Computational Materials Science Notes

Andrew Valentini

February 2026 -

Contents

1 Thermodynamic Stability

1.1 Hull Energy

Materials are mostly characterized by their **compositions** (the atomic types present and their ratios) and their **crystal structure** (how these elements are arranged in 3D space). Example crystal structure types include

- Cubic (simplest)
- Face-centered cubic (FCC)
- Body-centered cubic (BCC)
- Perovskite
- Rock Salt

If a new *crystalline* material is discovered, this typically means that a new composition and/or crystal structure has been identified. The only new materials that are worth considering, however, are those that are *synthesizable* (those that can actually be made in a lab). A **stable** material is energetically favorable and is thus more likely to be synthesizable. The **"hull energy"** is the quantity that quantifies the thermodynamic stability of a material.

For a given composition, DFT computes the formation energy (the energy required to make the material from its constituent parts). This formation energy will change as you change ratio of the elements in the material and its structure. The goal is then to find the optimized structure and ratio of the composition. The formation energy must then be computed for the various possibilities of a given composition.

As a pedagogical example, first consider a binary compound. Consider a material made of titanium and aluminum. Our goal is to vary the percentage of each element in this compound, and since we have 4 total atoms, we can only choose from $\{\mathrm{Ti}, \mathrm{Ti_3Al}, \mathrm{Ti_2Al_2}, \mathrm{TiAl_3}, \mathrm{Al}\}$ where $\mathrm{Ti_3Al}$ represents 3 titanium atoms for every aluminum atom in the crystal structure, and Al means only aluminum atoms. For each of these compositions, there are also various crystal structures to consider. The crystal structure with the lowest formation energy, and therefore the most thermodynamically stable of the arrangement of the composition, is said to exist on the **convex hull**.

The convex hull for our titanium and aluminum material will therefore feature 5 entries. When these five are plotted alongside each other as a function of energy, we will have the convex hull of our material. This data can be found for most compositions on the Materials Project

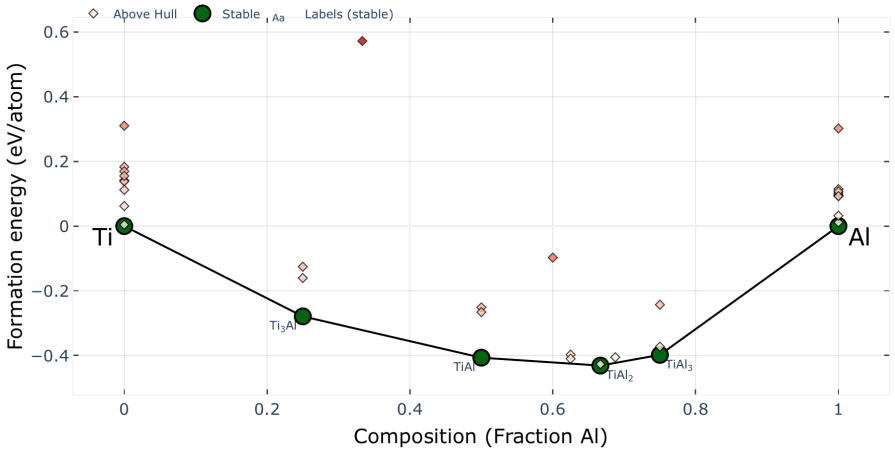

Figure 1: The formation energy for the different phases of Ti and Al The orange diamonds represent different crystal structures for a given composition. Those structures with a higher formation energy than the one along the convex hull are said to have some **distance above hull**. Conversely, the distance from the most stable structure of a given compound to the next most stable structure is defined as the **inverse hull distance**. A high inverse hull distance tells us that a compound with a specific crystal structure is very stable and unlikely to decompose into the other possible structures/phases when synthesized.

The same process is repeated for ternary, quaternary, etc compounds, though the phase space becomes higher dimensional (already becoming three dimensional in the ternary case) since we must introduce axes for the percentage of each element in the composition found in the material.

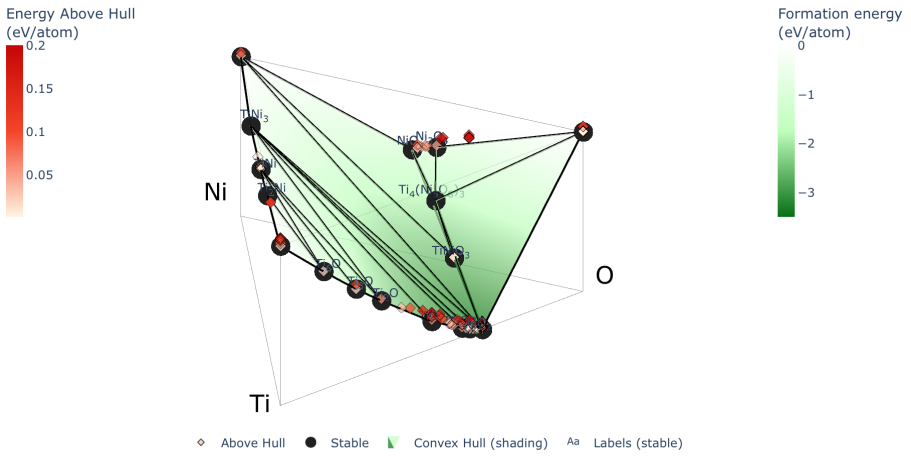

Figure 2: The three-dimensional convex hull for TiNiO generated in Materials Project.

2 Machine-Learned Interatomic Potentials

Machine-learned interatomic potentials (MLIPs) are AI-based models that are used to estimate atomic interactions of a structure and eventually find its most relaxed state.

2.1 "Relaxing" a Structure

Relaxing a structure involves finding the optimal geometry of a structure that minimizes its energy (and thus gives one the most probable structure of a compound that would be found after synthesis). This involves shifting atomic positions, computing the energy of the system, and moving the atoms in the direction that minimizes the energy of the material. Machine-learned potentials can do this iterative process much more efficiently than DFT calculations.

This process involves computing the forces on each atom:

$$\boldsymbol{F}_i = \frac{\partial E}{\partial \boldsymbol{R}_i} \tag{1}$$

where $E = E(\boldsymbol{R}, \boldsymbol{h})$ and $\boldsymbol{R}$ are the three-dimensional atomic coordinates of the material being considered:

$$\boldsymbol{R}_i = (\boldsymbol{R}_1, \boldsymbol{R}_2, \ldots, \boldsymbol{R}_N) \tag{2}$$

for N atoms in a material, each with three-dimensional coordinates. $\boldsymbol{h}$ is the unit cell that defines the periodic element of the material

$$\boldsymbol{h} = \begin{pmatrix} a_x & b_x & c_x \\ a_y & b_y & c_y \\ a_z & b_z & c_z \end{pmatrix} \tag{3}$$

The vectors $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}$ define

- Cell lengths a, b, c (magnitudes of these vectors)
- Angles α, β, γ (dot products between these vectors)
- Volume and orientation of the unit cell

Where $E(\boldsymbol{R}_i, \boldsymbol{h})$ is either computed by DFT or MLIPS.

2.2 Classical Pair Potentials

The first method of computing interatomic potentials, and therefore arriving at $E(\boldsymbol{R}_i, \boldsymbol{h})$ comes from 1924 with the Lennard-Jones potential, which considers the pair-wise interaction of each atom in a system. Because of it's pair-wise, nature it ignores the many-body effects that would need to be considered for realistic systems. The pair-wise potential looks like

$$\phi(r) = 4\epsilon\left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right] \tag{4}$$

where ϵ dictates the strength of the pair-wise interactions and σ is the distance between the two particles being considered. The total energy of a system therefore considered the summation of this potential over all possible pairs and then takes the gradient of the total potential.

2.3 Universal Potentials

Machine-learned potentials began with more simple neural network approaches, where the network was trained on data relevant to the crystal structure that one would wish to compute the interatomic potential for. The universal potential approach generalizes this approach by training a network on a massive scale of data, such that the network is able to accurately predict the potential of any given crystal. The most prominent examples of these today are M3Gnet and UMA. The initial version of UMA was trained on 500 million atomic structures.

3 Neural Networks

3.1 Neural Network Architecture

If a relationship between an input and output of a model is expected to be linear, then the model looks like

$$\boldsymbol{y} = f(\boldsymbol{x}) = \boldsymbol{w}^\top \boldsymbol{x} + \boldsymbol{b} \tag{5}$$

where $\boldsymbol{x}$ is the input of the data and $\{\boldsymbol{w}, \boldsymbol{b}\}$ are parameters/features of the model, lumped together as $\boldsymbol{\theta}$. $\boldsymbol{w}$ are the *weights* of the model and $\boldsymbol{b}$ are the *biases* of the model.

If the expected function is nonlinear, an activation function $\sigma(\cdot)$ is used to turn a linear function into a nonlinear output:

$$\boldsymbol{y} = f(\boldsymbol{x}) = \sigma\big(\boldsymbol{w}^\top \boldsymbol{x} + \boldsymbol{b}\big) \tag{6}$$

Activation functions introduce nonlinearity into the influence that the input data has on the outputs, which is essential for modeling non-linear phenonema. The most common activation function is

$$\text{ReLU}(x) = \begin{cases} 0 \text{ if } x \leq 0 \\ x \text{ else} \end{cases}, \tag{7}$$

which is ubiquitous in machine learning since it is simple to implement, easy to calculate from, and its derivatives are easy to calculate from. Though the function looks linear, its discontinuity makes the function strictly nonlinear.

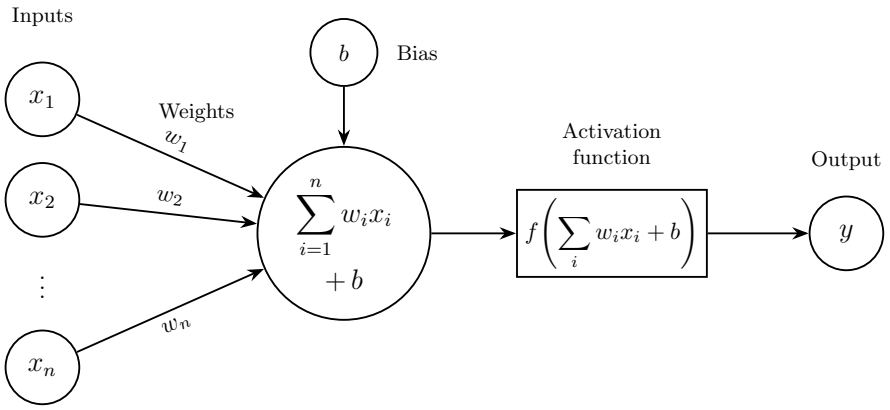

Figure 3: Diagram of how a single node in a machine learning model works. A cascade of these nodes is called a neural network, since it allows the machine learning model to learn complex, nonlinear trends in data.

If a model is comprised of only one linear transformation followed by a single activation function (its inputs are only updated once), the *expressivity* of the model will be limited; the model's ability to learn complex correlations will be limited, since it will only learn correlations close to a simple linear regression. We therefore **cascade** a sequence of these linear transformations and activation functions in a multi-layer model, called a neural network due to its mimicking of the processes of neurons in our brains.

To construct a multi-layer model,

$$\begin{aligned} & \boldsymbol{h}_1 = \sigma\big(\boldsymbol{w}_1^\top \boldsymbol{x} + \boldsymbol{b}_1\big) \\ \implies & \boldsymbol{h}_2 = \big(\boldsymbol{w}_2^\top \boldsymbol{x}_1 + \boldsymbol{b}_2\big) \\ \implies \cdots \implies & \boldsymbol{h}_j = \big(\boldsymbol{w}_j^\top \boldsymbol{x}_{j-1} + \boldsymbol{b}_j\big) \\ \implies & \boldsymbol{y} = \big(\boldsymbol{w}_i^\top \boldsymbol{x}_j + \boldsymbol{b}_i\big) \end{aligned} \tag{8}$$

where $\boldsymbol{h}_i$'s are called the **hidden layers** of the model. Neural networks with multiple hidden layers are often called **deep neural networks**, fully-connected networks, or multi-layer perceptrons, due to their exceptional ability to learn complex non-linear correlations in data.

3.2 Training Neural Networks

The goal of training a neural network is to update the weights $\boldsymbol{w}$ and biases $\boldsymbol{b}$ of a network so that the model best learns how to predict outputs. In other words, we want to optimize the performance of the model, which is described by the loss function $\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{x})$ where $\boldsymbol{\theta} = \{\boldsymbol{w}, \boldsymbol{b}\}$, which tells us that the performance of the model depends on the model parameters and how well these result in a fit of the model's predictions to the known output data.

Given input data and a known target output $\boldsymbol{y}$, if we write the neural network's response, or outputs, as $f(\boldsymbol{x}, \boldsymbol{\theta})$, we then want to minimize the error between the target output and the neural network's output:

$$\|\boldsymbol{y} - f(\boldsymbol{x}, \boldsymbol{\theta})\|^2 \tag{9}$$

Since we want to minimize the model's error in predicting all outputs $\boldsymbol{y}$ in the target distribution, the overall error of a neural network will be the weighted sm of all output errors:

$$\begin{aligned} \mathcal{L}(\boldsymbol{\theta}, \boldsymbol{x}) &= \frac{1}{N} \sum_{i=1}^{N} \|\boldsymbol{y}_i - f(\boldsymbol{x}_i, \boldsymbol{\theta})\|^2 \\ &\equiv \mathbb{E}_{x \sim \mathcal{D}} \Big[\|\boldsymbol{y}(\boldsymbol{x}) - f(\boldsymbol{x}, \boldsymbol{\theta})\| \Big] \end{aligned} \tag{10}$$

where N is the number of outputs and $\boldsymbol{x} \sim \mathcal{D}$ are the known samples we think of as being drawn from some underlying data distribution we would like to learn and sample from.

The most widely used method for minimizing the loss function is **gradient descent**. Given the current loss function $\mathcal{L}(\cdot)$ and corresponding parameters values at some time t, $\boldsymbol{\theta}^{(t)}$, one can minimize the loss by taking the gradient od the loss function with respect to the parameters and moving the parameter values *against* this gradient, since the gradient points in the direction in steepest ascent and we want to *minimize* the loss on this landscape.

If the parameters therefore move along the opposite direction of the gradient for a small step, the loss function will slowly become minimized. The goal of gradient descend is to continue moving the neural network's parameter values along the opposite direction of the space's gradient until the loss finds a minimum on the landscape. The step size that this descent, or learning, is done over is called the **learning rate** $\eta > 0$. The learning rate is

an example of a hyperparameter of the model since it is a parameter of the model itself, not of the data being learned. The neural network's parameters are therefore updated in gradient descent as

$$\begin{aligned} \boldsymbol{\theta}^{(t+1)} &= \boldsymbol{\theta}^{(t)} - \eta \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}) \\ &\equiv \boldsymbol{\theta}^{(t)} - \eta \mathbb{E}_{x \sim \mathcal{D}} \Big[\mathcal{E}(\boldsymbol{x}, \boldsymbol{\theta}) \Big] \end{aligned} \tag{11}$$

where $\nabla_{\boldsymbol{\theta}}$ indicates derivatives with respect to the parameter values, typically the model's weights and biases $\boldsymbol{\theta} = \{\boldsymbol{w}, \boldsymbol{b}\}$ since it is these values we are looking to optimize. $\mathcal{E}(\boldsymbol{x}, \boldsymbol{\theta})$ is any general error function, which we previously explicitly wrote as the mean-squared error.

Optimizing the loss function for an entire dataset $\boldsymbol{x} \sim \mathcal{D}$ is usually very computationally expensive given the large size of the dataset. The most common approach in optimizing the loss function in the presence of large data is splitting the data up into **batches** so that

$$\mathbb{E}_{x \sim \mathcal{D}} \Big[\mathcal{E}(\boldsymbol{x}, \boldsymbol{\theta}) \Big] \approx \frac{1}{\mathbb{B}} \sum_{i \in \mathbb{B}} \nabla_{\boldsymbol{\theta}}, \tag{12}$$

meaning we partition the full dataset into batches $\{\mathbb{B}_i\}$ which we sequentially feed into the optimization algorithm so that each batch equally contributes to the optimization. One **epoch** during the training represents one full pass through the entire dataset, which has been split into batches. Training will happen over multiple epochs, which represent repeated subdividing of the full dataset into new batches, and sequentially running gradient descent on the order of batches.

4 Machine Learning Techniques

4.1 Domain Generalization

4.2 Out-of-Distribution (OOD) Learning

4.3 Physics-Informed Constraints

4.4 Bayesian Optimization

5 Diffusion Models

Starting from a image $\boldsymbol{x}_0 \sim \mathcal{D}$, a diffusion model will gradually add Gaussian noise $\epsilon \sim \mathcal{N}(0, 1)$

6 Generative Modeling

Goal: Efficently learn the underlying distribution some data takes on, so that new samples can be drawn (in the example of image generation, this means determining the distribution for some set of images, say cats, so that new cat images can be created by the model). This is typically done by training a neural network to take a distribution of pure-noise to the desired distribution of data. One dominant way of doing this is through contintuous normalizing flows (CNFs) [1].

6.1 Continuous Normalizing Flow (CNF)

CNF defines a continuous-time transformation ϕ_t, for $t \in [0, 1]$, from a distribution of pure noise p_0 to a desired distribution of data p_1;

$$p_1 = [\phi_t]_\star p_0. \tag{13}$$

This transformation ϕ_t, called a *flow*, is invertible, meaning a neural network is first trained backwards to map given data samples to a distribution of noise, and then used forward in sampling from the distribution of noise to generate a new data sample.

The flow at a given time t_i, written as ϕ_{t_i}, takes the data distribution p_i and "pushes it forward" to a new shape p_j ($0 \leq t_i < t_j \leq 1$) until the target distribution p_1 is reached. How these transformations ϕ_t change over time, from $\phi_{t_i} \to \phi_{t_j}$ is determined by a time-dependent *vector-field* $v_\theta(\phi_t(x))$:

$$\frac{d\phi_t(x)}{dt} = v_\theta(\phi_t(x)), \tag{14}$$

where $x \in \mathbb{R}^d$ denotes the data points in the $d-$dimensional data space and $\theta \in \mathbb{R}^p$ are the parameters in the $p-$dimensional parameter space that the neural network can learn.

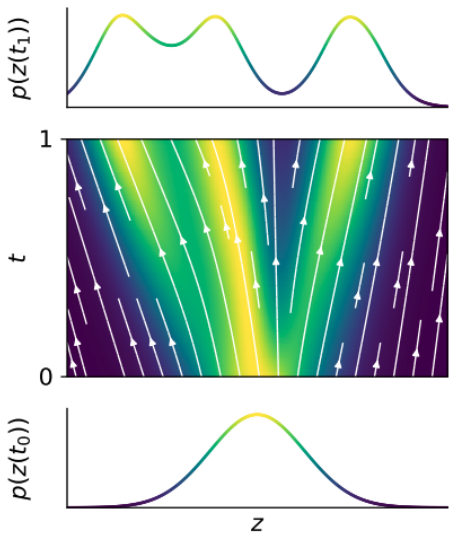

Figure 4: A figure from [2] demonstrating how a CNF vector field maps noise to a target distribution; $v_\theta(\phi_t(x)) : p_0 \to p_1$. One can imagine slicing horizontally across the t-axis, where at a given t-slice i, $\phi_i(x)$ transforms the data from p_{i-1} to p_i. The underlying vector field (which is a neural network) determines how $\phi_t(x)$ changes over time.

Training the neural network $v_\theta(\phi_t(x))$ is detailed in many papers and reviews, such as [3], but at a high-level, it requires solving an ODE and the divergence of the vector field, which is computationally expensive and unstable.

6.2 Flow Matching

Instead of learning evolution of the probability distribution p_t under $\phi_t(x)$ at each time to construct $v_\theta(\phi_t(x))$, we can instead *learn the velocity field directly*, which is the objective of Flow Matching [4].

Consider a target distribution $q(x)$, whose form we don't know exactly, but which we can approximate as a known distribution $p_1(x)$. We first construct a vector field $u_t(x)$ that takes us from a simple, known, distribution to the approximate form of the target distribution;

$$u_t(x) : p_0(x) = \mathcal{N}(x|0, I) \to p_1(x) \approx q(x). \tag{15}$$

Flow matching then trains a neural network to learn the optimal vector field $v_t(x)$ with the loss function

$$\mathcal{L}_{\text{FM}}(\theta) = \mathbb{E}_{t,p_t(x)}\|v_t(x) - u_t(x)\|^2, \tag{16}$$

which turns our objective into a regression problem, rather than needing to solve an ODEs and find the divergence of a vector field, as in CNF.

6.3 Riemannian Flow Matching

We have so far considered data that lives on a *Euclidean* manifold, denoted by $\mathbb{R}^d$. Examples of Euclidean data include:

- Tabular data (made up of feature vectors; ordered lists of numbers where each entry corresponds to the value of a different quantity)
- Images (made up of arrays of pixels)
- Word embeddings (words represented as vectors where the meaning of a word determines how close it is to other vectors/words in the space)

Standard neural networks assume this structure of data so that linear algebra principles can be applied.

Not all data is Euclidean, however, and can be represented on curved manifolds and graphs, for example. More common examples of non-Euclidean data include:

- Social networks
  - Traditional GNNs operate in Euclidean space by representing node features as vectors, but only do so after a mapping has been applied. Riemannian GNNs exist, however.
- **Materials**
  - Materials are also graph-structured, where the nodes represent atoms and the edges represent their bonds. The arrangement of atoms determines the connectivity of the graph and one must ensure that the graph respects physical symmetries. One should expect that a graph representing a material should not change its physics under translations, rotations, or reflections. In other

words, the positions of atoms in a material live in a Euclidean space $\mathbb{R}^{3N}$ where N is the number of atoms in the material, but to ensure the material respects physical symmetries, the material lives in a quotient space of $\mathbb{R}^{3N}/$(physical symmetries). For example, if we wish the atoms to be equivariant under rotational symmetry, the quotient space $\mathbb{R}^{3N}/SO(3)$ becomes a manifold.

– Reminder: for a quantity $y(x)$, such as the positions x^{3N} of a material, to be *invariant* under some transformation $R \in SO(3)$, for example, $y(R \cdot x) = y(x)$. For the quantity to be *equivariant* under the transformation, $y(R \cdot x) = Ry(x)$.

7 Generative AI Models for Crystal Structure

7.1 Performance Metrics

First introduced in the MatterGen paper, S.U.N. is a metric that quantifies a generative model's

- **Stability:** Out of 1,000 generated strcutures, the stability of a generative model is defined as what fraction of this number have $E_{\text{hull}} < 100$ meV/atom
- **Uniqueness:** Checks out of the 1,000 generated, how many are duplicates
- **Novelty:** checks out of the 1,000 generated, how many are already found in the **WHICH** database?

The actual ratio itself is constructed by taking ...

question to answer: how exactly does this metric compute the novelty? I would like to compute this for mine as well...

References

[1] Ricky T. Q. Chen et al. *Neural Ordinary Differential Equations*. 2019. arXiv: 1806.07366 [cs.LG]. URL: https://arxiv.org/abs/1806.07366.

[2] Will Grathwohl et al. *FFJORD: Free-form Continuous Dynamics for Scalable Reversible Generative Models*. 2018. arXiv: 1810.01367 [cs.LG]. URL: https://arxiv.org/abs/1810.01367.

[3] Ivan Kobyzev, Simon J.D. Prince, and Marcus A. Brubaker. "Normalizing Flows: An Introduction and Review of Current Methods". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 43.11 (Nov. 2021), pp. 3964–3979. ISSN: 1939-3539. DOI: 10.1109/tpami.2020.2992934. URL: http://dx.doi.org/10.1109/TPAMI.2020.2992934.

[4] Yaron Lipman et al. *Flow Matching for Generative Modeling*. 2023. arXiv: 2210.02747 [cs.LG]. URL: https://arxiv.org/abs/2210.02747.