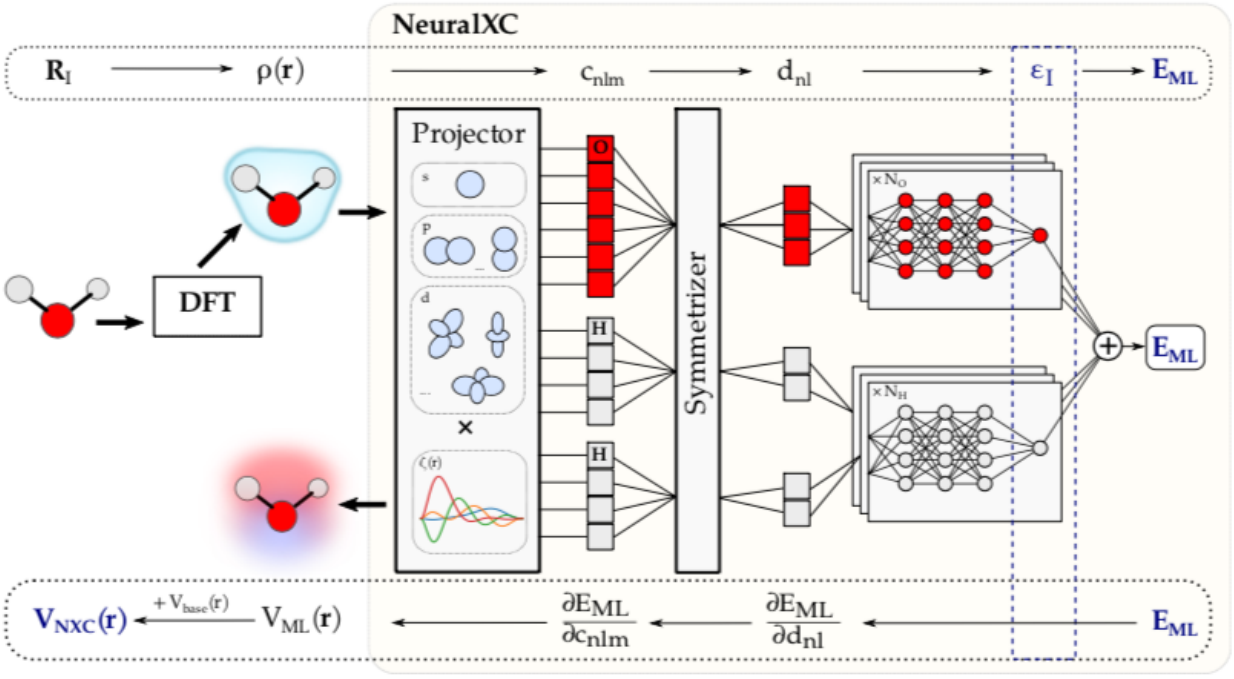
neuralxc Documentation

neuralxc

Mar 05, 2021

Contents:



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

To install NeuralXC using pip, 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 your installation, you can run unit tests with:

```
pytest -v
```

in the root directory.

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/.

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 basis.json hyperparameters.json
```

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

A minimal input file structure would look something like this:

basis.json

```
{
  "preprocessor":
  {
       "basis": "cc-pVTZ-jkfit",
       "extension": "chkpt",
       "application": "pyscf"
  },
  "n_workers" : 1,
  "engine_kwargs": {"xc": "PBE",
                    "basis" : "cc-pVTZ"}
```

(continues on next page)

(continued from previous page)

```
}
```

hyperparameters.json

```
{
    "hyperparameters": {
        "var_selector__threshold": 1e-10,
        "estimator__n_nodes": 16,
        "estimator__n_layers": 3,
        "estimator__b": 1e-5,
        "estimator__alpha": 0.001,
        "estimator__max_steps": 2001,
        "estimator__valid_size": 0.2,
        "estimator__batch_size": 32,
        "estimator__activation": "GELU",
    },
    "cv": 6
}
```

A detailed explanation of these files is given in *Input 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.

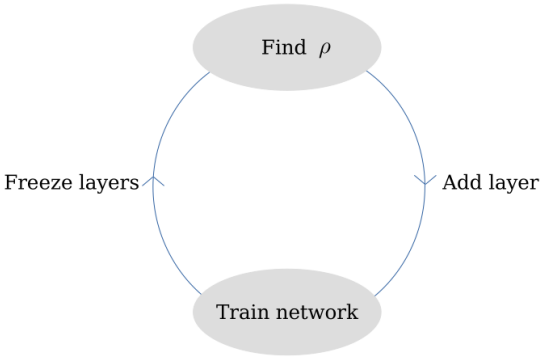
$$\min_{\omega} \mathbb{E}[(E_{\mathrm{ref}} - E_{\mathrm{ML}}[\rho_{base}|\omega])^2]$$

True loss:

$$\min_{\omega} \mathbb{E}[(E_{\mathrm{ref}} - \min_{\rho} E_{\mathrm{ML}}[\rho|\omega])^2]$$

How to optimize?

Consider ρ fixed during optimization of ω, then recompute ρ (SCF) → **iterative training**



CHAPTER 3

Projector

3.1 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

3.2 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.**OrthoProjector**(*unitcell*, *grid*, *basis_instructions*, ***kwargs*)

Implements orthonormal basis functions

_registry_name 'ortho'

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

_registry_name 'ortho_radial'

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

Implements GTO basis

_registry_name 'gaussian'

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

_registry_name 'gaussian_radial'

3.3 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 4

Symmetrizer

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

If

$$C_{nlm}$$

is the density projection with principal quantum number n, angular momentum l, and angular momentum projection m then trace symmetrizers create a rotationally invariant feature by taking the trace of the outer product over m of C with itself:

$$D_{nl} = \mathrm{Tr}_{mm'}[C \otimes_m C]$$

MixedTraceSymmetrizer generalizes this approach by mixing radial channels obtaining

$$D_{nn'l}$$

CHAPTER 5

Input Files

Most NeuralXC CLI commands require two types of input files:

- preprocessor file : Contains information regarding the density projection basis, as well

as the projector and symmetrizer type. Instructions on how to run SCF calculations such as the type of driver code as well as the basis sets etc. should also be contained.

- hyperparameters file : Contains the hyperparameters used the machine learning model (anything following symmetrization). Details are provided below.

5.1 Preproccesor

Let's start with an example:

pyscf with **analytical** gaussian projectors:

```
{
  "preprocessor":
  {
        "basis": "cc-pVTZ-jkfit",
        "extension": "chkpt",
        "application": "pyscf",
        "projector_type": "pyscf",
        "symmetrizer_type": "trace"

  },
  "n_workers" : 1,
  "engine_kwargs": {"xc": "PBE",
                    "basis" : "cc-pVTZ"}

}
```

"n_workers" determines over how many processes SCF calculations and subsequent density projections are distributed. Apart from this, there are two groups:

- "preprocessor" contains the necessary information to project the density and symmetrize it.
- "engine_kwargs" determines the behavior of the electronic structure code specified in preprocessor[application].

In our example we use PySCF and project the density analytically onto Gaussian type orbitals with the "pyscf" projector using the "cc-pVTZ-jkfit" basis . For a selection of different projectors, see *Projector*.

SIESTA with **numerical** polynomial projectors:

```
{
    "preprocessor": {
        "C": {
            "n": 4,
            "l": 5,
            "r_o": 2
        },
        "H": {
            "n": 4,
            "l": 5,
            "r_o": 2
        },
        "extension": "RHOXC",
        "applications": "siesta",
        "projector_type": "ortho",
        "symmetrizer_type": "trace"

    },
    "src_path": "workdir",
    "n_workers": 1,
    "engine_kwargs": {"pseudoloc" : ".",
                      "fdf_path" : null,
                      "xc": "PBE",
                      "basis" : "DZP",
                      "fdf_arguments": {"MaxSCFIterations": 50}
}
```

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Index

Symbols

B

E

G

M

O

P

R

T