Neural Metamorphosis –Supplementary Material–

Xingyi Yang and Xinchao Wang*

National University of Singapore
xyang@u.nus.edu, xinchao@nus.edu.sg

This supplementary material delves into "Neural Metamorphosis", starting with the its pipeline and pseudo-code in Section 1. Section 2 presents a proof of the orthogonal property of total variation. Section 3 establishes the INR as a generalized form of current continuous function representation. In Section 4, we extend our experiments to network depth morphing. Section 5 compares our weight permutation strategy with existing methods. Ablation studies in Section 6 examine the impact of model architecture, block-based INR, and EMA. Section 7 contrasts `NeuMeta` with similar works, supported by visual analyses in Section 8. Later we present further experimental details and the source code.

1 Pipeline for Neural Metamorphosis

In this section, we describe the pipeline for the Neural Metamorphosis, which is composed of three main stages: (1) Weight Permutation (2) INR Training (3) Weight Sampling. The pseudo-code is outlined in Algorithm 1, 2 and 3.

- **Weight Permutation.** This step involves modifying and smoothing a trained neural network's weights $\mathbf{W}$ by applying an optimal permutation matrix P^* on each clique graph. The aim is to generate a new, smoother set of weights, $\mathbf{W}^{(\text{smooth})}$, which are more readily learnable by the INR.
- **INR Training.** Leveraging the smoothed weights and the dataset, this step develops an Implicit Neural Network $F(\cdot;\theta)$. The primary objective here is to iteratively update the network's weights to optimize the INR. This optimization aims to minimize the overall loss on the dataset, thereby refining the performance of the network.
- **Weight Sampling.** The final stage is centered on extracting weights for the target network architecture $\mathbf{i}$ from INR. This process includes collecting K samples and averaging their weights to create a customized weight matrix $\mathbf{W}$. This matrix is specifically designed to suit the chosen architecture, ensuring that the target network is optimally configured for its intended tasks.

2 Axis Alignment of Total Variation: A Proof

Definitions and Notations:

* Corresponding author.

Algorithm 1 Neural Metamorphosis – Weight Permutation

Input: Trained neural network $f(\cdot; \mathbf{W})$ with dependency graph $G = (V, E)$. The weight of i-th layer is denoted as $\mathbf{W}_i$.
Output: The permuted and smoothed weight $\mathbf{W}^{(\text{smooth})}$

1: **for** $C = (V_C, E_C)$ **such that** ($C \subset G$ and C is a clique) **do**
2: Solve for optimal permutation matrix P^*

$$P^* = \underset{P}{\operatorname{argmin}} \sum_{e_{ij} \in E_C} \left(TV_{\text{out}}(P\mathbf{W}_i) + TV_{\text{in}}(\mathbf{W}_j P^{-1}) \right)$$

3: **for** e_{ij} **such that** $e_{ij} \in E_C$ **do**
4: Permute the weights according to the $P*$

$$\mathbf{W}_i^{(\text{smooth})} \leftarrow P\mathbf{W}_i; \mathbf{W}_j^{(\text{smooth})} \leftarrow \mathbf{W}_i P^{-1}$$

5: **end for**
6: **end for**
7: **return** $\mathbf{W}^{(\text{smooth})}$.

– Let $\mathbf{W}$ be an $m \times n$ matrix.
– $TV_{\text{in}}(\mathbf{W})$ denotes the sum of the absolute differences between consecutive elements within each row of $\mathbf{W}$.

$$TV_{\text{in}}(\mathbf{W}) = \sum_{i=1}^{m} \sum_{j=1}^{n-1} |w_{i,j+1} - w_{i,j}| \tag{1}$$

– $TV_{\text{out}}(\mathbf{W})$ denotes the sum of the absolute differences between consecutive rows (i.e., row-wise TV). For an $m \times n$ matrix, it's defined as:

$$TV_{\text{out}}(\mathbf{W}) = \sum_{i=1}^{m-1} \sum_{j=1}^{n} |w_{i+1,j} - w_{i,j}| \tag{2}$$

With these notations, the total variation of $\mathbf{W}$ can be expressed as:

$$TV(\mathbf{W}) = TV_{\text{in}}(\mathbf{W}) + TV_{\text{out}}(\mathbf{W})$$

Proof: When considering a permutation matrix P, we observe its effects on matrix $\mathbf{W}$ in terms of column and row permutations.

Case 1: Permutation of columns ($P\mathbf{W}$)

Column permutation, executed by P, rearranges the columns of $\mathbf{W}$ but does not affect the relative differences within each row. Consequently, the total variation within rows, TV_{in}, remains constant:

$$TV_{\text{in}}(P\mathbf{W}) = TV_{\text{in}}(\mathbf{W}) \quad \text{(row TV are unchanged)} \tag{3}$$

Case 2: Permutation of rows ($\mathbf{W}P$)

Algorithm 2 Neural Metamorphosis – INR Training

Input: A trained neural network $f(\cdot; \mathbf{W}^{(\text{smooth})})$, a training set $D_{tr} = \{\mathbf{x}_i, y_i\}_{i=1}^{M}$ and predefined configuration pool $\{\mathbf{i}_k\}_{k=1}^{K}$, total train iteration T.
Output: Implicit neural network $F(\cdot; \theta)$

for t **to** T **do**
  Sample a config from pool $\mathbf{i} \in \{\mathbf{i}_k\}_{k=1}^{K}$.
  for j such that $\mathbf{j} \in \mathcal{J}_{\mathbf{i}}$ **do**
    $(\mathbf{i}', \mathbf{j}') \leftarrow (\mathbf{i}, \mathbf{j}) + \boldsymbol{\epsilon}$ where $\boldsymbol{\epsilon} \sim \mathrm{U}(-\mathbf{a}, \mathbf{a})$.
    $\mathbf{v}' \leftarrow \left[\frac{l'}{L'}, \frac{c'_{\text{in}}}{C'_{\text{in}}}, \frac{c\prime_{\text{out}}}{C'_{\text{out}}}, \frac{L'}{N}, \frac{C'_{\text{in}}}{N}, \frac{C'_{\text{out}}}{N}\right]$.
    Generate weight $w_{(\mathbf{i},\mathbf{j})} \leftarrow F(\gamma_{\text{PE}}(\mathbf{v}'); \theta)$.
  end for
  Sample a batch of train data (X, Y).
  Inference with the generated weights $\hat{Y} = f(X; \mathbf{W})$.
  Calculate total loss $\mathcal{L} = \mathcal{L}_{\text{task}} + \lambda_1 \mathcal{L}_{\text{recon}} + \lambda_2 \mathcal{L}_{\text{reg}}$.
  Calculate gradient$\nabla_\theta \mathcal{L} = \frac{\partial \mathcal{L}_{\text{task}}}{\partial W} \frac{\partial W}{\partial \theta} + \lambda_1 \frac{\partial \mathcal{L}_{\text{recon}}}{\partial \theta} + \lambda_2 \frac{\partial \mathcal{L}_{\text{reg}}}{\partial \theta}$.
  Update the INR's weight $\theta \leftarrow \theta - \alpha \nabla_\theta \mathcal{L}$
end for
return $F(\cdot; \theta)$.

Algorithm 3 Neural Metamorphosis – Weight Sampling

Input: Trained INR $F(\cdot; \theta)$, number of sampling K and desired architecture $\mathbf{i}$.
Output: Weight $\mathbf{W}$ corresponds to $\mathbf{i}$.

Initialize all elements in $\mathbf{W}$ to be 0.
for $k = 1$ **to** K **do**
  $(\mathbf{i}', \mathbf{j}') \leftarrow (\mathbf{i}, \mathbf{j}) + \boldsymbol{\epsilon}$ where $\boldsymbol{\epsilon} \sim \mathrm{U}(-\mathbf{a}, \mathbf{a})$.
  $\mathbf{v}' \leftarrow \left[\frac{l'}{L'}, \frac{c'_{\text{in}}}{C'_{\text{in}}}, \frac{c\prime_{\text{out}}}{C'_{\text{out}}}, \frac{L'}{N}, \frac{C'_{\text{in}}}{N}, \frac{C'_{\text{out}}}{N}\right]$.
  Generate and average $w_{(\mathbf{i},\mathbf{j})} \leftarrow w_{(\mathbf{i},\mathbf{j})} + \frac{1}{K} F(\gamma_{\text{PE}}(\mathbf{v}'); \theta)$.
end for
return $\mathbf{W}$.

Conversely, row permutation modifies the order of rows in $\mathbf{W}$ without altering the internal composition of each row. Thus, the total variation between rows, TV_{out}, stays unchanged:

$$TV_{\text{out}}(\mathbf{W}P) = TV_{\text{out}}(\mathbf{W}) \quad \text{(column TV are unchanged)} \tag{4}$$

From the aforementioned cases, we can clearly infer that a permutation applied to one dimension of the matrix does not influence the total variation in its orthogonal dimension.

This proof underscores the *axis-alignment* characteristic of Total Variation (TV). However, this property may not be extended to other smoothness measurements. For example, in cases where smoothness is evaluated with respect to diagonal neighbors, defined by the expression $\sum_{i=1}^{m-1} \sum_{j=1}^{n-1} |w_{i+1,j+1} - w_{i,j}|$, the measurement is not axis-aligned anymore.

3 INR as Generalized Continuous Function

Currently, two methods exist for representing neural network weights as continuous functions: the piece-wise linear function approach [2] and as using kernel method [3]. This section aims to establish that the Implicit Neural Representation (INR) offers a more generalized approach compared to these traditional methods.

We show the proof in 1D signal, which can be easily extended to higher-dimensional scenarios.

3.1 INR Generalizes to Piece-wise Linear Method

In this section, we demonstrate that an INR can be viewed as a generalized form of Piece-wise Linear Function (PLF). We establish this by showing that a Multi-Layer Perceptron (MLP) can approximate any PLF, particularly using a single-layer neural network with ReLU activation.

Definition of Piece-wise Linear Function

A Piece-wise Linear Function is defined as a function consisting of several linear segments. Formally, for a PLF $f(x)$ over an interval $[a, b]$, it is expressed as:

$$f(x) = \begin{cases} a_1 x + b_1 & \text{if } x \in [x_0, x_1], \\ a_2 x + b_2 & \text{if } x \in (x_1, x_2], \\ \quad \vdots & \\ a_n x + b_n & \text{if } x \in (x_{n-1}, x_n], \end{cases}$$

where $x_0 = a$ and $x_n = b$, and a_i, b_i are constants.

Construction Using a Single-Layer Neural Network:

Consider a neural network with a single layer having n neurons, each with a ReLU activation function. The output of the i-th neuron for an input x is

$$y_i = \text{ReLU}(w_i x + b_i) = \max(0, w_i x + b_i),$$

where w_i and b_i are the weight and bias of the i-th neuron, respectively.

To approximate the PLF, we align $w_i = a_i$ and choose b_i so that the line $y = w_i x + b_i$ aligns with the i-th segment of the PLF. As such, The final output of the neural network is exactly the same as the original PLF

$$f_{NN}(x) = \sum_{i=1}^{n} y_i = \sum_{i=1}^{n} \text{ReLU}(a_i x + b_i).$$

Proof of Approximation:

For any given PLF, we can find a set of weights and biases $\{w_i, b_i\}$ for a single-layer neural network with ReLU activation such that its output $f_{NN}(x)$ approximates the PLF $f(x)$ over the interval $[a, b]$. This leverages the property of ReLU to emulate piece-wise linear segments.

3.2 INR Generalizes to Kernel Method

To demonstrate that an INR generalizes to any kernel method, we show that kernel method can be implemented using a neural network with the kernel function as the activation function.

Definition of Kernel Method

Kernel methods are a class of algorithms used in machine learning for pattern analysis. A kernel function takes two inputs and outputs a similarity measure. In mathematical terms, a kernel $K(x, y)$ is a function that for all $x, y \in \mathcal{X}$, where $\mathcal{X}$ is the input space, satisfies certain properties (e.g., symmetry, positive definiteness).

For example, In [3], they use a bicubic kernel $K(x, y)$ that computes the weighted average of points around a given position y from x. The kernel typically employs cubic polynomials and is defined as:

$$K(x,y) = \begin{cases} (1.5|x-y|^3 - 2.5|x-y|^2 + 1) & \text{for } |x-y| < 1, \\ (-0.5|x-y|^3 + 2.5|x-y|^2 - 4|x-y| + 2) & \text{for } 1 \leq |x-y| < 2, \\ 0 & \text{otherwise.} \end{cases}$$

Construction Using Neural Network:

In an INR framework, a neural network can utilize the kernel function as its last layer's activation function. For an input x and a set of data points $\{y_i\}_{i=1}^N$, the network's output is formulated as:

$$f_{NN}(x) = \sum_{i=1}^{N} w_i K(MLP(x), MLP(y_i)) + b,$$

where w_i are the weights, b is the bias, and N is the number of data points.

Proof of Approximation:

The Universal Approximation Theorem implies that if the MLP is used to replace the learned sampling points as described in [3], the two formulations essentially become identical. By adjusting the weights $\{w_i\}$ and bias b, the network can replicate traditional kernel methods.

In the preceding two subsections, we demonstrated that our INR serves as a generalized parameterization method for traditional continuous weight neural networks. This provides a strong theoretical foundation that reinforces the superior empirical results observed with our approach.

4 Extension: Depth Morphing

In this extension study, we explore the concept of *morphing network depth* as opposed to network width done in the main paper. This advances our goal of morphing arbitrary network weights beyond just the channel number.

Experiment Setup. This experiment utilizes the MNIST dataset with a specially structured ResNet. This ResNet is constructed in two stages, each comprising a variable number of residual blocks, denoted as $L1$ for stage 1 and $L2$ for stage 2, respectively. Each residual block contains two 3×3 convolutional

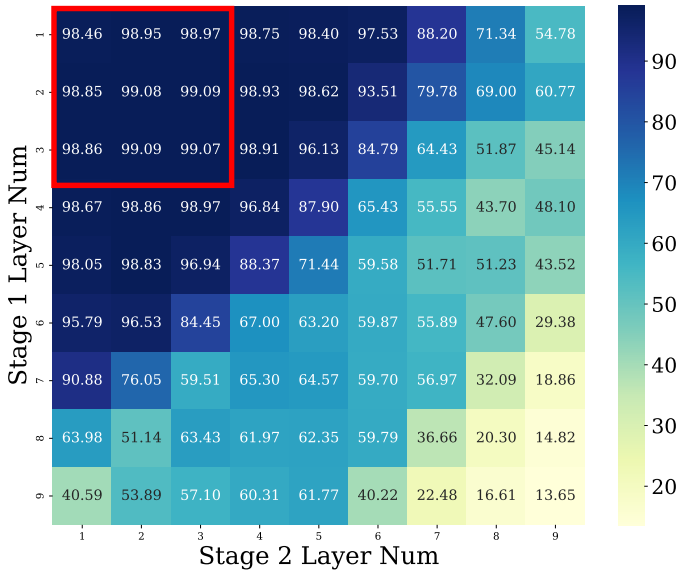

Fig. 1: Experiments on Depth Morphing. Configurations enclosed in the red rectangle represent those encountered during training. All other configurations displayed were not included in the training phase.

layers with ReLU activation. We initiate our process with a pretrained model where $L1 = 2$ and $L2 = 2$. We then train an INR by varying $L1$ and $L2$. During training, we sample $L1, L2 \in \{1, 2, 3\}$, but for testing, we evaluate performance across a wider range with $L1, L2 \in \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$. In this experiment, we employ a single INR to fit all layers.

Results. The performance are reported in Figure 1 as a heatmap. As expected, the models maintain a comparable performance, approximately 99% accuracy, across different depth configuration $L1, L2 \in \{1, 2, 3\}$. Interestingly, we observe that `NeuMeta` is capable of smoothly extrapolating to unseen architectural configurations. For example, with a ResNet configuration of $L1 = L2 = 7$, the accuracy remains to 56.97%, largely better than random guess. We believe this still showcases a positive signal that, `NeuMeta` presents some degree of **zero-shot generation** for unseen configuration. All together, it shows that our `NeuMeta` is not confined to width morphing alone, showing further exploration direction.

5 Weight Permutation Strategy

In the main paper, we have reformulated the within-network smoothness problem as a weight permutation problem, solved through a multi-objective Shortest Hamilton Path (mSHP) Problem. This section compares our method with the existing layer-wise Traveling Salesman Problem (TSP) permutation strategies.

Method	Accuracy	TV
Original (No Permute)	91.62	58141.80
Layer-TSP Greedy	91.67	50271.82
Layer-TSP 2-Opt	91.31	49132.78
mSHP (Ours)	**91.87**	**45576.02**

Table 1: Ablation results with different weight permutation strategies.

Baselines and Measurement. For comparison, we consider the original, unpermuted weights and three different permutation strategies. The first is the *Random* permutation, which shuffles the weights based on a randomly generated order. We also compare our results with two layer-wise TSP strategies: the *2-Opt* algorithm [3], and a *Greedy* solver approach [1]. Starting from the same

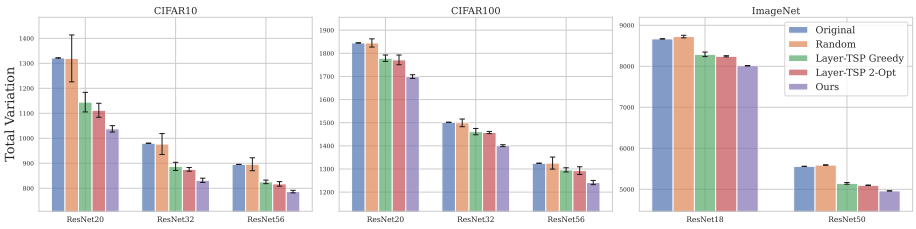

Fig. 2: The average total variation for model weights after different permutation strategies. We show the mean± over 5 runs.

trained network, we apply each permutation algorithm and measure the resulting total variation, averaged across all parameters. We repeat these experiments five times for each strategy and report the mean and standard deviation (mean ± std). Besides, we train the INR model on weights modified by these permutations using the CIFAR10 dataset, to assess the impact of weight permutation on final model performance.

Results. The results are presented in Figure 2. We evaluated our mSHP solution across eight different network architectures on three datasets. The results demonstrate that our mSHP approach consistently achieves the best performance in terms of TV score after permutation. These results are highlighted in Purple in the figure.

Furthermore, we investigated the performance of permuted weights on the CIFAR10 dataset, as detailed in Table 1. Our finding indicates that compared to existing permutation strategies, our approach yields a slight improvement in performance.

6 Additional Ablation Studies

6.1 Model Architecture

In the paper, we utilize a residual MLP with ReLU activation, as depicted in Figure 3. The input dimension, represented as INPUT_DIM, is computed as $(\text{NUM\_FREQ} \times 2 + 1) \times 6$. This formulation indicates that the input is expanded by Fourier features. In this section, our discussion mainly revolves around three critical parameters of the network: the number of layers (k), the number of frequencies, the hidden channel number (d), and the usage of residual connection.

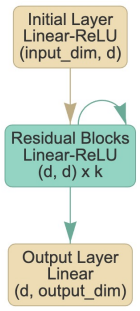

Fig. 3: Prototype of network architecture.

Experiment Setup. In our experimental analysis, we explored the impact of varying model architectures on accuracy. The primary variables altered in these experiments were the number of layers, the number of hidden units, and the number of frequencies within the model. Specifically, we tested models with 5, 8, and 12 layers, each with either 256 or 512 hidden units, and frequency counts of 16 and 32. Additionally, we conducted comparisons between models with and without residual connections to understand their impact on performance.

Results and Findings. As shown in Figure 2, the results provided insights into the optimal model architecture. The best performance was observed in models with 8 layers and 512 hidden units at 32 frequencies, achieving a peak accuracy of 91.87%. Models with 5 layers and 256 hidden units also performed well, particularly at 16 frequencies, reaching an accuracy of 91.76%. However, models with 12 layers and 256 hidden units experienced a decline in accuracy, regardless of frequency count. This indicates that a configuration of 8 layers, 512 hidden units, and 32 frequencies is most conducive to high accuracy in our model setup.

The results comparing performances with and without residual connections are presented in Table 3. It is observed that incorporating residual connections enhances performance by an increase of 1.69% in accuracy.

Num Layer	Num Hidden	Num Freq	Accuracy
5	256	16	91.76
5	256	32	91.23
8	256	16	91.68
8	256	32	91.27
8	512	16	91.84
8	512	32	**91.87**
12	256	16	91.27
12	256	32	91.03

Table 2: Ablation study on model architectures hyper-parameter.

Method	Accuracy
W/o residual	90.18
W residual	**91.87**

Table 3: Ablation study on residual connection.

6.2 Block-Based INR

In our efforts to enhance the capacity of the implicit function, we have implemented a block-based INR. This approach involves designating distinct MLP networks for predicting the different parameters (each `weight` or `bias`). The effectiveness of this method is assessed by comparing it against a conventional approach, where a singular INR is utilized to predict the weights for all layers.

Method	Accuracy
W/o block-based INR	91.74
W block-based INR	**91.87**

Table 4: Ablation study on block-based INR.

Results. The comparison is illustrated in Table 4. Our finding indicates that employing the block-based INR strategy leads to enhancement in performance. Specifically, there is an observed improvement of 0.13% in accuracy on the CIFAR10 dataset when using the block-based INR, highlighting its efficacy.

6.3 Exponential Moving Average (EMA)

In our study, we incorporate the Exponential Moving Average (EMA) technique during the training phase of the INR. This approach updates the weights of the INR using $\theta^t \leftarrow \gamma\theta^{t-1} + (1-\gamma)\theta^t$. We would like to evaluate the efficiency of the EMA technique and determining the optimal value for the γ parameter.

γ	Accuracy (%)
0	91.32
0.99	91.42
0.995	**91.87**
0.999	91.42

Table 5: Ablation study evaluating different γ values in EMA.

Results. The results of our ablation study, as shown in Table 5, reveal the impact of varying γ values on the accuracy of the model. It is evident that the optimal value of $\gamma = 0.995$ significantly enhances the model's performance.

7 Distinction From NeRN

While there are work that apply INR to fit neuron weight like Neural Representation for Neural Networks (NeRN) [1], our method, `NeuMeta`, stands distinct in several key aspects:

- **Memorize *vs.* Generalize.** NeRN is tailored to fit a *single* network, and lacks the ability to extrapolate to unseen architectures post-training. In contrast, `NeuMeta` is designed to adapt to the *entire manifold*. Once trained, it can generate weight for any network configuration on this manifold, without retraining.
- **Discrete *vs.* Continuous.** Unlike NeRN, which fits a discrete network with coordinate-wise inputs, `NeuMeta` embraces a continuous manifold approach. This allows weight values to be represent by an average over a small neighborhood, enhancing generalization to unseen architectural weights beyond the training scope.
- **Purpose.** NeRN focuses on model compression, storing the parameters of a full model within a smaller MLP to reduce the parameter count. `NeuMeta`, however, aims for resizeability and flexibility, enabling on-the-fly sampling of different network weights.

Given these factors, our `NeuMeta` represents a significant advancement beyond NeRN, moving past the confines of fitting weights for a single, discrete network.

8 Segmentation Visualization

Building upon our performance analysis of semantic segmentation in the main paper, we offer an in-depth qualitative comparison in Figure 4. This includes a comparison with the full-sized model (shown in column 3) and models trained using the Slimmable Neural Network approach (columns 4 and 6).

We make two major observations

- **Enhancement Over Slimmable NN.** Notably, our `NeuMeta` consistently outperforms the Slimmable NN baseline. In the 50% compression scenario, `NeuMeta` achieves more accurate mask predictions. Even in the challenging 75% compression case, where Slimmable NN struggles significantly, `NeuMeta` manages to produce reasonable segmentation outputs.
- **Comparison with Full-Sized Model**: Interestingly, `NeuMeta` not only competes with but also surpasses the full-sized model in several instances, such as in rows 3, 6, and 8. These observations might indicate the potential efficacy of `NeuMeta`, especially in terms of how its smoothed weight could

improve the model's generalization capabilities across various challenging contexts.

These findings underscore `NeuMeta`'s potential in providing more accurate semantic segmentation, particularly under large compress rate, demonstrating its value in enhancing model performance and efficiency.

Fig. 4: Semantic segmentation visualization on VOC2012 dataset. $*$ denotes that the model has not been trained with a 75% compression rate.

9 Experimental Setup

9.1 Selection of Pretrained Models

For ResNet models applied to CIFAR10 and CIFAR100 datasets, we utilize the models trained and available at [1]. In the case of ResNet on the ImageNet dataset,

[1] https://github.com/chenyaofo/pytorch-cifar-models

we employ the official pretrained models provided by PyTorch. For tasks involving MNIST, image generation, and VOC segmentation, we have trained the models ourselves and subsequently integrated them into our Neural Metamorphosis pipeline.

References

1. Ashkenazi, M., Rimon, Z., Vainshtein, R., Levi, S., Richardson, E., Mintz, P., Treister, E.: Nern: Learning neural representations for neural networks. In: The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023. OpenReview.net (2023), https://openreview.net/pdf?id=9gfir3fSy3J 7, 10
2. Le Roux, N., Bengio, Y.: Continuous neural networks. In: Artificial Intelligence and Statistics. pp. 404–411. PMLR (2007) 4
3. Solodskikh, K., Kurbanov, A., Aydarkhanov, R., Zhelavskaya, I., Parfenov, Y., Song, D., Lefkimmiatis, S.: Integral neural networks. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR). pp. 16113–16122 (June 2023) 4, 5, 7