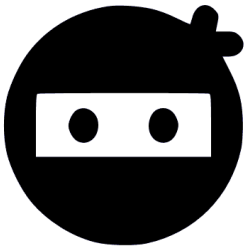
ML4Chem Documentation

Muammar El Khatib

Mar 19, 2020

GETTING STARTED:



ML4Chem

MACHINE LEARNING FOR CHEMISTRY AND MATERIALS

ML4Chem is a package to deploy machine learning for chemistry and materials science. It is written in Python 3, and intends to offer modern and rich features to perform machine learning (ML) workflows for chemical physics.

A list of features and ML algorithms are shown below.

- PyTorch backend.
- Completely modular. You can use any part of this package in your project.
- Free software <3. No secrets! Pull requests and additions are more than welcome!
- Documentation (work in progress).
- Explicit and idiomatic: `ml4chem.get_me_a_coffee()`.
- Distributed training in a data parallel paradigm aka mini-batches.
- Scalability and distributed computations are powered by Dask.
- Real-time tools to track status of your computations.
- Easy scaling up/down.
- Easy access to intermediate quantities: `NeuralNetwork.get_activations(X, numpy=True)` or `VAE.get_latent_space(X)`.
- Messagepack serialization.

CHAPTER

ONE

NOTES

This package is under heavy development and might break at some points until it gets stabilized. It is in its infancy, so if you find there is an error, you might want to report it so that it can be improved. We also welcome pull requests if you find any part of ML4Chem should be improved. That would be very nice.

CHAPTER

TWO

CITING

If you find this software useful, please use this bibtex to cite it:

```
@article{El_Khatib2020,
author = "Muammar El Khatib and Wibe de Jong",
title = "{ML4Chem: A Machine Learning Package for Chemistry and Materials Science}",
year = "2020",
month = "3",
url = "https://chemrxiv.org/articles/ML4Chem_A_Machine_Learning_Package_for_Chemistry_
↪and_Materials_Science/11952516",
doi = "10.26434/chemrxiv.11952516.v1"
}
```

CHAPTER

THREE

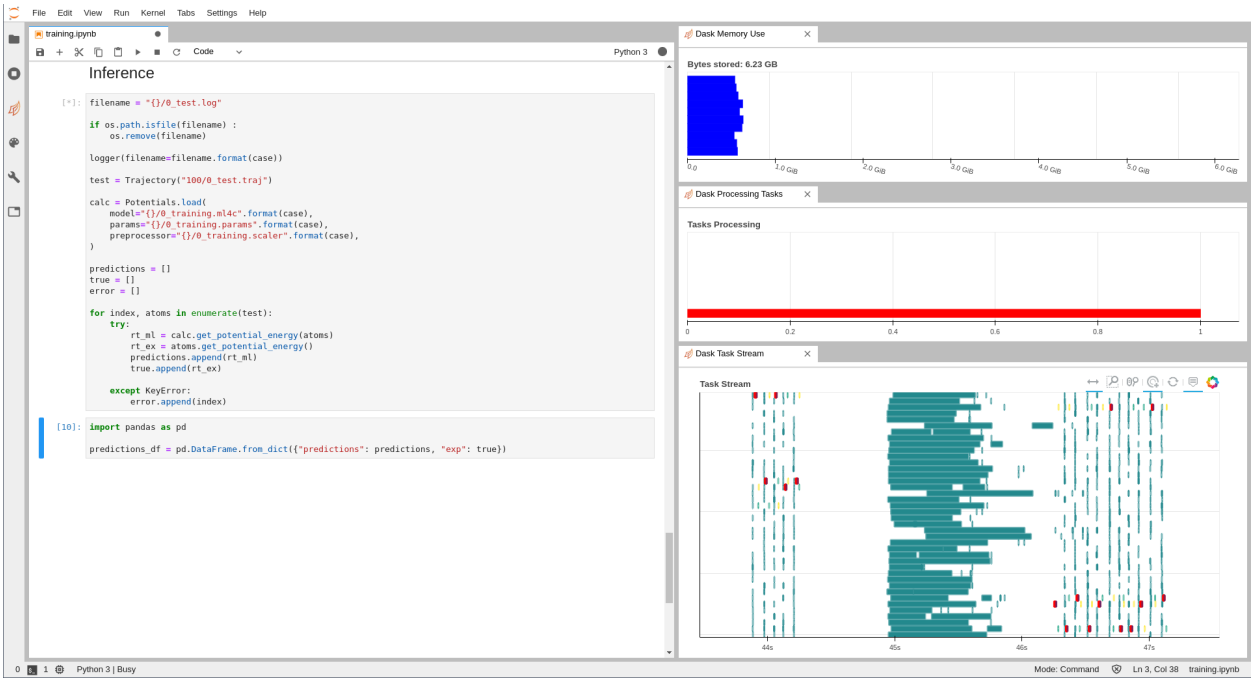
DOCUMENTATION

To get started, read the documentation at https://ml4chem.dev. It is arranged in a way that you can go through the theory as well as some code snippets to understand how to use this software. Additionally, you can dive through the module index to get more information about different classes and functions of ML4Chem. If you think the documentation has to be improved do not hesistate to state so in the bug reports and help out if you feel like it.

CHAPTER

FOUR

VISUALIZATIONS



CHAPTER

FIVE

COPYRIGHT

License: BSD 3-clause "New" or "Revised" License.

```
ML4Chem: Machine Learning for Chemistry and Materials (ML4Chem) Copyright (c)
2019, The Regents of the University of California, through Lawrence Berkeley
National Laboratory (subject to receipt of any required approvals from the U.S.
Dept. of Energy).  All rights reserved.

If you have questions about your rights to use or distribute this software,
please contact Berkeley Lab's Intellectual Property Office at
IPO@lbl.gov.

NOTICE.  This Software was developed under funding from the U.S. Department
of Energy and the U.S. Government consequently retains certain rights.  As
such, the U.S. Government has been granted for itself and others acting on
its behalf a paid-up, nonexclusive, irrevocable, worldwide license in the
Software to reproduce, distribute copies to the public, prepare derivative
works, and perform publicly and display publicly, and to permit other to do
so.
```

5.1 Install ML4Chem

You can install ML4Chem from `pip`, `conda` or `sources`.

5.1.1 Pip

You can install ML4Chem and all its dependencies with pip:

```
python3 -m pip install "ml4chem"
```

If you want to install the application for your user:

```
pyhon3 -m pip install --user ml4chem
```

If you want to install the development version of the *master* branch:

```
python3 -m pip install --upgrade git+https://github.com/muammar/ml4chem.git
```

5.1.2 Conda

For conda installation, execute:

```
conda install ml4chem
```

5.1.3 Sources

This type of installation is useful if you are going to develop new features for ML4chem.

1. Clone the application:

   ```
   git clone https://github.com/muammar/ml4chem
   ```

2. Install the requirements:

   ```
   cd ml4chem
   python3 -m pip install -r requirements.txt
   ```

3. After requirements are installed, you can proceed to add `ml4chem` to your `PYTHONPATH` and `PATH` (to use the `ml4chem` command line tool). Add the following to your `.bashrc` or `.zshrc`:

   ```
   PYTHOPATH=/path/to/ml4chem:$PYTHONPATH
   PATH=/path/to/ml4chem/bin:$PATH
   ```

5.2 License

You are under no obligation whatsoever to provide any bug fixes, patches, or upgrades to the features, functionality or performance of the source code ("Enhancements") to anyone; however, if you choose to make your Enhancements available either publicly, or directly to Lawrence Berkeley National Laboratory, without imposing a separate written license agreement for such Enhancements, then you hereby grant the following license: a non-exclusive, royalty-free perpetual license to install, use, modify, prepare derivative works, incorporate into other computer software, distribute, and sublicense such enhancements or derivative works thereof, in binary and source code form.

5.3 Building Documentation

Documentation is a very important part of any project, and in ML4Chem special attention is given to provide a clear documentation.

To locally build the docs you need to execute the *makedocs.sh* script:

```
sh makedocs.sh
```

This will automatically perform the following commands for you:

```
cd source
sphinx-apidoc -fo . ../../
cd ..
make html
```

When the Makefile is finished, you can check the documentation in the *build/html* folder.

5.4 Introduction

Data is central in Machine Learning and ML4Chem provides some tools to prepare your Datas. We support the following input formats:

1. Atomic Simulation Environment (ASE).

We will be adding support to other libraries, soon.

5.5 Data

The `ml4chem.data.handler` module allows users to adapt data to the right format to inter-operate with any other module of Ml4Chem.

Its usage is very simple:

```
from ml4chem.data.handler import Data
from ase.io import Trajectory

images = Trajectory("images.traj")
data_handler = Data(images, purpose="training")
traing_set, targets = data_handler.get_data(purpose="training")
```

In the example above, an ASE trajectory file is loaded into memory and passed as an argument to instantiate the `Data` class with `purpose="training"`. The `.get_images()` class method returns a hashed dictionary with the molecules in `images.traj` and the `targets` variable as a list of energies.

For more information please refer to `ml4chem.data.handler`.

5.6 Introduction

The *atomistic* module is designed to deploy machine learning (ML) models where the atom is the central object. ML potentials and force fields might be the best known cases of atom-centered models. These models are accurate for tasks such as the prediction of energy and atomic forces. They also are powerful because they can generalize to larger molecular as long as the local environments are sampled extensively to cover large domains.

5.7 Theory

The basic idea behind atomistic machine learning is to exploit the phenomenon of "locality" and predict molecular or periodic properties as the sum of local contributions:

$$P = \sum_{i=1}^{n} P_{atom}(R^{local})$$

where P_{atom} is a functional of atomic positions. In the case the properties are atomic by nature, e.g. atomic forces or atomic charges, there is no need to carry out the sum shown above as the output of the models are atomic.

5.8 Atomic Features

ML models require a set of measurable characteristics, properties or information related to the phenomenon that we want to learn. These are known as "features", and they play a very important role in any ML model. Features need to be relevant, unique, independent, and for our purposes be physics-constrained e.g. rotational and translational invariance.

ML4Chem supports by default Gaussian symmetry functions, and atomic latent features.

5.8.1 Gaussian Symmetry Functions

In 2007, Behler and Parrinello [Behler2007] introduced a fixed-length feature vector, referred also as "symmetry functions" (SF), to generalize the representation of high-dimensional potential energy surfaces with artificial neural networks and overcome the limitations related to the image-centered models. These SF are atomic feature vectors constructed purely from atomic positions, and their main objective is to define the *relevant chemical environment* of atoms in a material.

For building these features, we need to define a cutoff function (f_c) to delimit the *effective range* of interactions within the domain of a central atom.

$$f_c(r) = \begin{cases} 0.5(1 + cos(\pi \frac{r}{R_c})), & \text{if } r \leq R_c, \\ 0, & \text{if } r \geq R_c, \end{cases}$$

where R_c is the cutoff radius (in unit length), and r is the inter-atomic distance between atoms i and j. The cutoff function, having Cosine shape, vanishes for inter-atomic separations larger than R_c whereas it takes a finite value below the cutoff radius. Cutoff functions avoid abrupt changes on feature magnitudes near the boundary by smoothly damping them.

There are different types of SFs to consider when building Behler-Parrinello feature vectors: i) the radial (two-body term) and ii) angular (three-body terms) SFs. The radial SFs account for all interactions of a central atom i with its nearest neighbors atoms j. It is defined by,

$$\mathbf{G_i^2} = \sum_{j=1}^{N_{atom}} e^{-\eta(\mathbf{R_{ij}} - R_s)^2/R_c^2} f_c(R_{ij}),$$

where $\mathbf{R_{ij}}$ is the euclidean distance between central i and neighbor j atoms, R_s defines the center of the Gaussian, and η is related to its width. Each value in the sum is normalized by the square of the cutoff radius, R_c^2. In practice, one builds a high-dimensional feature vector by choosing different η values.

In addition to the radial SF, it is possible to include triplet many-body interactions within the cutoff radius R_c through the following angular SFs:

$$\mathbf{G_i^3} = 2^{1-\zeta} \sum_{j,k \neq i} (1 + \lambda cos\theta_{ijk})^{\zeta} e^{-\eta(\mathbf{R_{ij}}^2 + \mathbf{R_{ik}}^2 + \mathbf{R_{jk}}^2)/R_c^2} f_c(R_{ij}) f_c(R_{ik}) f_c(R_{jk}).$$

This part of the features is built from considering the Cosine between all possible θ_{ijk} angle of a central atom. There exists a variant of $\mathbf{G_i^3}$ that includes three-body interactions of atoms forming $180°$ inside the cutoff sphere but having an interatomic separation larger than Rc. These SFs account for long-range interactions [Behler2015]:

$$\mathbf{G_i^4} = 2^{1-\zeta} \sum_{j,k \neq i} (1 + \lambda cos\theta_{ijk})^{\zeta} e^{-\eta(\mathbf{R_{ij}}^2 + \mathbf{R_{ik}}^2)/R_c^2} f_c(R_{ij}) f_c(R_{ik}).$$

An atomic Behler-Parrinello feature vector will be composed by a subvector with radial SFs and another subvector of angular SFs. This represents an advantage when it comes to evaluate which type of SFs is more important when predicting energy and atomic forces.

```
from ml4chem.atomistic.features.gaussian import Gaussian

features = Gaussian(cutoff=6.5, normalized=True, save_preprocessor="features.scaler")
```

In the code snippet above we are building Gaussian type features using the `ml4chem.atomistic.features.gaussian.Gaussian` class. We use a `cutoff` radius of 6.5 angstrom, we normalized by the squared cutoff raidous, and the preprocessing is saved to the file `features.scaler` (by default the preprocessing used is `MinMaxScaler` in a range $(-1, 1)$ as implemented in `scikit-learn`). The `angular` symmetry functions used by default are G_i^3, if you are interested in using G_i^4, then you need to pass `angular_type` keyword argument:

```
features = Gaussian(cutoff=6.5, normalized=True,
                    save_preprocessor="features.scaler", angular_type="G4")
```

5.8.2 Atomic Enviroment Vector

The Gaussian feature vectors were modified by Smith et al [Smith2017] in the following way:

The radial symmetry functions are not normalized by the squared of the cutoff radius, and instead of several *eta* values they use a single value to produce thin Gaussian peaks, and different *R_s* parameters are used to probe outward from the atomic center.

$$\mathbf{G_i^2} = \sum_{j \neq j}^{N_{atom}} e^{-\eta(\mathbf{R_{ij}} - R_s)^2} f_c(R_{ij}),$$

The angular part adds an arbitrary number of shifts in the angular environment and an exponential factor that allows the angular environment to be considered within radial shells based on the average of the distance from the neighboring atoms.

$$\mathbf{G_i^4} = 2^{1-\zeta} \sum_{j,k \neq i} (1 + cos(\theta_{ijk} - \theta_s))^{\zeta} e^{-\eta[((\mathbf{R_{ij}} + \mathbf{R_{ik}}/2) - R_s)^2]} f_c(R_{ij}) f_c(R_{ik}).$$

5.8.3 Atomic Latent Features

Atomic latent features are those extracted using unsupervised learning.

5.9 Models

5.9.1 Neural Networks

Neural Network (NN) are models inspired on how the human brain works. They consist of a set of hidden-layers with some nodes (neurons). The most simple NN architecture is the *fully-connected* case in which each neuron is inter-connected to every other neuron in the previous/next layer, and each connection has its own weight. When an activation function is applied to the output of a neuron, the NN is able to learn non-linearity aspects from the data.

In ML4Chem, a neural network can be instantiated as shown below:

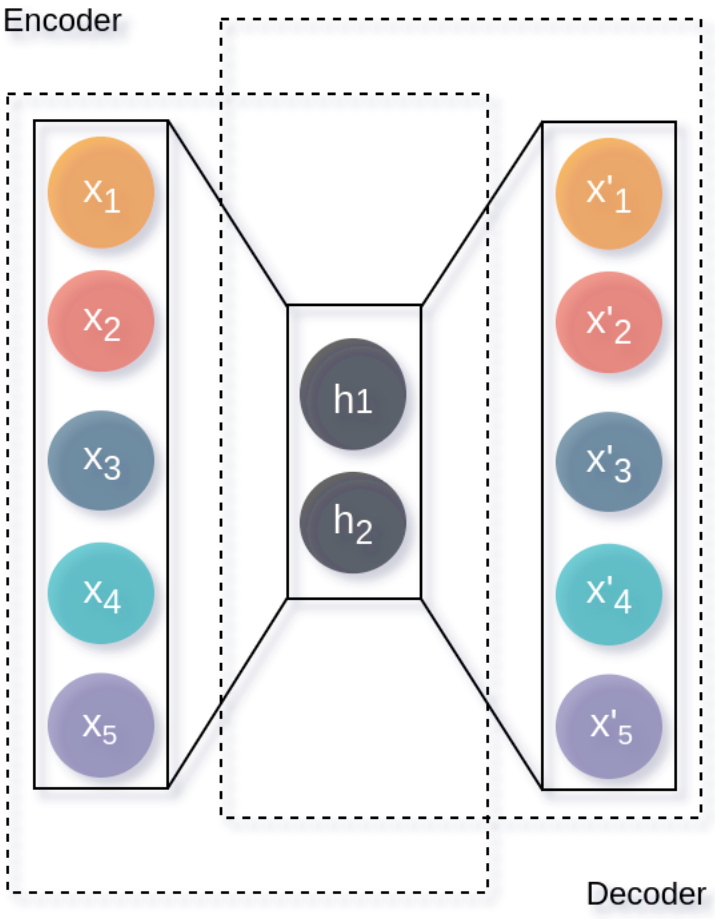
```
from ml4chem.atomistic.models.neuralnetwork import NeuralNetwork

n = 10
activation = "relu"
nn = NeuralNetwork(hiddenlayers=(n, n), activation=activation)
nn.prepare_model()
```

Here, we are building a NN with the `ml4chem.atomistic.models.neuralnetwork.NeuralNetwork` class with two hidden-layers composed 10 neurons each, and a ReLu activation function.

5.9.2 Autoencoders

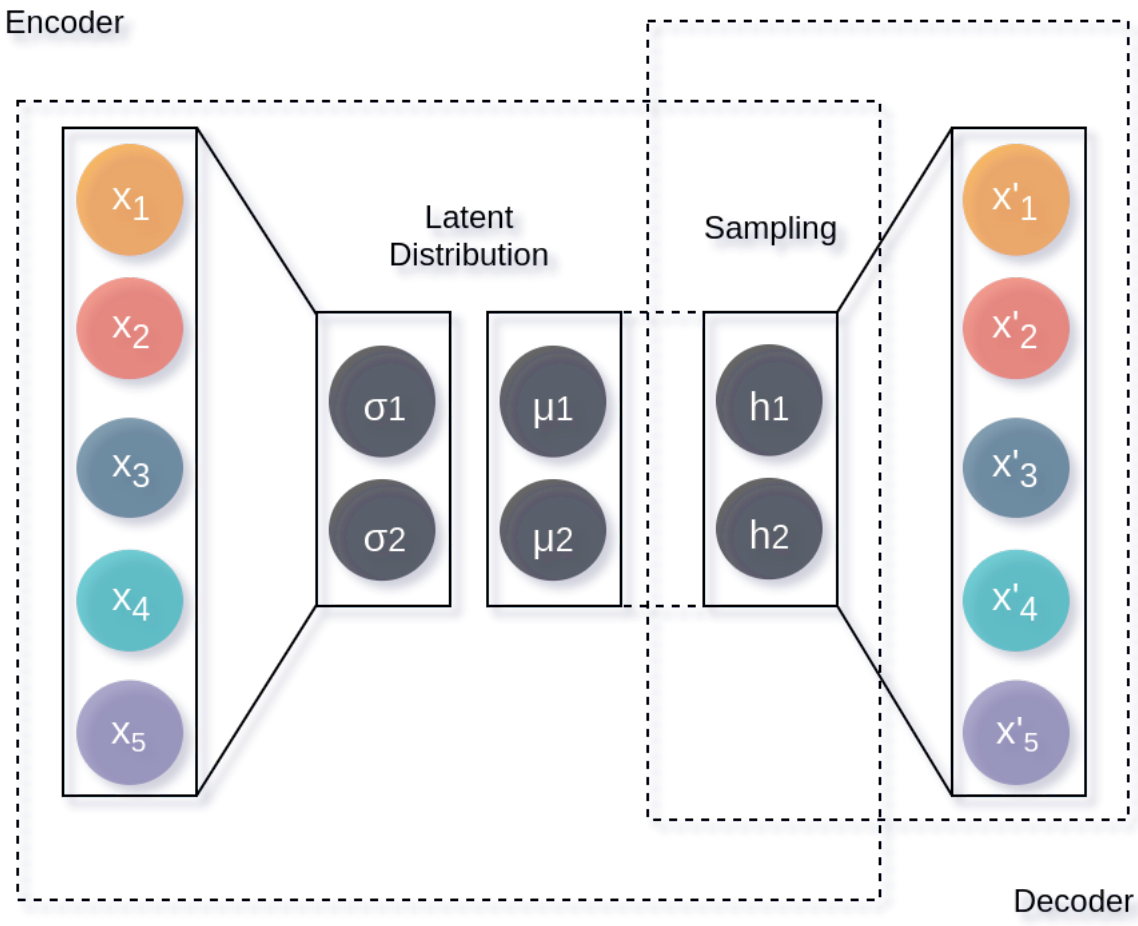
Autoencoders (AE) are NN architectures that able to extract features from data in an unsupervised learning manner. AE learns how to encode information because of a hidden-layer that serves as an informational bottleneck as shown in the figure below. In addition, this latent code is used by the decoder to reconstruct the input data.




```
from ml4chem.atomistic.models.autoencoders import AutoEncoder

hiddenlayers = {"encoder": (20, 10, 4), "decoder": (4, 10, 20)}
activation = "tanh"
autoencoder = AutoEncoder(hiddenlayers=hiddenlayers, activation=activation)
data_handler.get_unique_element_symbols(images, purpose=purpose)
autoencoder.prepare_model(input_dimension, output_dimension, data=data_handler)
```

ML4Chem also provides access to variational autoencoders (VAE) [Kingma2013]. These architectures differ from an AE in that the encoder codes a distribution with mean and variance (two vectors with the desired latent space dimension) instead of a single latent vector. Subsequently, this distribution is sampled and used by the decoder to reconstruct the input. This creates a generative model because now we will generate a latent distribution that allows a continuous change from one class to another.



To use this architecture, it just suffices to change the snippet shown above for an AE as follows:

```
from ml4chem.atomistic.models.autoencoders import VAE

hiddenlayers = {"encoder": (20, 10, 4), "decoder": (4, 10, 20)}
activation = "tanh"
vae = VAE(hiddenlayers=hiddenlayers, activation=activation, variant="multivariate")
data_handler.get_unique_element_symbols(images, purpose=purpose)
vae.prepare_model(input_dimension, output_dimension, data=data_handler)
```

5.9.3 Kernel Ridge Regression

Kernel Ridge Regression (KRR) is a type of support vector machine model that combines Ridge Regression with the kernel trick. In ML4Chem, this method is implemeted as described by Rupp in Ref. [Rupp2015]. Below there is a description of this implementation:

1. Molecules are featurized.
2. A kernel function $k(x, y)$ is applied to all possible pairs of atoms in the training data to build a covariance matrix, $\mathbf{K}$.
3. $\mathbf{K}$ is decomposed in upper- and lower- triangular matrices using Cholesky decomposition.
4. Finally, forward- and backward substitution is carried out with desired targets.

5.9.4 Gaussian Process Regression

Gaussian Process Regression (GP) is similar to KRR with the addition of the uncertainty of each prediction.

References:

5.10 Introduction

ML4Chem uses Dask which is a flexible library for parallel computing in Python. Dask allows easy scaling up and down without too much effort.

In this part of the documentation, we will cover how ML4Chem can be run on a laptop or workstation and how we can scale up to running on HPC clusters. Dask has a modern and interesting structure:

1. A scheduler is in charge of registering tasks.
2. Tasks can be registered in a delayed way (registered but not computed) or simply submitted as futures (submitted and computed).
3. When the scheduler receives a task, it sends it to workers that carry out the computations and keep them in memory.
4. Results from computations can be subsequently used for more calculations or just brought back to memory.

5.11 Scale Down

Running computations with ML4Chem on a personal workstation or laptop is very easy thanks to Dask. The `LocalCluster` class uses local resources to carry out computations. This is useful when prototyping and building your pipeline withouth wasting time waiting for HPC resources in a crowded cluster facility.

ML4Chem can run with `LocalCluster` objects, for which the scripts have to contain the following:

```
from dask.distributed import Client, LocalCluster

cluster = LocalCluster(n_workers=8, threads_per_worker=2)
client = Client(cluster)
```

In the snippet above, we imported `Client` that will connect to the scheduler created by the `LocalCluster` class. The scheduler will have 8 workers with 2 threads. As tasks are required, they are sent by the `Client` to the `LocalCluster` for being computed and kept in memory.

A typical script for running training in ML4Chem looks as follows:

```
from ase.io import Trajectory
from dask.distributed import Client, LocalCluster
from ml4chem.atomistic import Potentials
from ml4chem.atomistic.features import Gaussian
from ml4chem.atomistic.models.neuralnetwork import NeuralNetwork
from ml4chem.utils import logger


def train():
    # Load the images with ASE
    images = Trajectory("cu_training.traj")
```

(continues on next page)

(continued from previous page)

```
    # Arguments for fingerprinting the images
    normalized = True

    # Arguments for building the model
    n = 10
    activation = "relu"

    # Arguments for training the potential
    convergence = {"energy": 5e-3}
    epochs = 100
    lr = 1.0e-2
    weight_decay = 0.0
    regularization = 0.0

    calc = Potentials(
        features=Gaussian(
            cutoff=6.5, normalized=normalized, save_preprocessor="model.scaler"
        ),
        model=NeuralNetwork(hiddenlayers=(n, n), activation=activation),
        label="cu_training",
    )

    optimizer = ("adam", {"lr": lr, "weight_decay": weight_decay})
    calc.train(
        training_set=images,
        epochs=epochs,
        regularization=regularization,
        convergence=convergence,
        optimizer=optimizer,
    )


if __name__ == "__main__":
    logger(filename="cu_training.log")
    cluster = LocalCluster()
    client = Client(cluster)
    train()
```

5.12 Scale Up

Once you have finished with prototyping and feel ready to scale up, the snippet above can be trivially expanded to work with high performance computing (HPC) systems. Dask offers a module called `dask_jobqueue` that enables sending computations to HPC systems with Batch systems such as SLURM, LSF, PBS and others (for more information see https://jobqueue.dask.org/en/latest/index.html.

To scale up in ML4Chem with Dask, you only have to slightly change the snipped above as follows:

```
if __name__ == "__main__":
    from dask_jobqueue import SLURMCluster
    logger(filename="cu_training.log")


    cluster = SLURMCluster(
        cores=24,
```

(continues on next page)

(continued from previous page)

```
        processes=24,
        memory="100GB",
        walltime="24:00:00",
        queue="dirac1",
    )
    print(cluster)
    print(cluster.job_script())
    cluster.scale(jobs=4)
    client = Client(cluster)
    train()
```

We removed the `LocalCluster` and instead used the `SLURMCluster` class to submit our computations to a SLURM batch system. As it can be seen, the `cluster` is now a `SLURMCluster` requesting a job with 24 cores and 24 processes, 100GB of RAM, a wall time of 1 day, and the queue in this case is *dirac1*. Then, we scaled this up by requesting to the HPC cluster 4 jobs with these requirements for a total of 96 processes. This `cluster` is passed to the `client` and the training is effectively scaled up.

5.13 Visualization

We also offer a `ml4chem.visualization` module to plot interesting graphics about your model, features, or even monitor the progress of the loss function and error minimization.

Two backends are supported to plot in ML4Chem: Seaborn and Plotly.

An example is shown below:

```
from ml4chem.visualization import plot_atomic_features
fig = plot_atomic_features("latent_space.db",
                           method="pca",
                           dimensions=3,
                           backend="plotly")
fig.write_html("latent_example.html")
```

This will produce an interactive plot with plotly where dimensionality was reduced using PCA, and an html with the name *latent_example.html* is created.

To activate plotly in Jupyter or JupyterLab follow the instructions shown in https://plot.ly/python/getting-started/#jupyter-notebook-support

If plotly is not rendering correctly you need to install the jupyter extension:

```
jupyter labextension install @jupyterlab/plotly-extension
```

5.14 Indices and tables

- genindex
- modindex
- search

BIBLIOGRAPHY

[Behler2007] Behler, J. & Parrinello, M. Generalized Neural-Network Representation of High-Dimensional Potential-Energy Surfaces. Phys. Rev. Lett. 98, 146401 (2007).

[Behler2015] Behler, J. Constructing high-dimensional neural network potentials: A tutorial review. Int. J. Quantum Chem. 115, 1032–1050 (2015).

[Smith2017] 1. Smith, J. S., Isayev, O. & Roitberg, A. E. ANI-1: an extensible neural network potential with DFT accuracy at force field computational cost. Chem. Sci. 8, 3192–3203 (2017).

[Kingma2013] Kingma, D. P. & Welling, M. Auto-Encoding Variational Bayes. arXiv Prepr. arXiv1312.6114 (2013).

[Rupp2015] Rupp, M. Machine learning for quantum mechanics in a nutshell. Int. J. Quantum Chem. 115, 1058–1073 (2015).