# Neural Network Pruning with Scale Equivariant Graph Metanetworks

**Freek Byrman, Tobias Groot, Bart Kuipers, Daniel Uyterlinde**

*Supervised by Alejandro García Castellanos*

## Abstract

Metanetworks have been proposed to address the open challenge of automated pruning. To exploit neural network symmetries for this task, this work builds on the Scale Equivariant Graph Metanetwork (ScaleGMNs) [6] and introduces two novel pruning approaches. The first builds on traditional $L_1$ pruning, replacing the classification loss with a predicted accuracy term from a pretrained ScaleGMN. The second trains a ScaleGMN to directly generate pruning masks for a target network. We report that both methods are feasible and demonstrate their potential as symmetry-aware pruning strategies. The implementation of this work is available at `https://github.com/daniuyter/ScaleGMN-pruning`.

## 1 Introduction and background

The growing scale of Neural Networks (NNs) has intensified the need for efficient model optimization in tasks like architecture search, pruning, and knowledge transfer. A promising approach is the use of *metanetworks*, models designed to analyze and process other NNs. Formally, a metanetwork $\hat{f}$ can be represented as a mapping: $\hat{f} : \mathcal{G} \times \mathbf{\Theta} \rightarrow \mathcal{Y}$, where $\mathcal{G}$ denotes the space of possible architectures (computational graphs), and $\mathbf{\Theta}$ represents the space of learnable parameters. In this framework, we are particularly interested in two types of mappings: functionals and operators. Functionals are mappings where the output space $\mathcal{Y}$ is a subset of $\mathbb{R}^d$, capturing scalar or vector-valued outputs. Operators, on the other hand, map to spaces such as $\mathcal{G} \times \mathbf{\Theta}$ or $\mathbf{\Theta}$, reflecting transformations or modifications of the network's structure or parameters.

NNs are known to exhibit certain symmetries [5]. A fundamental example is permutation symmetry, which involves reordering nodes within a layer while adjusting their connections accordingly. Beyond permutation, scaling symmetries, arising from the non-zero scalar multiplication or division of weights and biases, also exist, though they have been less extensively explored. Such symmetries imply a significant gauge freedom in choosing sets of internal parameters, all of which lead to functionally equivalent models. Formally, a NN symmetry is induced by a set $\Psi$ of transformations $\psi : \mathcal{G} \times \mathbf{\Theta} \rightarrow \mathcal{G} \times \mathbf{\Theta}$, such that the function represented by the network $u_{G,\boldsymbol{\theta}} : \mathcal{X} \rightarrow \hat{\mathcal{X}}$ remains unchanged [6]: $u_{G,\boldsymbol{\theta}}(\mathbf{x}) = u_{\psi(G,\boldsymbol{\theta})}(\mathbf{x}), \forall \mathbf{x} \in \mathcal{X}, \forall (G, \boldsymbol{\theta}) \in \mathcal{G} \times \mathbf{\Theta}$.

The recognition of these symmetries has led to the development of symmetry-invariant metanetworks. Traditional metanetworks lack architectural guarantees for handling NN symmetries, though they might learn them through extensive training. In contrast, invariant metanetworks are designed to respect these symmetries by construction inherently. This design principle ensures consistency in functionally equivalent representations, enhances the efficiency of model learning by eliminating the need to rediscover known mathematical symmetries, and increases generalizability to unseen NN configurations. To effectively integrate this, a metanetwork acting as a functional must remain *invariant* under symmetry transformations, such that $\hat{f}(\psi(G, \boldsymbol{\theta})) = \hat{f}(G, \boldsymbol{\theta})$. In contrast, when acting as an operator, the metanetwork should be *equivariant*, ensuring that $\hat{f}(\psi(G, \boldsymbol{\theta})) = \psi(\hat{f}(G, \boldsymbol{\theta}))$.

Recent advances in metanetworks have led to improved performance on tasks such as classifying NNs and generating new weights, by leveraging permutation symmetries, either through architectural design [12] or Graph Meta Networks (GMNs) [10, 7]. Building on this, Kalogeropoulos et al. [6]

introduced the Scale Equivariant Graph Meta Network (ScaleGMN) to address the less explored scaling symmetries, extending the GMN framework with scale-equivariant message passing.

This work applies the ScaleGMN framework to NN pruning, which can broadly be divided into unstructured pruning (removing individual weights) and structured pruning (removing neurons or channels). Structured pruning poses significant automation challenges due to inter-layer dependencies and the exponential growth of possible channel configurations, making exhaustive search impractical. Consequently, traditional automated methods relying on non-differentiable reinforcement learning or evolutionary algorithms often require lengthy search times to converge [9].

To address these challenges, meta-networks were introduced as an alternative pruning strategy, initially by generating weights for candidate pruned structures to enable immediate evaluation [11], and later by directly pruning networks. Notably, the differentiable meta-pruning method by Li et al. [9] uses hypernetworks that generate weights from latent vectors encoding layer configurations. These meta-pruning approaches consistently outperform traditional methods [13], but their application within the ScaleGMN framework remains unexplored. Our work bridges this gap by proposing two pruning strategies using ScaleGMN: one as a functional, and another as an operator.

## 2 Methodology

**Scale Equivariant Graph Metanetworks**. The permutation symmetry of the scaleGMN is achieved by mapping the representing NNs to their inherently permutation-invariant graph representations. In this mapping, originally designed by Kofinas et al. [7], weights are mapped to edges, while biases are represented by vertices. In Appendix A, we detail this transformation and provide a visual representation of the mapping for one of the CNNs used in this work.

The novel contribution of Kalogeropoulos et al. [6] is recognizing that applying scale-equivariant initialization, message passing, and update functions to this graph representation results in an equivariant meta-network. They accomplish this by building an arsenal of expressive, learnable, equivariant, and invariant helper functions. At the heart of this method lies the canonical representation of objects. Defining $\tilde{o} = \text{canonical}(o)$, such that all objects equivalent $(\sim)$ under a scaling transformation map to the same canonical form. Formally, if $x \sim y \in \mathcal{X}$, then $\tilde{x} = \tilde{y}$. For scaling transformations, a natural canonical mapping is given by $\tilde{x} = \frac{x}{|x|}$. This transformation allows for any function $\rho(x)$ to be converted into a scale-invariant function by applying it to the canonicalized input $\rho(\tilde{x})$. In the ScaleGMN framework [6], an MLP is used for $\rho$, leveraging its universal approximation capability.

A candidate equivariant function would be a linear transformation $\Gamma : \mathcal{X} \to \mathbb{R}^d$, which inherently possess equivariance $(\Gamma(q\mathbf{x}) = q(\Gamma\mathbf{x}))$. Building on this set of transformations, a more expressive equivariant function can be constructed by recognizing that if a scale-equivariant function is multiplied (element-wise $\odot$) by a scale-invariant quantity, the result is itself scale-equivariant. Kalogeropoulos et al. [6] use this generalization to construct a learnable ScaleEquivariant function:

$$\text{ScaleInv}(\mathbf{X}) = \rho(\tilde{\mathbf{x}}_1, \dots, \tilde{\mathbf{x}}_n), \qquad \text{ScaleEq}(\mathbf{X}) = (\Gamma_1\mathbf{x}_1, \dots, \Gamma_n\mathbf{x}_n) \odot \text{ScaleInv}(\mathbf{X}). \quad (1)$$

Here, $\mathbf{X} = (\mathbf{x}_1, \dots, \mathbf{x}_n)$ is the input, $\Gamma_i$ are learnable linear transformations applied to each respective input component $\mathbf{x}_i$. $(\Gamma_1\mathbf{x}_1, \dots, \Gamma_n\mathbf{x}_n)$ is a concatenated vector which scales by $q$ if each $\mathbf{x}_i$ scales by $q$. This general learnable equivariant function (ScaleEq) can now be used to construct equivariant initialization (INIT), message (MSG), and update (UPD) functions.

A more comprehensive derivation, covering the identification of CNN symmetries to constructing the corresponding equivariant INIT, MSG, and UPD functions, is provided in Appendix B.

**Invariant Pruning**. Our first method, termed 'Invariant Pruning,' utilizes the ScaleGMN as a functional ($\hat{f} : \mathcal{G} \times \mathbf{\Theta} \to \mathbb{R}$). This approach consists of two stages. In the first stage, we train a ScaleGMN to predict generalization from weights using a subset of an augmented (see "Datasets and Baseline Models") version of the CIFAR-10 SmallCNNZoo [14]. In the second (pruning) stage, the resulting trained ScaleGMN serves as an evaluator, estimating the performance of the target network. During this pruning process, the parameters of this ScaleGMN are frozen, and optimization is performed over the parameters, $\boldsymbol{\theta}$, of the target network. The primary objective is to induce sparsity in $\boldsymbol{\theta}$ with the $L_1$ term, while maintaining the performance of the target network with the predicted accuracy term. To achieve this, we define the following loss function:

$$\mathcal{L}(\boldsymbol{\theta}) = \lambda \left\|\boldsymbol{\theta}\right\|_1 - \hat{f}(G, \boldsymbol{\theta}), \quad (2)$$

in which $\lambda$ is a hyperparameter to balance the effect of regularization and the predicted accuracy of the ScaleGMN. During this pruning process, we disable the gradients of any parameter that falls under a predefined threshold. We continue this process until a certain percentage of parameters fall below the threshold. After this, all parameters below the threshold are masked and set to zero. We then fine-tune the remaining nonzero parameters to compensate for the reduction in parameter mass. The architecture defining ScaleGMN $\hat{f}$ is identical to the one used in the CNN generalization prediction experiment described in Kalogeropoulos et al. [6]. The regularization coefficient $\lambda$ is chosen to ensure that the loss components have comparable magnitudes. All other hyperparameters vary by sparsity level and are selected through grid search based on validation accuracy after fine-tuning. The complete set of values is reported in Appendix C.1.

**Equivariant Pruning**. Our second pruning method employs a ScaleGMN, denoted $\hat{f}_{\boldsymbol{\phi}}$ parameterized by a set $\boldsymbol{\phi}$, as an operator ($\hat{f} : \mathcal{G} \times \boldsymbol{\Theta} \to \boldsymbol{\Theta}$) to prune a target network $u_{G,\boldsymbol{\theta}}(\mathbf{x})$. This operator receives the target network's architecture $G$ and its parameters $\boldsymbol{\theta}$ as input. It then directly outputs a set of structured pruned parameters $\boldsymbol{\theta}'$, according to the mapping $\hat{f}_{\boldsymbol{\phi}}(G, \boldsymbol{\theta}) = \boldsymbol{\theta}'$. The ScaleGMN $\hat{f}_{\boldsymbol{\phi}}$ is trained using the following optimization scheme:

$$\mathcal{L}(\boldsymbol{\phi}) = \lambda \cdot \left\| \hat{f}_{\boldsymbol{\phi}}(G, \boldsymbol{\theta}) \right\|_1 + \frac{1}{|\mathcal{B}|} \sum_{(\mathbf{x},y) \in \mathcal{B}} \mathcal{L}_{\text{CE}} \left( u_{G, \hat{f}_{\boldsymbol{\phi}}(G,\boldsymbol{\theta})}(\mathbf{x}), y \right), \tag{3}$$

where the first term represents the $L_1$ regularization on the pruned parameters, which ensures that the ScaleGMN produces sparse weights. The second term corresponds to the cross-entropy (CE) loss evaluated over the batch $\mathcal{B}$, applied to the output of the sparse network. This term ensures that the pruned network maintains its classification performance. Note that while the CE loss is typically optimized with respect to the network parameters $\boldsymbol{\theta}$, in this context, we are optimizing the hypernetwork parameters $\boldsymbol{\phi}$. Leveraging the chain rule we find:$\nabla_{\boldsymbol{\phi}} \mathcal{L}_{\text{CE}} = \nabla_{\boldsymbol{\theta}'} \mathcal{L}_{\text{CE}} \cdot \nabla_{\boldsymbol{\phi}} \hat{f}_{\boldsymbol{\phi}}(G, \boldsymbol{\theta})$. After training, all parameters below a specified threshold are masked to achieve the desired sparsity level, and the remaining nonzero parameters are fine-tuned to recover performance. Detailed hyperparameter settings are provided in Appendix C.2.

**Reproducibility Verification**. To validate our implementation of ScaleGMN, we replicate key experiments from the original paper [6], using the publicly available codebase[1]. This isolates pruning as the primary variable in our analysis. Specifically, we reproduce: 1) predicting CNN generalization from weights, and 2) INR editing, corresponding to the model's invariant and equivariant settings, respectively. These experiments follow the methodologies outlined in Kalogeropoulos et al. [6] and serve to verify the correctness of our setup.

**Datasets and Baseline Models**. For invariant pruning, we train the ScaleGMN using the *Small CNN Zoo* dataset [14], which provides weights and accuracy metrics for 270k CNNs. We focus on a subset of 30k small CNNs trained on grayscale CIFAR-10 [8]. To ensure that the network distribution in this dataset aligns with our objective of predicting the accuracy of pruned networks, we augment the data (CNN weights) by pruning a subset of models. Specifically, we are using magnitude-based pruning at sparsities from 0–50%, saving checkpoints after 0, 5, and 10 epochs of finetuning. We use the highest-performing small CNN from the Zoo as the pruning candidate for both invariant and equivariant pruning. Additionally, we evaluate equivariant pruning on a larger model using a pretrained VGG19 (20M+ parameters) trained on the RGB CIFAR-10 dataset [8]. For all experiments, we use a split of 45,000 training, 5,000 validation, and 10,000 test samples.

To evaluate the performance of the invariant and equivariant pruning methods, we compare them against three established baselines: Magnitude-based pruning [3, 4], Iterative Magnitude Pruning (IMP) [1] and $L_1$ pruning. All baseline method hyperparameters are reported in Appendix C.3.

## 3 Results

**Reproduction**. Reproducing the findings of Kalogeropoulos et al. [6] was relatively straightforward, largely due to their well-documented code. In the prediction generalization experiment, we achieved a Kendall-$\tau$ (a measure of ordinal correlation) of 0.9325, which is comparable to the results in the original work. For the INR editing task, our Mean Squared Error (MSE) of $2.55 \times 10^{-2}$ closely matches the $2.56 \times 10^{-2}$ reported by Kalogeropoulos et al. [6].

**Small CNN.** The left block of Table 1 presents pruning results for the Small CNN. We find that our methods generally outperform the simpler magnitude-based and IMP baselines. Invariant pruning

[1]https://github.com/jkalogero/scalegmn

achieves the highest accuracy across the 60–80% sparsity range, while the $L_1$ method performs best at 90% sparsity.

**VGG19**. The right block of Table 1 shows pruning results for the deeper VGG19 model. The Equivariant method performs comparably to the magnitude-based baseline. Performance stays consistent across methods until the extreme 99% sparsity level, where the $L_1$ and IMP-based methods perform best. This suggests that overparameterized networks can tolerate the removal of a large number of parameters.

Table 1: Performance comparison of ScaleGMN-based Invariant and Equivariant Pruning methods on two different architectures, CIFAR10-GS. Accuracy (%) is reported at different sparsity percentages (percentage of weights removed). Results are compared against Magnitude, IMP and $L_1$ baselines.

| **Method** | **Small CNN** (4,970 parameters) | | | | | **VGG19** (20M+ parameters) | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | **0%** | **60%** | **70%** | **80%** | **90%** | **0%** | **90%** | **95%** | **97%** | **99%** |
| Magnitude-based | 57.21 | 52.27 | 41.06 | 27.00 | 15.88 | 92.04 | 91.37 | 91.52 | 89.43 | 74.50 |
| IMP | 57.21 | 54.87 | 46.82 | 33.85 | 13.33 | 92.04 | 91.08 | 91.32 | 90.71 | 88.06 |
| $L_1$ | 57.21 | 57.30 | 57.02 | 55.84 | **47.31** | 92.04 | 89.89 | 89.91 | 89.89 | 66.84 |
| ScaleGMN-Inv | 57.21 | **60.31** | **59.66** | **56.29** | 46.27 | – | – | – | – | – |
| ScaleGMN-Eq | 57.21 | 53.54 | 51.15 | 50.75 | 46.09 | 92.04 | 90.72 | 90.59 | 89.87 | 74.40 |

## 4 Discussion

**Invariant Pruning**. While the ScaleGMN model effectively predicted performance on the Small CNN Zoo dataset, its use as a deterministic optimization function resulted in unrealistic outputs. The evaluator often predicted accuracies close to 90%, far exceeding the test accuracy of 57.21% corresponding to the best CNN in the small CNN Zoo. We suspect this arises because many parameter configurations explored during optimization lie outside the distribution represented in the training set, an issue the optimizer may exploit. Nonetheless, despite the overly optimistic predictions, the actual fine-tuned performance remained strong. This suggests that the evaluator still captures useful information, guiding the optimization toward promising regions of the weight space. Future research should focus on improving this evaluator component, and potentially extend weight-space/accuracy datasets (like SmallCNNZoo) to modern architectures.

**Equivariant Pruning**. The equivariant pruning approach performs reasonably well on the small CNN zoo dataset but does not outperform the simple magnitude baseline on the VGG19 architecture. Investigating the precise reasons for this discrepancy, such as extensive tuning of the ScaleGMN architecture, was beyond the primary focus of this study, which aimed to establish the feasibility of the approach. Although reparameterizing through a gradient space with favorable symmetrical inductive biases is theoretically appealing, practical challenges, such as architectural design and the high memory requirements of large GNNs, currently limit its effectiveness. These challenges currently favor simpler, well-established methods like IMP or magnitude-based pruning.

## 5 Conclusion

This work introduces Invariant Pruning and Equivariant Pruning, two novel methods based on the Scale Equivariant Graph Metanetwork (ScaleGMN) framework [6], which aim to address the challenge of automated pruning. Invariant Pruning leverages a ScaleGMN as a learned predictor of classification loss within an $L_1$ pruning scheme. While the complex loss landscape and high-dimensional feature space can cause the predictor to exploit out-of-distribution regions and yield overly optimistic accuracy estimates, the gradient directions it provides remain informative. Consequently, the method effectively guides optimization towards promising regions of the weight space. Equivariant pruning leverages a ScaleGMN as an operator to directly map a CNN to a pruned version. As part of this work, we provide an implementation that enables mapping operator-level outputs back to operational CNNs in a differentiable manner. Additionally, we contribute a dataset of approximately 4,000 pruned networks. The codebase is publicly available online[2].

These novel pruning methods serve as proof of concept, showing initial performance comparable to the standard $L_1$ pruning baseline for both methods. The results validate the potential of symmetry-aware network pruning.

[2] `https://github.com/daniuyter/ScaleGMN-pruning`

## References

[1] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *International Conference on Learning Representations (ICLR)*, 2019.

[2] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep learning*, volume 1. MIT press Cambridge, 2016.

[3] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[4] Stephen Hanson and Lorien Pratt. Comparing biases for minimal network construction with back-propagation. *Advances in neural information processing systems*, 1, 1988.

[5] Robert Hecht-Nielsen. On the algebraic structure of feedforward network weight spaces. 1990. URL `https://api.semanticscholar.org/CorpusID:115619723`.

[6] Ioannis Kalogeropoulos, Giorgos Bouritsas, and Yannis Panagakis. Scale Equivariant Graph Metanetworks, 2024. URL `https://arxiv.org/abs/2406.10685`.

[7] Miltiadis Kofinas, Boris Knyazev, Yan Zhang, Yunlu Chen, Gertjan J. Burghouts, Efstratios Gavves, Cees G. M. Snoek, and David W. Zhang. Graph neural networks for learning equivariant representations of neural networks, 2024. URL `https://arxiv.org/abs/2403.12143`.

[8] Alex Krizhevsky. Learning Multiple Layers of Features from Tiny Images. Technical report, Computer Science Department, University of Toronto, 2009. URL `https://api.semanticscholar.org/CorpusID:18268744`.

[9] Yawei Li, Shuhang Gu, Kai Zhang, Luc Van Gool, and Radu Timofte. Dhp: Differentiable meta pruning via hypernetworks, 2020. URL `https://arxiv.org/abs/2003.13683`.

[10] Derek Lim, Haggai Maron, Marc T. Law, Jonathan Lorraine, and James Lucas. Graph Metanetworks for Processing Diverse Neural Architectures, 2023. URL `https://arxiv.org/abs/2312.04501`.

[11] Zechun Liu, Haoyuan Mu, Xiangyu Zhang, Zichao Guo, Xin Yang, Kwang-Ting Cheng, and Jian Sun. Metapruning: Meta learning for automatic neural network channel pruning. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 3296–3305, 2019.

[12] Aviv Navon, Aviv Shamsian, Idan Achituve, Ethan Fetaya, Gal Chechik, and Haggai Maron. Equivariant architectures for learning in deep weight spaces, 2023. URL `https://arxiv.org/abs/2301.12780`.

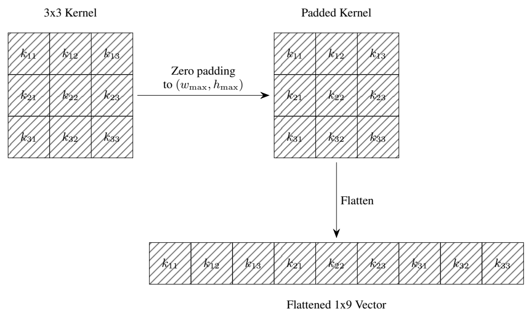
[13] D. Patilkulkarni, S. Shetty, P. Kulkarni, S. Hanchinamani, S. Kulkarni, and U. Kulkarni. A systematic overview of meta-pruning strategies in deep learning. In S. M. Thampi, J. Hu, A. K. Das, J. Mathew, and S. Tripathi, editors, *Applied Soft Computing and Communication Networks*, volume 966 of *Lecture Notes in Networks and Systems*. Springer, Singapore, 2024. ISBN 978-981-97-2003-3. doi: 10.1007/978-981-97-2004-0_40. URL `https://doi.org/10.1007/978-981-97-2004-0_40`.

[14] Thomas Unterthiner, Daniel Keysers, Sylvain Gelly, Olivier Bousquet, and Ilya Tolstikhin. Predicting neural network accuracy from weights, 2021. URL `https://arxiv.org/abs/2002.11448`.

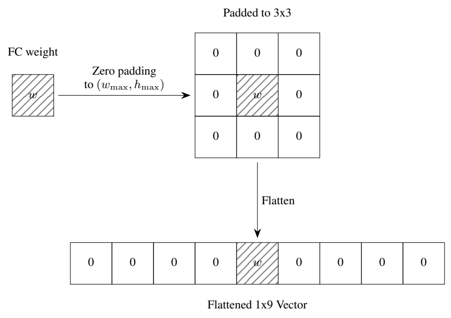
[15] Manzil Zaheer, Satwik Kottur, Siamak Ravanbakhsh, Barnabas Poczos, Russ R Salakhutdinov, and Alexander J Smola. Deep sets. *Advances in neural information processing systems*, 30, 2017.

## A Neural networks as graphs

In this section, we outline the methodology for constructing graph representations for FFNNs or CNNs, as established by Kofinas et al. [7]. This graph, subsequently processed by a metanetwork, is formally denoted as $G = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ represents the set of vertices and $\mathcal{E}$ the set of edges. Vertex features are denoted by $\mathbf{x}_V \in \mathbb{R}^{|\mathcal{V}| \times d_v}$, and edge features are denoted by $\mathbf{x}_E \in \mathbb{R}^{|\mathcal{E}| \times d_e}$. For an FFNN, constructing vertex and edge features is intuitive. Vertex features are associated with neuron biases, while edge features represent the connection weights between neurons. This typically results in feature dimensions of $d_v = 1$ and $d_e = 1$. In contrast, for a CNN, the mapping is more nuanced. Vertex features remain scalar ($d_e = 1$) and correspond to the biases of convolutional kernels and neurons in fully connected layers. Edge features, however, are structured to capture the convolutional weights. To ensure a uniform representation, each edge feature is padded to a fixed size $d_e = w_{\max} \cdot h_{\max}$, where $w_{\max}$ and $h_{\max}$denote the maximum kernel width and height across all channels, respectively. Edges corresponding to weights in fully connected layers are also mapped to this unified feature dimension to maintain consistency across the network representation. An example of this mapping is illustrated in Figure 1.


3x3 Kernel
$k_{11}$ $k_{12}$ $k_{13}$ $k_{21}$ $k_{22}$ $k_{23}$ $k_{31}$ $k_{32}$ $k_{33}$
Zero padding to $(w_{\max}, h_{\max})$
Padded Kernel
Flatten
Flattened 1x9 Vector

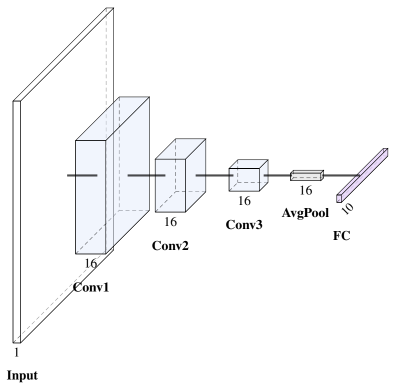
(a) Edge features from a convolutional kernel.



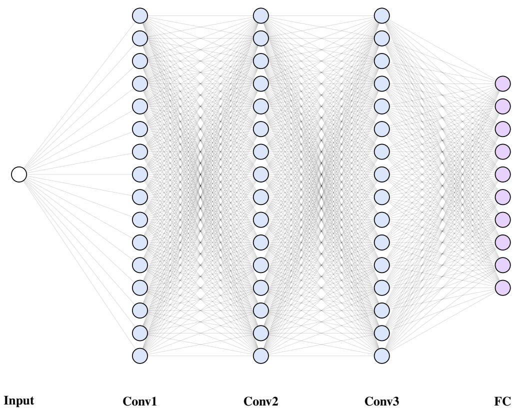
(b) Edge features from a fully connected weight.

Figure 1: Illustration of edge feature representations for different edge types: (a) edges corresponding to convolutional kernels and (b) edges corresponding to weights in fully connected layers, with $w_{\max} = h_{\max} = 3$. This figure is inspired by Figure 6 in Appendix C of Kofinas et al. [7].

Figure 2 illustrates a CNN alongside its corresponding graph representation. The input node corresponds to the number of input channels, 1 in this example, as the network processes grayscale images. Since input vertices do not represent actual parameters, their associated feature vectors are set to zero.



(a) CNN architecture.



(b) Graph representation.

Figure 2: Visualization of the mapping from a CNN to its corresponding graph structure. The CNN illustrated is an example architecture from the Small CNN Zoo [14]. Our methodology follows the approach described in Kofinas et al. [7] for constructing graph representations of CNNs.

Once a NN architecture has been converted into its graph representation, as illustrated in Figure 2, we can perform operations on the vertex and edge features using a GMN. The forward pass of a GMN with $T$ layers consists of four key stages: feature initialization, message passing, feature updating, and readout:

$$\mathbf{h}_V^0(i) = \text{INIT}_V(\mathbf{x}_V(i)), \quad \mathbf{h}_E^0(i,j) = \text{INIT}_E(\mathbf{x}_E(i,j)) \tag{Init}$$

$$\mathbf{m}_V^t(i) = \bigoplus\nolimits_{j \in \mathcal{N}(i)} \text{MSG}_V^t \left(\mathbf{h}_V^{t-1}(i), \mathbf{h}_V^{t-1}(j), \mathbf{h}_E^{t-1}(i,j)\right) \tag{Msg}$$

$$\mathbf{h}_V^t(i) = \text{UPD}_V^t(\mathbf{h}_V^{t-1}(i), \mathbf{m}_V^t(i)), \quad \mathbf{h}_E^t(i,j) = \text{UPD}_E^t \left(\mathbf{h}_V^{t-1}(i), \mathbf{h}_V^{t-1}(j), \mathbf{h}_E^{t-1}(i,j)\right) \tag{Upd}$$

$$\mathbf{h}_G = \text{READ}\left(\{\mathbf{h}_V^T(i)\}_{i \in \mathcal{V}}\right), \tag{Readout}$$

where $\mathbf{h}_V^t, \mathbf{h}_E^t$ are vertex and edge representations in the $t^{th}$ layer. The functions INIT, MSG, and UPD are general function approximators. In Kalogeropoulos et al. [6], these functions are designed to be equivariant to scaling transformations. Additionally, the READ function, beyond being permutation invariant, is also required to be invariant to different scalar multipliers applied to each vertex. Further details on these symmetry-preserving functions are provided in Appendix B.

## B Symmetry preserving graph meta neural nets

In this section, we derive the components of the ScaleGMN developed by Kalogeropoulos et al. [6]. Since the experiments in this work focus exclusively on CNNs, we then demonstrate how to construct the Initialisation, Update, Message, and Readout functions, directly from the permutation and scaling symmetries inherent to CNNs.[3]

### B.1 Derivation of the Graph Transformations corresponding to CNN Symmetries

A MetaNetwork designed to process CNNs should be invariant or equivariant to functionally equivalent representations of these input CNNs. Such equivalences stem from inherent scaling and permutation symmetries of neuron activations within the CNN architecture.

A typical convolutional layer $\ell$ in a CNN computes the activation $x_{\ell,i}$ at each spatial location $i$ by applying a shared kernel (or filter) over a local region of the input feature map. The kernel is defined by a set of weights $k_\ell(r)$, indexed by relative positions $r \in \mathcal{R}$, where $\mathcal{R}$ denotes the set of offsets covered by the kernel (In our experiments we only have 3×3 kernels e.g., $\mathcal{R} = \{-1, 0, +1\}^2$). At each location i, the kernel is centered and applied to the corresponding input patch, producing a weighted sum of input activations $x_{\ell-1,i+r}$. A shared bias term $b_\ell$ is added, and the result is passed through a nonlinearity $\sigma_\ell$:

$$x_{\ell,i} = \sigma_\ell \left( \sum_{r \in \mathcal{R}} k_\ell(r) \cdot x_{\ell-1,i+r} + b_\ell \right). \tag{4}$$

This operation can be vectorized for the entire layer as: $\mathbf{x}_\ell = \sigma_\ell(\mathbf{W}_\ell \mathbf{x}_{\ell-1} + \mathbf{b}_\ell)$. Here, $\mathbf{x}_{\ell-1}$ and $\mathbf{x}_\ell$ are vectors representing the neuron activations at the input and output of the layer, respectively. The matrix $\mathbf{W}_\ell$ is an effective weight matrix that represents the linear transformation performed by the convolution. Due to the shared filter weights and local receptive fields inherent in CNNs, $\mathbf{W}_\ell$ is typically a sparse matrix with a structured pattern (A doubly block Toeplitz matrix [2] for the 2D convolutions done in our experiments). Its entries are derived from the shared filter weights, arranged according to the specific connections (receptive fields, stride) defined by the convolution. The vector $\mathbf{b}\ell$ is the corresponding effective bias vector, formed by appropriately replicating the bias term associated with each filter across all spatial locations where the filter is applied .

Consider a transformation of neuron activations by permutation matrices $\mathbf{P}$ and diagonal scaling matrices $\mathbf{Q}$. If the input to layer $\ell$ transforms as: $\mathbf{x}'_{\ell-1} = \mathbf{P}_{\ell-1}\mathbf{Q}_{\ell-1}\mathbf{x}_{\ell-1}$ . For the CNN layer to be

[3]This section is included to provide theoretical background and is not part of the novel experiments presented in this work. Where possible, we adopt the notation of the original work [6], while presenting their framework from our perspective.

equivariant, meaning its output transforms correspondingly as $\mathbf{x}'_\ell = \mathbf{P}_\ell \mathbf{Q}_\ell \mathbf{x}_\ell$, the parameters $\mathbf{W}_\ell$ and $\mathbf{b}_\ell$ must transform as follows:

$$\mathbf{W}'_\ell = \mathbf{P}_\ell \mathbf{Q}_\ell \mathbf{W}_\ell \mathbf{Q}_{\ell-1}^{-1} \mathbf{P}_{\ell-1}^{-1}, \qquad \mathbf{b}'_\ell = \mathbf{P}_\ell \mathbf{Q}_\ell \mathbf{b}_\ell. \tag{5}$$

Furthermore, this requires the activation function $\sigma_\ell$ to satisfy $\sigma_\ell(\mathbf{P}_\ell \mathbf{Q}_\ell \mathbf{z}) = \mathbf{P}_\ell \mathbf{Q}_\ell \sigma_\ell(\mathbf{z})$, a property common to many element-wise activations under appropriate conditions on $\mathbf{Q}_\ell$. We can simply verify these transformations by writing out:

$$\begin{aligned}
\mathbf{x}'_\ell &= \sigma_\ell(\mathbf{W}'_\ell \mathbf{x}'_{\ell-1} + \mathbf{b}'_\ell) \\
&= \sigma_\ell\left((\mathbf{P}_\ell \mathbf{Q}_\ell \mathbf{W}_\ell \mathbf{Q}_{\ell-1}^{-1} \mathbf{P}_{\ell-1}^{-1})(\mathbf{P}_{\ell-1} \mathbf{Q}_{\ell-1} \mathbf{x}_{\ell-1}) + (\mathbf{P}_\ell \mathbf{Q}_\ell \mathbf{b}_\ell)\right) \\
&= \sigma_\ell\left(\mathbf{P}_\ell \mathbf{Q}_\ell \mathbf{W}_\ell (\mathbf{Q}_{\ell-1}^{-1} \mathbf{P}_{\ell-1}^{-1} \mathbf{P}_{\ell-1} \mathbf{Q}_{\ell-1}) \mathbf{x}_{\ell-1} + \mathbf{P}_\ell \mathbf{Q}_\ell \mathbf{b}_\ell\right) \\
&= \sigma_\ell\left(\mathbf{P}_\ell \mathbf{Q}_\ell \mathbf{W}_\ell \mathbf{x}_{\ell-1} + \mathbf{P}_\ell \mathbf{Q}_\ell \mathbf{b}_\ell\right) \quad (\text{as } \mathbf{P}_{\ell-1}^{-1} \mathbf{P}_{\ell-1} = \mathbf{I}, \mathbf{Q}_{\ell-1}^{-1} \mathbf{Q}_{\ell-1} = \mathbf{I}) \\
&= \sigma_\ell\left(\mathbf{P}_\ell \mathbf{Q}_\ell (\mathbf{W}_\ell \mathbf{x}_{\ell-1} + \mathbf{b}_\ell)\right) \\
&= \mathbf{P}_\ell \mathbf{Q}_\ell \sigma_\ell(\mathbf{W}_\ell \mathbf{x}_{\ell-1} + \mathbf{b}_\ell) \quad (\text{by activation property}) \\
&= \mathbf{P}_\ell \mathbf{Q}_\ell \mathbf{x}_\ell.
\end{aligned}$$

Thus, the parameter transformations defined in Equation 5 ensure CNN layer equivariance with respect to neuron activation permutations and scalings. When these transformations are applied consistently across all layers of a network, the internal permutations and scalings effectively cancel out between layers. This implies a significant gauge freedom in choosing sets of internal parameters, all of which lead to functionally equivalent models.

**Mapping CNN symmetries to Graph Features.**
A meta-network designed to process CNNs should be invariant or equivariant to functionally equivalent representations of its input. Because the meta-network operates on the graph representation of the CNN, we examine how the (functionally equivalent) transformations of equation 5 transform the corresponding graph representation of the CNN. In this graph, each neuron $i$ from CNN layer $\ell$, with bias $b_{\ell,i}$, is a vertex with feature $\mathbf{h}_V(i) = b_{\ell,i} \in \mathbb{R}$. An edge connecting a "sending" neuron $s$ (CNN layer $\ell - 1$) to a "receiving" neuron $r$ (CNN layer $\ell$), has an edge feature $\mathbf{h}_E(r,s) \in \mathbb{R}^{d_e}$ representing the flattened kernel. Edge features are zero-padded to $d_e = w_{\text{max}} \cdot h_{\text{max}}$ to unify kernel dimensions, as visualized in Figure 1. Consider a symmetry operation at layer $\ell$ defined by a permutation $\pi_\ell : \mathcal{V}_\ell \to \mathcal{V}_\ell$ and a scaling function $q_\ell : \mathcal{V}_\ell \to \mathbb{R} \setminus \{0\}$. The vertex and edge features then transform according to Equation 5 as follows:

$$\mathbf{h}'_V(i) = q_\ell(\pi_\ell(i)) \mathbf{h}_V(\pi_\ell(i)), \tag{6}$$

$$\mathbf{h}'_E(r,s) = q_\ell(\pi_\ell(r)) \mathbf{h}_E(\pi_\ell(r), \pi_{\ell-1}(s)) q_{\ell-1}^{-1}(\pi_{\ell-1}(s)). \tag{7}$$

Next, we will demonstrate how to build a meta-network (ScaleGMN) that is designed to be invariant/equivariant to these transformations.

## B.2 ScaleEquivariant Net

To construct an invariant or equivariant function, it is helpful to define a canonical representation of objects $\tilde{o} = \text{canonical}(o)$, such that all objects equivalent ($\sim$) under a scaling transformation map to the same canonical form. Formally, if $x \sim y \in \mathcal{X}$, then $\tilde{x} = \tilde{y}$. In the case of scaling transformations, a natural canonical mapping is given by $\tilde{x} = x/|x|$. This transformation allows for any function $\rho(x)$ to be converted into a scale-invariant function by first applying the canonicalization, i.e., using $\rho(\tilde{x})$. In the ScaleGMN framework [6], a MLP is employed for this base function $\rho$, leveraging its capabilities as a universal approximator. This invariant MLP belonging to layer $k$ is denoted as:

$$\text{ScaleInv}^k(\mathbf{X}) = \rho^k(\tilde{\mathbf{x}}_1, \dots, \tilde{\mathbf{x}}_n). \tag{8}$$

For a scale-equivariant network, the goal is to construct functions $f$ such that if the input $\mathbf{X}$ is scaled by a factor $q$ (i.e., each component $\mathbf{x}_i$ of $\mathbf{X}$ becomes $q\mathbf{x}_i$), the output also scales by the same factor: $f(q\mathbf{X}) = qf(\mathbf{X})$. Linear transformations $\Gamma : \mathcal{X} \to \mathbb{R}^d$ inherently possess this property, as $\Gamma(q\mathbf{x}) = q(\Gamma\mathbf{x})$. Building on this set of transformations, we can construct more expressive equivariant functions by recognizing that if a scale-equivariant function is multiplied (element-wise

$\odot$) by a scale-invariant quantity, the result is itself scale-equivariant [4]. Kalogeropoulos et al. [6] use this generalization to construct a learnable ScaleEquivarient function, potentially composed of $K$ layers $f^1, \ldots, f^K$, and denote each layer $f^k$ as:

$$\text{ScaleEq} = f^K \circ \cdots \circ f^1, \quad f^k(\mathbf{X}) = (\Gamma_1^k \mathbf{x}_1, \ldots, \Gamma_n^k \mathbf{x}_n) \odot \text{ScaleInv}^k(\mathbf{X}). \tag{9}$$

Here, $\mathbf{X} = (\mathbf{x}_1, \ldots, \mathbf{x}_n)$ is the input to the $k$-th layer. $\Gamma_i^k$ are learnable linear transformations applied to each respective input component $\mathbf{x}_i$. $(\Gamma_1^k \mathbf{x}_1, \ldots, \Gamma_n^k \mathbf{x}_n)$ is a concatenated vector which scales by $q$ if each $\mathbf{x}_i$ scales by $q$. The term $\text{ScaleInv}^k(\mathbf{X})$ is the scale-invariant function defined previously, whose output does not change with the scaling of $\mathbf{X}$. The Hadamard product $\odot$ then ensures that the overall output $f^k(\mathbf{X})$ scales by $q$, thus achieving scale equivariance for the layer.

This general learnable equivariant function (ScaleEq) can now be used to construct equivariant initialization (INIT), message (MSG), and update (UPD) functions. For initialization, Kalogeropoulos et al. [6] use the simplest instance of the equivariant function (Equation 9) with one layer ($k = 1$), and where the invariant component is omitted: $\text{INIT}_{V,E}(\mathbf{x}) = \mathbf{\Gamma x}$.

Integrating the scale equivalence into the MSG component of the Graph Neural Network is less trivial. The primary challenge arises from the nature of message passing in GNNs, which involves interactions between different types of features that may not share the same scaling behavior. A message function processes features from a sending node, a receiving node, and the connecting edge. If these components $(\mathbf{x}_{\text{sender}}, \mathbf{x}_{\text{receiver}}, \mathbf{e}_{\text{edge}})$ scale by different factors ($q_s, q_r, q_e$ respectively) under a global transformation, the simple ScaleEq formulation (9), is not directly applicable. Ultimately, for a GNN to achieve scale equivariance, the message passed to a receiving node should scale proportionally with that receiving node's scaling factor. This ensures that the information flow remains consistent with the node's transformed representation, e.g the message function should satisfy $\text{MSG}_V(q_r \mathbf{x}_\text{r}, q_s \mathbf{x}_\text{s}, q_e \mathbf{e}) = q_r \text{MSG}_V(\mathbf{x}_\text{s}, \mathbf{x}_\text{r}, \mathbf{e})$. To address this Kalogeropoulos et al. [6] first develop a learnable *rescale invariant* function $g$ that is defined by the property: $g(q_1 \mathbf{x}_1, \ldots, q_n \mathbf{x}_n) = g(\mathbf{x}_1, \ldots, \mathbf{x}_n) \prod_{i=1}^n q_i, \quad \forall q_i \in D_i$. In the ScaleGMN equivariant framework, $g$ is implemented as a set of component-wise learnable linear transformations followed by an element-wise product aggregation:

$$\text{ReScaleEq}(\mathbf{x}_1, \ldots, \mathbf{x}_n) = \bigodot_{i=1}^{n} \Gamma_i \mathbf{x}_i. \tag{10}$$

It trivially follows that this function satisfies the conditions for rescale invariance [5]. Leveraging this function, the scale equivariant message update function can now be constructed as:

$$\text{MSG}_V(\mathbf{x}_r, \mathbf{x}_s, \mathbf{e}) = \text{ScaleEq}([\mathbf{x}_r, \text{ReScaleEq}(\mathbf{x}_s, \mathbf{e})]). \tag{11}$$

Indeed, after applying the scaling transformations to vertex and edge features, as defined by equations 6 and 7 respectively, we can demonstrate the desired equivariance property of the message function $\text{MSG}_V$:

$$\begin{aligned}
\text{MSG}_V(q_r \mathbf{x}_r, q_s \mathbf{x}_s, q_r \mathbf{e} q_s^{-1}) &= \text{ScaleEq}\left(\left[q_r \mathbf{x}_r, \text{ReScaleEq}(q_s \mathbf{x}_s, q_r \mathbf{e} q_s^{-1})\right]\right) \\
&= \text{ScaleEq}\left(\left[q_r \mathbf{x}_r, (q_s)(q_r q_s^{-1}) \text{ReScaleEq}(\mathbf{x}_s, \mathbf{e})\right]\right) \\
&= \text{ScaleEq}\left([q_r \mathbf{x}_r, q_r \text{ReScaleEq}(\mathbf{x}_s, \mathbf{e})]\right) \\
&= q_r \text{ScaleEq}\left([\mathbf{x}_r, \text{ReScaleEq}(\mathbf{x}_s, \mathbf{e})]\right) \\
&= q_r \text{MSG}_V(\mathbf{x}_r, \mathbf{x}_s, \mathbf{e}).
\end{aligned}$$

This confirms that the message scales with the scaling factor of the receiving node $\mathbf{x}_r$, as required for scale equivariance. Note that in the transformation, we omit permutations $\pi$ as their effect is handled by the inherent design of a Graph Message-passing Network, which is naturally equivariant to node and edge permutations through its aggregation mechanisms.

Finally, having established the scaling properties of the aggregated message $\mathbf{m}$ (derived from the $\text{MSG}_V$ function detailed above), the scale-equivariant update function, $\text{UPD}_V(\mathbf{x}, \mathbf{m})$, is implemented

[4]Let $g_E$ and $g_I$ be equivariant and invariant functions respectively. And let $f(\mathbf{X}) = g_E(\mathbf{X}) \odot g_I(\mathbf{X})$, then $f(q\mathbf{X}) = g_E(q\mathbf{X}) \odot g_I(q\mathbf{X}) = (q g_E(\mathbf{X})) \odot g_I(\mathbf{X}) = q(g_E(\mathbf{X}) \odot g_I(\mathbf{X})) = q f(\mathbf{X})$.

[5]Indeed, $\text{ReScaleEq}(q_1 \mathbf{x}_1, \ldots, q_n \mathbf{x}_n) = \bigodot_{i=1}^n \Gamma_i(q_i \mathbf{x}_i)$. Linearity of $\Gamma_i \implies \bigodot_{i=1}^n (q_i \Gamma_i \mathbf{x}_i)$. Element-wise product $\bigodot \implies (\prod_{j=1}^n q_j)(\bigodot_{k=1}^n \Gamma_k \mathbf{x}_k) = (\prod_{j=1}^n q_j)\text{ReScaleEq}(\mathbf{x}_1, \ldots, \mathbf{x}_n)$.

directly as an instance of the ScaleEq function (defined in Equation 9). This direct application is valid because both inputs to the update function (the node's own features $\mathbf{x}$ and the aggregated message $\mathbf{m}$) have been shown to scale by the same factor $q_r$.

### B.3 ScaleInvariant Net

For the scaleGMN as a functional, the network needs to be invariant to CNN weight-space symmetries. This is achieved by applying an invariant readout function that acts on the output of the equivariant network described above. For the readout function $\text{READ}_V$, which maps node features to a graph-level output, Kalogeropoulos et al. [6] employ a strategy that combines canonicalization for scale invariance with Deep Sets [15] for permutation invariance. For hidden nodes, an inner MLP, $\phi$ (which can be integrated with the canonicalization step), transforms each $\tilde{\mathbf{x}}_i$. These transformed hidden features are subsequently aggregated ( via summation: $\sum_i \phi(\tilde{\mathbf{x}}_i)$). Finally, this aggregated representation of hidden nodes, concatenated with features from I/O nodes, is processed by an outer MLP to produce the graph output.

## C Hyperparameters

All fine-tuning procedures across methods are performed using a batch size of 256. Hyperparameter selection is based on the lowest validation cross-entropy loss. For finetuning, we use early stopping with a patience of 10 epochs.

### C.1 Invariant Pruning

Table 2: Hyperparameters for Invariant Pruning on SmallCNN. Some hyperparameters are specific for the sparsity ratio, which is indicated in brackets. All hyperparameters are found through a grid search.

| Hyperparameter (Pruning) | Value |
|---|---|
| Learning Rate | 1e-3 |
| Batch Size | 256 |
| Threshold ($\epsilon$) | {1e-3 (60%,90%), 1e-4 (70%, 80%)} |
| $L_1$ coefficient ($\lambda$) | 1e-4 |
| Optimizer | AdamW |
| **Hyperparameter (Finetuning)** | **Value** |
| Learning Rate | 1e-3 |
| Batch Size | 256 |
| Patience | 10 |
| Optimizer | AdamW |

### C.2 Equivariant Pruning

Our equivariant pruning approach involves two stages: first, training a ScaleGMN to produce a sparsified version of a given network, and second, fine-tuning the nonzero parameters independently, without involving the ScaleGMN. This two-step process complicates end-to-end hyperparameter optimization, so we adopt heuristics to guide the design of the ScaleGMN architecture.

We closely follow the original operator application from Kalogeropoulos et al. [6] in the INR editing task, using a residual learning technique to produce the new network weights:

$$\boldsymbol{\theta}' = \boldsymbol{\theta} + \hat{f}_{\boldsymbol{\phi}}(G, \boldsymbol{\theta}).$$

Our architectural choices for the pruning operator $\hat{f}$ are driven by the observation that pruning a known architecture is more tractable than tasks such as CNN generalization prediction or INR editing. To balance predictive capacity and computational feasibility, we configure ScaleGMN as a two-layer Message Passing Neural Network with a hidden dimension of 18 for both node and edge features.

This design is primarily informed by practical memory constraints, especially when processing large networks such as VGG19.

The GNN is trained using a batch size of 256 and the Adam optimizer. We set the learning rate to 0.0001 for the VGG architecture and 0.001 for the smallCNN Following the INR editing experiment in Kalogeropoulos et al. [6], each MLP block within the GNN uses the SiLU activation function, is equipped with layer normalization, and incorporates skip connections between GNN layers. We do not apply dropout or weight decay, as our focus is on achieving effective sparsity during pruning; the model is later fine-tuned for performance recovery. We also avoid applying any scaling constants to the output parameterization of $\hat{f}_{\phi}$.

The regularization strength $\lambda$ in Equation 3 is chosen so that the cross-entropy loss and $L_1$ sparsity penalty are of comparable scale, resulting in $\lambda = 0.0001$ for VGG and $\lambda = 0.001$ for smallCNN. We also employ gradient clipping with a norm threshold of 10, which proved critical for ensuring stable training dynamics. We train each ScaleGMN for 300 epochs and retain the parameterization that achieves the lowest validation loss throughout the training process.

During the fine-tuning phase, we use the Adam optimizer with a batch size of 256. For each sparsity level, we perform a grid search to identify the optimal learning rate. Specifically, we search over the set 1e-2, 1e-3, 1e-4 for the smallCNN, and over 1e-2, ..., 1e-7 for VGG19. This procedure yields optimal learning rates of 1e-4, 1e-2, 1e-4, and 1e-4 for smallCNN at 60%, 70%, 80%, and 90% sparsity, respectively. For VGG19, the optimal learning rates are 1e-6, 1e-5, 1e-7, and 1e-4 corresponding to sparsity levels of 90%, 95%, 97%, and 99%, respectively. The optimal learning rate is selected based on the minimum validation cross-entropy loss.

## C.3 Baselines

To evaluate the performance of the pruning methods proposed in this work, we compare ScaleGMN against three baselines. First, standard $L_1$ regularization serves as a simple, widely-used baseline. Second, we include magnitude-based pruning, another straightforward method that has demonstrated competitive results in earlier work [3, 4]. Finally, we compare against Iterative Magnitude Pruning (IMP) [1], an extension of magnitude pruning that repeatedly prunes and fine-tunes the model. This iterative process has been shown to maintain high accuracy even at high sparsity levels. All related hyperparameter are specified in the table below.

Table 3: Hyperparameters for fine-tuning after pruning with the pruning baselines. Learning rates are specified per pruning fraction.

| Model | Pruning Ratio | Magnitude-Based LR | IMP LR | $L_1$ LR |
|---|---|---|---|---|
| SmallCNN | 60% | 1e-3 | 1e-3 | 1e-3 |
| | 70% | 1e-2 | 1e-3 | 1e-3 |
| | 80% | 1e-2 | 1e-3 | 1e-2 |
| | 90% | 1e-2 | 1e-3 | 1e-2 |
| VGG19 | 90% | 1e-4 | 1e-4 | 1e-5 |
| | 95% | 1e-4 | 1e-4 | 1e-5 |
| | 97% | 1e-4 | 1e-4 | 1e-5 |
| | 99% | 1e-3 | 1e-4 | 1e-5 |

| Additional Hyperparameter | Magnitude-Based | IMP |
|---|---|---|
| Optimizer | Adam | Adam |
| Batch Size | 256 | 256 |
| Patience | 10 | 10 |