How to Train Your Metamorphic Deep Neural Network

Thomas Sommariva, Simone Calderara, and Angelo Porrello

AImageLab, University of Modena and Reggio Emilia, Italy
{thomas.sommariva, simone.calderara, angelo.porrello}@unimore.it

Abstract. Neural Metamorphosis (NeuMeta) is a recent paradigm for generating neural networks of varying width and depth. Based on Implicit Neural Representation (INR), NeuMeta learns a continuous weight manifold, enabling the direct generation of compressed models, including those with configurations not seen during training. While promising, the original formulation of NeuMeta proves effective only for the final layers of the undelying model, limiting its broader applicability. In this work, we propose a training algorithm that extends the capabilities of NeuMeta to enable full-network metamorphosis with minimal accuracy degradation. Our approach follows a structured recipe comprising blockwise incremental training, INR initialization, and strategies for replacing batch normalization. The resulting metamorphic networks maintain competitive accuracy across a wide range of compression ratios, offering a scalable solution for adaptable and efficient deployment of deep models. The code is available at: https://github.com/TSommariva/HTTY_NeuMeta

Keywords: Weight Manifold · Morphable Neural Network

1 Introduction and Related Works

The increasing demand for deploying deep neural networks (DNNs) on edge devices or under real-time constraints underscores the challenge of dynamically adapting models to heterogeneous computational environments while maintaining high predictive accuracy. Addressing this issue is key to enabling broad deployment in real-world systems, including industrial systems and IoT devices, where balancing efficiency and performance is critical [1, 6, 24].

To this end, several methods have been proposed to improve efficiency and adaptability. **Network Pruning**, reduces model size by removing parameters deemed less important, based on various importance heuristics [11, 12, 15, 17, 22, 30]. Another popular approach is **Knowledge Distillation**, where a compact student model is trained to replicate the behavior of a larger teacher model, thus maintaining accuracy while reducing inference cost [4, 18, 27, 34–36]. More recent research has focused on **Flexible Models**, which are trained to operate under multiple configurations, enabling dynamic adaptation to varying deployment scenarios. However, these methods are inherently limited to configurations observed during training, restricting their flexibility. [5, 7, 14, 19, 32, 33].

More recently, **Neural Metamorphosis** (NeuMeta) [31] introduced a novel paradigm in which architectural hyperparameters – such as the number of layers and channels – can be adapted on the fly. Specifically, given a target architecture for deployment, the NeuMeta framework can dynamically generate the corresponding weights as a function of the selected hyperparameters. To do so, the authors of NeuMeta have proposed a **block-based** approach based on Implicit Neural Representation (INR) [28]. The overarching goal is to learn the **weight manifold** of a neural network, enabling the sampling of new networks of arbitrary sizes – even for configurations not seen during training.

Despite its flexibility, the approach proposed by the authors of NeuMeta has been applied only to a limited portion of the network. For context, in ResNet-56 – which contains approximately 27 residual blocks – the method is used solely on the last. This highlights the complexity of extending metamorphosis to multiple layers, as also reflected in our experimental results. We observed that optimization becomes increasingly unstable as more blocks are incorporated into the morphing process. We attribute this trend to several potential shortcomings of NeuMeta, including its initialization strategy and the absence of batch normalization in the models sampled via INR.

In this work, we propose an improved training algorithm that extends the capabilities of NeuMeta **enabling full-network metamorphosis** of ResNet architectures [16]. Our method adopts a gradual training strategy, applying neural metamorphosis to one residual block at a time. Building on this framework, we introduce several additional techniques to enhance training stability. For example, to accelerate convergence, each INR is initialized with the weights of the previously trained one. We also address the absence of batch normalization by incorporating learnable scaling coefficients, and further stabilize training across multiple architectures using gradient aggregation.

Experiments show that our approach significantly improves over the original NeuMeta framework, narrowing the accuracy gap with pre-trained models. We observe strong results on high-capacity networks, achieving near-baseline performance even under aggressive compression, an outcome of particular relevance from an efficiency-oriented deployment perspective. These findings establish the feasibility of full-network metamorphosis and lay the foundation for deploying dynamically adaptable DNNs across a wide range of real-world applications.

2 Background

In our work, we focus on ResNet56 [16] and adopt the following nomenclature:

- **Block**: A residual block comprising two 3×3 convolutional layers, ReLU activations, and a shortcut connection.
- **Layer**: A group of residual blocks that share the same number of output channels. Specifically, ResNet56 consists of three such layers of increasing width, with each layer that contains nine residual blocks.
- **Base Layers**: Refers explicitly to parametric transformations, such as a convolutional layer or a linear layer.

2.1 Neural Metamorphosis

NeuMeta [31] proposes a novel training framework that treats the set of weights of a neural network as points sampled from a continuous, high-dimensional weight manifold. NeuMeta aims to learn this manifold, enabling the direct generation of weights for neural networks with varying hyperparameter configurations (*e.g.*, number of layers or channels).

To achieve this goal, NeuMeta employs a methodology based on Implicit Neural Representation (**INR**) [26]. An INR-based model is implemented through a Multi-Layer Perceptron (MLP), which takes as input a set of coordinates identifying a network configuration (see Eq. (1)) and outputs the corresponding weights. In particular, each INR receives as input a coordinate pair $\mathbf{v} = (i, j)$, where $i = (L, C_{\text{in}}, C_{\text{out}})$ refers to a reference architecture at maximum capacity, and $j = (l, c_{\text{in}}, c_{\text{out}})$ corresponds to a sampled configuration with reduced capacity. In particular, L denotes the number of base layers, while l is the index of the base layer being predicted. The l^{th} base layer of the reference architecture has C_{in} input channels and C_{out} output channels, while c_{in} and c_{out} are input\output channels of the generated base layer. Coordinates are normalized to capture relative proportions:

$$\mathbf{v} = \left[\frac{l}{L}, \frac{c_{in}}{C_{in}}, \frac{c_{out}}{C_{out}}, \frac{L}{N}, \frac{C_{in}}{N}, \frac{C_{out}}{N} \right], \text{ where } N = \max_{l \in L}(C_{in}^{(l)}, C_{out}^{(l)}). \tag{1}$$

Coordinates are further enriched using Fourier positional embeddings to facilitate learning high-frequency details [29].

To sample the weights across all layers, NeuMeta employs a block-based approach [28], where each weight and each bias of every base layer is modeled by a separate INR. All INRs share a common architecture, including output dimensionality $\mathcal{D}$; however, their outputs are interpreted according to their target.

- **Convolutional kernels:** For kernels of size k, when $\mathcal{D} \geq k \times k$, the sampled kernel is obtained by extracting the central $k \times k$ window from the output.
- **Bias:** The first value of the output vector is used as the bias.
- **Linear layers:** Each projection weight is computed via a separate forward pass, with the final value given by the mean of the $\mathcal{D}$ predicted values.

Smoothness of the weight manifold. A primary challenge arises because INRs excel at learning low-frequency functions [25,29], whereas the weight manifold of neural networks exhibits high-frequency variations. To address this discrepancy, NeuMeta leverages a pre-trained base model – referred to as the **prior model** – as a reference for supervising the training of the INR models. Smoothness is enforced through two **smoothing strategies** applied to the weights of the prior model: one promotes *intra-network smoothness* (as illustrated in Fig. 1), while the other encourages *cross-network smoothness*. These strategies help the INR models better interpolate between neighboring configurations.

- **Intra-network smoothing.** Consider a weight matrix $W \in \mathbb{R}^{C_{out} \times C_{in}}$, its local smoothness can be quantified using total variation (TV) [31], defined

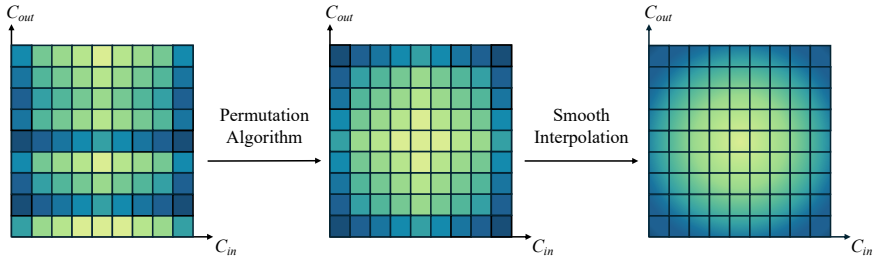

Fig. 1: 2D visualization of weight manifold smoothing via permutation alignment

as the variation along both channels $TV(W) = TV(W_{in}) + TV(W_{out})$. The authors of NeuMeta seeks a permutation matrix P that minimizes the total variation of the resulting permuted weight matrix. Notably, the behavior of the corresponding base layer remains unaffected by such a permutation, provided that the permutation is inverted on the layer's output.

– **Cross-network smoothing.** To ensure that small differences in input configurations do not significantly affect the predicted parameters, a small perturbation $\epsilon \sim \mathcal{U}(-0.5, 0.5)$ is added to each input coordinate. This encourages the model to learn a smoother mapping by operating over a continuous space. Since the introduction of stochasticity renders the forward process non-deterministic, the model is evaluated multiple times with independently sampled perturbations. The final weights are then obtained by averaging the resulting predictions.

Batch normalization layers (BN). During the metamorphic process, the BN layers of the prior model [20] are handled using a re-parameterization strategy [10], which absorbs their operations—and thus their parameters—into adjacent base layers. This design is motivated by the fact that BN parameters capture activation statistics (mean and variance) that are inherently tied to the network architecture. As a result, statistics computed for one configuration may not be valid under different compression ratios (see Eq. (2)). Re-parameterizing BN into neighboring base layers offers an architecture-agnostic alternative, ensuring compatibility across continuously changing network structures.

2.2 Training Procedure of NeuMeta

The training process of NeuMeta involves subsequent iterations of gradient-based optimization up to convergence. At each iteration, a configuration is sampled from a configuration pool, which is generated by applying varying compression ratios $\gamma \in [0.0, 0.5]$ to the original architecture. The rate γ is defined as:

$$\gamma = 1 - \frac{\text{sampled channel number}}{\text{full channel number}}. \tag{2}$$

For each sampled configuration, the resulting weights are optimized with a composite loss function integrating three terms: a **task-specific loss**, a **reconstruc-**

tion loss and a **regularization loss**. Each term is weighted by a coefficient:

$$\mathcal{L} = \lambda_1 \mathcal{L}_{\text{TASK}} + \lambda_2 \mathcal{L}_{\text{RECON}} + \lambda_3 \mathcal{L}_{\text{REG}}. \tag{3}$$

The **task-specific loss** $\mathcal{L}_{\text{TASK}}$ evaluates the performance of the network generated by the INR models on the downstream task: for classification, it corresponds to the standard cross-entropy loss $\mathcal{L}_{\text{TASK}} = -\sum_{k=1}^{K} y_k \log \hat{y}_k$. The **reconstruction term** $\mathcal{L}_{\text{RECON}} = \|\theta_{\text{PRIOR}} - \hat{\theta}\|_2^2$ encourages the predicted parameters $\hat{\theta}$ to stay close to the original, ideal weights θ_{PRIOR} of the smoothed pre-trained model. This term is optimized only when the sampled configuration matches the prior model, *i.e.*, when $\gamma = 0.0$. Finally, the **regularization component** $\mathcal{L}_{REG} = \|\hat{\theta}\|_2$ penalizes large weight magnitudes to mitigate overfitting [23].

3 Method

Empirically, we observe that NeuMeta struggles to learn the weight distribution across the entire network (see Fig. 2). We attribute this limitation mainly to two factors: the challenge of learning the full weight manifold across multiple layers simultaneously, and the absence of batch normalization layers in the sampled models. To address these issues and further improve training effectiveness, we introduce a multi-step algorithm. Our training process aims to progressively metamorphose the entire architecture while preserving the performance of the original network. Each component of this method is detailed in the following subsections, while the full architectural design of the metamorphic block and a complete pseudocode are provided in the supplementary material.

3.1 Incremental Training

As shown in our experiments (Tab. 2), NeuMeta demonstrates effective metamorphosis of single residual blocks with minimal accuracy degradation. However, NeuMeta struggles when extending this approach to more layers. Inspired by this finding, we adopt a *divide et impera* strategy, metamorphosing the network in a **block-wise, gradual** manner. Each stage of the training process focuses on transforming a single block into its metamorphic counterpart. During this phase, the previously learned INR models are jointly fine-tuned along with the current one. Moreover, the last block and the classification layer are not metamorphosed directly; instead, they are optimized across all sampled configurations to ensure consistent behavior of the last base layers under varying compression ratios.

This incremental approach allows each stage to focus on **localized** regions of the weight space, enabling more stable optimization and facilitating the learning of a continuous weight manifold across the architecture.

3.2 Substitution of Batch Normalization Layers

As discussed in Sec. 2.1, NeuMeta removes **Batch Normalization** layers from the prior network. While removing BN has minimal impact when only a few

blocks are transformed, applying it to larger portions of the network results in significant performance degradation—highlighting the critical role of batch normalization in training deep residual networks [2, 20].

To address this issue, we mitigate the absence of normalization layers by modifying the residual paths, drawing inspiration from [9]. These modified blocks preserve the standard residual structure but incorporate a **learnable coefficient** $\boldsymbol{\alpha}$, initialized to zero, which scales the residual branch before it is added to the skip connection. This is motivated by the observation that BN layers typically downscale residual activations during early stages of training. Consequently, the main contribution to the propagated signal is provided by the skip connections, causing residual blocks to approximate identity mappings. Hence, introducing α emulates the activations of BN-equipped networks, ensuring stable training without explicit normalization.

3.3 INR Pre-Initialization

While the original NeuMeta's framework initializes each INR model randomly, our incremental training paradigm enables a more effective strategy. Specifically, we initialize each newly added INR model with the final weights of its predecessor. This approach enables knowledge transfer across INR networks trained subsequently, leading to faster convergence, as each block can build upon the representations learned in earlier blocks.

3.4 Gradient Accumulation

The NeuMeta training procedure updates the weights of the INR based on gradients computed from a single sampled configuration. We argue that this strategy could bias the current optimization step toward the sampled configuration, limiting the generalization ability of the INR model. To mitigate this, we adopt gradient accumulation, aggregating losses over **multiple configurations** before performing a weight update. This strategy improves gradient quality and supports more effective joint learning across architectural variants.

Additionally, each mini-batch includes the uncompressed configuration ($\gamma = 0.0$), ensuring that the reconstruction term in the loss function (see Eq. (3)) contributes to every backward pass.

3.5 Disentangling Weights and Biases

As discussed in Sec. 2.1, the original block-based INR approach differentiates between weights and biases only after inference through selective extraction of the relevant output portion. We argue that this design introduces representational ambiguity, especially when the hypernetwork generates outputs that are mostly discarded, as happens in bias predictions.

To address this issue, we propose disentangling weights and biases at the architectural level by employing separate INRs for each. In this respect, every INR is designed with an output dimensionality tailored to its target: scalars for biases and full 3×3 matrices for convolutional kernels.

4 Experiments

4.1 Implementation Details

We use ResNet-56 [16] as the backbone architecture, trained on the CIFAR100 dataset [21], which provides sufficient complexity to validate the effectiveness of our proposed method. During training, input images are augmented via random cropping and horizontal flipping. Two variants of the ResNet-56 architecture are evaluated: the first configuration (Tab. 1 left side) is a lightweight model with 16, 32, and 64 channels in the first, second, and third layers, respectively; the second variant (Tab. 1 right side) features a higher-capacity design with 64, 128, and 256 channels in the corresponding layers. For computational efficiency, we apply the metamorphosis only to the final layer of the architecture (for a total of **7 blocks**). This represents an extension over NeuMeta, which modifies only **the last block**; however, our method can be easily extended to earlier layers with minimal accuracy degradation (see Figs. 2 and 3).

The base model of each INR network consists of an eight-layer MLP with ELU activation functions [8] and residual connections, each layer contains 512 neurons, and the positional embedding frequency is set to 32. Optimization is performed using AdamW [23], a warm-up phase of 20 epochs linearly increases the learning rate from 0 to 8×10^{-4}, after which it remains constant. Considering the coefficients employed in the loss function Eq. (3), we set $\lambda_1 = 10^2$, $\lambda_2 = 1$, and $\lambda_3 = 10^{-3}$. Each stage of our incremental training strategy lasts for 50 epochs (Sec. 3.1); in the non-incremental settings, the model is trained for the same total number of iterations as the baseline method. Gradient Accumulation (Sec. 3.4) is performed on a mini-batch of 4 configurations. We assign to each block a unique α, shared across all compression ratios.

Experiments are primarily conducted on a single Nvidia RTX 6000 GPU with 24 GB of VRAM. For the higher-capacity ResNet configuration, due to increased memory requirements for storing all model configurations at that scale, we utilize an Nvidia A40 GPU with 48 GB of VRAM.

4.2 Baselines

We compare our approach against the following methods:

- **Neural Metamorphosis** [31]: Serves as our primary baseline. Since our framework builds upon NeuMeta, this comparison is equivalent to an ablation of all the new components introduced in this work.
- **Individually Trained ResNet:** Separate ResNet models are trained from scratch at each compression ratio – denoted as *Individual* in our experiments.
- **Pruning:** We evaluate structural pruning techniques, including random pruning, Hessian-based pruning, Taylor-based pruning, and magnitude-based pruning (l_1 and l_2 norms). Pruned networks are obtained using the Dependency Graph method [11], which constructs the inter-layer dependency graph to enable group-wise pruning of structurally coupled parameters.

Table 1: Ablation study across different γ. $^{\dagger}\gamma = 0.75$ wasn't applied in training.

	ResNet56					ResNet56 with higher capacity				
	γ : 0.0 Acc	γ : 0.25 Acc	γ : 0.5 Acc	γ : $0.75^{\dagger}$ Acc	Train Cost GPUh	γ : 0.0 Acc	γ : 0.25 Acc	γ : 0.5 Acc	γ : $0.75^{\dagger}$ Acc	Train Cost GPUh
Individual	72.63	71.50	71.78	70.59	3.5	74.86	74.73	74.67	74.82	5.1
NeuMeta [31]	49.03	48.69	48.65	45.38	19.8	69.07	69.01	68.90	68.84	57.1
Ours	**66.13**	**66.00**	**66.10**	**65.36**	16.2	**71.69**	**71.66**	**71.60**	**71.67**	46.8
– Incremental training	57.16	57.24	56.92	53.22	19.6	70.61	70.72	70.59	70.70	56.7
– α scaling	62.56	62.68	62.56	61.35	**16.1**	71.02	70.93	71.03	71.01	**46.6**
– Grad Accumulation	64.54	64.71	64.48	63.78	17.7	71.15	71.16	71.14	71.15	53.2
– Initialization	64.81	64.76	64.67	63.37	16.2	71.04	71.04	71.09	71.01	46.8
– DisentaglmentW&B	65.45	65.26	65.24	64.74	16.3	71.09	71.11	71.10	71.03	46.9

Table 2: Ablation study on the last block of the third layer of ResNet56. $^{\dagger}\gamma = 0.75$ wasn't applied in training.

	One metamorphic block				
	γ : 0.0 Acc	γ : 0.25 Acc	γ : 0.5 Acc	γ : $0.75^{\dagger}$ Acc	Train Cost GPUh
Individual	72.63	71.94	72.31	71.99	3.5
NeuMeta [31]	71.84	71.82	71.79	71.76	1.8
Ours	**72.24**	**72.23**	**72.24**	**72.23**	**1.4**
– α scaling	72.17	72.14	72.14	72.17	1.4
– Grad Accumulation	72.22	72.21	72.21	72.20	1.5
– DisentaglmentW&B	72.21	72.19	72.20	72.19	1.4

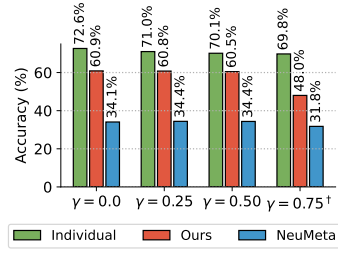

Fig. 2: Full-network metamorphosis across different γ.

4.3 Results and Ablative Studies

Table 1 presents the top-1 accuracy and training time for our experiments applying metamorphosis to the final layer of ResNet-56. Additionally, Tab. 2 reports results for the setting where metamorphosis is restricted to the last block, consistent with the original setup proposed in [31]. Notably, our results indicate that the original NeuMeta approach struggles in extended scenarios as evidenced by its performance drop (see Tab. 1 and Fig. 2) – highlighting the challenges of applying neural metamorphosis across multiple blocks. This effect is further illustrated in Fig. 3, which depicts the progressive accuracy degradation as the number of metamorphic blocks increases. With respect to the training times reported in Tables 1 and 2, each line corresponds to the training time of the hypernetwork, which generates the weights for all γ. For the line "Individual", instead, the reported time reflects the cumulative training time all ResNet models, each trained independently at a specific value of γ.

Although a performance gap remains with respect to individually trained models (see *Individual*, first row of Tabs. 1 and 2), our training algorithm significantly outperforms NeuMeta [31]. Specifically, our approach **improves overall accuracy**, **reduces training time**, and **enhances generalization** to unseen configurations (*i.e.*, $\gamma = 0.75^{\dagger}$).

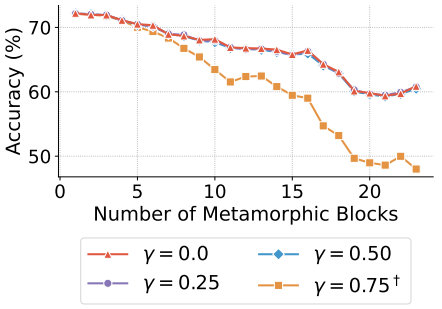

Fig. 3: Accuracy across varying numbers of metamorphic blocks for different γ.

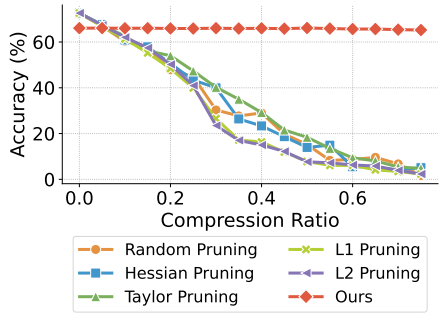

Fig. 4: Accuracy comparison with structural pruning techniques.

The experiments on the right side of Tab. 1 demonstrate that our method scales effectively with network width, achieving improved performance in higher-dimensional settings. On the wider, **higher-capacity** ResNet, our approach consistently delivers strong results, often approaching the performance of individually trained models. This suggests that optimizing larger networks is intrinsically easier – a trend also observed in NeuMeta, which narrows the performance gap compared to individually trained baselines. Nevertheless, our method still outperforms NeuMeta, even in this more favorable setting, reaching results close to that of the prior model. From an efficiency perspective, this is particularly advantageous, as compressing larger models yields significant resource savings.

To further evaluate the feasibility of **full-network metamorphosis**, Fig. 2 reports the accuracy of generated models when metamorphosis is applied to all residual blocks of ResNet-56, excluding the first block of each layer (those with projection shortcuts). These results confirm that full-network metamorphosis is achievable with our approach, resulting in only minor accuracy degradation.

We also compare our method with several structural **pruning strategies**. As shown in Fig. 4, across comparable compression settings, our approach consistently outperforms standard pruning techniques in terms of top-1 accuracy.

Impact of each component. Furthermore, both Tabs. 1 and 2 present a fine-grained ablation study quantifying the contribution of each component in our framework. The **incremental training** strategy (Sec. 3.1) has the most significant impact, improving accuracy by 12% and narrowing the gap between $\gamma = 0.0$ and the unseen $\gamma = 0.75^{\dagger}$ from 4% to under 1%. It also reduces training time by about 20%, as fewer parameters are generated during early stages. These results underscore the difficulty of modeling the full weight distribution and highlight the effectiveness of a progressive approach focused on localized subspaces.

Moreover, replacing BN layers with $\boldsymbol{\alpha}$ **scaling** (Sec. 3.2) improves accuracy by 3.5%, reducing the gap between $\gamma = 0.0$ and $\gamma = 0.75^{\dagger}$ by 0.5%. **Gradient accumulation** across multiple architecture configurations (Sec. 3.4) contributes a 1.5% accuracy gain and shortens training time by nearly 10% (due to the fewer

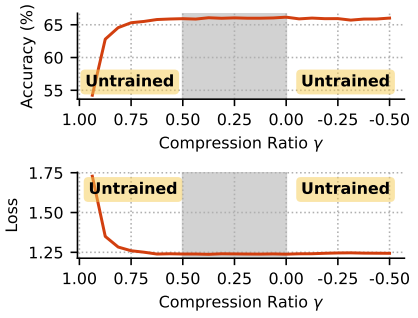

Fig. 5: Accuracy and loss at different γ

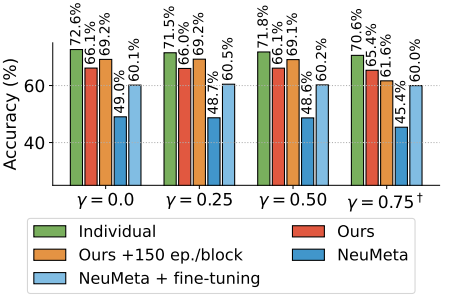

Fig. 6: Accuracy comparison across different γ for alternative training strategies.

performed optimization steps), indicating that simultaneous optimization across compression ratios promotes better generalization.

Considering our proposed **INR initialization** strategy (Sec. 3.3), it yields a 1.3% accuracy improvement on seen γ, and a 2% improvement on $\gamma = 0.75^{\dagger}$. This finding suggests that sequential weight transfer facilitates convergence to more meaningful features improving generalization.

Finally, **disentangling** the weights and biases (Sec. 3.5) yields a modest yet meaningful 0.6% accuracy boost, showing that dedicating separate modules to biases and kernel matrices avoids interference and improves learning dynamics.

4.4 Generalization and Fine-Tuning Analysis

In Fig. 5, we analyze performance trends across varying configurations. Notably, our approach maintains strong performance even for architectures with compression factors γ smaller than those seen during training – *i.e.*, more powerful models. This clearly demonstrates that our approach can generalize to unseen architectures. Conversely, for smaller models, performance begins to degrade as compression rates approach or fall below 0.80.

In Fig. 6, we further investigate the ability of INR models on untrained configurations. Specifically, we examine the effect of extending the training duration by a factor of three. Interestingly, while prolonged training improves accuracy on configurations seen during training, it hampers generalization to unseen ones. This reveals a trade-off between in-distribution performance and flexibility across untrained γ values. In contexts where adaptability is less critical, extended training can yield accuracy comparable to that of individually trained models.

Lastly, Fig. 6 also analyzes the effect of training the INR with the baseline method proposed in [31], augmented with the fine-tuning strategy applied to the last block of ResNet and the final classifier, as described in Sec. 3.1. The results demonstrate that the performance of the generated models can be significantly enhanced by simply adapting the final segment of the network to the preceding metamorphic blocks.

5 Conclusion

We focus on metamorphic neural networks – a class of models that can adjust their weights to fit desired network configurations. We observe that their training stability deteriorates when multiple blocks are transformed. To address this, we propose a multi-stage training algorithm that progressively enables the generation of fully metamorphic networks. While a performance gap remains between our generated models and individually trained counterparts, we show that our approach can be further improved by employing high-capacity prior networks or by extending the training duration. Our findings demonstrate the feasibility of full-network metamorphosis in ResNet architectures, and we hope this work paves the way for future research into extending metamorphic capabilities to a broader range of architectures, like Recurrent Neural Networks [3,13] and ViT.

References

1. Alajlan, N.N., Ibrahim, D.M.: Tinyml: Enabling of inference deep learning models on ultra-low-power iot edge devices for ai applications. Micromachines **13**(6), 851 (2022)
2. Bjorck, N., Gomes, C.P., Selman, B., Weinberger, K.Q.: Understanding batch normalization. NeurIPS **31** (2018)
3. Bolelli, F., Baraldi, L., Grana, C.: A Hierarchical Quasi-Recurrent approach to Video Captioning. In: 2018 IEEE International Conference on Image Processing, Applications and Systems (IPAS). pp. 162–167. IEEE (Dec 2018). https://doi.org/https://doi.org/10.1109/IPAS.2018.8708893
4. Buzzega, P., Boschini, M., Porrello, A., Abati, D., Calderara, S.: Dark experience for general continual learning: a strong, simple baseline. NeurIPS **33**, 15920–15930 (2020)
5. Cai, H., Gan, C., Wang, T., Zhang, Z., Han, S.: Once-for-all: Train one network and specialize it for efficient deployment. arXiv preprint arXiv:1908.09791 (2019)
6. Capogrosso, L., Cunico, F., Cheng, D.S., Fummi, F., Cristani, M.: A machine learning-oriented survey on tiny machine learning. IEEE Access **12**, 23406–23426 (2024)
7. Chavan, A., Shen, Z., Liu, Z., Liu, Z., Cheng, K.T., Xing, E.P.: Vision transformer slimming: Multi-dimension searching in continuous optimization space. In: ICCV. pp. 4931–4941 (2022)
8. Clevert, D.A., Unterthiner, T., Hochreiter, S.: Fast and accurate deep network learning by exponential linear units (elus). arXiv preprint arXiv:1511.07289 (2015)
9. De, S., Smith, S.: Batch normalization biases residual blocks towards the identity function in deep networks. NeurIPS **33**, 19964–19975 (2020)
10. Ding, X., Zhang, X., Ma, N., Han, J., Ding, G., Sun, J.: Repvgg: Making vgg-style convnets great again. In: CVPR (2021)
11. Fang, G., Ma, X., Song, M., Mi, M.B., Wang, X.: Depgraph: Towards any structural pruning. In: CVPR. pp. 16091–16101 (June 2023)
12. Frankle, J., Carbin, M.: The lottery ticket hypothesis: Finding sparse, trainable neural networks. arXiv preprint arXiv:1803.03635 (2018)
13. Gers, F.A., Schmidhuber, J., Cummins, F.: Learning to forget: Continual prediction with lstm. Neural computation **12**(10), 2451–2471 (2000)

14. Grimaldi, M., Mocerino, L., Cipolletta, A., Calimera, A.: Dynamic convnets on tiny devices via nested sparsity. IEEE Internet of Things Journal **10**, 5073–5082 (2022)
15. Hassibi, B., Stork, D.: Second order derivatives for network pruning: Optimal brain surgeon. NeurIPS **5** (1992)
16. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: CVPR (2016)
17. He, Y., Kang, G., Dong, X., Fu, Y., Yang, Y.: Soft filter pruning for accelerating deep convolutional neural networks. arXiv preprint arXiv:1808.06866 (2018)
18. Hinton, G., Vinyals, O., Dean, J.: Distilling the knowledge in a neural network. arXiv preprint arXiv:1503.02531 (2015)
19. Hou, L., Huang, Z., Shang, L., Jiang, X., Chen, X., Liu, Q.: Dynabert: Dynamic bert with adaptive width and depth. NeurIPS **33**, 9782–9793 (2020)
20. Ioffe, S., Szegedy, C.: Batch normalization: Accelerating deep network training by reducing internal covariate shift. In: ICML (2015)
21. Krizhevsky, A., Hinton, G.: Learning multiple layers of features from tiny images. Tech. Rep. 0, University of Toronto, Toronto, Ontario (2009)
22. Li, H., Kadav, A., Durdanovic, I., Samet, H., Graf, H.P.: Pruning filters for efficient convnets. arXiv preprint arXiv:1608.08710 (2016)
23. Loshchilov, I., Hutter, F.: Decoupled weight decay regularization. arXiv preprint arXiv:1711.05101 (2017)
24. Matsubara, Y., Callegaro, D., Baidya, S., Levorato, M., Singh, S.: Head network distillation: Splitting distilled deep neural networks for resource-constrained edge computing systems. IEEE Access **8**, 212177–212193 (2020)
25. Rahaman, N., Baratin, A., Arpit, D., Draxler, F., Lin, M., Hamprecht, F., Bengio, Y., Courville, A.: On the spectral bias of neural networks. In: ICML (2019)
26. Reiser, C., Peng, S., Liao, Y., Geiger, A.: Kilonerf: Speeding up neural radiance fields with thousands of tiny mlps. In: ICCV (2021)
27. Romero, A., Ballas, N., Kahou, S.E., Chassang, A., Gatta, C., Bengio, Y.: Fitnets: Hints for thin deep nets. arXiv preprint arXiv:1412.6550 (2014)
28. Tancik, M., Casser, V., Yan, X., Pradhan, S., Mildenhall, B., Srinivasan, P.P., Barron, J.T., Kretzschmar, H.: Block-nerf: Scalable large scene neural view synthesis. In: CVPR (2022)
29. Tancik, M., Srinivasan, P., Mildenhall, B., Fridovich-Keil, S., Raghavan, N., Singhal, U., Ramamoorthi, R., Barron, J., Ng, R.: Fourier features let networks learn high frequency functions in low dimensional domains. NeurIPS **33** (2020)
30. Wen, W., Wu, C., Wang, Y., Chen, Y., Li, H.: Learning structured sparsity in deep neural networks. NeurIPS **29** (2016)
31. Yang, X., Wang, X.: Neural metamorphosis. In: ECCV. Springer (2024)
32. Yu, J., Huang, T.S.: Universally slimmable networks and improved training techniques. In: iccv. pp. 1803–1811 (2019)
33. Yu, J., Yang, L., Xu, N., Yang, J., Huang, T.: Slimmable neural networks. arXiv preprint arXiv:1812.08928 (2018)
34. Yuan, L., Tay, F.E., Li, G., Wang, T., Feng, J.: Revisiting knowledge distillation via label smoothing regularization. In: ICCV. pp. 3903–3911 (2020)
35. Zhang, L., Song, J., Gao, A., Chen, J., Bao, C., Ma, K.: Be your own teacher: Improve the performance of convolutional neural networks via self distillation. In: ICCV (2019)
36. Zhang, Y., Xiang, T., Hospedales, T.M., Lu, H.: Deep mutual learning. In: CVPR (2018)

Supplementary Material

This supplementary material provides additional details for *How to Train Your Metamorphic Deep Neural Network*. Appendix A describes the design of the metamorphic block, while Appendix A presents the complete pseudocode of our method.

A Design of the Metamorphic Block

In this section, we present the complete design of our metamorphic block, which integrates both the original structure proposed in NeuMeta [31] and the α-scaling mechanism inspired by [9].

The metamorphic block consists of two 3×3 convolutional layers with a ReLU activation in between. The first convolution operates with C_{in} input channels and c_{out} output channels, where C_{in} corresponds to the input dimensionality of the standard residual block [16] in the same layer and c_{out} is computed as $c_{out} = (1 - \gamma) \times C_{out}$, effectively reducing the number of kernels proportionally to the compression ratio γ.

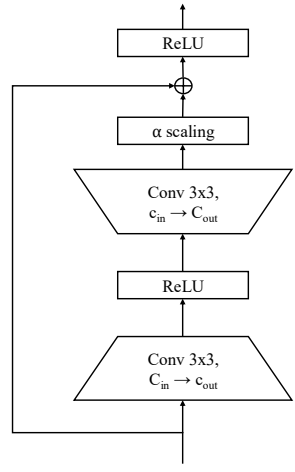

Fig. A1: Design of the metamorphic block

The second convolution takes c_{in} input channels and outputs C_{out} channels, restoring the original dimensionality. Consequently, the number of parameters in the second convolution is also reduced by a factor of γ, as it operates over a compressed intermediate feature space. The output of the second convolution is scaled by the learnable coefficient α and added to the shortcut connection.

This design reduces the total number of parameters in the residual block by a factor of γ, through compression applied to both convolutional layers. Furthermore, the metamorphic block remains fully compatible with the standard residual structure, enabling the integration of standard and metamorphic blocks within the same network. This flexibility supports both partial metamorphosis and the incremental strategy introduced in our training algorithm.

Training and Inference Pseudocode

Algorithm 1 Neural Metamorphosis – INR Training

Input: Smoothed ResNet: $f(\cdot;\theta_{\text{PRIOR}})$, Training set: $D_{tr} = \{\mathbf{x}_i, y_i\}_{i=1}^{M}$, number of metamorphic blocks: $\#blocks$, accumulation steps: $\#acc$, INR lr: η, θ_{shared} lr: η_w
Output: Implicit neural network $F(\cdot;W)$ and set of shared weights θ_{shared}

for b **in** $\#blocks$ **do**
  $C_{out} \leftarrow$ # output channels of the b-th block of the prior model
  Define the configuration pool $\mathbf{I} \leftarrow \{\frac{C_{out}}{2}, \frac{C_{out}}{2}+1, \ldots, C_{out}\}$
  Define the set of shared weights $\theta_{\text{shared}} \leftarrow \{\boldsymbol{\alpha},\ \text{classifier}\}$
  if $b \neq 0$ **then**
    Initialize $F(\cdot;W_b)$ with $F(\cdot;W_{b-1})$
  for idx, (X, Y) **in** $enumerate(D_{tr})$ **do**
    Sample a configuration $\mathbf{i} \in \mathbf{I}$
    for j such that j $\in \mathcal{J}_{\mathbf{i}}$ **do** ▷ $\mathcal{J}_{\mathbf{i}}$:= space of coordinates for **i**
      $\mathbf{v} \leftarrow \left[\frac{l}{L}, \frac{c_{\text{in}}}{C_{\text{in}}}, \frac{c_{\text{out}}}{C_{\text{out}}}, \frac{L}{N}, \frac{C_{\text{in}}}{N}, \frac{C_{\text{out}}}{N}\right] + \boldsymbol{\epsilon}$, where $\boldsymbol{\epsilon} \sim \text{U}(-0.5, 0.5)$
      Generate weight : $\theta_{\mathbf{i},\mathbf{j}} \leftarrow F(\mathbf{v};W)$
    Inference with the generated weights: $\hat{Y} \leftarrow f(X;\theta_{\mathcal{J}_{\mathbf{i}}})$
    Compute total loss : $\mathcal{L} \leftarrow \lambda_1\mathcal{L}_{\text{TASK}} + \lambda_2\mathcal{L}_{\text{RECON}} + \lambda_3\mathcal{L}_{\text{REG}}$
    Update θ_i : $\theta_i \leftarrow \theta_i - \eta_w\nabla_{\theta_i}\mathcal{L}$
    Accumulate the scaled the loss : $\mathcal{L}_{\text{AVG}} \leftarrow \mathcal{L}_{\text{AVG}} + \frac{\mathcal{L}}{s}$
    if idx % $\#acc = 0$ **then**
      Compute INR gradients: $\nabla_W\mathcal{L}_{AVG} \leftarrow \frac{\partial\mathcal{L}_{\text{AVG}}}{\partial\theta_i}\frac{\partial\theta_i}{\partial W}$
      Update the INR weights: $W \leftarrow W - \eta\nabla_W\mathcal{L}_{\text{AVG}}$

Algorithm 2 Neural Metamorphosis – Weight Sampling

Input: Trained INR $F(\cdot;W)$, Number of sampling iteration K, architecture configuration **i**, set of shared weights θ_{shared}
Output: Weights $\theta_{\mathbf{i}}$ corresponding to configuration **i**

Initialize $\theta_{\mathbf{i}}$ to 0: $\theta_{\mathbf{i}} \leftarrow \mathbf{0}$
for k **in** K **do**
  for j such that j $\in \mathcal{J}_{\mathbf{i}}$ **do** ▷ $\mathcal{J}_{\mathbf{i}}$:= space of coordinates for **i**
    $\mathbf{v} \leftarrow \left[\frac{l}{L}, \frac{c_{\text{in}}}{C_{\text{in}}}, \frac{c_{\text{out}}}{C_{\text{out}}}, \frac{L}{N}, \frac{C_{\text{in}}}{N}, \frac{C_{\text{out}}}{N}\right] + \boldsymbol{\epsilon}$, where $\boldsymbol{\epsilon} \sim \text{U}(-0.5, 0.5)$
    Generate and average weight : $\theta_{\mathbf{i},\mathbf{j}} \leftarrow \theta_{\mathbf{i},\mathbf{j}} + \frac{1}{K}F(\mathbf{v};W)$
Load the shared weights in the generated network: $\theta_{\mathbf{i}} \leftarrow \theta_{\text{shared}}$