ML4Chem: A Machine Learning Package for Chemistry and Materials Science

Muammar El Khatib* and Wibe A de Jong

Computational Research Division, Lawrence Berkeley National Laboratory, Berkeley, CA 94720, USA

(Dated: March 6, 2020)

ML4Chem is an open-source machine learning library for chemistry and materials science. It provides an extendable platform to develop and deploy machine learning models and pipelines and is targeted to the non-expert and expert users. ML4Chem follows user-experience design and offers the needed tools to go from data preparation to inference. Here we introduce its `atomistic` module for the implementation, deployment, and reproducibility of atom-centered models. This module is composed of six core building blocks: data, featurization, models, model optimization, inference, and visualization. We present their functionality and ease of use with demonstrations utilizing neural networks and kernel ridge regression algorithms.

I. INTRODUCTION

In the last decade, machine learning (ML) has undergone fast development due to large amounts of available data and advancements in computational hardware *e.g.* faster and cheaper central processing units (CPU), graphics process units (GPU), and more recently the introduction of tensor processing units (TPU). Algorithmic improvements on how to compute the gradient in weight space of feedforward neural networks with respect to a loss function[1] reduced the computational time of training deep neural networks significantly. As a result companies like Google, and Facebook, introduced the most useful deep learning platforms available right now: TensorFlow[2], and Pytorch[3]. These frameworks positively impacted and advanced ML research because they helped with democratizing and simplified access to ML technologies to a larger audience.

In the field of physical chemistry and materials sciences, ML models are being standardized and applied to solve tasks such as the acceleration of atomistic simulations[4–8], prediction of the electronic Hamiltonian with generative models[9, 10], extraction of continuous latent representations for the generation of molecules[11], and even the prediction of the scent of small organic molecules[12]. It also is becoming the norm to release software solutions as support to validate results of publications that apply ML models, and alleviate the "reproducibility crisis in artificial intelligence and machine learning"[13, 14]. Nevertheless, this obliquely fragments the software ecosystem because each software implementation *a)* requires specific data structures and *b)* would likely experience a lack of continuous support. There already are packages that democratize ML in chemistry. For example, DeepChem[15] has played a critical role in providing users a helpful platform of ML algorithms and featurizers for drug discovery, quantum chemistry, material sciences, and biology. More recently ChemML has been introduced as a machine learning and informatics program suite for the analysis, mining, and modeling of chemical and materials data[16]. What differentiates ML4Chem is that it focuses on easing the implementation of new functionality, extraction of intermediate quantities, interfacing with

* melkhatibr@lbl.gov

external programs, and exportation of any of its modules' outputs. Also, ML4Chem is in its infancy bringing up the possibility to shape its future directions based on current users' needs and ML paradigms.

Here we introduce the `atomistic` module where ML algorithms learn underlying relationships between molecules and properties treating atoms as central objects. They exploit the principle of locality in Physics: *a global quantity is defined as a sum that runs over many localized contributions.* These localized contributions usually account for interactions of an atom and its nearest-neighbor atoms (many-body interactions). Atomistic models are very useful and have been successfully applied for the acceleration of molecular dynamics simulations[17–19], identification of phase transitions in materials[20], determination of energy and atomic forces with high accuracy[21, 22], the search of saddle-points[23] and the prediction of atomic charges[24, 25].

This publication is organized as follows: in section II, we will discuss the design and architecture of ML4Chem's `atomistic` module. Each of its core blocks is introduced in Section III and we will demonstrate the code's capabilities through a series of demonstration examples in Section IV. Finally, conclusions and perspectives are drawn.

II. ATOMISTIC MODULE: DESIGN AND ARCHITECTURE

ML4Chem and its modules are written in Python in an object-oriented programming paradigm and are built on top of popular open-source projects to avoid duplication of efforts. In this regard, all deep learning computations are implemented with Pytorch[3]. Mathematical and linear algebra operations are executed by Numpy[26] or Scipy[27, 28] that are widely used and recognized for this purpose. Parallelism is achieved with a flexible library for parallel computing called Dask[29]. Dask enables computational scaling-up from a laptop to High-Performance Computing (HPC) clusters effortlessly and offers a web dashboard to real-time monitoring. This is particularly valuable because it provides a good estimation to users about the status of calculations, and helps at profiling computations. Good documentation is another important aspect, as the lack of it can harm usability. ML4Chem's source code is documented using Numpy Python docstrings and rendered in HTML and PDF format. Also, we provide diverse information ranging from installation, theory, usage of modules, and examples.

ML4Chem's modules are developed following user-experience (UX) design practices to deliver usability, accessibility, and desirability. For example, in ML4Chem the names of modules, classes, and functions tend to be idiomatic and easy to remember semantically. Getting the latent space from an autoencoder is performed by calling `autoencoder.get_latent_space(X)`, or the computation of atomistic features is done with a `.calculate()` class method that is provided in all featurizers under the `atomistic.features` module *e.g.* `features.calculate(X, purpose="inference")`. All modules are designed to have the same structure, enabling users to become familiar and gain intuition on their usage quickly. The library can be used in interactive Python environments such as iPython, Jupyter or JupyterLab notebooks. Or if desired, as scripts that are invoked by the Python interpreter.

Figure 1 shows a schematic representation of the machine learning workflow that drives the design philosophy used to develop the `atomistic` module in ML4Chem:

1. Atoms positions are mapped into atomistic feature vectors with the `atomistic.features` module and chemical symbols are used as labels.

2. ML models are instantiated utilizing atomistic feature vectors as input, and depending on the nature of the task

(supervised or unsupervised learning), targets might or might not be known.

3. Model's parameters are trained either by minimizing/maximizing a loss function or solving systems of equations. In the former case, ML4Chem provides `loss` and `optim` modules with sets of predefined loss functions and optimizers to train supervised and unsupervised atomistic models.

4. The resulting atomistic model outputs and predictions are expected to be scalar or vector quantities.

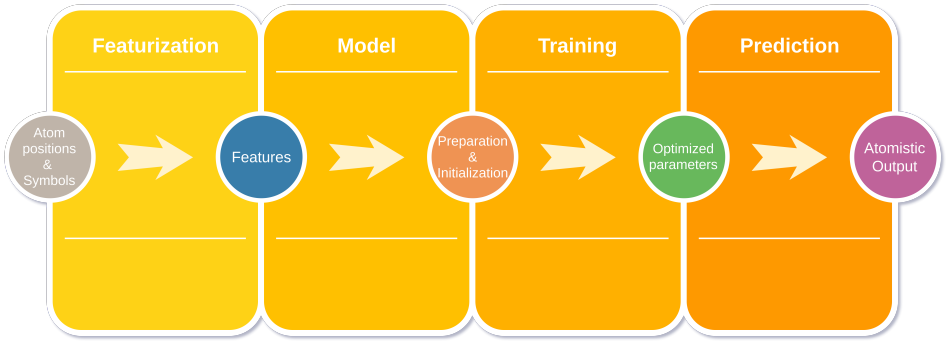


FIG. 1: Design approach of the `atomistic` module of ML4Chem.

As shown in Figure 2, the architecture of the `atomistic` module is composed of 6 building blocks. They correspond to the methods and tools required to deploy atom-centered simulations from input featurization to inference and visualization, according to our design philosophy. Modules inside the code blocks need to comply with being *derived classes* from *base classes* using Python mixins. This guarantees new classes are reusing the code base and inheriting already defined structures implicitly from the *base classes* to operate seamlessly with other `ML4Chem` components. This practice is encouraged by providing a `base.py` file within each module level that contains required base classes, and is enforced in the continuous integration (CI) system.

In the following sections, each of these blocks is discussed with particular attention on what can be achieved with them, their implementation, and code snippets on their usage. When relevant, we will discuss the theory and mathematics behind them as well.

III. CORE MODULES

A. Data

ML is focused on finding underlying patterns and relationships based on data examples. The format of data, a central part of ML, varies depending on the sources, and the ML algorithm adopted for solving a particular task. ML4Chem provides a `Data` class that creates an object with a data structure that facilitates interoperability with any of its available `atomistic` modules.

Our `atomistic` ML algorithms require molecules in the form of `Atoms` objects as implemented in the Atomic Simulation Environment (ASE)[30]. One of the reasons behind choosing ASE is its stability and that it supports more than 50 file formats in its `ase.io.read` module *e.g.* XYZ, NWChem, GPAW, and Gaussian. For instance, an XYZ file can

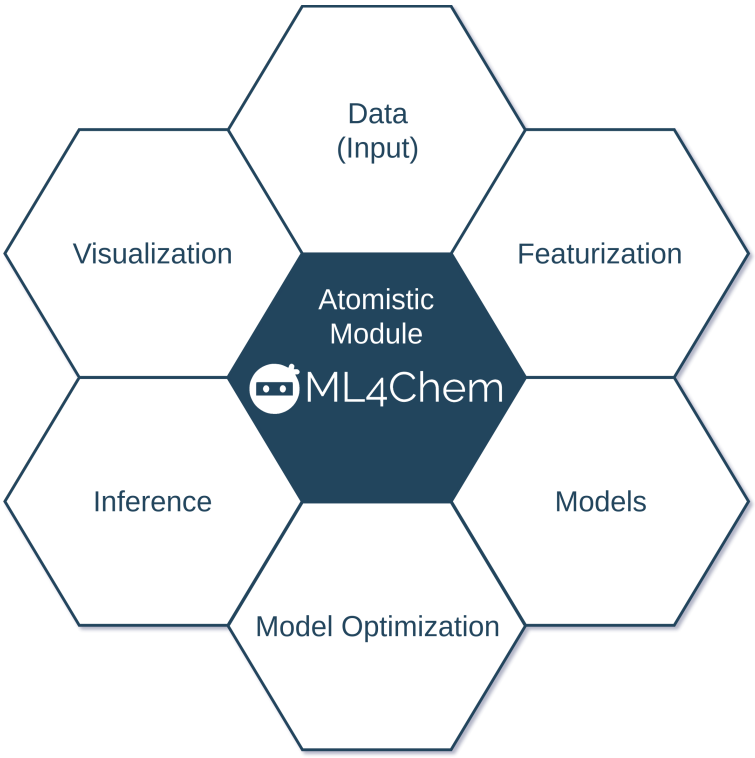


FIG. 2: Core modules of the `atomistic` module of ML4Chem.

be parsed and converted to an `Atoms` object in the following way: `molecule = ase.io.read("file.xyz")`. `Atoms` objects hold molecular information such as Cartesian coordinates, atom types, chemical symbols, molecular charge, cell type, energy, and atomic forces. Molecules in ASE format can be stored to disk as a list of molecules (`Atoms`) using ASE's `Trajectory` module. Besides the formats supported by the `ase.io.read` module, we also can parse the Chemical JSON (CJSON) format[31] and the ANI-1 data set[32] using the `data.parser` module. Support of input formats such as those available in pymatgen[33] and MolSSI's QCSchema are planned for future releases.

The `Data` class uses a list of molecules, or in other words *a list `Atoms` objects*, to generate a unique *sha1* hash to label each molecule and store their respective pairs input/targets. Targets refer to the expected output of ML models and in `atomistic` simulations they may correspond to total energy, atomic forces, dipole moments, etc. Duplicated data points are automatically removed during the hashing procedure to avoid poor performance and numerical instability. In Listing 1, the `Data` class is instantiated by passing an ASE trajectory file with name `dataset.traj` containing some `molecules` for the `purpose` of `"training"` an atomistic ML algorithm. After hashing the `molecules` present in the trajectory file, pairs of input/targets examples are yielded by invoking the `.get_data()` class method and assigned to the `training_set` and `targets` variables.

Finally, once the data is loaded in memory and arranged by the `Data` class, it can be mapped into features by the `atomistic.features` module of ML4Chem. The content of this object, an ordered dictionary, can be easily exported as a pandas dataframe using the `.to_pandas()` class method and saved or serialized for subsequent use.

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

molecules = Trajectory("dataset.traj")
purpose = "training"

data_handler = Data(molecules, purpose=purpose)
training_set, targets = data_handler.get_data(purpose=purpose)
df = data_handler.to_pandas()
```

Listing 1: Example of Data class usage in ML4Chem.

B. Featurization

ML features are defined as a set of measurable unique characteristics or properties of an observable. They are fundamental to any ML algorithm because they represent what models "see". Feature engineering is the process of applying domain knowledge to generate sets of numerical features that make ML algorithms work and learn meaningful representations from data. In physics constrained domains, like atomistic ML models, feature extraction is also elusive because features require to fulfill a series of properties that are commonly expected from physical systems such as rotational and translational invariance (equivariance). Featurization is a challenging time-consuming process and the feature engineering cycle encompasses their creation, selection according to importance, and validation for the task of interest.

Features can be classified in *i)* human-engineered features where assumptions about data are made by humans to assign properties to observables or *ii)* machine-engineered features where the ML algorithms discover meaningful representation during the training procedure. The `atomistic.features` module of ML4Chem supports the former case with Gaussian type features, and the latter with a `LatentFeatures` class (introduced in the following subsections).

To avoid duplication of efforts, atomistic features such as Coulomb matrix[34], smooth overlap of atomic positions (SOAP)[35], and many-body tensor representation (MBTR)[36] are supported through DScribe which is a software package for ML that provides popular feature transformations ("descriptors") for atomistic materials simulations. This accelerates the application of ML for atomistic property prediction by providing a user-friendly, off-the-shelf descriptor implementations[37]. Our own implementation of the Coulomb matrix feature vectors is available in the `atomistic.features` module of ML4Chem, and serves as an example of how easily our package can be extended.

1. Gaussian Features

In 2007, Behler and Parrinello[7] introduced Gaussian feature vectors, also referred to as "symmetry functions" (SF), for the representation of high-dimensional potential energy surfaces with artificial neural networks. These features overcome limitations related to the *image-centered* models and are built for each atom in a molecule or extended system. They *fingerprint* the **relevant chemical environment** of atoms in molecules, and their computation only requires chemical symbols and atomic positions.

To delimit the *effective range* of interactions within the domain of a central atom, a *cutoff function* (f_c) is introduced:

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

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, with Cosine shape as shown in Eq. 1, vanishes for inter-atomic separations larger than R_c and takes finite values below the cutoff radius. These cutoff functions aim to avoid abrupt changes in the magnitudes of the features near the boundary by smoothly damping them.

There are two sets of interactions to consider when building Gaussian features: *i)* the radial (two-body term) and *ii)* angular (three-body terms) SFs. The radial SFs account for all possible interactions between a central atom i and its nearest neighbors atoms j. It is defined by Eq. 2:

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

where $\mathbf{R_{ij}}$ is the Euclidean distance between central atom i and neighbor atom j, R_s defines the center of the Gaussian, and η is related to its width. Each pairwise contribution to the feature in the sum is normalized by the square of the cutoff radius R_c^2 as proposed in Ref. [38]. In practice, one builds a high-dimensional feature vector by choosing different η values.

In addition to the radial SFs (two-body term), it is possible to include triplet many-body interactions within the cutoff radius R_c using the following equation:

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

This part of the feature vector is built from considering the Cosine between all possible θ_{ijk} angles of a central atom i and a pair of neighbors j, and k. There exists a variant of $\mathbf{G_i^3}$ that includes three-body interactions of triplets forming 180° inside the cutoff sphere but having an inter-atomic separation larger than Rc. These SFs account for long-range interactions as described by Behler in Ref. [39]:

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

In ML4Chem, Gaussian features can be built with the `atomistic.features` module as shown in Listing 2. The `Gaussian()` class is instantiated with the desired cutoff radius (units are Å) to define the neighbor atoms, the type of angular symmetry functions (either $\mathbf{G_i^3}$ or $\mathbf{G_i^4}$), and we normalized dividing them by R_c^2. It also is possible to pass the `svm` keyword argument to calculate features for SVM algorithms. Note that we need to pass the `data_handler` and `training_set` objects created by the `Data` class (see Listing 1). It is important to preprocess, scale and normalize features. In this way, models learn meaningful and noiseless underlying representations. ML4Chem uses scikit-learn[40] for the preprocessing of atomistic features. This can be activated by passing the `preprocessor` keyword argument. Currently, the supported preprocessors in ML4Chem for atomistic features are: `MinMaxScaler`, `StandardScaler`, and `Normalizer`. Finally, the `.calculate()` method calculates features.

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

features = Gaussian(
    cutoff=6.5,
    normalized=True,
    preprocessor="MaxMinScaler",
    save_preprocessor="features.scaler",
    angular_type="G3",
)

X = features.calculate(
    training_set, purpose="training", data=data_handler, svm=False
)
```

Listing 2: Computing Gaussian features with the `atomistic.features` module of ML4Chem.

2. Latent Features

In deep learning, latent features are non-directly observed variables inferred by causality[41]. These features fall under the machine-engineered classification and are determined by the ML algorithm itself during training *without human intervention*. A clear example would correspond to the informational bottleneck inferred when training autoencoders (AE, see Section III C 2). This informational bottleneck encodes hidden information by making a dimensionality reduction that can reconstruct the input space (directly observed variables). In physics constrained ML models, latent spaces might correspond to chemical physics aspects of atoms in molecules depending on how the model's parameters are penalized and optimized. Their advantage over human-engineered features is that they facilitate the flexibility of the models, and when extracted with posterior inference they generalize well. Nevertheless, latent features tend to be difficult to interpret and posterior inference relies on Bayesian variational methods that are challenging to train[42]. The `atomistic.features` module in ML4Chem provides a class to ease latent feature extraction to train any of the available atomistic ML algorithms. It uses AE algorithms, like the ones described in Sections III C 2 and III C 3, and convert raw features into latent variables by forward-propagating them through the encoder part of the AE architecture. Listing 3 shows an example of the `LatentFeatures` featurization module where two keyword arguments are passed to instantiate this class:

- A tuple called `args` contains the name of the type of raw features and a dictionary with their respective parameters to be computed and subsequently converted into latent variables with a trained AE.
- We also assign to a variable `encoder` a dictionary with keys `"model"`, and `"params"` containing the paths on disk to load the model and its parameters.

After instantiation, latent variables are computed with the `.calculate()` class method as it was done with the `Gaussian` class, and are returned in the right structure to be used as input to train other ML algorithms. Note that they can also be converted to a Pandas data frame.

```python
from ml4chem.atomistic.features import LatentFeatures

ae_path = "my_autoencoder/"

# Arguments to build raw features
normalized = True
preprocessor = ("MinMaxScaler", {"feature_range": (-1, 1)})
args = (
    "Gaussian",
    {
        "preprocessor": preprocessor,
        "cutoff": 6.5,
        "normalized": normalized,
        "save_preprocessor": "iso.scaler",
        "overwrite": False,
    },
)
# Dictionary to load trained autoencoder
encoder = {"model": ae_path + "vae.ml4c",
           "params": ae_path + "vae.params"}

features = LatentFeatures(encoder=encoder, features=args)
latent = features.calculate(
    inputs,
    purpose="training",
    data=data_handler,
    svm=False
)
df = features.to_pandas()
```

Listing 3: Extraction of latent features using the `atomistic.features` module of ML4Chem.

C. Models

This section describes atomistic ML regression algorithms in ML4Chem under the `atomistic.models` module. At this point, it is important to differentiate ML algorithms from ML models. ML algorithms refer to all procedures and steps carried out to solve a determined ML task while models are well-defined results of algorithms. Another important difference is that ML models are fed inputs to infer or predict some output. Atomistic ML algorithms exploit the physical phenomenon of "locality" where atoms are fundamental entities and whose ML features can be extracted by measuring interactions between each atom and its nearest-neighbor atoms. Predictions P are therefore calculated as the sum of many individual contributions as shown in Eq. 5:

$$P = \sum_{i=1}^{n} p_i(\mathbf{F}_i(\mathbf{R}_i)), \tag{5}$$

where a local contribution p_i is a functional of a feature mapping function $\mathbf{F}_i$ that takes as arguments atom positions $\mathbf{R}_i$, and chemical symbols. There are two possible flavors of these algorithms: *i)* a sub regression model for each chemical symbol in the data set exists or *ii)* a unique regression model is used for all chemical element types.

To implement new deep learning atomistic algorithms in ML4Chem, developers have to derive their classes by inheriting the structure from the `DeepLearningModel` base class shipped in the `base.py` file and shown in Listing 4:

```
from abc import ABC, abstractmethod
import torch


class DeepLearningModel(ABC, torch.nn.Module):
    @abstractmethod
    def name(cls):
        """Return name of the class"""
        return cls.NAME

    @abstractmethod
    def __init__(self, **kwargs):
        """Arguments needed to instantiate the model"""
        pass

    @abstractmethod
    def prepare_model(self, **kwargs):
        """Prepare model for training or inference"""
        pass

    @abstractmethod
    def forward(self, X):
        """Forward propagation pass"""
        pass
```

Listing 4: Abstract base class for the implementation of new atomistic deep learning models under `atomistic.models` module of ML4Chem.

Deep learning classes require a `name()` method that returns the name of the model, list of keyword arguments to instantiate the model using the reserved `__init__()` constructor, a `prepare_model()` method where parameters are initialized and the algorithm is prepared for the purposes of training or inference. Finally, a `forward()` method is responsible to perform a forward pass and return predictions. Support vector machine algorithms, require all these methods except for the `forward()` function. The fulfillment of this structure enables inter-operability within ML4Chem.

1. Supervised Learning

Supervised learning refers to the ML task of determining a complex function that maps inputs into outputs from labeled pairs of input/target examples. Its application assumes there exists a good understanding of the structure and existent classes of the data. It is worth noting that the interpretation of these models is usually easier than the

ones obtained from unsupervised learning ML tasks.

In atomistic ML algorithms, the inputs correspond to attributes of molecules such as atom positions, atom types, total charge, band-gap, etc. All these can be directly used as features, or mapped applying domain knowledge rules. To train the models, targets can be any scalar or vector quantity associated with a molecule like total energy, dipole moment, or atomic forces. Supervised algorithms tend to require large amounts of data, and featurization tends to be biased because it is human-engineered. Thus, models are usually prone to perform poorly beyond training set regimes. Also, the data sets have to be designed with enough diversity to capture meaningful underlying structures from input data. Active learning protocols[43, 44] are known to help in assuring this diversity. After the training procedure, and assessment of the predictive power of models by cross-validation[45] or other model validation techniques, their parameters can be stored to perform inference in unknown data.

In the following sections, we discuss the type of neural network architectures and support vector machine algorithms supported in the `atomistic.models` module of ML4Chem.

a. Neural Networks

Neural Networks (NN) are algorithms inspired by how the human brain works. Their building blocks are constituted by hidden layers with interconnected neurons. NN can approximate any function with arbitrary accuracy and their outputs can be represented by the following equation:

$$\mathbf{y} = \mathbf{W}\mathbf{X} + \mathbf{b}, \tag{6}$$

where $\mathbf{X}$ are the inputs, $\mathbf{W}$ are learnable parameters, $\mathbf{b}$ are the biases and $\mathbf{y}$ are outputs. Eq. 6 is nothing but a linear function and without any extra modification, it can only fit data having a linear response. In the context of deep learning, the product $\mathbf{WX}$ is a nested function[46] composed of l hidden layers that returns either a vector or a scalar:

$$\mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{b} = \mathbf{f}_4(\mathbf{f}_3(\mathbf{f}_2(\mathbf{f}_1(\mathbf{x})))). \tag{7}$$

From the equation above, $l = 4$ means the model has four layers that output a vector. The nested function $\mathbf{f}_l(\mathbf{z})$ is defined as:

$$\mathbf{f}_l(\mathbf{z}) = \mathbf{a}_l(\mathbf{W}_l\mathbf{z} + \mathbf{b}_l), \tag{8}$$

where we have introduced an *activation function* denoted by $\mathbf{a}_l$. The effect of the activation function on the outputs $\mathbf{z}$ of the neurons is to add non-linear response making the NN suitable to approximate more complex non linear functions. Some of the most common activation functions are $tanh(z)$, $sigmoid(z)$, and $relu(z)$:

$$tanh(z) = \frac{(e^z - e^{-z})}{(e^z + e^{-z})} \tag{9}$$

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{10}$$

$$relu(z) = \begin{cases} 0 & \text{for } z \leq 0 \\ z & \text{for } z \geq 0 \end{cases} \tag{11}$$

When forward-propagating information through NN, the type of activation function applied to the last layer determines whether the task is regression (linear or non-linear activation function), or classification (logistic activation function)[1]. When more than two hidden layers are stacked between the input and output layers, NN are called *deep neural networks*.

In Figure 3 we show a neural network with one input layer, two hidden layers, and an output layer. In this multi-layer perceptron, we assume all outputs of a layer are connected to the inputs of succeeding layers.

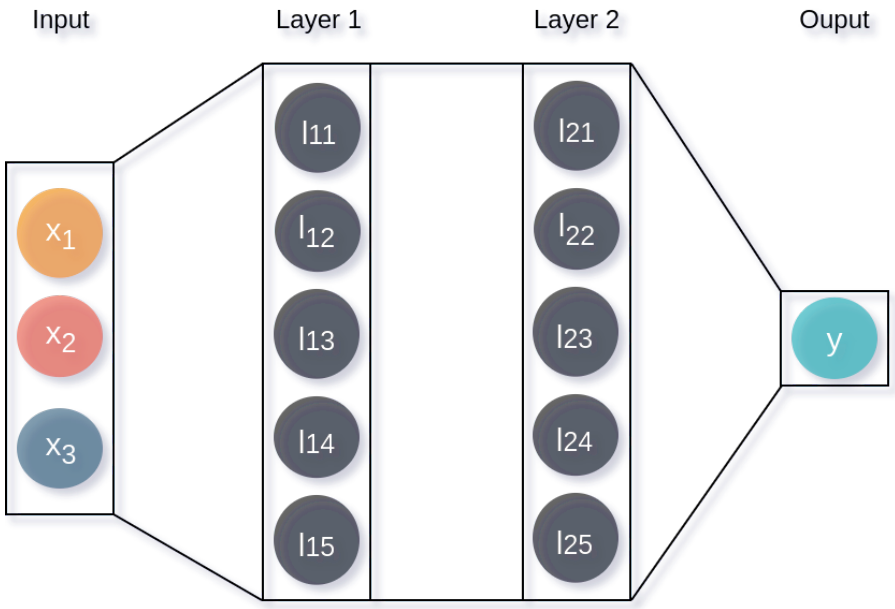


FIG. 3: Schematic representation of a neural network.

Our atomistic neural network implementation follows the structure described by Behler and Parrinello[7] where each set of chemical elements in the data set has its own NN algorithm. For instance, if the data set contains C, H, N, and O atoms, then there will be four different NN. Hidden layers are *fully connected* and the same activation function is applied to each of them. Different activation functions to each hidden layer and application of convolutions are features planned for future releases. However, various NN with different activation functions can be merged and trained simultaneously with the `ModelMerger` module described in Section III C 3. NN algorithms are trained by forward-propagating the atomistic feature vectors of molecules through their respective NN. The outputs of the models are atomic contributions. If targets are global quantities, such as the total energy, then the local atomic energy contributions of a molecule can be summed up and used to evaluate the loss function against corresponding targets. In this way, when backward-propagating the models, the optimization process will account for the global quantity. The usage of NN in the `atomistic.models` module is illustrated in Listing 5. A `NeuralNetwork` class with two hidden layers of 10 nodes each is instantiated, and a ReLU activation function is applied to all hidden layers except for the output

layer. The model is subsequently prepared for training with atomistic feature vectors holding 4 dimensions per atom. Note that we also feed the `data_handler` object created with the `Data` class.

```
from ml4chem.atomistic.models import NeuralNetwork

# Parameters
n = 10                  # Hidden layers
activation = "relu"     # Activation function
input_dimension = 4     # Input Dimension

# Model instantiation and preparation
nn = NeuralNetwork(hiddenlayers=(n, n), activation=activation)
nn.prepare_model(
    input_dimension,
    data=data_handler,
    purpose="training"
)
```

Listing 5: Usage example script of the atomistic neural network class in ML4Chem.

b. Support Vector Machines

A Support Vector Machine (SVM) is a discriminative classifier that for some given labeled examples, it outputs an optimal hyperplane which categorizes new examples. For atomistic ML models, Kernel Ridge Regression (KRR) is a very useful algorithm based on linear ridge regression that aims to minimize a squared error loss function with a $l2$-norm regularization term:

$$\underset{\beta \in \mathbb{R}^D}{\operatorname{argmin}} \sum_{i=1}^{N} (\beta \mathbf{x_i} - y_i)^2 + \lambda ||\beta||_2^2, \tag{12}$$

where β are regression coefficients, $\mathbf{x_i}$ is the i^{th} model input, y_i is its corresponding i^{th} target, and $\lambda \geq 0$ is a hyperparameter representing the penalty during the minimization of the loss function. Although the loss function in Eq. 12 is only suitable for linear regression, it can be applied to non-linear problems using the *kernel trick* (KT)[47]. The KT allows model inputs (atom features in this framework) to be mapped into other feature spaces through the use of kernel functions (KF). The importance of KF lies in that they are computed directly in raw input space (atomistic feature vectors are not explicitly modified). KF are non-negative real-valued functions between two vectors $k(\mathbf{x}, \mathbf{y})$. According to *Mercer's theorem*, matrices built from KFs are squared definite-positive covariant kernel matrices[48]. Thus, they provide some numerical advantages because they can be factorized and generally provide well-defined solutions. The KRR algorithm of the atomistic module of ML4Chem supports radial basis kernel functions (RBF), squared exponential kernel, linear, and Laplacian kernels in both their isotropic and anisotropic[49] variants as seen on Eqs. 13 and 14 respectively:

$$k(\mathbf{x}, \mathbf{y})_{rbf}^{iso} = \exp\left(-\frac{||\mathbf{x} - \mathbf{y}||_2^2}{2\sigma^2}\right), \tag{13}$$

$$k(\mathbf{x},\mathbf{y})^{aniso}_{rbf} = \exp\left(-\sum_{i=1}^{D}\frac{(x_i-y_i)}{2\sigma_i^2}\right), \tag{14}$$

where $\mathbf{x}$ and $\mathbf{y}$ are input and reference atom feature vectors respectively, σ is the Gaussian width, and D is the number of dimensions of the feature vectors. These kernels are bivariate functions that compute the similarity based on distances between two vectors in a normalized vector space. The output of these KF is within the interval $[0, 1]$. Note that anisotropic variants of kernels allow the assignment of specific variances for each dimension of the feature vectors without the need to modifying them explicitly.

Our implementation is inspired by Ref. [50] and uses the atomistic feature vectors to build the kernel matrix shown in Eqs. 15. Kernel matrices are positive definite which make their Cholesky decomposition to provide a unique solution to the regression coefficients. This is a particular advantage of these models since their solution is deterministic. By deterministic we mean a model that for a given input always reproduces the same regression coefficients, and consequently the same output. That is not the case in models such as neural networks that are randomly initialized leading to different local minima that are valid solutions.

$$\begin{bmatrix} k^E_{11} & k^E_{12} & k^E_{13} & \dots & k^E_{1N} \\ k^E_{21} & k^E_{22} & k^E_{23} & \dots & k^E_{2N} \\ \dots & \dots & \dots & \dots & \dots \\ k^E_{N1} & k^E_{N2} & k^E_{N3} & \dots & k^E_{NN} \end{bmatrix} \begin{bmatrix} \beta_1 \\ \beta_2 \\ \vdots \\ \beta_N \end{bmatrix} = \begin{bmatrix} E_1 \\ E_2 \\ \vdots \\ E_N \end{bmatrix} \tag{15}$$

Solving the system of equations in Eq. 15 requires atomic quantities that might not be available in quantum mechanics such as atomic energies. In those cases, one has to rely on atomic decomposition Ansatz as described by Bartók[35, 51, 52]. Instead of training using a loss function, the regression coefficients β in Eq. 15 are determined by forward and backward substitution after matrix factorization using the Cholesky decomposition method.

After training, a prediction is carried out by

$$f(\mathbf{x}) = \sum_{i=1}^{n}\beta_i k(\mathbf{x_i}, \mathbf{y_j}). \tag{16}$$

The `atomistic` module of ML4Chem can perform kernel ridge regression as shown in Listing 6. The `KernelRidge` class is instantiated with the kernel function to be used, the isotropic variance `sigma` and penalization values. Then, the model is prepared by passing the feature and reference features to build the covariance matrix. We also support Gaussian process regression in which case the listing has to be modified to import the `GaussianProcess` class instead. When doing so, the predictions will also return a tuple with the associated predictive uncertainty.

```
from ml4chem.atomistic.models import KernelRidge

# Parameters
sigma = 1.0             # Sigma kernel value
kernel = "rbf"          # Kernel type
lamda = 1e-5            # Penalization

# Model instantiation and preparation
krr = KernelRidge(sigma=sigma, kernel=kernel, lamda=lamda)
krr.prepare_model(
    feature_space,
    reference_features,
    data=data_handler
)
```

Listing 6: Usage example script of the atomistic Kernel ridge regression class in ML4Chem.

2. Unsupervised Learning

Probably the best example of an unsupervised learning algorithm corresponds to autoencoders (AE). AE are a type of artificial neural network architecture composed by an encoder and a decoder that can learn data representations without human intervention. As shown in Figure 4, an AE forward propagates the input through the encoder architecture to reach an informational bottleneck where latent features, denoted with h_d, are extracted. The bottleneck layer is usually of lower dimensionality compared to the input. Afterward, these latent features are used by the decoder to reconstruct the inputs. This reconstruction task might seem uninteresting but depending on the type of AE architecture, and what is being reconstructed, the model can learn low-dimensional representations of the input space, denoise data, or even how to generate new examples.

An autoencoder with fully connected hidden layers tends to only "memorize" how to reconstruct the input space or denoise data. Their power can be enhanced by penalizing the latent space with a loss function[53], attachment of external task[11], or just by forcing sparse activations[54, 55].

Our implementation of autoencoders supports two architectures that can be set with the `one_for_all` boolean keyword argument. If set to true, a single autoencoder is used for all types of atoms in the training data, otherwise, each set of atom type will have its autoencoder as in the Behler-Parrinelo scheme. In Listing 7, an `AutoEncoder()` class in the `atomistic.models` module of ML4Chem is instantiated with an encoder/decoder architecture where input's dimensions are lowered from 40 to 4 dimensions. In this case, the hyperbolic tangent activation function is applied to all layers with the `activation` keyword argument, and when the `purpose` is not set then training is assumed. Currently, only the same activation function for all layers is supported but this functionality can be extended in future releases. Note that the `.prepare_model()` class method sets the dimensionality of the output to be the same as the input.

In 2013 Kingma and Welling [56] proposed a modification to autoencoder architectures employing a variational Bayes method that infers a posterior probability. Instead of encoding a latent vector, as seen in Figure 5, the variational autoencoder (VAE) encodes a non-differentiable latent normal distribution with unit variance. To make the model trainable, a latent vector is randomly sampled from the distribution and used by the decoder to reconstruct the input

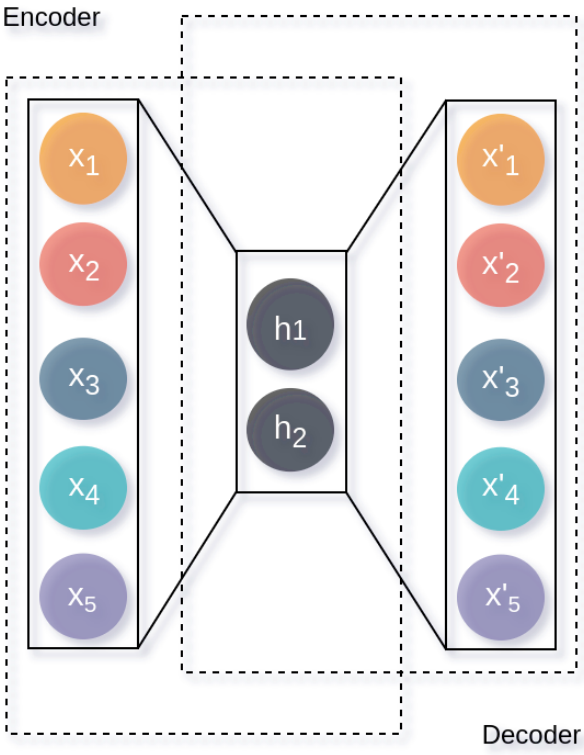


FIG. 4: Architecture of an Autoencoder.

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


hiddenlayers = {"encoder": (20, 10, 4), "decoder": (4, 10, 20)}
activation = "tanh"
purpose = "training"
input_dimension = 40

ae = AutoEncoder(
    hiddenlayers=hiddenlayers,
    activation=activation,
    one_for_all=False
)
ae.prepare_model(input_dimension, input_dimension, data=data_handler)
```

Listing 7: Training example script of the atomistic autoencoeder class in ML4Chem.

space. This is known as the *reparameterization trick* and allows the calculation of the gradient of the loss function with respect to the parameters of this architecture[56]. VAEs are known to produce blurry reconstructions[57, 58] of the inputs, but more meaningful data representations, unlike vanilla AEs. That is because the VAEs are forced to minimize reconstruction errors from feature vectors sampled from the latent distribution. VAEs are classified as *generative models* because they learn smooth conditional distributions of the input space $\mathbf{X}$ for given evidence and are

even able to generate new examples. VAEs can be invoked in ML4Chem by importing `VAE()` instead of the `AutoEncoder()` class in Listing 7.

In our VAE implementation, we also support the `one_for_all` keyword argument, and provide three variants when passing the `variant` string keyword argument:

1. "multivariate": the decoder outputs a distribution with mean and unit variance, and the model is trained by minimizing the negative of the log-likelihood plus the Kullback–Leibler divergence[59]. This is useful when outputs are continuous variables. Expected feature range $[-\infty, \infty]$.

2. "bernoulli": the sigmoid activation function is applied to the decoder's outputs and models are also minimized with the negative of the log-likelihood plus the Kullback–Leibler divergence[59]. Features must be in a range $[0, 1]$.

3. "dcgan": hyperbolic tangent activation function is applied to the decoder's outputs, and the model is trained by minimizing the negative of the log-likelihood plus the Kullback–Leibler divergence[59]. Useful for feature ranges $[-1, 1]$.

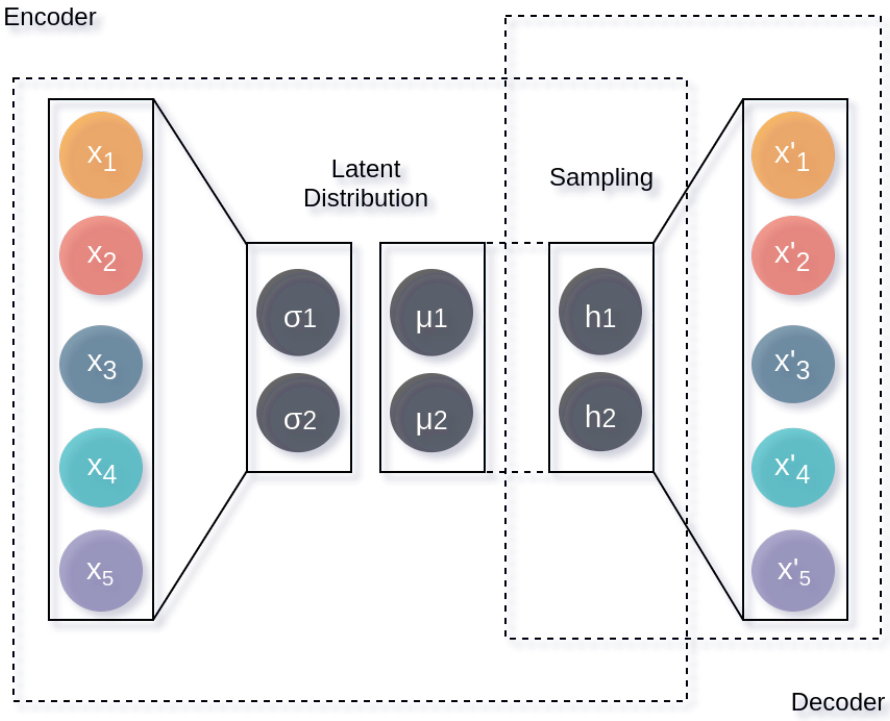


FIG. 5: Variational Autoencoder architecture.

3. Semi-Supervised and Hybrid Learning

In previous sections, we have discussed the supervised and unsupervised learning algorithms. Supervised learning requires large amounts of data to be labeled in input/target pairs which usually requires a very costly process carried

out by an ML engineer or a data scientist. On the other hand, unsupervised learning does not require labeled data but its applicability is very limited and dependent on the problem that is tried to be solved. To overcome these limitations, semi-supervised learning tasks consist of building algorithms that can work with both labeled and unlabeled data. Mixing both tasks is challenging as several loss functions are used to then backward propagate the ensemble model.

The atomistic.models module allows access to semi-supervised learning with the `ModelMerger()` class. In Listing 8, we use the neural network and autoencoder models already instantiated in Listings 5 and 7 and add them to a list called `models`. We import the required loss functions from the `models.loss` module and assigned them to a list named `losses`. Similarly, the targets for each model are added to a list with name `targets`, and for the inputs, note that the neural network model takes as input the latent space from the autoencoder.

During training, there are two important options that can be passed to the `ModelMerger` class:

- A boolean `"independent_loss"` keyword argument to set whether or not the loss functions are merged. If set to true, models are aware of each other.
- A `"loss_weights"` keyword argument with a list to set how much the loss of model(i) contributes to the total loss function.

More about training procedures is elaborated in Section III D.

```
from ml4chem.atomistic.models.loss import MSELoss, AtomicMSELoss

models = [ae, nn]
losses = [MSELoss, AtomicMSELoss]

# AE input/output dimensions are 20/4.
# NN input/output dimensions are 4/1.
inputs = [X, ae.get_latent_space]

# AE targets are X.
# NN targets are vector of scalars y.
targets = [X, y]

merged = ModelMerger(models)
merged.train(
        inputs=inputs,
        targets=targets,
        data=data_handler,
        lossfxn=losses
)
```

Listing 8: Example of a hybrid model in ML4Chem combining an autoencoder with a neural network.

D. Model Optimization

Deep learning is a challenging optimization problem that employs the gradient descent algorithm. Partial derivatives of the loss function with respect to the model's parameter space are required to update ML model's parameters in the direction of steepest descent as defined by the negative of the gradient. This computation experiences the vanishing or exploding gradient problems. In the former case, the partial derivatives of the loss function with respect to the parameters of the model become so small that the optimizers cannot adjust weights to minimize or maximize the loss function. In the latter case, partial derivatives are so large that optimizers oscillate around a minimum or maximum in the loss function space and convergence is never reached. These problems remained elusive for decades but were resolved by gradient clipping[60], regularization[46], and usage of activation functions that only saturate in one direction *e.g.* the ReLU activation function[61]. The `atomistic` module of ML4Chem supports most of the optimizers available in Pytorch such as ADAM[62], stochastic gradient descend[63], and Adagrad[64].

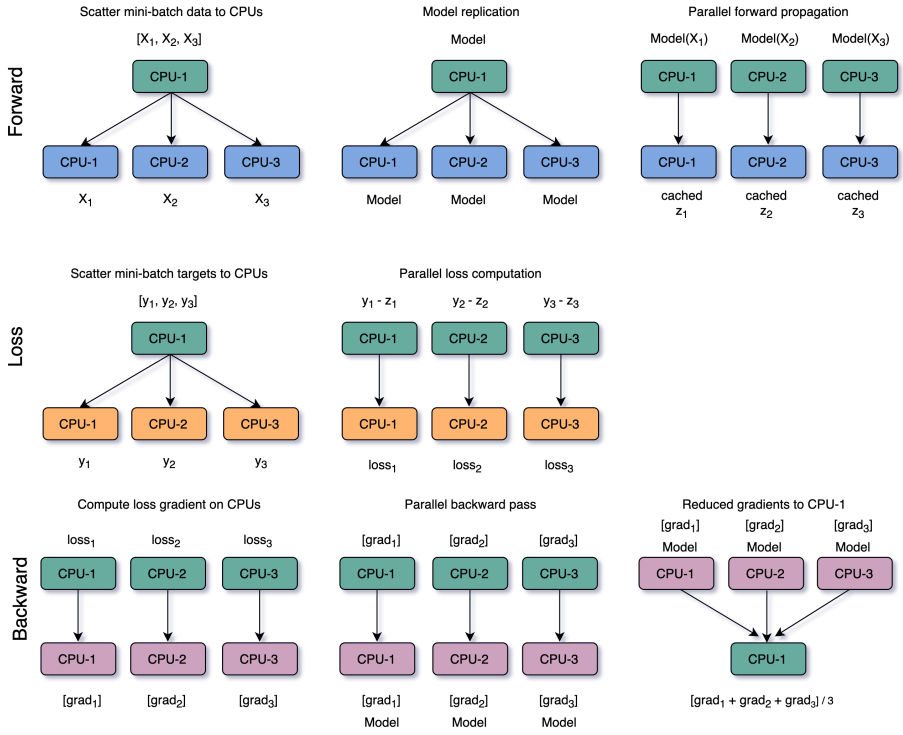


FIG. 6: Mini-batch data parallel scheme.

We train models exploiting the data parallelism paradigm as illustrated in Figure 6. In this scheme, data is partitioned in mini-batches that are set with the `batch_size` keyword argument. Mini-batches with the input data and targets are scattered through workers in a local or remote distributed cluster. The ML algorithm also is replicated on each of these workers, and parallel forward propagation is carried out. After the forward pass, we proceed with the parallel evaluation of the loss function for each mini-batch. In this step, the Pytorch automatic differentiation package `autograd` computes the gradient of the loss functions with respect to the weights of the model for each mini-batch and performs parallel backward-propagation passes. Finally, the gradients are reduced in CPU-1 and we call a step in the optimizer to update weights according to this gradient. This process is repeated until the number of epochs is

exhausted or some convergence criterion is reached. The parallelism scheme described above can also be visualized with the Dask dashboard as shown in Figure 7.

Each algorithm in the `atomistic.models` module provides a `train` class to optimize its parameters. Each model has its particularities but, in general, the `train` class requires at least input, outputs, batch size, optimizer, and loss function (if needed). The `train` class is responsible for taking all this information and carry out all necessary steps to train a model including calling steps in the optimizer. The optimizers are set with the `atomistic.models.optim` module that wraps Pytorch optimizer objects from a tuple composed by the name of the optimizer, and a dictionary with its required parameters *e.g.*: `optimizer = ("adam", {"lr": float, "weight_decay"=float})`. When ML models are trained, their parameters can be stored to disk and subsequently, be used for prediction.

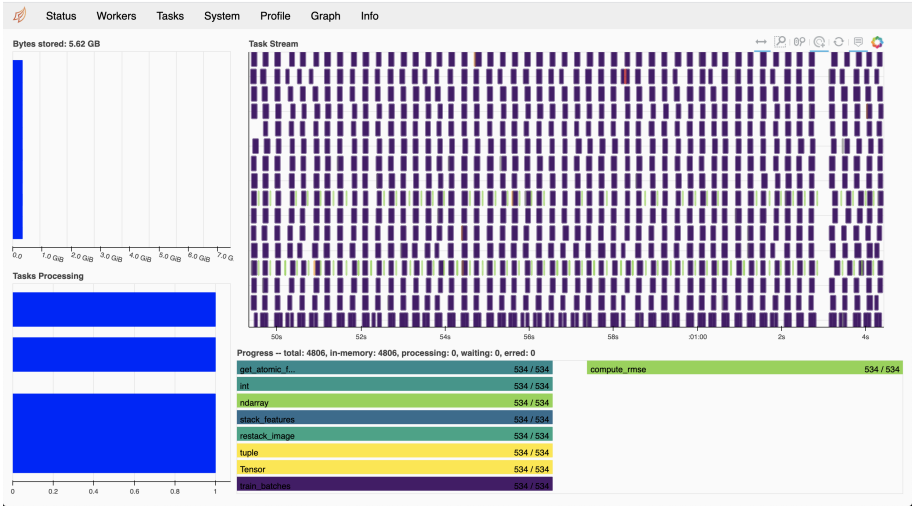


FIG. 7: Dask dashboard tool tracking computations in real time for the case where ML4Chem is optimizing a neural network over 16 processes.

E. Inference and Visualization

Inference is the step in an ML pipeline where a trained ML model is used to infer/predict over unseen samples. This procedure involves a similar forward pass as the carried out during training for the prediction of targets.

Probably the biggest difference with a model in training mode is that during inference no backward-propagation is carried out. Therefore, the inference is a very fast computation that requires matrix multiplication and summation. In ML4Chem, atomistic models can be saved and loaded as seen in Listing 9.

We use the `.save()` method of the `Atomistic` class to save a neural network and gaussian features passing as arguments their objects. The `label` argument is used to save them to disk with name `"nn"`. A trajectory file is loaded in memory that contains molecules stored as `Atoms()` ASE objects. Then, we assign the model to the `calc` variable with the `load()` function and the following arguments: *i)* path to optimized parameters stored in a file named `nn.ml4c`, *ii)* path to the `nn.params` file that contains parameters in JSON format to recreate the model and features, and *iii)* the preprocessor to scale the features from `nn.scaler`.

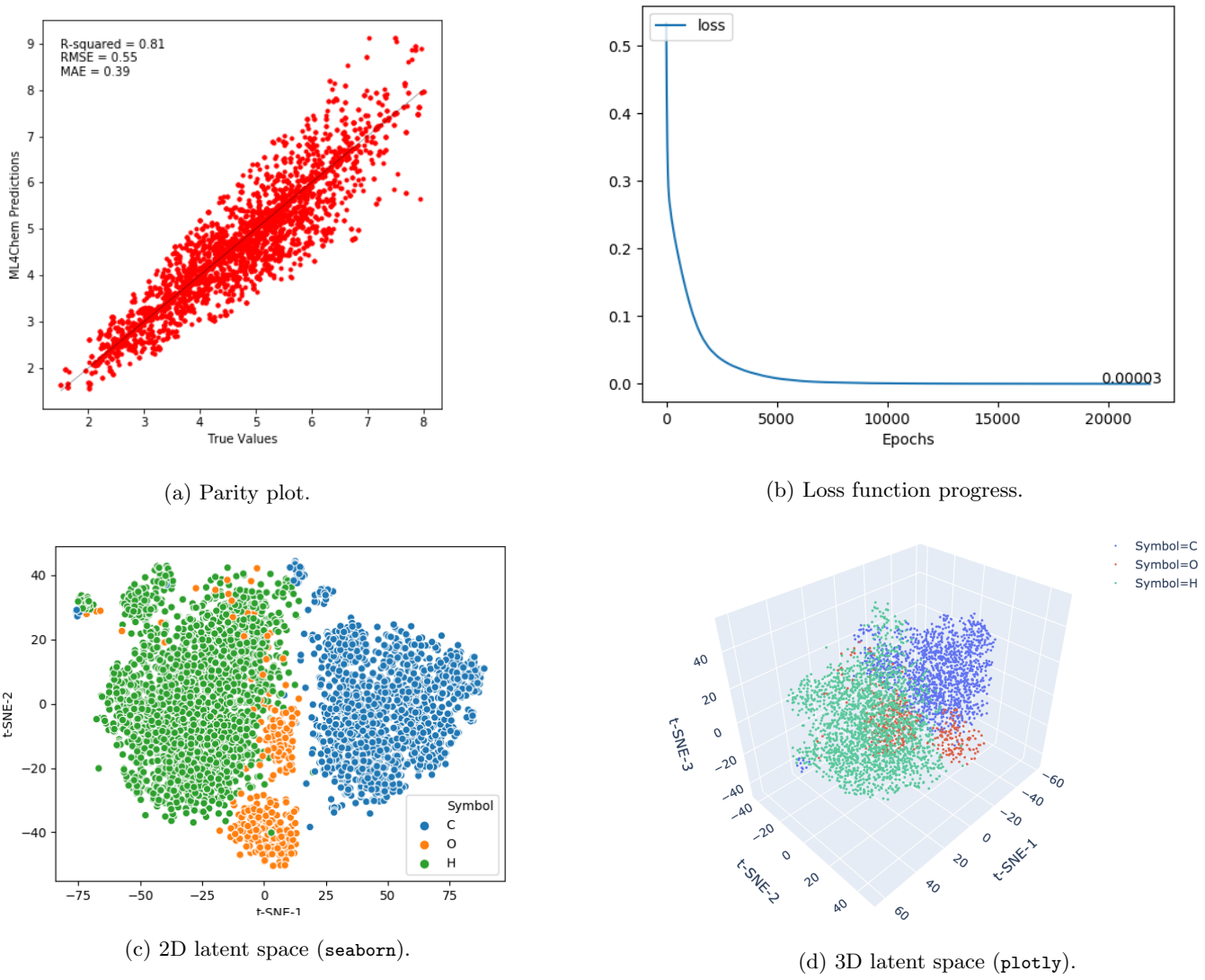
```python
from ml4chem.atomistic import Atomistic
from ase.io import Trajectory

# Save a model
Atomistic.save(nn, features=gaussian, path="", label="nn")

# Load a model
molecules = Trajectory("test.traj")
calc = Atomistic.load(model="nn.ml4c", params="nn.params", preprocessor="nn.scaler")

for molecule in molecules:
    energy = calc.get_potential_energy(molecule)
```

Listing 9: Example of save, and load trained neural network model in ML4Chem.



(a) Parity plot.

(b) Loss function progress.

(c) 2D latent space (`seaborn`).

(d) 3D latent space (`plotly`).

FIG. 8: Visualization tools offered in ML4Chem.

Another important part of ML pipelines has to do with visualization. We provide a visualization module built on top of matplotlib[65], seaborn and plotly[66]. In Figure 8 we show a parity plot between ML4Chem predictions and true values, the progress of the loss function in real-time, a 2D visualization of atomistic latent space obtained using a VAE, and its 3D interactive visualization with `plotly`. All these plots are offered as standalone functions. We also provide a command-line tool called ml4chem that allows quick access to these visualizations. For example, the latent space visualization shown in Figure 8c can be plotted from a stored features file using the command line `ml4chem --plot tsne --file latent_space.db`.

IV. DEMONSTRATIONS OF THE ML4CHEM FRAMEWORK

In this section, we present some demonstrations of running ML pipelines with ML4Chem. An atomistic neural network is trained with the ANI-1 data set, and a kernel ridge regression algorithm is trained with the QM7 data set.

A. Data Sets

The ANI-1 data set[32] is publicly available at `https://github.com/isayev/ANI1_dataset`. It is provided in HDF5 format, and can be used with the `atomistic` module of ML4Chem when converted to ASE and passed to the `Data` class. In Listing 10, we show how to load ANI-1 files to memory with `pyanitools` and assign them to a `ani_data` variable. Next, this list is passed as an argument to the ML4Chem's `ani_to_ase()` parser function that converts HDF5 ANI-1 data sets to ASE trajectory files. After the conversion is done, the trajectory file is saved to disk as `ani.traj`.

```
import ase
import pyanitools as pya
from ml4chem.data.parser import ani_to_ase
from ml4chem.data.utils import split_data

# Load ANI-1 hdf5 files to a list
files = ["ani_gdb_s01.h5", "ani_gdb_s02.h5","ani_gdb_s03.h5"]
ani_data = [pya.anidataloader(f) for f in files]

# Pass list of hdf5 files to ML4Chem ani_to_ase() parser function.
ani_dataset = ani_to_ase(
    ani_data,
    data_keys=["energies"],
    trajfile="ani.traj"
)

# Load trajectory file and split data in 80% training and 20% test set
traj = ase.io.Trajectory("ani.traj")
split_data(traj, test_set=20, randomize=True)
```

Listing 10: Conversion of ANI-1 data set to the atomic simulation environment format (ASE).

The QM7[67, 68] is available at `http://quantum-machine.org/datasets` as a Matlab file. According to the website, "The data set is composed of three multidimensional arrays X (7165 x 23 x 23), T (7165) and P (5 x 1433) representing the inputs (Coulomb matrices), the labels (atomization energies) and the splits for cross-validation, respectively. The data set also contains two additional multidimensional arrays Z (7165) and R (7165 x 3) representing the atomic charge and the Cartesian coordinate of each atom in the molecules". Its conversion into an ASE trajectory file to be used in `atomistic` module of ML4Chem is trivial as illustrated in Listing 11.

```
import scipy.io as sio
import ase
from ml4chem.data.utils import split_data



traj = ase.io.Trajectory("qm7.traj", mode="w")
dataset = sio.loadmat("qm7.mat")

# Cartesian coordinates
cartesian = dataset["R"]

# Atomic charge (Z), atomization energy (T)
Z = dataset["Z"]
T = dataset["T"]

for i, molecule in enumerate(Z):
    numbers, positions = [], []
    for j, z in enumerate(molecule):
        if z != 0:
            numbers.append(z)
            pos = tuple(cartesian[i][j])
            pos = [c * ase.units.Bohr for c in pos]
            positions.append(pos)
    atoms = ase.Atoms(numbers=numbers, positions=positions)

    traj.write(atoms, energy=float(T[0][i]))

# Load trajectory file and split data in 80% training and 20% test set
traj = ase.io.Trajectory("qm7.traj")
split_data(traj)
```

Listing 11: Conversion of QM7 data set to the atomic simulation environment format (ASE).

Each data set was randomized and split in 80% as training set and 20% as test set using the `split_data()` function.

B. Neural Network with Gaussian Features

In this demonstration we trained an atomistic machine learning potential using the `NeuralNetwork` class with Gaussian type features and the ANI-1 data set. For this example, we will use the high-level `Potentials` class.

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


def run():
    # Use 500 molecules for a total of 3606 feature vectors
    n_molecules, batch_size = 500, 30
    logger("nn.log")

    training = read("../training_images.traj", index="0:{}:1".format(n_molecules))

    # Instatiate the Potentials class
    calc = Potentials(
        features=Gaussian(
            batch_size=batch_size, cutoff=6.5, normalized=True, save_preprocessor="model.scaler"
        ),
        model=NeuralNetwork(
            hiddenlayers=(10, 10), activation="tanh"
        ),
        label="nn_training",
    )

    # Optimizer options and convergence criterion
    convergence = {"energy": 5e-2}
    lr = 1.0e-2
    weight_decay = 1e-5
    optimizer = ("adam", {"lr": lr, "weight_decay": weight_decay})

    # Train the algorithm
    calc.train(
        training_set=training, convergence=convergence, optimizer=optimizer, batch_size=batch_size
    )


if __name__ == "__main__":
    cluster = LocalCluster(n_workers=16, threads_per_worker=1)
    client = Client(cluster)
    run()
```

Listing 12: Training a neural network algorithm in ML4Chem using the `Potentials` class.

In Listing 12 the `Potentials` class can be instantiated with any of the `atomistic.features` and `atomistic.models` objects of ML4Chem, and automates all of the necessary steps to train an atomistic ML potential.

The architecture of the NN is of two hidden layers of 10 nodes each, a hyperbolic tangent activation function,

and the ADAM optimizer. The convergence criterion is set to be a root-mean-squared error of 0.05 Hartree. The optimization converged at about 5000 epochs.

In Figure 9 we show a parity plot with results obtained by predicting over 1000 unknown molecules of the test set. The NN model can do predictions with high accuracy, achieving a MAE 0.04 Hartree.

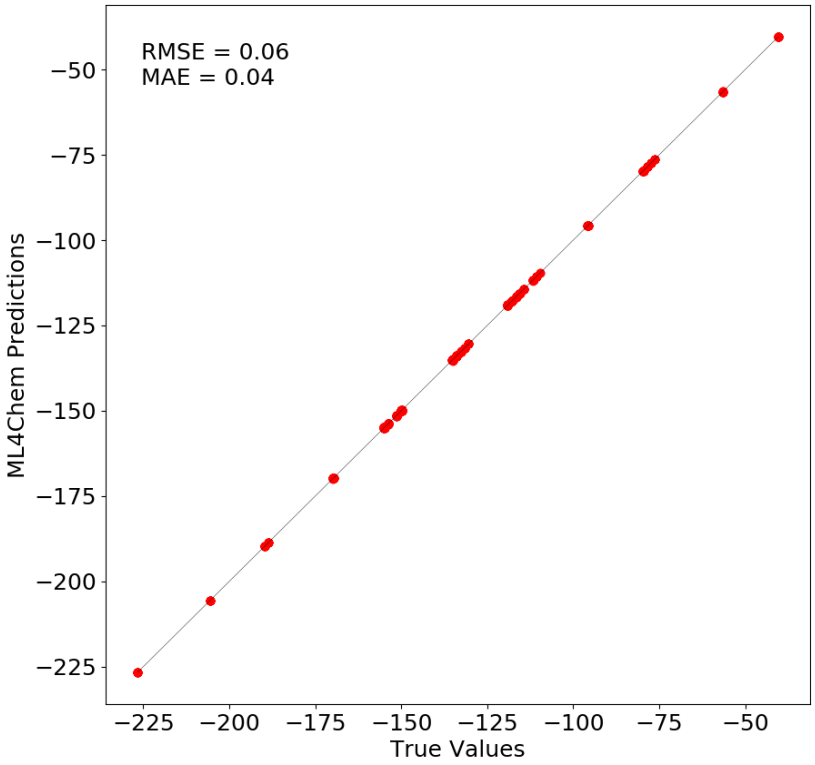


FIG. 9: Parity plot of neural network atomistic model predictions over 1000 molecules in the ANI-1 data set. Units are in Hartree.

It is worth noting that data points in the ANI-1 dataset are obtained through normal model sampling (NMS) around the equilibrium geometry. Therefore, Gaussian features of atoms are very close to each other in the feature space. To prove this hypothesis we carried out a principal component analysis (PCA) dimensionality reduction of the Gaussian features of the training and test sets. In Figure 10, we can see how the training and test data points are close in the lower PCA dimensional space making it easy for the model to correctly predict them.

C. Kernel Ridge Regression with Coulomb Matrix Features

In this demonstration, we train a kernel ridge regression (KRR) model for the task of predicting atomization energies with the QM7 data set. Instead of the `Potentials` class, we will use a modular approach to show the flexibility of ML4Chem. We proceed, as shown in Section III A, to the instantiation of the `Data()` class with the `qmt7.traj` trajectory

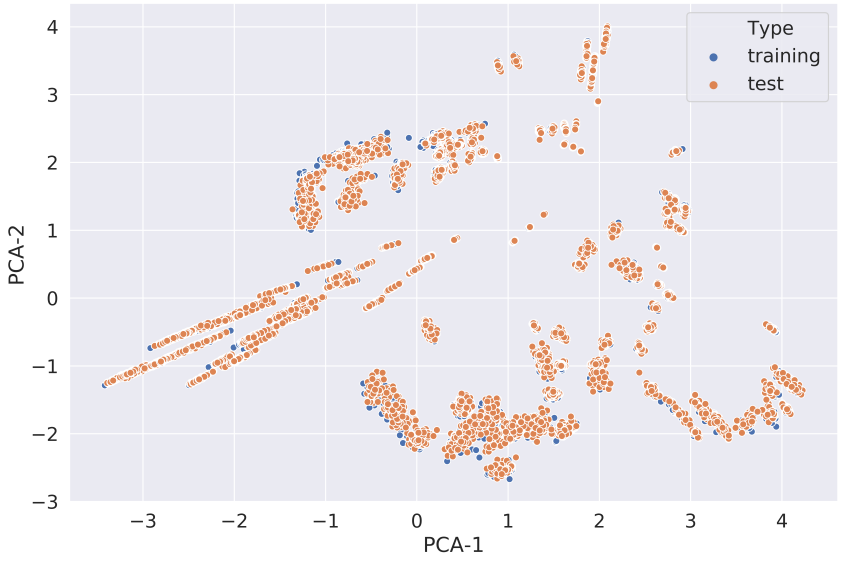


FIG. 10: Dimensionality reduction with principal component analysis (PCA) of Gaussian features of training and test sets.

file for the purpose of `"training"`. The training set and targets are obtained with the `.get_data()` class method to interoperate with ML4Chem (see Listing 13).

We use the rows of the Coulomb matrix features [69] as atomistic features vectors. The training set is featurized by instantiating the `CoulombMatrix` class implemented using the DScribe package and calling the `.calculate()` method. In the next step, the `KernelRidge` class is instantiated with keyword arguments to set training data, batch size, kernel function (radial basis function, or RBF), and a sigma value. The model is prepared with features and reference space. In the last step, we call the `.train()` method to fit the KRR algorithm passing as arguments the training set, targets, and `Data` objects and the model's parameters are saved to disk.

Now, we can proceed to load this model and predict unknown data points (see Listing 14). The results are shown in a parity plot in Figure 11 with the root-mean-squared (RMSE) and mean squared (MAE) error metrics. This model showed an MAE of 14.2 kcal/mol, compared to 9.9 kcal/mol when using the same RBF kernel on the Coulomb matrix sorted eigenspectrum in Ref. [34]. This significant difference is expected because, for this demonstration, we determined sigma to be the average of the euclidean distance of the Coulomb matrix atomic feature vectors. This is not an exhaustive way of determining this hyperparameter, and in practice, one has to rely on k-fold cross-validation to find the best value that fits the data of interest. Also, the number of training set data points in our demonstration is smaller than the used in Ref [34].

```python
from ml4chem.data.handler import Data
from ase.io import read
from ml4chem.utils import logger
from ml4chem.atomistic.features import CoulombMatrix
from ml4chem.atomistic.models import KernelRidge
from ml4chem.atomistic import Atomistic
from dask.distributed import Client, LocalCluster


def run():
    # Use 500 molecules for a total of 7727 feature vectors
    n_molecules, batch_size = 500, 10

    # Start a logger object to write to file
    logger("experiments.log")

    # Read training data
    training = read("training_images.traj", index="0:{}:1".format(n_molecules))

    # Prepare Data object
    data = Data(training, purpose="training")
    training, targets = data.get_data(purpose="training")

    # Featurization using Coulomb Matrix
    n_atoms_max = max(data.atoms_per_image)
    cm = CoulombMatrix(n_atoms_max=n_atoms_max, batch_size=batch_size)
    features, reference_space = cm.calculate(training, data=data, svm=True, purpose="training")

    # Instatiate model, prepare and train. sigma is set to average Euclidean
    # distance of feature vectors
    krr = KernelRidge(
        trainingimages="training_images.traj", batch_size=batch_size, kernel="rbf",
        sigma=26.808046418478668
    )

    krr.prepare_model(features, reference_space, data=data, purpose="training")
    krr.train(training, targets, data=data)

    # Save model to disk
    Atomistic.save(krr, features=cm, path="krr/", label="publication")


if __name__ == "__main__":
    cluster = LocalCluster(n_workers=15, threads_per_worker=1)
    client = Client(cluster)
    run()
```

Listing 13: Training a kernel ridge regression algorithm in ML4Chem.

```python
from ase.io import read
from ml4chem.utils import logger
from ml4chem.atomistic import Atomistic
from dask.distributed import Client, LocalCluster
from ml4chem.data.visualization import parity
import pandas as pd


def run():
    # Use 1000 molecules from the test set
    n_molecules = 1000

    # Start a logger object to write to file
    logger("inference.log")

    # Read test data
    calc = Atomistic.load(model="krr/publication.ml4c", params="krr/publication.params")

    # Set the reference space
    calc.reference_space = "features.db"

    # Compute energies
    energies = []
    trues = []

    for index, atoms in enumerate(test):
        energy = calc.get_potential_energy(atoms)
        true = atoms.get_potential_energy()
        print(true, energy)
        trues.append(true)
        energies.append(energy)

    df = pd.DataFrame(
        {"ML4Chem Energies": energies, "True Values": trues}
    )
    df.to_pickle("inference_results.pkl")

    parity(
        energies, trues, scores=True, filename="parity_inference.png"
    )


if __name__ == "__main__":
    cluster = LocalCluster(n_workers=15,threads_per_worker=1)
    client = Client(cluster)
    run()
```

Listing 14: Inference using trained kernel ridge regression ML4Chem parameters.

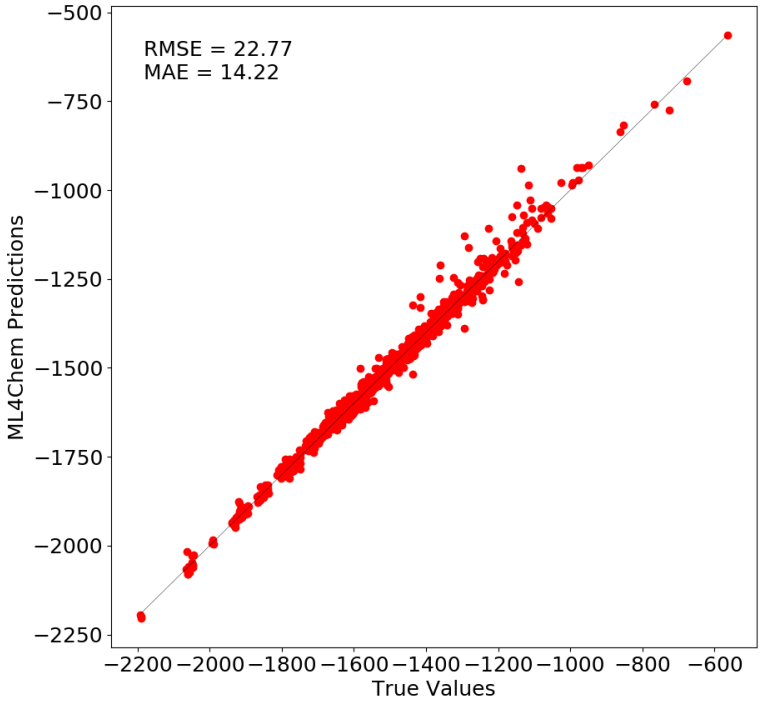


FIG. 11: Parity plot of kernel ridge regression predictions over 1000 molecules in the QM7 data set. Units are in kcal/mol.

V. CONCLUSIONS

We presented the `atomistic` module of ML4Chem, an open-source software package aiming to ease the deployment and implementation of ML models in chemistry and materials science. Its structure is designed with strict modularity and defined in such a way that each of its parts can be used as standalone programs. ML4Chem provides all needed methods and tools for an ML discovery pipeline – that is to go from raw data to inference and visualization. We showed with code snippets and demonstration cases what can be achieved with the core `atomistic` modules, and the intended intuitiveness derived from the use of UX design rules. These attributes make ML4Chem a potential platform for the implementation of new models, targeted to both non-experts and expert users.

For future development directions, we plan to extend the support of other input data formats of the `atomistic` module beyond the atomic simulation environment (ASE); introduce a `geometric` module in ML4Chem to work with geometric deep learning; addition of convolutional neural networks for both the `atomistic` and `geometric` modules; transfer learning; and implementation of active learning protocols.

VI. ACKNOWLEDGEMENT

We acknowledge the Laboratory Directed Research and Development of Lawrence Berkeley National Laboratory for funding under the project ID: 105702. MEK acknowledges Thom Popovici (LBL) for fruitful discussions related to high-performance computing parallelism, Scott Sievert (UW–Madison) and Matthew Rocklin (NVIDIA) for helpful advice on the Dask library.

---

[1] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[2] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

[3] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In H Wallach, H Larochelle, A Beygelzimer, F d\textquotesingle Alché-Buc, E Fox, and R Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[4] Nongnuch Artrith and Jörg Behler. High-dimensional neural network potentials for metal surfaces: A prototype study for copper. *Physical Review B*, 85(4):045439, jan 2012.

[5] Nongnuch Artrith, Tobias Morawietz, and Jörg Behler. High-dimensional neural-network potentials for multicomponent systems: Applications to zinc oxide. *Physical Review B*, 83(15):153101, apr 2011.

[6] Jörg Behler. Neural network potential-energy surfaces for atomistic simulations. In *Chemical Modelling*, pages 1–41. Royal Society of Chemistry, Cambridge, 2010.

[7] Jörg Behler and Michele Parrinello. Generalized Neural-Network Representation of High-Dimensional Potential-Energy Surfaces. *Physical Review Letters*, 98(14):146401, apr 2007.

[8] Ghanshyam Pilania, Chenchen Wang, Xun Jiang, Sanguthevar Rajasekaran, and Ramamurthy Ramprasad. Accelerating materials property predictions using machine learning. *Scientific Reports*, 3(1):2810, dec 2013.

[9] Peter Toth, Danilo Jimenez Rezende, Andrew Jaegle, Sébastien Racanière, Aleksandar Botev, and Irina Higgins. Hamiltonian Generative Networks. pages 1–17, sep 2019.

[10] K. T. Schütt, M. Gastegger, A. Tkatchenko, K.-R. Müller, and R. J. Maurer. Unifying machine learning and quantum chemistry with a deep neural network for molecular wavefunctions. *Nature Communications*, 10(1):5024, dec 2019.

[11] Rafael Gómez-Bombarelli, Jennifer N. Wei, David Duvenaud, José Miguel Hernández-Lobato, Benjamín Sánchez-Lengeling, Dennis Sheberla, Jorge Aguilera-Iparraguirre, Timothy D. Hirzel, Ryan P. Adams, and Alán Aspuru-Guzik. Automatic Chemical Design Using a Data-Driven Continuous Representation of Molecules. *ACS Central Science*, 4(2):268–276, feb 2018.

[12] Benjamin Sanchez-Lengeling, Jennifer N. Wei, Brian K. Lee, Richard C. Gerkin, Alán Aspuru-Guzik, and Alexander B. Wiltschko. Machine Learning for Scent: Learning Generalizable Perceptual Representations of Small Molecules. oct 2019.

[13] Matthew Hutson. Artificial intelligence faces reproducibility crisis. *Science*, 359(6377):725 LP – 726, feb 2018.

[14] Aaron Stupple, David Singerman, and Leo Anthony Celi. The reproducibility crisis in the age of digital medicine. *npj Digital Medicine*, 2(1):2, dec 2019.

[15] Democratizing deep-learning for drug discovery, quantum chemistry, materials science and biology. `https://github.com/deepchem/deepchem`, 2016.

[16] Mojtaba Haghighatlari, Gaurav Vishwakarma, Doaa Altarawy, Ramachandran Subramanian, Bhargava U. Kota, Aditya Sonpal, Srirangaraj Setlur, and Johannes Hachmann. Chemml: A machine learning and informatics program package for the analysis, mining, and modeling of chemical and materials data. *WIREs Computational Molecular Science*, n/a(n/a):e1458.

[17] Zhenwei Li, James R. Kermode, and Alessandro De Vita. Molecular Dynamics with On-the-Fly Machine Learning of Quantum-Mechanical Forces. *Physical Review Letters*, 114(9):096405, mar 2015.

[18] Venkatesh Botu and Rampi Ramprasad. Adaptive machine learning framework to accelerate ab initio molecular dynamics. *International Journal of Quantum Chemistry*, 115(16):1074–1083, aug 2015.

[19] Stefan Chmiela, Alexandre Tkatchenko, Huziel E. Sauceda, Igor Poltavsky, Kristof T. Schütt, and Klaus-Robert Müller. Machine learning of accurate energy-conserving molecular force fields. *Science Advances*, 3(5):e1603015, may 2017.

[20] Linglong Li, Yaodong Yang, Dawei Zhang, Zuo Guang Ye, Stephen Jesse, Sergei V. Kalinin, and Rama K. Vasudevan. Machine learning-enabled identification of material phase transitions based on experimental data: Exploring collective dynamics in ferroelectric relaxors. *Science Advances*, 4(3), 2018.

[21] Venkatesh Botu and R. Ramprasad. Learning scheme to predict atomic forces and accelerate materials simulations. *Physical Review B*, 92(9):094306, sep 2015.

[22] Venkatesh Botu, R. Batra, J. Chapman, and R. Ramprasad. Machine Learning Force Fields: Construction, Validation, and Outlook. *The Journal of Physical Chemistry C*, 121(1):511–522, 2017.

[23] Andrew A. Peterson. Acceleration of saddle-point searches with machine learning. *The Journal of Chemical Physics*, 145(7):074106, aug 2016.

[24] S. Alireza Ghasemi, Albert Hofstetter, Santanu Saha, and Stefan Goedecker. Interatomic potentials for ionic systems with density functional accuracy based on charge densities obtained by a neural network. *Physical Review B*, 92(4):045131, jul 2015.

[25] Somayeh Faraji, S. Alireza Ghasemi, Samare Rostami, Robabe Rasoulkhani, Bastian Schaefer, Stefan Goedecker, and Maximilian Amsler. High accuracy and transferability of a neural network potential through charge equilibration for calcium fluoride. *Physical Review B*, 95(10):104105, mar 2017.

[26] Stéfan Van Der Walt, S Chris Colbert, and Gaël Varoquaux. The NumPy array: A structure for efficient numerical computation. *Computing in Science and Engineering*, 13(2):22–30, 2011.

[27] K. Jarrod Millman and Michael Aivazis. Python for scientists and engineers. *Computing in Science and Engineering*, 13(2):9–12, 2011.

[28] Pauli Virtanen, Ralf Gommers, Travis E Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J van der Walt, Matthew Brett, Joshua Wilson, K Jarrod Millman, Nikolay Mayorov, Andrew R.˜J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, \.Ilhan Polat, Yu Feng, Eric W Moore, Jake Vand erPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E.˜A. Quintero, Charles R Harris, Anne M Archibald, Antônio H Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1. 0 Contributors. SciPy 1.0–Fundamental Algorithms for Scientific Computing in Python. *arXiv e-prints*, page arXiv:1907.10121, jul 2019.

[29] Dask Development Team. Dask: Library for dynamic task scheduling, 2016.

[30] Ask Hjorth Larsen, Jens Jørgen Mortensen, Jakob Blomqvist, Ivano E Castelli, Rune Christensen, Marcin Dułak, Jesper Friis, Michael N Groves, Bjørk Hammer, Cory Hargus, Eric D Hermes, Paul C Jennings, Peter Bjerre Jensen, James Kermode, John R Kitchin, Esben Leonhard Kolsbjerg, Joseph Kubal, Kristen Kaasbjerg, Steen Lysgaard, Jón Bergmann

Maronsson, Tristan Maxson, Thomas Olsen, Lars Pastewka, Andrew Peterson, Carsten Rostgaard, Jakob Schiøtz, Ole Schütt, Mikkel Strange, Kristian S Thygesen, Tejs Vegge, Lasse Vilhelmsen, Michael Walter, Zhenhua Zeng, and Karsten W Jacobsen. The atomic simulation environment—a python library for working with atoms. *Journal of Physics: Condensed Matter*, 29(27):273002, jul 2017.

[31] Marcus D. Hanwell, Wibe A. De Jong, and Christopher J. Harris. Open chemistry: RESTful web APIs, JSON, NWChem and the modern web application. *Journal of Cheminformatics*, 9(1):1–10, 2017.

[32] J. S. Smith, O. Isayev, and A. E. Roitberg. Ani-1: an extensible neural network potential with dft accuracy at force field computational cost. *Chem. Sci.*, 8(4):3192–3203, 2017.

[33] Shyue Ping Ong, William Davidson Richards, Anubhav Jain, Geoffroy Hautier, Michael Kocher, Shreyas Cholia, Dan Gunter, Vincent L. Chevrier, Kristin A. Persson, and Gerbrand Ceder. Python Materials Genomics (pymatgen): A robust, open-source python library for materials analysis. *Computational Materials Science*, 68:314–319, feb 2013.

[34] Matthias Rupp, Alexandre Tkatchenko, Klaus-Robert Müller, and O. Anatole von Lilienfeld. Fast and Accurate Modeling of Molecular Atomization Energies with Machine Learning. *Physical Review Letters*, 108(5):058301, jan 2012.

[35] Albert P. Bartók, Risi Kondor, and Gábor Csányi. On representing chemical environments. *Physical Review B*, 87(18):184115, may 2013.

[36] Haoyan Huo and Matthias Rupp. Unified Representation of Molecules and Crystals for Machine Learning. apr 2017.

[37] Lauri Himanen, Marc O. J. Jäger, Eiaki V. Morooka, Filippo Federici Canova, Yashasvi S. Ranawat, David Z. Gao, Patrick Rinke, and Adam S. Foster. DScribe: Library of Descriptors for Machine Learning in Materials Science. apr 2019.

[38] Alireza Khorshidi and Andrew A. Peterson. Amp : A modular approach to machine learning in atomistic simulations. *Computer Physics Communications*, 207:310–324, oct 2016.

[39] Jörg Behler. Constructing high-dimensional neural network potentials: A tutorial review. *International Journal of Quantum Chemistry*, 115(16):1032–1050, aug 2015.

[40] F Pedregosa, G Varoquaux, A Gramfort, V Michel, B Thirion, O Grisel, M Blondel, P Prettenhofer, R Weiss, V Dubourg, J Vanderplas, A Passos, D Cournapeau, M Brucher, M Perrot, and E Duchesnay. Scikit-learn: Machine Learning in {P}ython. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[41] Denny Borsboom, Gideon J. Mellenbergh, and Jaap Van Heerden. The Theoretical Status of Latent Variables. *Psychological Review*, 110(2):203–219, 2003.

[42] Hao Fu, Chunyuan Li, Xiaodong Liu, Jianfeng Gao, Asli Celikyilmaz, and Lawrence Carin. Cyclical Annealing Schedule: A Simple Approach to Mitigating. pages 240–250, 2019.

[43] Francesco Ricci, Lior Rokach, and Bracha Shapira. Introduction to recommender systems handbook. In *Recommender systems handbook*, pages 1–35. Springer, 2011.

[44] Yu-Hang Tang and Wibe A. de Jong. Prediction of Atomization Energy Using Graph Kernel and Active Learning. pages 1–19, oct 2018.

[45] Seymour Geisser. *Predictive inference*. Routledge, 2017.

[46] Andriy Burkov. *The Hundred-Page Machine Learning Book*. Kindle Direct Publishing, 1 edition, 2019.

[47] Kevin P. Murphy. *Machine Learning*. Springer-Verlag, Berlin/Heidelberg, 2012.

[48] Stephen Marsland. *Machine learning: an algorithmic perspective*. Chapman and Hall/CRC, dec 2014.

[49] A. Sowmya. The anisotropic Gaussian kernel for SVM classification of HRCT images of the lung. *Proceedings of the 2004 Intelligent Sensors, Sensor Networks and Information Processing Conference, 2004.*, pages 439–444, 2004.

[50] Matthias Rupp. Machine learning for quantum mechanics in a nutshell. *International Journal of Quantum Chemistry*, 115(16):1058–1073, aug 2015.

[51] Albert P. Bartók, Mike C. Payne, Risi Kondor, and Gábor Csányi. Gaussian Approximation Potentials: The Accuracy of Quantum Mechanics, without the Electrons. *Physical Review Letters*, 104(13):136403, apr 2010.

[52] Albert P. Bartók and Gábor Csányi. Gaussian approximation potentials: A brief tutorial introduction. *International Journal of Quantum Chemistry*, 115(16):1051–1057, aug 2015.

[53] Tobias Lemke and Christine Peter. EncoderMap: Dimensionality Reduction and Generation of Molecule Conformations. *Journal of Chemical Theory and Computation*, 15(2):1209–1215, feb 2019.

[54] Alireza Makhzani and Brendan Frey. k-Sparse Autoencoders. dec 2013.

[55] Devansh Arpit, Yingbo Zhou, Hung Ngo, and Venu Govindaraju. Why Regularized Auto-Encoders learn Sparse Representation? may 2015.

[56] Diederik P Kingma and Max Welling. Auto-Encoding Variational Bayes. *arXiv preprint arXiv:1312.6114*, dec 2013.

[57] X Hou, L Shen, K Sun, and G Qiu. Deep Feature Consistent Variational Autoencoder. In *2017 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pages 1133–1141, mar 2017.

[58] Shengjia Zhao, Jiaming Song, and Stefano Ermon. Towards Deeper Understanding of Variational Autoencoding Models. feb 2017.

[59] Solomon Kullback and Richard A Leibler. On information and sufficiency. *The annals of mathematical statistics*, 22(1):79–86, 1951.

[60] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. Understanding the exploding gradient problem. *CoRR, abs/1211.5063*, 2, 2012.

[61] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 315–323, 2011.

[62] Diederik P. Kingma and Jimmy Lei Ba. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings*, dec 2015.

[63] Herbert Robbins and Sutton Monro. A stochastic approximation method. *The annals of mathematical statistics*, pages 400–407, 1951.

[64] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

[65] J D Hunter. Matplotlib: A 2D graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[66] Plotly Technologies Inc. Collaborative data science. 2015.

[67] M. Rupp, A. Tkatchenko, K.-R. Müller, and O. A. von Lilienfeld. Fast and accurate modeling of molecular atomization energies with machine learning. *Physical Review Letters*, 108:058301, 2012.

[68] L. C. Blum and J.-L. Reymond. 970 million druglike small molecules for virtual screening in the chemical universe database GDB-13. *J. Am. Chem. Soc.*, 131:8732, 2009.

[69] Christopher R. Collins, Geoffrey J. Gordon, O. Anatole Von Lilienfeld, and David J. Yaron. Constant size descriptors for accurate machine learning models of molecular properties. *J. Chem. Phys.*, 148(24), jun 2018.