Neural Structure Fields with Application to Crystal Structure Autoencoders

Naoya Chiba[1] **Yuta Suzuki**[2,*] **Tatsunori Taniai**[1] **Ryo Igarashi**[1] **Yoshitaka Ushiku**[1]
Kotaro Saito[3,4] **Kanta Ono**[4]
[1]OMRON SINIC X Corporation [2]TOYOTA Motor Corporation [3]Randeft, Inc.
[4]Osaka University
{naoya.chiba, tatsunori.taniai, ryo.igarashi, yoshitaka.ushiku}@sinicx.com
yuta_suzuki_ah@mail.toyota.co.jp
kotaro.saito@randeft.jp
ono@ap.eng.osaka-u.ac.jp

Abstract

Representing crystal structures of materials to facilitate determining them via neural networks is crucial for enabling machine-learning applications involving crystal structure estimation. Here we propose *neural structure fields* (NeSF) as an accurate and practical approach for representing crystal structures using neural networks. Inspired by the concepts of vector fields in physics and implicit neural representations in computer vision, the proposed NeSF considers a crystal structure as a continuous field rather than a discrete set of atoms. Unlike existing grid-based discretized spatial representations, the NeSF overcomes the tradeoff between spatial resolution and computational complexity and can represent any crystal structure. To evaluate the NeSF, we propose an autoencoder of crystal structures. Quantitative results demonstrate the superior performance of the NeSF compared with the existing grid-based approach.

1 Introduction

Representing crystal structures in a neural-network-familiar expression has promising utility in machine-learning (ML)-assisted material development and materials informatics (MI). For example, graph-based representations of crystal structures have allowed the encoding of crystal structures via graph neural networks [1, 2, 3, 4] for high-throughput material property predictions. When this type of task is viewed as the discovery of *structure-to-property relationships* among materials, there is yet another important type of task, namely, the discovery of *property-to-structure relationships*, which constitutes an inverse design problem [5, 6, 7, 8, 9]. Despite the great potential utility of this inverse approach in developing materials, few studies have addressed this [5, 6, 7, 8, 9] or its underlying problem [10, 11, 12], that is, the estimation of crystal structures under given conditions. Regarding MI and ML, whether to input or output crystal structures (*i.e.*, *encode* or *decode* crystal structures in MI and ML terms) induces a crucial difference. Although encoding crystal structures is suitably established using graph neural networks [1, 2, 3, 4], a technical bottleneck persists for decoding crystal structures. We addressed the bottleneck in this study.

The crystal structure of an inorganic material is a regular and periodic arrangement of atoms in a three-dimensional (3D) space. This arrangement is usually described by the 3D positions and species of atoms in a unit cell and the lattice constants defining the translations of the unit cell in 3D space. The atoms in a unit cell have no explicit order, and their quantity varies from one to hundreds in number. Because ML models, including neural networks, generally accept fixed-dimensional

*This work is mainly done in The Graduate University for Advanced Studies (SOKENDAI), Ibaraki, Japan.

36th Conference on Neural Information Processing Systems (NeurIPS 2022).

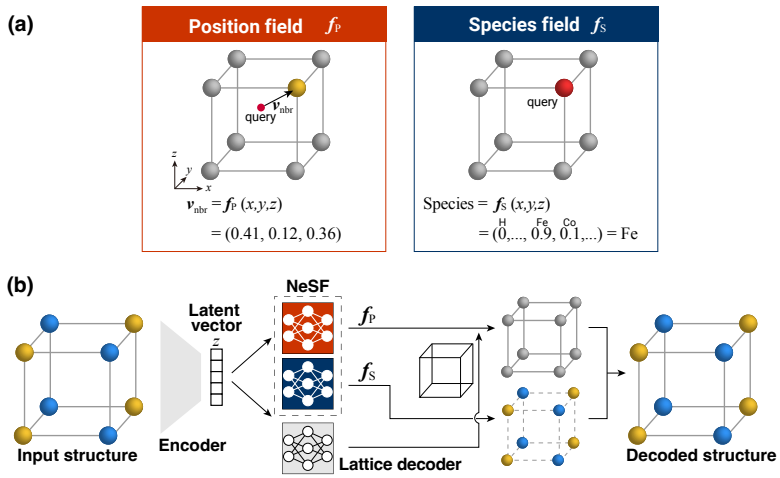

Figure 1: **NeSF and crystal structure autoencoder using the NeSF**. **a**, The NeSF consists of two vector fields, namely, position field $\boldsymbol{f}_{\mathrm{p}}$ and species field $\boldsymbol{f}_{\mathrm{s}}$, which are defined in 3D space. **b**, Network architecture of proposed NeSF-based crystal structure autoencoder.

and consistently ordered tensors for processing, treating crystal structures with ML models is not straightforward [1], and determining crystal structures via the models is even more difficult.

In this paper, we propose a continuous vector-field-based crystal structure representation named *neural structure fields* (NeSF). The proposed NeSF is inspired by the concepts of vector fields in classical physics and implicit neural representations [13, 14, 15, 16, 17] in computer vision. Implicit neural representations have recently been proposed to handle some representation issues in 3D computer vision applications, such as 3D shape estimation [14, 15, 16] and free-viewpoint image synthesis [13, 17]. The NeSF follows the basic idea of implicit neural representations and further extends it to the estimation of crystal structures described by atomic positions and species.

Our idea of representing crystal structures as continuous vector fields has been partially and implicitly explored by grid-based discretization (*i.e.*, voxelization) in recent MI studies [5, 6, 12, 8, 10]. In those studies, the 3D space within the unit cell is discretized into voxels, and each voxel is then assigned an electron density, which essentially represents the presence or absence of an atom around the voxel. However, the discretization of 3D data considerably suffers from the tradeoff between spatial resolution and computational complexity in terms of both computational time and memory space. For instance, the ICSG3D method [10] based on 3D convolutional neural networks (CNNs) uses a limited resolution of $32 \times 32 \times 32$ voxels. The NeSF overcomes the limitation of voxelization. Theoretically, NeSF can achieve infinitely high spatial resolution with compact (memory- and parameter-efficient) neural networks in place of costly 3D CNNs.

2 Method

In this section, we first describe the key concept of the NeSF and the procedure for estimating crystal structures with the NeSF. We then propose an autoencoder of crystal structures using NeSF.

2.1 Crystal structure representation via NeSF

We propose the NeSF as a general representation of crystal structures that enables neural networks to decode or determine such structures. The key concept underlying our approach is illustrated in Fig. 1a, where a crystal structure is represented as a continuous vector field tied to 3D space rather than as a discrete set of atoms. The NeSF uses two types of vector fields, namely, the position and species fields, to implicitly represent the positions and species of the atoms in the unit cell of the crystal structure, respectively.

To illustrate the concept of the NeSF, assume that we are given target material information as a fixed-dimensional vector, $\boldsymbol{z}$, and consider the problem of recovering a crystal structure from $\boldsymbol{z}$.

In the NeSF, we use neural network $\boldsymbol{f}$ as an implicit function to indirectly represent the crystal structure embedded in $\boldsymbol{z}$ instead of letting neural network $\boldsymbol{f}$ directly output the crystal structure as $\boldsymbol{f}(\boldsymbol{z})$. Specifically, we treat $\boldsymbol{f}$ as a vector field on the Cartesian 3D coordinates, $\boldsymbol{p}$, conditioned on the target material information, $\boldsymbol{z}$:

$$\boldsymbol{s} = \boldsymbol{f}(\boldsymbol{p}, \boldsymbol{z}) \tag{1}$$

In the position field, network $\boldsymbol{f}$ is trained to output a 3D vector pointing from query point $\boldsymbol{p}$ to its nearest atom position, $\boldsymbol{a}$, in the crystal structure of interest. Thus, we expect output $\boldsymbol{s}$ to be $\boldsymbol{a} - \boldsymbol{p}$. If the position field is ideally trained, we can retrieve position $\boldsymbol{a}$ of the nearest atom at any query point $\boldsymbol{p}$ as $\boldsymbol{p} + \boldsymbol{f}(\boldsymbol{p}, \boldsymbol{z})$. The precise description of atomic positions in crystal structures is of crucial interest in materials science. Thus, the NeSF outputs vectors pointing to the nearest atoms to represent atomic positions more directly than existing implicit neural representations of 3D geometries [14, 15, 16]. Analogously, the species field is trained to output a categorical probability distribution that indicates the species of the nearest atom. Thus, the output dimension of the species field is the number of candidate atomic species.

The training procedure of NeSF is simple and similar to those for existing implicit neural representations [14, 18]. We randomly sample 3D query points $\{\boldsymbol{p}_i^s\}$ in the unit cell and compute loss values for the field outputs at these points, thus supervising $\boldsymbol{f}_\mathrm{p}(\boldsymbol{p}_i^s, \boldsymbol{z})$ and $\boldsymbol{f}_\mathrm{s}(\boldsymbol{p}_i^s, \boldsymbol{z})$ to indicate the position and species of the nearest atom, respectively. To efficiently train the NeSF, we introduce two sampling methods. 1) Global grid sampling considers 3D grid points that uniformly cover the entire unit cell and samples the points with perturbations that follow a Gaussian distribution. 2) Local grid sampling considers local 3D grid points centered at each atomic position and samples the points with perturbations that follow a Gaussian distribution. To train the position field, we combine both sampling methods. Hence, query points are sampled uniformly over the entire unit cell and densely around the atoms. To train the species field, we use local grid sampling to concentrate training query points in the neighborhood of atoms. See Appendix A for more details.

2.2 Crystal structure estimation using NeSF

Given the target material information as vector $\boldsymbol{z}$, the estimation of the crystal structure from $\boldsymbol{z}$ amounts to estimating the positions and species of the atoms in the unit cell along with the lattice constants. The lattice constants are modeled as lengths a, b, and c and angles α, β, and γ, and are estimated via simple multilayer perceptrons (MLPs) with input $\boldsymbol{z}$. On the other hand, the atomic positions and species are estimated by position field $\boldsymbol{f}_\mathrm{p}$ and species field $\boldsymbol{f}_\mathrm{s}$ of the NeSF, respectively. These fields are also implemented as simple MLPs, each taking query position $\boldsymbol{p}$ and vector $\boldsymbol{z}$ as input and predicting a field value (*i.e.*, 3D pointing vector or categorical probability distribution). An overview of the NeSF network architecture is illustrated in the right part of Fig. 1b.

Given vector $\boldsymbol{z}$ from the encoder, the estimation of the atomic positions and species using the NeSF is summarized in the five steps:

1. **Initialize particles.** We first estimate the lattice constants via MLPs. Then, we regularly spread initial query points $\{\boldsymbol{p}_i^0\}$, which we call *particles*, at 3D grid points within a bounding box. The bounding box is common to each dataset and is given to loosely encompass the atoms of all training samples.
2. **Move particles.** We update the position of each particle $\boldsymbol{p}_i^t$ using the position field according to $\boldsymbol{p}_i^{t+1} = \boldsymbol{p}_i^t + \boldsymbol{f}_\mathrm{p}(\boldsymbol{p}_i^t, \boldsymbol{z})$. We iterate this process for all particles to obtain $\{\boldsymbol{p}_i\}$ as candidate atomic positions. Through this process, the particles travel toward their nearest atoms.
3. **Score particles.** We score each particle $\boldsymbol{p}_i$ and filter outliers. Since the norm of the output of the position field, $\|\boldsymbol{f}_\mathrm{p}(\boldsymbol{p}_i, \boldsymbol{z})\|$, indicates an estimated distance from $\boldsymbol{p}_i$ to its nearest atom, we score each particle $\boldsymbol{p}_i$ by $\|\boldsymbol{f}_\mathrm{p}(\boldsymbol{p}_i, \boldsymbol{z})\|$ and discard the particle if the score is above a certain threshold (set to 0.9 Å in this study).
4. **Detect atoms.** Until this point, the particles are expected to form clusters around atoms. Thus, we apply a simple clustering algorithm to detect each cluster position as an atomic position, determining the number of atoms in the crystal structure. Here we employ a well-known clustering algorithm in object detection, non-max suppression, and set its suppressing radius to 0.5 Å.

5. **Estimate species.** Finally, we use species field $\boldsymbol{f}_{\mathrm{s}}(\boldsymbol{p}, \boldsymbol{z})$ to estimate the atomic species at each atomic position $\boldsymbol{a}_i$. For robust estimation against errors in $\boldsymbol{a}_i$, we spread new particles intensively around each $\boldsymbol{a}_i$ as queries to the species field. We select the most frequent atomic species among them as the final estimate.

2.3 Crystal structure autoencoder

To demonstrate and evaluate the expressive power of the NeSF, we propose an autoencoder of crystal structures. Similar to other common autoencoders, the proposed NeSF-based autoencoder consists of an encoder and decoder. The encoder is a neural network that transforms an input crystal structure (*i.e.*, positions and species of the atoms in the unit cell and lattice constants) into abstract latent vector $\boldsymbol{z}$. The decoder, for which we use the NeSF, reconstructs the input crystal structure from latent vector $\boldsymbol{z}$. Autoencoders are typically used to learn latent vector representations of data via self-supervised learning, in which the input data can supervise the learning via a reconstruction loss.

While we focus on decoding crystal structures, their encoding requires a specific type of neural network called *set encoders*. Among them, the family of graph neural networks [1, 2, 3, 4] serves as popular crystal-structure encoders. However, these networks implicitly represent atomic positions as edges, encoding distances between atoms while discarding the exact coordinates. Although this distance-based graph representation is key to ensure the invariance to coordinate systems, its information loss in input may unintentionally hinder the reconstruction performance. Thus, we adopt the basic encoder architecture from PointNet [19] and DeepSets [20]. This architecture not only represents the simplest type of set encoders but can also preserve the information of input crystal structures, thus being appropriate for the performance evaluation of the NeSF decoder. The proposed autoencoder architecture is detailed in Appendix B.

During training, the four types of outputs from the decoder (*i.e.*, atomic positions, species, lattice lengths, and angles) were evaluated using the following loss function:

$$L = \lambda_{\mathrm{pos}} L_{\mathrm{pos}} + \lambda_{\mathrm{spe}} L_{\mathrm{spe}} + \lambda_{\mathrm{len}} L_{\mathrm{len}} + \lambda_{\mathrm{ang}} L_{\mathrm{ang}} \tag{2}$$

Here, L_{pos} is the mean squared error (MSE) between estimated and true atomic positions, L_{spe} is the cross-entropy loss function for the atomic species distributions, and L_{len} and L_{ang} are the mean square error for the lattice lengths and angles, respectively. The total loss function, L, is given by the weighted sum of these loss functions with weights $(\lambda_{\mathrm{pos}}, \lambda_{\mathrm{spe}}, \lambda_{\mathrm{len}}, \lambda_{\mathrm{ang}}) = (10, 0.1, 1, 1)$. We optimized the loss function using stochastic gradient descent with a batch size of 128. We used Adam [21] as the optimizer with an initial learning rate of 10^{-3} decaying every 640 epochs by a factor of 0.5. We conducted iterative training for 3200 epochs for all training samples in the dataset with early stopping. These hyperparameters were chosen through the procedure described in Appendix C. To reduce the performance variation owing to randomness, we repeated training and evaluation 10 times with different random seeds and evaluated the average performance.

3 Results and Discussions

We compared the performance of the proposed NeSF-based autoencoder with the grid-based ICSG3D baseline [10]. We reproduced the ICSG3D dataset [10] for training and evaluation by curating 7897 material samples from the Materials Project [22] (see Appendix D for details).

The reconstruction performance was measured in terms of errors in the number of atoms, position, and species. The error in the number of atoms is the rate of materials for which the number of atoms in the unit cell is incorrectly estimated. The position error is the average error of the reconstructed atomic positions. Depending on the denominator of the metric, we evaluated the position error in two ways. An *actual* metric was used to evaluate the mean position errors at the actual atomic sites of the crystal structure by computing their shortest distances to estimated atomic sites. By contrast, a *detected* metric was used to evaluate the errors at the estimated sites by computing their shortest distances to the actual atomic sites. The actual metric is more sensitive to errors related to underestimation of the number of atoms, whereas the detected metric is more sensitive to errors related to overestimation. The species error is the average rate of atoms with incorrectly estimated species. Analogous to the position error, the species error was evaluated using actual and detected metrics. Lower values of these metrics indicate better performance.

Table 1: Reconstruction results (mean $\pm$ standard deviation) on the ICSG3D dataset [10].

Method	Proposed	ICSG3D [10]
Error in number of atoms [%]	**0.53** ± 0.25	2.67 ± 0.84
Position error (actual) [Å]	**0.0308** ± 0.0112	0.0877 ± 0.0306
Position error (detected) [Å]	**0.0359** ± 0.0226	0.1057 ± 0.0284
Species error (actual) [%]	**4.31** ± 0.39	64.39 ± 1.91
Species error (detected) [%]	**4.36** ± 0.39	65.05 ± 1.85

Table 1 lists the reconstruction errors of the proposed NeSF-based autoencoder and ICSG3D baseline on the test set. Overall, the proposed method consistently outperforms ICSG3D in all the evaluation metrics, with substantial performance improvements for the species error. Meanwhile, ICSG3D achieves good performance for the error in number of atoms and the position. The evaluated dataset [10] consists of materials with limited crystal systems (*i.e.*, cubic) and prototypes (*i.e.*, AB, ABX2, and ABX3) and may thus offer relatively simple problem instances. Evaluating the reconstruction performance on more diverse crystal structures is future work. We further discuss potential negative societal impacts of this work in Appendix E.

We believe that the notable high performance of the proposed method is attributable to two reasons. First, our method does not use discretization, being advantageous over the grid-based ICSG3D for estimating crystal structures. In a grid-based method, the spatial resolution is limited by the cubically increasing computations and memory usage. The proposed NeSF is free from such a tradeoff between resolution and computational complexity, and it can thus effectively represent complex structures. Second, the model size of the proposed NeSF using MLPs is much smaller than that of the 3D CNN architecture of ICSG3D. In general, the number of training samples required for an ML model is correlated with the number of trainable parameters. Grid-based methods use layers of 3D convolution filters that involve many trainable parameters. On the other hand, NeSF employs implicit neural representations to indirectly describe the 3D space as a field instead of voxels. Thus, it is efficiently implemented by MLPs with fewer parameters than a 3D CNN. Specifically, the NeSF-based autoencoder has 0.76 million parameters, which is only 2.24% of the number of parameters in the 3D CNN-based ICSG3D (34 million parameters).

4 Conclusion

We propose the NeSF to estimate crystal structures using neural networks. Determining crystal structures directly using neural networks is challenging because these structures are essentially represented as an unordered set with varying numbers of atoms. The NeSF overcomes this difficulty by treating the crystal structure as a continuous vector field rather than as a discrete set of atoms. Unlike existing grid-based approaches for representing crystal structures [10], the NeSF is free from the tradeoff between spatial resolution and computational complexity and can represent any crystal structure. The NeSF was applied as an autoencoder of crystal structures and demonstrated its expressive power. A quantitative performance analysis showed a clear advantage of the NeSF-based autoencoder over an existing grid-based method. Meanwhile, the present study is subject to the following limitations. First, the NeSF does not explicitly consider space-group symmetry. Symmetry is an important concept in crystallography and incorporating the constraint of space-group symmetry into NeSF is thus an important direction of future work. Second, we adopted the autoencoder architecture rather than generative models, such as variational autoencoders [11, 23, 1] and generative adversarial networks [24, 23, 25, 12], for an evaluation purpose. While these generative models are more suitable for novel structure discovery, the lack of ground-truth structures prevents quantitative and reliable performance analysis. The NeSF should be applied to generative models in future work with appropriate performance analysis. Such generative models for crystal structures will be important for inverse design of materials, which is a major challenge in MI.

Acknowledgments

This work is partly supported by JST-Mirai Program, Grant Number JPMJMI19G1 and JST ACT-I grant number JPMJPR18UE. Computational resource of AI Bridging Cloud Infrastructure (ABCI) provided by National Institute of Advanced Industrial Science and Technology (AIST) was used.

References

[1] Xie, T. & Grossman, J. C. Crystal Graph Convolutional Neural Networks for an Accurate and Interpretable Prediction of Material Properties. *Phys. Rev. Lett.* **120**, 145301 (2018).

[2] Chen, C., Ye, W., Zuo, Y., Zheng, C. & Ong, S. P. Graph Networks as a Universal Machine Learning Framework for Molecules and Crystals. *Chem. Mater.* **31**, 3564–3572 (2019).

[3] Park, C. W. & Wolverton, C. Developing an improved crystal graph convolutional neural network framework for accelerated materials discovery. *Phys. Rev. Materials* **4**, 063801 (6 1, 2020).

[4] Cheng, J., Zhang, C. & Dong, L. A geometric-information-enhanced crystal graph network for predicting properties of materials. *Commun Mater* **2**, 1–11 (2021).

[5] Noh, J. *et al.* Inverse Design of Solid-State Materials via a Continuous Representation. *Matter* **1**, 1370–1384 (2019).

[6] Noh, J., Gu, G. H., Kim, S. & Jung, Y. Machine-enabled inverse design of inorganic solid materials: Promises and challenges. *Chem. Sci.* **11**, 4871–4881 (2020).

[7] Yao, Z. *et al.* Inverse design of nanoporous crystalline reticular materials with deep generative models. *Nat. Mach. Intell* **3**, 76–86 (2021).

[8] Long, T. *et al.* Constrained crystals deep convolutional generative adversarial network for the inverse design of crystal structures. *npj Comput Mater* **7**, 1–7 (2021).

[9] Fung, V., Zhang, J., Hu, G., Ganesh, P. & Sumpter, B. G. Inverse design of two-dimensional materials with invertible neural networks. *npj Comput Mater* **7**, 1–9 (2021).

[10] Court, C. J., Yildirim, B., Jain, A. & Cole, J. M. 3-D Inorganic Crystal Structure Generation and Property Prediction via Representation Learning. *J. Chem. Inf. Model.* **60**, 4518–4535 (2020).

[11] Hoffmann, J. *et al.* Data-driven approach to encoding and decoding 3-d crystal structures. *arXiv preprint arXiv:1909.00949* (2019).

[12] Kim, S., Noh, J., Gu, G. H., Aspuru-Guzik, A. & Jung, Y. Generative Adversarial Networks for Crystal Structure Prediction. *ACS Cent. Sci.* **6**, 1412–1420 (2020).

[13] Mildenhall, B. *et al.* NeRF: Representing scenes as neural radiance fields for view synthesis. *arXiv* (2020). `2003.08934`.

[14] Park, J. J., Florence, P., Straub, J., Newcombe, R. & Lovegrove, S. Deepsdf: Learning continuous signed distance functions for shape representation. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (2019).

[15] Chen, Z. & Zhang, H. Learning implicit fields for generative shape modeling. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (2019).

[16] Mescheder, L., Oechsle, M., Niemeyer, M., Nowozin, S. & Geiger, A. Occupancy networks: Learning 3d reconstruction in function space. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (2019).

[17] Xie, Y. *et al.* Neural fields in visual computing and beyond. *Computer Graphics Forum* (2022).

[18] Duan, Y. *et al.* Curriculum deepsdf. In Vedaldi, A., Bischof, H., Brox, T. & Frahm, J.-M. (eds.) *Computer Vision – ECCV 2020*, 51–67 (Springer International Publishing, Cham, 2020).

[19] Charles, R. Q., Su, H., Kaichun, M. & Guibas, L. J. Pointnet: Deep learning on point sets for 3d classification and segmentation. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (2017).

[20] Zaheer, M. *et al.* Deep sets. In *Neural Information Processing Systems* (2017).

[21] Kingma, D. P. & Ba, J. Adam: A method for stochastic optimization. In *The International Conference on Learning Representations (ICLR)* (2015).

[22] Jain, A. *et al.* The Materials Project: A materials genome approach to accelerating materials innovation. *APL Materials* **1**, 011002 (2013).

[23] Goodfellow, I. J., Bengio, Y. & Courville, A. *Deep Learning* (MIT Press, Cambridge, MA, USA, 2016). `http://www.deeplearningbook.org`.

[24] Goodfellow, I. *et al.* Generative adversarial nets. In Ghahramani, Z., Welling, M., Cortes, C., Lawrence, N. & Weinberger, K. (eds.) *Advances in Neural Information Processing Systems*, vol. 27 (Curran Associates, Inc., 2014). URL `https://proceedings.neurips.cc/paper/2014/file/5ca3e9b122f61f8f06494c97b1afccf3-Paper.pdf`.

[25] Nouira, A., Sokolovska, N. & Crivello, J.-C. CrystalGAN: Learning to Discover Crystallographic Structures with Generative Adversarial Networks. *arXiv:1810.11203 [cs, stat]* (2019). `1810.11203`.

Appendix

A Sampling parameters

For the training of the NeSF, we use the global grid sampling and the local grid sampling. Sampling ranges and parameters are shown in followings.

- For position field training:
  - Global grid sampling:
    - Interval: 1 Å
    - σ: 0.5 Å
  - Local grid sampling:
    - Interval: 0.3 Å
    - Range for each axis: 1.2 Å
    - σ: 0.5 Å
- For species field training:
  - Local grid sampling:
    - Interval: 0.1 Å
    - Range for each axis: 0.5 Å
    - σ: 0.5 Å

B Network architecture

The proposed network architecture is specified as follows, where N is the number of atoms in the input crystal structure and A is the number of atomic species in the dataset.

- Encoder
  - Atom position encoder
    - Input: Atom positions: [Nx3]
    - Shared MLP: [Nx3] -> [Nx128]
      - Layers: 3, 64, 128, 128
      - Activation function: ReLU
      - Using batch normalization
      - No dropout
    - Output: Atom position features: [Nx128]
  - Atom species encoder
    - Input: Atom species with one-hot encodings: [NxA]
    - Shared MLP: [NxA] -> [Nx128]
      - Layers: A, 128, 128
      - Activation function: ReLU
      - Using batch normalization
      - No dropout
    - Output: Atom species features: [Nx128]
  - Crystal structure encoder
    - Input: Atom position features and atom species features: [Nx128], [Nx128]
    - Concatenation of atom position features and atom species features: [Nx128], [Nx128] -> [Nx256]
    - Shared MLP: [Nx256] -> [Nx512]
      - Layers: 256, 512
      - Activation function: ReLU
      - Using dropout with 0.3 dropout probability
    - Max-Pooling (along the atoms): [Nx512] -> [512]
    - Concatenation with lattice parameters (length and angles): [512], [6] -> [518]

        - MLP: [518] -> [192]
            - Layers: 518, 256, 256, 192
            - Activation function: ReLU
            - Using batch normalization
            - Using dropout with 0.3 dropout probability
        - Output: Latent vector: [192]
- Decoder
    - Lattice lenght parameter decoder
        - Input: Latent vector: [192]
        - MLP: [192] -> [3]
            - Layers: 192, 128, 3
            - Activation function: ReLU
            - No batch normalization
            - No dropout
        - Output: Lattice length parameter: [3]
    - Lattice angle parameter decoder
        - Input: Latent vector: [192]
        - MLP: [192] -> [3]
            - Layers: 192, 128, 3
            - Activation function: ReLU
            - No batch normalization
            - No dropout
        - Output: Lattice angle parameter: [3]
    - Position field decoder:
        - Input: Latent vector and query position: [192], [3]
        - Concatenation of Latent vector and query position: [192], [3] -> [195]
        - FC: [195] -> [256]
            - Activation function: ReLU
            - Using batch normalization
            - No dropout
        - Concatenation of above output and query position: [256], [3] -> [259]
        - FC: [259] -> [128]
            - Activation function: ReLU
            - Using batch normalization
            - No dropout
        - Concatenation of above output and query position: [128], [3] -> [131]
        - FC: [131] -> [128]
            - Activation function: ReLU
            - Using batch normalization
            - No dropout
        - Concatenation of above output and query position: [128], [3] -> [131]
        - FC: [131] -> [128]
            - Activation function: ReLU
            - Using batch normalization
            - No dropout
        - Concatenation of above output and query position: [128], [3] -> [131]
        - FC: [131] -> [128]
            - Activation function: ReLU
            - Using batch normalization
            - No dropout
        - Concatenation of above output and query position: [128], [3] -> [131]
        - FC: [131] -> [128]
            - Activation function: ReLU
            - Using batch normalization

        - No dropout
    - Concatenation of above output and query position: [128], [3] -> [131]
    - FC: [131] -> [128]
        - Activation function: ReLU
        - Using batch normalization
        - No dropout
    - Concatenation of above output and query position: [128], [3] -> [131]
    - FC: [131] -> [128]
        - Activation function: ReLU
        - Using batch normalization
        - No dropout
    - Concatenation of above output and query position: [128], [3] -> [131]
    - FC: [131] -> [3]
        - Activation function: None
        - No batch normalization
        - No dropout
    - Output: Position field value of the query position: [3]
- Species field decoder:
    - Input: Latent vector and query position: [192], [3]
    - Concatenation of Latent vector and query position: [192], [3] -> [195]
    - FC: [195] -> [256]
        - Activation function: ReLU
        - Using batch normalization
        - No dropout
    - Concatenation of above output and query position: [256], [3] -> [259]
    - FC: [259] -> [128]
        - Activation function: ReLU
        - Using batch normalization
        - No dropout
    - Concatenation of above output and query position: [128], [3] -> [131]
    - FC: [131] -> [A]
        - Activation function: None
        - No batch normalization
        - No dropout
    - Output: Species field value (class likelihood) of the query position: [A]

C Hyperparameter search

We searched hyperparameters by the procedure summarized as the eight steps below. In each step, we evaluated multiple configurations of certain hyperparameters by training and validating models on a dataset and chose the best configuration among them for the next step. We evaluated the Number of Atom Accuracy as the primal performance indicator. When the performance was the same by this metric, the Position Error (Actual) was used as the secondary performance indicator.

Since the ICSG3D dataset [10] is restricted to certain material classes and prototypes, conducting the hyperparameter search on it may result in a model with limited generalizability despite the huge computational effort. We thus prepared a more general dataset, named Limited Cell Size (6 Å) (LCS6Å), for the hyperparameter search. The LCS6Å dataset is a collection of the material entries in the Materials Project whose unit cell sizes are 6 Å or less along the x, y, and z axes. The dataset amounts to 6005 material entries (4.31 % of the Materials Project database) and was split into the training (5418 entries), validation (286 entries), and test (301 entries) sets.

The training of a NeSF model on the LCS6Å dataset took approximately 9 hours using a single NVIDIA RTX8000 GPU (48GB VRAM). The hyperparameter search was carried out on a single NVIDIA DGX-1 system (NVIDIA V100 GPU 32GB VRAM x8), taking approximately 10 days in total.

0. Target parameters and the initial configuration
    - Weights of loss functions: loss_weight.pos=1.0, loss_weight.spec=1.0, loss_weight.abc=1.0, loss_weight.angle=1.0
    - The dimension of the latent vector: latent_size=256
    - The learning rate of the Adam optimizer: training.lr=0.001
    - Layers of the position field decoder: decoder.pos.channels=[256,128,128,128,128,128,128,128,128]
    - Layers of the species field decoder: decoder.spec.channels=[256,128,128,128,128,128,128,128,128]
    - Layers of the atom position encoder: encoder.smlp_pos.channels=[64,128]
    - Layers of the atom species encoder: encoder.smlp_spec.channels=[128]
    - Layers of the crystal structure encoder: encoder.smlp_mix.channels=[512], encoder.mlp.channels=[256]
    - Batch size: batch_size=64
1. Weights of loss functions
    - loss_weight.pos=0.1, loss_weight.spec=0.1, loss_weight.abc=1.0, loss_weight.angle=1.0
    - loss_weight.pos=0.1, loss_weight.spec=1.0, loss_weight.abc=1.0, loss_weight.angle=1.0
    - loss_weight.pos=0.1, loss_weight.spec=10.0, loss_weight.abc=1.0, loss_weight.angle=1.0
    - loss_weight.pos=1.0, loss_weight.spec=0.1, loss_weight.abc=1.0, loss_weight.angle=1.0
    - loss_weight.pos=1.0, loss_weight.spec=1.0, loss_weight.abc=1.0, loss_weight.angle=1.0
    - loss_weight.pos=1.0, loss_weight.spec=10.0, loss_weight.abc=1.0, loss_weight.angle=1.0
    - loss_weight.pos=10.0, loss_weight.spec=0.1, loss_weight.abc=1.0, loss_weight.angle=1.0
    - loss_weight.pos=10.0, loss_weight.spec=1.0, loss_weight.abc=1.0, loss_weight.angle=1.0
    - loss_weight.pos=10.0, loss_weight.spec=10.0, loss_weight.abc=1.0, loss_weight.angle=1.0
2. Size of latent vector
    - latent_size=64
    - latent_size=92
    - latent_size=128
    - latent_size=192
    - latent_size=256
    - latent_size=384
    - latent_size=512
    - latent_size=1024
3. Learning rate
    - training.lr=1.0
    - training.lr=0.1
    - training.lr=0.01
    - training.lr=0.005
    - training.lr=0.001
    - training.lr=0.0005
    - training.lr=0.0001
    - training.lr=0.00001
4. Decoder MLP
    - decoder.pos.channels=[256,128,128,128,128,128,128,128], decoder.spec.channels=[256,128,128,128,128,128,128,128]
    - decoder.pos.channels=[256,128,128,128,128,128,128,128], decoder.spec.channels=[256,128,128,128,128,128]
    - decoder.pos.channels=[256,128,128,128,128,128,128,128], decoder.spec.channels=[256,128,128,128,128]
    - decoder.pos.channels=[256,128,128,128,128,128,128,128], decoder.spec.channels=[256,128,128,128]
    - decoder.pos.channels=[256,128,128,128,128,128,128,128], decoder.spec.channels=[256,128]
    - decoder.pos.channels=[256,128,128,128,128,128], decoder.spec.channels=[256,128,128,128,128,128,128,128]
    - decoder.pos.channels=[256,128,128,128,128,128], decoder.spec.channels=[256,128,128,128,128,128]
    - decoder.pos.channels=[256,128,128,128,128,128], decoder.spec.channels=[256,128,128,128,128]
    - decoder.pos.channels=[256,128,128,128,128,128], decoder.spec.channels=[256,128,128,128]
    - decoder.pos.channels=[256,128,128,128,128,128], decoder.spec.channels=[256,128]
    - decoder.pos.channels=[256,128,128,128,128,128], decoder.spec.channels=[256,128,128,128,128,128,128,128]
    - decoder.pos.channels=[256,128,128,128,128,128], decoder.spec.channels=[256,128,128,128,128,128]
    - decoder.pos.channels=[256,128,128,128,128,128], decoder.spec.channels=[256,128,128,128,128]
    - decoder.pos.channels=[256,128,128,128,128,128], decoder.spec.channels=[256,128,128,128]
    - decoder.pos.channels=[256,128,128,128,128,128], decoder.spec.channels=[256,128]
    - decoder.pos.channels=[256,128,128,128], decoder.spec.channels=[256,128,128,128,128,128,128,128]
    - decoder.pos.channels=[256,128,128,128], decoder.spec.channels=[256,128,128,128,128,128]
    - decoder.pos.channels=[256,128,128,128], decoder.spec.channels=[256,128,128,128,128]
    - decoder.pos.channels=[256,128,128,128], decoder.spec.channels=[256,128,128,128]
    - decoder.pos.channels=[256,128], decoder.spec.channels=[256,128,128,128,128,128,128,128]
    - decoder.pos.channels=[256,128], decoder.spec.channels=[256,128,128,128,128,128]
    - decoder.pos.channels=[256,128], decoder.spec.channels=[256,128,128,128,128]
    - decoder.pos.channels=[256,128,128,128], decoder.spec.channels=[256,128]
    - decoder.pos.channels=[256,128], decoder.spec.channels=[256,128,128,128]
    - decoder.pos.channels=[256,128], decoder.spec.channels=[256,128]

5. Encoder MLP
    - encoder.smlp_pos.channels=[64,128], encoder.smlp_spec.channels=[128]
    - encoder.smlp_pos.channels=[64,128], encoder.smlp_spec.channels=[128,128,128]
    - encoder.smlp_pos.channels=[64,128,128], encoder.smlp_spec.channels=[128]
    - encoder.smlp_pos.channels=[64,128,128], encoder.smlp_spec.channels=[128,128]
    - encoder.smlp_pos.channels=[64,128,128], encoder.smlp_spec.channels=[128,128,128]
    - encoder.smlp_pos.channels=[64,128,128,128], encoder.smlp_spec.channels=[128]
    - encoder.smlp_pos.channels=[64,128,128,128], encoder.smlp_spec.channels=[128,128]
    - encoder.smlp_pos.channels=[64,128,128,128], encoder.smlp_spec.channels=[128,128,128]
    - encoder.smlp_pos.channels=[64,128], encoder.smlp_spec.channels=[128,128]
6. Crystal structure encoder
    - encoder.smlp_mix.channels=[256], encoder.mlp.channels=[512]
    - encoder.smlp_mix.channels=[256], encoder.mlp.channels=[512,512,256]
    - encoder.smlp_mix.channels=[256,256], encoder.mlp.channels=[512]
    - encoder.smlp_mix.channels=[256,256], encoder.mlp.channels=[512,512]
    - encoder.smlp_mix.channels=[256,256], encoder.mlp.channels=[512,512,256]
    - encoder.smlp_mix.channels=[256,256,512], encoder.mlp.channels=[512]
    - encoder.smlp_mix.channels=[256,256,512], encoder.mlp.channels=[512,512]
    - encoder.smlp_mix.channels=[256,256,512], encoder.mlp.channels=[512,512,256]
    - encoder.smlp_mix.channels=[256], encoder.mlp.channels=[512,512]
7. Batch size
    - batch_size=2
    - batch_size=4
    - batch_size=8
    - batch_size=16
    - batch_size=32
    - batch_size=64
    - batch_size=128
    - batch_size=256
8. Total verification[2]
    - latent_size=192, training.lr=0.01, loss_weight.pos=0.1, loss_weight.spec=0.1
    - latent_size=192, training.lr=0.01, loss_weight.pos=0.1, loss_weight.spec=1.0
    - latent_size=192, training.lr=0.01, loss_weight.pos=0.1, loss_weight.spec=10.0
    - latent_size=192, training.lr=0.01, loss_weight.pos=1.0, loss_weight.spec=0.1
    - latent_size=192, training.lr=0.01, loss_weight.pos=1.0, loss_weight.spec=1.0
    - latent_size=192, training.lr=0.01, loss_weight.pos=1.0, loss_weight.spec=10.0
    - latent_size=192, training.lr=0.01, loss_weight.pos=10.0, loss_weight.spec=0.1
    - latent_size=192, training.lr=0.01, loss_weight.pos=10.0, loss_weight.spec=1.0
    - latent_size=192, training.lr=0.01, loss_weight.pos=10.0, loss_weight.spec=10.0
    - latent_size=192, training.lr=0.001, loss_weight.pos=0.1, loss_weight.spec=0.1
    - latent_size=192, training.lr=0.001, loss_weight.pos=0.1, loss_weight.spec=1.0
    - latent_size=192, training.lr=0.001, loss_weight.pos=0.1, loss_weight.spec=10.0
    - latent_size=192, training.lr=0.001, loss_weight.pos=1.0, loss_weight.spec=0.1
    - latent_size=192, training.lr=0.001, loss_weight.pos=1.0, loss_weight.spec=1.0
    - latent_size=192, training.lr=0.001, loss_weight.pos=1.0, loss_weight.spec=10.0
    - latent_size=192, training.lr=0.001, loss_weight.pos=10.0, loss_weight.spec=0.1
    - latent_size=192, training.lr=0.001, loss_weight.pos=10.0, loss_weight.spec=1.0
    - latent_size=192, training.lr=0.001, loss_weight.pos=10.0, loss_weight.spec=10.0
    - latent_size=192, training.lr=0.0001, loss_weight.pos=0.1, loss_weight.spec=0.1
    - latent_size=192, training.lr=0.0001, loss_weight.pos=0.1, loss_weight.spec=1.0
    - latent_size=192, training.lr=0.0001, loss_weight.pos=0.1, loss_weight.spec=10.0
    - latent_size=192, training.lr=0.0001, loss_weight.pos=1.0, loss_weight.spec=0.1
    - latent_size=192, training.lr=0.0001, loss_weight.pos=1.0, loss_weight.spec=1.0
    - latent_size=192, training.lr=0.0001, loss_weight.pos=1.0, loss_weight.spec=10.0
    - latent_size=192, training.lr=0.0001, loss_weight.pos=10.0, loss_weight.spec=0.1
    - latent_size=192, training.lr=0.0001, loss_weight.pos=10.0, loss_weight.spec=1.0
    - latent_size=192, training.lr=0.0001, loss_weight.pos=10.0, loss_weight.spec=10.0
    - latent_size=256, training.lr=0.01, loss_weight.pos=0.1, loss_weight.spec=0.1
    - latent_size=256, training.lr=0.01, loss_weight.pos=0.1, loss_weight.spec=1.0
    - latent_size=256, training.lr=0.01, loss_weight.pos=0.1, loss_weight.spec=10.0
    - latent_size=256, training.lr=0.01, loss_weight.pos=1.0, loss_weight.spec=0.1

[2]We performed this step to try to further tune hyperparameters, although no configuration evaluated in this step was adopted as the final result after all.

- latent_size=256, training.lr=0.01, loss_weight.pos=1.0, loss_weight.spec=1.0
- latent_size=256, training.lr=0.01, loss_weight.pos=1.0, loss_weight.spec=10.0
- latent_size=256, training.lr=0.01, loss_weight.pos=10.0, loss_weight.spec=0.1
- latent_size=256, training.lr=0.01, loss_weight.pos=10.0, loss_weight.spec=1.0
- latent_size=256, training.lr=0.01, loss_weight.pos=10.0, loss_weight.spec=10.0
- latent_size=256, training.lr=0.001, loss_weight.pos=0.1, loss_weight.spec=0.1
- latent_size=256, training.lr=0.001, loss_weight.pos=0.1, loss_weight.spec=1.0
- latent_size=256, training.lr=0.001, loss_weight.pos=0.1, loss_weight.spec=10.0
- latent_size=256, training.lr=0.001, loss_weight.pos=1.0, loss_weight.spec=0.1
- latent_size=256, training.lr=0.001, loss_weight.pos=1.0, loss_weight.spec=1.0
- latent_size=256, training.lr=0.001, loss_weight.pos=1.0, loss_weight.spec=10.0
- latent_size=256, training.lr=0.001, loss_weight.pos=10.0, loss_weight.spec=0.1
- latent_size=256, training.lr=0.001, loss_weight.pos=10.0, loss_weight.spec=1.0
- latent_size=256, training.lr=0.001, loss_weight.pos=10.0, loss_weight.spec=10.0
- latent_size=256, training.lr=0.0001, loss_weight.pos=0.1, loss_weight.spec=0.1
- latent_size=256, training.lr=0.0001, loss_weight.pos=0.1, loss_weight.spec=1.0
- latent_size=256, training.lr=0.0001, loss_weight.pos=0.1, loss_weight.spec=10.0
- latent_size=256, training.lr=0.0001, loss_weight.pos=1.0, loss_weight.spec=0.1
- latent_size=256, training.lr=0.0001, loss_weight.pos=1.0, loss_weight.spec=1.0
- latent_size=256, training.lr=0.0001, loss_weight.pos=1.0, loss_weight.spec=10.0
- latent_size=256, training.lr=0.0001, loss_weight.pos=10.0, loss_weight.spec=0.1
- latent_size=256, training.lr=0.0001, loss_weight.pos=10.0, loss_weight.spec=1.0
- latent_size=256, training.lr=0.0001, loss_weight.pos=10.0, loss_weight.spec=10.0
- latent_size=384, training.lr=0.01, loss_weight.pos=0.1, loss_weight.spec=0.1
- latent_size=384, training.lr=0.01, loss_weight.pos=0.1, loss_weight.spec=1.0
- latent_size=384, training.lr=0.01, loss_weight.pos=0.1, loss_weight.spec=10.0
- latent_size=384, training.lr=0.01, loss_weight.pos=1.0, loss_weight.spec=0.1
- latent_size=384, training.lr=0.01, loss_weight.pos=1.0, loss_weight.spec=1.0
- latent_size=384, training.lr=0.01, loss_weight.pos=1.0, loss_weight.spec=10.0
- latent_size=384, training.lr=0.01, loss_weight.pos=10.0, loss_weight.spec=0.1
- latent_size=384, training.lr=0.01, loss_weight.pos=10.0, loss_weight.spec=1.0
- latent_size=384, training.lr=0.01, loss_weight.pos=10.0, loss_weight.spec=10.0
- latent_size=384, training.lr=0.001, loss_weight.pos=0.1, loss_weight.spec=0.1
- latent_size=384, training.lr=0.001, loss_weight.pos=0.1, loss_weight.spec=1.0
- latent_size=384, training.lr=0.001, loss_weight.pos=0.1, loss_weight.spec=10.0
- latent_size=384, training.lr=0.001, loss_weight.pos=1.0, loss_weight.spec=0.1
- latent_size=384, training.lr=0.001, loss_weight.pos=1.0, loss_weight.spec=1.0
- latent_size=384, training.lr=0.001, loss_weight.pos=1.0, loss_weight.spec=10.0
- latent_size=384, training.lr=0.001, loss_weight.pos=10.0, loss_weight.spec=0.1
- latent_size=384, training.lr=0.001, loss_weight.pos=10.0, loss_weight.spec=1.0
- latent_size=384, training.lr=0.001, loss_weight.pos=10.0, loss_weight.spec=10.0
- latent_size=384, training.lr=0.0001, loss_weight.pos=0.1, loss_weight.spec=0.1
- latent_size=384, training.lr=0.0001, loss_weight.pos=0.1, loss_weight.spec=1.0
- latent_size=384, training.lr=0.0001, loss_weight.pos=0.1, loss_weight.spec=10.0
- latent_size=384, training.lr=0.0001, loss_weight.pos=1.0, loss_weight.spec=0.1
- latent_size=384, training.lr=0.0001, loss_weight.pos=1.0, loss_weight.spec=1.0
- latent_size=384, training.lr=0.0001, loss_weight.pos=1.0, loss_weight.spec=10.0
- latent_size=384, training.lr=0.0001, loss_weight.pos=10.0, loss_weight.spec=0.1
- latent_size=384, training.lr=0.0001, loss_weight.pos=10.0, loss_weight.spec=1.0
- latent_size=384, training.lr=0.0001, loss_weight.pos=10.0, loss_weight.spec=10.0

For ICSG3D, we also tuned the default hyperparameters provided by the authors for the LCS6Å dataset, by searching for the best configuration from the list below.

- batch_size=10, latent_size=1024
- batch_size=10, latent_size=128
- batch_size=10, latent_size=256
- batch_size=10, latent_size=512
- batch_size=10, latent_size=64
- batch_size=1, latent_size=1024
- batch_size=1, latent_size=128
- batch_size=1, latent_size=256
- batch_size=1, latent_size=512
- batch_size=1, latent_size=64

- batch_size=20, latent_size=1024
- batch_size=20, latent_size=128
- batch_size=20, latent_size=256
- batch_size=20, latent_size=512
- batch_size=20, latent_size=64
- batch_size=40, latent_size=1024
- batch_size=40, latent_size=128
- batch_size=40, latent_size=256
- batch_size=40, latent_size=512
- batch_size=40, latent_size=64
- batch_size=5, latent_size=1024
- batch_size=5, latent_size=128
- batch_size=5, latent_size=256
- batch_size=5, latent_size=512
- batch_size=5, latent_size=64

D ICSG3D Dataset Reproduction

We reproduced the dataset used for ICSG3D [10] by curating crystal structure data from the Materials Project [22], a public online database distributed under Creative Commons Attribution 4.0 License. Unfortunately, the authors of ICSG3D do not provide specific material IDs used for their experiments. Therefore, by following the procedures in the paper, we crawled the three material classes having cubic structures, *i.e.*, binary alloys (AB), ternary perovskites (ABX_3), and Heusler compounds (ABX_2), from the Materials Project. We consequently obtained 7896 material entries, which were split into the training (7194 entries), validation (342 entries), and test (360 entries) sets.

E Potential Negative Societal Impacts

We are aware of some potential negative societal impacts of our work. Since our work proposes a fundamental idea for the decoding of 3D structures consisting of atoms, it may be broadly related to ML-based approaches for the discovery of novel materials and chemical substances. Therefore, misuses of our idea may lead to the designing of substances that are harmful to humans.