Working Title

University of Oslo

Andreas Godø Lefdalsnes

May 4, 2019

Abstract

Abstract goes here [e.g. 1, page 300].

Contents

Chapter 1

Introduction

Intro.

1.0.1 Molecular dynamics

1.0.2 Atom-centered descriptors

1.0.3 Goals

1.0.4 Contributions

1.0.5 Structure

Part I

Theory

Chapter 2

Quantum Mechanics

The discussion in this chapter follows closely the discussion in [Sakurai — Modern Quantum Mechanics].
In quantum mechanics, a physical state is represented by a *state vector* in a complex vector space. Such a vector is called a *ket*, denoted by $|\alpha\rangle$. The state ket is postulated to contain all information about the physical state. Two kets can be added to produce a new ket:

$$|\alpha\rangle + |\beta\rangle = |\gamma\rangle \,.$$

They can also be multiplied by a complex number:

$$c\,|\alpha\rangle = |\alpha\rangle\, c = |\delta\rangle \,.$$

If c is zero the resulting ket is called a *null ket*. If c is non-zero it is postulated that the resulting ket contains the same information.
Observables such as momentum and spin are represented by operators acting on the vector space in question. Operators act on a ket from the left to produce a new ket:

$$A\,|\alpha\rangle = |\delta\rangle \,.$$

Of particular importance is when the action of an operator on a ket is the same as multiplication:

$$A\,|\alpha\rangle = c\,|\alpha\rangle = |\delta\rangle \,.$$

These kets are known as *eigenkets* and the corresponding complex numbers are known as *eigenvalues*. The physical state represented by an eigenket is

known as an *eigenstate*. Any ket can be written as an expansion of eigenkets $|a'\rangle$:

$$|\alpha\rangle = \sum_{a'} c_{a'} |a'\rangle ,$$

where $c_{a'}$ is a complex coefficient. In principle there are infinitely many linearly indepedent eigenkets, depending on the dimensionality of the vector space. The uniqueness of the expansion can be proven with orthonogality of the eigenkets, which we will simply postulate.

A *bra space* is a vector space "dual" to the ket space. We postulate that for every ket $|\alpha\rangle$ there exists a bra $\langle\alpha|$. The bra space is spanned by eigenbras $\langle a'|$ corresponding to the eigenkets $|a'\rangle$. The ket and bra spaces have a dual correspondence:

$$|\alpha\rangle \leftrightarrow \langle\alpha|$$
$$|\alpha'\rangle , |\alpha''\rangle , \cdots \leftrightarrow \langle\alpha'| , \langle\alpha''| , \ldots$$
$$|\alpha\rangle + |\beta\rangle \leftrightarrow \langle\alpha| + \langle\beta| .$$

The bra dual to $c|\alpha\rangle$ is postulated to be $c^* |\alpha\rangle$, and more generally:

$$c_\alpha |\alpha\rangle + c_\beta |\beta\rangle \leftrightarrow c_\alpha^* \langle\alpha| + c_\beta^* \langle\beta| .$$

The *inner product* of a bra and a ket is a complex number written as a bra on the left and a ket on the right. It has the fundamental property:

$$\langle\alpha|\beta\rangle = \langle\beta|\alpha\rangle^* ,$$

in other words they are complex conjugates. For this to satisfy the requirements of an inner product we must have

$$\langle\alpha|\alpha\rangle \geq 0,$$

with equality if and only if $|\alpha\rangle$ is a null ket. We define the *norm* of a ket as

$$\sqrt{\langle\alpha|\alpha\rangle},$$

which can be used to form normalized kets

$$|\alpha^\sim\rangle \frac{1}{\sqrt{\langle\alpha|\alpha\rangle}} |\alpha\rangle ,$$

with the property

$$\langle\alpha^\sim|\alpha^\sim\rangle = 1.$$

Two kets are said to be *orthogonal* if

$$\langle\alpha|\beta\rangle = 0.$$

2.0.1 Operators

As we briefly mentioned above, operators act on kets from the left to produce a new ket. Two operators A and B are equal

$$A = B$$

if

$$A\left|\alpha\right\rangle = B\left|\alpha\right\rangle$$

for an arbitrary ket in the relevant ket space. An operator A is said to be the *null operator* if

$$A\left|\alpha\right\rangle = 0.$$

Operators can be added, and addition operations are commutative and associative.

$$X + Y = Y + X,$$

$$(X + Y) + Z = X + (Y + Z).$$

Operators act on bras from the right to produce a new bra

$$\left\langle\alpha\right| A = \left\langle\beta\right|.$$

The ket $A\left|\alpha\right\rangle$ and the bra $\left\langle\alpha\right| A$ are in general not dual to each other. We define the *hermitian adjoint* $A^\dagger$ as

$$A\left|\alpha\right\rangle \leftrightarrow \left\langle\alpha\right| A^\dagger.$$

An operator is said to be *hermitian* if

$$A = A^\dagger.$$

Operators can be multiplied. Multiplication is associative, but non-commutative:

$$XY \neq YX,$$

$$X(YZ) = (XY)Z.$$

The left product of a ket and a bra is known as the *outer product*:

$$\left|\alpha\right\rangle\left\langle\beta\right|.$$

The outer product should be treated as an operator, while the inner product $\langle\alpha|\beta\rangle$ is a complex number.
If an operator is to the left of a ket $|\alpha\rangle A$ or to the right of a bra $A\langle\beta|$ these are illegal products, in other words not defined within the ruleset of quantum mechanics. The associative properties of operators are postulated to hold true as long as we are dealing with legal multiplications among kets, bras and operators. As an example, the outer product acting on a ket:

$$(|\alpha\rangle\langle\beta|)|\gamma\rangle,$$

can be equivalently regarded as scalar multiplication

$$|\alpha\rangle(\langle\alpha|\gamma\rangle) = |\alpha\rangle c = c|\alpha\rangle,$$

where $c = \langle\alpha|\gamma\rangle$ is just a complex number.

2.0.2 Time evolution

In quantum mechanics, time is treated not as an observable, but as a parameter. Relativistic quantum mechanics treats space and time on the same footing, but only by demoting position to a parameter.
Suppose we have a physical system $|\alpha\rangle$ at a time t_0. Denote the ket at a later time $t > t_0$ by

$$|\alpha, t; t_0\rangle.$$

As time is assumed to be a continuous parameter we expect as we evolve the system backward in time

$$\lim_{t\to t_0} |\alpha, t; t_0\rangle = |\alpha\rangle.$$

The kets separated by a time $t - t_0$ are related by the *time-evolution operator* $\mathcal{U}$:

$$|\alpha, t; t_0\rangle = \mathcal{U}(t, t_0)|\alpha, t_0\rangle.$$

If the state ket is normalized to unity at a time t_0, it must remain normalized at a later time:

$$\langle\alpha, t_0|\alpha, t_0\rangle = \langle\alpha, t; t_0|\alpha, t; t_0\rangle = 1.$$

This is guaranteed if the time evolution operator $\mathcal{U}$ is a *unitary* operator:

$$\mathcal{U}^\dagger\mathcal{U} = 1.$$

We also require the composition property:

$$\mathcal{U}(t_2, t_0) = \mathcal{U}(t_2, t_1)\mathcal{U}(t_1, t_0),\ (t_2 > t_1 > t_0).$$

If we consider an infinitesimal time-evolution operator

$$|\alpha, t_0 + dt; t_0\rangle = \mathcal{U}(t_0 + dt, t_0)\,|\alpha, t_0\rangle\,,$$

it must reduce to the identity operator as the infinitesimal time interval dt goes to zero:

$$\lim_{dt\to 0} \mathcal{U}(t_0 + dt, t_0) = 1,$$

and we expect the difference between the operators to be of first order in dt. These requirements are all satisfied by

$$\mathcal{U}(t_0 + dt, t_0) = 1 - i\Omega dt,$$

where Ω is a Hermitian operator:

$$\Omega^\dagger = \Omega.$$

The operator Ω has the dimension inverse time. Frequency or inverse time is related to energy through the Planck-Einstein relation:

$$E = \hbar\omega.$$

In classical mechanics the Hamiltonian is the generator of time evolution, so we postulate that t Ω is related to the Hamiltonian operator H:

$$\Omega = \frac{H}{\hbar}.$$

The Hamiltonian operator represents the energy of our system, which is a physical observable and must therefore be Hermitian.

2.0.3 The Schrodinger equation

By exploiting the composition property of the time-evolution operator we obtain:

$$\mathcal{U}(t + dt, t_0) = \mathcal{U}(t + dt, t)\mathcal{U}(t, t_0) = (1 - \frac{iHdt}{\hbar})\mathcal{U}(t, t_0),$$

where the time difference $t-t_0$ is not required to be infinitesimal. This implies

$$\mathcal{U}(t+dt,t_0)-\mathcal{U}(t,t_0)=-i(\frac{H}{\hbar})dt\mathcal{U}(t,t_0),$$

and taking the limit $dt\to 0$:

$$i\hbar\frac{\partial}{\partial dt}\mathcal{U}(t,t_0)=H\mathcal{U}(t,t_0).$$

This is known as the Schrodinger equation for the time-evolution operator. Multiplying both sides by a ket $|\alpha,t_0\rangle$ leads to the Schrodinger equation:

$$i\hbar\frac{\partial}{\partial dt}\mathcal{U}(t,t_0)\,|\alpha,t_0\rangle=H\mathcal{U}(t,t_0)\,|\alpha,t_0\rangle\,.$$

As the ket does not depend on time this is the same as

$$i\hbar\frac{\partial}{\partial dt}\,|\alpha,t_0\rangle=H\,|\alpha,t_0\rangle\,.$$

Chapter 3

Manybody Quantum Mechanics

This chapter will give a brief overview of the Hartree-Fock and Density Functional Theory methods, which are the primary workhorses in manybody quantum theory.

In most applications of quantum theory we are interested in finding solutions to the non-relativistic time-independent Schrodinger equation:

$$\hat{H}\left|\Psi\right\rangle = E\left|\Psi\right\rangle,$$

with the Hamiltonian $\hat{H}$ describing a system of nuclei and electrons with coordinates $\boldsymbol{R}_i,\ i = 1, 2, \ldots, A$ and $\boldsymbol{r}_i,\ i = 1, 2, \ldots, N$ respectively. The distance between nuclei a and electron i is given as $r_{ai} = |\boldsymbol{R}_a - \boldsymbol{r}_i|$ and correspondingly for the nuclei-nuclei and electron-electron distances.

The full Hamiltonian for a set of N electrons and A nuclei in atomic units is

$$\begin{aligned}\hat{H} = &-\sum_{i=1}^{N}\frac{1}{2}\nabla_i^2 - \sum_{a=1}^{A}\frac{1}{M_a}\nabla_a^2 - \sum_{i=1}^{N}\sum_{a=1}^{A}\frac{Z_a}{r_{ia}} \\ &+\sum_{i=1}^{N}\sum_{j=i+1}^{N}\frac{1}{r_{ij}} + \sum_{a=1}^{A}\sum_{b=a+1}^{A}\frac{Z_a Z_b}{R_{ab}}\end{aligned}. \tag{3.1}$$

The first two terms describe the kinetic energy operators of the electrons and nuclei, with M_a the ratio of the mass of nuclei a to the electron mass. The third term describes the coulomb attraction between electron and nuclei,

while the fourth and fifth terms describe the repulsion between electrons and nuclei respectively.

From this description we are most often interested in the electronic structure problem presented by applying the *Born-Oppenheimer* approximation. Since the nuclei are approximately 2000 times heavier than the electrons, the electrons can to a good approximation be described as moving in the field of fixed nuclei. This means we can neglect the kinetic energy terms of the nuclei, while considering an averaged effect from the nuclei-nuclei repulsion. The nuclei-nuclei repulsion energy adds a constant to the energy eigenvalues, but has no effect on the energy eigenfunctions. The remaining terms are known as the electronic Hamiltonian:

$$\hat{H}_e = -\sum_{i=1}^{N} \frac{1}{2}\nabla_i^2 - \sum_{i=1}^{N}\sum_{a=1}^{A} \frac{Z_A}{r_{ia}} + \sum_{i=1}^{N}\sum_{j=i+1}^{N} \frac{1}{r_{ij}}. \tag{3.2}$$

The electronic wavefunction $\Psi_e = \Psi_e(\{r_i\};\{R_a\})$ is a function of the electronic coordinates with a parametric dependence on the fixed nucleic coordinates. The electronic energy is obtained in the usual way $E_e = \langle \Psi_e | \hat{H}_e | \Psi_e \rangle$. The total energy of our system must now include the constant nuclear repulsion:

$$E_{tot} = E_e + \sum_{a=1}^{A}\sum_{b=a+1}^{A} \frac{Z_a Z_b}{R_{ab}}.$$

If one has solved the Schrodinger equation for the electronic Hamiltonian, one can subsequently solve for the nucleic motion using the same method, i.e. substituting the electronic coordinates for their average values, averaged over the electronic wave function. We are then left with a nuclear Hamiltonian $\hat{H}_n$:

$$\begin{aligned} \hat{H}_n &= -\sum_{a=1}^{A} \frac{1}{2M_a}\nabla_a^2 + \langle -\sum_{i=1}^{N} \frac{1}{2}\nabla_i^2 - \sum_{i=1}^{N}\sum_{j=i+1}^{N} \frac{1}{r_{ij}} \rangle \\ &\quad + \sum_{a=1}^{A}\sum_{b=a+1}^{A} \frac{Z_a Z_b}{R_{ab}} \\ &= -\sum_{a=1}^{A} \frac{1}{2M_a}\nabla_a^2 + E_{tot}. \end{aligned} \tag{3.3}$$

Under this approximation the nuclei move on a potential energy surface obtained by solving the electronic Hamiltonian.

Finally, we would also like to go one step further and neglect the relative positions of the nuclei. To this end we consider only the center of mass of the nucleus with a total charge Z, which gives us the final expression we want for our Hamiltonian:

$$\hat{H} = -\sum_{i=1}^{N} \frac{1}{2}\nabla_i^2 - \sum_{i=1}^{N} \frac{Z}{r_i} + \sum_{i=1}^{N} \sum_{j=i+1}^{N} \frac{1}{r_{ij}} \tag{3.4}$$

3.0.1 Hartree-Fock

[cite](http://vergil.chemistry.gatech.edu/notes/hf-intro/node3.html) The Hartree-Fock method is a method for finding solutions to the electronic Hamiltonian assuming the electron-electron repulsion can be approximated with a set of single-particle functions or *orbitals* moving in a mean field generated by the presence of other electrons. Assuming that the electrons do not interact the Hamiltonian is separable and the wavefunction is simply a product of orbitals ψ which are solutions to a onebody Hamiltonian. This gives us an ansatz for the manybody wavefunction Ψ known as the *Hartree product*:

$$\Psi(\boldsymbol{r}_1, \ldots, \boldsymbol{r}_N) = \psi(\boldsymbol{r}_1) \cdot \cdots \cdot \psi(\boldsymbol{r}_N).$$

Since we are dealing with fermions this ansatz fails to satisfy the antisymmetry principle, i.e. the wavefunction is not antisymmetric with respect to the interchange of any two particles. Fermions in addition to three spatial degrees of freedom also have a spin degree of freedom σ which means the fermion can be described by the space-spin coordinate $\boldsymbol{x} = (\boldsymbol{r}, \sigma)$ with $\boldsymbol{x} \in \mathbb{R}^3 \otimes \sigma$.

The problem of antisymmetry in a system of N fermions is satisfied by the introduction of *Slater determinants*

$$\Psi(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N) = \frac{1}{\sqrt{N}} \begin{vmatrix} \chi_1(\boldsymbol{x}_1) & \chi_2(\boldsymbol{x}_1) & \cdots & \chi_N(\boldsymbol{x}_1) \\ \chi_1(\boldsymbol{x}_2) & \chi_2(\boldsymbol{x}_2) & \cdots & \chi_N(\boldsymbol{x}_2) \\ \vdots & \vdots & \ddots & \vdots \\ \chi_1(\boldsymbol{x}_N) & \chi_2(\boldsymbol{x}_N) & \cdots & \chi_N(\boldsymbol{x}_N) \end{vmatrix}, \tag{3.5}$$

with $\chi(\boldsymbol{x})$ spin orbitals and a normalization factor $(N!)^{-1/2}$. The introduction of this ansatz is equivalent to assuming that all electrons move

independently of each other in a mean field generated by the electron-electron repulsion.

Define the one-electron operator of the electronic Hamiltonian as

$$\hat{h}_1(\boldsymbol{x}_i) = -\frac{1}{2}\nabla_i^2 - \frac{Z}{r_i},$$

with a twobody interaction term

$$\hat{v}(\boldsymbol{x}_i, \boldsymbol{x}_j) = \frac{1}{r_{ij}},$$

with the electronic Hamiltonian written more compactly as

$$\hat{H} = \sum_i \hat{h}_1(\boldsymbol{x}_i) + \sum_{i<j} \hat{v}(\boldsymbol{x}_i, \boldsymbol{x}_j).$$

The expectation value of the energy is given as

$$\langle \Psi | \hat{H} | \Psi \rangle \,.$$

The *variational theorem* says the expectation value of any normalized wavefunction with respect to the energy is an upper bound to the ground state energy. This suggests a procedure wherein we vary the parameters of a set of approximate wavefunction Ψ_T until an energy minimum is reached.

The Hartree-Fock energy can be written in terms of integrals over the onebody and interaction terms:

$$E_{HF} = \sum_i \langle i | \hat{h} | i \rangle + \sum_{i<j} \langle ij | \hat{v} | ij \rangle_{AS} \,,$$

where we have introduced an antisymmetrized matrix element

$$\langle ij | \hat{v} | ij \rangle_{AS} = \langle ij | \hat{v} | ij \rangle - \langle ij | \hat{v} | ji \rangle \,,$$

and the shorthand integrals

$$\langle i | \hat{h}_1 | i \rangle = \int d\boldsymbol{r} \chi_i^* \hat{h}_1 \chi,$$

and

$$\langle ij | \hat{v} ij \rangle = \int d\boldsymbol{r}_i d\boldsymbol{r}_j \chi_i^* \chi_j^* \hat{v} \chi_i \chi_j.$$

In order to solve these integrals numerically we perform a linear expansion of the spin orbitals χ in terms of a fixed orthogonal basis ϕ:

$$\chi_i = \sum_\lambda C_{i\lambda}\phi_\lambda,$$

in principle an infinite sum, but in practice truncated.
This allows us to rewrite the Hartree-Fock energy as

$$E_{HF} = \sum_i \sum_{\alpha,\beta} C^*_{i\alpha} C_{i\beta} \langle \alpha | \hat{h}_1 | \beta \rangle + \sum_{i<j} \sum_{\alpha,\beta,\delta,\eta} C^*_{i\alpha} C^*_{j\beta} C_{i\delta} C_{j\eta} \langle \alpha\beta | \hat{v} | \delta\eta \rangle .$$

Work out the remainder here...

3.0.2 Density functional theory

Density functional theory is a method for investigating the electronic structure of a manybody system by finding approximations to the ground state density $\rho(\boldsymbol{r})$. Our starting point is again the electronic Hamiltonian

$$\hat{H} = -\sum_{i=1}^{N} \frac{1}{2}\nabla_i^2 - \sum_{i=1}^{N} \frac{Z}{r_i} + \sum_{i=1}^{N} \sum_{j=i+1}^{N} \frac{1}{r_{ij}}. \tag{3.6}$$

Any electronic wavefunction Ψ which solves this equation is in principle a function of $4N$ coordinates $\boldsymbol{x}_i = (\boldsymbol{r}_i, \sigma_i),\ i = 1, \ldots, N$ where N is the number of electrons. Once we have obtained a solution to the Schrodinger equation we can obtain the one-electron density $\rho(\boldsymbol{r})$ as:

$$\rho(\boldsymbol{r}) = \int |\Psi(\boldsymbol{r}, \sigma, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_N)|^2 \, d\sigma d\boldsymbol{x}_2 \ldots d\boldsymbol{x}_N. \tag{3.7}$$

Since the wavefunction is a unique functional of the Hamiltonian $\hat{H}$, the one-electron density is uniquely determined by the Hamiltonian. Hohenberg and Kohn showed in 1964 in their first theorem that this mapping can be inverted, i.e. that the one-electron density uniquely determines the Hamiltonian of our system (up to an arbitrary constant). Taken altogether, this means that all properties of our system, including the Hamiltonian and the manybody wavefunction are fixed by a one-electron density carrying a dependency on only 3 spatial coordinates.

The electronic Hamiltonian can be rewritten as

$$\hat{H} = \hat{F} + \hat{V}_{ne},$$

where $\hat{F}$ is an operator consisting of the kinetic energy and electron-electron operators and $\hat{V}_{ne}$ is the electron-nuclei interaction.

For their second theorem, Hohenberg and Kohn defined the universal density functional

$$F[n] = \langle \Psi[n] | \hat{F} | \Psi[n] \rangle ,$$

and the total electronic energy functional

$$E[n] = F[n] + \int \hat{V}_{ne} n(\boldsymbol{r}) d\boldsymbol{r}.$$

Hohenberg and Kohn showed the energy functional with respect to N-electron densitites $n(\boldsymbol{r})$ is an upper bound to the ground state energy:

$$E_0 \leq F[n] + \int \hat{V}_{ne} n(\boldsymbol{r}) d\boldsymbol{r},$$

with equality if and only if the one-electron density is the one-electron density corresponding to our system $\hat{H}$. This suggests a variational procedure, wherein we minimize the total electronic energy functional until we reach an energy minimum:

$$E_{min} = \min_{n} E[n],$$

which serves as our best estimate for the ground state one-electron density $\rho_0(\boldsymbol{r})$.

Levy and Lieb proposed to redefine the universal density functional in terms of normalized antisymmetric wavefunctions Ψ which yield a fixed density ρ:

$$F[n] = \Psi \underset{\min}{\rightarrow} \rho \langle \Psi | \hat{F} | \Psi \rangle = \langle \Psi[n] | \hat{F} | \Psi[n] \rangle ,$$

wherein the minima search is performed over wavefunctions which yield the fixed density ρ. A search is then performed over densitities ρ until we reach an energy minimum. This method is known as the *constrained search formulation*. However, we still do not have an explicit expression for the

universal density functional $F[n]$, and direct approximations have proved difficult.

Kohn and Sham proposed to decompose $F[n]$ as

$$F[n] = T_s[n] + E_{Hxc}[n],$$

where $T_s[n]$ is a non-interacting kinetic-energy functional which can be defined through the constrained-search formulation:

$$T_s[n] = \min_{\Phi \to n} \langle \Phi | \hat{T} | \Phi \rangle = \langle \Phi[n] | \hat{T} | \Phi[n] \rangle \,,$$

wherein the minima search is now performed over normalized single-determinant wavefunctions Φ which yield the fixed density ρ. The functional $E_{Hxc}[n]$ is known as the Hartree-exchange-correlation functional. The variational procedure is then performed over single-determinant wavefunctions which yield a fixed density ρ and then minimized over densities:

$$\begin{aligned} E_0 &= \min_n \{ \, F[n] + \int \hat{V}_{ne} n(\boldsymbol{r}) d\boldsymbol{r} \, \} \\ &= \min_n \min_{\Phi \to n} \{ \, \langle \Phi \mid \hat{T} + \hat{V}_{ne} \mid \Phi \rangle + E_{Hxc}[n_\Phi] \, \} \\ &= \min_\Phi \{ \, \langle \Psi \mid \hat{T} + \hat{V}_{ne} \mid \Phi \rangle + E_{Hxc}[n_\Phi] \, \} \, . \end{aligned} \tag{3.8}$$

As with the Hartree-Fock method, the single determinant wavefunctions are constructed from an orthonormal basis of spin orbitals $\chi_i(\boldsymbol{x}),\ i = 1, \ldots, N$ with $\boldsymbol{x} = (\boldsymbol{r}, \sigma)$. The total electronic energy can be expressed in terms of spatial orbitals $\phi_i(\boldsymbol{r})$ after integrating over the spin variables:

$$E = \sum_i \int \phi_i^*(\boldsymbol{r}) (-\frac{1}{2}\nabla^2 + \hat{v}_{ne}) \phi_i(\boldsymbol{r}) d\boldsymbol{r} + E_{Hxc}[n], \tag{3.9}$$

with the density expressed as

$$n(\boldsymbol{r}) = \sum_i |\phi_i(\boldsymbol{r})|^2 \, . \tag{3.10}$$

With the constraint that the spatial orbitals be normalized we obtain the following Lagrangian:

$$\mathcal{L}[\{ \, \phi_i \, \}] = E[\{ \, \phi_i \, \}] - \sum_i \epsilon_i \left(\int \phi_i^*(\boldsymbol{r}) \phi_i(\boldsymbol{r}) d\boldsymbol{r} - 1 \right),$$

with ϵ_i the associated Lagrangian multiplier. We find the energy minimum where the Lagrangian is stationary:

$$\frac{\partial \mathcal{L}}{\partial \phi_i^*(\boldsymbol{r})} = 0.$$

The spatial orbitals are expanded as a linear combination of a known basis, such as hydrogen-like functions or Gaussian-type orbitals:

$$\phi_i(\boldsymbol{r}) = \sum_{\lambda} C_{\lambda i} \chi_i(\boldsymbol{r}).$$

Chapter 4

Molecular Dynamics

Our starting point for molecular dynamics is the full Hamiltonian for a set of N electrons and A nuclei:

$$\begin{aligned}\hat{H} = &-\sum_{i=1}^{N}\frac{1}{2}\nabla_i^2 - \sum_{a=1}^{A}\frac{1}{M_a}\nabla_a^2 - \sum_{i=1}^{N}\sum_{a=1}^{A}\frac{Z_a}{r_{ia}} \\ &+\sum_{i=1}^{N}\sum_{j=i+1}^{N}\frac{1}{r_{ij}} + \sum_{a=1}^{A}\sum_{b=a+1}^{A}\frac{Z_a Z_b}{R_{ab}}\end{aligned}. \tag{4.1}$$

We want to find solutions to the time-dependent non-relativistic Schrodinger equation:

$$i\hbar\frac{\partial}{\partial t}\Psi = \hat{H}\Psi.$$

4.0.1 From quantum mechanics to molecular dynamics

We will follow the route of Tully described in [https://core.ac.uk/download/pdf/35009882.pdf?reposit
The wavefunction is separated in terms of the electronic and nuclear coordinates with an ansatz

$$\Psi(\{\boldsymbol{r}_i\},\{\boldsymbol{R}_I\},t) \approx \Psi(\{\boldsymbol{r}_i\})\chi(\{\boldsymbol{R}_I\})\exp\left[\frac{i}{\hbar}\int_{t_0}^{t} dt^{'}\hat{E}_e(t^{'})\right],$$

with the electronic and nuclear wavefunctions normalized to unity at every instance of time. A phase factor is introduced to make the equations look nice:

$$\hat{E}_e = \int d\boldsymbol{r} d\boldsymbol{R} \Psi * \chi * \hat{H} \Psi \chi,$$

where the integration occurs over all spatial coordinates $\{ \boldsymbol{r}_i \}, \{ \boldsymbol{R}_I \}$. This is a single determinant ansatz which must lead to a mean-field description of the dynamics. This ansatz differs from the Born-Oppenheimer approximation:

$$\Psi_{BO} = \sum_{k=0}^{\infty} \Psi_k \chi_k,$$

even in the single-determinant limit where only a single state k is included in the expansion.

Inserting this ansatz into the Schrodinger equation reveals the following set of equations:

$$i\hbar \frac{\partial \Psi}{\partial t} = -\sum_i \frac{\hbar^2}{2m_e} \nabla_i^2 \Psi + \{ \int d\boldsymbol{R} \chi^* V_{ne} \chi \} \Psi,$$

$$i\hbar \frac{\partial \chi}{\partial t} = -\sum_i \frac{\hbar^2}{2m_e} \nabla_I^2 \chi + \{ \int d\boldsymbol{r} \Psi^* \hat{H} \Psi \} \chi.$$

This set of equations is the basis for the time-dependent self-consistent field (TDSCF) method, wherein particles move in time-dependent effective potentials obtained from quantum mechanical expectation values.

In the framework of classical molecular dynamics we approximate the nuclei as classical point particles. This can be done be rewriting the nuclear wavefunction as

$$\chi = A \exp[iS/\hbar],$$

with an amplitude factor A and a phase S which are both considered to be real. The TDSCF equations are rewritten in terms of these variables

$$\frac{\partial S}{\partial t} + \sum_I \frac{1}{2M_I} (\nabla_I S)^2 + \int d\boldsymbol{r} \Psi^* \hat{H} \Psi = \hbar^2 \sum_I \frac{1}{2M_I} \frac{\nabla_I^2 A}{A},$$

$$\frac{\partial A}{\partial t} + \sum_I \frac{1}{M_I}(\nabla_I A)(\nabla_I S) + \sum_I \frac{1}{2M_I} A(\nabla_I^2 S) = 0.$$

This set of equations is known as the "quantum fluid dynamical representation". The term for S contains a term for $\hbar$ which vanishes in the classical limit $\hbar \to 0$:

$$\frac{\partial S}{\partial t} + \sum_I \frac{1}{2M_I}(\nabla_I S)^2 + \int d\boldsymbol{r} \Psi^* \hat{H} \Psi = 0.$$

This formulation of the nuclear dynamics is isomorphic to the Hamilton-Jacobi formulation:

$$\frac{\partial S}{\partial t} + \hat{H} = 0,$$

with the classical Hamilton function

$$\hat{H} = T(P_I) + V(R_I),$$

with coordinates R_I and conjugate momenta P_I.

If we identify the conjugate momenta

$$\boldsymbol{P}_I = \nabla_I S,$$

we obtain the Newtonian equations of motion:

$$\begin{aligned} \frac{d\boldsymbol{P}}{dt} &= -\nabla_I V = -\nabla_I \int d\boldsymbol{r} \Psi^* \hat{H} \Psi \quad \text{or} \\ M_I \frac{d^2 \boldsymbol{R}_I}{dt^2} &= -\nabla_I \int d\boldsymbol{r} \Psi^* \hat{H} \Psi \\ &= -\nabla_I V_e^E(R_I(t)). \end{aligned} \tag{4.2}$$

The nuclei now move according to classical mechanics in an effective potential V_e^E generated by the electrons. This potential is a function only of the nuclear degrees of freedom at time t after averaging out the electronic degrees of freedom.

For consistency the nuclear wavefunction appearing in the TDSCF equation for the electronic degrees of freedom has to be replaced by the positions of the nuclei point particles. This is done by replacing the nuclear density

$|\chi|^2$ in the limit $\hbar \to 0$ by a product of delta functions $\prod_I \delta(\boldsymbol{R}_I - \boldsymbol{R}_I(t))$ centered at the instantaneous positions $\{ \boldsymbol{R}_I(t) \}$ of the classical nuclei. This leads to a time-dependent wave equation for the electrons:

$$i\hbar \frac{\partial \Psi}{\partial t} = -\sum_i \frac{\hbar}{2m_e} \nabla_i^2 \Psi + V_{ne} \Psi,$$

which evolve quantum mechanically as the nuclei propagate classically. This mixed approach is commonly referred to as *Ehrenfest molecular dynamics*. Although the underlying equations describe a mean-field theory, the Ehrenfest approach includes transitions between electronic states.

To arrive at a purely classical description of the dynamics of both the nuclei and the electrons we need to make further simplifications.
Firstly we restrict the electronic wave function Ψ to the ground state wave function Ψ_0 at every instant of time. This means the nuclei move on a single potential energy surface

$$V_e^E = \int d\boldsymbol{r} \Psi_0^* \hat{H} \Psi_0 = E_0(R_I),$$

that is determined by solving the Schrodinger equation

$$\hat{H} \Psi_0 = E_0 \Psi_0.$$

In this limit the Ehrenfest potential is identical to the ground state Born-Oppenheimer potential.

Since we are now only dealing with a single potential energy surface, the problem of computing the energy surface can be decoupled from computing the expactation values through equation (). First one produces an appropriate set of nuclear configurations by solving the time-independent Schrodinger equation. Second, these configurations are fitted to an analytical functional form to produce a global potential energy surface. Finally the Newtonian equations of motions are solved on this energy surface, producing a set of classical trajectories.

To deal with the large number of degrees of freedom as the number of nuclei in the system increases, the global potential energy surface is approximated as an expansion of manybody contributions:

$$V_e^E \approx V_e^{\text{approx}} = \sum_I v_1(R_I) + \sum_{I<J} v_2(R_I, R_J) + \sum_{I<J<K} v_3(R_I, R_J, R_K),$$

typically truncated at 2, 3 or 4-body interactions depending on the complexity of the molecules in the system.

This renders the problem of computing dynamics purely classical:

$$M_I \frac{d^2 R_I}{dt^2} = -\nabla_I V_e^{\text{approx}}.$$

4.0.2 Molecular dynamics simulations

Classical molecular dynamics is a method for computing equilibrium and transport properties of manybody systems obeying classical laws of motion. While a large number of simplifications have to be made in order to describe quantum mechanical systems classically, the approximation works surprisingly well except for atoms which are quite light (He, H^2) or for atoms with a vibrational energy which is substantially larger than the thermal energy of the system ($h\nu > k_B T$).

In order to calculate properties of the system they have to be expressed in terms of the positions and velocities of the constituent nuclei. For instance the temperature can be related to the average kinetic energy of the system:

$$\langle \frac{1}{2} m v^2 \rangle = \frac{N_f}{2} k_B T,$$

where N_f is the number of degrees of freedom in our system. At every instant of time the total kinetic energy of our system defines an instantaneous temperature, which has to averaged over a large number of timesteps in order to produce the equilibrium property. In practice, one is satisfied when the fluctuations in the instantaneous temperature appear reasonably small.

To run a molecular dynamics simulation one requires a set of initial conditions, i.e. a set of initial positions and velocities for every atom in the system. Typically the atoms are placed by replicating a unit cells a number of times in every dimension. A unit cell consists of a set of lattice vectors which define the placement of every atom in the unit cell. For instance the face-centered cubic cell (FCC) contains 4 atoms:

$$
\begin{aligned}
\boldsymbol{r}_1 &= (0, 0, 0) \\
\boldsymbol{r}_2 &= (\frac{b}{2}, \frac{b}{2}, 0) \\
\boldsymbol{r}_3 &= (0, \frac{b}{2}, \frac{b}{2}) \\
\boldsymbol{r}_4 &= (\frac{b}{2}, 0, \frac{b}{2}),
\end{aligned}
\tag{4.3}
$$

where b is known as the lattice constant and defines the size of the unit cells.

The velocities are typically initialized with a random uniform distribution or the Maxwell-Boltzmann distribution. The Maxwell-Boltzmann distribution is the one most often used, since the equilibrium distribution tends towards this distribution. The exact form however will differ from the one we started with.

Given these initial conditions, the system will not be in an equilibrium state at $t = 0$. To evolve the system to an equilibrium state one most commonly evolves the system for a number of timesteps until fluctuations in dynamic properties such as the total potential energy or the temperature settle down. Once we are in equilibrium we can start calculating thermodynamic averages.

As we mentioned before, the global energy surface is approximated as an expansion of manybody contributions:

$$
V_e^E(\{\,\boldsymbol{r}\,\}) = \sum_I v_1(R_I) + \sum_{I<J} v_2(R_I, R_J) + \sum_{I<J<K} v_3(R_I, R_J, R_K),
$$

wherein each n-body term is an analytical function of n coordinates. As an atom moves on the energy surface it feels a force which is the gradient of the potential energy surface. This means atom i feels an acceleration:

$$
\boldsymbol{F}_i = m_i \frac{d^2 \boldsymbol{r}_i}{dt^2} = -\nabla V_e^E(\boldsymbol{r}_i, \{\,\boldsymbol{r}\,\}).
$$

For a system of N atoms with only pairwise interactions this means the forces must be calculated $N(N-1)/2$ times for every timestep which means

we have a time complexity of order $\mathcal{O}(N^2)$. The force calculation is by far the most important part of any molecular dynamics simulation, and the most time consuming. A number of techniques are employed in order to reduce the time usage, perhaps the most common is the use of neighbor lists. Using neighbor lists, each atom carries a list of neighbors within a cut-off radius r_{cut} and interactions beyond this cut-off are neglected. This reduces the time-complexity to merely $\mathcal{O}(cN) = \mathcal{O}(N_{r_c} \cdot N)$, where N_{r_c} is the average number of neighbors in the system within a cutoff r_c. For a large system this can be a huge reduction in complexity, but the choice of cut-off can obviously massively impact the dynamics of the system.

In order to simulate the dynamics of a system governed by a conservative force $\boldsymbol{F} = -\nabla V_e^E$ we need to integrate the Newtonian equations of motion. The equations of motion are typically not solvable analytically, which means we require an effective numerical method for integration. Some important considerations for molecular dynamics are conservation of energy and accuracy for large time steps. The most common method used is the Velocity-Verlet algorithm. At any given time step t, the position $\boldsymbol{r}(t + \Delta t)$ and velocity $\boldsymbol{v}(t + \Delta t)$ at the next time step $t + \Delta t$ is calculated as:

$$\begin{aligned} \boldsymbol{r}(t + \Delta t) &= \boldsymbol{r}(t) + \boldsymbol{v}(t)\Delta t + \frac{1}{2}\boldsymbol{a}(t)\Delta t^2, \\ \boldsymbol{a}(t + \Delta t) &= -\frac{1}{m}\nabla V_e^E(\boldsymbol{r}(t + \Delta t)), \\ \boldsymbol{v}(t + \Delta t) &= \boldsymbol{v}(t) + \frac{1}{2}(\boldsymbol{a}(t) + \boldsymbol{a}(t + \Delta t))\Delta t. \end{aligned} \tag{4.4}$$

The error in the Velocity-Verlet method is of order $\mathcal{O}(\Delta t^2)$, which means that it is not particularly accurate for large time steps over a long time. However, the long term energy drift of the method is small, which is very desirable. It is also not very memory-intensive, which matters for simulating very large systems.

Molecular dynamics is usually performed within a cubic box of fixed volume $V = L_x \cdot L_y \cdot L_z$, where L_i is the length of the box in direction i. Molecular dynamics is typically limited by the number of particles we are able to simulate, of the order $10^6 - 10^8$, which means the size of the box is often decided by the desired density $\rho = N/V$. Since the number of particles is always much smaller than the number of particles in realistic systems, approximations are

required. Periodic boundary conditions can be applied to the box in order to approximate an infinite system. Typically particle coordinates are restricted to the simulation box, in pseudocode:

Algorithm 1 Continuity

if $x < -L_x/2$ **then**
    $x \mathrel{+}= L_x$
end if
if $x > L_x/2$ **then**
    $x \mathrel{-}= L_x$
end if

Distance and distance vectors between particles should also obey the minimum image convention:

Algorithm 2 Minimum image

$dx = x_j - x_i$
if $dx < -L_x/2$ **then**
    $dx \mathrel{+}= L_x$
end if
if $dx > L_x/2$ **then**
    $dx \mathrel{-}= L_x$
end if

These conditions should be applied in every dimension. This approach runs the risk of introducing nonphysical artifacts of the simulation, such as a macromolecule interacting with its own image, and for coulombic interactions the system must be charge neutral to avoid summing to an infinite charge. The optimal system size with periodic boundary conditions will therefore depend on the intended simulation length, the desired accuracy and the dynamics which are being studied.

Thus far we have discussed molecular dynamics for a system of N particles within a fixed volume V and constant energy E, i.e. in the microcanonical ensemble NVE. Other ensembles are also impossible, such as the canonical ensemble NVT and the isothermal-isobaric ensemble NPT.

Simulations in the canonical ensemble can be achieved by modifying the verlet integration algorithm. The simplest thermostat possible follows from the equipartion theorem:

$$T \propto \langle mv^2 \rangle,$$

meaning some amount of kinetic energy i.e. velocity can be added or subtracted to every atom in order to maintain a constant temperature at every timestep. Multiplying every velocity by a factor $\lambda = \sqrt{T_0/T(t)}$ where $T(t)$ is the instantaneous temperature and T_0 is the desired temperature will achieve desired effect. This approach significantly alters the trajectories of the system however, which means this thermostat should only be applied for adjusting the temperature of the system and not for taking ensemble averages in equilibrium.

The most common thermostat used is the Nosé-Hoover thermostat, which is one of the most accurate and efficient algorithms for achieving realistic constant-temperature conditions. [cite: https://engineering.ucsb.edu/~shell/che210d/Advanced_molecular_dynamics.pdf] Nosé introduced an extended Hamiltonian with two additional degrees of freedom:

- s - the position of an imaginary coupled heat reservoir
- p_s - the conjugate momentum of the heat reservoir

In addition it introduces an effective mass Q such that $p_s = Q\frac{ds}{dt}$.
Hoover modified Nosé's approach by introducing the Hamiltonian:

$$H = \frac{1}{2}\sum m_i \left|\boldsymbol{p}_i\right|^2 + U(\boldsymbol{r}) + \frac{\xi Q}{2} + 3Nk_BT\ln s,$$

where ξ is a friction coefficient and $\boldsymbol{p}_i = m_i\boldsymbol{v}_i \times s$ are the particle momenta. This leads to a new set of Newtonian equations of motions with an additional force that is proportional to the velocity:

$$\begin{aligned}
\frac{d\boldsymbol{r}_i}{dt} &= \boldsymbol{v}_i \\
\frac{d\boldsymbol{v}_i}{dt} &= -\frac{1}{m_i}\frac{\partial U(\boldsymbol{r})}{\partial \boldsymbol{r}_i} - \xi\boldsymbol{v}_i \\
\frac{d\xi}{dt} &= \left(\sum m_i \left|\boldsymbol{v}_i\right|^2 - 3Nk_bT\right)/Q \\
\frac{d\ln s}{dt} &= \xi.
\end{aligned} \tag{4.5}$$

These can then be solved with a numerical integration scheme such as the velocity-Verlet algorithm.

4.0.3 Molecular dynamics potentials

The dynamics of an ensemble of particles is governed by their interactions. In molecular dynamics we stipulate that the interactions are decided only by the relative positions of the particles, i.e. only conservative forces act on the atoms. This means that the force on atom i is fully described by a potential energy U:

$$\boldsymbol{F}_i = -\nabla U(\boldsymbol{r}_i),$$

which in principle depends on the position of atom i and all other atoms in the system. Finding an appropriate potential for the system of atoms which you intend to study can be ardous work, and usually involves fitting an analytical functional form with a large set of parameters to a potential energy surface from ab initio quantum mechanical calculations.

Potentials can be classified as either bonded or non-bonded. Bonded potentials compute the interactions for a predefined set of atoms and molecules in the simulations, while non-bonded potentials compute the interactions between *all* pairs, triplets etc. of atoms (usually within a certain radius). Larger, more complex systems typically contain a mix of bonded and non-bonded potentials, for example a system of rigid water molecules interacting with a surface of carbon atoms.

One of the simplest potentials meant to simulate a realistic system is the Lennard-Jones potential:

$$U(r_{ij}) = 4\epsilon \left(\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right),$$

with $r_{ij} = |\boldsymbol{r}_j - \boldsymbol{r}_i|$. For the Lennard-Jones potential there are only two parameters to be decided, a characteristic energy ϵ and a characteristic length σ. This potential is meant to emulate the relatively weak interactions between noble gas atoms such as Argon. It can be separated into two terms:

- $-\left(\frac{\sigma}{r}\right)^6$ - owing to the long-term attraction from van der Waals interactions
- $+\left(\frac{\sigma}{r}\right)^{12}$ - owing to the short-term repulsion from the Pauli principle

While the van der Waals term is justified by theory, the repulsion term is justified by numerical efficiency - as it contains the square of the van der Waals term - and because it models the Pauli repulsion accurately.

While the Lennard-Jones potential is very simple, requiring only two parameters to be determined, it has shown to be effective at modelling noble gas atoms and is commonly used as a building block for more complicated interactions.

[cite http://www.pages.drexel.edu/~cfa22/msim/node41.html] Another simple and common potential is the Stillinger-Weber potential, which is meant to model the interactions between silicon atoms. Silicon forms tetrahedral bonded structures as well as pairwise interactions which means the potential includes a twobody and a threebody interaction:

$$U = \sum_{i<j} v_2(r_{ij}) + \sum_{i<j<k} v_3(\boldsymbol{r}_i, \boldsymbol{r}_j, \boldsymbol{r}_k). \tag{4.6}$$

The twobody interaction models the pairwise interaction:

$$v_2(r) = \begin{cases} \epsilon A(Br^{-p} - r^{-q}) \exp\left[(r-a)^{-1}\right], & r < a \\ 0 & r \geq a \end{cases} \tag{4.7}$$

This twobody term resembles the Lennard-Jones potential, but with an exponential cutoff. The threebody term models the tetrahedral angles, and is a sum over three triplets:

$$v_3(\boldsymbol{r}_i, \boldsymbol{r}_j, \boldsymbol{r}_k) = h_{jik} + h_{ijk} + h_{ikj}, \tag{4.8}$$

with the angular interaction $h_{jik} = h(r_{ij}, r_{ik}, \theta_{jik})$ and

$$h_{jik} = \begin{cases} \epsilon\lambda \exp\left[\dfrac{\gamma}{r_{ij}-a} + \dfrac{\gamma}{r_{ik}-a}\right] \left(\cos\theta_{jik} - \cos\theta^0_{jik}\right)^2 & r_{ij} < a \\ 0 & r_{ij} \geq a \end{cases} \tag{4.9}$$

where θ^0_{jik} is an "equilibrium" angle. The terms $\epsilon, A, B, p, q, \lambda, \gamma$ are parameters to be decided and a is a cutoff radius.

Chapter 5

Machine learning

Machine learning is the study of algorithms and statistical models employed by computing systems capable of performing tasks without explicit instruction. While traditional algorithms rely on some specified input and a ruleset for determining the output, machine learning is instead concerned with a set of generic algorithms which can find patterns in a broad class of data sets. This section will give a brief overview of machine learning, and more specifically the class of algorithms known as neural networks, and will follow closely the review by [Mehta et. al.] which the reader is encouraged to seek out.

Examples of machine learning problems include identifying objects in images, transcribing text from audio and making film recommendations to viewers based on their watch history. Machine learning problems are often subdivided into estimation and prediction problems. In both cases, we choose some observable $\boldsymbol{x}$ (e.g. the period of a pendulum) related to some parameters $\boldsymbol{\theta}$ (e.g. the length and the gravitational constant) through a model $p(\boldsymbol{x}|\boldsymbol{\theta})$ that describes the probability of observing $\boldsymbol{x}$ given $\boldsymbol{\theta}$. Subsequently we perform an experiment to obtain a dataset $\boldsymbol{X}$ and use these data to fit the model. Fitting the model means finding the parameters $\hat{\boldsymbol{\theta}}$ that provide the best explanation for the data. *Estimation* problems are concerned with the accuracy of $\hat{\boldsymbol{\theta}}$, whereas prediction problems are concerned with the ability of the model $p(\boldsymbol{x}|\boldsymbol{\theta})$ to make new predictions. Physics has traditionally been more concerned with the estimation of model parameters, while in this thesis we will be focused on the accuracy of the model.

Many problems in machine learning are defined by the same set of ingredients. The first is the dataset $\mathcal{D} = (\boldsymbol{X}, \boldsymbol{Y})$, where $\boldsymbol{X}$ is a matrix containing observations of the independent variables $\boldsymbol{x}$, and $\boldsymbol{Y}$ is a matrix containing

observations of dependent variables. Second is a model $\boldsymbol{F} : \boldsymbol{x} \to \boldsymbol{y}$ which is a function of the parameters $\boldsymbol{\theta}$. Finally we have a cost function $\mathcal{C}\left(\boldsymbol{Y}, \boldsymbol{F}\left(\boldsymbol{X}; \boldsymbol{\theta}\right)\right)$ that judges the performance of our model at generating predictions.
In the case of linear regression we consider a set of independent observations $\boldsymbol{X} = \begin{bmatrix}\boldsymbol{x}_1 & \boldsymbol{x}_2 & \dots & \boldsymbol{x}_N\end{bmatrix}$ related to a set of dependent observations $\boldsymbol{y} = (y_1, y_2, \dots, y_N)$ through a linear model $f(\boldsymbol{x}; \boldsymbol{\theta}) = x_1 \cdot w_1 + x_2 \cdot w_2 + \dots + x_P \cdot w_P$, with parameters $\boldsymbol{\theta} = (w_1, w_2, \dots, w_P)$. The cost function is the well known sum of least squares $\mathcal{C}(\boldsymbol{y}, f(\boldsymbol{X}; \boldsymbol{\theta})) = \sum_i^N (y_i - f(\boldsymbol{x}_i; \boldsymbol{\theta}))^2$ and the best fit is chosen as the set of parameters which minimize this cost function: $\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta}}{\operatorname{argmin}}\, \mathcal{C}(\boldsymbol{Y}, f(\boldsymbol{X}; \boldsymbol{\theta}))$.

5.0.1 Basics of statistical learning

Statistical learning theory is a field of statistics dealing with the problem of making predictions from data. We begin with an unknown function $y = f(x)$ and our goal is to develop a function $h(x)$ such that $h \sim f$. We fix a hypothesis set $\mathcal{H}$ that the algorithm is willing to consider. The expected error of a particular h over all possible inputs x and outputs y is:

$$E[h] = \int_{X \times Y} \mathcal{C}(h(x), y) \rho(x, y) dx dy,$$

where $\mathcal{C}$ is a cost function and $\rho(x, y)$ is the joint probability distribution for x and y. This is known as the *expected error*. Since this is impossible to compute without knowledge of the probability distribution ρ, we instead turn to the *empirical error*. Given n data points the empirical error is given as:

$$E_S[h] = \frac{1}{n} \sum_i^n \mathcal{C}(h(x_i), y_i).$$

The *generalization error* is defined as the difference between the expected and empirical errors:

$$G = E[h] - E_S[h].$$

We say an algorithm is able to learn from data or *generalize* if

$$\lim_{n \to \infty} G = 0.$$

We are in general unable to compute the expected error, and therefore unable to compute the generalization error. The most common approach

known as *cross-validation* is to estimate the generalization error by subdividing our dataset into a *training* set and a *test* set. The value of the cost function on the training set is called the *in-sample* error and the value of the cost function on the test set the *out-of-sample* error. Assuming the dataset is sufficiently large and representative of f, and the subsampling into train and test datasets is unbiased, the in-sample error can serve as an appropriate proxy for the generalization error.
In figure 5.1 we show the typical evolution of the errors as the number of data points increase. It is assumed that the function being learned is sufficiently complicated that we cannot learn it exactly, and that we have a sizeable number of data points available. The in-sample error will decrease monotonically, as our model is not able to learn the underlying data exactly. In contrast, the out-of-sample error will decrease, as the sampling noise decreases and the training data set becomes more representative of the underlying probability distribution. In the limit, these errors both approach same value, which is known the model *bias*. The bias represents the best our model could do in the infinite data limit. The out-of-sample error produced from the sampling noise is known as *variance*, and will vanish completely given an infinite representative data set.

In figure 5.2 we show the typical evolution of the out-of-sample error as the model *complexity* increases. Model complexity is a measure of the degrees of freedom in the model space, for example the number of coefficients in a polynomial regression. In the figure we can see that bias decreases monotonically as model complexity increases, as the model is able to fit a larger space of functions. However, the variance will also increase as the model becomes more susceptible to sampling noise. In general the lowest out-of-sample error, and therefore generalization error, is achieved at an intermediate model complexity. We also find that as model complexity increases, a larger amount of data points is required to be able to reasonably fit the true function.

5.0.2 Bias-variance decomposition

Consider a dataset $\mathcal{D}(\boldsymbol{X}, \boldsymbol{y})$ of n pairs of independent and dependent variables. Assume the true data is generated from a noisy model:

$$y = f(\boldsymbol{x}) + \epsilon,$$

where ϵ is normally distributed with mean μ and standard deviation σ. Assume that we have an estimator $h(\boldsymbol{x}; \boldsymbol{\theta})$ trained by minimizing a cost

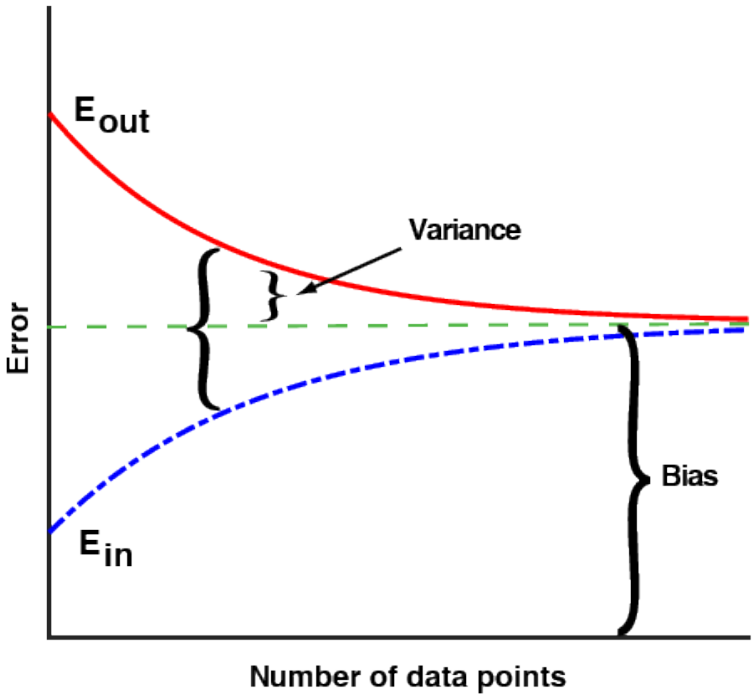

Figure 5.1: Typical in-sample and out-of-sample error as a function of the number of data points. It is assumed that the number of data points is not small, and that the true function cannot be exactly fit.

function $\mathcal{C}(\boldsymbol{y}, h(\boldsymbol{x}))$ which we take to be the sum of squared errors:

$$\mathcal{C}(\boldsymbol{y}, h(\boldsymbol{x})) = \sum_{i}^{n} (y_i - h(\boldsymbol{x}_i; \boldsymbol{\theta}))^2.$$

Our best estimate for the model parameters:

$$\boldsymbol{\theta}_{\mathcal{D}} = \arg\min_{\boldsymbol{\theta}} \mathcal{C}(\boldsymbol{y}, h(\boldsymbol{x}; \boldsymbol{\theta})),$$

is a function of the dataset $\mathcal{D}$. If we imagine we have a set of datasets $\mathcal{D}_j = (\boldsymbol{y}_j, \boldsymbol{X}_j)$, each with n samples, we would like to calculate the expectation value of the cost function over all these datasets $E_{\mathcal{D},\epsilon}$. We would also like to calculate the expectation value over different instances of the noise ϵ. The

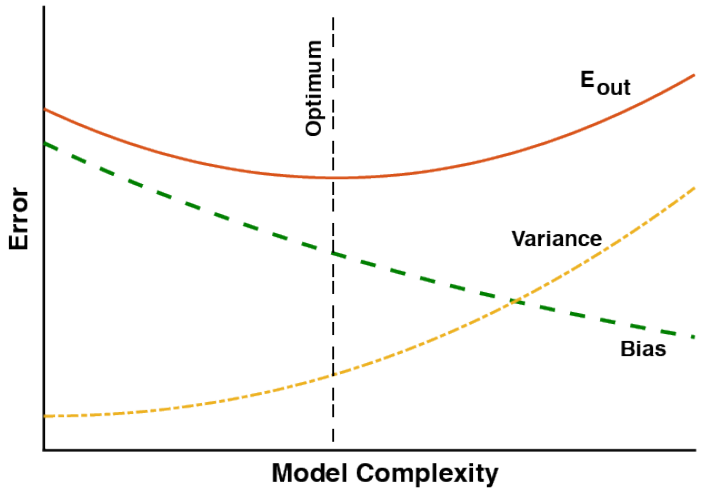

Figure 5.2: Typical out-of-sample error as a function of model complexity for a fixed dataset. Bias decreases monotonically with model complexity, while variance increases as a result of sampling noise.

expected generalization error can be decomposed as:

$$\begin{aligned} E_{\mathcal{D},\epsilon}[\mathcal{C}(\boldsymbol{y}, h(\boldsymbol{X};\boldsymbol{\theta}_{\mathcal{D}}))] &= E\left[\sum_i (y_i - h(\boldsymbol{x}_i;\boldsymbol{\theta}_{\mathcal{D}}))^2\right] \\ &= \sum_i \sigma_\epsilon^2 + E_{\mathcal{D}}[(f(\boldsymbol{x}_i) - f(\boldsymbol{x}_i;\boldsymbol{\theta}_{\mathcal{D}}))^2]. \end{aligned} \tag{5.1}$$

The second term can be further decomposed as

$$\begin{aligned} &E_{\mathcal{D}}[(f(\boldsymbol{x}_i) - f(\boldsymbol{x}_i;\boldsymbol{\theta}_{\mathcal{D}}))^2] \\ &= (f(\boldsymbol{x}_i) - E_{\mathcal{D}}[h(\boldsymbol{x}_i;\boldsymbol{\theta}_{\mathcal{D}})])^2 + E[(h(\boldsymbol{x}_i;\boldsymbol{\theta}_{\mathcal{D}}) - E[h(\boldsymbol{x}_i;\boldsymbol{\theta}_{\mathcal{D}}])^2] \end{aligned} \tag{5.2}$$

The first term is what we have referred to as the bias:

$$\text{Bias}^2 = \sum_i (f(\boldsymbol{x}_i) - E_{\mathcal{D}}[h(\boldsymbol{x}_i;\boldsymbol{\theta}_{\mathcal{D}})])^2. \tag{5.3}$$

The bias measures the expectation value of the deviation of our model from the true function, i.e. the best we can do in the infinite data limit.

The second term is what we have referred to as the variance:

$$\text{Var} = \sum_i E[(h(\boldsymbol{x}_i; \boldsymbol{\theta}_{\mathcal{D}}) - E[h(\boldsymbol{x}_i; \boldsymbol{\theta}_{\mathcal{D}}])^2] \tag{5.4}$$

The variance measures the deviation of our model due to finite-sampling effects. Combining these effects we can decompose the out-of-sample error into:

$$E_{\text{out}} = \text{Bias}^2 + \text{Var} + \text{Noise}, \tag{5.5}$$

with Noise $= \sum_i \sigma_\epsilon^2$.
In general it can be much more difficult to obtain sufficient good data than to train a very complex model. Therefore it is often useful in practice to use a less complex model with higher bias, because it is less susceptible to finite-sampling effects.

5.0.3 Neural networks

Artificial Neural Networks (ANN) or Deep Neural Networks (DNN) are supervised learning models vaguely inspired by biological neural networks. The building blocks of neural networks are neurons that take a vector input of d features $\boldsymbol{x} = (x_1, \ldots, x_d)$ and produce a scalar output $a(\boldsymbol{x})$. A neural networks consists of layers of these neurons stacked together with the output of one layer serving as input for another. The first layer is typically known as the *input layer*, the middle layers as *hidden layers* and the final layer the *output layer*. The basic architecture is shown in figure 5.3. In almost all cases the output $a_i(\boldsymbol{x})$ of neuron i can be decomposed into a linear operation on the inputs passed through a non-linear activation function:

$$a_i(\boldsymbol{x}) = \sigma_i(z_i),$$

where σ_i is a non-linear function and z_i is the dot product between the inputs $\boldsymbol{x}$ and a set of neuron-specific weights $\boldsymbol{w}_i$:

$$z_i = \boldsymbol{x}^T \boldsymbol{w}_i + b_i.$$

The term b_i is a neuron-specific re-centering of the input.
Typical choices of non-linearities/activation functions include the sigmoid and hyperbolic tangent functions, and Rectified Linear Units (ReLU). When the activation function is non-linear, the neural network with a single hidden layer

can be proven to be a *universal function approximator*, given an arbitrarily large number of neurons. We typically also want functions that are monotonic and smooth with a monotonic derivative.

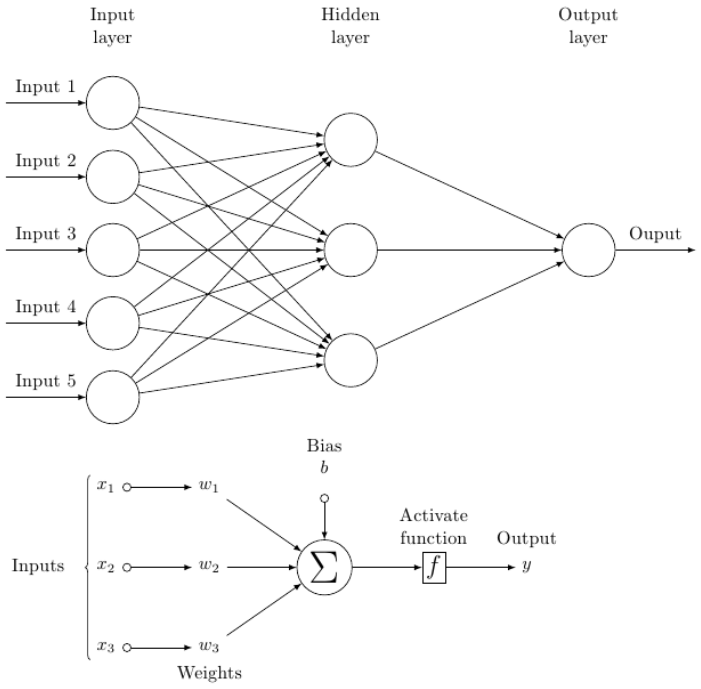

Figure 5.3: Text.

The simplest neural networks are known as *feed-forward* neural networks (FNN). The input layer is the vector $\boldsymbol{x}$ of inputs, while each neuron in the first hidden layer performs a dot product between its weights $\boldsymbol{w}_i$ and the inputs and passes it through a non-linearity σ_i. The activation function is typically shared across one or multiple layers $\sigma_i = \sigma$. The vector of neuron outputs $\boldsymbol{a}_i$ serves as input to the next hidden layer until we reach the final layer. In the final layer the choice of activation function is dependent on the problem we are trying to solve. If we are performing non-linear regression the final activation function is often the identity $\sigma_i(z) = z$, or if we are doing classification the soft-max function is often employed.

Let $\boldsymbol{x}$ be a vector of $d = 1, \ldots, D$ inputs or *features*. Let $a_i^{(h)}$ denote the output of neuron $i = 1, \ldots, N_d$ in layer $h = 1, \ldots, H$. The output of neuron i in the first hidden layer $a_i^{(1)}$ is thus:

$$z_i^{(1)} = \boldsymbol{x}^T \boldsymbol{w}_i^{(1)} + b_i^{(1)}.$$

$$a_i^{(1)} = \sigma_i^{(1)}(z_i^{(1)}),$$

The inputs are iterated through each hidden layer until we reach the final layer:

$$\begin{aligned} z_i^{(H)} &= (\boldsymbol{a}_{H-1})^T \boldsymbol{w}_i^{(H)} + b_i^{(H)}. \\ a_i^{(H)} = o_i &= \sigma_i^{(H)}(z_i^{(H)}) \\ &= \sigma_i^{(H)} \left((\boldsymbol{a}_{H-1})^T \boldsymbol{w}_i^{(H)} + b_i^{(H)} \right) \\ &= \sigma_i^{(H)} \left(\left((\sigma_1^{(H-1)}, \ldots, \sigma_{N_{H-1}}^{(H-1)}) \right)^T \boldsymbol{w}_i^{(H)} + b_i^{(H)} \right). \end{aligned} \tag{5.6}$$

This allows us to compose a complicated function $\boldsymbol{F} : \mathbb{R}^D \to \mathbb{R}^O$, with D the number of inputs and O the number of outputs. The *universal approximation theorem* tells us that this simple architecture can approximate any of a large set of continuous functions given appropriate choice of weights $\boldsymbol{w}_i^h$ and mild assumptions on the activation functions. The theorem requires only a single hidden layer, where the strength of the approximation relies on the number of neurons. In practice it has been found that adding more layers produces faster convergence and higher accuracy, which has given rise to the field of *deep learning*.

5.0.4 Backpropagation

Given a set of datapoints $(\boldsymbol{x}_i, y_i),\ i = 1, \ldots, n$, the value of the cost function is entirely determined by the weights and biases of each neuron in the network. We define learning narrowly as adjusting the parameters of the network in order to minimize the cost function.
Gradient descent is a simple, but powerful method of finding the minima of differentiable functions. Given a function $F : \mathbb{R}^d \to \mathbb{R}$, and an initial value $\boldsymbol{x}_0$ we define an iterative procedure:

$$\boldsymbol{x}_{n+1} = \boldsymbol{x}_n - \eta \nabla F(\boldsymbol{x}_n),$$

where η is known as the *learning rate*. The procedure terminates when the norm $|\nabla F(\boldsymbol{x}_n)|$ or $|x_{n+1} - x_n|$ is appropriately small.
The learning rate is not necessarily fixed throughout the procedure, and proves crucial to the convergence of the method. If f is convex, and η is reasonably small, convergence is guaranteed. Convergence may be very slow however, and if f is not convex you are only guaranteed to find local minima, and this makes the method very sensitive to initial conditions.

In order to train the model, we need to calculate the derivative of the cost function with respect to a very large number of parameters multiple times. However, numerical calculation of gradients is very time consuming. The *backpropagation* algorithm is a clever use of the chain rule that allows us to calculate gradients efficiently.

Assume that there are L layers in our network with $l = 1, 2, ..., L$ indexing the layers, including the output layer and all the hidden layers. Let w_{ij}^l denote the weight for the connection from the i-th neuron in layer $l-1$ to the j-th neuron in layer l. Let b_j^l denote the bias of this j-th neuron.

The activation a_j^l of the j-th neuron in the l-th layer is related to the activities of the neurons in the layer $l-1$ by:

$$a_j^l = f\left(\sum_i w_{ij}^l a_i^{l-1} + b_j^l\right) = f\left(z_j^l\right),$$

where f is some activation function.

The cost function $\mathcal{C}$ depends directly on the activations in the output layer, and indirectly on the activations in all the lower layers. Define the error Δ_j^L of the j-th neuron in the L-th (final) layer as the change in cost function with respect to the weighted input z_j^L:

$$\Delta_j^L = \frac{\partial \mathcal{C}}{\partial z_j^L}.$$

Define analogously the error Δ_j^l of neuron j in the l-th layer as the change in cost function with respect to the weighted input z_j^l:

$$\Delta_j^l = \frac{\partial \mathcal{C}}{\partial z_j^l}.$$

This can also be interpreted as the change in cost function with respect to the bias b_j^l:

$$\Delta_j^l = \frac{\partial \mathcal{C}}{\partial z_j^l} = \frac{\partial \mathcal{C}}{\partial b_j^l}\frac{\partial b_j^l}{\partial z_j^l} = \frac{\partial \mathcal{C}}{\partial b_j^l},$$

since $\partial b_l^j / \partial z_j^l = 1$.

The error depends on neurons in layer l only through the activation of neurons in layer $l+1$, so using the chain rule we can write:

$$\begin{aligned}
\Delta_j^l = \frac{\partial \mathcal{C}}{\partial z_j^l} &= \sum_i \frac{\partial \mathcal{C}}{\partial z_i^{l+1}}\frac{\partial z_i^{l+1}}{\partial z_j^l} \\
&= \sum_i \Delta_i^{l+1}\frac{\partial z_i^{l+1}}{\partial z_j^l} \\
&= \sum_i \Delta_i^{l+1} w_{ij}^{l+1} f'(z_j^l) \\
&= \left(\sum_i \Delta_i^{l+1} w_{ij}^{l+1}\right) f'(z_j^l).
\end{aligned} \tag{5.7}$$

The sum comes from the fact that any error in neuron j in the l-th layer propagates to all the neurons in the layer $l+1$, so we have to sum up these errors.

This gives us the equations we need to update the weights and biases of our network:

$$\frac{\partial \mathcal{C}}{\partial w_{ij}^l} = \frac{\partial \mathcal{C}}{\partial z_j^l}\frac{\partial z_j^l}{\partial w_{ij}^l} = \Delta_j^l a_i^{l-1}.$$

$$\frac{\partial \mathcal{C}}{\partial b_j^l} = \Delta_j^l.$$

Now, if we have the error of every neuron j at the output layer, Δ_j^L, equation **??** gives us the recipe for calculating the error in the preceding layer until we reach the first hidden layer, and we are done. All we are missing is the error at the output layer:

$$\Delta_j^L = \frac{\partial \mathcal{C}}{\partial z_j^L} = \frac{\partial}{\partial z_j^L}\left(-\sum_c y_c \log f(z_c^L)\right),$$

where f is the softmax function. Taking the derivative of each term:

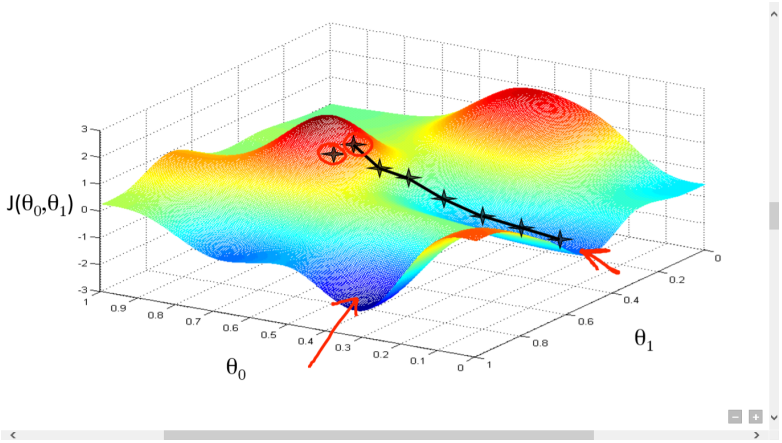

Figure 5.4: Text.

$$\frac{\partial \log f(z_c^L)}{\partial z_j^L} = \frac{\partial}{\partial z_j^L}\left(z_c^L - \log\left(\sum_c \exp\left(z_c^L\right)\right)\right) = \delta_{jc} - f(z_j^L),$$

where δ_{jc} is the Kronecker-Delta. This gives us the final expression we need:

$$\begin{aligned}
\Delta_j^L &= -\sum_c y_c\left(\delta_{jc} - f(z_j^L)\right) \\
&= -\sum_c y_c\delta_{jc} + \sum_c y_c f(z_j^L) \\
&= -y_j + f(z_j^L)\sum_c y_c \\
&= f(z_j^L) - y_j \\
&= \hat{y}_j - y_j.
\end{aligned} \tag{5.8}$$

5.0.5 Optimization

Gradient descent, momentum, ADAM, batching, normalization.

In order to begin the minimization procedure and find an optimal set of weights and biases for our network we first need some initial values. Often

weights are initialized with small values distributed around zero, drawn from a uniform or normal distribution. The bias can be initalized to zero, but enforcing that all biases have some small value ensures that every neuron has output which can be backpropagated in the first training cycle. In the Pytorch neural network framework - which is the one we will be using - weights are initialized uniformly as

$$\begin{aligned} \sigma &= 1/N_{in} \\ w &= \mathcal{U}(-\sigma, \sigma) \end{aligned} \tag{5.9}$$

where N_{in} is the number of inputs to the weight/layer of weights and $\mathcal{U}$ is the uniform distribution.

A critical choice for building neural networks is the choice of activation function. As we have mentioned briefly above, we have a small set of requirements in order for the neural network to be an universal function approximator, namely that the function be nonconstant, bounded, continuous and monotonically increasing. We also desire that the function be fast to evaluate, with a derivative that is simple to calculate.

A function that fulfills all of these criteria is the sigmoid function:

$$\begin{aligned} \sigma(x) &= \frac{1}{1+e^{-x}}, \\ \sigma'(x) &= \sigma(x)(1-\sigma(x)). \end{aligned} \tag{5.10}$$

The sigmoid function is defined on the entire real number line, and outputs a number between 0 and 1. It also has a continuous derivative which is simple to calculate. The sigmoid function is well suited for binary classifiers, since it easily creates a decision boundary between two categories 0 and 1, and is often the first goto for AI programmers.

Another simple and commonly used activation function is the hyperbolic tangent

$$\begin{aligned} \tanh(x) &= \frac{e^{2x}-1}{e^{2x}-1}, \\ \tanh'(x) &= (1-\tanh(x)^2). \end{aligned} \tag{5.11}$$

The hyperbolic tangent is defined on the real number line and outputs a number between -1 and 1.

Both the sigmoid and the hyperbolic tangent functions exhibit the problem of vanishing gradients. In deep learning, you typically have several layers of neurons outputting some linear combination of the activation functions. Through the backpropagation algorithm, each neuron receives a weight update proportional to the partial derivative of the loss with respect to its weights. For the sigmoid and hyperbolic tangent activations these gradients will be in the range $(-1, 1)$, and therefore often exceedingly small. This means that many neurons may receive effectively no weight update in any given training epoch, which severely impedes training.

The Rectified Linear Unit (ReLU) has gained popularity in recent years for its effectiveness in the field of convolutional neural networks. The ReLu function is defined as:

$$\begin{aligned} \mathrm{ReLU}(x) &= \begin{cases} x & x \geq 0 \\ 0 & x < 0 \end{cases} \\ \mathrm{ReLU}'(x) &= \begin{cases} 1 & x \geq 0 \\ 0 & x < 0 \end{cases} \end{aligned} \tag{5.12}$$

Since the ReLU function in principle has no upper bound, it does not suffer the effect of vanishing gradients. Instead, the ReLU function can produce exploding gradients, if one or more activations becomes very large. In practice the gradients are often "clipped", i.e. truncated above a certain upper bound. With the ReLU function one can also encounter "dying" neurons, since the input to any neuron that does not exceed zero is set to zero, which means that the number of neurons receiving an effective weight update

From the point of view of the neural network, the inputs are just dimensionless numbers, which are passed through layers until it spits out some outputs. It is therefore often useful to standardize the inputs. A common method is to shift the inputs by their mean and normalize by their standard deviation:

$$x_i' = \frac{x_i - \bar{x}}{\sigma_x}.$$

This is often useful in speeding up the training of neural networks. In theory any shifting and rescaling can be reproduced simply by updating the weights and biases of the network. In practice however, as the weights often

start as small numbers, it can take quite a few training cycles before the weights are of appropriate size. Standardizing the inputs also makes it impose to impose a penalty on the weights in order to reduce overfitting.

The most common methods for training neural networks are variations on the simple gradient descent scheme as mentioned above:

$$\boldsymbol{x}_{n+1} = \boldsymbol{x}_n - \eta \nabla F(\boldsymbol{x}_n).$$

Since we are interested in training neural networks, $\boldsymbol{x}_n$ represents the weights and biases of the network after n training cycles, η is our learning rate and $\nabla F(\boldsymbol{x}_n)$ is the gradient of the loss function.

The learning rate η controls how fast we reach a given minimum. A small learning rate will require more training cycles and a larger learning rate will require fewer. The learning rate also controls the numerical stability of the method, if it is too large the weights may diverge and if it is too small the weight may never converge. Gradient descent also tends to exhibit oscillating behaviour around the minimum. Often a schedule is imposed on the learning rate, such as reducing it by a fixed amount every few epochs or incorporating an exponentially decaying learning rate. [cite: Mehta et al] As discussed in Mehta et. al. using gradient descent to optimize our neural network imposes some limitations:

- *Gradient descent finds local minima* - if the GD algorithm converges, it will converge to a local minimum. This can lead to poor performance in complicated cost function landscapes
- *Gradients are expensive to compute* - the loss function often includes a term for each data point, which means the gradient also does.
- *GD is sensitive to learning rate* - as mentioned above, GD is very sensitive to learning rate
- *GD is sensitive to initial conditions* - gradient descent can take two different initial values and converge at two drastically different values
- *GD does not take curvature into account* - the learning rate for gradient descent is the same in all directions of parameter space, which means the maximum learning rate is set by the behaviour of the steepest direction

A common method for ameliorating some of these concerns is minibatching the inputs. Instead of calculating the gradient on the entire dataset for every epoch, we instead calculate the gradient on a subset of the data called a minibatch. If there are n datapoints and a minibatch size of m the total number of batches is n/m. If we denote each minibatch $b_k,\ k = 1, 2, \ldots, n/m$ the gradient becomes:

$$\nabla \mathcal{C} = \frac{1}{n} \sum_{i=1}^{N} \nabla \mathcal{L}_i \rightarrow \frac{1}{m} \sum_{i \in b_k} \nabla \mathcal{L}_i,$$

i.e. instead of averaging the loss over the entire dataset we instead average over a minibatch. This significantly speeds up the calculation, since we do not use the entire dataset for every update. In addition, introducing stochasticity in the division of the dataset into minibatches introduces stochasticity to the weights update, which should help gradient descent in overcoming local minima.

It is also common to introduce regularization to the weights and biases of the networks. Regularization in the context of neural networks means adding a term proportional to the L_p norm of the weights and biases. For example using the L2-norm our cost function becomes:

$$\nabla \mathcal{C} = \frac{1}{n} \sum_{i} \nabla \mathcal{L}_i \rightarrow \frac{1}{n} \sum_{i} \nabla \mathcal{L}_i + \lambda \left|\left|\boldsymbol{w}\right|\right|^2 = \frac{1}{n} \sum_{i} \nabla \mathcal{L}_i + \lambda \sum_{ij} w_{ij}^2,$$

i.e. we sum up all the weights squared. The parameter λ is known as a regularization parameter. This extra term adds a penalty on the size of the weights dependant on the regularization parameter. This has been shown to reduce overfitting, as the weights cannot be adjusted to arbitrary size in order to fit the training dataset.

Gradient descent is often paired with a momentum parameter that serves as a "memory" of the direction we are moving. This can be implemented as a modification to the gradient descent update:

$$\begin{aligned} \boldsymbol{v}_n &= \gamma \boldsymbol{v}_{n-1} + \eta \nabla F(\boldsymbol{x}_n), \\ \boldsymbol{x}_{n+1} &= \boldsymbol{x}_n - \boldsymbol{v}_n, \end{aligned} \tag{5.13}$$

where $0 < \gamma < 1$ is a memory parameter that controls the time scale for the memory. It is termed a momentum parameter because the equations for

updating the parameters $\boldsymbol{x}$ are analogous to the equations of motion for a particle moving in a viscous medium. Momentum helps the gradient descent algorithm gain speed in directions where the curvature is flat while dampening speed in high curvature directions.

First-order gradient descent methods differ from quasi-Newton methods in that they do not keep track of the curvature which is encoded in the so-called Hessian matrix of second order derivatives. Second-order methods accomplish this by calculating or approximating the Hessian and normalizing the learning rate by the curvature.

The RMSprop algorithm keeps a running average of both the first and second moment of the gradient. If we term the gradient as $\boldsymbol{g} = \nabla F(\boldsymbol{x})$ and the first and second orders as $\boldsymbol{m} = \mathbb{E}[\boldsymbol{g}]$ and $\boldsymbol{s} = \mathbb{E}[\boldsymbol{g}^2]$ we can write the update rule as:

$$\begin{aligned} \boldsymbol{g}_n &= \nabla F(\boldsymbol{x}_n) \\ \boldsymbol{s}_n &= \beta \boldsymbol{s}_{n-1} + (1-\beta)\boldsymbol{g}_n^2 \\ \boldsymbol{x}_{n+1} &= \boldsymbol{x}_n - \eta \frac{\boldsymbol{g}_n}{\sqrt{\boldsymbol{s}_n + \epsilon}} \end{aligned} \tag{5.14}$$

where β is a parameter which controls the time scale of the averaging and ϵ is a small regularization constant to prevent divergences. Operations involving vectors are understood to be element-wise. This results in a dampening of the learning rate in directions where the gradient is consistently large, and allows us to use a larger learning rate for areas of flat curvature.

Perhaps the most commonly used optimizer today is the closely related ADAM optimizer. In addition to keeping a running average of the first and second-order moments ADAM performs an additional bias correction to account for the fact that we are estimating first and second order moments using a running average. The update rule is given as:

$$
\begin{aligned}
\boldsymbol{g}_n &= \nabla F(\boldsymbol{x}_n) \\
\boldsymbol{m}_n &= \beta_1 \boldsymbol{m}_{n-1} + (1-\beta_1)\boldsymbol{g}_n \\
\boldsymbol{s}_n &= \beta_2 \boldsymbol{s}_{n-1} + (1-\beta_2)\boldsymbol{g}_n^2 \\
\hat{\boldsymbol{m}}_n &= \frac{\boldsymbol{m}_n}{1-\beta_1^n} \\
\hat{\boldsymbol{s}}_n &= \frac{\boldsymbol{s}_n}{1-\beta_2^n} \\
\boldsymbol{x}_{n+1} &= \boldsymbol{x}_n - \eta \frac{\hat{\boldsymbol{m}}_n}{\sqrt{\hat{\boldsymbol{s}}_n} + \epsilon},
\end{aligned}
\tag{5.15}
$$

with β_1 and β_2 as memory parameters for the first and second moments respectively. This update rule effectively normalizes the learning rate by the standard deviation of the gradient, which is a natural measurement scale. This also has the effect of cutting off directions where the gradient varies by a lot.

Chapter 6

Atom-centered descriptors

Electronic structure calculations are among the most computationally intensive scientific calculations, and the rapid development of modern computers have made many previously unthinkable computer simulations a central part of the scientific toolkit. Because of their computationally demanding nature, electronic structure calculations today occupy a large portion of resources on modern scientific supercomputer facilities.

For small-scale systems, we have discussed the Hartree-Fock and Density Functional Theory methods as the primary workhorses of ab-initio (i.e. from first principles quantum mechanical principles) electronic structure calculations. However, these methods suffer from very poor scaling as the system size increases, with Hartree-Fock scaling as $\mathcal{O}(N^4)$ with N the number of electrons and Density Functional Theory scaling as $\mathcal{O}(N^4)$.

In many cases the exact details are electronic structure are less important than the long-time behaviour of the atoms and molecules involved in the simulation, and classical approximations can be made as in molecular dynamics, which comes close to linear scaling. This allows us to simulate systems of up to millions or hundreds of millions of atoms, which can approximate nano- or micro-scale atoms if periodic boundary conditions are applied. However, the question remains as to how you develop an accurate classical potential which can accurately reproduce fundamentally quantum systems, with a speed that allows us to enter into realistic timescales (i.e. nano or microseconds).

The most common approach to developing molecular dynamics potentials is to guess a functional form based on your physical intuition and experience with the systems and calculate appropriate parameters from data obtained from DFT calculations. The number of parameters involved can range from

two in the case of Lennard-Jones or hundreds of parameters in the case of complex, many-atom potentials such as the AMBER and CHARMM force fields. The imposition of functional forms to quantum data is an artform, and the potential must often be tailored to not only the chemical species and number of atoms in your system, but also the specific experimental quantity you are trying to extract, such as the energy, radial distribution function or transport coefficients. Notably there are dozens of MD potentials describing different models of water (H2O), each fine-tuned for a specific system structure or parameter.

Due to recent developments in the field of machine learning, the question has been raised as to how it may be possible to automate the process of developing potentials. [cite](https://www.sciencedirect.com/science/article/pii/S0010465516301266) The idea is to approximate the potential energy with a regression model:

$$\{\,\boldsymbol{R}\,\} \overset{\text{Regression}}{\longrightarrow} E = E(\{\,\boldsymbol{R}\,\}),$$

where $\{\,\boldsymbol{R}\,\}$ is the set of nuclear coordinates of our system. Most machine learning methods operate over a set of one-dimensional so-called *feature vectors*, where every vector element represents a feature of the data set. For example the amount of precipation in a given area at a given time is a function of features such as humidity, cloud cover, air pressure etc. This is a vector of some length F, while the nuclear coordinates represent a point in $3N$-dimensional phase space. This difference in representation requires some way of mapping the nuclear coordinates to features which can be employed by a machine learning method.

The naive approach would be to simply feed in the nuclear coordinates as a 1D vector, and then perform a regression on the dataset in order to obtain the potential energy. However, our physical intuition imposes some constraints on the potential energy. In particular, the potential energy of a microscopic system should be translationally, rotationally and permutationally invariant.

Translational invariance implies that the addition of any three-dimensional vector to every coordinate in the system should not in any way alter the potential energy of the system. This should not be the case with a naive mapping, as for a given set of weights (or equivalent) smaller/larger coordinates values would be mapped to smaller/larger activations, and therefore alter the final output. Rotational invariance implies that the potential energy of the system should not change as the system is rotated about an axis. This

also should not be the case in the context of a naive mapping, as any change to any of the inputs would be mapped in a non-linear way to produce a different output. Finally, permutation invariance implies that swapping the coordinates of any two atoms of the same chemical species would produce the same potential energy. This should also not be the case, for the same reasons as we just discussed. These constraints together heavily restrict the functional form that any mapping to the potential energy could have, which means a more careful analysis should be considered.

In order for the mapping to be applicable to systems of varying size, a decomposition into atomic energy contributions is performed:

$$E(\{\,\boldsymbol{R}\,\}) = \sum_{i=1}^{N} E_{\text{atom}}(\{\,\boldsymbol{R}\,\}).$$

The individual energy contributions E_{atom} are then approximated by performing a regression analysis. The atomic energy contributions are usually limited to its local environment through the introduction of a cutoff radius R_c:

$$E_{\text{atom}}(\{\,\boldsymbol{R}\,\}) \approx E_{\text{atom}}(\boldsymbol{R}_i, \{\,\boldsymbol{R}_j \mid |\boldsymbol{R}_{ij}| < R_c\,\})$$

meaning that interactions are only treated if the interatomic distance is smaller than the cutoff. This is a good approximation for a sensible choice of cutoff radius if no electrostatic interactions are involved. Long-range interactions can also be treated through the introduction of methods such as Ewald summation, but this will not be discussed here.

A mapping that satisfies the above constraints we will refer to as a *descriptor*, and is used as input to the regression method:

$$\{\,\boldsymbol{R}\,\} \rightarrow \boldsymbol{G}(\{\,\boldsymbol{R}\,\}) \overset{\text{regression}}{\longrightarrow} E_{\text{atom}} = E_{\text{atom}}(\boldsymbol{G}(\{\,\boldsymbol{R}\,\})).$$

Once we have a descriptor and a regression model the dynamics can be readily obtained by taking derivatives:

$$
\begin{aligned}
\boldsymbol{F}_i &= -\nabla_i E \\
&= -\nabla_i \sum_{i}^{\text{local}} E_{\text{atom}}(\boldsymbol{G}(\{\,\boldsymbol{R}\,\})) \\
&= -\sum_{i}^{\text{local}} \sum_{j} \frac{\partial E_{\text{atom}}}{\partial G_j} \frac{\partial G_j}{\partial \boldsymbol{R}_i},
\end{aligned}
\tag{6.1}
$$

where we have applied the chain rule to break the gradient into derivatives with respect to the network inputs (obtained through backpropagation) and derivatives of the network inputs with respect to the coordinates of atom i. Once we have the forces the system can be propagated through time using the Velocity-Verlet equations.

6.0.1 Gaussian descriptors

Behler and Parrinello [cite](https://journals.aps.org/prl/abstract/10.1103/PhysRevLett.98.146401) suggested the decomposition of the mapping $\boldsymbol{G}_i$ of atom i into two subvectors $\boldsymbol{G}_i^I$ and $\boldsymbol{G}_i^{II}$ representing pairwise and three-body interactions respectively. The components of $\boldsymbol{G}_i^I$ are comprised of sums of gaussian functions of the pairwise distance R_{ij}:

$$
f_i^I = \sum_{j\neq i}^{\text{local}} \exp\left(-\eta(R_{ij} - R_s)^2/R_c^2\right) f_c(R_{ij}),
$$

with the sum over the local environment of atom i. The parameters η and R_s represent the width and center of the gaussian functions respectively. The term f_c is a cutoff function which decays smoothly to zero at the cutoff radius. Behler and Parrinello proposed the following cutoff:

$$
f_c(R) = \begin{cases} \frac{1}{2}\left(1 + \cos\left(\pi R/R_c\right)\right) & R < R_c \\ 0 & R > R_c, \end{cases}
\tag{6.2}
$$

however other functional forms are possible. For the angular part such terms are multiplied, which means the function decays much more rapidly as we approach the cutoff. The only requirement we pose is that the function be continuous with a continuous first derivative in $r \in [0, \infty)$, approach one as $R \to 0$ and zero as $R \to R_c$.

The components of the three-body subvector are defined incorporating the angles θ_{ijk} between every triplet of atoms:

$$f_i^{II} = 2^{1-\zeta} \sum_{j,k\neq i}^{\text{local}} (1 + \lambda \cos\theta_{ijk})^{\zeta} \exp\left(-\eta \left(R_{ij}^2 + R_{ik}^2 + Rjk^2\right)/R_c^2\right) \times f_c(R_{ij}) f_c(R_{ik}) f_c(R_{jk}). \tag{6.3}$$

The components for each subvector is calculated by varying the different parameters $\eta, R_s, \zeta, \lambda$. The choice of neither symmetry functions, cutoff, nor the parameters employed by Behler and Parrinello are unique. The guiding wisdom is that atomic environments with different potential energies should give differing energies, while remaining invariant under translation, rotation and permutation. Finally we note that the descriptors and the neural network models are not interchangeable, as each neural network is trained for a specific set of input vectors, and must be retrained if the way inputs are composed changes.

6.0.2 Deep Potential Molecular Dynamics

Deep Potential Molecular Dynamics (DPMD) is a method proposed by [Wang et. al.](https://reader.elsevier.com/reader/sd/pii/S0010465518300882?token=2AFD04526C4190A36 in response to the successes of methods such as Behler-Parrinello, Gaussian Approximation Potentials (GAP) and Gradient-Domain Machine Learning (GDML). These methods all involve some amount of handcrafting the inputs, and building these inputs for larger, more complex systems is not straightforward. The Deep Potential method assigns a local environment and reference frame to each atom. The total potential energy is a sum of atomic contributions as before:

$$E = \sum_i E_i,$$

with the atomic energy determined by its nearest neighbors:

$$E_i = E(\boldsymbol{R}_i, \{\, \boldsymbol{R}_j : |\boldsymbol{R}_{ij}| < R_c \,\}).$$

The position of each neighbor of atom i is described by the relative positions $\boldsymbol{R}_{ij} = \boldsymbol{R}_j - \boldsymbol{R}_i$, which preserves translational symmetry.

Rotational symmetry is conserved by constructing a local frame for each atom. Two neighboring atoms a and b are picked by a user-specified rule (default: two closest). The environment of atom i is then described by three unit vectors:

$$\begin{aligned} \boldsymbol{e}_{i1} &= \boldsymbol{e}(\boldsymbol{R}_{ia}), \\ \boldsymbol{e}_{i2} &= \boldsymbol{e}(\boldsymbol{R}_{ib} - (\boldsymbol{R}_{ib} \cdot \boldsymbol{e}_{i1})\boldsymbol{e}_{i1}), \\ \boldsymbol{e}_{i3} &= \boldsymbol{e}_{i1} \times \boldsymbol{e}_{i2}, \end{aligned} \tag{6.4}$$

where $\boldsymbol{e}(\boldsymbol{R})$ denotes the normalized vector $\boldsymbol{e}(\boldsymbol{R}) = \boldsymbol{R}/\left|\boldsymbol{R}\right|$. Together these vectors form an orthonormal basis for the reference frame of atom i. The local coordinates $\boldsymbol{R}_{ij}$ can then be obtained from the global coordinates $\boldsymbol{R}^0_{ij}$ through the transformation:

$$\boldsymbol{R}_{ij} = \boldsymbol{R}^0_{ij} \cdot \mathcal{R}, \tag{6.5}$$

where $\mathcal{R} = [\boldsymbol{e}_{i1}\ \boldsymbol{e}_{i2}\ \boldsymbol{e}_{i3}]$ is a rotation matrix with columns given by the local basis vectors.

The neural network input vector for every atom-to-atom interaction $\boldsymbol{D}_{ij}$ can be given with radial-only or full radial-angular information:

$$D^{\alpha}_{ij} = \begin{cases} \left(\dfrac{1}{R_{ij}}, \dfrac{\boldsymbol{R}_{ij}}{|\boldsymbol{R}_{ij}|}\right) & \text{full information,} \\ \left(\dfrac{1}{R_{ij}}\right) & \text{radial-only information,} \end{cases} \tag{6.6}$$

with $\alpha = 0$ when only radial information is specified and $\alpha = 0, 1, 2, 3$ when full information is provided. Radial information is typically sufficient for long-range interactions such as van-der-Waals forces, while covalent bonding can be modeled by including only the closest atoms. We therefore specify two separate cutoff shells, one for the radial information R_c and one for treating angular interactions R_a.

In order to preserve permutation symmetry the inputs vectors $\boldsymbol{D}_{ij}$ are sorted first according to chemical species, and then within each chemical species according to their inverse distance $1/R_{ij}$. The vector of subvectors $\boldsymbol{D}_i$ is then fed through a neural network to produce the atomic energy contribution. The network input size is fixed according to the maximum number of neighbors in the system which is being studied, with $\boldsymbol{D}_{ij} = \boldsymbol{0}$ if there are fewer neighbors within the radial cutoff.

The authors also propose a scheme of force learning, wherein the force Root Mean Square Error (RMSE) is incorporated into the Loss Function:

$$\mathcal{L} = \frac{p_\epsilon}{N} \sum_i \left(\Delta E_i^2\right) + \frac{p_f}{3N} \sum_i \left|\Delta \boldsymbol{F}_i\right|^2, \tag{6.7}$$

with p_ϵ and p_f energy and force pre-factors which are adjusted throughout the learning process. The term ΔE_i denotes the error between the sum of network outputs and the correct potential energy, while the term $\Delta \boldsymbol{F}_i$ denotes the error in the force output. The pre-factors are adjusted based on the learning rate:

$$p(t) = p^{\text{limit}} \left[1 - \frac{r_l(t)}{r_l^0}\right] + p^{\text{start}} \left[\frac{r_l(t)}{r_l^0}\right], \tag{6.8}$$

with $r_l(t)$ and r_l^0 the learning rate at time step t and time step 0 respectively. The learning rate decays exponentially as:

$$r_l(t) = r_l^0 \cdot d_r^{t/d_s},$$

with d_r the decay rate and d_s the decay steps. The decay rate should be less than 1. The force error is often a magnitude or two larger than the energy error, and it is believed that incorporating the force into the loss should improve the learning rate for physics-based applications which incorporate forces. The virial information can also be treated in this manner, but this will not be discussed here.

Part II

Implementation

Chapter 7

Atomic Simulation Environment

The Atomic Simulation environment (ASE[1]) [cite](https://iopscience.iop.org/article/10.1088/1361-648X/aa680e) is a software package written in Python for the purpose of setting up, steering and analyzing atomistic simulations. Python is an interpreted, high-level general purpose language, with a powerful, consise syntax which allows one to perform very complex tasks with few lines of code. Python can also easily be extended and interfaced with fast and mature libraries. The modular interface of Python makes ASE easily extensible: in particular the calculator interface for evaluating energies, forces and much more has been implemented for software packages such as LAMMPS, VASP, Quantum Espresso and many more. The Atomic Simulation Environment is intended to be:

- Easy to use
- Flexible
- Customizable
- Pythonic
- Open to participation

[1] https://wiki.fysik.dtu.dk/ase/index.html

The real drawback of Python is that it is an interpreted language, which results in slow execution. It can also be quite memory-intensive, which makes pure Python unsuitable for large scale computations and simulations. It is therefore common to write the computationally demanding tasks in a lower level compiled language, and build a Python interface for calling functions and classes.

7.0.1 Installation

ASE requires an installation of

- Python 2.7, 3.4-3.6
- Numpy 1.9 or newer
- Scipy 0.14 or newer

This can be easily obtained through the Anaconda or Miniconda packages[2], or follow the instructions on the Python website[3]. Once you have the prerequisites ASE can be installed using pip:

```
pip install ase
```

7.0.2 Molecular Dynamics

Here we will demonstrate how to setup a simple Argon crystal, set the velocities and integrate the system using the Velocity Verlet equations. First we import some prerequisites and define the system:

```
from ase.lattice.cubic import FaceCenteredCubic
from ase import units
from ase.md.velocitydistribution import
   MaxwellBoltzmannDistribution
from ase.md.verlet import VelocityVerlet

symbol = "Ar"
size = (3, 3, 3)
```

[2] https://anaconda.org/
[3] https://www.python.org/

```
atoms = FaceCenteredCubic(symbol=symbol, size=size, pbc=
    True)
MaxwellBoltzmannDistribution(atoms, 300 * units.kB)
```

This defines a face-centered-cubic (FCC) crystal unit cell with 4 atoms, and a system size of $3 \times 3 \times 3$ unit cells for a total of $4 \cdot 3^3 = 108$ atoms with periodic boundary conditions. We thereafter give the atoms velocities according to the Maxwell-Boltzmann distribution such that the system temperature is approximately 300 Kelvin.

Next we give the atoms a Lennard-Jones *calculator* for calculating dynamics and create a Velocity Verlet *integrator* for operating on the atoms according to their forces. Calculators will be discussed in the next section.

```
calc = LennardJones(sigma=3.405, epsilon=1.0318e-2)
atoms.set_calculator(calc)
dyn = VelocityVerlet(atoms, 5 * units.fs)
```

The parameters σ and ϵ define the well known length and energy scales for the Lennard-Jones potential. The Velocity Verlet integrator is initialized with a timestep of 5 femtoseconds, which strikes a balance between accuracy and the timescale of the simulation. Finally we run the system for 1000 steps:

```
for i in range(100):
    dyn.run(10)
```

Every 10 timesteps the system can be printed, written to file or some other form of analysis and processing.

7.0.3 Calculators

The calculator interface gives ASE an easy to use, flexible and customizable way of computing dynamics, and makes ASE viable for a wide range of electronic structure calculations. For ASE a calculator is a black box that receives positions, atomic numbers and so on and outputs energies, forces, stresses; all that is required to perform atomistic simulations. The basic interface is defined in the ASE source code as:

```
import numpy as np


class Calculator:
```

```
    def get_potential_energy(self, atoms=None,
                             force_consistent=False):
        return 0.0

    def get_forces(self, atoms):
        return np.zeros((len(atoms), 3))

    def get_stress(self, atoms):
        return np.zeros(6)

    def calculation_required(self, atoms, quantities):
        return False
```

There is also an interface for DFT calculators, which also requires the implementation of spins, occupation numbers, the fermi level and so on.

In the previous section we used the Lennard-Jones calculator as an example, however this is generally not advised. Apart from the Lennard-Jones being a toy potential unsuited for real systems, the ASE implementation is also written in pure Python, which makes it very, very slow. Usually the calculator interface is used as a wrapper for much faster compiled code or software packages; for example the Asap calculator is a Python wrapper for the Effective Medium Theory potential which is implemented in Fortran, and is orders of magnitudes faster. A Python molecular dynamics code is justified since the bulk of computation code is spent on computing forces, however for large scale molecular dynamics simulations one would be advised to use fast compiled code such as LAMMPS[4] or GROMACS[5]. ASE really shines when it comes to small-scale simulations, experimentation and testing out new methods, and this is why it has been chosen as a basis for this thesis. Fortunately, ASE has an extensive code base of calculators for Molecular Dynamics and Density Functional Theory codes such as Asap, CP2K, LAMMPS, GROMACS which are all implemented in lower-level languages.

[4] https://lammps.sandia.gov/
[5] http://www.gromacs.org/

Chapter 8

Atomistic Machine-learning Package

The Atomistic Machine-learning Package (AMP) is a software package written in Python with the intent of bringing machine learning to electronic structure calculations. The software is intended to interface with ASE and the OpenKIM API for usage in LAMMPS. The interface to AMP is written purely in Python while computationally intensive tasks are outsourced to Fortran modules and the machine learning performed through a Tensorflow backend. The Python interface makes AMP flexible and easily extended, and this makes the package ideal for prototyping and testing both newer and more established machine learning methods.

A suggested workflow is as follows:

- Use AMP for training, testing and validation of novel descriptors and systems
- Use the AMP calculator for smaller scale simulation in the ASE environment
- Export the network compliant with the OpenKIM API for usage in more mature, large-scale electronic structure calculations such as LAMMPS

Unfortunately, the software is not currently fully compliant with the latest version of the OpenKIM API, which introduces artifacts in the simulation.

However, the authors have recently received a large grant from the U.S. Department of Energy[1], which should facilitate further development.

8.0.1 Theory

AMP is intended to interface extensively with ASE, and the primary interface to AMP is the AMP calculator. The AMP calculator is an ASE compliant calculator that accepts cartesian coordinates and outputs energies and forces, just as a classical molecular dynamics calculator. The primary difference between the AMP calculator and ASE calculators such as the Lennard-Jones calculator is the train method, which takes a set of *images*, i.e. a set of snapshots of atomic configurations labeled with the potential energy and forces. The calculator is fitted to the images and can subsequently used to predict atomic energies and forces from never before seen atomic configurations.

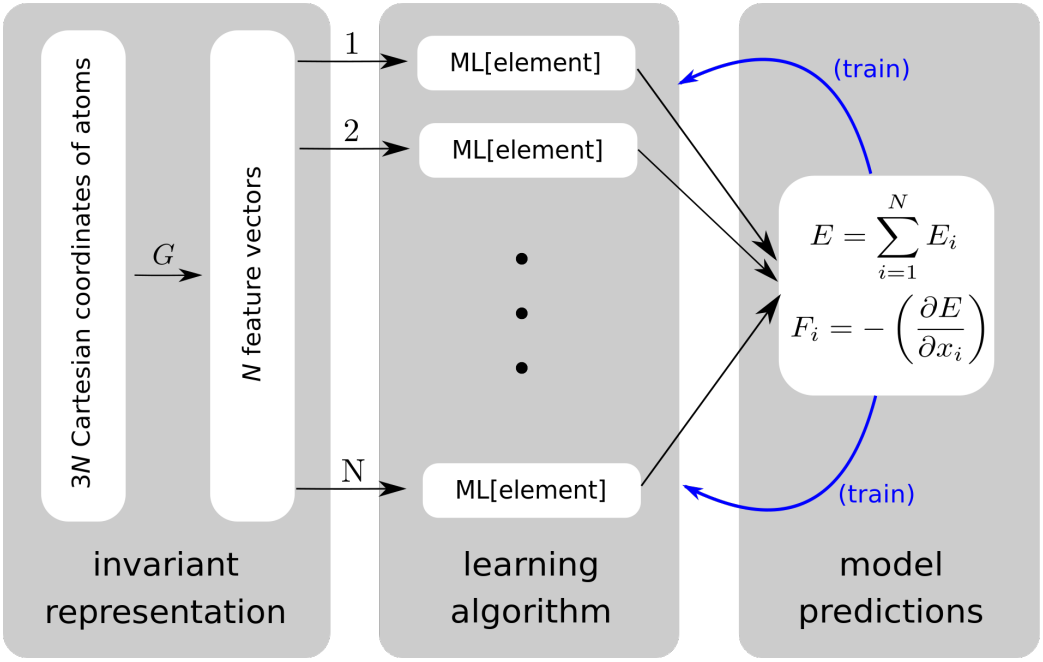

Figure 8.1: Schematics of how AMP works in atom-centered mode. Reprinted from article.

[1] https://www.brown.edu/news/2018-09-20/simulations

8.0.2 Installation

AMP requires an installation of

- Python 2.7, 3.4-3.6, 3.6 is recommended
- Numpy 1.9 or newer
- Scipy 0.14 or newer
- ASE

Python installations can be easily obtained through the Anaconda or Miniconda packages[2], or follow the instructions on the Python website[3]. ASE installation is described in the chapter on ASE. Once you have the prerequisites AMP can be installed using pip:

```
pip install amp-atomistics
```

8.0.3 Training example

In the previous chapter we showed how to run a molecular dynamics simulation using ASE. Here we will show how an AMP calculator can be fitted to molecular dynamics data with only minor modifications to the code.

First we import the prerequisites and define the system:

```
import ase.io
from ase.lattice.cubic import FaceCenteredCubic
from ase import units
from ase.md.velocitydistribution import
   MaxwellBoltzmannDistribution
from ase.md.verlet import VelocityVerlet

from amp import AMP
from amp.descriptor.gaussian import Gaussian
from amp.model.neuralnetwork import NeuralNetwork
from amp.model import LossFunction
```

[2] https://anaconda.org/
[3] https://www.python.org/

```
symbol = "Ar"
size = (3, 3, 3)
atoms = FaceCenteredCubic(symbol=symbol, size=size, pbc=
    True)
MaxwellBoltzmannDistribution(atoms, 300 * units.kB)
```

The class AMP is the primary object of the AMP package, and is implemented through the ASE interface. We will use a neural network machine learning model and Gaussian descriptors, which are an implementation of the Behler-Parrinello method. To implement force training we require access to the LossFunction class. We will also be using the ASE Input/Output (ase.io) module to generate an ASE *Trajectory*, which is an object storing the time-evolution of a simulation and usually interpreted as a time series of atoms.

```
traj = ase.io.Trajectory("training.traj", "w")
calc = LennardJones(sigma=3.405, epsilon=1.0318e-2)
atoms.set_calculator(calc)
atoms.get_potential_energy()
atoms.get_forces()
dyn = VelocityVerlet(atoms, 5 * units.fs)
traj.write(atoms)
```

In order to write the potential energy and forces to file they must first be calculated using the ASE Atoms methods, which require a calculator and otherwise raise an error. We can then evolve the system forward in time and save the atomic configuration every 10 time steps:

```
for i in range(100):
    dyn.run(10)
    atoms.get_potential_energy()
    atoms.get_forces()
    traj.write(atoms)
```

After the data has been generated we can train the AMP calculator using the Lennard-Jones calculations as input:

```
calc = Amp(descriptor=Gaussian(),
           model=NeuralNetwork(hiddenlayers=(10, 10, 10)))
calc.model.lossfunction = LossFunction(convergence={"
   energy_rmse": 1E-2,
```

```
                                                    "force_rmse":
    1E-2})
calc.train(images="training.traj")
```

8.0.4 Descriptors

Chapter 9

Pytorch

Part III

Results

Chapter 10

Empirical potentials

Chapter 11

Ab-Initio Molecular Dynamics

Chapter 12

Conclusion

Appendices

Appendix A

Title 1

Appendix B

Title 2

Bibliography

[1] Albert Einstein. "Zur Elektrodynamik bewegter Körper. (German) [On the electrodynamics of moving bodies]". In: *Annalen der Physik* 322.10 (1905), pp. 891–921. DOI: http://dx.doi.org/10.1002/andp.19053221004.