

Implementation of a nonlinear Atomic Cluster Expansion

by

Andres Ross

B.S. Honours, McGill University, 2020

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL
STUDIES

(Mathematics)

The University of British Columbia
(Vancouver)

April 2022

© Andres Ross, 2022

The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, the thesis entitled:

Implementation of a nonlinear Atomic Cluster Expansion

submitted by **Andres Ross** in partial fulfillment of the requirements for the degree of **Master of Science in Mathematics**.

Examining Committee:

Christoph Ortner, Professor, Mathematics, UBC
Supervisor

Chad Sinclair, Professor, Materials Engineering, UBC
Supervisory Committee Member

Khanh Dao Duc, Professor, Mathematics, UBC
Supervisory Committee Member

Abstract

In this thesis, we present a proof of concept implementation of linear and nonlinear models based on the Atomic Cluster Expansion (ACE) introduced in [16]. We introduce machine-learned interatomic potentials and derive the ACE as an atomic descriptor. This produces a model linear in its coefficient that serves to approximate the energies and forces of an atomic configuration. We train its coefficients for Silicon, Copper, and Molybdenum, and analyze the fit accuracy for energies and forces benchmark training sets [37]. Furthermore, we extend the ACE model to approximate energies and forces through a nonlinear combination of linear ACE models. We describe how to implement this model, and in particular, how to efficiently compute the derivatives, and present example results for the same data sets. We summarize the Julia implementation of these nonlinear models and provide an overview of the direction the code base will take in the future.

Lay Summary

Modelling materials at the atomic scale has become a crucial part of scientific research. However, simulating thousands or even millions of atoms directly with quantum mechanics is highly costly. A way to reduce such a cost is to generate a surrogate model trained by machine learning with data from a high fidelity model. In this thesis, we explore a particular class of surrogate models. We develop the mathematical theory, describe their practical implementation, and test them on benchmark data.

Preface

This thesis is original, unpublished, independent work by the author, Andres Ross under the supervision of Dr. Christoph Ortner.

Table of Contents

Abstract	iii
Lay Summary	iv
Preface	v
Table of Contents	vi
List of Tables	viii
List of Figures	ix
Nomenclature	x
1 Introduction and Background	1
1.1 Materials Modeling	1
1.2 Calculating the PES	1
1.3 Interatomic potentials	4
1.3.1 Pair Potentials	4
1.4 Machine Learned Interatomic Potentials	5
1.4.1 Gaussian approximation potentials	7
1.4.2 Moment Tensor Potentials	8
2 The Linear ACE Model	9
2.1 The Atomic Cluster Expansion	9
2.2 Tensor approximation	11

2.3	Parameter estimation	16
2.4	Results	18
3	Nonlinear ACE Model	22
3.1	Model Definition	22
3.1.1	Loss function	24
3.2	Efficient computation	25
3.3	Results	27
4	Implementation	31
4.1	List of neighbours	31
4.2	Reverse mode differentiation: Example in ACE	32
4.3	Introduction to the code base	35
4.4	Implementing nonlinear combinations	37
4.4.1	Multiple properties	37
4.4.2	Forward pass	37
4.4.3	Making models differentiable	38
4.5	Training	40
4.5.1	Multiprocessing	40
4.5.2	Interacting with optimization packages in Julia	40
4.6	Example	41
5	Conclusion and Outlook	44
5.1	Conclusion	44
5.2	Outlook	44
	Bibliography	48

List of Tables

Table 2.1	RMSE for energy and forces. κ represents the basis size used for that value.	20
-----------	--	----

List of Figures

Figure 1.1	Different scales of materials modeling (courtesy of Gabor Csanyi).	2
Figure 2.1	An example of a set R of $J = 22$ particles described by their position vectors \mathbf{r}_j , with the distances between them as \mathbf{r}_{ij} . R_i is the atomic neighbourhood of i such that $ \mathbf{r}_{ij} < r_{\text{cut}}$. In the figure $i = 9$ has neighbourhood of all hollow atoms.	10
Figure 2.2	Log RMSE for varying ϵ and α_E/α_F	19
Figure 2.3	Log RMSE v.s log basis size for energy and forces.	21
Figure 3.1	(ii) embedding, 3 dense layers.	28
Figure 3.2	Log RMSE v.s iterations for energy and forces.	29
Figure 4.1	Example situation of f_i for 14 atoms. The left and the right represent $f_i(R)$ for two different atoms i . The number of atoms n inside the cutoff radius can (and usually is) of different length for different i	32
Figure 5.1	Example of f on a configuration for two atoms $i = \{7, 4\}$. On the right we see the atomic environment, and on the left we see the action of f on the input layer.	46
Figure 5.2	ACE model divided into layers.	46

Nomenclature

$\mathcal{E}(R_i)$ Site energy of R_i .

\mathcal{F} Nonlinear embedding.

\mathcal{N} correlation order.

\mathcal{P} The radial basis.

$\alpha_{\{E,F\}}$ Weighting of energies v.s forces.

ϵ RRQR tolerance.

κ Number of basis functions.

ϕ_{nlm} One particle basis.

φ_i An atomic property.

A_{inlm} Atomic basis.

\mathbf{A}_{inlm} Product basis.

\mathbf{B} ACE basis functions.

c_B coefficients corresponding to basis function B .

$E(R)$ Total energy of R .

F The forces.

F_i The gradient of energy centered at i .

f	Function to find the neighbours of an atom.
J	Number of atoms.
R	$:= \{\mathbf{r}_1, \dots, \mathbf{r}_J\}$.
R_i	$:= \{\mathbf{r}_{ij}\}_{i \neq j}$.
r_{cut}	A cutoff radius to limit the range of interaction.
\mathbf{r}_j	Position of an atom j .
\mathbf{r}_{ij}	$:= \mathbf{r}_j - \mathbf{r}_i$.
V_N	N^{th} body term in the expansion of \mathcal{E} .
Y_l^m	The angular basis.
$y^{(t)} = (y_E^t, y_F^t)$	A DFT calculated training point.

Chapter 1

Introduction and Background

1.1 Materials Modeling

Modelling of materials at the atomic scale has become a crucial part of scientific research [4]. From quantum mechanical models to interatomic potentials, approximations of the potential energy surface (PES) have a variety of applications. A PES describes the energy of a system as a function of specific atomic parameters, namely the positions of the atoms. Macroscopic properties of a material depend on its atomic structure, and hence the accurate prediction of the atomic structure is crucial to computational material discovery [32]. Employing the PES enables us to address challenges like atomic-scale deposition and growth of amorphous carbon films [7], proton-transfer mechanisms [20], or dislocations in materials [18] involving thousands or even millions of atoms. The performance of these simulations relies heavily on the quality of our PES, both its accuracy and computational cost.

1.2 Calculating the PES

The first approach to calculating the PES comes from Quantum Chemistry. Using the many-body Schrödinger equation, one can calculate the potential energy in terms of the wave function by minimizing the following expectation:

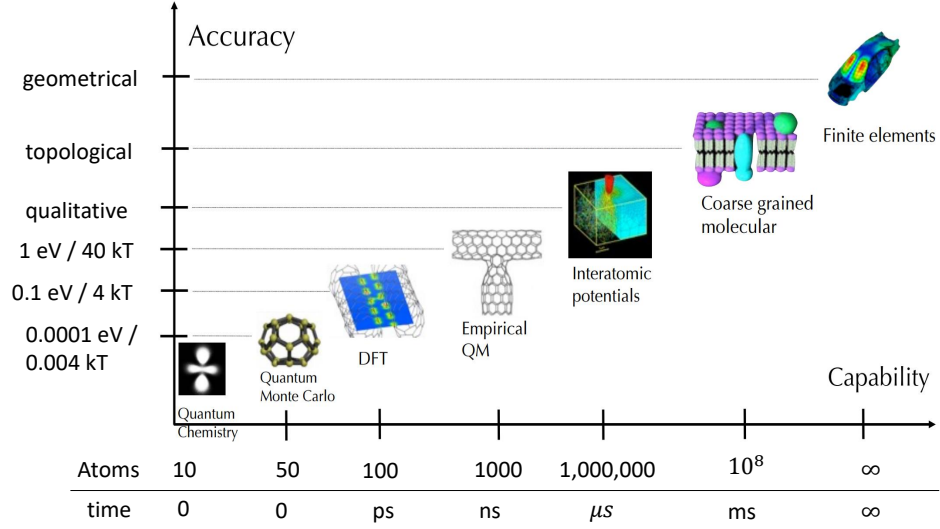


Figure 1.1: Different scales of materials modeling (courtesy of Gabor Csanyi).

$$E = \frac{\langle \Psi | H | \Psi \rangle}{\| \Psi \|^2}. \quad (1.1)$$

This allows for a complete description of the system, but it requires numerical approximation. For J atoms, let $R := \{\mathbf{r}_1, \dots, \mathbf{r}_J\}$ be an atomic structure, with $\mathbf{r}_j = (x, y, z)$ the positions of atom j , $r_j = |\mathbf{r}_j|$ its magnitude, and $\mathbf{r}_{j_1 j_2} := \mathbf{r}_{j_1} - \mathbf{r}_{j_2}$ the distance between atoms i and j . Then we can formulate the time-independent electronic Schrödinger equation with the Born-Oppenheimer PES

$$H(R)\Psi = E(R)\Psi, \quad (1.2)$$

as an eigenvalue problem where the energy levels are the eigenvalues of the system. However, solving this PDE with simple discretization results in exponential growth of the degrees of freedom with the number of electrons (the cause of high dimensionality), which is very costly. As we see in Figure 1.1, Quantum Chemistry is limited to only small atomic structures, on the order of at most a few atoms [6]. Although highly accurate, its cost makes it unfeasible for almost all applications.

In 1964 Hohenberg and Kohn [21] proved the existence of a universal density

functional that allowed for the calculation of energy. This functional is based on the electronic density and serves as an approximation of E in 1.2. It is the basis of what we now know as DFT (Density Functional Theory) and is faster than quantum chemistry while still retaining good accuracy [21]. We briefly present the general idea of DFT but quickly move on to methods without electrons.

In DFT models, the energy is given as the sum of external potential energy (V), kinetic energy (T), and interaction energy of the atoms (U) and is described by the Hamiltonian operator $H = T + V + U$, which is evaluated with (1.1). Assuming a non-degenerate ground state (i.e. a unique quantum state represents that energy), let us denote the electronic density by $\rho(\mathbf{r})$. This allows us to define a universal functional:

$$F[\rho(\mathbf{r})] := \langle \Psi, (T + U) \Psi \rangle. \quad (1.3)$$

On the other hand V is given by an external potential $v(\mathbf{r})$, as:

$$V = \int v(\mathbf{r}) \Psi^*(\mathbf{r}) \Psi(\mathbf{r}) d\mathbf{r}. \quad (1.4)$$

We can then replace the wavefunction by the electronic density $\rho(\mathbf{r})$ and use the density functional $F[\rho(\mathbf{r})]$ to replace the other energy contributions, which will leave us with a representation of the energy that only depends on the external potential $v(\mathbf{r})$ and the electronic density

$$E_v[\rho] := \int v(\mathbf{r}) \rho(\mathbf{r}) d\mathbf{r} + F[\rho]. \quad (1.5)$$

If the functionals v and F were chosen to only depend on the atomic position \mathbf{r} , then energy will also only depend on \mathbf{r} . One way to choose $\rho(\mathbf{r})$ is the Kohn-Sham method [25]. It assumes that the electron density of a system of J electrons can be written as the sum of one-electron orbitals ψ_i :

$$\rho(\mathbf{r}) = \sum_{i=1}^J |\psi_i(\mathbf{r})|^2. \quad (1.6)$$

Even when optimized, using the ab-initio DFT approach scales with $O(J^3)$ since the ψ_i solve our eigenvalue problem (1.2) [35], which limits the size of sim-

ulations one can realistically run with this approach and is the reason why interatomic potentials are useful.

1.3 Interatomic potentials

To overcome the cubic scaling cost of Kohn-Sham DFT, one can employ interatomic potentials. We start again from (1.2), and approximate the energy E without using the positions of electrons. Not using electrons is what accounts for the jump in capability in Figure 1.1 between 10^3 ns and 10^6 μ s [13].

One idea to construct interatomic potentials is to formally expand E into a series with each term a specific interaction order. The first term is the summation of the energies of each atom individually, the second term is pairwise interactions, and so forth:

$$\begin{aligned} E(R) \approx & V_0 + \sum_{j_1} V_1(\mathbf{r}_{j_1}) + \sum_{j_1 < j_2} V_2(\mathbf{r}_{j_1}, \mathbf{r}_{j_2}) + \dots \\ & + \sum_{j_1 < \dots < j_N} V_N(\mathbf{r}_{j_1}, \mathbf{r}_{j_2}, \dots, \mathbf{r}_{j_N}) + \dots \end{aligned} \quad (1.7)$$

There are several ways to model V_N , which depend on the type of interactions we care about in a specific case. For example, since the density of electrons around an atom decreases exponentially with distance, short-range interactions can be modeled with a repulsive functional $V_2(r_{j_1 j_2}) = A e^{-\alpha r_{j_1 j_2}}$, for some parameters A and α . Similarly if we wanted to account for Van der Waals interactions, we could use a functional of the form $V_2(r_{j_1 j_2}) = \frac{A}{r_{j_1 j_2}^6}$ for some material dependent parameter A [25]. Several energy models are derived using physics-inspired functions and truncating the series expansion. These models tend to be material-dependent, thus some will perform better than others in different cases.

1.3.1 Pair Potentials

When we truncate at pairwise interactions, we obtain pair potentials. Although, these models will generally be empirical and not particularly accurate [25], they provide value when investigating materials processes. One example is the Lennard-Jones potential which provides a good description of central-force interatomic in-

interactions $V_2(r_{j_1j_2}) = 4\epsilon((\frac{r_0}{r_{j_1j_2}})^{12} - (\frac{r_0}{r_{j_1j_2}})^6)$, where r_0 is the equilibrium distance that minimizes the potential. The long term contribution is represented with $r_{j_1j_2}^{-6}$, but when this potential was originally developed, short-range decay was not known, so it was approximated with $r_{j_1j_2}^{-12}$. We would get the Born-Mayer potential if we wanted to use exponential decay for short-range interactions. There are many other pair potentials that can perform well in some cases, but most of them fail to describe the properties of metals adequately because they fail to capture the local electron density [25]. Embedded-atom model potentials (EAM) were developed to address this by adding an energy functional of the local electron density. Motivated by DFT, they have the general form:

$$V = \sum_{j_1} F_i \left[\sum_{j_1 \neq j_2} f_{j_1j_2}(r_{j_1j_2}) \right] + \frac{1}{2} \sum_{j_1 \neq j_2} V_2(r_{j_1j_2}), \quad (1.8)$$

where f is a function that approximates the electron density and F models the cost of embedding on a nucleus into an electron cloud. A major limitation of EAM-type potentials is that they do not reflect dynamic changes that arise from changing the local environment [25]. Finnis and Sinclair introduced a similar potential which, based on a second-moment approximation to the tight-binding density of states, uses $f_{j_1j_2}(r_{j_1j_2}) \sim \sqrt{r_{j_1j_2}}$ [14].

Adding more interaction orders improves performance, which leads to higher-order potentials like the Stillinger-Weber potential [22] and modified embedded-atom method (MEAM) potential [24]. However, increasing the order of interaction N increases the cost of evaluation by $\binom{J}{N}$ where J is the number of atoms, which makes potentials with $N > 3$ very expensive.

1.4 Machine Learned Interatomic Potentials

One of the major problems with Interatomic potentials is that they are not transferable. As seen earlier, selecting the parameters and the functionals for a potential is a very situational choice. These potentials were often fitted using experimental quantities within an analytic framework which makes them transfer poorly, not only among materials but also among applications. In 1994 first-principles methods (methods with electrons) were making big leaps in their capabilities but were

nowhere near the capabilities of interatomic potentials. In an attempt to bridge this gap, Ercolessi and Adams proposed using data created by first principles to train interatomic potentials, arguing that a richer dataset would allow for increased transferability [17]. This was the beginning of Machine learned interatomic potentials which sparked a new movement to employ data from high fidelity models to fit interatomic potentials. In 2007 Behler and Parrinello proposed the use of a neural network representation of a potential energy surface using DFT data [3]. This network provided the energy and the forces directly as a function of all the atomic positions in a system. Their method was orders of magnitude faster than DFT, and they demonstrated high accuracy for bulk silicon compared to standard empirical interatomic potentials. They used a densely connected neural network with radial G_i^1 and angular G_i^2 functions as the input and energy as the output. The radial functions were constructed as a sum of Gaussians with parameters η and r_s ,

$$G_i^1(r) = \sum_{j \neq i} e^{-\eta(r_{ij}-r_s)^2} f_{\text{cut}}(r_{ij}) \quad (1.9)$$

where f_{cut} is a cutoff function that is 0 for values $r_{ij} > r_{\text{cut}}$ for some cutoff value r_{cut} . Define θ_{ijk} as the angle between \mathbf{r}_{ij} and \mathbf{r}_{ik} for a central atom i . Then The angular functions were constructed for all triplets of atoms by summing over the cosine values of θ_{ijk} ,

$$G_i^2(r) = 2^{1-\zeta} \sum_{j,k \neq i} \left[(1 + \lambda \cos(\theta_{ijk}))^\zeta \times \right. \quad (1.10)$$

$$\left. e^{-\eta(r_{ij}^2 + r_{ik}^2 + r_{jk}^2)} f_{\text{cut}}(r_{ij}) f_{\text{cut}}(r_{ik}) f_{\text{cut}}(r_{jk}) \right],$$

with the parameters λ , η , and ζ .

By using a Silicon data set with the positions of atoms as training points and the energies as the targets, Behler and Parrinello optimized a cost function to train the network. In total, 8200 DFT energies were used as training data and 800 were used as testing data, reaching 5-6 meV per atom root mean squared error (RMSE).

1.4.1 Gaussian approximation potentials

In 2010 Bartók, Payne, Kondor and Csányi introduced Gaussian approximation potentials (GAP), a class of interatomic potentials without a fixed functional form. These were created to maintain special symmetries of the system and to be automatically generated from DFT energies and forces data. We start by writing the total energy of a system as the sum of atomic energies E_i (introduced by Behler and Parrinello),

$$E := \sum_{i=0}^J \mathcal{E}(\{\mathbf{r}_{ij}\}_{i \neq j}), \quad (1.11)$$

where $\{\mathbf{r}_{ij}\}_{i \neq j}$ is the set of distances between the central atom i and the neighbouring atoms j . We then define a local density for atom i and its neighbours,

$$\rho_i(\mathbf{r}) := \delta(\mathbf{r}) + \sum_j \delta(\mathbf{r} - \mathbf{r}_{ij}) f_{\text{cut}}(|\mathbf{r}_{ij}|), \quad (1.12)$$

with f_{cut} a cutoff function as previously shown. With (1.12) we build a kernel G such that

$$\mathcal{E}(\{\mathbf{r}_{ij}\}_{i \neq j}) := \sum_n \alpha_n G(\mathbf{b}_i, \mathbf{b}_n), \quad (1.13)$$

with n ranging over the configurations, and α_n coefficients to be calculated. \mathbf{b}_i are the elements of the bispectrum [2] for atom i . The actual construction is omitted here but can be found in the paper [2].

They define a matrix C as

$$C_{nn'} := \delta^2 G(\mathbf{b}_n, \mathbf{b}_{n'}) + \sigma^2 I, \quad (1.14)$$

with σ and δ hyperparameters, and then solve for α_n with

$$\{\alpha_n\} = \boldsymbol{\alpha} = \mathbf{C}^{-1} \mathbf{y}. \quad (1.15)$$

This produces a symmetry preserving method that improves with more data. However, two problems that arise are that data contains only total energies and

forces and that these will be heavily correlated. To solve these, the authors propose using a sparsification procedure that reduces the data to a much smaller set of configurations and replaces \mathbf{y} with a linear combination of all data values. With this approach, they reached, for example, an RMSE of 1 meV per atom for Silicon energies [2].

1.4.2 Moment Tensor Potentials

In 2016 Shapeev proposed a class of systematically improvable interatomic potentials called Moment Tensor Potentials (MTP) [35]. We divide interatomic potentials into two broad classes, parametric and nonparametric. Parametric potentials have a fixed number of parameters, like empirical potentials, whose accuracy cannot be systematically improved. Nonparametric potentials can be systematically improved in theory. They are composed of two main components, a representation (also called a descriptor) of the atomic environment and a regression model. The aforementioned Behler and Parrinello’s neural network potentials (NNP) use neural networks as the regression model and radial and angular symmetry functions as the descriptors. GAP uses linear regression as the regression model (1.15) and the bispectrum as a descriptor. MTP also uses linear regression, but its novelty comes from using invariant polynomials as descriptors. These polynomials can provably approximate any regular function that satisfies smoothness with respect to the number of atoms and permutation, rotational and reflection invariance [35]. The main idea is that a potential V can be approximated by a polynomial that can be symmetrized, which allows us to construct a basis of such polynomials $B(\mathbf{r})$ as

$$V(R) \approx \sum_{B \in \mathcal{B}} c_B B(R). \quad (1.16)$$

The idea of using symmetrized polynomials in MTP was the precursor for the Atomic Cluster Expansion (ACE) which is at the centre of this thesis. We therefore omit the derivation of $B(R)$ in favour of a detailed description of ACE in the next chapter.

Chapter 2

The Linear ACE Model

In this chapter we derive the Atomic Cluster Expansion (ACE) and describe its use to model the PES of an atomic configuration. We present a linear model based on the ACE and show results for benchmark data set for Silicon, Copper and Molybdenum.

2.1 The Atomic Cluster Expansion

Let us consider J particles described by their position vectors \mathbf{r}_j . A set $R := \{\mathbf{r}_1, \dots, \mathbf{r}_J\} \in \mathbb{R}^{3J}$ of J particle positions is called an atomic configuration (Figure 2.1). In practice in materials simulations, computational cells are endowed with periodic boundary conditions. During simulations we take this into account, but for the sake of simplicity we will ignore it in our derivation.

A PES is a mapping from the set of all configurations to a real number $E(R) \in \mathbb{R}$. In our current context, E is a configuration's total energy. This mapping is naturally permutation invariant since configurations are defined as sets, but we will further require it to be also invariant under isometries, that is,

$$E(\{Q\mathbf{r}_1, \dots, Q\mathbf{r}_J\}) = E(\{\mathbf{r}_1, \dots, \mathbf{r}_J\}) \quad \forall Q \in O(3).$$

Interatomic potential models represent E in terms of lower dimensional components [16]. Let $\mathbf{r}_{ij} = \mathbf{r}_j - \mathbf{r}_i$ be the distances between atom j and a central atom i , and let $R_i = \{\mathbf{r}_{ij}\}_{j \neq i}$ denote the atomic neighbourhood of atom i (Figure 2.1).

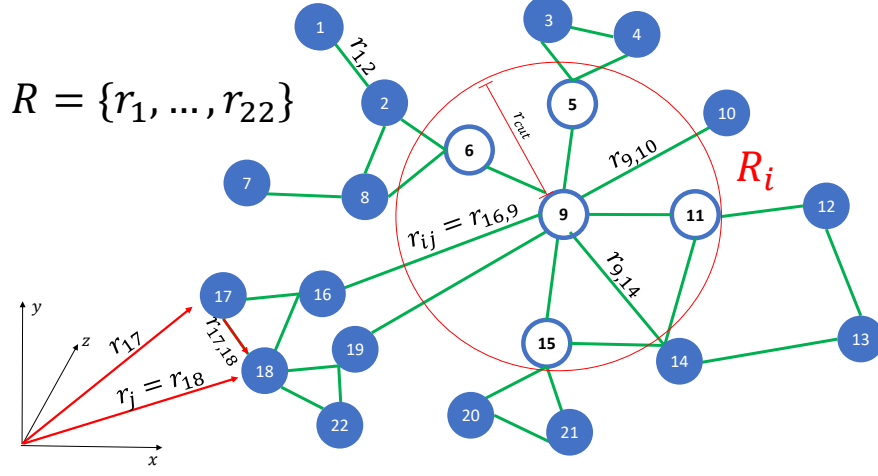


Figure 2.1: An example of a set R of $J = 22$ particles described by their position vectors \mathbf{r}_j , with the distances between them as r_{ij} . R_i is the atomic neighbourhood of i such that $|\mathbf{r}_{ij}| < r_{\text{cut}}$. In the figure $i = 9$ has neighbourhood of all hollow atoms.

Now let us assume that E can be represented in terms of the following body order expansion:

$$E(R) = \sum_{i=1}^J \mathcal{E}(R_i), \quad (2.1)$$

$$\mathcal{E}(R_i) = V_0 + \sum_{N=1}^{\mathcal{N}} \sum_{j_1 < j_2 < \dots < j_N} V_N(\mathbf{r}_{ij_1}, \dots, \mathbf{r}_{ij_N}),$$

where $V_0 \in \mathbb{R}$, $\mathcal{N} \in \mathbb{N}$ is the maximal order of interaction and $V_N : \mathbb{R}^{3J} \rightarrow \mathbb{R}$. These V_N are functions that map N \mathbf{r}_{ij} to a real number, and are called N -body functions. \mathcal{E} is called a site energy function, and it depends on its atomic environment R_i . We write the equation above for site energies more explicitly as

$$\begin{aligned} \mathcal{E}(R_i) \approx & V_0 + \sum_{j_1} V_1(\mathbf{r}_{ij_1}) + \sum_{j_1 < j_2} V_2(\mathbf{r}_{ij_1}, \mathbf{r}_{ij_2}) + \dots \\ & + \sum_{j_1 < \dots < j_N} V_N(\mathbf{r}_{ij_1}, \dots, \mathbf{r}_{ij_N}). \end{aligned} \quad (2.2)$$

In this form, we can see the role of N in the expansion. We are representing the site energy \mathcal{E} as a summation of increasing body order terms, i.e. terms that account for higher-order interactions. For example, V_1 is a pair potential that accounts for all pairwise interactions in the atomic environment, and similarly, V_N accounts for $N + 1$ particle interactions. The goal of this expansion is to truncate at $N \ll J$, which significantly reduces evaluation cost.

When we defined E , we required isometry invariance and noticed it already possessed permutation invariance. Therefore, we will assume that our components V_N are also isometry and permutation invariant. Moreover, we will further assume regularity and locality.

$$V_N \in C^t(\mathbb{R}^{3N} \setminus \{0\}), \quad \text{for some } t \geq 1, \quad (2.3)$$

$$\exists r_{\text{cut}} > 0 \quad \text{s.t. } V_N(\mathbf{r}_{ij_1}, \dots, \mathbf{r}_{ij_N}) = 0 \quad \text{if } \max_{1 \leq j < N} |\mathbf{r}_j| \geq r_{\text{cut}}.$$

where r_{cut} is a cutoff radius to limit the range of interaction (Figure 2.1). Similarly, we also restrict the domain by introducing a minimal radius $r_0 > 0$ since we are not interested in atomic collisions. We will include r_{cut} in \mathcal{E} such that $\mathcal{E}(R_i)$ takes $R_i = \{\mathbf{r}_{ij}\}_{j \neq i}$, but only considers $\{\mathbf{r}_{ij}\}_{r_{ij} < r_{\text{cut}}}$.

2.2 Tensor approximation

We now approximate V_N by using a tensor product basis consisting of a radial and a spherical function,

$$\phi_{\mathbf{n}\mathbf{l}\mathbf{m}}(\mathbf{r}_1, \dots, \mathbf{r}_N) := \prod_{\alpha=1}^N \phi_{n_\alpha l_\alpha m_\alpha}(\mathbf{r}_\alpha), \quad \phi_{nlm}(\mathbf{r}) := \mathcal{P}_n(r) Y_l^m(\hat{\mathbf{r}}) \quad (2.4)$$

where $\mathbf{n} = (n_1, \dots, n_N)$ and similarly for \mathbf{l} and \mathbf{m} , $n = 0, 1, 2, \dots$ dictates the

radial functions while $l = 0, 1, 2, \dots$ and $m = -l, \dots, l$ (the azimuthal and magnetic quantum numbers respectively) dictate the angular functions, which in our case are the spherical harmonics. We also denote $\hat{\mathbf{r}}$ as the unit vector of \mathbf{r} , r as it's magnitude, and $\hat{R} = (\hat{\mathbf{r}}_1, \dots, \hat{\mathbf{r}}_N)$. The choice of spherical harmonics will later allow us to conveniently impose rotational symmetries, but the choice of \mathcal{P}_n has considerable freedom. This allows us to play with different choices to improve convergence, but we will not pursue this freedom in the present work. Let the radial basis

$$\mathcal{P} := \{\mathcal{P}_n(r) | n = 1, 2, \dots\} \quad (2.5)$$

be a linearly independent subset of $\{f \in C^t([r_0, \infty)) | f = 0 \text{ in } [r_{\text{cut}}, \infty)\}$, where t is the parameter from (2.3). Moreover, we assume that any $f \in C^\infty$ with support in $[0, r_{\text{cut}}]$ can be approximated to within arbitrary accuracy from $\text{span } \mathcal{P}$, i.e.,

$$\overline{\text{span}}_{C^t} \mathcal{P} \supset \{f \in C^\infty([r_0, \infty)) | f = 0 \text{ in } [r_{\text{cut}}, \infty)\}, \quad (2.6)$$

where $\overline{\text{span}}_{C^t} \mathcal{P}$ denotes the closure of \mathcal{P} with respect to the norm $\|\cdot\|_{C^t}$. These two assumptions of the radial function mean that V_N can be approximated with a linear combination of the tensor product ϕ ,

$$\tilde{V}_N \approx \sum_{\mathbf{n}, \mathbf{l}, \mathbf{m}} c_{\mathbf{n} \mathbf{l} \mathbf{m}} \phi_{\mathbf{n} \mathbf{l} \mathbf{m}}. \quad (2.7)$$

It has been shown in [16] that since the $\phi_{\mathbf{n} \mathbf{l} \mathbf{m}}$ are linearly independent and \tilde{V}_N is permutation invariant ($\tilde{V}_N = \tilde{V}_N \circ \sigma$), we can assume $c_{\mathbf{n} \mathbf{l} \mathbf{m}} = c_{\sigma \mathbf{n}, \sigma \mathbf{l}, \sigma \mathbf{m}}$ without loss of accuracy. This allows us to write

$$\tilde{V}_N \approx \sum_{(\mathbf{n}, \mathbf{l}, \mathbf{m}) \text{ ordered}} c_{\mathbf{n} \mathbf{l} \mathbf{m}} \sum_{\sigma \in S_N} \phi_{\mathbf{n} \mathbf{l} \mathbf{m}} \circ \sigma, \quad (2.8)$$

where $c_{\mathbf{n} \mathbf{l} \mathbf{m}}$ could be different coefficients, and " $(\mathbf{n}, \mathbf{l}, \mathbf{m})$ ordered" denotes the lexicographically ordered tuples. Since we assumed point reflection symmetry for V_N , all basis functions $\phi_{\mathbf{n} \mathbf{l} \mathbf{m}}$ for which $\sum \mathbf{l}$ is odd vanish. Hence

$$\tilde{V}_N(R) \approx \sum_{\substack{(\mathbf{n}, \mathbf{l}, \mathbf{m}) \text{ ordered} \\ \sum l \text{ even}}} c_{\mathbf{n} \mathbf{l} \mathbf{m}} \sum_{\sigma \in S_N} (\phi_{\mathbf{n} \mathbf{l} \mathbf{m}} \circ \sigma)(R). \quad (2.9)$$

To make 2.9 rotationally invariant, we integrate over all rotations using the Haar integral [16],

$$\tilde{V}_N \approx \sum_{\substack{(\mathbf{n}, \mathbf{l}, \mathbf{m}) \text{ ordered} \\ \sum l \text{ even}}} c_{\mathbf{n} \mathbf{l} \mathbf{m}} \sum_{\sigma \in S_N} \int_{SO(3)} (\phi_{\mathbf{n} \mathbf{l} \mathbf{m}} \circ \sigma)(QR) dQ. \quad (2.10)$$

Recall that the radial functions \mathcal{P} are already rotationally invariant, so we focus on Y_l^m . Now we represent the rotated spherical harmonics in terms of the Wigner D-matrices [16]

$$Y_l^m(Q\hat{R}) = \sum_{\mu \in \mathcal{M}_l} D_{\mu m}^l(Q) Y_l^\mu(\hat{R}) \quad \forall Q \in SO(3), \quad (2.11)$$

where $\mathcal{M}_l := \{\mu \in \mathbb{Z}^N \mid -l_\alpha \leq \mu_\alpha \leq l_\alpha, \text{ for } \alpha = 1, \dots, N\}$, and

$$D_{\mu m}^l(Q) = \prod_{\alpha=1}^N D_{\mu_\alpha m_\alpha}^{l_\alpha}(Q). \quad (2.12)$$

Integrating yields a spanning set $\{b_{lm}\}$, with

$$b_{lm}(\hat{R}) := \sum_{\mu \in \mathcal{M}_l} \bar{D}_{\mu m}^l Y_l^\mu(\hat{R}), \quad (2.13)$$

where

$$\bar{D}_{\mu m}^l(Q) = \int_{SO(3)} D_{\mu m}^l(Q) dQ. \quad (2.14)$$

The $\bar{D}_{\mu m}^l$ coefficients can be efficiently computed with a recursive formula involving Clebsch-Gordan coefficients [16]. Then we reduce this set to a basis by defining $\tilde{\mathcal{U}}_{\mu i}^l$ and $\tilde{n}_l := \text{rank } \bar{D}^l$,

$$b_{li}(\hat{R}) := \sum_{\mu \in \mathcal{M}_l} \tilde{\mathcal{U}}_{\mu i}^l Y_l^\mu(\hat{R}), \quad i = 1, \dots, \tilde{n}_l, \quad (2.15)$$

where we require the columns of $\tilde{\mathcal{U}}_{\mu i}^l$ to span the same space as the columns of $\bar{D}_{\mu m}^l$. Adding the radial component again, we define the rotational and permutation invariant basis

$$\tilde{B}_{nli}(R) := \sum_{\sigma \in S_N} \sum_{m \in \mathcal{M}_l^0} \tilde{\mathcal{U}}_{mi}^l (\phi_{nlm} \circ \sigma)(R), \quad i = 1, \dots, \tilde{n}_l. \quad (2.16)$$

However, these are not linearly independent. Therefore we define a new set of coefficients by diagonalizing the Gramian $G_{i,i'}^{nl} = \langle \tilde{B}_{nli}, \tilde{B}_{nli'} \rangle$, with respect to the abstract inner product $\langle \langle \phi_{nlm}, \phi_{n'l'm'} \rangle \rangle := \delta_{nn'} \delta_{ll'} \delta_{mm'}$.

$$\mathcal{U}_{mi}^{nl} := \frac{1}{\Sigma_{ii}} \sum_{\alpha=1}^{\tilde{n}_l} [\mathcal{V}_{\alpha i}]^* \tilde{\mathcal{U}}_{m\alpha}^l, \quad i = 1, \dots, n_{nl} \quad (2.17)$$

where $n_{nl} = \text{rank}(G^{nl})$ and we diagonalized $G^{nl} = \mathcal{V} \Sigma \mathcal{V}^T$. With this new coefficients we obtain

$$\mathcal{B}_{nli}(R) := \sum_{m \in \mathcal{M}_l} \mathcal{U}_{mi}^{nl} \sum_{\sigma \in S_N} (\phi_{nlm} \circ \sigma)(R). \quad (2.18)$$

So far, we have only treated a single correlation order N . We now move back to treating all atoms J by defining B_{nli} as

$$B_{nli}(\mathbf{r}_1, \dots, \mathbf{r}_J) := \sum_{j_1 < j_2 < \dots < j_N} \mathcal{B}_{nli}(\mathbf{r}_{j_1}, \dots, \mathbf{r}_{j_N}). \quad (2.19)$$

We currently have (2.19) scale as $\binom{J}{N}$, which is terribly inefficient. We will now leverage the tensor products to reduce the computational cost of our current basis. We start by completing the summation from $\sum_{j_1 < j_2 < \dots < j_N}$ to $\sum_{j_1, j_2, \dots, j_N}$ and use (2.18) to get

$$B_{nl i}(R) = \sum_{j_1, \dots, j_N} \sum_{\mathbf{m} \in \mathcal{M}_l} \mathcal{U}_{\mathbf{m} i}^{nl} \sum_{\sigma \in S_N} \phi_{nl \mathbf{m}}(\mathbf{r}_{j_{\sigma 1}}, \dots, \mathbf{r}_{j_{\sigma N}}) \quad (2.20)$$

$$= \sum_{\mathbf{m} \in \mathcal{M}_l} \mathcal{U}_{\mathbf{m} i}^{nl} \sum_{j_1, \dots, j_N} \phi_{nl \mathbf{m}}(\mathbf{r}_{j_1}, \dots, \mathbf{r}_{j_N}) \quad (2.21)$$

$$= \sum_{\mathbf{m} \in \mathcal{M}_l} \mathcal{U}_{\mathbf{m} i}^{nl} \sum_{j_1, \dots, j_N=1}^J \prod_{\alpha=1}^N \phi_{n_{\alpha} l_{\alpha} m_{\alpha}}(\mathbf{r}_{j_{\alpha}}). \quad (2.22)$$

Since we sum over tensor products, we may interchange the summations and the product in the following way

$$\dots = \sum_{\mathbf{m} \in \mathcal{M}_l} \mathcal{U}_{\mathbf{m} i}^{nl} \prod_{\alpha=1}^N \sum_{j=1}^J \phi_{n_{\alpha} l_{\alpha} m_{\alpha}}(\mathbf{r}_j). \quad (2.23)$$

We now define $A_{in_{\alpha} l_{\alpha} m_{\alpha}}$ as

$$B_{nl i}(R) =: \sum_{\mathbf{m} \in \mathcal{M}_l} \mathcal{U}_{\mathbf{m} i}^{nl} \prod_{\alpha=1}^N A_{in_{\alpha} l_{\alpha} m_{\alpha}}(R), \quad (2.24)$$

and a product basis $\mathbf{A}_{inl \mathbf{m}}(R) := \prod_{\alpha=1}^N A_{in_{\alpha} l_{\alpha} m_{\alpha}}(R)$. This avoids the $N!$ cost for symmetrising the basis as well as the $\binom{J}{N}$ cost of summation over all clusters in (2.19). We denote the resulting basis by

$$B_N := \left\{ B_{nl i} | (\mathbf{n}, \mathbf{l}) \in \mathbb{N}^{2N} \text{ ordered, } \sum \mathbf{l} \text{ even, } i = 1, \dots, n_{nl} \right\}, \quad (2.25)$$

and redefine the site energy as

$$\mathcal{E}(R_i) := \sum_{N=0}^{\mathcal{N}} B_N(R_i). \quad (2.26)$$

Finally, we choose a sub set of (2.25) by further restricting $(\mathbf{n}, \mathbf{l}, \mathbf{m})$. We define a maximal degree $d \in \mathbb{R}$. Then choose all n_{α} and l_{α} such that

$$\sum_{\alpha} (n_{\alpha} + w_L l_{\alpha}) \leq d, \quad (2.27)$$

where w_L is a relative weighting of the angular and radial basis functions. Higher w_L leads to higher resolution in the radial component. With this choice of $(\mathbf{n}, \mathbf{l}, \mathbf{m})$ we define a set of basis functions $B \in \mathbf{B}$ and their coefficients c_B as

$$\mathcal{E}(R_i) =: \sum_{B \in \mathbf{B}} c_B B(R_i), \quad (2.28)$$

and the resulting total energy as

$$E(R) = \sum_{i=1}^J \sum_{B \in \mathbf{B}} c_B B(R_i). \quad (2.29)$$

Let κ be the number of basis functions, i.e. the length of \mathbf{B} and the number of parameters c_B . We can then switch the summations and define a new basis function B , but now over whole configurations, resulting in the linear model

$$E(R) = \sum_{B \in \mathbf{B}} c_B B(R) = \mathbf{c} \cdot \mathbf{B}(R). \quad (2.30)$$

We can compute the forces, $F_i := -\frac{\partial E(R)}{\partial \mathbf{r}_i}$, using

$$\mathbf{F}(R) := -\left\{ \frac{\partial E(R)}{\partial \mathbf{r}_i} \right\}_{i=1}^J. \quad (2.31)$$

2.3 Parameter estimation

Similar to Section 1.4, our goal is to train parameters \mathbf{c} through a set of energies and forces calculated from a high fidelity model (DFT). Let us then consider a training set of T atomic configurations $R^{(t)}$, each with J_t atoms. We define the extended design matrix Ψ using equations (2.29) and (3.3) as follows:

$$\Psi \cdot \mathbf{c} := \begin{pmatrix} \alpha_E B_1(R^{(1)}) & \dots & \alpha_E B_\kappa(R^{(1)}) \\ -\alpha_F \frac{\partial B_1(R^{(1)})}{\partial \mathbf{r}_1} & \dots & -\alpha_F \frac{\partial B_\kappa(R^{(1)})}{\partial \mathbf{r}_1} \\ \vdots & \ddots & \vdots \\ -\alpha_F \frac{\partial B_1(R^{(1)})}{\partial \mathbf{r}_{J_1}} & \dots & -\alpha_F \frac{\partial B_\kappa(R^{(1)})}{\partial \mathbf{r}_{J_1}} \\ \alpha_E B_1(R^{(2)}) & \dots & \alpha_E B_\kappa(R^{(2)}) \\ \vdots & \ddots & \vdots \\ -\alpha_F \frac{\partial B_1(R^{(T)})}{\partial \mathbf{r}_{J_T}} & \dots & -\alpha_F \frac{\partial B_\kappa(R^{(T)})}{\partial \mathbf{r}_{J_T}} \end{pmatrix} \begin{pmatrix} c_1 \\ \vdots \\ c_\kappa \end{pmatrix} = \begin{pmatrix} \alpha_E E(R^{(1)}) \\ \alpha_F F(R^{(1)}) \\ \alpha_E E(R^{(2)}) \\ \vdots \\ \alpha_F F(R^{(T)}) \end{pmatrix}, \quad (2.32)$$

where the length of \mathbf{B} is κ , i.e the number of basis functions and parameters. We also define $\alpha_E, \alpha_F \in \mathbb{R}$ as weightings to multiply the energies and forces in Ψ . Now define a set of DFT calculations to act as targets in the training. For each training configuration $R^{(t)}$, let $\mathbf{y}^{(t)} = (y_E^{(t)}, y_F^{(t)})$ be the corresponding DFT calculated energy and forces. Then let $\mathbf{y} = [y_E^{(1)}, y_F^{(1)}, y_E^{(2)}, \dots, y_F^{(T)}]$. We seek parameters \mathbf{c} such that the loss

$$L(\mathbf{c}) := \|\Psi \mathbf{c} - \mathbf{y}\|^2, \quad (2.33)$$

is minimized. To avoid over-fitting to the training set, we multiply Ψ by the diagonal of a matrix Λ that estimates the scaling of $\nabla^2 \phi_{nlm}$. From 2.24 we can create a matrix Λ such that $\Lambda_{nl} = n^2 + l^2$ where n and l correspond to a specific $B \in \mathbf{B}$ through 2.25. Once Λ is calculated we update $\Psi = \Psi \text{diag}(\Lambda)$.

We solve this problem using rank revealing QR factorization (RRQR). Given a matrix Ψ , it can be shown that there exists a permutation Π and an integer k such that the QR factorization

$$\Psi \Pi = QR = Q \begin{pmatrix} R_{11} & R_{12} \\ & R_{22} \end{pmatrix} \quad (2.34)$$

has upper-triangular $k \times k$ matrix R_{11} , and R_{12} is linearly dependent on R_{11} [19]. We perform the factorization above using [23], and terminate when a certain tolerance ϵ is reached. Finally, we solve $QR_{11}\mathbf{c} = \mathbf{y}$.

To measure the performance of \mathbf{c} , we calculate the root mean squared error

(RMSE) on a test set for both energies and forces separately. For energies, we use

$$\text{RMSE}_E = \sqrt{\frac{\sum_{t=1}^T \frac{(\Psi_E^{(t)} \cdot \mathbf{c} - y_E^t)^2}{J_t^2}}{T}} \quad (2.35)$$

where $\Psi_E^{(t)}$ is the t^{th} row of a matrix Ψ_E of only the energies, $y_E^{(t)} \in \mathbf{y}_E$ are the elements of a vector with only the DFT energies, and T is the total number of energies. For forces we use

$$\text{RMSE}_F = \sqrt{\frac{\|\Psi_F \cdot \mathbf{c} - \mathbf{y}_F\|_2^2}{3\|\mathbf{J}\|_1}}, \quad (2.36)$$

where similarly, Ψ_F is a matrix containing only the forces of Ψ and \mathbf{y}_F are the forces in \mathbf{y} .

2.4 Results

To test the ACE model, we use the data sets from [37], which contains molecular dynamics, elastic, surface and vacancies structures for bulk silicon, copper and molybdenum (as well as other materials). The data set contains training sets of sizes 214, 262, and 194, and test sets of sizes 25, 31, and 23, of DFT energies and forces, for Si, Cu, and Mo, respectively. It is worth mentioning that these data sets are pretty small and should only be used as proof of concept. Practical data sets can be much larger (order of thousands of structures). It is also common to include virials in ACE fits in addition to energies and forces.

We used the codes supplied by `ACE1pack.jl` [33]. There are several parameters one can modify to achieve better results, but in this work, we focus on (i) the RRQR tolerance ϵ , (ii) the relative weighting of energies and forces α_E/α_F , and (iii) the basis size κ . Further parameters such as the cutoff radius are taken from [37]. We used `LowRankApprox.jl` [23] for the RRQR factorization, where we can manually set the tolerance on the error of the factorization.

Since the data set is small, and linear models are relatively fast we generate a heat map with ϵ and α_E/α_F as parameters. We set the correlation order to $\mathcal{N} = 3$ and choose a basis size of $\kappa = 964$ to keep computational cost low. With higher

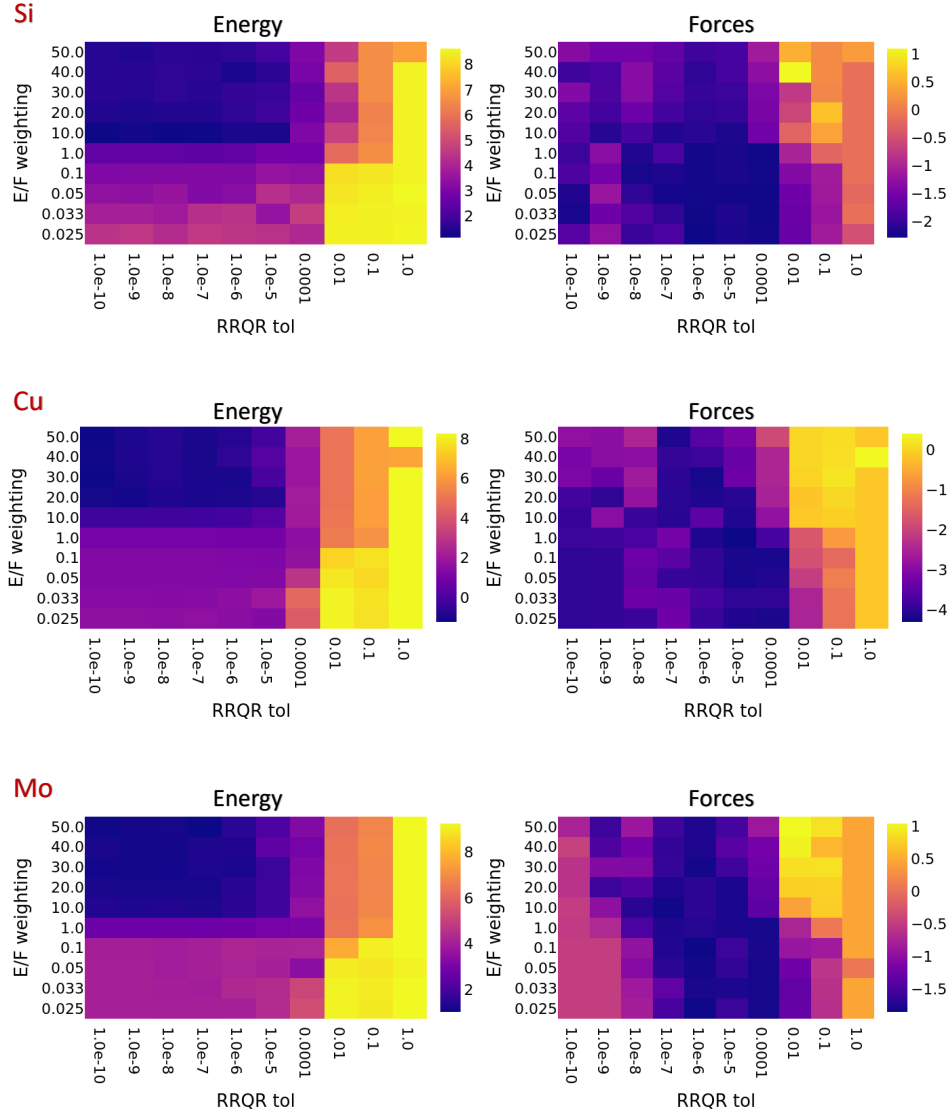


Figure 2.2: Log RMSE for varying ϵ and α_E/α_F .

Table 2.1: RMSE for energy and forces. κ represents the basis size used for that value.

RMSE	silicon	copper	molybdenum
energy (meV)	2.923 ($\kappa = 1706$)	0.320 ($\kappa = 3695$)	2.857 ($\kappa = 1291$)
forces (meV/at)	0.111 ($\kappa = 1706$)	0.015 ($\kappa = 3695$)	0.182 ($\kappa = 1291$)

basis sizes we could have over fitting as well as more costly simulations. We used a cutoff radius of 5.5\AA for silicon, 4.1\AA for copper, and 5.2\AA for molybdenum. Figure 2.2 shows the log of the RMSE of the test set for energy and forces in a heat map against the parameters ϵ and α_E/α_F . We see in Figure 2.2 that both low ϵ and low α_E/α_F , as well as high ϵ and high α_E/α_F , give higher RMSE, likely due to over-fitting. Therefore, we chose parameters closer to the middle. ($\epsilon = 10^{-7}, \alpha_E/\alpha_F = 10$), ($\epsilon = 10^{-6}, \alpha_E/\alpha_F = 30$), and ($\epsilon = 10^{-7}, \alpha_E/\alpha_F = 10$) where visually chosen for silicon, copper and molybdenum respectively.

Using these parameters, we calculated the RMSE of energy and forces for correlation order $\mathcal{N} = 3$ and increasing polynomial degrees, meaning increasing basis sizes. Figure 2.3 shows the log scaled RMSE of the test set against the log scaled basis size. The lowest RMSE for energy and forces gave different κ . Therefore, we chose the κ visually from Figure 2.3 to give the both low RMSE for energies and for forces, this results can be found in Table 2.1.

We can see in Figure 2.3 that for basis sizes below 1000, a larger basis size leads to better accuracy, which we expect since increasing basis size increases the number of parameters. However, for κ bigger than 1000, the RMSE for forces increases again, likely due to over-fitting. Since we report results on the test data set, increasing parameters without adjusting regularization overfits to the training set, hence dropping performance in the test set. We likely only see this in forces and not in energy since Ψ contains significantly more entries for forces which indicates we could use a bigger α_E/α_F for larger κ .

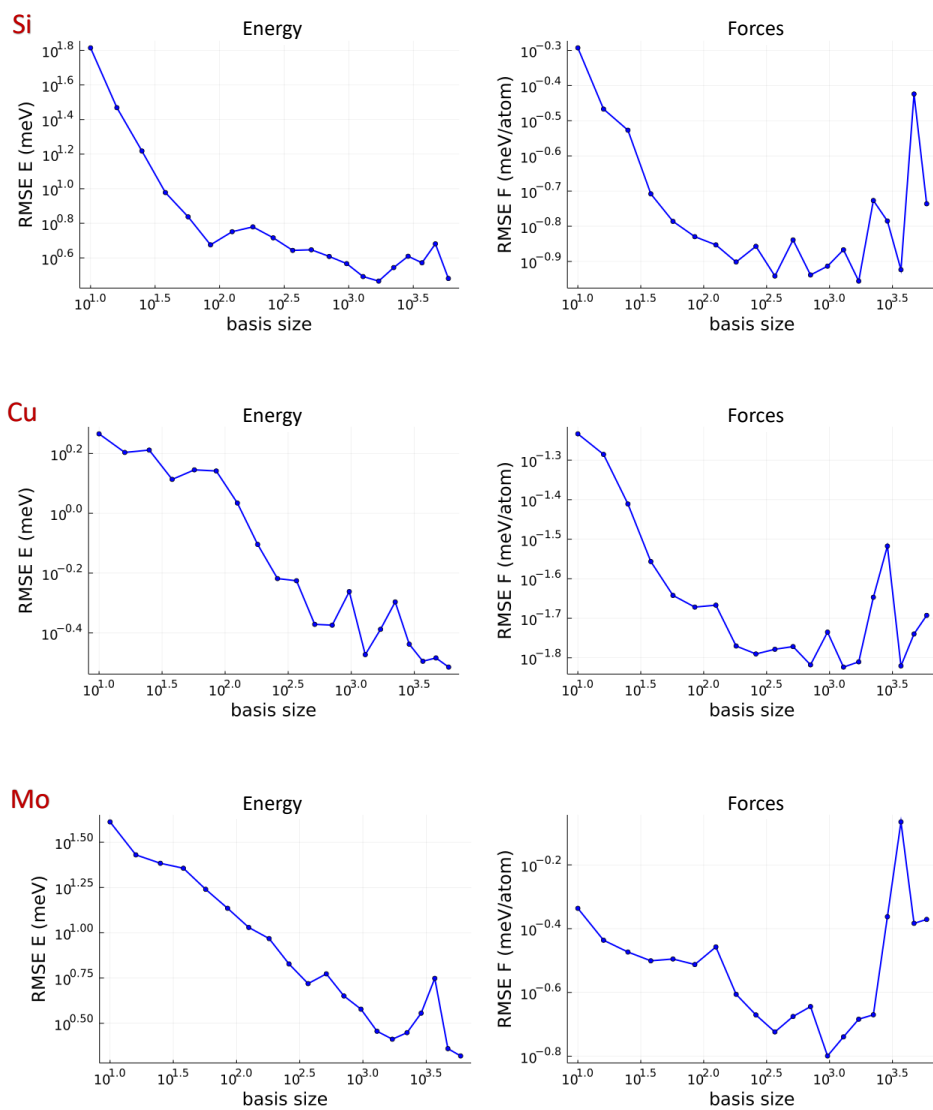


Figure 2.3: Log RMSE v.s log basis size for energy and forces.

Chapter 3

Nonlinear ACE Model

In this chapter we extend the models seen in Chapter 2 to allow for a nonlinear composition of linear models. We describe the model as well as the efficient computation of its derivatives. We present results based on the same data sets studied in Chapter 2.

3.1 Model Definition

The atomic cluster expansion (ACE) we introduced in Chapter 2 can be used to model any invariant atomic property φ . In the current chapter we will extend this model to allow for a nonlinear combination of such properties. Furthermore, we will present a method to efficiently evaluate the gradients of the nonlinear model and show initial regression results that serve as a proof of concept.

Consider (2.29), and define an atomic property centred at atom i in terms of an ACE basis B as

$$\varphi_i := \sum_{B \in \mathcal{B}} c_B B(R_i) = \mathbf{c} \cdot \mathbf{B}(R_i), \quad (3.1)$$

Then we define the energy of a nonlinear model as

$$E(R) = \sum_{i=1}^J \mathcal{E}(R_i) := \sum_{i=1}^J \mathcal{F}(\varphi_i^{(1)}, \varphi_i^{(2)}, \dots, \varphi_i^{(P)}), \quad (3.2)$$

where $p \in \{1, \dots, P\}$ indexes the atomic properties, and $\mathcal{F} : \mathbb{R}^P \rightarrow \mathbb{R}$ is a nonlinear embedding function. The forces would be

$$F(R) := - \left\{ \frac{\partial E(R)}{\partial \mathbf{r}_i} \right\}_{i=1}^J, \quad (3.3)$$

but now there is not a simple formula like (3.3).

This type of nonlinear model is called a nonlinear combination of properties and was proposed in [15]. Models of this type will have κP parameters, where we assume all $\varphi_i^{(p)}$ have basis length κ . There are other ways one can incorporate physics-inspired nonlinearities but this thesis will focus on (3.2) as a proof of concept. We briefly visit other nonlinear models in Chapter 5.

Some examples of the embedding \mathcal{F} include

$$\mathcal{F}(\varphi_i^{(1)}, \varphi_i^{(2)}) = \varphi_i^{(1)} - \sqrt{\varphi_i^{(2)}} \quad (3.4)$$

inspired by the Finnis-Sinclair [14] and embedded atom models, or its generalization

$$\mathcal{F}(\varphi_i^{(1)}, \dots, \varphi_i^{(P)}) = \sum_{p=1}^P |\varphi_i^{(p)}|^{\alpha_p}. \quad (3.5)$$

where $\{\alpha\}_{p=1}^P$ is a set of exponents. Finally, one could consider a neural network parametrization of \mathcal{F} , e.g.,

$$\mathcal{F}(\varphi_i^{(1)}, \dots, \varphi_i^{(P)}) = W_3 \left[W_2 \left[W_1 \boldsymbol{\varphi}_i^T + b_1 \right] + b_2 \right] + b_3 \quad (3.6)$$

a feed forward neural network with 3 dense layers. In (3.6) $\boldsymbol{\varphi}_i^T = [\varphi_i^{(1)}, \dots, \varphi_i^{(P)}]^T$, while $\mathbf{W} = [W_1, W_2, W_3]$ and $\mathbf{b} = [b_1, b_2, b_3]$ are the weights and biases of the dense layers respectively. Notice that in (3.6) and (3.5) \mathcal{F} has trainable parameters. We denote the trainable parameters of a nonlinear model by $\theta = [\theta_{\mathcal{F}}, \theta_{\varphi_i}]$, where \mathcal{F} has parameters $\theta_{\mathcal{F}}$ and φ_i has parameters $\theta_{\varphi} := [\mathbf{c}^{(1)}, \dots, \mathbf{c}^{(P)}]$ for all i .

3.1.1 Loss function

As in Section 2.3, we define a training set $\mathbf{R} = [R^{(1)}, \dots, R^{(T)}]$, with J_t the number of atoms in $R^{(t)}$, and its respective DFT energies and forces $Y = [y^{(1)}, \dots, y^{(T)}]$, with $y^{(t)} = (y_E^{(t)}, y_F^{(t)})$. Recall that $y_E^{(t)} \in \mathbb{R}$ and $y_F^{(t)} \in \mathbb{R}^{3 \times J}$. We then define a loss function to minimize as follows

$$\mathcal{L}(\mathbf{R}, Y) = \frac{\sum_{t=1}^T L(R^{(t)}, y^{(t)})}{T} + \lambda \|\theta\|_2^2, \quad (3.7)$$

$$L(R, y) = \alpha_E^2 (E(R) - y_E)^2 + \alpha_F^2 \sum_{i=1}^{J_t} |F_i - y_F|^2 =: L_E(R, y_E) + L_F(R, y_F), \quad (3.8)$$

where $R = \{\mathbf{r}_1, \dots, \mathbf{r}_J\}$, λ , α_E and α_F are hyperparameters. We define the loss as an explicit function of the training set (\mathbf{R}, Y) with implicit dependence on the trainable parameters θ . Therefore, we seek parameters θ such that

$$\min_{\theta} \mathcal{L}(\mathbf{R}, Y) \quad (3.9)$$

To solve (3.9) we require the gradient

$$\frac{\partial \mathcal{L}(\mathbf{R}, Y)}{\partial \theta} = \frac{\sum_{t=1}^T \frac{\partial L(R^{(t)}, y^{(t)})}{\partial \theta}}{T} + 2\lambda\theta, \quad (3.10)$$

where

$$\frac{\partial L(R, y)}{\partial \theta} = 2\alpha_E^2 (E(R) - y_E) \frac{\partial E(R)}{\partial \theta} + 2\alpha_F^2 \sum_{i=1}^{J_t} |F_i - y_F| \frac{\partial^2 E(R)}{\partial \mathbf{r}_i \partial \theta} \quad (3.11)$$

where we used (3.3). A naive evaluation of the gradients is very costly due to the computation of $\frac{\partial^2 E(R)}{\partial \mathbf{r}_i \partial \theta}$. We will review an efficient way to evaluate it in Section 3.2 via backpropagation.

Similar to Section 2.3, we evaluate the performance of the model by measuring the RMSE of the energy and the forces for a test set. For the energy we measure

$$\text{RMSE}_E = \sqrt{\frac{\sum_{t=1}^T \frac{(E(R^{(t)}) - y_E)^2}{J_t^2}}{T}}, \quad (3.12)$$

and for the forces

$$\text{RMSE}_F = \sqrt{\sum_{t=1}^T \frac{\sum_{i=1}^{J_t} |F_i - y_F|^2}{3J_t}}. \quad (3.13)$$

3.2 Efficient computation

In this section we will demonstrate how to efficiently compute the derivative of the forces with respect to parameters, but we will postpone its Julia implementation to Chapter 4. Let us start with (3.11) and consider only the second term, L_F , representing the fit accuracy on the forces:

$$\begin{aligned} \frac{\partial L_F(R, y)}{\partial \theta} &= 2\alpha_F \sum_{i=1}^J |F_i - y_F| \frac{\partial^2 \mathcal{F}(\varphi_i)}{\partial \mathbf{r}_{ij} \partial \theta} \\ &= \sum_{i=1}^J 2\alpha_F |F_i - y_F| \frac{\partial^2 \mathcal{F}}{\partial \varphi_i \partial \theta_F} \frac{\partial^2 \varphi_i}{\partial \mathbf{r}_{ij} \partial \theta_\varphi} \\ &=: \sum_{i=1}^J \omega_i \frac{\partial^2 \varphi_i}{\partial \mathbf{r}_{ij} \partial \theta_\varphi}, \end{aligned} \quad (3.14)$$

where we defined ω_i implicitly. Then using (2.28) and (2.24) we can compute

$$\frac{\partial \varphi_i^{(p)}}{\partial \mathbf{r}_{ij}} = \sum_{N=0}^{\mathcal{N}} \sum_{\mathbf{v}} \tilde{c}_{\mathbf{v}}^{(p)} \frac{\partial \mathbf{A}_{i\mathbf{v}}}{\partial \mathbf{r}_{ij}} \quad (3.15)$$

with \mathbf{v} representing the summation over the corresponding (nlm) , as described by (2.25). Then $\tilde{c}_{\mathbf{v}}^{(p)}$ are the corresponding parameters \mathcal{U}_{mi}^{nl} from (2.24) for a property p . Defining $N_{\mathbf{v}} := \{N\}_{N=0}^{\mathcal{N}}$, expanding the product basis, and using the product rule we obtain

$$\begin{aligned}
\frac{\partial \varphi_i^{(p)}}{\partial \mathbf{r}_{ij}} &= \sum_{\mathbf{v}} \tilde{c}_{\mathbf{v}}^{(p)} \frac{\partial}{\partial \mathbf{r}_{ij}} \left(\prod_{\alpha=1}^{N_{\mathbf{v}}} \sum_{s=1}^J \phi_{v_s}(\mathbf{r}_{ij}) \right) \\
&= \sum_{\mathbf{v}} \tilde{c}_{\mathbf{v}}^{(p)} \sum_{\alpha=1}^{N_{\mathbf{v}}} \prod_{\alpha \neq s} A_{iv_s} \nabla \phi_{v_t}(\mathbf{r}_{ij}).
\end{aligned} \tag{3.16}$$

Expression (3.16) has cost equal to $\#\tilde{c} \times \mathcal{N}^2 \times J \times P$. We have our first cost reduction by switching the order of the expression above.

$$\begin{aligned}
\dots &= \sum_{\mathbf{v}} \nabla \phi_{\mathbf{v}}(\mathbf{r}_{ij}) \left[\sum_{\mathbf{v}} \tilde{c}_{\mathbf{v}}^{(p)} \sum_{\alpha=1}^{N_{\mathbf{v}}} \delta_{vv_t} \prod_{\alpha \neq s} A_{iv_s} \right] \\
&=: \sum_{\mathbf{v}} \nabla \phi_{\mathbf{v}}(\mathbf{r}_{ij}) \cdot \omega_{\mathbf{v}}^{\phi},
\end{aligned} \tag{3.17}$$

where we implicitly defined $\omega_{\mathbf{v}}^{\phi}$. The cost is now $\#\tilde{c} \times \mathcal{N}^2 \times P + \#v \times J$. Now using (3.14) and (3.19)

$$\begin{aligned}
\frac{\partial L_F(R, y)}{\partial \theta} &= \sum_{i=1}^J \omega_i \sum_{\mathbf{v}} \nabla \phi_{\mathbf{v}}(\mathbf{r}_{ij}) \frac{\partial \omega_{\mathbf{v}}^{\phi}}{\partial \theta_{\boldsymbol{\varphi}}} \\
&= \sum_{\mathbf{v}} \left[\sum_{i=1}^J \omega_i \cdot \nabla \phi_{\mathbf{v}}(\mathbf{r}_{ij}) \right] \frac{\partial \omega_{\mathbf{v}}^{\phi}}{\partial [\mathbf{c}^{(1)}, \dots, \mathbf{c}^{(P)}]} \\
&=: \sum_{\mathbf{v}} \omega'_{\mathbf{v}} \frac{\partial \omega_{\mathbf{v}}^{\phi}}{\partial [\mathbf{c}^{(1)}, \dots, \mathbf{c}^{(P)}]},
\end{aligned} \tag{3.18}$$

where we defined $\omega'_{\mathbf{v}}$ to hold all the i dependence. The cost of evaluating $\omega'_{\mathbf{v}}$ over all v is $J \times \#v$. Then, using (3.19)

$$\begin{aligned}
\dots &= \sum_v \omega'_v \frac{\partial}{\partial[\mathbf{c}^{(1)}, \dots, \mathbf{c}^{(P)}]} \left[\sum_{\mathbf{v}} \tilde{c}_{\mathbf{v}}^{(p)} \sum_{\alpha=1}^{N_v} \delta_{vv_t} \prod_{\alpha \neq s} A_{iv_s} \right] \\
&= \frac{\partial}{\partial[\mathbf{c}^{(1)}, \dots, \mathbf{c}^{(P)}]} \sum_{\mathbf{v}} \tilde{c}_{\mathbf{v}}^{(p)} \sum_{\alpha=1}^{N_v} \sum_v \delta_{vv_t} \omega'_v \prod_{\alpha \neq s} A_{iv_s} \\
&= \frac{\partial}{\partial[\mathbf{c}^{(1)}, \dots, \mathbf{c}^{(P)}]} \sum_{\mathbf{v}} \tilde{c}_{\mathbf{v}}^{(p)} \left[\sum_{\alpha=1}^{N_v} \omega'_{v_\alpha} \prod_{\alpha \neq s} A_{iv_s} \right] \\
&=: \frac{\partial}{\partial[\mathbf{c}^{(1)}, \dots, \mathbf{c}^{(P)}]} \sum_{\mathbf{v}} \tilde{c}_{\mathbf{v}}^{(p)} \mathbf{A}'_{\mathbf{v}},
\end{aligned} \tag{3.19}$$

where we defined $\mathbf{A}'_{\mathbf{v}}$. Now using vector notation over all \mathbf{v} ,

$$=: \frac{\partial}{\partial[\mathbf{c}^{(1)}, \dots, \mathbf{c}^{(P)}]} \tilde{\mathbf{c}}^{(p)} \cdot \mathbf{A}', \tag{3.20}$$

and using (2.24)

$$= \frac{\partial}{\partial[\mathbf{c}^{(1)}, \dots, \mathbf{c}^{(P)}]} \mathbf{c}^{(p)} \cdot \mathcal{U} \mathbf{A}'. \tag{3.21}$$

The cost of computation of $\mathbf{A}'_{\mathbf{v}}$ is $\#\tilde{\mathbf{c}} \times \mathcal{N}^2$ for all $\mathbf{A}' = \{\mathbf{A}'_{\mathbf{v}}\}_{\mathbf{v}}$, but it can be further reduced to \mathcal{N} [26]. The final cost of the expression is $\text{cost}(\mathcal{U} \times \mathbf{A}') + \#\tilde{\mathbf{c}} \times \mathcal{N}^2 \times P + J \times \#v$. The cost of \mathcal{U} can be reduced further through symmetries which make it a sparse matrix [16].

3.3 Results

Using the techniques mentioned in this chapter we minimized equation (3.7) using two different embeddings \mathcal{F} , (i) a Finnis-Sinclair inspired model from [26], and (ii) three dense layers (equation 3.6). For both of them we used 2 properties with basis size of $\kappa = 489$, which gave a $\theta_{\varphi} \in \mathbb{R}^{978 \times 2}$.

(i) For the Finnis-Sinclair embedding we use a function very closely inspired by what was used in the copper model in [26]. We defined \mathcal{F} as:

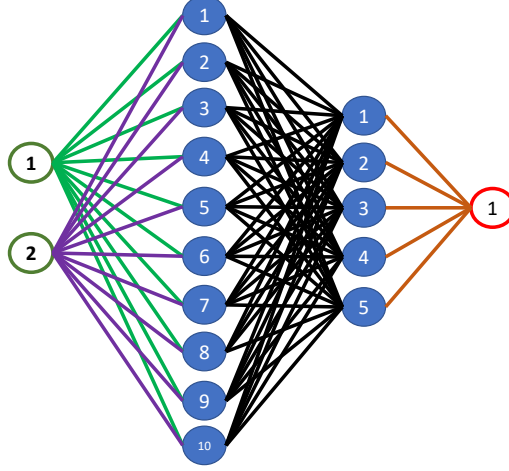


Figure 3.1: (ii) embedding, 3 dense layers.

$$\mathcal{F}(\varphi_i^{(1)}, \varphi_i^{(2)}) = \varphi_i^{(1)} + \text{sign}(\varphi_i^{(2)}) \left[\sqrt{|\varphi_i^{(2)}| + \frac{e^{-|\varphi_i^{(2)}|}}{4}} - \frac{e^{-|\varphi_i^{(2)}|}}{2} \right]. \quad (3.22)$$

(ii) For the dense layers we used the architecture in figure 3.1 and (3.6). In our case $W_1 \in \mathbb{R}^{2 \times 10}$, $W_2 \in \mathbb{R}^{10 \times 5}$, $W_3 \in \mathbb{R}^{5 \times 1}$, $b_1 \in \mathbb{R}^1$, $b_2 \in \mathbb{R}^5$, and $b_3 \in \mathbb{R}^1$. This gives $\theta_{\mathcal{F}} = [W_1, b_1, W_2, b_2, W_3, b_3]$.

We solve (3.9) using the same data sets and cutoff radius as we did for the linear model, and correlation order $\mathcal{N} = 3$. We used BFGS [31] on (3.7) for 500 iterations. Usually, models would be ran for longer iterations, but we choose 500 as a proof of concept. We empirically chose $\alpha_E/\alpha_F = 1/10$ for all models and $\lambda = \{10^{-7}, 10^{-6}, 10^{-7}\}$ for Si, Co, and Mo respectively.

We present 6 plots in figure 3.2, where we plot the log of the RMSE for energy and forces for both embeddings. We compare them against the best linear RMSE in table 2.1. For all materials we beat the best linear forces, but we don't beat the energies. There is similar performance for the two embeddings, except for the forces of copper where we see (ii) converge faster, and for the forces of molybdenum, where (ii) retains a low RMSE while (i) seems to over-fit.

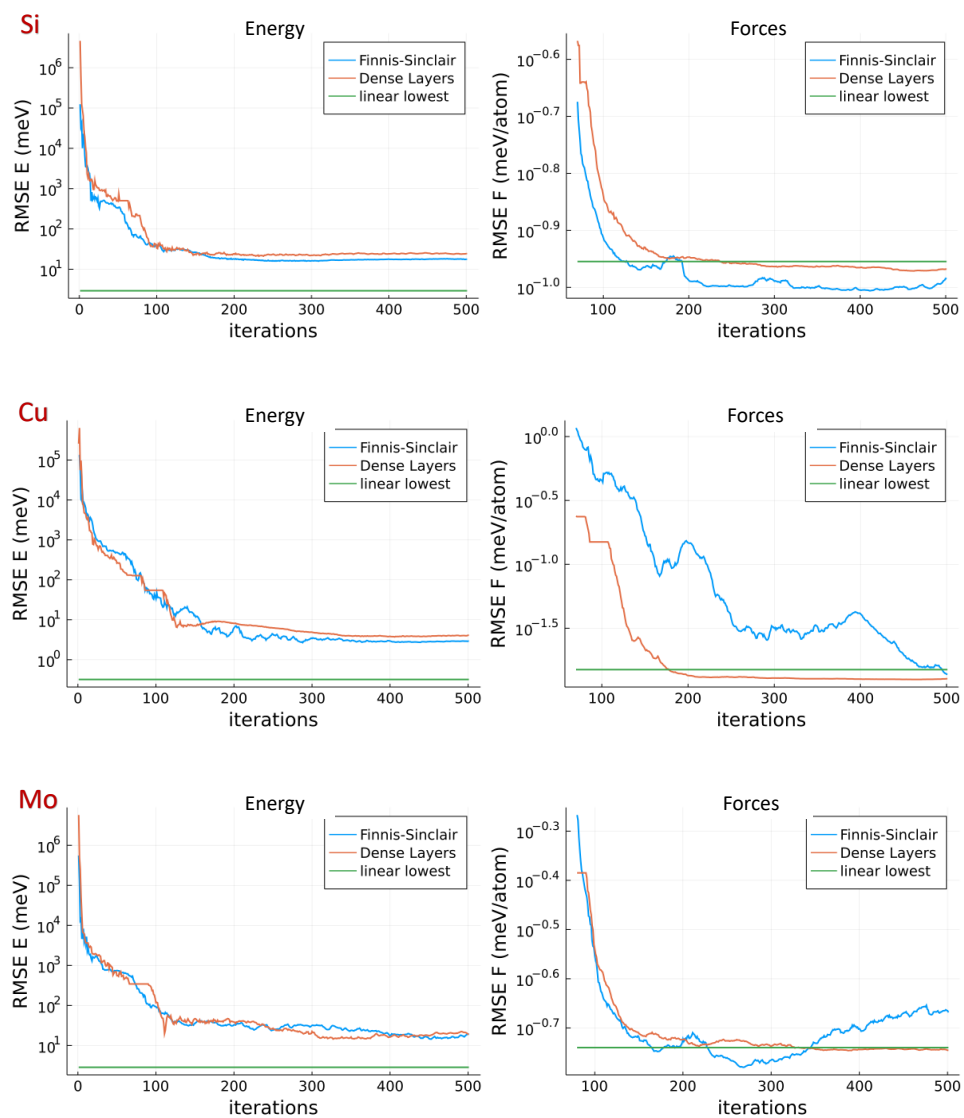


Figure 3.2: Log RMSE v.s iterations for energy and forces.

These results should serve as a proof of concept. However, more work is needed for our current implementation to be competitive.

Chapter 4

Implementation

In this chapter we go over the implementation of the nonlinear models described in Chapter 3. The goal is to overview the challenges involved in implementing such models in Julia [5]. We do so for two main reasons, (1) as documentation for further development and (2) as a starting point for a Journal of Open Source Software paper.

4.1 List of neighbours

Recall that when calculating the energy and forces of an ACE model, we only consider the atoms that are a distance $r_{ij} < r_{\text{cut}}$, i.e. $\{\mathbf{r}_{ij}\}_{r_{ij} < r_{\text{cut}}}$. In Chapters 2 and 3 we incorporated this cutoff into \mathcal{E} , but for implementation it is necessary to define a separate function

$$f_i(R) = (\mathbf{r}_{ij})_{r_{ij} < r_{\text{cut}}}. \quad (4.1)$$

$f(R_i)$ works by finding a list of neighbours of atom \mathbf{r}_i . Figure 4.1 shows an example of $f(R_i)$ for two different central atoms i with $J = 14$ and cutoff radius r_{cut} . We can then define $f(R) = [f_1(R), \dots, f_J(R)]$, which returns the neighbour lists for all atoms. With this new definition, we implement

$$E(R) = \sum_{i=1}^J \mathcal{F}(\varphi(f_i(R))), \quad (4.2)$$

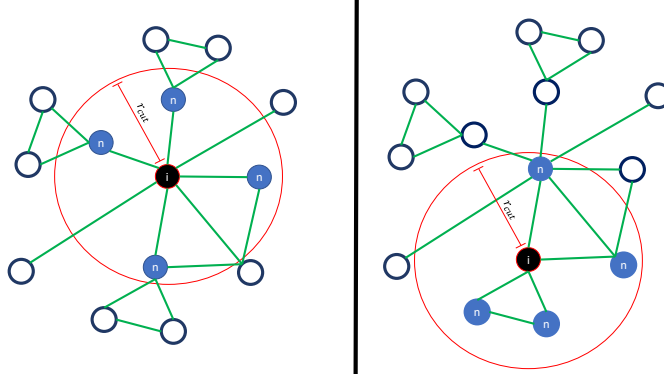


Figure 4.1: Example situation of f_i for 14 atoms. The left and the right represent $f_i(R)$ for two different atoms i . The number of atoms n inside the cutoff radius can (and usually is) of different length for different i .

with $\varphi(\{\mathbf{r}_{ij}\}_{r_{ij} < r_{\text{cut}}}) = [\varphi_i^{(1)}, \dots, \varphi_i^{(P)}]$.

4.2 Reverse mode differentiation: Example in ACE

To implement the derivatives of the models described, we use automatic differentiation. Specifically, we use reverse mode differentiation, starting with the outermost function rather than the innermost (in contrast to forward mode). As an example, consider (4.2) and take it's derivative according to \mathbf{r}_{ij} for some fixed i

$$\frac{\partial E(R)}{\partial \mathbf{r}_{ij}} = \frac{\partial E(R)}{\partial \mathcal{F}} \frac{\partial \mathcal{F}}{\partial \varphi} \frac{\partial \varphi}{\partial f_i} \frac{\partial f_i(R)}{\partial \mathbf{r}_{ij}}. \quad (4.3)$$

First, we define a Jacobian vector product (*JVP*) as the projection of a given vector \mathbf{g} into the Jacobian matrix of an operator h according to θ [1]

$$\omega_{\theta}(\mathbf{g}, h) := \mathbf{g} \frac{\partial h}{\partial \theta}. \quad (4.4)$$

Forward mode differentiation defines a *JVP* for each function to differentiate. We also call *JVP* push-forwards to be consistent with naming convention in the Julia package `ChainRules.jl`. We can represent the computation of (4.3) in the following order

$$w_{f_i} := \frac{\partial f_i(R)}{\partial \mathbf{r}_{ij}} \quad (4.5)$$

$$w_{\varphi} := \frac{\partial \varphi}{\partial f_i} w_{f_i} \quad (4.6)$$

$$w_{\mathcal{F}} := \frac{\partial \mathcal{F}}{\partial \varphi} w_{\varphi} \quad (4.7)$$

$$w_E := \frac{\partial E(R)}{\partial \mathcal{F}} w_{\mathcal{F}} \quad (4.8)$$

$$\frac{\partial E(R)}{\partial \mathbf{r}_{ij}} := 1 w_E \quad (4.9)$$

where all w 's are placeholders for numerical values, not symbolic expressions. Then we can represent the same computation as the composition of push-forwards,

$$\frac{\partial E(R)}{\partial \mathbf{r}_{ij}} = \omega_{\mathbf{r}_{ij}}(\omega_{f_i}(\omega_{\varphi}(\omega_{\mathcal{F}}(1, E), \mathcal{F}), \varphi), f_i). \quad (4.10)$$

For reverse mode differentiation we further define functions

$$\omega_{\theta}^T(g, h) := \left(\frac{\partial h}{\partial \theta} \right)^T g \quad (4.11)$$

to carry out the Jacobian transpose vector product (J^TVP), and call them pullbacks. The naming convention comes from the Julia package `ChainRules.jl` and has a definition broadly in agreement with their use in differential geometry [36]. We will continue to use the word "pullback" throughout this chapter, and also its `ChainRules.jl` functional implementation name "rule". Let's look at the same example, but now with reverse mode differentiation. We compute the derivative in a "reverse" order

$$w_E(1) := \frac{\partial E}{\partial E} 1 \quad (4.12)$$

$$w_{\mathcal{F}}(w_E) := \frac{\partial E}{\partial \mathcal{F}} w_E \quad (4.13)$$

$$w_{\varphi}(w_{\mathcal{F}}) := \frac{\partial \mathcal{F}}{\partial \varphi} w_{\mathcal{F}} \quad (4.14)$$

$$w_{f_i}(w_{\varphi}) := \frac{\partial \varphi}{\partial f_i} w_{\varphi} \quad (4.15)$$

$$\frac{\partial E(R)}{\partial \mathbf{r}_{ij}} = w_E(w_{\mathcal{F}}(w_{\varphi}(w_{f_i}(\frac{\partial f_i}{\partial \mathbf{r}_{ij}})))) \quad (4.16)$$

where the w 's are now stored as functions. These functions are already the pullbacks (or rrules), so in the notation of 4.11

$$\frac{\partial E(R)}{\partial \mathbf{r}_{ij}} = \omega_{\mathcal{F}}^T(\omega_{\varphi}^T(\omega_{f_i}^T(\omega_{\mathbf{r}_{ij}}^T(1, f_i), \varphi), \mathcal{F}), E). \quad (4.17)$$

In our implementation we use reverse mode differentiation, and in Section 4.4.3 we will see how the pullbacks were implemented. We used reverse mode differentiation to allow for the computation shown in Section 3.2. As an example, let us consider (3.14). We start with

$$\frac{\partial L_F(R, y)}{\partial \theta} = \frac{\partial L_F}{\partial F} \cdot \frac{\partial F}{\partial \varphi} \odot \frac{\partial^2 \varphi}{\partial \mathbf{r}_{ij} \partial \theta_{\varphi}}, \quad (4.18)$$

where \odot is the Hadamard product, and we use contraction over the gradients of vectors F and φ instead of the summations. Then we define $\omega_{\theta_{\varphi}}^T$ (the pullback of φ) as

$$\frac{\partial L_F(R, y)}{\partial \theta} = \omega_{\theta_{\varphi}}^T(\frac{\partial L_F(R, y)}{\partial F} \cdot \frac{\partial F}{\partial \varphi}, \frac{\partial \varphi}{\partial \mathbf{r}_{ij}}), \quad (4.19)$$

where $\omega_{\theta_{\varphi}}^T$ can be thought of as ω_i in (3.14).

4.3 Introduction to the code base

We first introduce the most relevant repositories and their functionality. Most of these can be found under the Github organization ACESuit [10].

- **ACE1pack.jl**: Convenience functionality for fitting inter atomic potentials using ACE [33].
 - Used in Section 2.4 to create linear fits.
- **ACE.jl**: General approximation schemes for permutation and isometry equivariant functions [12].
 - The backbone of the codes, everything needed to calculate $(\varphi_i^{(1)}, \dots, \varphi_i^{(P)})$ given a configuration $\{\mathbf{r}_{ij}\}_{j \neq i}$ is here.
- **ACEatoms.jl**: Generic atomistic modelling related extensions of ACE.jl [9].
 - Utility functions to calculate the energy and forces, including f .
 - This package takes R and calls functions in ACE.jl with $\{\mathbf{r}_{ij}\}_{r_{ij} < r_{\text{cut}}}$.
 - Then given \mathcal{E} and it's gradients it computes E and F .
- **JuLIP.jl**: Rapid implementation and testing of new interatomic potentials and molecular simulation algorithms [11].
 - Very similar to ACEatoms.jl.
 - This package handles everything relating to atomic neighbours.
 - Handles the computation of the total energy and forces of an atomic environment given the values for each site.
- **ChainRules.jl**: Common utilities that can be used by downstream automatic differentiation (AD) [36].
 - Provides "rules" which are custom pullback functions for each function we want to differentiate.

- We coded the optimized differentiation (Section 3.2) by creating custom pullbacks ω within this package.
- **Zygote.jl**: Source-to-source automatic differentiation (AD) in Julia [27].
 - Performs the actual derivatives for the nonlinear models.
 - Given a loss function, it differentiates it using reverse mode AD.
 - Calls all of our custom defined pullbacks in `ChainRules.jl` when needed.
- **Flux.jl**: Julia machine learning package [29].
 - Using `Zygote.jl`, it differentiates and trains models.
 - Allows you to customize layers, as well as providing several machine learning utilities, for example Dense layers.
 - Allowed us to implement (3.6).
- **ACEflux.jl**: Interface between `ACE.jl` and `Flux.jl` [8].
 - Using most of the previous packages it provides a framework to generate nonlinear ACE models.
 - Generates \mathcal{E} as a `Flux.jl` layer.
 - A `ACEflux.jl` layer takes an R , calls `ACE.jl` to generate ψ_i , then uses `Flux.jl` to generate \mathcal{F} .
 - It overloads `Julip.jl`'s E and F functions to accept `ACEflux.jl` models as input.
 - Ensures E and F are fully differentiable with respect to parameters.
- **Optim.jl**: Univariate and multivariate optimization in Julia [34].
 - Provides the BFGS optimizer used in the nonlinear fits.

4.4 Implementing nonlinear combinations

`ACEflux.jl` is a wrapper around `ACE.jl` and `JuLIP.jl` to allow compatibility with `Flux.jl`. This bridge allows us to leverage `Flux.jl` layers to generate and compose nonlinear functions on an ACE model φ_i . However, as we will see in this section, there were several caveats and issues when bringing all the packages together. The package is now operational but limited in what it supports. New efforts are now placed into making `ACE.jl` more general by splitting φ_i into several layers (the one-particle basis, the product basis, and the symmetric basis.). This new model is outside the scope of the current work, but we will briefly overview it in the Chapter 5.

4.4.1 Multiple properties

We quickly give an overview of the implementation on `ACE.jl` for completeness and so the reader can more easily understand the code base.

Before implementing nonlinear models, we enabled our ACE models to have multiple properties, i.e implement φ_i . This simply means generating a structure that allows for multiple linear models $(\varphi_i^{(1)}, \dots, \varphi_i^{(P)})$ with the same basis B , but different parameters c . The way `ACE.jl` works, is by taking full advantage of the typecasting and multiple dispatching Julia offers. An ACE linear model (**LinearACEModel**) is a structure that contains 3 elements, the ACE basis (B), the parameters of the model (θ_φ), and an evaluator object which exists only for dispatching. This structure has 5 important helper functions: **set_params()** (mutates the structure by setting the parameters), **evaluate()** (evaluates the site energy $\mathcal{E}(R_i)$), **grad_params()** (returns the derivative according to parameters), **grad_config()** (derivative according to $\{r_i\}_{i=1}^J$) and **grad_params_config()** (mixed derivative according to θ_φ and $\{r_i\}_{i=1}^J$). To allow for multiple properties, we had to write the same functions, but dispatching on the type of θ_φ .

4.4.2 Forward pass

The next step was defining evaluation of the site energies $\mathcal{E}(R_i) = \mathcal{F}(\varphi(f(R_i)))$, following the `Flux.jl` framework. We start by creating a structure called **LinearACE**, to hold both the parameters θ and a **LinearACEModel**. Then we set the

forward pass of this layer to simply call `evaluate()` on a configuration $f(R_i)$. This layer is then equivalent to φ_i . It can be composed with a nonlinearity \mathcal{F} to produce site energies. \mathcal{F} can be a user specified function or could be a `Flux.jl` layer. We can then define $\mathcal{F} \circ \varphi \circ f$ through `Chain()`, a composition function in `Flux.jl`. This creates a structure that holds θ and evaluates \mathcal{E} . We usually call this object a **model**. The gradients are then computed by taking the derivative of `model()` with respect to $\{\mathbf{r}_i\}_{i=1}^J$.

To obtain total energy E and forces F we rely on two functions from `JuLIP.jl`, `energy()` and `forces()`. These were extended in `ACEflux.jl` to support a **Flux-Potential**, which is a structure containing a **model** and a cutoff radius r_{cut} . `energy()` is (4.2), with $x \rightarrow \mathcal{F}(\varphi(x))$ being `model()` and f using the r_{cut} saved in the **Flux-Potential**. `forces()` uses the pullback of f to add and subtract contributions of the gradients of each atom according to the following equation:

$$F_i = -\frac{\partial E(R)}{\partial \mathbf{r}_i} + \sum_{\{j|r_{ij} < r_{\text{cut}}\}} \frac{\partial E(R)}{\partial \mathbf{r}_j} \quad (4.20)$$

where we use the negative gradient centered at i , like before, but now we add the contributions of the gradients of atoms inside the atomic neighbourhood $\{j|r_{ij} < r_{\text{cut}}\}$, $1 \leq j \leq J$. The set of neighbours is computed with a function similar to f , and the gradients are computed using `Zygote.jl`.

These functions can then be placed inside a loss function to optimize. In our case, the final forward pass for (3.8) is given as follows:

$$\begin{aligned} \text{loss}(\text{pot}, R, y) = & \alpha_E * (\text{energy}(\text{pot}, R) - y_E)^2 + \\ & \alpha_F * \text{sum}(\text{sum}((\text{forces}(\text{pot}, R) - y_F)^2)) \end{aligned} \quad (4.21)$$

with `pot` a **FluxPotential** object, R a simple structure, and y the corresponding DFT and force observations.

4.4.3 Making models differentiable

For the actual implementation of gradients, we decided to rely on `Zygote.jl`. `ChainRules.jl` and `Zygote.jl` work together to generate an automatic dif-

differentiation framework. Simply put, `ChainRules.jl` stores pullbacks for different functions (called **rrules**) and `Zygote.jl` reads a function and calls the necessary pullbacks in the proper order. However, `Zygote.jl` has reduced performance when differentiating complex functionals, for example φ_i , which makes differentiation slow and, in some rare cases, impossible. Therefore, we implemented custom pullbacks. Defining pullback functions with efficient gradient computations allows our codes to be fully differentiable efficiently.

An **rrule** is a `ChainRules.jl` function that dispatches on the type of h and computes both the value of $h(\theta)$ (the forward pass) and the pullback $\omega_\theta^T(g, h)$. We had to define a custom **rrule** for the derivative of forces with respect to parameters since `Zygote.jl` does not support second derivatives. We defined a helper function **adj_evaluate** as the pullback of **Linear_ACE**. Then we defined two **rrules** one that simply called **adj_evaluate**, and one that computed its pullback. This second **rrule** returned the second order derivatives.

Using `Zygote.jl` provides great flexibility. Even though we provide some custom pullbacks, `Zygote.jl` has built-in functionality to differentiate a great variety of functions. This allows us to compose several functions on top of **energy()** and **forces**, without defining **rrules** for them, which allows users to define the loss functions without having to implement their derivatives. Furthermore, the **rrules** that we define allow us to optimize the derivatives. For example, in Section 3.2, we saw how one could implement efficient derivatives of the forces, and by defining the **rrule** of **forces()** explicitly, we ensure efficient computation.

While `Zygote.jl` is crucial for differentiation and training of parameters, `Flux.jl` is not. Currently `Flux.jl` manages the parameters and creates the **Linear_ACE** layer. However, one could manage the parameters manually with **get_params()** and create **Linear_ACE** as a simple structure. The original motivation to make `ACE.jl` compatible with `Flux.jl` was to allow for the immediate use of machine learning utilities in our models, which would allow us to implement models with $\theta_{\mathcal{F}}$. At the time, it seemed that moulding our codes to fit this framework would allow for great flexibility and connection to well tested and maintained codes. However, in hindsight, these same features could have been implemented in comparable time without the use of `Flux.jl`. As we will see in the next chapter, the current path of ACE models is to allow for even more general nonlinearities.

With this goal in mind, it is likely that the framework enforced by `Flux.jl` will be too restrictive. Nonetheless, these ideas are still novel, and there is currently no clear best path.

4.5 Training

4.5.1 Multiprocessing

To speed up simulations, we parallelize computations among different workers. Since most of the cost of computation is the gradient evaluation, we decided to parallelize across training points $(R, y) \in (\mathbf{R}, Y)$. Similarly, the time spent sending and retrieving data from available cores is small compared to the time of a gradient evaluation. Therefore we settled on a multiprocessing implementation, in contrast to multithreading. We also evaluated using a GPU to perform optimization, however our matrices are very sparse, because of the symmetries on the $(\mathbf{n}, \mathbf{l}, \mathbf{m})$, and there is not enough support for sparse operations in GPUs in Julia.

We begin by dividing the training data (\mathbf{R}, Y) into subsets $(\mathbf{R}_\rho, Y_\rho)$ where $\rho \in [1, 2, \dots, \rho_c]$ and ρ_c is the number of processes available minus 1. This is so that we have a main process to feed the others and take optimization steps. The main process constructs a **model**, and sets random starting parameters θ . This model is then shared to all cores ρ , along with their corresponding $(\mathbf{R}_\rho, Y_\rho)$. The simulation then starts by sharing a starting θ_0 with all the processes, which subsequently set their local **model**'s parameters to θ_0 . Once done, each process computes its piece of the loss function and its gradient. Then all processes send their current losses and gradients to the main process, which adds them together. This is equivalent to splitting the summation over T in 3.7 and 3.10 into ρ_c parts, and then adding them together. The main process then adds the regularization and its gradient to create the total loss and gradient. These are then used to take an optimization step, and progress is logged into **JLD** files through `JLD.jl`.

4.5.2 Interacting with optimization packages in Julia

Since we adopted the `Flux.jl` framework, we can use any of its built-in optimizers. These include gradient descent, Nesterov's momentum descent, and several

versions of the ADAM algorithm. However, we wanted to use BFGS to optimize the Finnis-Sinclair model, and this method is not contained in Flux. `Optim.jl` offers BFGS, but we cannot simply plug our codes into it because of the non-standard structures in which `Zygote.jl` stores gradients. When taking gradients according to the attributes of our layer, `Zygote.jl` returns a **Grads(...)** object, which is a dictionary with the parameters as keys and the gradients as values, but `Optim.jl` expects the gradients in a flattened array. In order to make this work, we defined flattening and reshaping functions for both the gradients and the parameters. The main process performs these operations before and after taking an optimization step. We could theoretically use any optimizer in a similar fashion, as long as we can reshape the parameters and gradients accordingly.

4.6 Example

We will now go over an example of the functionality of `ACEflux.jl`. We begin by importing the necessary packages.

```
using ACE, ACEflux, Zygote, Flux, ACE, StaticArrays, ASE, JuLIP
using Zygote: gradient
```

We create a random configuration for testing.

```
R.i = ACE.ACEConfig([ACE.State(rr=rand(SVector{3, Float64}))) for _ = 1:10])
```

Recall that a nonlinear model is of the form (4.2), then the function call to construct a model for φ_i is

```
phi = Linear_ACE(max_deg, cor_order, num_props)
```

where **Linear_ACE()** takes a maximum degree of the polynomial, the correlation order \mathcal{N} , and the number of properties P to evaluate. Calling **phi(R.i)**, which will return P atomic properties. To add a nonlinearity, for example a Finnis-Sinclair like embedding, we simply compose the nonlinearity with **phi()**. We use `Flux.jl`'s **chain()**:

```
FS(phi) = phi[1] - sqrt(abs(phi[2])) + 1/100 - 1/10
E.i = Chain(phi, GenLayer(FS))
```

where **GenLayer** creates a `Flux.jl` structure to surround the nonlinear embedding. **E.i** now represents the function $\mathcal{F} \circ \varphi_i$. We can do more complex embeddings with trainable parameters:

```
E.i = Chain(phi, Dense(2,7), Dense(7,2), GenLayer(FS), sum)
```

where we use several Dense layers and a Finnis-Sinclair model. To compute gradients, we simply need to call the **gradient()** function in `Zygote.jl`. There are two ways to call this function, with explicit and with implicit parameters. The syntax is a little confusing at first, but the implicit parameters section in `Zygote`'s documentation is very helpful [28]. For an explicit call, we simply call **grads(function, parameters)** with a function and the parameters we want to differentiate according to

```
g.configs = gradient(E.i, R.i)
```

In this case, we are differentiating site energy with respect to configurations, which we use to calculate forces.

Now, to get a gradient according to the parameters, we need to do it implicitly because the parameters are defined implicitly in a `Flux.jl` layer. This is identical to the way `Flux.jl` is differentiated, so their documentation could prove helpful [30]. The function **params()** is `Flux.jl` native and allows us to extract the parameters of the model. To get the derivative of site energy according to its parameters, we would call:

```
g.params = gradient(()->E.i(R.i), params(E.i))
```

E.i() and **g.configs()** can be combined into a loss function or composed with more complex functions. The advantage of utilizing `Zygote.jl` and `Flux.jl` is that all these outer functions can be differentiated out of the box. However, `Zygote.jl` will still call our custom pullbacks when necessary, meaning the derivative will leverage the efficient adjoints. However, `Zygote.jl` does not differentiate functions with object mutation, so keep this in mind when creating these functions.

Even though this functions exist, the goal is not to differentiate site energies by hand, but rather call **energy()** and **forces()**, and for that we need to create a

FluxPotential(model, cutoff).

```
pot = FluxPotential(E.i, 6.0)
```

Now we create an atoms object to represent R , see `JuLIP` or `ACEatoms` documentation.

```
at = bulk(:Cu, cubic=true) * 3  
rattle !(at, 0.6) #adding random noise
```

We can now evaluate the energy and forces with our potential. This is the same syntax `JuLIP` and `ACEatoms` use.

```
e = energy(pot, at)  
f = forces(pot, at)
```

To calculate the gradients, we simply do an implicit **gradient()** call like before.

```
ge = gradient(() -> energy(pot, at), params(E.i))  
gf = gradient(() -> sum(forces(pot, at)^2), params(E.i))
```

Now we can create a loss function with **e** and **f** as in (4.21) and compute its derivative via

```
gl = gradient(() -> loss(pot, R, y), params(E.i))
```

Once we have this we can flatten the gradients like we mentioned in Section 4.5.2 and plug into any optimizer. For Section ?? we used BFGS in `Optim.jl` and used `ACEatoms.jl` to create R and y from the imported datasets [37]. We implemented multiprocessing by calling `@spawnat` and `@everywhere` from `Distributed.jl`.

Chapter 5

Conclusion and Outlook

5.1 Conclusion

Machine learned interatomic potentials continue to be a hot area of research, and nonlinear models might be crucial in the future. In this thesis, we provided an overview of the atomic cluster expansion as an atomic descriptor. We showcased results on the test data sets [37] for silicon, copper and molybdenum. To that end, we explored the parameter space of the weighting of energy versus forces and the tolerance for RRQR. Furthermore, we presented accuracy as a function of the number of parameters κ . We then extended this model by composing several linear models inside a nonlinearity. We demonstrated efficient evaluation of the gradients and comprehensively explained the Julia implementation of the models. We showed results for silicon, copper and molybdenum as a proof of concept. The codes are still in the experimental phase, and more work is needed for them to be competitive.

5.2 Outlook

In chapter 4 we went over the current implementation of nonlinear ACE models in Julia, which are still experimental. Much of the code had to be implemented manually, especially the gradients, which made our implementation efficient and flexible. In the future, we strive to capitalize on the generality of `ACE.jl`, rather

than focusing on a specialized type of models. Moving in this direction, the implementation of gradients through `Zygote.jl` will stay as the primary way to differentiate models, but the `Flux.jl` wrapper is subject to change. This is because a **Linear ACE** layer is quite a restrictive structure. We treat the computation of the basis \mathbf{B} as a black box. This allows for the nonlinearities implemented in this thesis but restricts the implementation of other types of physics-inspired nonlinearities. A few examples are (i) parametrization of the radial basis, (ii) composition of layers, and (iii) changing architecture.

To understand how these nonlinearities would work, we need to consider the basis evaluation in layers. Let us start with the input layer $\mathbf{R} = \{R^{(1)}, \dots, R^{(T)}\}$ where we have T atomic environments defined by their atomic positions $R^{(t)} = \{\mathbf{r}_1, \dots, \mathbf{r}_J\}$. One can extend the set $R^{(t)}$ to hold more properties. In fact this is already implemented in `ACE.jl`. A user can define an input

$$R^{(t)} := (\{\mathbf{r}_1, \dots, \mathbf{r}_J\}, \mathbf{W}_1, \dots, \mathbf{W}_K),$$

where \mathbf{W}_k can be other atomic features like magnetic properties, spin or even the output of another nonlinear ACE model. The treatment of features \mathbf{W}_k would have to be defined in each layer and is already being implemented.

The input layer is then fed into a one-particle basis ϕ using f . We can compare the action of f to a kernel (sometimes called a filter) in a convolutional neural network (figure 5.1). ϕ is then passed to an atomic basis A_{inlm} , which gives the product basis \mathbf{A}_{inlm} , and finally an atomic property φ_i . We do this for every node i in the J_t starting nodes for every $R^{(t)}$ in the training set. This is finally fed into a nonlinearity, and then everything is summed over to generate an energy (figure 5.2).

To calculate the forces, we would need to backpropagate and derivate according to \mathbf{r}_{ij} . Then we can combine the forward pass and the backward pass in a loss function to train energies and forces. This implementation is more flexible since we now have every computation step as a standalone layer. For example, to implement (i), one simply needs to create a parametric structure to substitute ϕ , where \mathcal{P} contains parameters to train. For (ii), we would simply need to compose the structures, and similarly in (iii). With all the steps defined as layers, we can com-

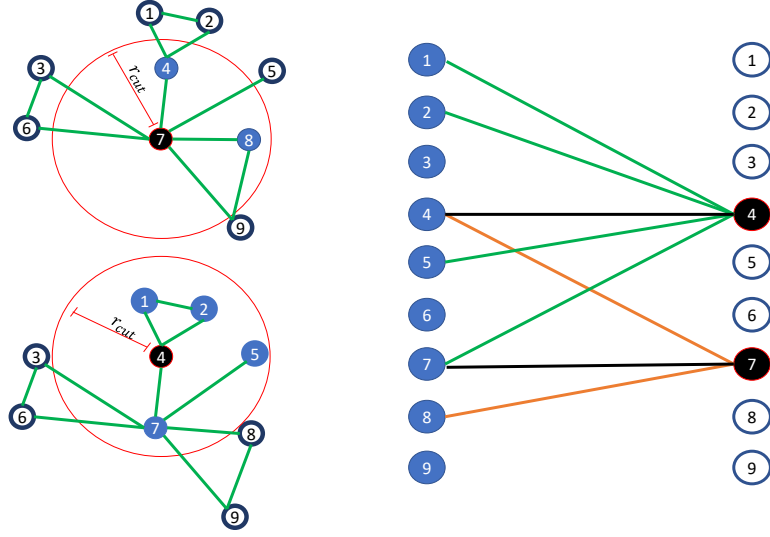


Figure 5.1: Example of f on a configuration for two atoms $i = \{7, 4\}$. On the right we see the atomic environment, and on the left we see the action of f on the input layer.

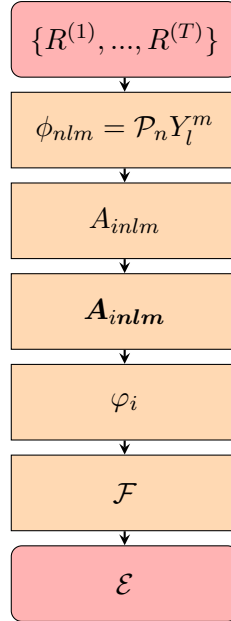


Figure 5.2: ACE model divided into layers.

pose them however we want. We could compose several atomic properties φ_i , or neural networks, or even switch layers like the one-particle basis for other descriptors. This new framework could be extremely general, but will require changes in the current Julia packages. We will need to implement the different layers as structures as well as a way to manage their parameters and the required pullbacks to differentiate through them with `Zygote.jl`.

Bibliography

- [1] R. Balestrierio and R. Baraniuk. Fast jacobian-vector product for deep networks, 2021. URL <https://arxiv.org/abs/2104.00219>. → page 32
- [2] A. P. Bartók, M. C. Payne, R. Kondor, and G. Csányi. Gaussian approximation potentials: The accuracy of quantum mechanics, without the electrons. *Physical Review Letters*, 104(13), Apr 2010. ISSN 1079-7114. doi:10.1103/physrevlett.104.136403. URL <http://dx.doi.org/10.1103/PhysRevLett.104.136403>. → pages 7, 8
- [3] J. Behler and M. Parrinello. Generalized neural-network representation of high-dimensional potential-energy surfaces. *Phys. Rev. Lett.*, 98:146401, Apr 2007. doi:10.1103/PhysRevLett.98.146401. URL <https://link.aps.org/doi/10.1103/PhysRevLett.98.146401>. → page 6
- [4] N. Bernstein, G. Csányi, and V. L. Deringer. De novo exploration and self-guided learning of potential-energy surfaces. *npj Computational Materials*, 5(1), 2019. doi:10.1038/s41524-019-0236-6. → page 1
- [5] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah. Julia: A fresh approach to numerical computing. *SIAM review*, 59(1):65–98, 2017. URL <https://doi.org/10.1137/141000671>. → page 31
- [6] E. Cancès, C. J. García-Cervera, and Y. A. Wang. Density functional theory: Fundamentals and applications in condensed matter physics. *BIRS workshop*, Jan 2011. doi:<https://www.birs.ca/workshops/2011/11w5121/report11w5121.pdf>. → page 2
- [7] M. A. Caro, V. L. Deringer, J. Koskinen, T. Laurila, and G. Csányi. Growth mechanism and origin of high sp^3 content in tetrahedral amorphous carbon. *Phys. Rev. Lett.*, 120:166101, Apr 2018. doi:10.1103/PhysRevLett.120.166101. URL <https://link.aps.org/doi/10.1103/PhysRevLett.120.166101>. → page 1

- [8] A. R. Christoph Ortner. Aceflux.jl. <https://github.com/ACESuit/ACEflux.jl>, 2022. → page 36
- [9] D. P. K. Christoph Ortner. Aceatoms.jl. <https://github.com/ACESuit/ACEatoms.jl>, 2022. → page 35
- [10] G. C. Christoph Ortner, James Kermode and et al. Acesuit. <https://github.com/ACESuit>, 2022. → page 35
- [11] J. G. D. P. K. T. K. C. v. d. O. F. E. F. P. Christoph Ortner, James Kermode and et al. Julip.jl. <https://github.com/JuliaMolSim/JuLIP.jl>, 2022. → page 35
- [12] M. S. A. R. C. v. d. O. Christoph Ortner, Liwei Zhang. Ace.jl. <https://github.com/ACESuit/ACE.jl>, 2022. → page 35
- [13] G. Csanyi. Machine learning the quantum mechanics of materials and molecules. URL <https://www.youtube.com/watch?v=ZjBff6-5amo>. → page 4
- [14] X. Dai, Y. Kong, J. Li, and B. Liu. Extended finnis–sinclair potential for bcc and fcc metals and alloys. *Journal of Physics: Condensed Matter*, 18:4527, 04 2006. doi:10.1088/0953-8984/18/19/008. → pages 5, 23
- [15] R. Drautz. Atomic cluster expansion for accurate and transferable interatomic potentials. *Phys. Rev. B*, 99:014104, Jan 2019. doi:10.1103/PhysRevB.99.014104. URL <https://link.aps.org/doi/10.1103/PhysRevB.99.014104>. → page 23
- [16] G. Dusson, M. Bachmayr, G. Csanyi, R. Drautz, S. Etter, C. van der Oord, and C. Ortner. Atomic cluster expansion: Completeness, efficiency and stability, 2021. → pages iii, 9, 12, 13, 27
- [17] F. Ercolessi and J. B. Adams. Interatomic potentials from first-principles calculations: The force-matching method. *Europhysics Letters (EPL)*, 26(8): 583–588, Jun 1994. ISSN 1286-4854. doi:10.1209/0295-5075/26/8/005. URL <http://dx.doi.org/10.1209/0295-5075/26/8/005>. → page 6
- [18] M. R. Fellingner, A. M. Z. Tan, L. G. Hector, and D. R. Trinkle. Geometries of edge and mixed dislocations in bcc fe from first-principles calculations. *Phys. Rev. Materials*, 2:113605, Nov 2018. doi:10.1103/PhysRevMaterials.2.113605. URL <https://link.aps.org/doi/10.1103/PhysRevMaterials.2.113605>. → page 1
- [19] M. Gu and S. C. Eisenstat. Efficient algorithms for computing a strong rank-revealing qr factorization. *SIAM Journal on Scientific Computing*, 17(4):848–869, 1996. doi:10.1137/0917055. → page 17

- [20] M. Hellström, V. Quaranta, and J. Behler. One-dimensional vs. two-dimensional proton transport processes at solid–liquid zinc-oxide–water interfaces. *Chemical Science*, 10(4):1232–1243, 2019. [doi:10.1039/c8sc03033b](https://doi.org/10.1039/c8sc03033b). → page 1
- [21] P. Hohenberg and W. Kohn. Inhomogeneous electron gas. *Physical Review*, 136(3B), 1964. [doi:10.1103/physrev.136.b864](https://doi.org/10.1103/physrev.136.b864). → pages 2, 3
- [22] J.-W. Jiang and Y.-P. Zhou. Parameterization of stillinger-weber potential for two- dimensional atomic crystals. *Handbook of Stillinger-Weber Potential Parameters for Two-Dimensional Atomic Crystals*, Dec 2017. [doi:10.5772/intechopen.71929](https://doi.org/10.5772/intechopen.71929). URL <http://dx.doi.org/10.5772/intechopen.71929>. → page 5
- [23] S. O. M. S. E. J. Ken Ho, Sheehan Olver. Lowrankapprox.jl. <https://github.com/JuliaMatrices/LowRankApprox.jl>, 2021. → pages 17, 18
- [24] B.-J. Lee, W.-S. Ko, H.-K. Kim, and E.-H. Kim. The modified embedded-atom method interatomic potentials and recent progress in atomistic simulations. *Calphad*, 34(4):510–522, 2010. ISSN 0364-5916. [doi:https://doi.org/10.1016/j.calphad.2010.10.007](https://doi.org/10.1016/j.calphad.2010.10.007). URL <https://www.sciencedirect.com/science/article/pii/S0364591610000817>. → page 5
- [25] R. LeSar. *Introduction to computational materials science: Fundamentals to applications*. Cambridge University Press, 2016. → pages 3, 4, 5
- [26] Y. Lysogorskiy, C. van der Oord, A. Bochkarev, S. Menon, M. Rinaldi, T. Hammerschmidt, M. Mrovec, A. Thompson, G. Csányi, C. Ortner, and R. Drautz. Performant implementation of the atomic cluster expansion (pace): Application to copper and silicon, 2021. → page 27
- [27] C. L. Mike J Inness, Michael Abbott and et al. Zygote.jl. <https://github.com/FluxML/Zygote.jl>, 2022. → page 36
- [28] C. L. Mike J Inness, Michael Abbott and et al. Zygote.jl documentation. <https://fluxml.ai/Zygote.jl/latest/#Explicit-and-Implicit-Parameters-1>, 2022. → page 42
- [29] D. G. Mike J Inness, Carlo Lucibello and et al. Flux.jl. <https://github.com/FluxML/Flux.jl>, 2022. → page 36
- [30] D. G. Mike J Inness, Carlo Lucibello and et al. Flux.jl documentation. <https://fluxml.ai/Flux.jl/stable/models/basics/>, 2022. → page 42

- [31] J. Nocedal and S. J. Wright. *Numerical optimization*. Springer, 2006. → page 28
- [32] A. R. Oganov, C. J. Pickard, Q. Zhu, and R. J. Needs. Structure prediction drives materials discovery. *Nature Reviews Materials*, 4(5):331–348, 2019. doi:10.1038/s41578-019-0101-8. → page 1
- [33] C. Ortner. Ace1pack.jl. <https://github.com/ACEsuit/ACE1pack.jl>, 2022. → pages 18, 35
- [34] J. M. W. Patrick Kofod Mogensen and et al. Optim.jl. <https://github.com/JuliaNLSolvers/Optim.jl/>, 2022. → page 36
- [35] A. V. Shapeev. Moment tensor potentials: A class of systematically improvable interatomic potentials. *Multiscale Model. Simul.*, 14:1153–1173, 2016. → pages 3, 8
- [36] M. Zgubic. Chainrules.jl documentation. <https://juliadiff.org/ChainRulesCore.jl/stable/>, 2022. → pages 33, 35
- [37] Y. Zuo, C. Chen, X. Li, Z. Deng, Y. Chen, J. Behler, G. Csányi, A. V. Shapeev, A. P. Thompson, M. A. Wood, and et al. Performance and cost assessment of machine learning interatomic potentials. *The Journal of Physical Chemistry A*, 124(4):731–745, 2020. doi:10.1021/acs.jpca.9b08723. → pages iii, 18, 43, 44