Neural Structure Fields with Application to Crystal Structure Auto-Encoders

Anonymous Author(s)
Affiliation
Address
email

Abstract

Representing crystal structures of materials in a form preferable for neural networks is crucial for enabling machine learning applications involving the estimation of crystal structures. This paper proposes *Neural Structure Fields* (NeSF) as an accurate and practical approach to representing crystal structures by neural networks. Our crucial idea, inspired by the concepts of both vector fields in physics and implicit neural representations in computer vision, is to consider a crystal structure as a continuous field rather than a discrete set of atoms. Unlike existing grid-based discretized spatial representations, NeSF is free from a trade-off between spatial resolution and computational complexity and can represent complex crystal structures. We demonstrate the expressibility of NeSF successfully in applying it to auto-encoding of crystal structures.

1 Introduction

Representing crystal structures in a neural network-familiar expression has excellent potential utility in machine learning (ML)-assisted material development. For example, graph-based representations of crystal structures have allowed the encoding of crystal structures via graph neural networks (GNN) [1, 2, 3, 4] for high-throughput material property predictions. When this type of task is considered as the discovery of *structure-to-property relationships* among materials, there is yet another important type of task in materials informatics (MI): the estimation of crystal structures under given property criteria [5, 6, 7, 8, 9] (or the discovery of *property-to-structure relationships* as in the inverse design of materials.) However, there is still a technical bottleneck in decoding crystal structures by neural networks, which we address in this study.

The crystal structure of inorganic materials is a regular and periodic arrangement of atoms in the three-dimensional (3D) space. This arrangement is usually described by the 3D positions and species of the atoms in a unit cell and the lattice constants defining the unit cell translations in the 3D space. Here, the atoms in a unit cell have no permutation, and their number varies among materials from one to hundreds. Because ML models, including neural networks, generally prefer fixed-dimensional, consistently ordered tensors as the computation units, treating crystal structures with ML models is not straightforward [1] and outputting them is even more difficult.

In this paper, we propose a continuous vector field-based crystal structure representation named Neural Structure Fields (NeSF). The idea of NeSF is inspired by the concepts of both vector fields in classical physics and implicit neural representations (INRs) [10, 11, 12, 13, 14] in computer vision. Specifically, INRs have recently been proposed to handle representation issues in 3D computer vision applications such as 3D shape representation of objects [11, 12, 13] and free-viewpoint image synthesis [10, 14]. Our NeSF follows the basic idea of INRs and further extends it for estimating crystal structures, *i.e.*, atomic positions and species.

Submitted to 36th Conference on Neural Information Processing Systems (NeurIPS 2022). Do not distribute.

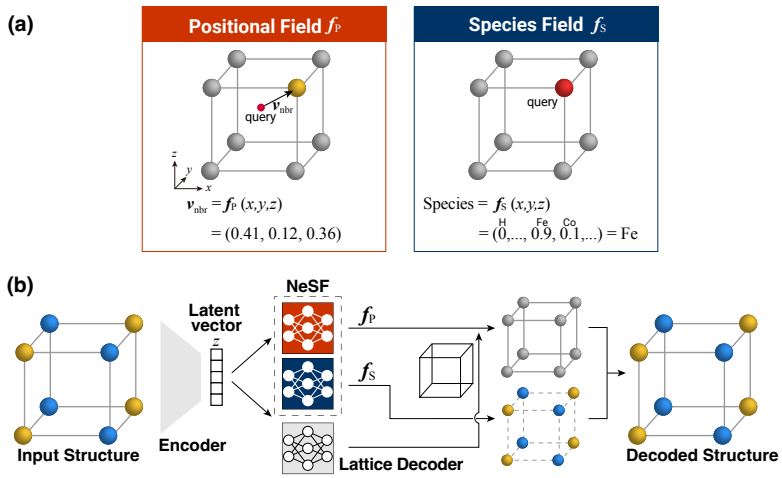

Figure 1: **Neural Structure Fields (NeSF) and the crystal structure auto-encoder using NeSF**. **a**, NeSF consists of two vector fields, the Position Field $\boldsymbol{f}_{\mathrm{p}}$ and Species Field $\boldsymbol{f}_{\mathrm{s}}$, both defined in the 3D space. **b**, The network architecture of our NeSF-based crystal structure auto-encoder.

Our idea of representing crystal structures as continuous vector fields has been partly and implicitly explored by grid-based discretization (*i.e.*, voxelization) approaches in some existing MI studies [5, 6, 15, 8], although those authors did not explicitly explain them as discretized vector fields. In these approaches, they discretize the 3D space within the unit cell with voxels and then assign each voxel an electron density, which essentially represents the presence or absence of an atom around the voxel. However, the discretization of 3D data severely suffers from a trade-off between spatial resolution and computational complexity in terms of computational time and memory space. For instance, an existing grid-based method (ICSG3D [16]) based on 3D convolutional neural networks (CNNs) uses a limited resolution of $32 \times 32 \times 32$ voxels.

NeSF overcomes this limitation of the voxelization approach. Theoretically, NeSF can enjoy infinitely high spatial resolution with compact (memory and parameter-efficient) neural networks in place of costly 3D CNNs. Our NeSF-based auto-encoder has 0.76 million parameters, which is only 2.2 % of the number of parameters used in the 3D CNN-based ICSG3D (34 million parameters.)

2 Method

In this section, we describe how NeSF represents crystal structures, how to train NeSF, and how to reconstruct crystal structures from NeSF.

2.1 Crystal structure representation by NeSF

Here we present our idea of NeSF that enables neural networks to decode or output crystal structures. As illustrated in Fig. 1, NeSF consists of two kinds of vector fields, the Position Field and the Species Field, to implicitly represent the atoms' positions and species in the crystal structure's unit cell.

To illustrate the idea of NeSF, let us assume that we are given some target material information as a fixed-dimensional vector $\boldsymbol{z}$ and consider the problem of recovering a crystal structure from $\boldsymbol{z}$. In NeSF, we use a neural network $\boldsymbol{f}$ as an implicit function to indirectly represent the crystal structure embedded in $\boldsymbol{z}$, rather than letting a 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 case of the Position Field, the network $\boldsymbol{f}$ is trained to output the 3D pointing vector from a query point $\boldsymbol{p}$ to its nearest atom's position, $\boldsymbol{a}$, in the crystal structure of interest. Thus, we expect the output $\boldsymbol{s}$ to be $\boldsymbol{a} - \boldsymbol{p}$. If the Position Field is trained ideally, we can retrieve the position of the nearest atom $\boldsymbol{a}$ 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 crucial to materials science. By outputting pointing vectors of nearest atoms, NeSF can more directly represent atomic positions than existing geometric INRs that represent 3D surfaces [11, 12, 13]. Similarly, the Species Field is trained to output a categorical probability distribution indicating the species of the nearest atom. Thus, the output dimension of the Species Field is the number of candidate atom species.

The training procedure of NeSF is simple and similar to those for existing INRs [11, 17]. We randomly sample 3D query points $\{\boldsymbol{p}_i^s\}$ in the unit cell and compute losses 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. To train NeSF efficiently, we introduce two types of sampling schemes. 1) Global grid sampling considers 3D grid points that uniformly cover the entire unit cell and samples the points with perturbations according to a Gaussian distribution. 2) Local grid sampling considers local 3D grid points centered at each atomic position and samples the points with perturbations according to a Gaussian distribution. For training the Position Field, we use both sampling methods in combination. This way, query points are sampled uniformly over the entire unit cell and, at the same time, densely around the atoms. For training the Species Field, we use the local grid sampling to concentrate training query points in the neighborhood of atoms. See Appendix A for more details.

2.2 Crystal structure estimation with NeSF

Given target material information as a vector $\boldsymbol{z}$, estimating a 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, c and angles α, β, γ, and are estimated via simple multi-layer perceptrons (MLPs) taking $\boldsymbol{z}$ as input. On the other hand, the atomic positions and species are estimated by the Position Field $\boldsymbol{f}_\mathrm{p}$ and the Species Field $\boldsymbol{f}_\mathrm{s}$ of NeSF, respectively. These fields are also implemented as simple MLPs, each taking a query position $\boldsymbol{p}$ and the vector $\boldsymbol{z}$ as input and predicting a field value (*i.e.*, either a 3D pointing vector or a categorical probability distribution) as output. The overview of the network architecture of NeSF is illustrated in the right part of Fig. 1b.

The procedure for estimating the atomic positions and species with NeSF is summarized below.

1. **Initiate particles.** We first estimate the lattice constants via MLPs to obtain an estimate of the unit cell. Then, we regularly spread initial query points $\{\boldsymbol{p}_i^0\}$ (which we call *particles*) within the bounding box region of the estimated unit cell.
2. **Move particles.** We update the position of each particle $\boldsymbol{p}_i^t$ using the Position Field as $\boldsymbol{p}_i^{t+1} = \boldsymbol{p}_i^t + \boldsymbol{f}_\mathrm{p}(\boldsymbol{p}_i^t, \boldsymbol{z})$. We iterate this process for all the particles to get $\{\boldsymbol{p}_i\}$ as candidate atomic positions. The particles will travel toward their nearest atoms through this process.
3. **Score particles.** We score each particle $\boldsymbol{p}_i$ and filter out 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 reject it if the score is above a certain threshold (which we set to 0.9 Å).
4. **Detect atoms.** By the steps, the particles are expected to form clusters around atoms. Here, we apply the non-max suppression with a suppressing radius of 0.5 Å to detect each cluster position as an atomic position.
5. **Estimate species.** Finally, we use the Species Field $\boldsymbol{f}_\mathrm{s}(\boldsymbol{p}, \boldsymbol{z})$ to estimate the atomic species at each atomic position $\boldsymbol{a}_i$. To make the estimation robust to errors in $\boldsymbol{a}_i$, we spread new particles intensively around each $\boldsymbol{a}_i$ as queries to the Species Field. We then select the most frequent atomic species as the final estimate.

2.3 Crystal structure auto-encoder

As an application of NeSF, we present an auto-encoder of crystal structures. Like other common auto-encoders, the proposed NeSF-based auto-encoder consists of an encoder and a decoder. The encoder is a neural network that transforms an input crystal structure (*i.e.*, the positions and species of the atoms in the unit cell and the lattice constants) into an abstract latent vector $\boldsymbol{z}$. Here, we borrow a basic network architecture from PointNet [18] and DeepSets [19] as the simplest type of encoder applicable to crystal structures. The decoder, for which we use NeSF, reconstructs the input crystal structure from the latent vector $\boldsymbol{z}$. The implementation details are described in Appendix B.

In training, the following loss function evaluated the four kinds of outputs from the decoder (*i.e.*, the atomic position, species, lattice length parameters, and angle parameters).

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

Here, L_{pos} is the Mean Squared Error (MSE) loss 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 MSE losses for the lattice length and angle parameters. The total loss function L is given as their weighted summation with the weights $(\lambda_{\text{pos}}, \lambda_{\text{spe}}, \lambda_{\text{len}}, \lambda_{\text{ang}}) = (10, 0.1, 1, 1)$. We optimized the loss function using stochastic gradient descent with a batch size of 128. We used the Adam optimizer [20] with the initial learning rate of 10^{-3} and decayed it every 640 epochs by a factor of 0.5. We conducted iterative training for a total of 3200 epochs for training data in a dataset with early stopping. These hyperparameters were chosen by the procedure described in Appendix C. To reduce performance variation due to randomness, we repeated the whole training and evaluation process ten times with different random seeds and evaluated the average performance.

3 Results

We compared the performance of our method with ICSG3D [16] as a grid-based baseline method. We reproduced the ICSG3D dataset [16] for training and evaluation by curating 6005 material entries from the Materials Project [21] (see Appendix D for details).

The reconstruction accuracy was measured in terms of the Position Error, the Number of Atom Accuracy, and the Species Accuracy. The Position Error is the average error of the reconstructed atomic positions. The Number of Atom Accuracy is the rate of the materials for which the number of atoms was correctly estimated. The Species Accuracy is the average rate of the atoms whose species were correctly estimated. Here, depending on the denominator of the metrics, the Position Error and the Species Accuracy were evaluated in two ways: at the actual atomic positions of the crystal structure (*Actual*) and the estimated atomic positions (*Detected*). The better a method performs, the lower the Position Error and the higher the Number of Atom Accuracy and the Species Accuracy.

Table 1: Reconstruction results (mean $\pm$ standard deviation) on the ICSG3D dataset [16].

Method	Proposed	ICSG3D [16]
Position Error (Actual) [Å]	**0.0308** ± 0.0112	0.0877 ± 0.0306
Position Error (Detected) [Å]	**0.0359** ± 0.0226	0.1057 ± 0.0284
Number of Atom Accuracy [%]	**99.47** ± 0.25	97.33 ± 0.84
Species Accuracy (Actual) [%]	**95.69** ± 0.39	35.61 ± 1.91
Species Accuracy (Detected) [%]	**95.64** ± 0.39	34.95 ± 1.85

Table 1 reports reconstruction accuracies of the proposed method and ICSG3D on the test set. The proposed method consistently outperformed ICSG3D in all the evaluation metrics, with significant performance gains in the Species Accuracy. Meanwhile, ICSG3D also performed relatively well in the Position Error and the Number of Atom Accuracy. The dataset [16] 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 reconstruction performance on more diverse crystal structures is future work. We further discuss potential negative societal impacts of this work in Appendix E.

4 Conclusion

In this study, we have proposed Neural Structure Fields (NeSF) as a method to estimate crystal structures using neural networks. Outputting crystal structures directly with neural networks is challenging because crystal structures are essentially represented as an unordered set of a variable number of atoms. NeSF overcomes this difficulty by treating a crystal structure as a continuous vector field rather than as a discrete set of atoms. Unlike existing grid-based approaches to representing crystal structures [16], NeSF is free from the trade-off between spatial resolution and computational complexity and can represent complex crystal structures. As an application of NeSF, we have presented an auto-encoder of crystal structures. A quantitative comparison with an existing grid-based auto-encoder [16] has shown superior performance of the proposed approach.

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). URL `https://www.nature.com/articles/s43246-021-00194-3`.

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

[11] 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).

[12] 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).

[13] 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).

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

[15] 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).

[16] 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).

[17] 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).

[18] 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).

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

[20] Kingma, D. P. & Ba, J. Adam: A method for stochastic optimization. In *The International Conference on Learning Representations (ICLR)* (2015). URL `http://arxiv.org/abs/1412.6980`.

[21] Jain, A. *et al.* The Materials Project: A materials genome approach to accelerating materials innovation. *APL Materials* **1**, 011002 (2013). URL `https://materialsproject.org/`.

Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes]
   (b) Did you describe the limitations of your work? [Yes] See Section 3.
   (c) Did you discuss any potential negative societal impacts of your work? [Yes] See Appendix E.

(d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]

2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [N/A]
   (b) Did you include complete proofs of all theoretical results? [N/A]
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes] See Appendix F.
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes] See Appendix D for dataset reproduction and splits, Appendix B for detailed network architecture, Section 2.3 and Appendix A for other hyperparameters, and Appendix C for hyperparameter search.
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [Yes] See Table 1.
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes] See Appendix C.
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [Yes] See the citation to the Materials Project [21] in Section 3.
   (b) Did you mention the license of the assets? [Yes] See Appendix D.
   (c) Did you include any new assets either in the supplemental material or as a URL? [Yes] See Appendix F.
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [Yes] Creative Commons Attribution 4.0 License noted in Appendix D implies the consent.
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A] Crystal structure data contains obviously no personally identifiable information or offensive content.
5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]

Appendix

A Sampling parameters

For the training of 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 [16] 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], decoder.spec.channels=[256,128,128,128,128,128,128,128]
   - decoder.pos.channels=[256,128,128,128,128], decoder.spec.channels=[256,128,128,128,128,128]
   - decoder.pos.channels=[256,128,128,128,128], decoder.spec.channels=[256,128,128,128,128]
   - decoder.pos.channels=[256,128,128,128,128], decoder.spec.channels=[256,128,128,128]
   - decoder.pos.channels=[256,128,128,128,128], 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], decoder.spec.channels=[256,128,128,128]
   - 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[1]
   - 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
   - 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

[1] 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=0.0001, 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 [16] by curating crystal structure data from the Materials Project [21], 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.

F Code

For reviewing, the code is available:
`https://drive.google.com/file/d/1_Z2lyr8FyvFvDbaPoztw6qvdkseJAUyf/view?usp=sharing`