MAE should be below 10 meV.

The final model can then be used to perform self-consistent calculations on other systems. This can be done by utilizing Libnxc to run standard DFT calculations while accessing NeuralXC models. However, in case testing needs to be conducted across other datasets (e.g. the structures stores in `more_testing.xyz`), it is easier to do so using the following command:

```
neuralxc engine config_with_model.json more_testing.xyz
```

`config_with_model.json` is identical to the original `config.json` except for instructions to use `final_model.jit`. This command will run self-consistent calculations for every structure contained in the xyz file while saving the resulting energies in `results.traj`. In order to quickly evaluate error metrics we can also use NeuralXC:

```
neuralxc data add data.hdf5 more_testing final_model energy --traj results.traj
neuralxc data add data.hdf5 more_testing reference energy --traj more_testing.traj
```

Will add both refernce values and the ones obtained with our NeuralXC functionals to a newly created `data.hdf5`:

```
neuralxc eval data.hdf5 more_testing/final_model more_testing/reference --plot
```

will print error statistics and show a correlation plot.

CHAPTER 3

Filetypes

NeuralXC uses three different file types.

System geometries can either be provided as a `.xyz` or an atomic simulation environment (ASE) native `.traj` file. Both file types should be readable by ASE and if energies are needed, which is almost always the case, the files need to be formatted so that the ASE method `get_potential_energy()` returns the appropriate value.

Configuration files need to be provided in a `.json` format. Two different kinds of configuration files are used within NeuralXC. **config.json** defines the projection basis set as well as instructions to the engine (the electronic structure code). **hyper.json** contains the hyperparameters used for the machine learning model. Both configuration file types are discussed in detail further below.

Processed data is generally stored in hierarchical data format `.hdf5` files. These files are subdivided by groups which can (but do not have to) indicate the data set name (e.g. "ethanol") and the method used to generate the data (e.g. "PBE" or "CCSD") separated by a "/".

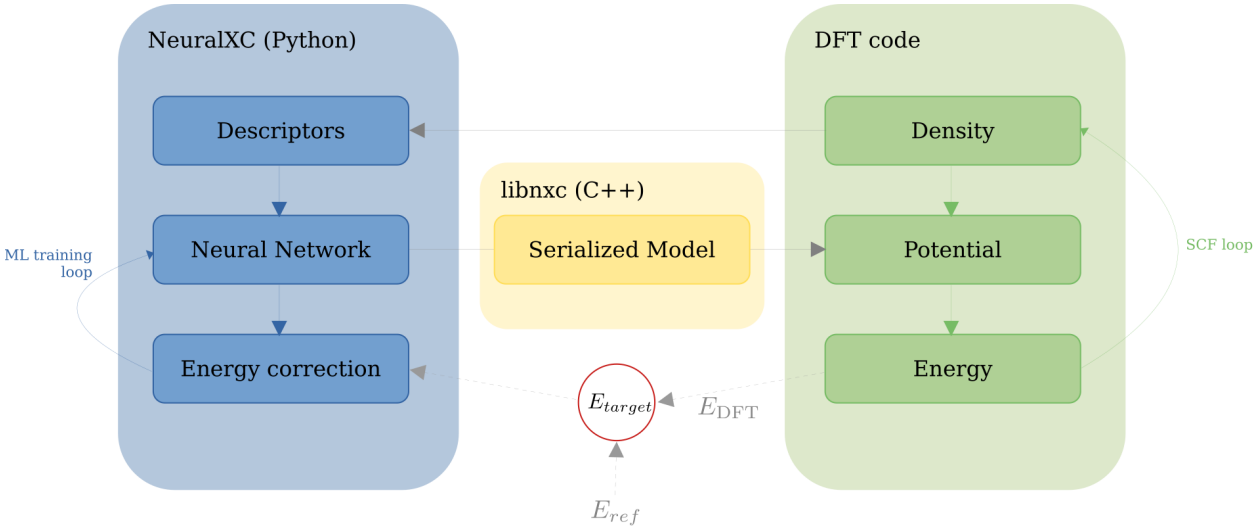
The content of a typical file will look like this:

```
water/PBE/energy
water/PBE/density/05e5598680faeecf6f5d8ebe4283e76d
water/PBE/forces
water/CCSD/energy
ethanol/PBE/energy
ethanol/PBE/density/05e5598680faeecf6f5d8ebe4283e76d
ethanol/CCSD/energy
```

We will refer to this example file when discussing *CLI* commands. The hash-codes shown act as unique identifiers for the basis set chosen to conduct the density projection and are automatically created from the content of the `config.json` input file.

CHAPTER 4

Self-consistent training



In order to use NeuralXC functionals within DFT calculations, models need to be trained self-consistently on the provided structures. This means, an initial model is fit to the reference energies. This model is then used to run self-consistent calculations on the dataset producing updated baseline energies. The model is re-fitted on the difference between the reference and updated baseline energies and self-consistent calculations are run with the updated model. This is done iteratively until the model error converges within a given tolerance.

The self-consistent training procedure requires that a set commands is executed in a well-defined order. In addition to running the self-consistent calculations which requires communication with an electronic structure code, data needs to be tracked, models optimized and error statistics need to be collected.

In order to facilitate self-consistent training, NeuralXC provides a simple command that combines the required steps and automatically produces a final model that is ready for use within DFT calculations:

```
neuralxc sc <xyz> <config> <hyper>
```

conducts self-consistent training for the structures specified in a system geometry file <*xyz*> using the engine and basis set specified in <*config*> and ML hyperparameters provided in <*hyper*>. Optional flags include:

```
--maxit <int>    Maximum number of iterations,
                   if zero create an energy-correcting model only (default: 5)
--tol <float>    Energy tolerance in eV that
                   determines whether self-consistent training has converged (default␣
↪0.0005)
--hyperopt       If set, optimize hyperparameters
                   by cross-validation at every ML training iteration
```

CHAPTER 5

CLI

The recommended way for users to interact with NeuralXC is through the command line interface (CLI). While the self-consistent training command `neuralxc sc` introduced above should cover 95% of all use cases, sometimes a more fine-grained control flow is warranted. This can be achieved by utilizing the following commands which can be grouped into three categories: Data, Model and Other.

5.1 Data

Commands in this category deal with managing data, i.e. input features along with target energies. All data-related commands are prefaced with:

```
neuralxc data
```

so that, e.g. in order to add data to an `.hdf5` file, the command:

```
neuralxc data add <args>
```

needs to be executed.

We provide a complete list of these commands, with required arguments shown within the command prompt and optional arguments listed underneath.

add

```
add <hdf5> <system> <method> <add>
```

Store data to file `<hdf5>` under the group `<system>/<method>/`. The quantity to add is specified as `<add>` and can be either `energy`, `forces` or `density`. If adding energies or forces `--traj <str>` needs to be set to point to an `.xyz` or `.traj` file containing the required quantity. If adding densities, `--density <str>` needs to be set to the path were density projections are stored.

`--zero <float>` Shift energies by this value. If not set, shifts energies so that minimum of dataset is zero.

Example: `neuralxc data add data.hdf5 water PBE energy --traj water_pbe.traj`

delete

```
delete <hdf5> <group>
```

Delete data from file `<hdf5>` within group `<group>`. Cannot be reversed.

Example: `neuralxc data delete water/PBE`

split

```
split <hdf5> <group> <label>
```

Split slice off data from file `<hdf5>` within group `<group>`. Slicing can be provided in numpy notation by setting `--slice <str>`. `--comp <str>` stores the complementary slice as its own dataset

Example: `neuralxc data split data.hdf5 water/PBE training --slice :15 --comp testing`

This splits of the first 15 datapoints from water/pbe stored in data.hdf5, stores it as training and stores the remaining datapoints as testing.

sample

```
sample <config> <size> --hdf5 <hdf5> --dest <dest>
```

Sample `<size>` data points for the basis set defined in `<config>` from `<hdf5>`, saving it to `<dest>` using k-means clustering in feature space.

Example: `neuralxc data sample config.json 50 --hdf5 data.hdf5/water/PBE --dest sample_50.npy`

5.2 Model

Commands in this category deal with the machine learning model, they are prefaced with

```
neuralxc
```

fit

```
fit <config> <hyper> --hdf5 <path> <baseline> <reference>
```

Use features generated with basis defined in `<config>` and hyperparameters defined in `<hyper>` to fit a neuralxc model that corrects `<baseline>` data in hdf5 file found at `<path>` using targets given by `<reference>`.

`--model <str>` Continue training model found at this location `--hyperopt` If set, conduct hyperparameter optimization.

Example: `neuralxc fit config.json hyper.json --hdf5 data.hdf5 water/PBE water/CCSD`

eval

```
eval --hdf5 <path> <baseline> <reference>
```

Evaluate accuracy of `<baseline>` with respect to `<reference>`

`--model <str>` If set, correct baseline with this model before evaluation. `--plot` Create error histogram and correlation plot. `--sample <str>` Only evaluate on this sample (.npy file containing integer indices) `--keep_mean` If set, don't subtract parallelity errors.

Example: `neuralxc eval --hdf5 data.hdf5 water/PBE water/CCSD --model best_model`

predict

```
predict --model <model> --hdf5 <hdf5>
```

Predict energy corrections to data in `<hdf5>` using `<model>`.

`--dest <str>` Store to this location (default: prediction.npy)

Example: `neuralxc predict --model best_model --hdf5 data.hdf5/water/PBE`

serialize

```
serialize <in_path> <jit_path>
```

Serialize model found at `<in_path>` and store to `<jit_path>` to be used with libnxc.

`--as_radial` serializes model to be used with radial grids.

5.3 Other

Commands in this category deal with running and processing SCF calculations, they are prefaced with

```
neuralxc
```

engine

```
engine <config> <xyz>
```

Run engine (electronic structure code) specified in `<config>` for all molecules contained in `<xyz>`. Stores results (energies) of calculations in `results.traj`

`--workdir <str>` Specify work-directory. Default is to use .tmp/ and delete after calculation has finished

default

```
default <kind>
```

Generates a default input file either containing basis set information (`<kind> = pre`) or hyperparameters (`<kind> = hyper`)

preprocess

```
pre <config> --xyz <xyz> --dest <dest> --srcdir <srcdir>
```

Preprocesses (projects) electron densities found at `<srcdir>` for systems found in the `<xyz>` .xyz or .traj file and stores features in '' <dest>'' (a hdf5 file path with group name).

Example: `neuralxc pre config.json --xyz water_pbe.traj --dest data.hdf5 water/PBE --srcdir workdir/`

CHAPTER 6

Projector

A crucial step in NeuralXC is the creation of machine learning features c_α by projecting the density $n(\mathbf{r})$ onto a set of basis functions $\psi_\alpha(\mathbf{r})$

$$c_\alpha = \int_{\mathbf{r}} \psi_\alpha(\mathbf{r}) n(\mathbf{r}).$$

Here, α summarizes the quantum numbers n, l, m. In practice, this integration is either performed on a grid, or, if both density matrix and projection basis can be expressed in terms of Gaussian type orbitals (GTO), analytically.

For grid integrations, NeuralXC distinguishes between three dimensional euclidean grids

$$c_\alpha = V_{\text{grid}} \cdot \sum_x \sum_y \sum_z \psi_{\alpha,xyz} \cdot n_{xyz}$$

and radial grids

$$c_\alpha = \sum_i \psi_{\alpha,i} \cdot n_i \cdot w_i$$

with integration weights w_i. For euclidean grids, NeuralXC automatically assumes periodic boundary conditions. Radial grids are only supported for finite systems.

If the density can be expressed in terms of GTOs $\phi(\mathbf{r})$ with density matrix $\rho_{\mu\nu}$

$$n(\vec{r}) = \sum_{\mu\nu} \rho_{\mu\nu} \phi_\mu(\mathbf{r}) \phi_\nu(\mathbf{r})$$

then the projection can be computed analytically as

$$c_\alpha = \sum_{\mu\nu} \rho_{\mu\nu} \int_{\mathbf{r}} \psi_\alpha(\mathbf{r}) \phi_\mu(\mathbf{r}) \phi_\nu(\mathbf{r}).$$

NeuralXC takes advantage of this to avoid grid computations when working with PySCF cite{pyscf}.

Beyond the type of integration methods, Projectors are defined by their radial basis functions (the angular part is always given by spherical harmonics).

These can either be polynomials

$$\zeta_n(r) = \begin{cases} \frac{1}{N} r^2 (r_o - r)^{n+2} \text{ for } r < r_o \\ \text{else} \end{cases}$$

or GTOs.

For polynomial basis functions the user needs to specify the number of radial functions n, the number of spherical shells l (this is the maximum angular momentum l_{max} plus one) and an outer cutoff r_o in the configuration file. Basis sets can either be specified on a per-element basis or a single, species independent, basis set can be used across elements.

For GTO basis functions, the user can either provide a basis-set name (same nomenclature as used by PySCF) such as "6-311G*" or a path pointing to a file containing a NWChem cite{nwchem} formatted basis definition (see Basis Set Exchange cite{bse}). If the projection is done on a grid (as opposed to analytically) the Gaussians should be localized to have finite support

$$\begin{aligned} r_o &= \sigma \cdot \alpha^{1/2} \cdot (1 + \frac{l}{5}) \\ \tilde{r} &= \frac{r}{\gamma} \\ f_c(r) &= 1 - \left[0.5 \cdot \left\{ 1 - \cos(\pi \frac{r}{r_o}) \right\} \right]^8 \\ \zeta(r) &= \begin{cases} \tilde{r}^l \exp(-\kappa \tilde{r}^2) \cdot f_c(\tilde{r}) & \text{for } r < r_o \\ 0 & \text{else} \end{cases} \end{aligned}$$

We have introduced two parameters that can be tuned by the user to control the localization of the basis functions. σ (default: 2.0) controls the effective cutoff radius as a function of the Gaussian exponent κ and the angular momentum l. γ (default: 1.0) allows the user to re-scale the Gaussian functions.

The type of radial basis to be used can be specified in the configuration file using the keyword

`projector`

- `ortho` Polynomial basis
- `gaussian` GTO basis

In addition, the grid (or lack thereof) has to be specified through the keyword

`grid`

- `euclidean` Euclidean grid
- `radial` Radial grid
- `analytical` Perform integrals analytically

6.1 Base classes and inheritance

All real space projectors are either derived from EuclideanProjector or RadialProjector. The former implements density projections on a euclidean grid with periodic boundary conditions. The latter can be used for projections on flexible grids for which grid coordinates and integration weights are explicitly provided (as e.g. in PySCF). Both of these projector types inherit from the BaseProjector class.

class neuralxc.projector.projector.**BaseProjector**

get_basis_rep(*rho*, *positions*, *species*, ***kwargs*)
Calculates the basis representation for a given real space density

Parameters

- **rho** (*np.ndarray float (npoints) or (xpoints, ypoints, zpoints)*) – Electron density in real space
- **positions** (*np.ndarray float (natoms, 3)*) – atomic positions
- **species** (*list string*) – atomic species (chem. symbols)

Returns **c** – Basis representation, dict keys correspond to atomic species.

Return type dict of np.ndarrays

class neuralxc.projector.projector.**EuclideanProjector**(*unitcell*, *grid*, *basis_instructions*, ***kwargs*)

__init__(*unitcell*, *grid*, *basis_instructions*, ***kwargs*)
Projector on euclidean grid with periodic bounday conditions

Parameters

- **unitcell** (*numpy.ndarray float (3,3)*) – Unitcell in bohr
- **grid** (*numpy.ndarray float (3)*) – Grid points per unitcell
- **basis_instructions** (*dict*) – Instructions that define basis

class neuralxc.projector.projector.**RadialProjector**(*grid_coords*, *grid_weights*, *basis_instructions*, ***kwargs*)

__init__(*grid_coords*, *grid_weights*, *basis_instructions*, ***kwargs*)
Projector for generalized grid (as provided by e.g. PySCF). More flexible than euclidean grid as only grid point coordinates and their integration weights need to be provided, however does not support periodic boundary conditions. Special use case: Radial grids, as used by all-electron codes.

Parameters

- **grid_coords** (*numpy.ndarray (npoints, 3)*) – Coordinates of radial grid points
- **grid_weights** (*numpy.ndarray (npoints)*) – Grid weights for integration
- **basis_instructions** (*dict*) – Instructions that defines basis

6.2 Radial basis

Starting from from these definitions, NeuralXC implements two projectors that differ in their radial basis functionals. OrthoProjector implements an orthonormal polynomial basis whereas GaussianProjector uses Gaussian type orbitals similar to those used in quantum chemistry codes. Both projectors come in a euclidean and radial version.

class neuralxc.projector.polynomial.**OrthoRadialProjector**(*grid_coords*, *grid_weights*, *basis_instructions*, ***kwargs*)

_registry_name 'ortho_radial'

class neuralxc.projector.gaussian.**GaussianRadialProjector**(*grid_coords*, *grid_weights*, *basis_instructions*, ***kwargs*)

_registry_name 'gaussian_radial'

6.3 PySCF projector

If GTO orbitals in both projection and DFT calculation, projection integrals can be computed analytically. For this purpoes we have implemented a projector that works with PySCF. Future version of NeuralXC will implement a more general density matrix projector class that works with other gto codes as well.

class neuralxc.projector.pyscf.**PySCFProjector**(*mol*, *basis_instructions*, ***kwargs*)

_registry_name 'pyscf'

__init__(*mol*, *basis_instructions*, ***kwargs*)

Projector class specific to usage with PySCF. Instead of working with electron density on real space grid, density matrix is projected using analytical integrals.

Parameters

- **mol** (*pyscf.gto.M*) – Contains information about atoms and GTO basis
- **basis_instructions** (*dict*) –

  Basis instructions containing following values:

  - **spec_agnostic, bool (False)** Use same basis for every atomic species?
  - **operator, {'delta', 'rij'} ('delta')** Operator in overlap integral used for projection, delta means standard 3-center overalp, rij with coulomb kernel.
  - **delta, bool (False)** Use delta density (atomic density subracted)
  - **basis, str** Either name of PySCF basis (e.g. ccpvdz-jkfit) or file containing basis.

get_basis_rep(*dm*, ***kwargs*)

Project density matrix dm onto set of basis functions and return the projection coefficients (coeff)

CHAPTER 7

Symmetrizer

In order for the energy (the model output) to be invariant with respect to global rotations NeuralXC symmetrizes the descriptors c_{nlm}. Two symmetrizers are currently supported by NeuralXC and can be set with the keyword

symmetrizer_type

- `trace` $d_{nl} = \sum_m c_{nlm}^2$
- `mixed_trace` $d_{nn'l} = \sum_m c_{nlm} c_{n'lm}$

All Symmetrizer classes are derived from BaseSymmetrizer

class neuralxc.symmetrizer.symmetrizer.**BaseSymmetrizer**(*symmetrize_instructions*)

__init__(*symmetrize_instructions*)
Symmetrizer :param symmetrize_instructions: Attributes needed to symmetrize input (such as angular momentum etc.) :type symmetrize_instructions: dict

get_symmetrized(*C*)
Returns a symmetrized version of the descriptors c (from DensityProjector)

Parameters **C** (*dict of numpy.ndarrays or list of dict of numpy.ndarrays*) – Electronic descriptors

Returns **D** – Symmetrized descriptors

Return type dict of numpy.ndarrays

Customized Symmetrizers can be created by inheriting from this base class and implementing the method *_symmetrize_function*. As of now two symmetrizers are implemented by default:

class neuralxc.symmetrizer.symmetrizer.**TraceSymmetrizer**(**args*, ***kwargs*)
Symmetrizes density projections with respect to global rotations.

_registry_name 'trace'

static _symmetrize_function(*c*, *n_l*, *n*, **args*)
Returns the symmetrized version of c

Parameters

- **c** (*np.ndarray of floats*) – Stores the tensor elements in the order (n,l,m)
- **n_l** (*int*) – number of angular momenta (not equal to maximum ang. momentum! example: if only s-orbitals n_l would be 1)
- **n** (*int*) – number of radial functions

Returns Casimir invariants

Return type np.ndarray

class neuralxc.symmetrizer.symmetrizer.**MixedTraceSymmetrizer**(**args*, ***kwargs*)

_registry_name 'mixed_trace'

static _symmetrize_function(*c*, *n_l*, *n*, **args*)

Return trace of c_m c_m' with mixed radial channels of the tensors stored in c

Parameters

- **c** (*np.ndarray of floats/complex*) – Stores the tensor elements in the order (n,l,m)
- **n_l** (*int*) – number of angular momenta (not equal to maximum ang. momentum! example: if only s-orbitals n_l would be 1)
- **n** (*int*) – number of radial functions

Returns Casimir invariants

Return type np.ndarray

CHAPTER 8

Configuration Files

Users can fine-tune the behavior of NeuralXC through the use of configuration files. These come in two forms, one of which contains information about the projection basis set and instructions to the engine (the electronic structure code). The other file specifies the hyperparameters used in the ML model. Although file names are arbitrary we will stick to our convention and refer to the former as `config.json` and the latter as `hyper.json`.

8.1 config.json

A configuration file to be used together with **PySCF** could look like the following:

```
{
    "preprocessor":
    {
        "basis": {"name": "cc-pVDZ-JKFIT"},
        "projector": "gaussian",
        "grid": "analytical"
    },
    "engine":
    {
        "application": "pyscf",
        "xc": "PBE",
        "basis" : "def2-TZVP"
    },
    "n_workers" : 1,
    "symmetrizer_type": "trace"
}
```

`preprocessor` contains all information required to perform the density projection, whereas `engine` captures everything concerning the SCF calculations (which application/engine to use, the XC-functional etc.). `n_workers` defines the number of processes to be used to conduct SCF calculations and projections in parallel.

The projection basis inside `preprocessor` can be provided in several different ways.

GTO basis

The user can either provide the PySCF internal name of a basis set

- `"basis": {"name": "cc-pVDZ-JKFIT"}`

or the path to a file containing the basis set definition (NWChem format is used)

- `"basis": {"file": "./my_basis_set"}`

In addition, if the projection is done on a grid, the two additional parameters that control the basis set localization can be provided (see Sec. ref{sec:basis})

- `"basis": {"name": "cc-pVDZ-JKFIT", "sigma": 2.0, "gamma": 1.0}`

Polynomial basis

For polynomial basis sets the basis needs to be specified as

- `"basis": {"n": 3, "l": 4, "r_o": 2.0}`

if the same basis is to be used for every element.

If different basis sets are desired for each element, they can be specified as

```
"basis": {
    "O" : {"n": 3, "l": 4, "r_o": 2.0},
    "H" : {"n": 2, "l": 3, "r_o": 2.5},
}
```

An example of a configuration file to be used together with **SIESTA** could be:

```
{
    "preprocessor":
    {
        "basis": {"n": 3, "l": 4, "r_o": 2.0},
        "projector": "ortho",
        "grid": "euclidean"
    },
    "engine":
    {
        "application": "siesta",
        "xc": "PBE",
        "basis" : "DZP",
        "pseudoloc" : ".",
        "fdf_path" : "./my_input.fdf",
        "extension": "RHOXC"
    },
    "n_workers" : 1
}
```

Compared to PySCF, there are three notable differences in the `engine` section. `pseudoloc` specifies the location where pseudopotential files are stored. `fdf_path` is optional and can be used to point to a SIESTA input file (`.fdf`) which is used to augment the engine options set in config.json. The .fdf file should textbf{not} contain any system specific information such as atomic positions as these are automatically filled by NeuralXC. `extension` specifies the extension of the files storing the electron density and can be switched between `RHOXC`, `RHO`, and `DRHO` (see SIESTA cite{siesta} documentation for details).

8.2 hyper.json

An example for a hyperparameter configuration file is shown below. We have added explanations of each line preceded by # (The json file format does not support comments, therefore these lines need to be removed before using the

example).

```
{
 "hyperparameters": {
  # Remove features with Var < 1e-10
     "var_selector__threshold": 1e-10,
  # Number of hidden nodes (same for every layer)
     "estimator__n_nodes": 4,
  # Number of hidden layers (0: linear regression)
     "estimator__n_layers": [0, 3],
  # L2 regularization strength
     "estimator__b": [0.01, 0.001],
  # Learning rate
     "estimator__alpha": 0.001,
  # Maximum number of training steps
     "estimator__max_steps": 20001,
  # Relative size of validation set to be split of training
     "estimator__valid_size": 0,
  # Minibatch size (use entire dataset if 0)
     "estimator__batch_size": 0,
  # Activation Function
     "estimator__activation": "GeLU"
 },
 # Number of folds for cross-validation
    "cv": 2
}
```

Parameters can either provided as a single value, or through a list (indicated by square brackets). A hyperparameter optimization using k-fold cross validation (CF) can be performed over all values provided inside the list, using a number of folds specified in `cv`. If multiple lists are given, the hyperparameter search is performed over the outer product of all lists. In case hyperparameter optimization is disabled, the first entry of each list is used by default.

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

Index

Symbols

__init__() (*neuralxc.projector.projector.EuclideanProjector method*), 19
__init__() (*neuralxc.projector.projector.RadialProjector method*), 19
__init__() (*neuralxc.projector.pyscf.PySCFProjector method*), 20
__init__() (*neuralxc.symmetrizer.symmetrizer.BaseSymmetrizer method*), 21
_symmetrize_function() (*neuralxc.symmetrizer.symmetrizer.MixedTraceSymmetrizer static method*), 22
_symmetrize_function() (*neuralxc.symmetrizer.symmetrizer.TraceSymmetrizer static method*), 21

B

BaseProjector (*class in neuralxc.projector.projector*), 18
BaseSymmetrizer (*class in neuralxc.symmetrizer.symmetrizer*), 21

E

EuclideanProjector (*class in neuralxc.projector.projector*), 19

G

GaussianRadialProjector (*class in neuralxc.projector.gaussian*), 19
get_basis_rep() (*neuralxc.projector.projector.BaseProjector method*), 18
get_basis_rep() (*neuralxc.projector.pyscf.PySCFProjector method*), 20
get_symmetrized() (*neuralxc.symmetrizer.symmetrizer.BaseSymmetrizer method*), 21

M

MixedTraceSymmetrizer (*class in neuralxc.symmetrizer.symmetrizer*), 22

O

OrthoRadialProjector (*class in neuralxc.projector.polynomial*), 19

P

PySCFProjector (*class in neuralxc.projector.pyscf*), 20

R

RadialProjector (*class in neuralxc.projector.projector*), 19

T

TraceSymmetrizer (*class in neuralxc.symmetrizer.symmetrizer*), 21