Supplementary Information

Deep learning prediction of electrode voltage for metal-ions batteries

Zhenkun Cai, Da Liu, Yufei He, Jihuang Jiao, Jin Zhou, and Renbing Wu*

Department of Materials Science, Fudan University, Shanghai 200438, P. R. China

*E-mail: rbwu@fudan.edu.cn

Table S1 Prediction results for electrode materials in the test dataset

Electrode		Voltage	
Charged	**Discharged**	**DET (V)**	**Deep learning (V)**
$FePO_4F$	$LiFePO_4F$	6.83	6.61
$Co_3(P_2O_7)_2$	$Li_2Co_3(P_2O_7)_2$	6.35	6.34
V_4OF_{11}	LiV_4OF_{11}	6.39	6.28
$Cr(PO_3)_4$	$Li\ Cr\ (PO_3)_4$	5.97	5.92
$Co(CO_3)_2$	$Li_2Co\ (CO_3)_2$	5.87	5.87
NiF_3	$LiNiF_3$	5.91	5.84
CrF_6	Cs_2CrF_6	5.87	5.78
MnF_4	$LiMnF_4$	5.61	5.69
$Cr\ (NO_3)_5$	$Cs_2\,Cr\ (NO_3)_5$	5.64	5.66
$Cu\ (PO_3)_3$	$Li\ Cu\ (PO_3)_3$	5.62	5.61
CrP_2O_7	$LiCrP_2O_7$	5.51	5.48
$Mn\ (PO_3)_5$	$Na_2\,Mn\ (PO_3)_5$	5.52	5.46
CrF_6	Rb_3CrF_6	5.43	5.38
$Cr\ (SO_4)_2$	$Li\ Cr\ (SO_4)_2$	5.37	5.36
$V(PO_3)_5$	$Li_2V(PO_3)_5$	5.37	5.36
$Ni_2(PO_3)_5$	$LiNi_2(PO_3)_5$	5.44	5.32
$Fe\ (PO_3)_4$	$Li\ Fe\ (PO_3)_4$	5.37	5.32
$Ni_2P_2O_9$	$LiNi_2P_2O_9$	5.27	5.21
$Cr\ (PO_3)_4$	$Li\ Cr\ (PO_3)_4$	5.23	5.18
$Cu\ (PO_3)_3$	$Na\ Cu\ (PO_3)_3$	5.16	5.12
$Mn\ (SO_4)_2$	$Li\ Mn\ (SO_4)_2$	5.11	5.11
$NiPO_4$	$LiNiPO_4$	5.07	5.09
$Ni_2P_5O_{16}$	$Li_3Ni_2P_5O_{16}$	5.07	5.08
FeP_2O_7	$LiFeP_2O_7$	5.05	5.05
CrP_2O_7	$LiCrP_2O_7$	5.03	5.02
$CoH_8(SO_6)_2$	$Na_2CoH_8(SO_6)_2$	5.00	5.01

One-hot encoding: One-hot encoding is a technique used to transform categorical variables into binary vectors. Given N distinct categories, each category is represented as a binary vector of length N, with only one element set to 1 and the remaining elements set to 0. Specifically, for category C, its one-hot encoding is represented as:

$$One - Hot(C) = [0,\ldots,1,\ldots,0]\#(1)$$

where the 1 appears at the position corresponding to category C, and all other positions are set to 0.

One-hot encoding is widely applicable in machine learning, particularly in deep learning, and is often used as a standard approach for handling discrete features.

Embedding Layer Encoding: Embedding layer encoding is a technique used to transform categorical variables (into dense, low-dimensional vectors. It is widely employed in natural language processing, recommender systems, and other applications involving high-dimensional feature spaces. An embedding layer is a learnable parameter matrix designed to address the issues of high-dimensional sparsity and the lack of semantic information typically encountered with traditional one-hot encoding and label encoding methods. The core concept of the embedding layer is to map each category into a continuous low-dimensional vector, which can be optimized through backpropagation during model training. Suppose we have N categories, each of which is mapped by the embedding layer into a vector space of size d, where d is a user-defined dimension (typically much smaller than N). Each row of the embedding matrix represents an embedding vector corresponding to a category, as shown below:

$$v_{c_i} = W_{embed}[i]\#(2)$$

where W_{embed} is the embedding matrix with size N × d, i is the index of the category; and v_i is the embedding vector of category i.

Principal Component Analysis (PCA): PCA is a statistical technique based on linear algebra that aims to reduce the dimensionality of high-dimensional datasets while preserving as much of the original information as possible. The fundamental principle of PCA lies in analyzing the covariance matrix of the data to identify the directions of maximum variance, known as principal components. These principal components are derived through eigenvalue decomposition or singular value decomposition (SVD), where the eigenvectors represent the directions of the largest data variability and the eigenvalues quantify the amount of variance explained by each component. The components are ranked in descending order of explained variance, with the first principal component capturing the highest variance, followed by the subsequent components capturing progressively smaller, orthogonal patterns of variance. An essential feature of PCA is the orthogonality of the principal components, ensuring that each component represents unique and independent information. By selecting a subset of principal components that account for the majority of the variance, PCA reduces data dimensionality while mitigating redundancy and noise. The process involves centering the data to ensure zero mean, calculating the covariance matrix to determine feature correlations, and projecting the original data onto the lower-dimensional space spanned by the selected principal components. The essence of PCA is to identify a new orthogonal coordinate system where the projections of the data maximize the retained variance, thereby uncovering dominant patterns and structures in the data. With its rigorous mathematical foundation and capability to distill critical features, PCA is a powerful tool widely employed in materials science for analyzing complex datasets, revealing intrinsic relationships, and facilitating quantitative insights.

Multi-Head Attention: The multi-head attention mechanism is a fundamental component of the Transformer model[2] that enhances its expressive power by parallelizing multiple attention mechanisms. Compared to a single attention mechanism, multi-head attention enables the model to capture information from different subspaces, allowing it to focus more precisely on distinct features in the

input sequence. The core idea behind multi-head attention is to compute the input query, key, and value vectors through different heads, each performing independent attention computations. Each head utilizes a distinct linear transformation matrix to capture different subspace features of the input vectors. Ultimately, the outputs of all the heads are concatenated, and a linear transformation is applied to generate the final output.

Single-head attention is calculated as:

$$Attention(Q,K,V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V\#(4)$$

where Q is the query matrix, K is the key matrix, and V is the value matrix. d_k represents the dimension of the key vector and serves as a scaling factor to prevent excessively large dot product values. The softmax function normalizes the result, producing a weight distribution that is used to weight the value matrix V.

The formula for multi-head attention is as follows:

$$MultiHead(Q,K,V) = Concat\left(head_1,head_2,\ldots,head_h\right)W^O\#(5)$$

Each head is calculated as:

$$head_i = Attention\left(QW_i^Q,KW_i^K,VW_i^V\right)\#(6)$$

where W_i^Q,W_i^K,W_i^V are the linear transformation matrices for the query, key, and value, respectively, and W^O is the linear transformation matrix for the output. Here, h denotes the number of heads. Each head learns different subspace features using distinct weight matrices W_i^Q,W_i^K,W_i^V, and the results of all heads are concatenated and linearly transformed by W^O to obtain the final output.

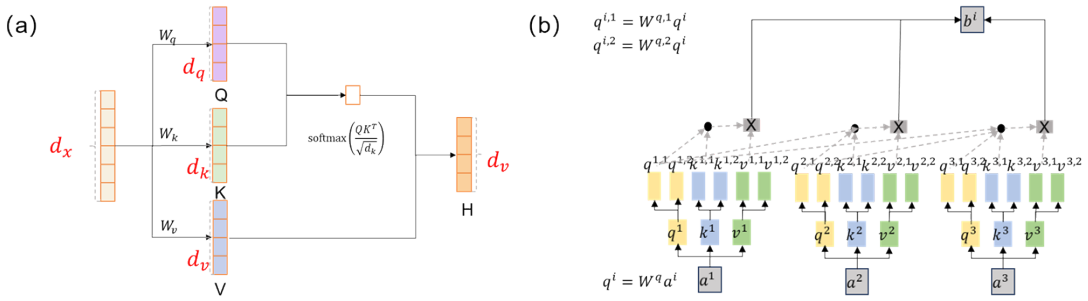

Fig. S1 Schematic Diagram of Attention. (a) Single-head attention. (b) Multi-Head Attention

Residual block: The residual module is a network architecture introduced by He et al.[1] in 2015 as a core component of deep residual networks (ResNet). Figure S3 illustrates the residual structure module. Its primary purpose is to address the common issues of gradient vanishing and gradient explosion during the training of deep neural networks, significantly enhancing both the trainability and performance of these networks. The key idea behind the residual module is to pass input information directly to the output by incorporating an identity mapping (constant mapping) between network layers. This design enables the network to optimize by learning the residuals, i.e., the differences between the input and output, thereby effectively mitigating the degradation phenomenon commonly observed in deep networks.

A typical residual module can be represented by the following equation:

$$y = F(x,\{W_i\}) + x \#(3)$$

where x is the input, $F(x,\{W_i\})$ represents the transformation obtained through several layers of the neural network, and the output y includes both the input x and the residual component.

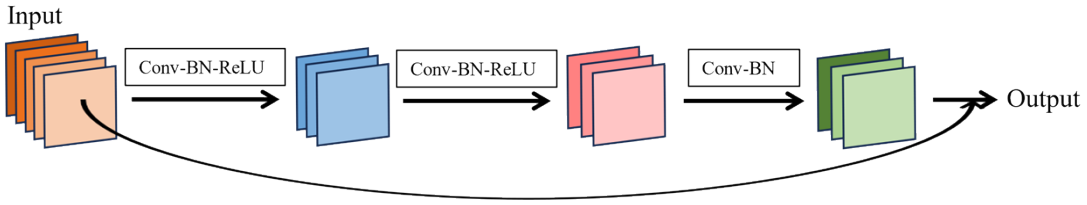

Fig. S2 Schematic diagram of residual block.

The implementation code of MHA-ResNet:

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset


class Bottlrneck(torch.nn.Module):
    def __init__(self,In_channel,Med_channel,Out_channel,downsample=False):
        super(Bottlrneck, self).__init__()
        self.stride = 1
        if downsample == True:
            self.stride = 2

        self.layer = torch.nn.Sequential(
            torch.nn.Conv1d(In_channel, Med_channel, 1, self.stride),
            torch.nn.BatchNorm1d(Med_channel),
            torch.nn.ReLU(),
            torch.nn.Conv1d(Med_channel, Med_channel, 3, padding=1),
            torch.nn.BatchNorm1d(Med_channel),
```

```
                torch.nn.ReLU(),
                torch.nn.Conv1d(Med_channel, Out_channel, 1),
                torch.nn.BatchNorm1d(Out_channel),
                torch.nn.ReLU(),
            )

            if In_channel != Out_channel:
                self.res_layer=torch.nn.Conv1d(In_channel,Out_channel,1,self.stride)
            else:
                self.res_layer = None

        def forward(self,x):
            if self.res_layer is not None:
                residual = self.res_layer(x)
            else:
                residual = x
            return self.layer(x)+residual


class ResNet(torch.nn.Module):
        def __init__(self,in_channels=1,classes=4):
            super(ResNet, self).__init__()
            self.features = torch.nn.Sequential(
                torch.nn.Conv1d(in_channels,64,kernel_size=7,stride=2,padding=3),
                torch.nn.MaxPool1d(3,2,1),

                Bottlrneck(64,64,256,False),
                Bottlrneck(256,64,256,False),
                Bottlrneck(256,64,256,False),
```

```
                torch.nn.AdaptiveAvgPool1d(1)
            )
            self.classifer = torch.nn.Sequential(
                torch.nn.Linear(512,1)
            )

        def forward(self,x):
            x = x.reshape(-1, 1, x_train.shape[1])
            x = self.features(x)
            x = x.view(-1,512)
            return x
class MultiheadAttentionBlock(nn.Module):
        def __init__(self, embed_dim, num_heads, ff_dim, dropout=0.1):
            super(MultiheadAttentionBlock, self).__init__()
            self.att = nn.MultiheadAttention(embed_dim, num_heads)
            self.ffn = nn.Sequential(
                nn.Linear(embed_dim, ff_dim),
                nn.ReLU(),
                nn.Linear(ff_dim, embed_dim)
            )
            self.layernorm1 = nn.LayerNorm(embed_dim)
            self.layernorm2 = nn.LayerNorm(embed_dim)
            self.dropout1 = nn.Dropout(dropout)
            self.dropout2 = nn.Dropout(dropout)

        def forward(self, x):
            att_output, _ = self.att(x, x, x)
            out1 = self.layernorm1(x + self.dropout1(att_output))
```

```
            ffn_output = self.ffn(out1)
            out2 = self.layernorm2(out1 + self.dropout2(ffn_output))
            return out2

class MHA_ResNet (nn.Module):
    def __init__(self, in_channels=1, embed_dim=512, num_heads=8, ff_dim=2048, num_transformer_blocks=1, dropout=0.1):
            super(MHA_ResNet, self).__init__()
            self.resnet = ResNet(in_channels)
            self.embed_dim = embed_dim

            self.transformer_blocks = nn.ModuleList(
                [MultiheadAttentionBlock(embed_dim,num_heads,ff_dim,dropout) for _ in range(num_transformer_blocks)]
            )

            self.classifier = nn.Sequential(
                nn.Linear(embed_dim, 1)
            )

    def forward(self, x):
            x = x.reshape(-1, 1, x.shape[1])
            x = self.resnet.features(x)
            x = x.view(-1, self.embed_dim)
            x = x.unsqueeze(0)

            for transformer in self.transformer_blocks:
                x = transformer(x)
```

```
        x = x.squeeze(0)
        x = self.classifier(x)
        return x
```

To ensure full reproducibility, both the dataset and the complete source code have been made publicly available via a GitHub repository: https://github.com/Aig00015/Deep-learning-prediction-of-electrode-voltage-for-metal-ions-batteries.

Table S2 Comparison of prediction performance across different models

Model	MAE	RMSE	R^2

MHA-ResNet	0.04	0.07	0.997
MLP model1[3]	0.41	0.63	0.902
MLP model2[3]	0.20	0.32	0.978
DNN[4]	0.35	-	-
SVM	0.64	0.67	0.653
KRR	0.72	0.81	0.583
LSTM	0.10	0.16	0.945

Table S3 Performance of different training sample sizes on the validation set

Maximum number of training samples in each voltage interval	MAE	RMSE	R^2
-	0.04	0.07	0.997
200	0.10	0.15	0.989
150	0.11	0.20	0.980
100	0.12	0.24	0.973
50	0.21	0.28	0.963

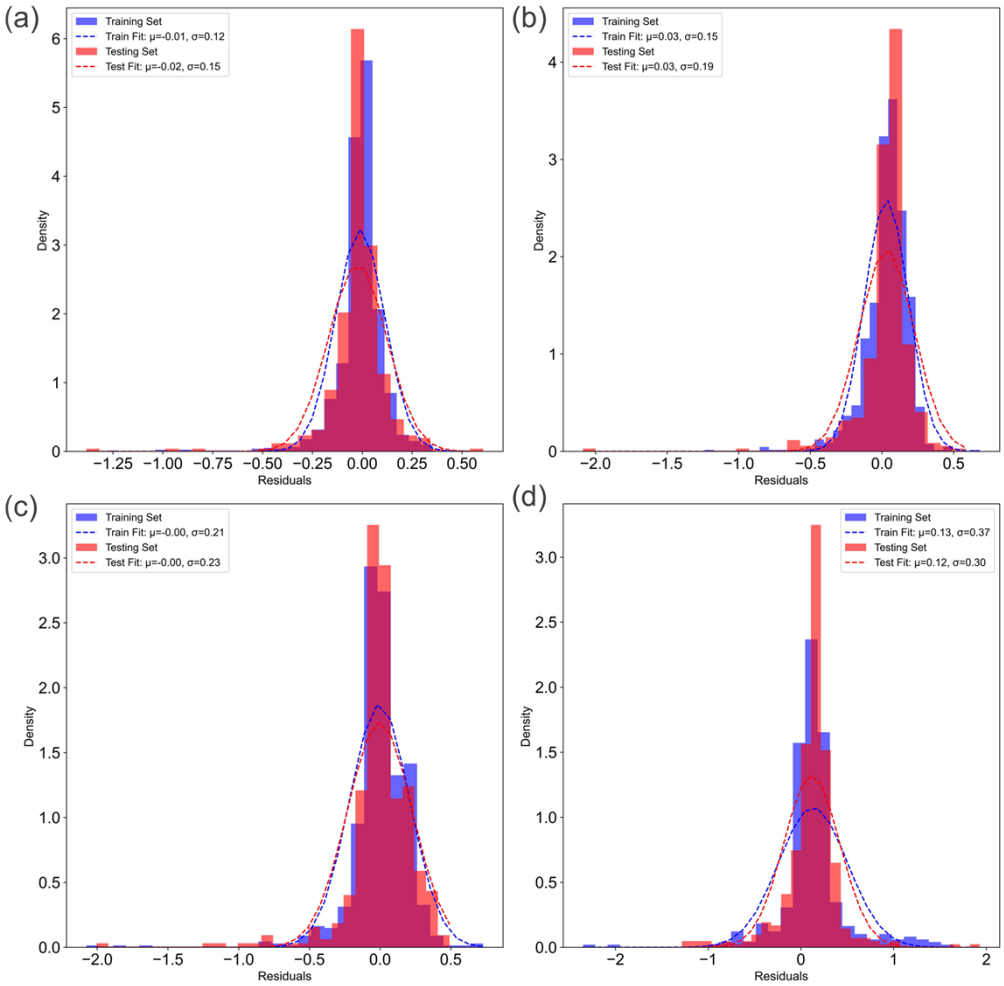

Fig. S3 Residual distributions from progressively reducing the training dataset size. (a) 200 samples per voltage interval. (b) 150 samples per voltage interval. (c) 100 samples per voltage interval. (d) 50 samples per voltage interval.

Table S4 Prediction performance of different metal-ion battery voltages

Metal ions	MAE	RMSE	R^2
Li	0.13	0.15	0.985
Na	0.19	0.24	0.942
K	0.22	0.26	0.940
Ca	0.18	0.29	0.950
Zn	0.06	0.08	0.994
Mg	0.08	0.13	0.983
Y	0.13	0.14	0.967

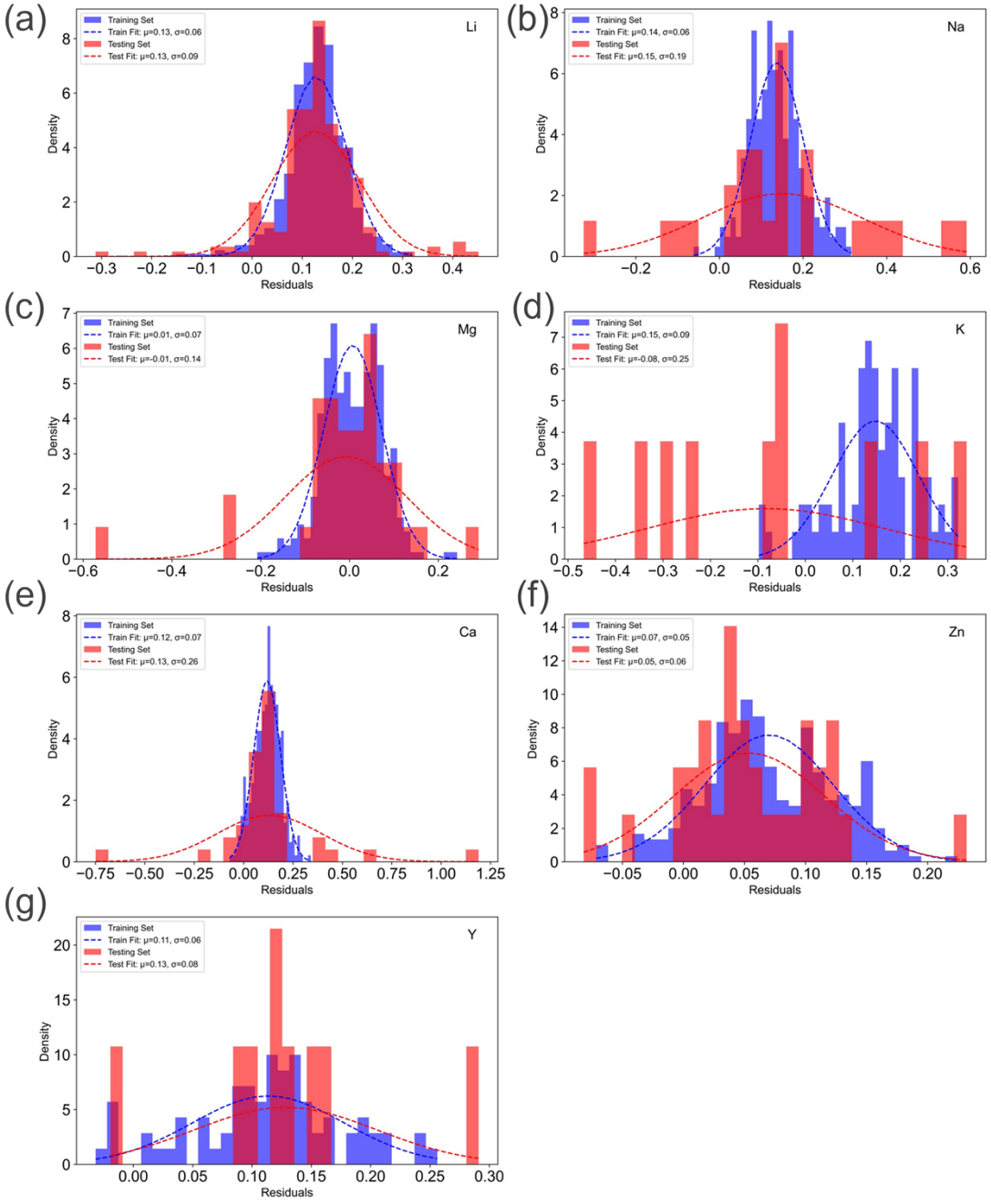

Fig. S4 Residual distributions for electrode material voltage.

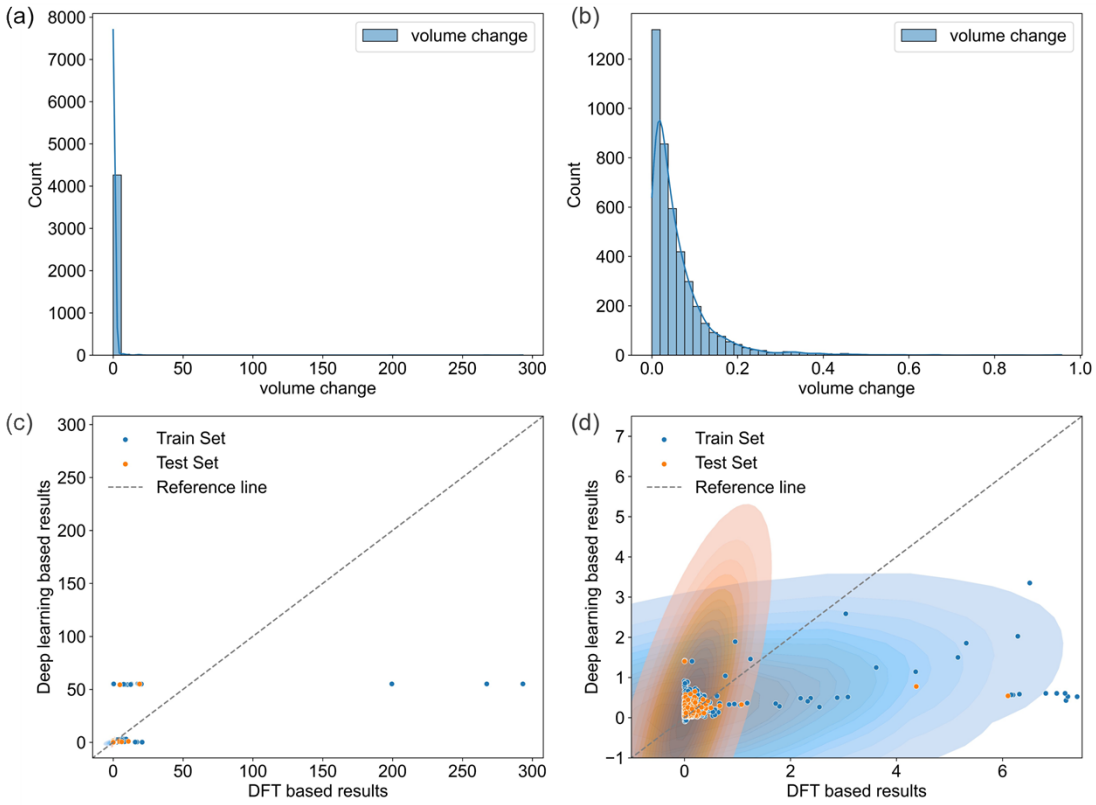

Fig. S5 (a) Distribution of percentage volume change across the full dataset. (b) Zoomed-in view of volume change values in the range of 0–1. (c) Parity plot comparing predicted and actual volume change values on the test set. (d) Enlarged view of the parity plot for the range 0–7.

We conducted additional investigation using the same feature encoding pipeline and MHA-ResNet architecture. However, the model exhibited lower performance on this task compared to voltage prediction. This outcome is primarily due to the following reasons: (i) Imbalanced target distribution: the percentage volume change data is highly skewed, with the majority of samples concentrated near zero and a small number of extreme outliers. This makes it difficult for the model to learn a balanced mapping across the entire range. The overall distribution is shown in Fig. S5a, and a zoomed-in view of the [0–1] region is provided in Fig. S5b to highlight the dense low-change region. (ii) Lack of structural sensitivity in input features: volume change generally arises from local atomic rearrangements, lattice distortions and interlayer spacing variations, which are not adequately captured by the composition-based and basic crystallographic features used in our model. (iii) Intrinsic noise in the target property: the volume change values from the Materials Project are derived from DFT-based structural relaxation and may include numerical uncertainty due to convergence issues or data inconsistency, potentially introducing noise that limits learnability. (iv) High physical complexity of the target: the mechanisms underlying volume change are highly nonlinear and depend on subtle structural effects, such as phase transitions, that are difficult to predict without explicitly modeling detailed atomic configurations. While the overall predictive accuracy was limited, we present the results in Fig. S5(c), which shows the parity plot between predicted and actual

volume change values. A zoomed-in view of the [0–7] range is shown in Fig. S5(d) to better visualize the model's behavior in the low-volume-change region. Although these results were not included in the main text due to their limited performance, they offer valuable insight into the limitations of composition-based modeling for structure-sensitive targets.

In future work, we plan to explore more expressive structural representations and techniques for handling imbalanced regression targets, such as target transformation and outlier filtering.

References

1 K. He, X. Zhang, S. Ren, J. Sun, presented in part at IEEE Conf. Comput. Vis. Pattern Recognit. Las Vegas, NV, June 2016.

2 A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, I. Polosukhin, presented in part at NeurIPS, Long Beach, CA, December 2017.

3 B. Ma, L. Zhang, W. Wang, H. Yu, X. Yang, S. Chen, H. Wang, X. Liu, *Green Energy Environ.*, 2022, **9**, 877-889.

4 A. Moses, R. P. Joshi, B. Ozdemir, N. Kumar, J. Eickholt, V. Barone, *ACS Appl. Mater. Interfaces*, 2021, **13**, 53355-53362.