UNIMORE
UNIVERSITÀ DEGLI STUDI DI
MODENA E REGGIO EMILIA

UNIVERSITY OF MODENA AND REGGIO EMILIA

"Enzo Ferrari" Department of Engineering

Master's Degree in Artificial Intelligence Engineering (LM-32)

Adapting Pre-trained Neural Networks for Dynamic Inference: from Residual Weight Generation to Slimmable Fine-tuning

Supervisor:
Prof. Simone Calderara
Co-supervisors:
Angelo Porrello, PhD.
Pietro Buzzega

Candidate:
Pietro Moriello
Student ID 195236

ACADEMIC YEAR 2024/2025

Abstract

The rigid computational requirements of traditional neural networks often conflict with the need for hardware-agnostic deployment across a broad range of diverse real-world systems. Frameworks like Slimmable Neural Networks offer a dynamic alternative within a single model but traditionally require training from scratch, failing to leverage the presence of existing pre-trained weights.

This work investigates adapting pre-trained models for runtime flexibility. We start from Neural Metamorphosis, a generative approach that uses hypernetworks to produce weights for various sub-network configurations. To address scalability issues in generating weights for larger architectures, we incorporate ideas from classical pruning and reframed the task as a residual problem. Instead of generating weights from scratch, the hypernetwork produces an additive change to be applied to the pre-trained weights, making the optimization landscape more tractable.

Building on the insights of this first part, we transition to a strategy of direct fine-tuning. Specifically, we explore the possibility of adapting static, pre-trained neural networks into dynamic ones by applying the training principles of Slimmable Networks. This progression highlights the challenges of scaling weight generation and evaluates the viability of converting standard pre-trained models into dynamic ones without the necessity of full retraining.

Contents

1 Introduction **1**

2 Background **3**

2.1 Neural Networks 3

2.1.1 Fully-Connected Neural Networks 3

2.1.2 Convolutional Neural Networks 4

2.1.3 Vision Transformer 7

2.2 Efficient Neural Networks 9

2.2.1 Pruning 10

2.2.2 Neural Metamorphosis 12

2.2.3 Slimmable Networks 17

3 Extending NeuMeta **24**

3.1 Initial experiments 28

3.1.1 Focusing on reconstruction 31

3.2 Weight magnitude based permutations 37

3.3 Residual approach 40

3.3.1 Motivation 42

3.3.2 Implementation 44

3.3.3 Experiments 44

3.3.4 Data-driven permutation via ThiNet. 45

4 Slimmable Fine-Tuning of Pre-Trained Networks **52**

4.1 Slimmable adaptation with CNNs 53

4.1.1 Initial experiments with direct fine-tuning 55

4.1.2 Reducing the update size via Low-Rank Adaptation (LoRA) 57

4.2 Moving to Vision Transformers 63

4.2.1 ViT slicing strategy 64

4.2.2 Full Slimmable adaptation experiments 71
4.2.3 Slimmable adaptation with LoRA 74

5 Conclusions **77**

References **78**

List of Figures

2.1 Residual Neural Networks. 6
2.2 Vision Transformer. 9
2.3 Slimmable Networks. 18

3.1 Bottleneck slicing. 25
3.2 INR architecture. 27
3.3 "W-B Hypernet" metamorphism on ResNet20 Layer3 without smoothing of pre-trained weights. 33
3.4 Comparison of the effect of coordinate noise during training of metamorphic ResNet20's Layer1. 34
3.5 Potential benefits of weights permutations for saliency-aware slicing. 36
3.6 Weight magnitude based ordering of convolutional filters. 38
3.7 TV smoothing vs. L1 ordering for ground-truth conditioning in sliced reconstruction approach. 39
3.8 "W-B Hypernet" with L1 ordering sliced reconstruction approach. 40
3.9 Accuracy on CIFAR-10 for ResNet20 with L1 ordering and residual weight adaptation. 46
3.10 Impact of different filter ordering on ResNet20 CIFAR-10 accuracy vs. slicing ratio without retraining. 48
3.11 Residual adaptation accuracy on CIFAR-10 for ResNet20 with ThiNet ordering. . . 50
3.12 Performance degradation of residual adaptation on Resnet20 on unseen width ratios. 50

4.1 Test accuracy of Universally Slimmable ResNet20 trained on CIFAR-10. 54
4.2 Example of LoRA update matrices with slicing applied. 60
4.3 Test accuracy of pre-trained ResNet20 adapted with different LoRA ranks, across various slimmable ratios. 62
4.4 Slicing scheme for ViT layer. 66
4.5 Slicing of Attention projections with Attention-aware slicing. 67
4.6 Evaluation of semantic-aware slicing of Attention-related weight matrices. 71

4.7 ViT-Tiny/16 slimmable adaptation with different ordering strategies. 74
4.8 LoRA-based Slimmable adaptation accuracy for ViT-Tiny/16 with L1-score ordering. 76

1. Introduction

In recent years, the field of deep learning has been characterized by the pursuit of increasingly larger model architectures. Driven by empirical observations of scaling laws, which suggest that model performance improves predictably with increases in parameter count, dataset size, and computational budget, modern Deep Neural Networks have grown significantly in scale. Consequently, while these models achieve impressive results across various downstream tasks, their computational and storage requirements might often exceed the capabilities of more constrained deployment environments. Given the recent rise in interest for Deep Learning applications, deployment to mobile or embedded devices, which clearly offer less resource capacity than large server-based environments, has become increasingly popular. Thus the efficiency of Neural Networks has begun to play a central role in both industry and academic research.

Another driver of Deep Learning's popularity has been the dissemination of a vast number of these high-performance models, made publicly available by the research community through dedicated online repositories, such as the HuggingFace Hub [1] or PyTorch Hub [2]. These platforms provide access to pre-trained checkpoints that can be readily integrated into diverse applications. However, these repositories typically offer only a single, fixed configuration of a given architecture. This "one-size-fits-all" delivery presents a significant hurdle: if the provided checkpoint's resource demands exceed the hardware budget of a specific system, the model cannot be easily deployed without structural modification. This limitation makes the development of methods for adapting pre-trained models highly desirable, as it would allow practitioners to leverage existing knowledge while enabling a necessary accuracy-efficiency trade-off.

Traditionally, achieving runtime flexibility has been addressed through two primary frameworks. The first is pruning, which creates multiple independent versions of a network tailored to specific budgets. While effective, pruning necessitates the storage and management of numerous model instances, leading to significant storage overhead. The second framework is Slimmable Neural Networks, which allow a single model to support multiple sub-network configurations. However,

[1] `https://huggingface.co/`

[2] `https://pytorch.org/hub/`

slimmable models typically require training from scratch, which fails to leverage the extensive knowledge already present in existing pre-trained weights.

This thesis investigates strategies for adapting static, pre-trained neural networks into dynamic models capable of runtime adjustment without the necessity of full retraining. We explore two distinct methodologies for this adaptation. The first approach is generative, building upon Neural Metamorphosis (NeuMeta). This method utilizes hypernetworks to generate weights for various sub-network configurations. To address scalability issues inherent in generating weights for larger architectures, we reframe the task as a residual problem: instead of generating weights from scratch, the hypernetwork produces additive changes to be applied to the pre-trained weights.

The second approach transitions from generative weight production to a strategy of direct fine-tuning. We investigate the feasibility of inducing the nested weight structure of Slimmable Networks into non-slimmable pre-trained models. This part of the work evaluates whether standard pre-trained models can adapt to these structural constraints through lightweight fine-tuning or if the process necessitates more intensive retraining.

In Chapter 2, we begin by introducing Deep Neural Networks, specifically the architecture families of Residual Convolutional Neural Networks and Vision Transformers, and we follow with an overview of various methods for implementing the accuracy-efficiency trade-off in DNNs. Chapter 3 illustrates the path undertaken for extending the generative adaptation framework outlined in Neural Metamorphosis, focusing on residual weight generation and permutations of pre-trained weights to enhance performance and stability. In Chapter 4, we transition our focus from a generative setting to the direct fine-tuning of pre-trained networks into their Slimmable versions, examining the role of channel ordering and parameter-efficient fine-tuning techniques. Chapter 5 concludes with a brief summary of our work and ideas for future research directions.

2. Background

2.1 Neural Networks

2.1.1 Fully-Connected Neural Networks

The elementary building block of Artificial Neural Networks (ANN) is the *neuron*, or alternatively called *perceptron*. In its simplest form, a neuron computes a weighted sum of its inputs, mapping a vector $x \in \mathbb{R}^N$ to a scalar output $y \in \mathbb{R}$. This transformation is parametrized by a set of learned *weights* $w \in \mathbb{R}^N$, and a bias term $b \in \mathbb{R}$, which shifts the output:

$$y = \sum_{i=0}^{N} w_i x_i + b.$$

A *layer* is a set of M neurons that share the same inputs. The layer's weights can be compactly represented as a matrix $W \in \mathbb{R}^{N \times M}$, where each column corresponds to a neuron's weights. This formulation enables efficient computation via matrix-vector multiplication, producing an output vector $y \in R^M$: $y = W^T x + b$, where the bias term is now a vector $b \in \mathbb{R}^M$. The dimension of each neuron's weight vector is called layer's *input channels*, while the output dimension defines its *output channels*.

In *Multi-Layer Perceptrons* networks, inputs are transformed into high-level representations through a sequence of interconnected layers. The defining characteristic of an MLP is its fully-connected structure, in which every neuron in a given layer L_i is connected to every neuron in the preceding layer L_{i-1}. Each intermediate representation produced by a *hidden layer* is processed through a non-linear *activation function* ϕ. Popular non-linear operators found in modern ANNs are based on the Rectified Linear Unit (*ReLU*) [8], the sigmoid function *sigmoid* and the hyperbolic tangent function *tanh*. Given x as the input, W_l and b_l as parameters of layer l, an L-layer MLP is

then described by the following set of equations:

$$h_{l+1} = \phi\left(W_l^T h_l + b_l\right),$$
$$l = 0, \dots, L,$$
$$h_0 = x.$$

Theoretical work showed that fully-connected neural networks are universal function approximators: given a mapping $f : X \to Y$, a network with at least one hidden layer of arbitrary width can approximate f to any degree of accuracy [13]. In practice, fully-connected networks are implemented with multiple layers of fixed width, which is a compromise between their expressive power and the computational requirements.

2.1.2 Convolutional Neural Networks

A natural way of representing images is as a 3-dimensional tensor $I \in \mathbb{R}^{C \times H \times W}$, where C is the number of channels (commonly red, green, and blue for colour images), H and W are the height and width of the image in pixels, respectively. A Convolutional Neural Network (*CNN*) is specifically designed to process such tensor representations. Just like MLPs, CNNs are built by a sequence of layers and activation functions. Convolutional layers consist of a set of C filter banks, each composed of K kernels, which are modelled as small $H_k \times W_k$ matrices (typically $H_k = W_k = 3$). When working with images, these layers process 3-dimensional (or 4-dimensional in case of batching) inputs and produce tensors with the same dimensions but potentially different shape. Given channel c in the input, convolution is performed by sliding the n-th kernel in a filter across the width and height of the n-th input channel: at each step, the dot product is computed between the $H_k \times W_k$ patch and the kernel's weights. Values produced by different kernels in a filter are then summed together, so that the output will have the same number of channels as there are filters in the layer. As for MLPs, Convolutional Networks are built by a sequence of layers and activation functions (which in this case are pointwise applied to feature maps). Additionally, kernel sliding can be performed with *stride*, e.g. setting a stride of s will advance the kernel s pixels along its input at each step. Having $s > 1$ without introducing any padding to the input feature maps will result in feature maps with *downsampled* spatial dimensions. By leveraging depth and progressive downsampling, convolutional networks are able to generate high-level semantic representations of objects at the

scale of the entire image. This ability to generate low-dimensional vector representation that match the contents of an image is crucial for the overall performance of this kind of neural networks. It is worth mentioning that another system used for downsampling feature maps are *pooling* operations, which consist in applying aggregation functions (commonly mean or maximum) on $n \times n$ adjacent non-overlapping patches.

Utilizing a single kernel to process an entire channel brings a notable advantage to convolutional networks in terms of the number of parameters employed when compared to fully-connected ones, with the former being considerably more lightweight than the latter. In a fully-connected approach, all pixels of a feature map would become inputs to every neuron in the layer, resulting in $O(CHW)$ space complexity; on the other hand, the dimensions of kernels does not depend on the input's shape, so even a kernel with very few parameters is sufficient for processing an entire channel. Such level of *parameter sharing* stems from the structure of CNNs and their *inductive biases*, which are a consequence of assumptions made about the underlying properties of natural images: first, that meaningful global features can be constructed by looking at *local* features in neighbourhoods of adjacent pixels; second, that semantically relevant information does not depend on its location inside the image, which supposes *invariance under translation*.

CNNs have gained popularity thanks to their impressive abilities achieved in various image understanding tasks, such as image classification, object detection, instance segmentation, etc. Recently, a class of CNNs identified as *ResNets* [10] have been extensively employed in research. ResNets have opened the doors to convolutional networks with more than one hundred convolutional layers, which had been impossible to achieve up to that point due to challenges of training very deep networks. We briefly overview the characteristics of the ResNet-like architecture applied to image classification, since it will be extensively employed throughout this work. Following the example of previous approaches such as VGG [25] and GoogLeNet [27], ResNets are constructed by stacking a series of *blocks*. Each block is structurally identical as the others, but layers in different blocks may be defined according to different hyperparameters (i.e. kernel shape, output channels, stride, etc.). A block contains usually two or more convolutional layers, as depicted in Figure 2.1 (right). Batch Normalization [15], which normalizes activations with per-batch statistics through a affine transformation, is placed between convolutions and is followed by a ReLU operator. The main innovation presented in ResNets lies in the presence of a *skip connection* around a block:

through this connections, inputs are added to outputs computed by the block's function F, resulting in $y = F(x) + x$. In case F modifies the number of channels or spatial dimensions, a convolutional layer with 1×1 kernels, with appropriate stride and output channels, is placed in the skip path to map inputs to the correct output's shape. As intermediate representations are processed along the network, usually their number of channels grows with depth, while feature maps are downsampled. A latent, high-dimensional representation is obtained by condensing 2D feature maps from the last convolutional layer's output to a scalar value via global average pooling. For image classification, a fully-connected *classifier head* finally projects this latent representation to a vector of logits with dimension equal to the number of possible classes.

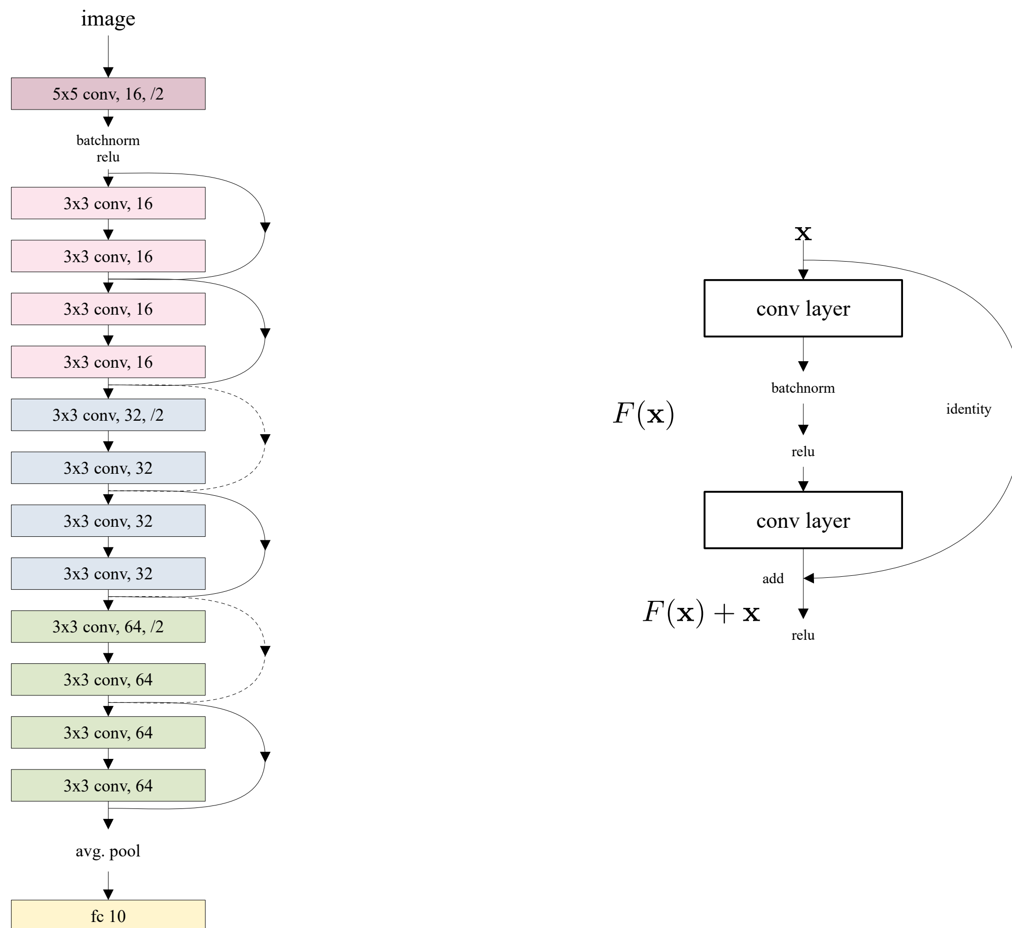

Figure 2.1: **Left**: a simple ResNet network. **Right**: structure of a residual block.

2.1.3 Vision Transformer

Vision Transformers [6] (*ViT*) are a family of deep neural networks based on the Transformer architecture from the field of NLP. Since this kind of architecture was originally conceived to learn from strings of character representing natural text, Transformers are not able to process images as tensors in the same way CNNs do; a Transformer, in particular the *encoder* configuration, which is the one used for ViTs, instead work by mapping a *sequence* of continuous vectors $(\mathbf{x}_1, \ldots, \mathbf{x}_n)$ to an output sequence of the same length $(\mathbf{e}_1, \ldots, \mathbf{e}_n)$. Given this fundamental difference in the shape of inputs to Transformers and the standard shape of image tensors, a transformation must be applied to get an an image $I \in \mathbb{R}^{C \times H \times W}$ into a sequence of *embeddings*. ViT's approach starts by dividing the image into $P \times P$ non-overlapping *patches*. Each patch is then flattened over the spatial dimensions, and the vectors obtained from each channel are concatenated. These (CP^2)-dimensional vectors are then projected to a dimension D. The final results is a sequence $\mathbf{x} \in \mathbb{R}^{N \times D}$, with $N = \frac{HW}{P^2}$ being its length. Elements of the sequence are also referred to as *tokens*.

The core of a Vision Transformer model's mechanism for sequence-to-sequence translation, just like for the original variant for text, is built upon the concept of *Attention*. Attention is a function that maps an input *query* to a output which is a weighted sum of *values*, where weights are computed by a similarity function between the query and a set of *keys* (the set of keys has the same cardinality as the set of values). The result is that the more similar a query is to a key, the more influence the value corresponding to that key will have on the output vector. In the case of *Self-Attention* commonly used in a Transformer Encoder, the sets of queries, keys and values are representations obtained from the same sequence of vectors.

Practically, Vision Transformers employ *Scaled Dot-Product Attention* to implement Self-Attention. SDPA computes similarity by composing a dot-product with the Softmax function. Given a input sequence of vectors stacked row-wise in a matrix $X \in \mathbb{R}^{n \times d}$, queries, keys and values result from projection through learnable linear transformations $W_Q \in \mathbb{R}^{d \times d_k}$, $W_K \in \mathbb{R}^{d \times d_k}$, $W_V \in \mathbb{R}^{d \times d_v}$:

$$Q = XW_Q, K = XW_K, V = XW_V.$$

Dot-products can be efficiently computed in parallel for all queries via matrix multiplication QK^T; then Softmax applied to each row, resulting in a $n \times n$ *attention matrix*. Finally, multiplying the

attention scores with V yields the outputs:

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

Before applying softmax, uniform scaling by $\sqrt{d_k}$ is applied in order to avoid large magnitues in dot-products when d_k is large. Experimental evidence in [29] has shown that computing multiple separate attention operations, called *heads*, in parallel using distinct projections for Q, K, V results in better performing models. Each head's output is then concatenated together and projected again via a final linear transformation:

$$\text{MultiHeadAttention}(Q, K, V) = \text{concat}(\text{head}_i, \ldots, \text{head}_h)W_O.$$

Since the embedding dimension D is kept constant throught the model, the common approach (and the one taken by ViTs) is to have $d_k = d_v = D/h$.

The architecture of the Vision Transformer is virtually identical to the encoder presented in [29]. It consists in a stack of identical layers, each composed of a Multi-Head Self-Attention block followed by a fully-connected 2-layer MLP with GELU [11] activation, which is applied independently to every position in the sequence. Layer Normalization [3] is applied both before Attention and MLP: this normalization approach differs from Batch Normalization in that mean and variance are not computed over a batch of inputs, but are instead computed considering outputs produced by the layer singularly. Skip connections are also employed around each subpart of the layer, as depicted in Figure 2.2. Notice that the computational complexity of the Attention operator is $O(L^2)$ given sequence length L; L is a function of both patch size P and image size, and in ViT models is a constant across all Transformer layers.

Following the approach presented in BERT [5], image representations are extracted via a learnable `[class]` token $\mathbf{x}_{\text{class}}$, which is appended to the sequence of embedded patches. This token is processed as any other, and is free to attend to any other position in the sequence. At the output of the encoder, $\mathbf{x}_{\text{class}}$ is passed as input to a fully-connected classification head, which outputs classification scores for the given task.

In stark contrast with CNNs, vanilla Transformers do not intrinsically present image-specific inductive biases such as locality or translation equivariance. In Vision Transformers, all tokens attend to every position in the sequence without restrictions based on the location of the corresponding

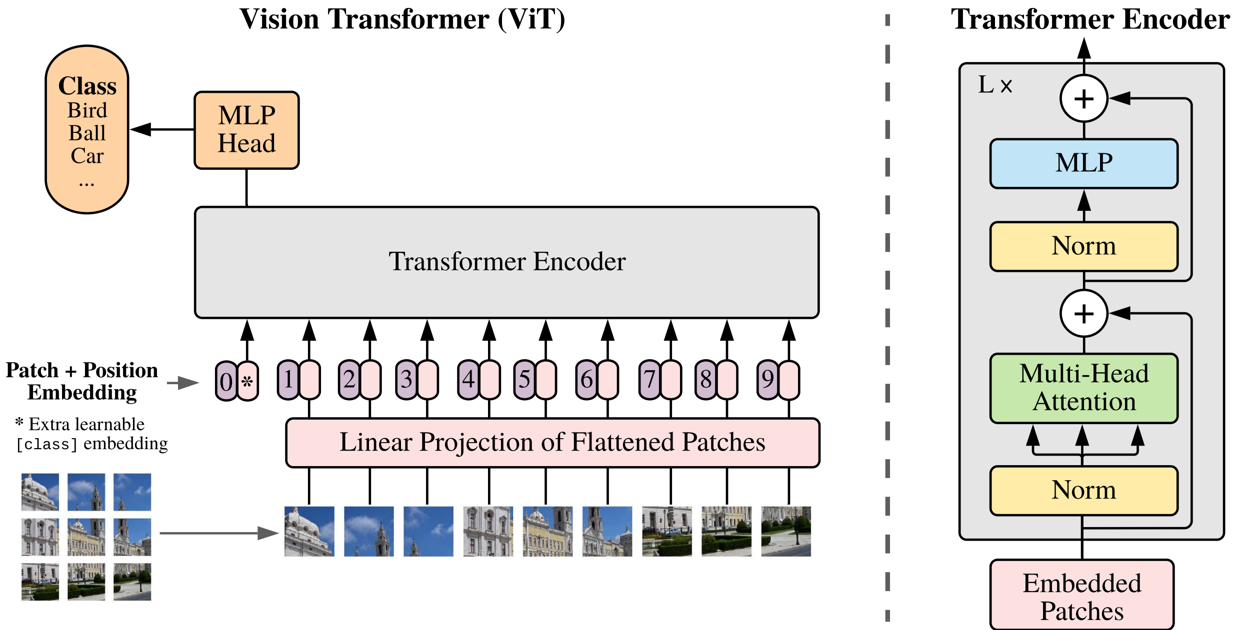

Figure 2.2: Illustration of Vision Transformer network architecture. Reprinted from [6].

patches in the image. Since Attention is invariant to permutations of the input sequence, in order to allow modelling relationships based on the relative position of tokens inside the sequence, in ViTs a *positional encoding* embedding is added to tokens after patch embedding. Still, such a lack of strong inductive biases allows these models to learn robust image representations directly from data, as corroborated by experiments conducted in [6]: ViTs were shown to gradually outperform ResNets as larger amounts of data are used for training. Consistent with this result, CNNs retain an advantage in low-data regimes.

2.2 Efficient Neural Networks

Models presented so far utilize static network architectures, where learned parameters are intrinsically tied to the predefined structure. Consequently, any modification to the layers (such as the removal of weights or neurons) typically results in a significant degradation of model performance. Consequently, standard neural networks are unable to dynamically adjust their parameter count after completing training. While high-capacity models generally offer superior performance, smaller architectures are prioritized for their computational efficiency and reduced memory footprint. With the recent interest in deploying neural networks on platforms with diverse capabilities, extensive

research has been conducted for developing techniques that can reconcile these objectives, enabling the deployment of compressed models that retain the performance characteristics of their larger counterparts under reduced hardware budgets.

2.2.1 Pruning

One of the most consolidated ways of reducing memory and computational requirements of deep neural networks is Pruning. Given a trained neural network, pruning consists in removing or disabling potentially unimportant parameters from the network until a satisfactory trade-off between the given requirements and accuracy is reached. Pruning relies on the empirical observation that modern deep neural networks are usually overparametrized, meaning that the dimensionality of their parameter space usually exceeds the intrinsic dimensionality of the learning objective [19]: the practical result is that a fraction (which may be considerable) of parameters will have a negligible influence on the final output of the network. One particularly interesting experimental demonstration of this fact is given by the Lottery Ticket Hypothesis (LTH) [7], which posits that within a randomly initialized, overparameterized neural network, there exists a sparse sub-network (a "winning ticket") which can achieve performance comparable to or even surpass that of the original dense network when trained in isolation. This hypothesis challenges the conventional wisdom that all components of a neural network are necessary for its success and suggests that much of the network's structure is redundant. In the case of pruning, we hope to retrieve such sub-network from one that has already been trained, without having to incur in the full cost of training a network, albeit smaller, from scratch.

In the context of pruning, *saliency scoring* (or importance estimation) serves as the selection criterion that identifies which components of the network are redundant. The objective is to assign a scalar value S to each parameter (or group of parameters, like a single kernel in a convolutional layer), such that the scores reflect the relative contribution of that element to the model's overall performance. By ranking these scores, we can define a threshold τ and remove all elements where $S < \tau$. This transforms the pruning problem into an optimization task: finding a scoring function that accurately predicts the increase in the loss function when a specific parameter is set to zero.

Scoring methods can be broadly categorized based on the information they utilize from the network into *heuristic* methods and *analytical* methods. Heuristic methods rely solely on the

intrinsic properties of the weights, making them computationally "cheap" as they do not require feeding data to the network or computing gradients. The most prevalent heuristic is *Magnitude-based Scoring*: it assumes that the "strength" of a connection is proportional to its absolute value. The saliency is defined as:

$$S_i = \|\mathbf{w}\|_p$$

where p is typically the ℓ_1 or ℓ_2 norm for structured groups. The underlying assumption is that weights close to zero have a negligible impact on the post-activation values and, by extension, the final prediction. *Analytical* methods estimate the importance of weights by observing how the loss function behaves in response to the input data. These methods employ usually the first of second-order derivative of the loss $\mathcal{L}$ with respect to the weight w to estimate saliency. One example among the first works on pruning, Optimal Brain Damage [18], proposed to use the second order gradient of the loss function with respect to the weights as the importance measure. In general, the idea is to give a lower score to those network components that have a small impact on the network's task performance.

Orthogonally to scoring mechanisms, the *granularity* at which pruning is applied strongly determines the characteristics of the resulting model. For simplicity, we will overview the two ends of the granularity spectrum: *structured* and *unstructured pruning*. The defining difference between these two approaches is the "level" of the architecture hierarchy at which parameters are removed. In structured pruning, scores are assigned coarsely to entire groups of parameters such as neurons of a linear layer or filters in a convolutional layer. This results in pruned networks that retain the same architecture configuration, crucially producing weights that can still be organized in dense matrices and tensors. Conversely, unstructured pruning treats each single parameter as an independent unit: each scalar weight w is set to zero, effectively disabling the "connection" in the network graph, without regard for its position within a weight matrix or tensor. Matrices and tensors of layer parameters retain the same exact shapes as those of the original model, but unstructured pruning renders them *sparse*. Sparsity is particularly challenging to handle if efficiency is key, because it introduces the need of specialized hardware accelerators or complex sparse-matrix software libraries, which often introduce overhead that negates the benefits of reduced parameter counts. Structured pruning has the advantage of not requiring any additional hardware or implementation support; models produced by applying structural pruning results in immediate reductions in latency

and memory footprint without requiring specialized implementations.

Removing entire groups of parameters introduces significant perturbations to the model's activations that lead to drops in accuracy. In order to recover performance, a recovery phase of training is carried out to adapt the remaining weights to the new network structure. Usually this fine-tuning phase is kept lightweight, with limited number of epochs on relatively fewer examples than what was used for initially training the model [20].

One major limitation of standard pruning methods is that they yield only a single static instance of the network. Handling multiple levels of sparsity requires repeatedly applying pruning and storing multiple models. This lack of flexibility introduces significant overhead when a system may need to *dynamically* adjust its active parameter count on-the-fly to accommodate changing resource constraints. In such scenarios, conventional pruning would require loading an entirely different network into memory: a process that is both storage-inefficient and too slow for real-time deployment.

2.2.2 Neural Metamorphosis

Neural Metamorphosis [31] introduces a novel learning paradigm for adaptable neural networks. The core innovation is the use of a *hypernetwork* [9] that models a *neural implicit function* to dynamically generate the weights of a target network. A *neural implicit function* is a function that maps *coordinates* that represent the location of a parameter in the network structure to its scalar value. Instead of learning a fixed set of weights for one particular configuration, an implicit representation learns to model the high-dimensional weight manifold of multiple configurations at the same time. Given the set of parameters of a pre-trained network θ_{pt}, a function $F(.;\theta^*)$ is trained on a dataset D. In the experiments provided by the paper, D is the same dataset used for training θ_{pt}. As introduced in previous works [2, 9] F takes as inputs a set of *virtual coordinates*. Since the aim is to sample parameters for a whole *space* of neural networks, each coordinate vector v uniquely identifies a parameter's position by representing its location inside the network and the specific network configuration it belongs to. Specifically, NeuMeta employs a normalized coordinate system:

$$v = \left(\frac{l}{L}, \frac{c_{in}}{C_{in}}, \frac{c_{out}}{C_{out}}, \frac{L}{N}, \frac{C_{in}}{N}, \frac{C_{out}}{N}\right).$$

where (L, C_{in}, C_{out}) represents the global configuration (depth and maximum input and output channel widths), (l, c_{in}, c_{out}) denotes the local position, and N is a rescaling constant. Following [2], Fourier Features [28] are applied to coordinates before being fed to the INR model. A simple MLP with skip connections is used to model the INR. Its output can either be a single scalar or a vector, depending on the kind of weights to generate.

Since the task the INR is expected to learn involves generating performant weights for multiple instances of a network without foregoing the pre-trained model's knowledge, the training objective adopted in NeuMeta is a combination of three signals:

$$\mathcal{L} = \mathcal{L}_{\text{task}} + \lambda_1 \mathcal{L}_{\text{recon}} + \lambda_2 \mathcal{L}_{\text{reg}}.$$

The task-specific loss $\mathcal{L}_{\text{task}}$ (such as Cross-Entropy for classification) is computed using the weights generated by the INR for a sampled configuration and gives a direct guidance on how well those weights perform on the downstream task of interest. The reconstruction term $\mathcal{L}_{\text{recon}}$ encourages the INR to align with the original weights θ_{pt}; Notably, this signal is active exclusively when sampling the "full" network configuration and is zero otherwise. $\mathcal{L}_{\text{reg}}$ constrains the complexity of the sampled weights to promote manifold stability. Reconstruction term and regularization terms are modulated by two coefficients λ_1 and λ_2, which are heuristically determined by experiments.

Authors then postulate that ensuring *manifold smoothness* is central to the capacity of INR to generate performant weights. Generally, weights of neural networks do not exhibit "smoothness", in the sense that they mainly exhibit high-frequency variations in their tensor representations. The need for smoothness stems from observing that deep neural networks tend to more easily learn to represent low-frequency components of functions [23]. To bridge this gap, NeuMeta proposes two strategies that tackle smoothness at different scales.

First, smoothness of the pre-trained weights θ_{pt} is obtained by leveraging *weight permutations* and symmetries in the dependency graph of neural network architectures. Given the existence of these symmetries and inter-layer relationships, permutations are applied independently on groups of interconnected layers, termed *cliques*, with the goal of obtaining a network functionally equivalent to the original. Each P is the result of an optimization problem that minimizes *Total Variation*, which is computed as the sum of difference of adjacent weights along each axis of the weight tensor. Specifically, the optimization is posed as a multi-objective *Shortest Hamiltonian Path* problem.

Second, to promote generalization across different configurations, rather than training F to generate weights for fixed virtual coordinates, noise is sampled from a uniform distribution $\epsilon \sim U(-a, a)$ and added (before Fourier feature expansion) to the input coordinates during training. This perturbation should promote robustness, as the learned mapping must generalize to slight variations in the input coordinates. Empirically, this improves the quality of generated weights for unseen architectures. Since coordinate noise is employed also after the hypernetwork has terminated training, generating weights becomes a stochastic process. Consequently, at inference time weights are generated by sampling K times with different coordinate noise and finally averaging all obtained outputs. The authors use $K = 50$ in their experiments.

The training process consists in sampling one configuration of the network at each iteration and feed the corresponding virtual coordinates into the INR. The generated weights are then used to perform a forward pass on the task data. Notably, to handle large number of parameters, the authors decide to employ multiple hypernetworks instead of a single one for the whole model; each hypernetwork is responsible for generating only a subset of the model's weights.

The authors conducted extensive experiments across three primary vision tasks: image classification, semantic segmentation, and image generation. The central finding is that NeuMeta maintains high performance even at high compression rates and, remarkably, can generalize to network configurations that were not seen during training. Nevertheless, it is worth noting that experiments on the ResNet architecture on image classification see NeuMeta applied only to the last two residual blocks of the network. More recent work [26] tries to extend generation to the whole network, achieving positive results. Still, the need for multiple dense hypernetworks introduces a non-negligible memory overhead.

Hypernetworks. The foundational concept of utilizing one neural network to predict the parameters of another was popularized by the framework of *Hypernetworks* [9]. In this paradigm, a smaller neural network, termed the *hypernetwork*, is tasked with generating the weights for a larger *main network*. These hypernetworks are trained end-to-end using standard backpropagation, making them significantly faster and more scalable than previous approaches which relied on evolutionary techniques.

The original work categorizes the approach into two primary use cases based on the architecture

of the main network: *static* and *dynamic* hypernetworks. For feedforward and convolutional architectures, the authors introduce static hypernetworks. Instead of learning a full, independent set of parameters for every layer, the main network's weights are generated from learned embedding vectors. Let the kernel weights of a convolutional layer j be denoted as K^j. The hypernetwork $g(\cdot)$ receives a specific layer embedding vector $z^j \in \mathbb{R}^{N_z}$ as input and outputs the required weights, such that $K^j = g(z^j)$. To achieve this efficiently, the hypernetwork is designed as a two-layer linear network that linearly projects the input vector into the kernel tensor. This configuration acts as a relaxed form of weight-sharing across layers, striking a structural balance between the hard weight-sharing of recurrent networks and the completely unshared parameters of standard convolutional networks. Consequently, the total number of learnable parameters in the hypernetwork is often much lower than what would be required for the main convolutional network directly.

To validate the efficacy of static hypernetworks, the authors conducted experiments on both shallow and deep convolutional architectures. In an initial proof-of-concept on the MNIST dataset, a hypernetwork was tasked with generating the filters for the second layer of a small two-layer convolutional network, which contained the bulk of the trainable parameters in the system. Impressively, a kernel originally comprising $12,544$ weights was represented by an embedding vector of size $N_z = 4$. The hypernetwork responsible for this generation required only $4,240$ parameters. This highly compressed model achieved a test accuracy highly competitive with the accuracy of the conventional baseline method.

Scaling this approach to deeper architectures, the framework was applied to Wide Residual Networks (WRN) [34] on the CIFAR-10 [17] dataset. The hypernetwork was configured to generate all of the kernels in the intermediate and deep convolution groups (identified with `conv2`, `conv3`, and `conv4` in the paper), totalling 36 layers of kernels, utilizing an embedding size of $N_z = 64$ for each layer. To accommodate the varying kernel dimensions inherent to residual networks, the hypernetwork generated basic kernels that were subsequently concatenated to form larger filters when required. Enforcing this relaxed weight-sharing constraint across the deep residual network resulted in an accuracy degradation of approximately 1.25% to 1.5%. The authors attribute this drop to the fact that different layers in deep networks are typically trained to extract different levels of features and require different kinds of filters to perform optimally; the hypernetwork enforces commonality across layers, restricting their ability to learn entirely independent optimal filters.

Despite the slight reduction in accuracy, the hypernetwork approach yielded drastic reductions in parameter count, addressing the core efficiency objectives. For instance, a baseline WRN 40-1 model achieved a 6.85% test error using 0.6M parameters, whereas its hypernetwork counterpart achieved an 8.02% error with merely 0.097M parameters. Similarly, the WRN 40-2 baseline required 2.2M parameters for a 5.33% error, while the hypernetwork version compressed this to 0.148M parameters with a 7.23% error.

Furthermore, the authors explored generating weights for fully connected networks using a different paradigm, namely feeding virtual coordinate locations (the spatial x, y coordinates of both the input pixel and the weight) into the hypernetwork. In an experiment on MNIST classifying digits with a simple MLP with a single hidden layer, an 801-parameter hypernetwork was used to generate a massive weight matrix of $200,704$ parameters. However, this configuration yielded poor performance, achieving only 93.5% accuracy compared to the 98.5% accuracy of a standard fully connected network. This result highlighted the limitations of the virtual coordinates-based approach for practical tasks, effectively justifying the authors' novel layer-embedding strategy.

Neural Representations for Neural Networks. Another work that utilizes hypernetwork to represent weights of pre-trained neural networks is *NeRN* (Neural Representations for Neural Networks). While both NeuMeta and Hypernetworks focus on generating such weights to manipulate the behaviour of the target network, NeRN instead aims at *compressing* a pre-trained model by encoding them as an implicit neural representation. Inspired by *Neural Radiance Fields* (NeRF), which represent continuous 3D scenes through coordinate-based mappings, NeRN treats the parameter space of a network as a signal to be reconstructed. It utilizes a compact multi-layer perceptron to map a coordinate tuple (l, f, c) denoting the layer, filter, and channel to the weights of the corresponding convolutional kernel.

Despite striving to use implicit representations for a different objective than NeuMeta, these two methods share some similarities. For instance, NeRN also recognizes the importance of managing the spectral bias typical of neural networks. Thus in order to effectively capture the high-frequency variations within the weight space, NeRN transforms these coordinates into a high-dimensional space via periodic positional embeddings. Another key challenge related to learning implicit representations over neural networks pointed out by the authors lies in overcoming the inherent lack

of spatial smoothness in raw weights; NeRN addresses this through *permutation-based smoothness*, which reorders kernels to minimize distance and simplify the learning task for the MLP.

The training process is driven by a multi-objective loss function that balances parameter-level accuracy with functional preservation:

$$\mathcal{L} = \mathcal{L}_{\text{recon}} + \alpha \mathcal{L}_{\text{KD}} + \beta \mathcal{L}_{\text{FMD}}. \tag{2.1}$$

Beyond a standard ℓ_2 reconstruction loss ($\mathcal{L}_{\text{recon}}$), the objective incorporates Knowledge Distillation (KD) [12] and feature map distillation (FMD). Both these last two components are computed by having the hypernetwork generate weights for the target network, executing the forward pass with said weights on training examples, and comparing the produced intermediate activations and soft probabilities with those output using the original weights. These distillation signals encourage the MLP to prioritize the reconstruction of weights that most significantly influence the original model's activations and final predictions. To maintain efficiency during training, NeRN adopts a stochastic sampling strategy, often utilizing magnitude-oriented sampling to focus optimization on the most influential parameters. Ultimately, like NeuMeta, this paradigm demonstrates the efficacy of representing model parameters as a continuous, coordinate-dependent function rather than a discrete tensor.

2.2.3 Slimmable Networks

A simple but effective approach to dynamic accuracy-efficiency trade-offs is the framework of *Slimmable Neural Networks* (S-Nets) [33], which introduces a simple and general method to train a single neural network executable at different predefined widths. Instead of training individual networks for varying resource constraints, a shared network is trained so that it can instantly adjust its active channels on the fly. Typically, a network is trained to switch between a discrete set of width multipliers, such as $0.25\times, 0.5\times, 0.75\times$, and $1.0\times$ the full channel capacity.

To achieve this, the training mechanism is designed to jointly optimize the shared parameters across all permissible width configurations. The primary objective when training an S-Net is to optimize its average accuracy across all available nested sub-networks (one sub-network is usually termed a "switch"). In practice, this is accomplished by computing an un-weighted sum of the training losses from all the different switches. For a given mini-batch of data, the forward

pass is executed sequentially for each switch in the predefined list. Rather than updating the weights immediately after a single sub-network is evaluated, the back-propagated gradients from all switches are accumulated. Once the gradients for every configuration have been aggregated, a single optimization step is performed to update the shared weights.

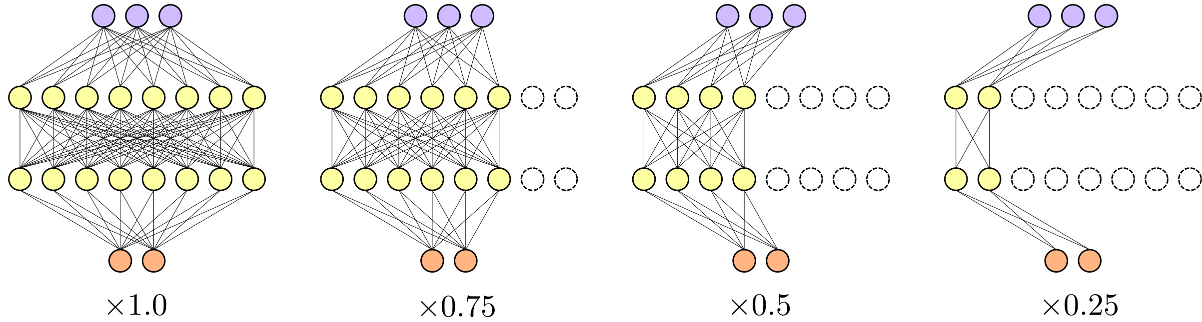

Figure 2.3: Example of Slimmable fully-connected network with two hidden layers. Slimmable Networks are trained to run at arbitrary widths. Smaller sub-networks are nested into larger ones, creating a hierarchical ordering of parameters inside the network.

A fundamental obstacle in natively training a shared network with varying widths is that accumulating a different number of channels significantly alters the feature mean and variance. This discrepancy leads to inaccurate statistics in shared Batch Normalization layers, causing the naive training approach to fail drastically, yielding test accuracies near 0.1% on ImageNet (random guessing over 1000 classes).

To recall, standard Batch Normalization [15] is a normalization technique that supposedly reduces "internal covariate shift" by normalizing the input feature x as follows:

$$y' = \gamma \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta,$$

where μ and σ^2 are the mean and variance of the current mini-batch during training, and γ and β are learnable scale and bias parameters. During inference, moving averaged statistics accumulated over the training data are utilized instead. To overcome the statistic inconsistency, S-Nets employ *Switchable Batch Normalization*, which allocates dedicated Batch Normalization layers for each specific width configuration in the network. By maintaining independent variables for the moving averaged means and variances, as well as separate scale and bias parameters for each switch, the

network resolves the feature aggregation inconsistency. The training process involves accumulating the gradients of all active switches before updating the weights, as formalized in Algorithm 1.

Algorithm 1 Training Slimmable Neural Network (S-Net)

Require: Predefined switchable width list (e.g., $[0.25, 0.5, 0.75, 1.0]\times$)

1: Initialize shared convolutions/linear layers for network M
2: Initialize independent BN parameters for each width
3: **for** $i = 1, \ldots, n_{iters}$ **do**
4: Get next mini-batch (x, y)
5: Clear gradients: `optimizer.zero_grad()`
6: **for** width $\in$ switchable width list **do**
7: Switch BN parameters of M to current width
8: Execute sub-network: $\hat{y} = M(x)$
9: Compute loss: $\mathcal{L}$ = criterion$(\hat{y}, y)$
10: Accumulate gradients: $\mathcal{L}$`.backward()`
11: **end for**
12: Update weights: `optimizer.step()`
13: **end for**

Universally Slimmable Networks

While S-Nets provide discrete adaptability, *Universally Slimmable Networks* (US-Nets) [32] expand this paradigm to allow execution at *arbitrary* widths. The theoretical justification for arbitrary slimming lies in observing feature aggregation as a form of *channel-wise residual learning*. If an output neuron aggregates n input channels as $y = \sum_{i=1}^{n} w_i x_i$, the residual error between the full aggregation y^n and a partial aggregation y^k decreases monotonically and is bounded:

$$|y^n - y^{k+1}| \leq |y^n - y^k| \leq |y^n - y^{k_0}|,$$

where $k \in [k_0, n)$, and k_0 represents a defined minimum width bound.

However, transitioning to arbitrary widths renders Switchable Batch Normalization highly impractical. Maintaining independent statistics for every possible sub-network introduces an $O(n^2)$

complexity of variables to update, rendering it computationally intensive and inefficient. Conversely, selectively sampling widths during training leaves the corresponding BN statistics insufficiently accumulated and inaccurate.

US-Nets bypass this limitation by entirely deferring the calculation of Batch Normalization statistics until *after* the training process. Because the network weights are fixed post-training, the exact feature mean and variance for all widths can be rapidly computed (possibly in a parallel fashion) using a randomly sampled subset of the training data. These post-statistics are computed via exact averages:

$$\mu_t = m\mu_{t-1} + (1-m)\mathbb{E}_B[x_B]$$
$$\sigma_t^2 = m\sigma_{t-1}^2 + (1-m)\mathrm{Var}_B[x_B]$$

where $m = (t-1)/t$.

To optimize performance across all arbitrary capacities, US-Nets rely on two specific training strategies: the *Sandwich Rule* and *In-place Distillation*. Guided by the bounds of the residual error, the Sandwich Rule posits that optimizing the model at its smallest and largest width bounds implicitly optimizes the configurations in between. Thus, during each training iteration, the forward pass is executed at the maximum width, the minimum width, and $(n-2)$ randomly sampled intermediate widths.

Furthermore, evaluating the maximum width in every iteration seamlessly enables In-place Distillation. Instead of using standard ground-truth labels for the intermediate and minimal sub-networks, the output logits (soft-probabilities) predicted by the full network are used as the teaching signal. Such distillation scheme requires no extra memory or computational cost and promotes representation consistency between different nested sub-networks. The unified US-Net training procedure is summarized in Algorithm 2.

Nested Dropout. The fundamental principle of Slimmable Networks, that a specific subset of weights can autonomously approximate the full set of network weights, is rooted in the concept of *ordered representations*, formally introduced in the work on *Nested Dropout* [24]. This work provides the theoretical precursor to the "sandwich rule" by proposing a stochastic regularization technique that enforces a strict importance ranking on hidden units.

Algorithm 2 Training Universally Slimmable Network (US-Net)

Require: Width range $[k_0, 1.0]\times$, number of sampled widths n

1: Initialize shared network M
2: **for** $i = 1, \ldots, n_{iters}$ **do**
3: Get next mini-batch (x, y)
4: Clear gradients: `optimizer.zero_grad()`
5: Execute full-network: $y' = M_{1.0}(x)$
6: Compute loss: $\mathcal{L}$ = criterion(y', y)
7: Accumulate gradients: $\mathcal{L}$`.backward()`
8: Detach predictions for distillation: $y'_{detach} = y'$.detach()
9: Randomly sample $(n-2)$ widths, add minimum width k_0 to $W_{samples}$
10: **for** width $\in W_{samples}$ **do**
11: Execute sub-network: $\hat{y} = M_{width}(x)$
12: Compute loss via distillation: $\mathcal{L}$ = criterion$(\hat{y}, y'_{detach})$
13: Accumulate gradients: $\mathcal{L}$`.backward()`
14: **end for**
15: Update weights: `optimizer.step()`
16: **end for**
17: **Post-Training**: Compute BN exact averages for all desired widths using data subset.

In standard dropout, the binary mask **m** is sampled independently for each unit. In contrast, Nested Dropout samples masks from a distribution over *nested subsets*. Let h_k denote the k-th unit of the representation. The dropping mechanism ensures that if unit h_k is retained, all preceding units $\{h_1, \ldots, h_{k-1}\}$ are strictly retained. This is analogous to the slimmable width constraint where a network of width k must utilize exactly the first k channels. The network is trained to optimize the expected loss over truncated sub-models:

$$\mathcal{L}(\theta) = \mathbb{E}_{k\sim P(\cdot)} \left[\mathcal{L}_k(\theta; \mathbf{x})\right],$$

where the truncation index k is sampled from a distribution $P(k)$ (e.g., geometric). This objective forces the network to pack the most critical information into the earliest dimensions $(h_1, h_2, \ldots)$, ensuring that the error between the partial representation and the full representation decreases monotonically.

The authors provide a rigorous proof that for single-layer linear autoencoders, this stochastic process is mathematically equivalent to Principal Component Analysis (PCA). Specifically, the learned weights converge to the principal eigenvectors of the data covariance matrix, ordered by eigenvalue magnitude. This establishes a theoretical bridge between stochastic regularization and spectral decomposition, justifying why "slimming" a network is feasible: the network learns a basis where information density is heavily skewed toward the lower indices, facilitating applications such as adaptive compression and retrieval via binary tree structures.

NestedNet. While Universally Slimmable Networks achieve dynamic efficiency by scaling the width of layers, *NestedNet* [16] proposes a more generalized framework for learning *Nested Sparse Structures* within a single parameter set. The core philosophy mirrors the "slimmable" concept: a single deep neural network is trained to behave as a set of nested sub-networks, where a model of lower capacity is strictly contained within a model of higher capacity. However, NestedNet extends this by considering not just width, but a broader hierarchy of nested parameters that can be invoked to meet varying resource constraints.

To formalize this, consider a full network with parameters Θ. NestedNet defines a series of n nested levels, where the parameters for level j are a subset of level $j+1$: $\Theta_1 \subset \Theta_2 \subset \cdots \subset \Theta_n = \Theta$. To better understand this structure, recall that in US-Nets, the subset of weights W_k is trained to approximate the full W. In NestedNet, this is achieved through a *Nested Sparse Strategy* that masks

weights based on their importance. For a given layer with weight tensor $\mathbf{W}$, the sub-network at level j uses only the weights $\mathbf{W} \odot \mathbf{M}_j$, where $\mathbf{M}_j$ is a binary mask defining the j-th nested structure.

The training process employs a multi-task learning objective, which can be viewed as a formalized precursor to the sandwich rule. The total loss is a weighted sum of the losses from all nested levels:

$$\mathcal{L}_{total} = \sum_{j=1}^{n} \lambda_j \mathcal{L}(\Theta_j; \mathbf{x}),$$

where λ_j are coefficients balancing the importance of each capacity level. Similar to the In-place Distillation found in US-Nets, NestedNet utilizes Knowledge Distillation [12] to maintain consistency across levels. Specifically, the outputs of the full-capacity network Θ_n serve as soft targets for the lower-capacity sub-networks $\Theta_{j<n}$. This ensures that even the most "slim" version of the network retains the high-level discriminative features learned by the full model.

The authors demonstrate that this nested architecture effectively mitigates the performance degradation typically seen in standard pruning. By enforcing the nested constraint during training, the network learns a weight importance hierarchy that allows it to transition between different bit-widths or filter counts at runtime without requiring separate parameter sets. This addresses the same accuracy-efficiency trade-off as US-Nets but focuses on the architectural sparsity required for diverse hardware environments, ranging from high-end GPUs to resource-constrained mobile devices.

3. Extending NeuMeta

In this chapter, we introduce our research steps into the generative approach of pre-trained network adaptation, following the work of Neural Metamorphosis. Although the original Neural Metamorphosis presents compelling results, especially regarding the capability of generating dynamic convolutional neural networks with the ResNet architecture, there are still some limitations to address. The most important one is that the method has been evaluated on a small subset of the full pre-trained network, namely the last convolutional block of the last layer. As illustrated in Figure 2.1 (right), a block consists of two convolutional layers, with the output channels of one convolution matching the number of input channels of the subsequent one. The block in question features convolutions with weight matrices of shape $[C_{\text{out}} = 64, C_{\text{in}} = 64, k_{\text{H}} = 3, k_{\text{W}} = 3]$, for a total of approximately 36.8K parameters, 7.25% of parameters belonging to convolutional layers in the whole network.

Another major limitation is represented by the ratio of hyperparameter parameters to the number of parameters in the network. NeuMeta proposes what the authors call a "block-based" INR method: the hypernetwork is divided into multiple dedicated INRs, each one receiving during training only the coordinates computed from a single weight tensor's shape. Before proceeding further, we give a closer look to the actual code implementation of this method. Upon analysing the code made publicly available by the authors, we noticed that the method described in the paper and its implementation present a slight discrepancy, which we briefly address. The former splits hypernetworks among different INRs based on the structural *type* of parameters: flat 1D tensors like biases are handled separately from multi-dimensional (4D for convolutions, 2D for fully-connected) layer weights. The latter has block-based hypernetworks implemented by allocating one INR for each *unique* tensor belonging to layers subject to metamorphic training: thus, as an example, two layers of identical type, each composed by a "weight" tensor and a "bias" tensor, would require a hypernetwork with 4 INRs.

With this minor technical aspect clarified, we can give an estimate the effective ratio of hypernetwork parameters to "metamorphic" main network parameters. Considering the ResNet20 [10] on CIFAR-10 [17] experiment setup, with $L = 4$ metamorphic layers and MLP architecture as

described by NeuMeta, the hypernetwork packs approximately 1.5M parameters, or 41.3× the number of weights to be generated. If we were to extend this setup, without modifications, simply by including more layers of the main network into the metamorphic set, considering all blocks of CIFAR ResNet20 would require around 12M parameters. In a practical scenario, ideally we would like to be able to target larger parts of the network (if not the whole network itself). Having broader scope enables a wider range of options for the performance-accuracy trade-off. Thus we question if it is feasible to train more memory-efficient hypernetworks while still ensuring the generated weights provide satisfactory performance-accuracy trade-off, but with improved scalability.

Baseline. The pre-trained network of choice for our experiment is a ResNet20 trained on CIFAR-10 for image classification. The network architecture and block structure correspond to those shown in Figure 2.1, with the only difference being that each layer (set of blocks with the same colour) packs three instead of only two blocks as presented in the diagram. The network has three layers to which we will refer to as **Layer1**, **Layer2** and **Layer3**; each one is composed of three blocks. The first blocks of Layer2 and Layer3 have a shortcut connection that matches the number of channels of the residual path with the convolution output. To create smaller versions of ResNet20, we modulate the number of channels of the intermediate feature maps in each residual block, as shown in Figure 3.1, creating a "*bottleneck*" shape. Since the space and computational complexity of a convolutional layer depend linearly on both input and output channels $O(C_{\text{in}} \cdot C_{\text{out}})$, reducing only one of the two dimensions at a time in each layer by a factor r results in a $r\times$ reduction in both memory and FLOPs. As the ground-truth, full size network we utilize a publicly available checkpoint [4]. To provide baselines for comparison of resized networks, we individually trained three additional configurations of ResNet20 resized with the "bottleneck" ap-

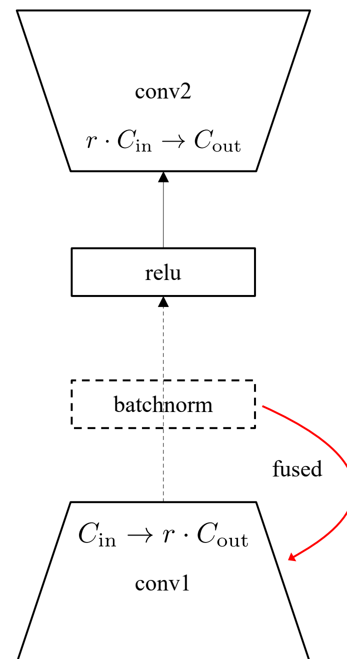

Figure 3.1: Bottleneck slicing

proach at $r \in \{0.25, 0.5, 0.75\}$. We follow the same training recipe of the ground-truth network. The training lasts 200 epochs with an initial learning rate of 0.1 and cosine annealing. The optimizer is SGD with decay 0.0005, momentum 0.9 and Nesterov optimization. Batch size is 256; each image undergoes random cropping and random horizontal flipping before normalization. Results are reported in Table 3.1.

Ratio	Top-1 Accuracy (%)	Param. count (conv + FC)	FLOPs
1.0	92.21	271.7K	40.81M
0.75	92.32	208.8K	30.79M
0.5	90.99	138.0K	20.76M
0.25	89.27	71.2K	10.73M

Table 3.1: Resnet20 CIFAR-10 individual training.

To handle Batch normalization layers, we follow NeuMeta's approach and fuse them into adjacent convolutional layers, leveraging the linearity of kernel convolution:

$$W' = \frac{\gamma}{\sqrt{\sigma^2 + \epsilon}} \cdot W,$$
$$b' = -\frac{\gamma}{\sqrt{\sigma^2 + \epsilon}} \cdot \mu + \beta,$$

where $b' \in \mathbb{R}^{C_{\text{out}}}$ is added as a bias parameter to the convolutional layer, and the scaling of W is broadcasted to the input and spatial dimensions.

INR and Hypernetworks. We made use of a single type of INR, following the architecture described in NeuMeta. The INR consists in a MLP with N hidden layers. Each hidden layer is composed by a $d \times d$ fully-connected block followed by ReLU activation. Positional encoding with f base frequencies is implemented as in NeRN [2]. As for the hypernetwork configuration, we decided to experiment with three setups differing in the number of INRs allocated and in the rule for assigning coordinates to each INR:

- **Single INR Hypernet**. As suggested by the name, this configuration features a single INR which processes all input coordinates of the metamorphic parameters, without distinctions.

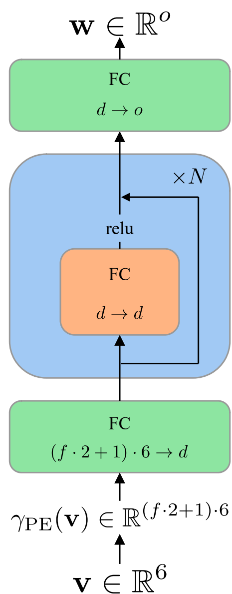

Figure 3.2: INR architecture.

This approach requires setting an appropriate output dimension for handling both vector parameters like $k \times k$ kernels and scalar parameters, like elements of fully-connected weight matrices or biases. Thus, for a given set of parameters comprising both convolutional square kernels and scalar weights, we set the MLP's output size to $o = k^2 + 1$ (we do not experiment with multiple kernels sizes at the same time, which simplifies the implementation). Scalar weights are obtained by selecting the first element of the output vector (hence the additional element); the remaining k^2 elements are first reshaped in a 2-D matrix before being returned.

- **Compose-Dict Hypernet**. This setup is identical to the practical implementation of NeuMeta. Parameters are input to the hypernetwork as dictionary and routed to the corresponding INR though their key. Since each INR processes only one parameter, a single output shape can be defined for each MLP, thus not requiring manipulation of the output vector (except for reshaping).

- **W-B Hypernet**. Following the original paper's setup, two INRs are created for separate handling of parameters marked with the `.weight` tag (using PyTorch's notation) from parameters marked with the `.bias` tag. The `bias` INR's output dimension is fixed to $o = 1$.

3.1 Initial experiments

The initial experiments we conducted revolved around a simple question: what happens when we naively extend the generations of weight from just a few convolutional layers to a whole network? We started from the setup of NeuMeta and applied some simplifications to it. The guiding principle of our choices was taking out factors that empirically provided minimal benefits in the experiments proposed in the original paper. The idea was to eliminate minor influencing factors and to get a "minimal" setting to study that is simple enough to be fast to train and evaluate and complete enough so that the experiments we conduct are nonetheless informative. Then any part removed can be later added to provide an additional benefit.

The first difference is that we do not perform sampling of coordinates. The hypernetwork always receives all coordinates from the given training configurations of the ground truth architecture and thus always produces a "complete" set of weights. We retain the sampling strategy and thus pick a different configuration at each iteration. The second difference is that no exponential moving average is applied to the hypernetwork weights.

The rest of our training setup closely follows the original one. We train solely on CIFAR-10, thus we usually apply a global batch size of 128. To ensure reproducibility, all experiments were conducted with a fixed random seed of 42. Excluding the training of baselines, we employ the AdamW optimizer [21] in our experiments. We use the exact same implementation of TV smoothing provided by the authors. Except when explicitly stated, we do not modify the weights of the loss components, which we leave at $\lambda_{\text{recon}} = 1.0$, and $\lambda_{\text{reg}} = 10^{-4}$. We also set the hyperparameter a for uniform sampling of coordinate noise to 1.0 and we keep it fixed. We use $K = 50$ sampling steps as in the original method and do not change this value either. We also keep the coordinate vector normalization constant $N = 64$ fixed for every experiment.

Metamorphism across the network. The first experiments we conducted aimed to uncover how the performance of the hypernetwork is affected by the choice of the "part" of network to generate. We trained a "Single INR" hypernetwork on a ResNet20 model with two different "scopes": reconstructing the full network (18 total convolutional layers) and reconstructing only Layer3 (excluding the shortcut). Table 3.2 provides a breakdown of the number of parameters of

the ResNet20 architecture chosen.

Table 3.2: Architecture parameters breakdown for CIFAR-10 ResNet20

Component	Params (No Bias)	Params (With Bias)	% of Conv Total
Stem (Conv1)	1,200	-	-
Layer 1	13,824	13,920	5.19%
Layer 2	50,688	50,880	18.99%
Layer 3	202,752	203,136	75.82%
Shortcuts	2,560	-	-
FC Layer	640	-	-
Conv Total	**267,264**	**267,936**	**100.00%**
Grand Total	**271,664**	-	-

Following the notation presented in Figure 3.2, we set $f = 16$, $d = 256$, $N = 4$ and $o = 10$ (accounting for 3×3 convolutional kernels and a scalar bias). This choice amounts to 315.4K parameters. In order to observe the capability of the various hypernetworks to generalize to unseen width configurations, the lower bound of the interval of possible ratios to be sampled is set to 0.5: coordinates constructed from configurations under this resizing ratio are never observed by the network during training. We train the hypernet for 100 epochs with an initial learning rate of 0.001.

Table 3.3 shows the results. It is clear that the setup with a single hypernetwork is not able to learn parameters that are performant enough on the downstream task when reconstructing the whole network. Still, in neither case the performance of generated weights is remotely close to that of individually trained compressed models of Table 3.1, which suggested there was more room for improvement. Given that the parameters of Layer3 make up a consistent part of the network, we decided to keep experimenting with reconstructing it only.

Exploring hypernetwork setups. The next step was to investigate the behaviour of different hypernetwork configurations. Given that different setups require a different number of INRs, which also depend on the number of layers to be generated in the case of "Compose-Dict", we decided to control the dimensionalities of the INR such that in total all hypernetworks had roughly the same amount of parameters. The aim was to provide a fair comparison among these approaches. Taking

Ratio	Accuracy full	Accuracy Layer3 only
1.0	44.79	74.57
0.75	44.94	75.01
0.5	44.48	74.75
0.25	20.99	61.41

Table 3.3: Accuracy of full reconstruction vs. reconstruction of Layer3 only in ResNet20 on the CIFAR-10 dataset. Widths not seen during training are highlighted in bold.

as target the "Single INR" setup already presented, we chose the following configurations for the other two hypernetworks. For "Compose-Dict", we set $N = 2$, $f = 8$, $d = 128$; output dimension is set based on the parameter assigned to the INR to be either 9 or 1. For 12 different weight tensors to handle, its total size is 557.5K parameters. This hypernetwork features the smallest INR configuration and the largest number of parameters, since it relies on more than one INR at the same time. For "W-B Hypernet", we set $N = 4$, $f = 16$, $d = 128$, $o = 9$ for the `weight` INR; leveraging the fact that biases are much less numerous than convolutional kernels, we selected a tiny `bias` INR with $N = 1$, $f = 8$, $d = 128$, $o = 1$. The total size is 344.7K parameters. Table 3.4 summarizes our choices. As for the training setup, we kept the same exact recipe used in the previous training with "Single INR".

Table 3.4: Hyperparameter configurations and parameter counts for the proposed INR architectures. N denotes the number of hidden layers, f the number of positional embedding frequencies, d the hidden layer width, and o the output dimension.

Architecture	N	f	d	o	**Params (K)**
Single INR	4	16	256	10	315.4
Compose-Dict	2	8	128	9 or 1	557.5
W-B Hypernet					**344.7**
Weight INR	4	16	128	9	–
Bias INR	1	8	128	1	–

Ratio	Independent training	Single INR	Compose-Dict	W-B Hypernet
1.0	0.9212	0.7457	0.3934	0.7831
0.75	0.9099	0.7487	0.3919	0.7836
0.5	0.8923	0.7496	0.3917	0.7809
0.25	0.8191	0.6130	0.2537	0.6253

Table 3.5: Accuracy on CIFAR-10 of different hypernetwork setups trained on generating the last layer of a ResNet20. The original accuracy is 92.12%

The results are presented in Table 3.5. "W-B Hypernet" shows the best results in this setup, followed closely by "Single INR". Interestingly, the "Compose-Dict" setup was the worst performing one, despite having the most parameters. One possible explanation we give to this behaviour is that the numerous INRs of this hypernetwork might struggle to coordinate effectively the generation of parameters across different layers. In light of its empirical performance, we decided to concentrate our efforts mainly on the "W-B Hypernet" configuration.

Once again, even if some setups where able to at least produce seemingly functional hypernetworks, we observe a wide gap with the individual baselines, and most importantly, with the original results in NeuMeta.

3.1.1 Focusing on reconstruction

The empirical evidence collected during initial experiments suggests that training a single hypernetwork to represent an entire target architecture presents significant optimization challenges. To understand the root of these difficulties, it is necessary to examine the factors influencing how effectively a hypernetwork learns to generate weight distributions. A primary consideration is the permutation factor, specifically the utility of smoothing the target network according to a Total Variation score. The objective of such a smoothing transformation is to mitigate the spectral bias inherent in neural networks. By overcoming this bias, the task of learning the multidimensional weight manifold should theoretically become more tractable for the hypernetwork.

However, whether Total Variation smoothing represents the optimal strategy remains an open question. While its adoption is grounded in the need to manage manifold smoothness, other factors

may play an equally significant role in the complexity of the problem. To identify these variables, we analysed the experimental outcomes of NeRN [2], which operates on a similar yet fundamentally simpler objective: training a coordinate based representation to map spatial indices to the weights of a pretrained network.

Recall from Chapter 2 that NeRN shares several architectural similarities with NeuMeta. It employs a MLP as the hypernetwork and utilizes positional embeddings to project input coordinates into a high dimensional space. Its training objective is governed by a multicomponent loss function that includes reconstruction via normalized ℓ_2 loss, Knowledge Distillation to match the logits of the generated network with the pretrained target, and feature map distillation to align internal representations. This loss structure reflects a hierarchical philosophy. The low level component representing weights captures knowledge specific to the task in a noisy format due to the high dimensionality and redundancy of overparameterized models. The middle level comprising feature maps captures high level semantic representations and serves as a proxy for the behaviour of the network. Finally, the highest level represented by logits provides the most compact representation of the learned data distribution.

Notably, the results from NeRN offer a critical insight because the model achieves respectable performance even when trained without data or with random inputs. This suggests that pure reconstruction is the primary driver of weight generation quality. Furthermore, a single NeRN hypernetwork successfully reconstructs a full ResNet20 trained on CIFAR-10, which is a feat that is orders of magnitude more efficient in terms of parameters than current NeuMeta capabilities. Like NeuMeta, NeRN addresses the difficulty of modelling high frequency weight components through permutation smoothing. However, NeRN demonstrates that while smoothing is beneficial by improving performance by approximately 3% in smaller configurations, it is not strictly mandatory for convergence. This indicates that while the effect of smoothing is significant, the specific reliance on Total Variation smoothing could potentially be superseded by alternative strategies. Ultimately, the divergence between the two frameworks lies in the role of reconstruction. Whereas it is the central pillar of the success of NeRN, NeuMeta treats reconstruction as a secondary objective that is prioritized only during full network synthesis.

To validate this idea, we ran some experiments by changing the reconstruction scheme of NeuMeta training. Instead of limiting reconstruction to the full dimension weights, we extend it to

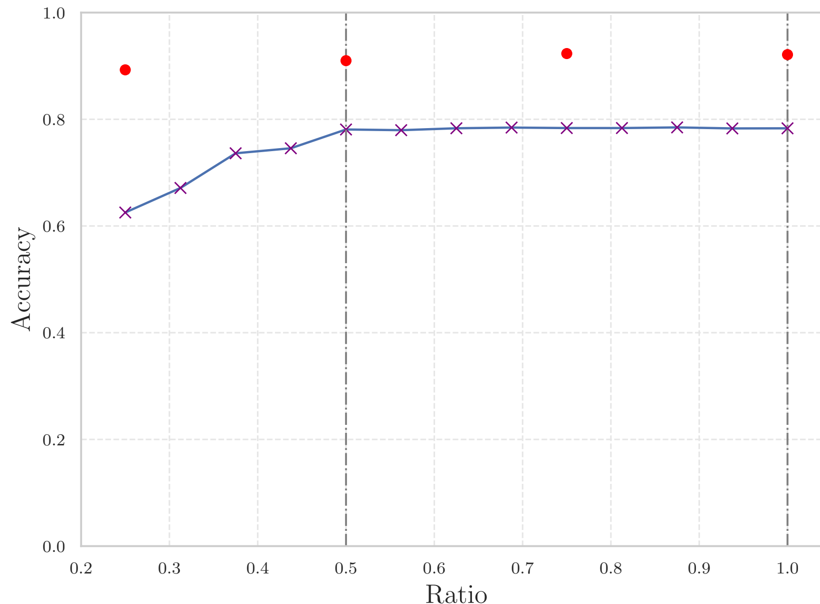

Figure 3.3: Test evaluation on CIFAR-10 achieved by a ResNet20 with metamorphism applied to Layer3. Weights are generated with "W-B Hypernet", without any smoothing of the pre-trained network.

every possible ratio by slicing the original weights and using these "partial" weights as ground truth. This allows to always have reconstruction loss. After this change of setup, we also reformulated the loss signal by determining new coefficients for each loss component, following a small grid search: we set $\lambda_{\text{reg}} = 0$ (effectively disabling regularization), $\lambda_{\text{task}} = 0.01$ and $\lambda_{\text{rec}} = 1$.

We repeat the experiment using the two best performing hypernetwork setups from previous experiments, both in term of task accuracy and number of parameters over total learned parameters. The results are encouraging: "W-B Hypernet" is able to achieve a new best over 100 epochs without any kind of smoothing: the hypernetwork learns by seeing the ground truth network *as is*. Still we have to notice that we are far from the original claims of NeuMeta. We observe a much sharper degradation in performance for weights generated at unseen compression ratios (below $r = 0.5$ in this case). This highlights the increased difficulty of our problem when we start to extend it to larger parts of the network. Probably in this same setting a Slimmable Neural Network trained from scratch would obtain a better result.

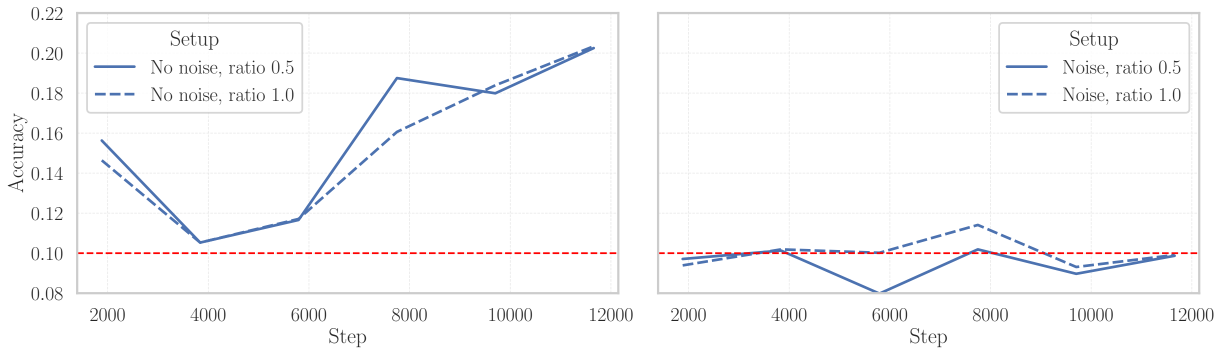

Figure 3.4: Comparison of the effect of coordinate noise during training of metamorphic ResNet20's Layer1. **Right**: coordinate noise is applied. **Left**: no coordinate noise is applied. Test accuracy is measured every 5 epochs.

Failing with smaller layers. One interesting thing happens when we go train on the first layer of ResNet20. Packing considerably less parameters than Layer3, intuition would suggest that our hypernetwork with the **same number of parameters** that was able to train successfully on a larger layer will be able to learn it. Surprisingly, the first exploratory experiments we conducted showed that in fact the best performing setup on Layer3 was not able to learn. We investigated further this failure.

The first, albeit simple, consideration we made concerns the application of uniformly sampled noise to coordinate vectors. In the original NeuMeta, noise is added as a way to promote smoothness in the weight manifold, which the authors argue is a key regularization strategy when generating weights through a neural network. Currently, our choice of shifting focus from smoothing to reconstruction essentially requires to rethink those considerations in this new context. Once again, going back to NeRN, we observe that no type of coordinate noise is employed to promote smoothness. Thus we asked if noise is still required at this stage and what are its effects. We ran two tests with a reduced training length, where the hypernetwork is trained on Layer1 without permutation smoothing, with and without coordinate noise. The learning rate is scheduled on the full 100 epochs training, but the training run itself is only 30 epochs, to simulate the rate of the initial part of a longer training run. In the case without any coordinate noise (left plot in Figure 3.4), we observed that the hypernetwork was able to start training. By adding back coordinate noise (right plot in Figure 3.4), the hypernetwork struggles to train and the average downstream task accuracy of the network with

generated weights does not significantly deviate from random guessing. Consequently, we decided to avoid using coordinate noise in our approach.

The second consideration is about the potential impact of weight location in the ground truth network when only a subset of weights is considered, i.e. slicing is applied to weight tensors. Imagine for instance that, as a result of resizing the intermediate channel dimension in the block as shown in Figure 3.1, some important filter/kernels are "deleted" at high values of r (e.g 0.9, when the lower limit of training is 0.5). Those weights would go missing also for all configurations with less channels than the one where the "cut" happens. Given the more contained number of channels in blocks of Layer1 (16 versus 64 of Layer3), it is possible that the available parameters in most configurations do not provide a sufficiently "good" base for the hypernetwork to effectively learn performant weights. In neural networks trained with stochastic initialization, in general no assumption can be made on the property of weights based on their location inside the network. In a standard training scheme the values of learned parameters are determined *relative* to other parameters by virtue of the connections between neurons, but not not on their *absolute* location inside their layer: no bias on weight location is imposed during training. One example where such a bias is instead present are Slimmable Neural Networks: training the network at various widths implicitly enforces an ordering to parameters, where weights at lower indices in the modulated weight tensor's dimension tend to store the "core" knowledge.

Figure 3.5 attempts to illustrate the possible downside of applying slicing on weight matrices without ordering structure. The top left matrix represents a generic weight matrix from a linear layer (the same argument can be applied to convolutional layers) $W \in \mathbb{R}^{N \times M}$. In this example, $N = M = 8$. Weights are visually represented with colours representing their "saliency": more salient parameters are coded with lighter colours, while a dark hue means less "interesting" weights. We note that that a measure of saliency could be determined by the numerical values of parameters themselves, such as in the case of ℓ_p scores, or it could be determined by other criteria.

The top-right diagram shows the slicing operation applied on the row axis, with a ratio $r = 0.5$: the rows occupying the first $rN = 4$ in lower indices are kept as the resized weight matrix. All rows with higher indices than the cutting point are discarded.

Two salient rows identified with $\mathbf{w}_i$ and $\mathbf{w}_j$ are highlighted in W. Without any intervention on their absolute location in the matrix, both would be discarded by the slicing operation considered in

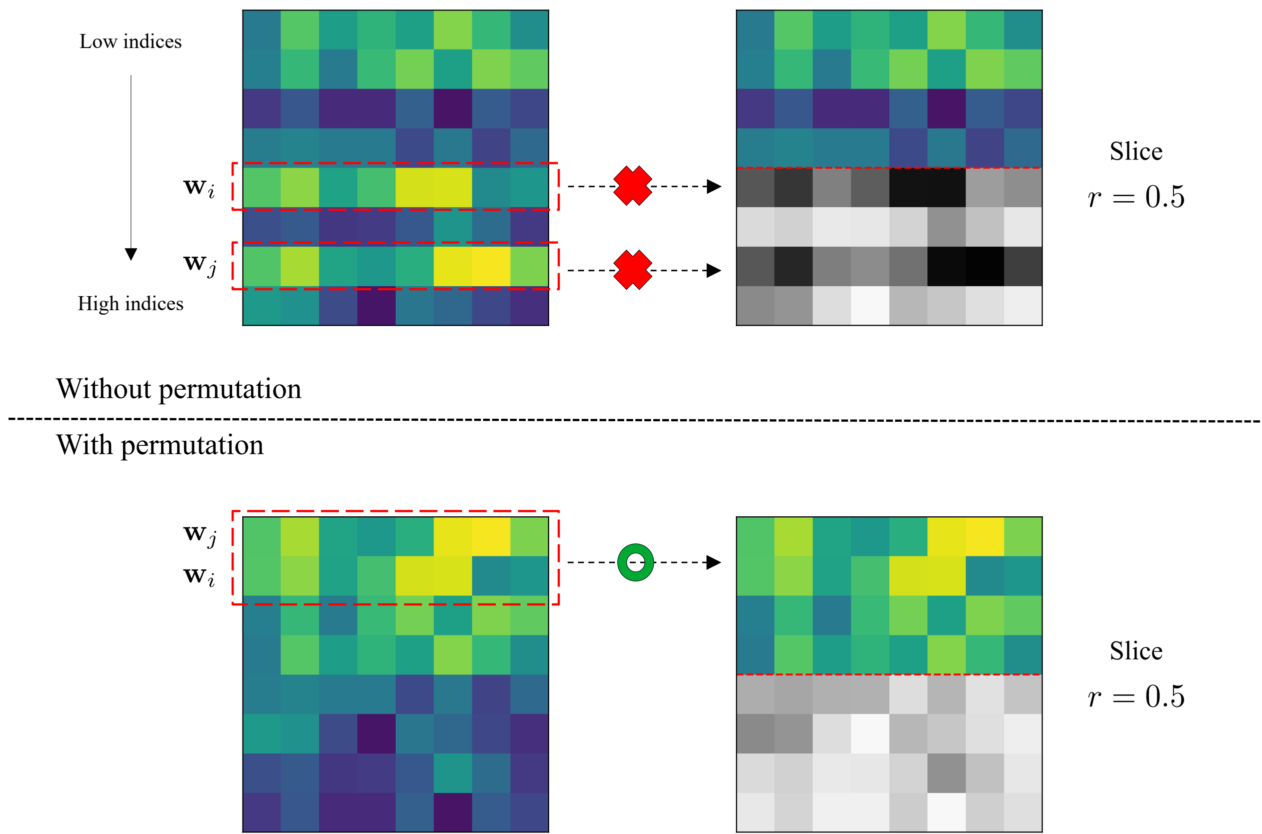

Figure 3.5: **Top**: A weight matrix is sliced along its vertical axis to half of its original dimension without modifying the position of weights. As highlighted by the red boxes, some "important" rows could end up being discarded from the set of rows preserved after slicing.

Bottom: A permutation is applied such that rows are ordered by highest "importance" first. Slicing should eliminate only the less important weights, while the important ones are preserved.

this example. In order to avoid such an unfavourable outcome, one solution would be to order rows based on the aggregated measure of saliency of their weights. The bottom-left part of the figure features the matrix W with such a permutation applied. $\mathbf{w}_j$ and $\mathbf{w}_i$ are mapped to first and second index rows. After slicing, salient weights are preserved as shown in the bottom-right part of the figure.

Consistently selecting the most salient weights after slicing provides first and foremost a possible "performant" ground-truth example (notice that, given the broad definition of "saliency" and "importance", there is realistically not one single best configuration) for the hypernetwork during training. With a reconstruction-oriented training, that would mean presenting the hypernetwork with a point on the weight manifold θ'_{pt} which, when projected down to lower dimensional parameter spaces by effect of slicing, would yield points $\Theta = \{\theta'_r\}$ with lower loss function than those obtained by projecting the unpermuted pre-trained weights, or weights permuted with strategies geared towards achieving different outcomes from slicing. We notice that TV permutation employed by NeuMeta is an example of the latter. Its permutation strategy aims at organizing weight matrices for minimizing the difference between adjacent weights, but such condition entails no guarantees about the quality of projected points θ_r.

3.2 Weight magnitude based permutations

In order to give the hypernetwork a ground truth prior that has channels ordered based on importance, we exploit the permutation equivariance of convolutional layers in ResNet. In the blocks under consideration, we generate weights for two sequential convolutional layers, $W^1 \in \mathbb{R}^{C_o \times C_i^1 \times k \times k}$ and $W^2 \in \mathbb{R}^{C_o^2 \times C_o \times k \times k}$. Since we aim to modulate the number of channels produced of feature maps between the two layers, W^1 is sliced in the output channel dimension, while W^2 is sliced in the number of input channels (in order to match the size of intermediate feature maps). For any permutation matrix P, the functional mapping of the two layers remains invariant if P is applied to the output filters of W^1 and the input kernels of W^2. In practice, the permutation reorders the channels of the intermediate feature map without changing the output of the block:

$$W^2 * (W^1 * x) = W^2 P^T * (P W^1 * x).$$

As introduced in the previous section, we want our permutation to order following some kind

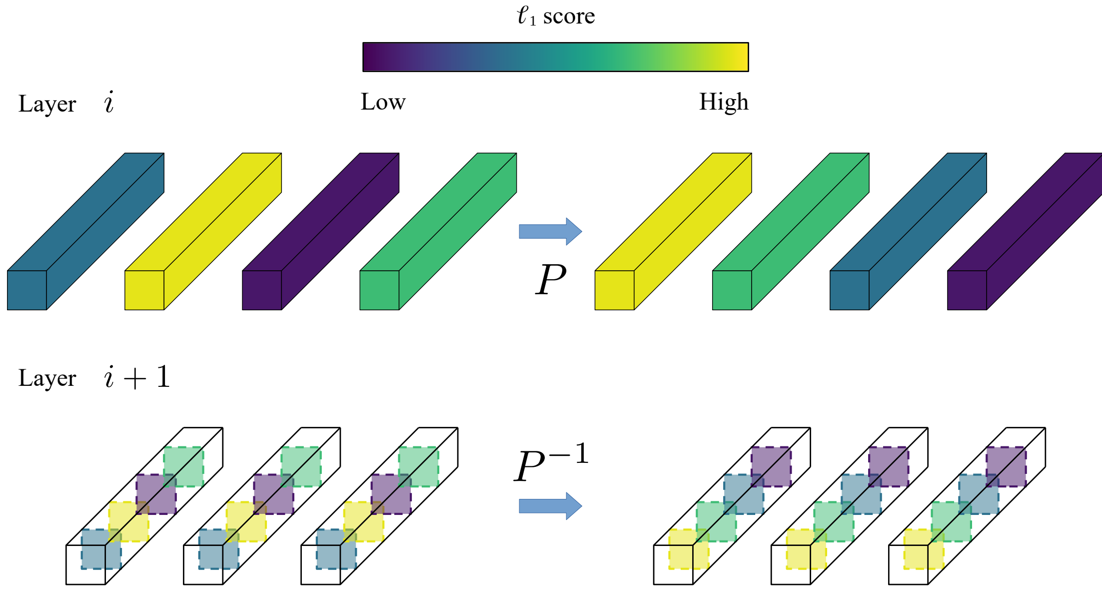

Figure 3.6: Example of ordering filters in a convolutional layer with ℓ_1-norm based ordering. To guarantee functional equivalence, the same permutation must be applied to the kernels inside each filter of the following layer.

of score that can capture the importance or saliency of weights. The problem of finding saliency of weights is typical of classic pruning techniques. Work by Li et al. [20] explored using the ℓ_1 norm to measure the importance of weights in convolutional layers. The importance of the i-th filter $W_i \in \mathbb{R}^{C_i \times k \times k}$ is computed as the sum of magnitudes of its kernels:

$$\|W_i\|_1 = \sum_{l=0}^{C_i} |\mathcal{K}_l|$$

The magnitude of kernel $\mathcal{K}_l \in \mathbb{R}^{k \times k}$ is the sum of magnitudes of its weights: $|\mathcal{K}_l| = \sum_{m,n} \left|\mathcal{K}_{l,m,n}\right|$. The reasoning behind this scoring approach is that filters with smaller kernel weights tend to generate feature maps with weaker activations than those corresponding to filters with larger magnitudes [20]. ℓ_1 scoring is data-free, meaning that no examples need to be run to the model in order for computing scores, and is also extremely computationally lightweight.

Given these definitions, the permutation P is determined by the scores computed on the first convolutional layer W^1. P reorders output filters W_i^1, $i = 1, \ldots, C_o$ such that the scores are monotonically non-increasing: $s((PW^1)_j) \leq s((PW^1)_k)$ for every $j < k$. Then the same permutation is

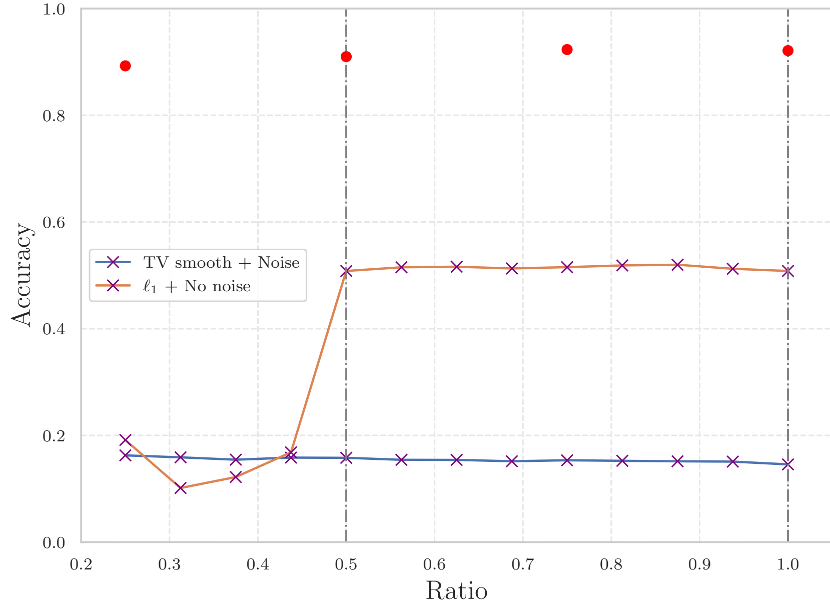

Figure 3.7: TV smoothing vs. ℓ_1 ordering for ground-truth conditioning in sliced reconstruction approach.

applied identical to input kernels of W^2. Essentially, the permutation is determined exclusively by parameters of the first layer.

We validated the score-based permutation approach by training the "W-B Hypernet" for 100 epochs *on the whole network* with two setups: the first following NeuMeta, using coordinate noise and TV smoothing, the second following our approach with ℓ_1 scoring permutation and without coordinate noise. Results are illustrated in Figure 3.7. Our setup achieved the new best result on full network reconstruction. Unfortunately, we also notice a sharp degradation in the capability of generalizing to untrained configurations. For this reason, we also ran an extended training of 300 epochs, hoping to see that train emerge again.

Thanks to the longer training, "WB Hypernet" reaches the best performance so far on *full* ResNet reconstruction as shown in Figure 3.8. Unfortunately, even after more training the hypernetwork still suffers a consistent degradation of performance for ratios below the lower training bound. Faced with the fact that still the generated weights performed consistently worse than the baselines we considered for this scenario, we explored a novel way of approaching the generative adaptation setup.

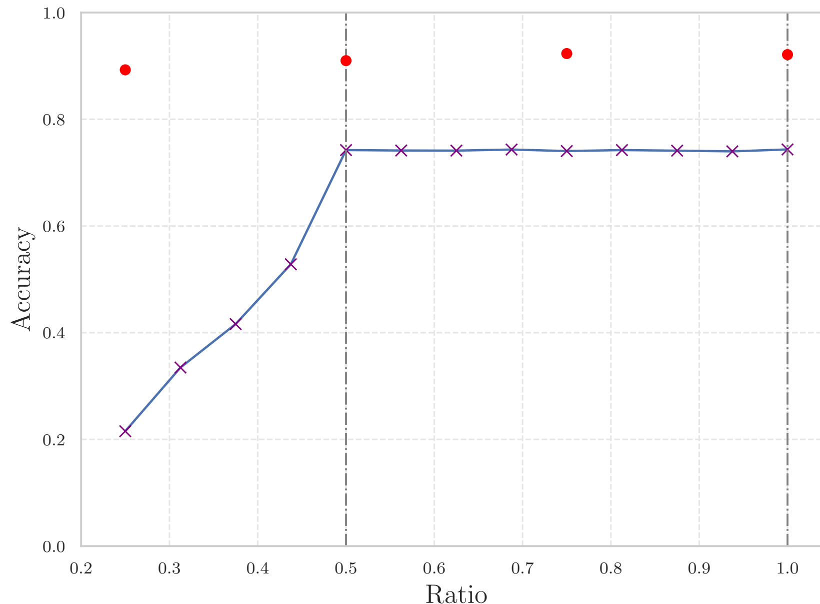

Figure 3.8: 300 epochs training on "W-B Hypernet" with ℓ_1 ordering, no coordinate noise, sliced reconstruction approach. Trained on $r = [0.5, 1.0]$.

3.3 Residual approach

Up until this point, we trained hypernetworks to generate all parameters of the target network starting from scratch. However, our experiments on full-scale ResNet20 models revealed that a single hypernetwork often struggles to learn performant parameters for the downstream task when its learning objective consists in estimating the structure of the pre-trained weight manifold. We observed that giving the hypernetwork a stronger conditioning on the ground truth weights, such as through sliced reconstruction loss or ℓ_1-based weight permutations, is empirically more beneficial than independent generation.

In this section, we explore a more extreme setting by posing the task as a *residual learning* problem. Instead of tasking the hypernetwork with modelling the high-dimensional weight manifold by combining the ground-truth network reconstruction and downstream loss objectives, we fix a point in the weight manifold and task the hypernetwork with producing a *change vector* in the weight space to obtain a metamorphic version of the network.

Formally, let θ_{pt} represent the parameters of the pre-trained base network that we want to

make metamorphic. While we initially treat this as a fixed point in the weight space, it is crucial to recognize that θ_{pt} actually identifies an equivalence class of parameters. As discussed in the previous section on weight permutations, neural networks exhibit symmetries where applying a permutation operator Π to the weights results in a functionally identical network $f(x;\theta_{\text{pt}}) = f(x;\Pi(\theta_{\text{pt}}))$. The operator Π thus represents a set of possible permutations, mapping θ_{pt} to a set of points on the weight manifold $\{\theta^i_{\text{pt}}\}_i$. While this set is populated with points that are distinct in parameter space, notice that the functional invariance guaranteed by Π means that the whole set lies on the same level curve in the loss landscape (by virtue of the network function producing the same exact results for each θ^i_{pt}).

This observation is critical for our residual formulation because the slicing operation $\mathcal{S}$ is *not* permutation invariant. Depending on the permutation applied, extracting the first k channels from a randomly ordered weight tensor will yield a different result than extracting the first k channels from a sorted one. Therefore, the "quality" of the anchor point for residual learning depends heavily on the specific permutation π chosen from the set of possible symmetries.

In a standard hypernetwork setup like NeuMeta [31], the weights θ_{meta} for a specific compression ratio $r \in [0, 1)$ are generated directly by the INR function F:

$$\theta_{\text{meta}}(r) = F(\mathbf{v}_{\mathcal{N}}(r)), \tag{3.1}$$

where $\mathbf{v}_{\mathcal{N}}$ are the coordinate embeddings determined for the network $\mathcal{N}$ as a function of ratio r. Recall that the dependency between r and the coordinate vectors follows the definition of coordinates provided by NeuMeta, that packs both intra-configuration coordinates and inter-configuration coordinates (the latter affected by r). Our residual formulation can be written as:

$$\theta_{\text{meta}}(r) = \mathcal{S}(\pi^*(\theta_{\text{pt}}), r) + \Delta\theta_r(\mathbf{v}_{\mathcal{N}}(r)). \tag{3.2}$$

In Equation (3.2), we introduce distinct components governing the transformation:

- π^* represents the function-preserving permutation, applied to pre-trained weights. This permutation can be selected following diverse criteria and scoring of parameters, such as the ℓ_1 saliency scores. This ensures that the weights are aligned such that indices $i = 1, \ldots, r \cdot C$ contain the most "important" features for preserving the original network's behaviour, effectively creating a "nested" structure within the weights.

- $\mathcal{S}(\cdot, r)$ represents the selection or "slicing" function. Given the target ratio r, this function extracts the relevant subset of the permuted pre-trained parameters (e.g., the top-$(r \cdot C_{out})$ sorted filters). Unlike randomly slicing θ_{pt}, slicing $\pi^*(\theta_{\text{pt}})$ guarantees that the static component retains maximum functional integrity.
- $\Delta\theta_r$ is the residual change vector generated by the Implicit Neural Representation (INR): $\Delta\theta_r = F(\mathbf{v}_{\mathcal{N}}(r))$. It takes the coordinates of the *permuted* parameters as input. Its output is treated as an additive shift rather than the parameter value itself.

By defining the problem this way, the hypernetwork is relieved of the burden of "re-learning" the fundamental features already captured during pre-training. Furthermore, by anchoring to the *sorted* permutation $\pi^*(\theta_{\text{pt}})$, we ensure the residual $\Delta\theta$ only needs to learn the *fine-tuning delta* required to adapt the most salient pre-trained weights to the capacity constraints of the narrower width r, rather than compensating for the loss of critical features that might otherwise be pruned by an arbitrary slicing order. Notice how, following the fact that parameters coordinates depend on the compression ratio, the residual component is in itself dependent on the r (highlighted by the subscript notation $\Delta\theta_r$).

3.3.1 Motivation

As noted in our ResNet experiments, generating weights from scratch becomes a computational burden rather than a compression benefit. By employing a residual formulation, the magnitude of the function learned by the INR is significantly reduced. The INR is no longer required to memorize the global structure of the filters, but only the perturbations necessary to compensate for the degradation of performance induced by the loss of parameters after slicing.

While a single INR can successfully reconstruct a full ResNet20 in simpler frameworks like NeRN [2], the metamorphic task is strictly harder because the weights must remain functional across *multiple* configurations. Anchoring to θ_{pt} provides a strong inductive bias that the metamorphic weights should remain within the neighbourhood of a known local minimum in the loss landscape. This effectively initializes the optimization of the metamorphic network at a high-performing point (the pre-trained model) rather than requiring the hypernetwork itself to fully learn to replicate it.

Both Neural Metamorphosis and NeRN identify the phenomenon of "spectral bias", the tendency

to learn low-frequency components of a function quickly while struggling with high-frequency variations [28, 23] observed in neural networks (in this case, the MLPs that model the INR), as a key aspect that must be managed when training hypernetworks to generate parameters of common neural networks. In general, pre-trained weights of deep neural networks do not present any bias towards smooth, continuous-like distribution of values. CNNs for example learn filters that contain many diverse patterns, but there is no guarantee that kernels are "arranged" smoothly inside the same filter or across different filters in the same layer. Kernels embed significant high-frequency spatial information essential for feature extraction. In the generation approach followed so far, the INR must synthesize these high-frequency components from scratch. In the residual approach, θ_{pt} already supplies the necessary high-frequency structure. By reformulating the generation problem as a residual estimation, the INR's role is reduced to modulating these existing features, which likely requires lower-frequency adjustments over the weight coordinate space, aligning better with the inductive biases of the hypernetwork.

To rigorously ground the residual generation approach, we explicitly articulate the two core hypotheses behind this idea:

- **Existence of a Compressible Permutation.** We assume that there exists a permutation π^* such that the semantic information within the pre-trained weights θ_{pt} is ordered hierarchically. Under this ordering, the slicing operation $\mathcal{S}(\cdot, r)$ acts as an effective projection that retains the maximally informative subset of parameters, minimizing the initial approximation error before any adaptation. This premise aligns with established findings in network pruning, where magnitude-based criteria (e.g., ℓ_1-norm) successfully identify salient filters that preserve network function when isolated [20, 18].

- **Reduced Spectral Complexity of Residuals.** We posit that the function mapping coordinates to the residual vector $\Delta\theta$ is spectrally simpler to learn than the mapping to the full weights θ. As established in [28], standard MLPs (and by extension, INRs) suffer from spectral bias, struggling to approximate high-frequency functions. While the base weights θ_{pt} essentially contain high-frequency spatial features (such as sharp edges and textures in convolutional kernels), the residual $\Delta\theta$ represents a *fine-tuning* signal. We hypothesize that this signal, which modulates existing features rather than synthesizing them, exhibits a lower-frequency

profile or higher sparsity, thereby posing a significantly more tractable optimization objective for the INR.

3.3.2 Implementation

The implementation of the residual generation approach builds on top of the slicing operation introduced for sliced reconstruction. Instead of employing the sliced weights only as ground truth for computing the reconstruction loss, the residual $\Delta\theta$ is added directly to these weights. Consider a standard convolutional layer with weight tensor $W \in \mathbb{R}^{C_{out} \times C_{in} \times K \times K}$. For a target width ratio r, let $c'_{out} = r \cdot C_{out}$.

The slicing function $\mathcal{S}$ returns the sub-tensor corresponding to the kept channels. The INR generates a delta tensor $\Delta W_r \in \mathbb{R}^{c'_{out} \times C_{in} \times K \times K}$ (assuming input channels are not sliced for this specific layer, or sliced accordingly if they are). The final effective weights W_{meta} used for the forward pass are:

$$W_{\text{meta}} = W[0 : c'_{out}, :, :, :] + \text{INR}(\mathbf{v}_r). \tag{3.3}$$

This formulation ensures that at $r = 1.0$, the hypernetwork can theoretically learn to produce $\Delta\theta = \mathbf{0}$ to perfectly recover the original pre-trained performance. This provides an upper bound on performance that was previously difficult to approach with the standard generation method.

3.3.3 Experiments

Given the novel formulation of the target task, we opted to reformulate the loss function:

$$\mathcal{L} = \mathcal{L}_{\text{task}} + \lambda \mathcal{L}_{\text{reg}}.$$

We set $\lambda = 0.01$ based on previous experiments outlined in Section 3.1.1, but we did not run extensive hyperparameter tuning after noticing that slightly different values did not provide any meaningful benefit.

The experiments in this sections follow a similar setup from the previous sections. First of all, we elected to utilize the "W-B Hypernet" since it empirically demonstrated to be the most competitive setup on the initial experiments. The main difference is in the weight initialization prior to training: we initialize the final output layer of both INRs with zeros. During the first forward pass, the

hypernetwork outputs $\Delta W = \mathbf{0}$, thus having no effect on the original weights. All gradients of the first backward pass update only the last layer, leaving the rest of the network unchanged. This choice aims to train the hypernetwork as a slow "adaption" from the original weights.

A direct consequence of this initialization choice is that larger values of r have lower loss values right from the beginning of training, in light of the larger amount of pre-trained parameters active during inference. At the same time, it is clear that there is less need of training at large ratios, since performance degradation decreases naturally. The effect of training large ratios is already reduced as a natural consequence of lower loss. With lower loss, the gradients propagated during training will cause less change than, for example, smaller ratios that require more pronounced optimization. This simple effect acts as a implicit regularizer on the training. We decided to limit our training interval ratios as to never reach $r = 1.0$: we set the upper bound of the ratio sampling interval to $r = 0.95$. Additionally, we adapted the learning rate for this setting to 0.0001, while retaining the cosine scheduling, to ensure the adaptation of weights is not too disruptive. We expect that for most configurations, the weights modulated with residuals will be closer to some kind of minimum than when trying to generate them from scratch.

The training recipe we employed here is based on the one detailed in previous sections. We only adapted the learning rate by lowering it tenfold to 0.0001, in order to avoid weight updates being too abrupt.

Results with L1-score based ordering. The initial results with ℓ_1 ordering (pictured in Figure 3.9) indicate a notable sensitivity to channel reduction, suggesting potential limitations in the current adaptation strategy. Despite extending the hypernetwork's training to 0.25 following what we observed in our previous training runs, the hypernetwork still exhibits, even if less abrupt, a steep and continuous downward trend in accuracy toward lower values of slicing ratio. The widening gap between the adapted curve and the static baselines suggests a hypothesis that weight adaptation alone may be insufficient to compensate for a suboptimal structural foundation as the network narrows.

3.3.4 Data-driven permutation via ThiNet.

To overcome the limitations of simple magnitude-based scoring such as the ℓ_1 norm, we adopted the data-driven approach proposed in ThiNet [22]. While ThiNet was originally formulated as a

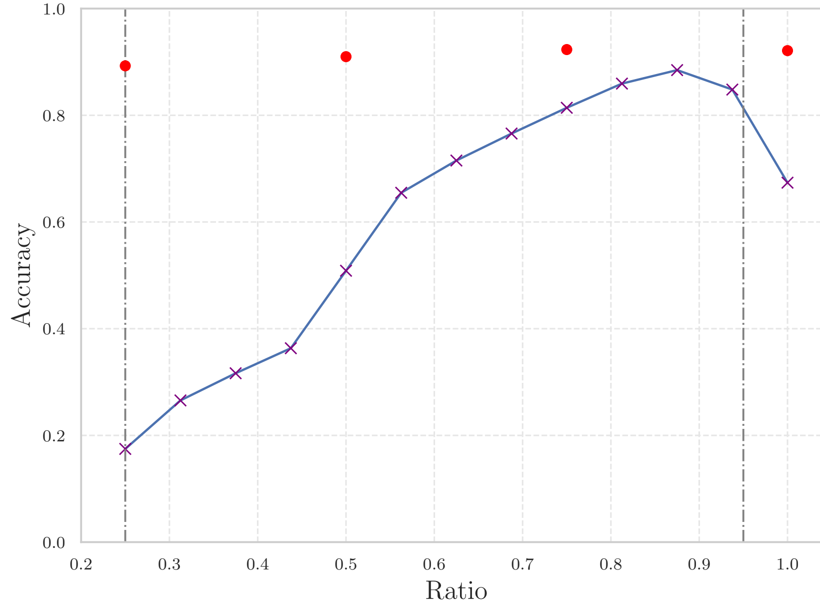

Figure 3.9: Accuracy on CIFAR-10 for ResNet20 with ℓ_1 ordering and residual weight adaptation.

filter-level pruning method to compress and accelerate networks, its core evaluation metric can be seamlessly repurposed to induce a principled ordering on the intermediate feature maps.

The fundamental insight of ThiNet is that the importance of a filter in layer i should not be evaluated based on its own intrinsic statistics (such as weight magnitude), but rather on how much its output contributes to the activations of the *subsequent* layer $i+1$. If a subset of channels in the input of layer $i+1$ can accurately approximate the output of layer $i+1$, the remaining channels can be considered redundant. Since each input channel of layer $i+1$ is uniquely produced by a specific filter in layer i, evaluating the importance of layer $i+1$'s input channels directly evaluates the filters of layer i.

Formally, let us consider the convolution process in layer $i+1$. We denote the input tensor to this layer as $\mathcal{I}_{i+1} \in \mathbb{R}^{C \times H \times W}$, which is generated by the C filters of layer i. The filters of layer $i+1$ are denoted as $\mathcal{W}_{i+1} \in \mathbb{R}^{D \times C \times K \times K}$, which produce an output tensor $\mathcal{I}_{i+2}$.

To determine the functional importance of each channel, we collect a set of training examples. A scalar value y is randomly sampled from the output tensor $\mathcal{I}_{i+2}$ (before the application of the ReLU activation). Based on the spatial location of y, we can identify the corresponding sliding window $x \in \mathbb{R}^{C \times K \times K}$ from the input $\mathcal{I}_{i+1}$ (after ReLU) and the specific filter $\hat{\mathcal{W}} \in \mathbb{R}^{C \times K \times K}$ from

$\mathcal{W}_{i+1}$ that produced y. The convolution operation yielding y is defined as:

$$y = \sum_{c=1}^{C} \sum_{k_1=1}^{K} \sum_{k_2=1}^{K} \hat{\mathcal{W}}_{c,k_1,k_2} \times x_{c,k_1,k_2} + b.$$

By defining the channel-wise convolution result as $\hat{x}_c = \sum_{k_1=1}^{K} \sum_{k_2=1}^{K} \hat{\mathcal{W}}_{c,k_1,k_2} \times x_{c,k_1,k_2}$, we can simplify the relationship to $\hat{y} = \sum_{c=1}^{C} \hat{x}_c$, where $\hat{y} = y - b$.

Because the variables $\hat{x}_c$ are independent across channels, removing a filter in layer i is equivalent to removing the corresponding $\hat{x}_c$ in this summation. Given m sampled training examples (spanning various images and spatial locations), ThiNet formulates channel selection as minimizing the reconstruction error of the output $\hat{y}$. If T represents the subset of channels to be removed, the optimization objective is to minimize the impact of their removal:

$$\arg\min_{T} \sum_{i=1}^{m} \left(\sum_{j \in T} \hat{x}_{i,j} \right)^2 \quad \text{s.t.} \quad |T| = C \times (1 - r),$$

where r is the compression rate.

Because finding the optimal subset T is an NP-hard problem, ThiNet employs a greedy strategy. It initializes $T = \emptyset$ and iteratively adds the channel c that results in the smallest increase to the objective function, effectively identifying the least important channels first.

Adapting ThiNet for Permutation. While ThiNet halts this greedy algorithm once the target sparsity $1 - r$ is reached, our residual hypernetwork approach requires a full ordering of the channels to define the permutation π^* introduced in Section 3.3.

To construct π^*, we execute the ThiNet greedy selection algorithm until all C channels have been added to T. The sequence in which channels are appended to T naturally defines an inverse ranking of their functional importance:

1. **Initialization:** Set $T \leftarrow \emptyset$ and the pool of available channels $I \leftarrow \{1, \ldots, C\}$.

2. **Greedy Selection:** For each step k from 1 to C, find the channel $c^* \in I$ that minimizes the squared sum of the activations for the set $T \cup \{c^*\}$.

3. **Update:** Append c^* to T and remove it from I.

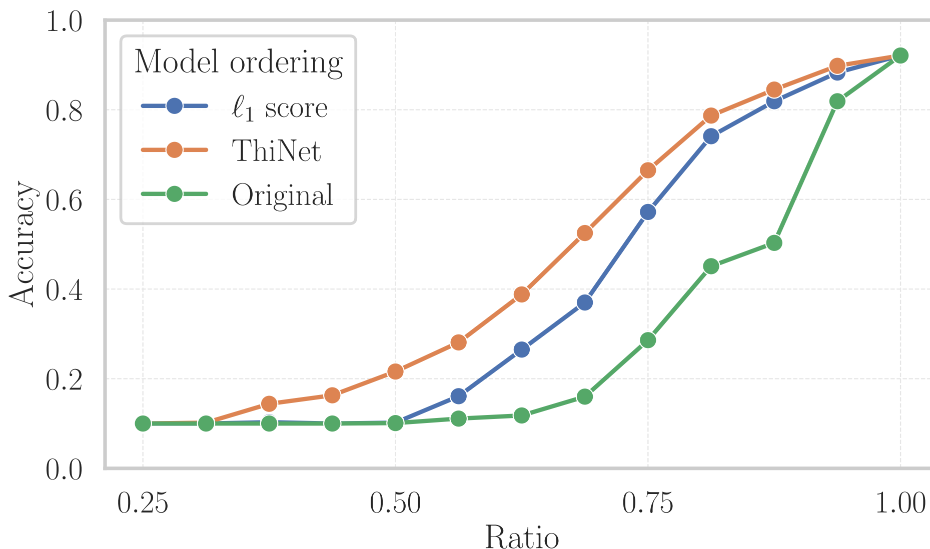

Figure 3.10: Accuracy of ResNet20 on CIFAR-10 under different filter ordering methods. The plot compares the robustness of ThiNet-based, ℓ_1-based and no ordering when slicing the network without retraining. As the ratio of retained channels decreases, ThiNet consistently maintains higher accuracy compared to the other heuristics, demonstrating its effectiveness in mitigating degradation from channel loss.

The ordered set $T = (c_1, c_2, \ldots, c_C)$ represents channels from the least important to the most important. Reversing this sequence yields the permutation matrix π^*, which maps the pre-trained weights to a state where the most functionally critical features occupy the lowest indices. By anchoring our hypernetwork generation to this sorted configuration, the slicing operation $\mathcal{S}(\pi^*(\theta_{\text{pt}}), r)$ effectively acts as a dynamic ThiNet pruner, dynamically retaining the optimal subset of channels for any given continuous ratio r.

Figure 3.10 compares the performance of a sliced ResNet20 when its weights are first reordered following different criteria, without then undergoing any adaptation (all its parameters retain their original value). Ordering with ThiNet exhibits the highest resilience to slicing; ℓ_1-score based permutation is better than leaving the network as-is, but its performance degrades faster than the ThiNet counterpart, falling to the random-guessing floor already at moderate slicing ratios ($r = 0.5$).

Thinet Experiments. Without changing the training setup from the previous experiments, we trained the hypernetwork on three different intervals of width slicing ratios: $r \in [0.25, 0.95]$, $r \in [0.5, 0.95]$ and $r \in [0.25, 0.75]$. The calibration set for computing the ordering is collected following the observations in [22]: we select 10 examples per class from the training set, and for each image we sample 10 instances across both channels and spatial dimensions of the corresponding output feature maps. We did not experiment with changing these hyperparameters. Since ThiNet relies on sampling for estimating the saliency of weights, we run experiments with multiple initialization seeds $\{42, 2001, 2026, 1\}$ and present their aggregated results.

Results are plotted in Figure 3.11 and Figure 3.12. A primary observation across all three configurations is that the hypernetwork still presented strong specialization to the ratios of the training interval. When evaluated outside these bounds the model exhibits severe out-of-distribution performance degradation and increased variance. This confirms that the hypernetwork relies heavily on the specific range of structural configurations seen during training and struggles to extrapolate to unseen widths.

However, when examining the widest training interval in Figure 3.11 (right plot), the benefits of the ThiNet ordering become clear. Compared to the previous experiment with ℓ_1-score based permutations, the downward trend in accuracy at lower ratios is significantly less severe. At the $r = 0.25$ lower bound, the ThiNet-ordered model maintains an accuracy near 0.45, a marked improvement over the 0.2 observed with the ℓ_1 score. This smoother degradation curve supports the earlier hypothesis: by prioritizing more critical features during the initial ordering phase, ThiNet establishes a fundamentally stronger structural baseline. Consequently, the hypernetwork is no longer forced to compensate for a sub-optimal set of weights at lower dimensions, allowing it to more effectively adapt the weights and preserve performance. Nonetheless, the performance at lower values of r is far from the baselines' accuracy, highlighting how producing a good adaptation is still a challenge.

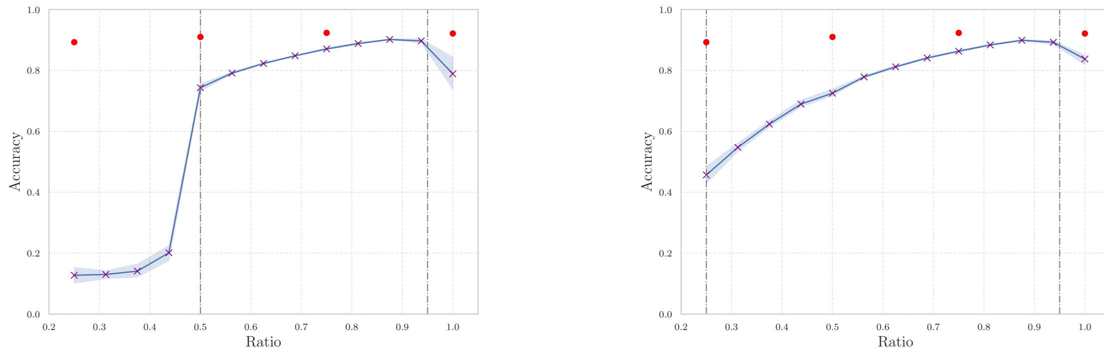

Figure 3.11: Test accuracy on CIFAR-10 of ResNet20, with channels permuted via ThiNet, adapted through residual weight generation. **Left**: training on interval ratio $r \in [0.5, 0.95]$. **Right**: training on interval ratio $r \in [0.25, 0.95]$.

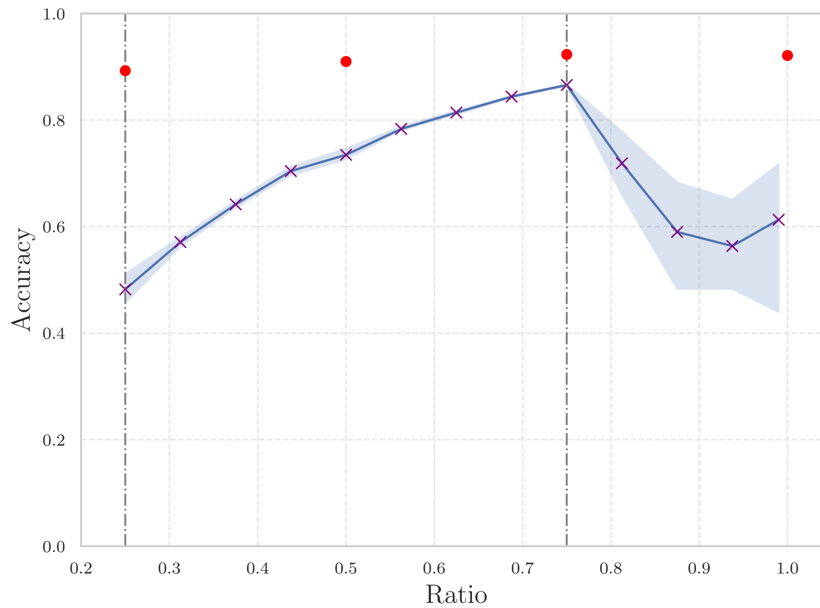

Figure 3.12: When evaluated on residual adaptation of width ratios unseen during training, ResNet20 accuracy drops sensibly even if the hypernetwork was not trained on larger ratios, which should be easier to handle given the limited accuracy degradation caused by slicing.

Algorithm 3 ThiNet-based Greedy Channel Ordering for Permutation π^*

Require: Training set of spatial activations $\{(\hat{x}_i, \hat{y}_i)\}_{i=1}^m$, Total number of channels C

Ensure: Permutation matrix π^*

1: $T \leftarrow \emptyset$ ▹ Ordered list of channels (from least to most important)

2: $I \leftarrow \{1, 2, \ldots, C\}$ ▹ Pool of available channel indices

3: **while** $|T| < C$ **do**

4: $E_{\text{min}} \leftarrow +\infty$

5: $c^* \leftarrow$ null

6: **for all** $c \in I$ **do**

7: $T_{\text{temp}} \leftarrow T \cup \{c\}$

8: $E \leftarrow \sum_{i=1}^{m} \left(\sum_{j \in T_{\text{temp}}} \hat{x}_{i,j} \right)^2$ ▹ Evaluate reconstruction error

9: **if** $E < E_{\text{min}}$ **then**

10: $E_{\text{min}} \leftarrow E$

11: $c^* \leftarrow c$

12: **end if**

13: **end for**

14: Append c^* to the end of T

15: $I \leftarrow I \setminus \{c^*\}$

16: **end while**

17: $T_{\text{rev}} \leftarrow \text{Reverse}(T)$ ▹ Reorder to prioritize the most important channels

18: $\pi^* \leftarrow \text{ConstructPermutationMatrix}(T_{\text{rev}})$

19: **return** π^*

4. Slimmable Fine-Tuning of Pre-Trained Networks

In the previous chapter, we explored adapting pre-trained models using hypernetworks. We formulated the task as a residual learning problem and used data-driven permutations, such as ThiNet, to improve the network's structural resilience at lower capacities. However, the hypernetwork still struggled to produce performant adaptation residual when evaluated on more aggressive slicing ratios and also outside its training interval. Relying on a secondary network to generate weights also introduces significant optimization complexity, creating memory and computational bottlenecks when scaling to full architectures.

These limitations prompted us to experiment with a more direct approach. Rather than training an additional network to map virtual coordinates to weight adaptations, we thought of fine-tuning the pre-trained weights directly, with the same end goal of obtaining a network that supports runtime adjustment of its performance-accuracy profile. We turned to the framework of Slimmable Neural Networks (more precisely, its "Universal" formulation outlined in Section 2.2.3). While standard Slimmable networks are trained entirely from scratch, our goal is to instead "induce" their distinctive nested weight structure into a network previously trained without such bias, while still retaining as much of its original functionality as possible. In a ideal scenario, the post-training adaptation should produce a network as least as "good" on the downstream task as its Slimmable counterpart trained from scratch.

This shift in perspective introduces several challenges. First, determining if it is genuinely feasible to take a standard, non-slimmable pre-trained network and make it slimmable through fine-tuning. A key aspect of this is understanding exactly how easily the pre-trained network can adapt and to the novel structure imposed on its weights. We aim to observe whether slimmable fine-tuning acts as a lightweight adaptation of existing features, or if enforcing nested constraints is so disruptive that the process essentially becomes a full retraining from scratch.

To help smooth this transition, we also revisit the permutation techniques introduced in Chapter 3. We investigate what effects these sorting strategies might have when applied in this new context. By organizing the pre-trained channels using magnitude-based or data-driven scoring before fine-tuning, we explore whether these permutations can provide a beneficial structural prior, giving the

network a more favourable starting point to learn the nested slimmable structures.

Finally, we consider the overall efficiency of the adaptation process. Since the base network already possesses strong feature extraction capabilities, we question whether it is strictly necessary to fine-tune every single parameter. We explore opportunities to "compress" the training phase by investigating whether parameter-efficient fine-tuning techniques can be used. By training fewer parameters than those present in the main network, we hope to achieve the desired adaptation while minimizing the computational cost of the transition.

4.1 Slimmable adaptation with CNNs

To ensure a rigorous comparison between generative adaptation and direct slimmable fine-tuning, our initial experiments utilize the exact controlled environment established in Chapter 3. By retaining the pre-trained ResNet20 architecture evaluated on the CIFAR-10 dataset, we eliminate architectural and dataset variables. This continuity allows us to cleanly isolate the impact of the new fine-tuning paradigm and measure its success directly against the performance of our previous residual hypernetwork models.

A major finding from our prior experiments was the severe structural degradation that occurs when slicing a convolutional layer without reorganizing its weights. To counteract this, we applied magnitude-based ℓ_1 score and data-driven ThiNet permutations, which successfully shifted the most salient feature maps to the lowest channel indices. Because Slimmable Networks inherently depend on this nested, hierarchical channel structure to function at reduced capacities, these initial experiments test whether those same permutation techniques can serve as an effective structural prior. We hypothesize that initializing the pre-trained network with sorted weight matrices can provide a more stable starting point for slimmable fine-tuning.

Baselines. We prepared a Slimmable baseline to provide a grounded comparison with our method. Our target model is the same ResNet20 architecture of Figure 2.1 on CIFAR-10, trained with the Universally Slimmable framework. Following the the original recipe of the public ResNet model, we employed the SGD optimizer for 200 epochs, with batch size 256 on a single GPU. The initial learning rate is set at 0.1 and is decayed to zero with a cosine decay schedule. A momentum factor of

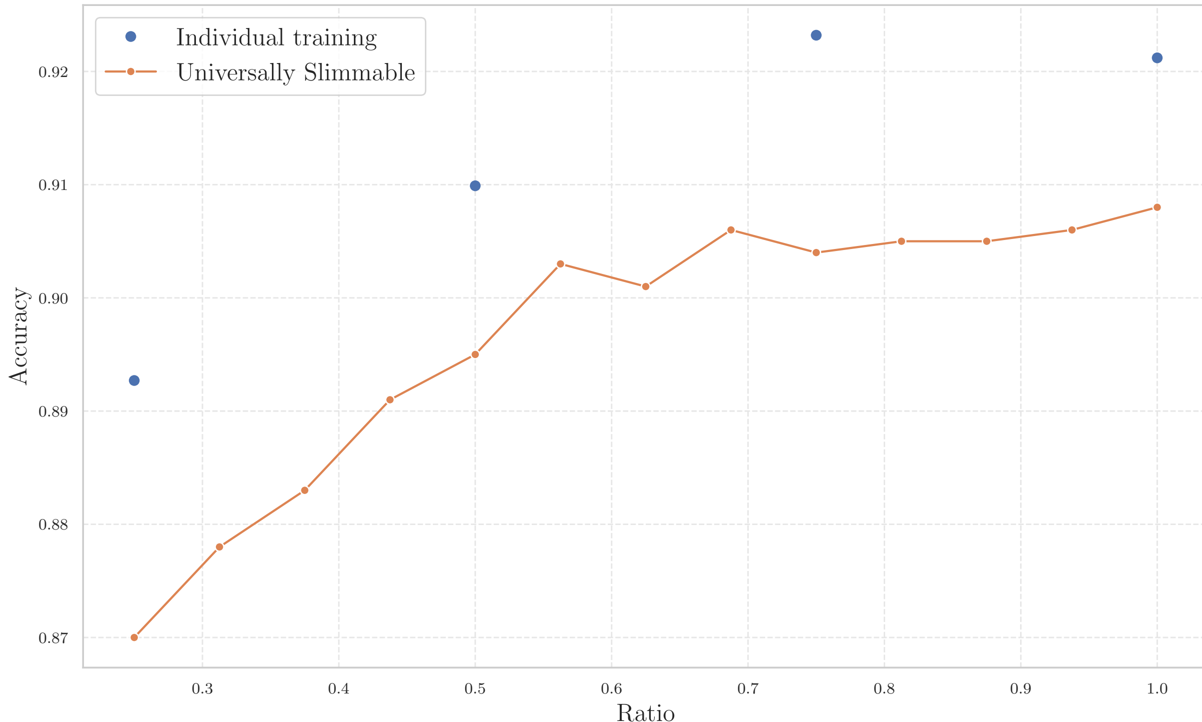

Figure 4.1: Test accuracy of Universally Slimmable ResNet20 trained on CIFAR-10

0.9 is used in gradient updates; additionally, Nesterov optimization is applied. Data augmentations consist in resizing to 36×36 with zero padding, then randomly cropping a 32×32 patch; random horizontal flip is applied before finally normalizing the image.

We then adopted the setup outlined in the paper of US-Nets [32] for Slimmable training. The number of samples to employ for the sandwich rule is set to $n = 4$. The range of width ratios is $r \in [0.25, 1.0]$ with two intermediate widths sampled in between. We enable in-place distillation as per the original method. Regarding calibration of Batch Normalization statistics, the calibration set is composed of 5 example batches extracted from the training set, corresponding to 1280 unique training images at batch size 256.

The result is presented in Figure 4.1. The performance we obtained lies slightly below the individually trained resized configurations. This is a possible indication that the training recipe adopted is actually not optimal. Nonetheless, we deemed it sufficient to provide a sensible approximation of the performance of a Slimmable model; thus we did not run additional tests to tune hyperparameters.

4.1.1 Initial experiments with direct fine-tuning

We began our experiments by running a 200 epoch fine-tuning on the setups with ordering and also on the pre-trained network itself. Notice that the training is deliberately made longer than what is realistically necessary (since it matches the *original* training length). The main goal is to explore how fast adaptation can happen under various ordering configurations: to do that, we decided to simply fine-tune the models and to measure their performance on the test set as the training proceeds. Another important thing to note is that calibration is repeated at every evaluation, across all width ratios chosen for testing. In this case, to get a comprehensive view of the performance on the slimmable resizing range $r \in [0.25, 1.0]$ used for training, we tested sub-networks corresponding to $r_t = \{0.25, 0.5, 0.75, 1.0\}$.

For this specific experiment, we trained with the AdamW optimizer instead of SGD originally used during pre-training. Initial learning rate was set at 0.001, scheduled with cosine decay. All other training details are equal to the baseline. Since our "bottleneck" resizing strategy only affects the number of intermediate feature maps in each block (referring to Figure 3.1), in principle the minimum set of weights that need to be adapted are obviously those belonging to the two convolutional layers and those belonging to the enclosed Batch Normalization layer. Given this observation, we decided to keep all other weights frozen during finetuning. This includes the statistics of the batch normalization which follows the last convolutional layer. By imposing these constraints, we wanted to observe if the network can maintain similar intermediate representation capabilities to those produced with its original configuration.

Results. Results are presented in Table 4.1. Looking at the evolution of test accuracy, it appears that incorporating a structured ordering strategy can facilitate adaptation compared to fine-tuning without any ordering. Across most of the epoch-ratio configurations showed in the table, both ordered methods achieve slightly lower error rates.

In particular, during the early-intermediate stage of the training run, ThiNet ordering shows a slight advantage in achieved accuracy over both the other two setups. After just one epoch, the ℓ_1 ordering setup instead has a slight advantage over the rest, across most width ratios. Nonetheless, we must notice that the results do not appear to show strong or remarkable trends: in the end, all configurations are able to undergo adaptation without any major disruption of representational

Table 4.1: Error rates by Epoch and Ratio for different ordering methods

Epoch	Ratio	ℓ_1-order	No-order	ThiNet-order
1	1.00	**88.95**	86.75	88.68
	0.75	**86.98**	84.32	86.32
	0.50	**81.89**	79.24	79.78
	0.25	71.84	71.03	**73.44**
10	1.00	**89.54**	89.48	88.81
	0.75	88.50	**88.85**	88.08
	0.50	86.42	**86.89**	85.35
	0.25	**83.31**	83.08	82.14
20	1.00	89.77	89.58	**89.97**
	0.75	88.96	88.52	**89.38**
	0.50	**87.70**	87.45	87.69
	0.25	85.05	**85.45**	84.59
50	1.00	90.32	90.76	**91.09**
	0.75	89.55	89.89	**90.19**
	0.50	88.82	88.76	**88.92**
	0.25	86.30	86.28	**86.81**
100	1.00	91.12	91.52	**91.59**
	0.75	90.49	90.85	**91.26**
	0.50	89.60	89.89	**90.07**
	0.25	**88.02**	87.99	87.99
200	1.00	91.77	91.60	**91.95**
	0.75	91.35	**91.54**	91.47
	0.50	**90.71**	90.53	90.66
	0.25	**88.84**	88.80	88.55

capacity. Another fact to notice is that the performance of the full network (at $r = 1.0$) is lower that the pre-trained baseline, recovering to close or slightly above that accuracy target around the 100-epochs mark. Realistically, this effect is a consequence of the "interference" between the

numerous nested sub-networks being optimized at the same time; each sub-network needs to be performant in isolation, but also it must work in unison as part of enclosing networks of the nested slimmable structure. This kind of "global" optimization seems to converge slower that what single sub-networks can do in isolation.

Beside the apparent diminishing returns of weight permutations in this context, arguably the most interesting fact is that, after only one epoch of training, even low width ratios have already regained much of the lost performance: comparing with Figure 3.10, for instance the $r = 0.25$ sub-network goes from about 10% (random guessing level for CIFAR-10) to more than 70% accuracy, across every ordering configuration. Observing the result obtained by the baselines in Figure 4.1, this translates to recovering around 85% of the accuracy of the from-scratch slimmable model and 80% of the individually trained baseline, both at the same 0.25× slicing ratio. By epoch 50 (just a quarter of the baselines' training length), the performance approaches the slimmable baseline. This result suggests that, with a moderately-sized network and relatively simple dataset, direct slimmable fine-tuning can recover most of the original performance, even with little computational cost.

4.1.2 Reducing the update size via Low-Rank Adaptation (LoRA)

After observing the efficacy of direct fine-tuning of all convolutional parameters, we decided to explore a different question: whether it is possible to *reduce* the number of parameters updated during adaptation. The end goal of updating less parameters than those of the full network is that of directly reducing the fine-tuning cost associated with adaptation; at the same time, the impact to the performance of the adapted network should be minimal (ideally none) in order to maximize the benefit of reducing training cost.

We chose to integrate into the training pipeline the work from Hu et al. Low-Rank Adaptation [14] (*LoRA*). LoRA achieves parameter reduction by factorizing weight matrices into two components with a small *inner rank*, whose product then represents an approximation of the original full-rank matrix. To better illustrate, once again consider a weight matrix $W_{pt} \in \mathbb{R}^{N \times M}$ belonging to the pre-trained network. During full fine-tuning, the optimization process applies updates to each component of the matrix, resulting in a final W_{ft}. We can isolate the changes by defining an update matrix ΔW:

$$W_{\text{ft}} = W_{\text{pt}} + \Delta W. \tag{4.1}$$

In a traditional setting, ΔW has the same dimensionality as W_{pt}. Both the computational cost and the storage cost of fine-tuning depend directly on the size of ΔW.

Recent work in the literature has explored the question of how many parameters are necessary for successfully training neural networks. [19] for instance is an interesting work that attempts to provide a measure of "dimensionality" of an training objective. What the results highlighted is that the dimensionality of the objective is usually smaller than the capacity of the network employed to solve it. Such an observation implies that most neural networks can be compressed without necessarily causing excessive degradation to their performance. The same reasoning can be applied also in the context of fine-tuning. Even more so, given that the network has already solved much of the problem, we usually expect tuning to be less difficult. Such concepts have been thus used to alleviate the burden of fine-tuning large neural networks.

Inspired by the observations on dimensionality of DNN parametrizations, LoRA's authors hypothesize that the change in weights from fine-tuning (ΔW) resides on a low-dimensional manifold. Therefore, it is unnecessary to optimize the update in the full parameter space. Instead, LoRA proposes decomposing the update matrix into the product of two lower-rank matrices, $B \in \mathbb{R}^{d_{out} \times \rho}$ and $A \in \mathbb{R}^{\rho \times d_{in}}$, where the rank ρ is a hyperparameter satisfying $\rho \ll \min(d_{in}, d_{out})$. The forward pass for a linear layer, originally $h = W_{\text{pt}}x$, is modified to:

$$h = W_{\text{pt}}x + \frac{\alpha}{\rho}(BA)x. \tag{4.2}$$

Here, α is a scaling constant used to stabilize training. During the adaptation process, W_{pt} remains frozen, and only A and B are optimized. A standard initialization strategy involves using a random Gaussian initialization for A and setting B to zero, ensuring that $\Delta W = 0$ at the beginning of training. This guarantees that the fine-tuning process starts exactly at the pre-trained baseline.

To illustrate the computational advantage of this method, consider a weight matrix with input and output dimensions $d_{in} = d_{out} = 4096$. A full-rank update would require training approximately 16.7 million parameters (4096^2). In contrast, applying LoRA with a rank of $\rho = 8$ requires learning only the parameters in A (8×4096) and B (4096×8). This results in approximately 65000 trainable parameters, reducing the memory requirement by approximately 250×.

After completing fine-tuning, the computed update result can be merged into the original pre-trained weights. Expanding the product AB produces a matrix with dimensions compatible with

the pre-trained counterpart: then the change can be simply added element-wise to the latter. As a result, LoRA introduces no additional inference latency to the original model.

Adapting LoRA to the Slimmable framework. Integrating LoRA into the Slimmable Network framework requires a specialized formulation because the dimensions of the weight matrices in a Slimmable Network are not static. A Slimmable "supernet" is designed to adjust its width dynamically as a function of width ratio $r \in (0, 1]$. Let C_{in} and C_{out} denote the maximum (full) input and output dimensions of a layer. For a given active width r, the network utilizes only a sub-tensor of the weights, defined by the active dimensions $c_{in} = rC_{in}$ and $c_{out} = rC_{out}$.

The challenge lies in defining an update ΔW that remains valid and consistent across these variable dimensions. We propose to slice the update matrices A and B without altering the bottleneck rank r across all configurations. We instantiate the full adapter matrices $B_{full} \in \mathbb{R}^{C_{out} \times \rho}$ and $A_{full} \in \mathbb{R}^{\rho \times C_{in}}$. Unlike the spatial or channel dimensions of the network, the rank ρ is a hyperparameter of the adapter itself and is treated as invariant to the network width.

During a forward pass at a specific width r, we slice the input dimension of A and the output dimension of B to match the active dimensions of the pre-trained weight matrix W^r_{pt}. Formally, the adapted forward pass is expressed as:

$$h = W_{\text{pt}}[0 : c_{out}, 0 : c_{in}]x + \frac{\alpha}{\rho} \left(B_{full}[0 : c_{out}, :] \cdot A_{full}[:, 0 : c_{in}] \right) x. \tag{4.3}$$

In this formulation, the slicing operator $[0 : k, :]$ extracts the first k rows, while $[:, 0 : k]$ extracts the first k columns. Figure 4.2 illustrates an example of such slicing scheme, where to slice the output dimension only B undergoes modulation (along with the pre-trained weight matrix).

For convolutional layers, typically represented by kernels of shape $C_{out} \times C_{in} \times k \times k$, we adhere to the standard LoRA implementation which often utilizes 1×1 convolutions (or reshapes the kernel to a 2D matrix). Our slicing logic is applied strictly to the channel dimensions corresponding to C_{out} and C_{in}, leaving the spatial kernel size and the adapter rank fixed. This allows the weights in the upper-left sub-matrices of A_{full} and B_{full} to be shared across all width configurations, enforcing consistency and enabling the fine-tuning of a single set of LoRA parameters that effectively service the entire slimmable supernet.

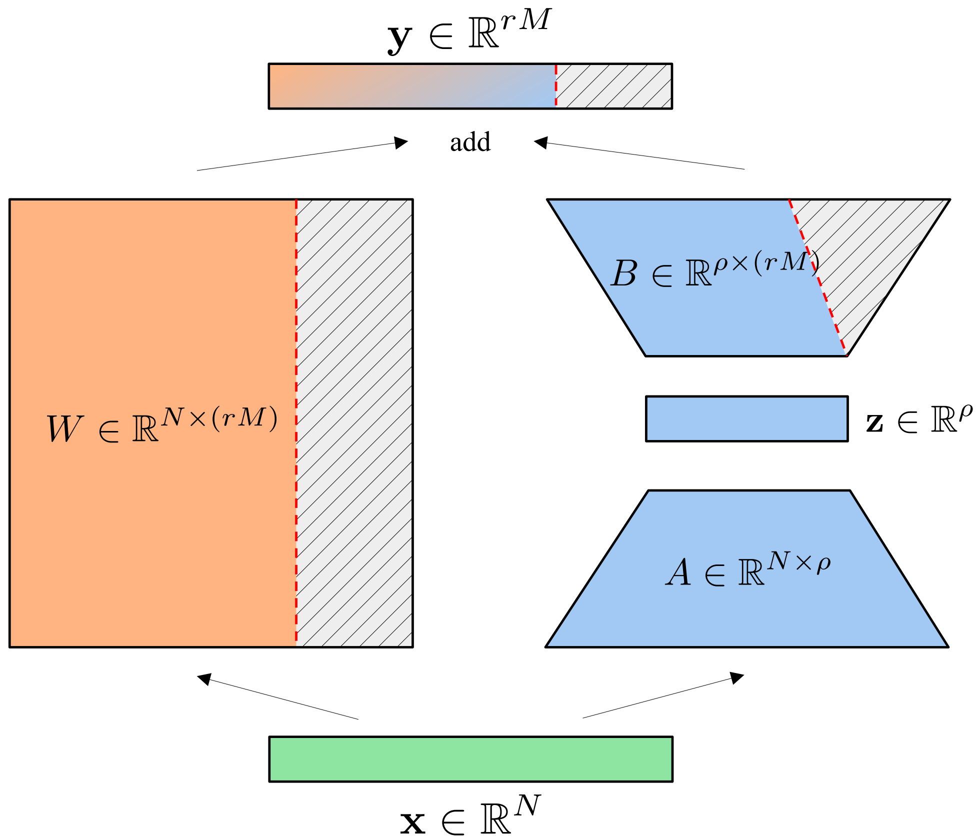

Figure 4.2: Example of LoRA update matrices with slicing applied. In this instance, only the output dimension is modulated by r

Impact of rank selection on adaptability. Based both on the theoretical and empirical advantage provided by the ThiNet ordering setup, we decided to exclusively run experiments with such configuration. Mainly motivated by the previously discussed results of Table 4.1, the training run was shortened to 30 epochs. Leaving the initial learning rate unchanged, the length of the decay schedule was shortened accordingly (we reckon there might be more optimal choices of hyperparameters, but for reducing the computational burden of running numerous tuning experiments, we did not further explore that aspect). We ran experiments selecting a set of ranks $\rho \in \{2, 4, 8, 12, 16, 32, 64\}$. Once selected, the same rank is applied uniformly to all convolutional weight matrices across the network. Table 4.2 compares the number of parameters of LoRA versus the original size of layers. Notice that uniformly selecting the same rank across all convolutional layers, regardless of their

initial parameter size, results in Layer1 and Layer2 with *more* parameters than original for large values of $\rho \in \{32, 64\}$. Despite this, all ranks chosen provide some degree of compression when considering the total number of parameters of the three layers.

Table 4.2: Parameter count breakdown and comparison: original full network versus compressed LoRA layers for ResNet20. LoRA entries marked with † are those with more parameters than the original network.

Rank	**Layer 1**	**Layer 2**	**Layer 3**	**% of total (# params.)**
Original	13,824	50,688	202,752	100% (267,264)
2	1,152	2,208	4,416	2.91% (7,776)
4	2,304	4,416	8,832	5.82% (15,552)
8	4,608	8,832	17,664	11.64% (31,104)
12	6,912	13,248	26,496	17.46% (46,656)
16	9,216	17,664	35,328	23.28% (62,208)
32	18,432†	35,328	70,656	46.55% (124,416)
64	36,864†	70,656†	141,312	93.10% (248,832)

Test accuracy measured after each epoch, at ratios $r \in \{0.25, 0.5, 0.75, 1.0\}$, achieved with ranks previously reported is illustrated in Figure 4.3. The results clearly display the influence of rank choice on task performance: increasing the rank size grants better adaptation capabilities, especially at more aggressive slicing ratios (0.25 and 0.5). Already at $\rho = 16$, which yields just 23.28% of the original update size, the achieved accuracy after 30 epochs across all ratios is reasonably close to the US baseline, which is represented in each chart by the red dashed line. With $\rho = 32$ and double the number of parameters of $\rho = 16$, accuracy rises even closer to the baseline. Unfortunately, concurrently with better accuracy, diminishing returns arise as rank size increases. We notice additionally that slicing at $r = 0.5$ appears to be the hardest width ratio to adapt in this particular case. For the full size ratio, we do not observe any particular degradation of performance.

Once again, we stress that the extent of these results might be limited by the characteristics of the dataset and size of pre-trained network chosen. In this context we do not further explore the scaling behaviour of our setup, which could be an interesting avenue for future work.

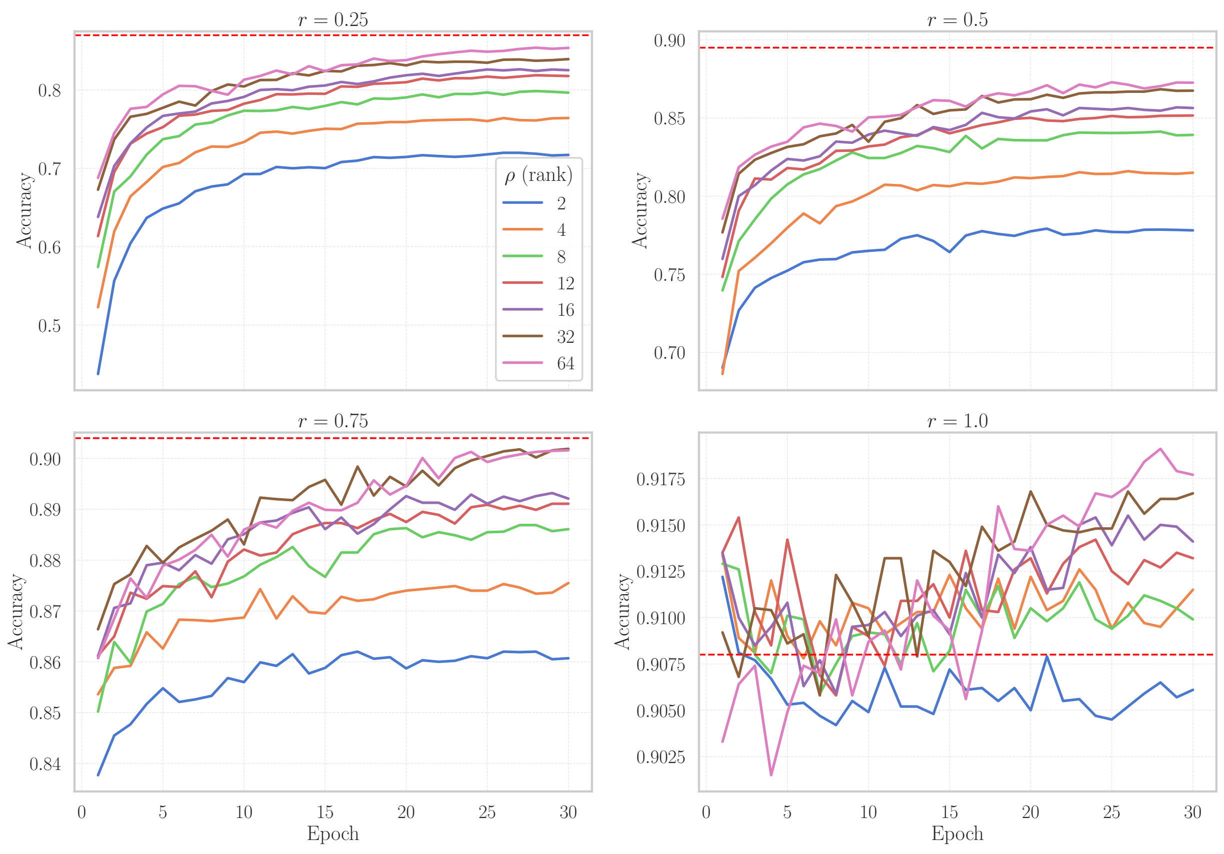

Figure 4.3: Test accuracy of pre-trained ResNet20 on CIFAR-10 adapted with different LoRA ranks, across various slimmable ratios. Red lines represent the accuracy achieved by the US baseline at the corresponding ratio r.

4.2 Moving to Vision Transformers

In this section, we introduce Slimmable adaptation to a different class of models for image classification: Vision Transformers. Being more recent than CNNs, the original Slimmable Networks paper did not feature this family of models in its experiments. Their larger model size compared to most CNN models (driven mainly by the adoption of fully-connected layers), powerful representational capabilities and high computational costs for training make Vision Transformers a suitable target for pre-trained Slimmable adaptation. Thus we asked whether the US-Nets framework can be applied to ViTs. Ideally, we would like to adhere to the Slimmable formulation applied to Convolutional Networks, with minimal modifications, keeping our implementation conceptually simple.

In principle, training a Universally Slimmable Vision Transformer models is technically straightforward, since these models (**i**) do not contain normalization layers that rely on batch-dependent statistics (no calibration is required) and (**ii**) all other non-normalization layers are fully-connected linear layers. However, the Attention operator represents a novelty that has not been addressed in the original work on US-Nets.

Scala framework for US-ViT. In the literature, the work by Zhang et al. [35] introduces *Scala*, a framework for training Universally Slimmable ViT models from scratch. The authors start by highlighting that, when considering uniform slicing of ViT weight matrices in both their input and output dimensions, the architecture structure is not impacted by the slicing operation: with the same underlying architecture across all nested sub-networks, the number of parameters of any given sub-network is directly controlled by simply modulating the embedding dimension D. Thus a Vision Transformer model can be naturally regarded as a collection of sub-networks that share weights in a nested structure, in which a specific sub-network arises by slicing weight matrices of each layer at a given ratio r. The authors point out that this is valid regardless of the fact that variants of ViTs might differ also in the number of *attention heads*, since usually $D = hd_{model}$. Thus one can change D by modulating d_{model} without affecting the number of heads: $rD = h(rd_{model})$. This approach also increases the granularity of the range of possible dimensions, since usually $d_{model} >> h$. While technically correct, we believe however that this specific architectural property pertaining to Vision Transformers must be carefully taken into account when implementing the slicing mechanism in

code. The aspect of guaranteeing the nested structure of the Slimmable framework in Attention-based architectures is thus one the central topics around which the following section will develop around.

The experiments proposed in the paper paint a peculiar picture of Slimmable Vision Transformers. After training a ViT on a set of fixed switches $r \in [0.25, 0.5, 0.75, 1.0]$ (as first proposed in [33]), the authors show that evaluating the model on untrained values of r causes test accuracy to catastrophically drop to random guessing: ViTs appear to completely lack interpolation and extrapolation abilities to widths not directly trained. This striking result positions Vision Transformers in contrast with CNNs trained in the same fashion, for which sub-networks not explicitly seen during training nonetheless benefit from the performance of the trained ones, by virtue of their nested structure. Moreover, by analysing the performance across the full range of dimensionalities trained following the Universally Slimmable training recipe, results suggest that constantly activating the smallest sub-network $r = s$ (e.g., $s = 0.25$) at each iteration hinders the slimmable performance of larger sub-networks. Motivated by this last observation, Scala introduces a technique for "isolating" the parameters of the the smallest sub-network from the others during training. When $r = s$, the weight matrix $W \in \mathbb{R}^{C_o \times C_i}$ is sliced in the opposite direction:

$$W_s = W[: sC_o, : sC_i]$$
$$W_{r \neq s} = W[-rC_o :, -rC_i :]$$

Such solution breaks the core of the Slimmable paradigm: while W still encompasses all smaller sub-networks, slicing does not happen in a fully nested fashion. $W_{r \neq s}$ still retains its nested structure, but W_s now acts as a stand-alone sub-network.

4.2.1 ViT slicing strategy

A primary consideration in this study was the development of a slicing strategy tailored for the Vision Transformer architecture. The objective was to extend the application of permutations and saliency-aware slicing to ViTs, paralleling the methodology established for CNNs in Section 4.1. Preliminary analysis indicated that implementing uniform slicing of the embedding dimension D, as proposed by Scala, while simultaneously incorporating permutations presented significant technical challenges.

Avoiding residual path dependencies. As with the CNN case, the set of transformations that preserve computational invariance are determined by the Transformer architecture. With reference to Figure 2.2, a Transformer layer can be decomposed into two main sub-components each enclosed in a skip connection: (**i**) a LayerNorm block followed by Attention; (**ii**) another LayerNorm block followed by a MLP. Each of the two skip connections define a *dependency* between the origin and the destination of the skip path: thus every permutation that affects the activations at the end of the connection must be applied identically to the activations at the origin of the connection. As a direct consequence of the ViT architecture, these dependencies are propagated across all components of every layer in the network. The end result is that only one single global permutation can be applied on weight matrices that either produce or consume intermediate activations affected by skip connections.

One possible approach to break these dependencies is the one taken in SliceGPT [1], where a linear transformation is inserted in the skip connection. The new linear layer must "undo" the permutation applied at the source of the skip connection and subsequently apply the permutation matching the destination of the connection. Adding a linear layer in this fashion introduces additional parameters (that are not learned) and a computational overhead. For the sake of simplicity, we follow a similar solution as the residual block case pictured in Figure 3.1 and bypass the dependency problem by restricting slicing of weight matrices to the interfaces between layers free from skip connection dependencies. As pictured in Figure 4.4, we perform slicing (**a**) on the output dimension of W_Q, W_K, W_V and subsequently on the input dimension of W_O, and (**b**) on the output dimension of W_{up} and input dimension of W_{down}. Exactly like the "bottleneck" slicing of the CNN residual block, the FLOP reduction and parameter count compression are directly proportional to the slicing ratio r.

Attention-aware slicing. After defining our slicing scheme for supporting permutation of weight matrices, we turned to inspecting the uniform slicing scheme proposed in Scala. Our first step consisted in studying the original publicly available code implementation. One aspect we observed right away was the method with which weight matrices related to the attention layer were sliced. To better understand this, we have to briefly discuss the practical implementation of the attention layer in Vision Transformers.

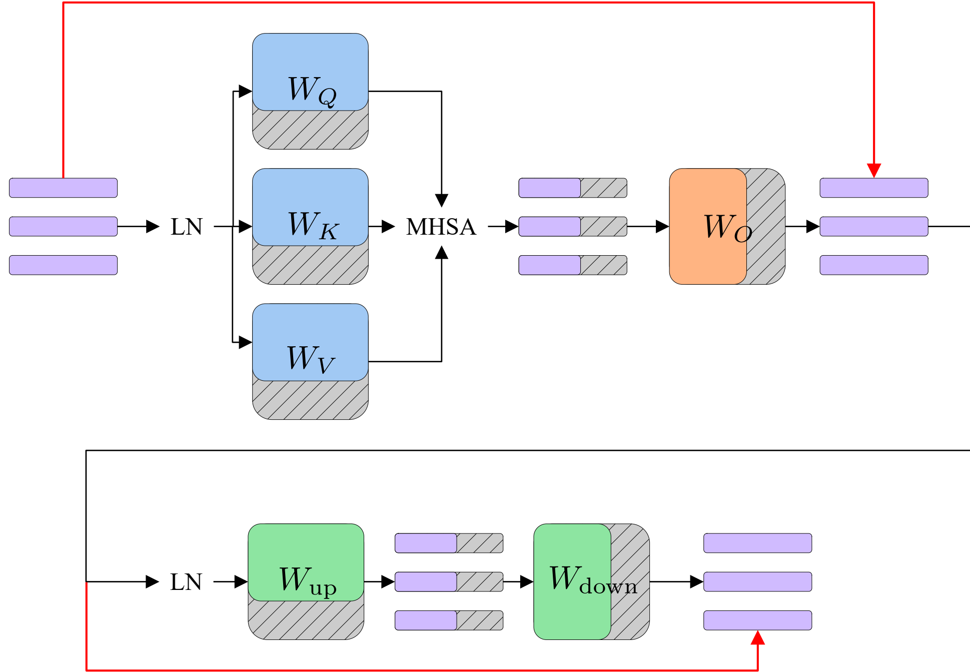

Figure 4.4: Our proposed slicing scheme for a ViT layer. Restricting slicing to interfaces free from skip connection dependencies allows dedicated permutations for each weight matrix.

Section 2.1.3 outlines how Q, K, V embeddings of an input token for the n-th attention head are obtained via dedicated linear projections $W_{(n,Q)}$, $W_{(n,K)}$, $W_{(n,V)}$. Given the absence of data dependencies between these $3 \times h$ projections, their computation can be trivially parallelized by first concatenating per-head matrices along their output dimension, and then concatenate the per-embedding matrices again along the same dimension. This result in a single matrix $W_{QKV} \in \mathbb{R}^{(3hC_o) \times C_i}$. Listing 4.1 illustrates a simplified implementation in PyTorch-like code.

Even if coming from the output of a single linear projection, it is clear that components retain their semantic meaning in the context of the Attention operator. mathematically, this is identical to computing each projection independently from the others.

When slicing W_{QKV}, Scala follows the implementation shown in Listing 4.2

Figure 4.5 illustrates how slicing proposed by Scala affects the structure of W_{QKV}. Supposing that $r = 2/3$, the resulting matrix after slicing has entirely lost W_V. Distributing the remaining weights across the same number of heads concretely reduces the embedding dimension by the given ratio r, the end result is quite different than having instead resized *every* head: this imposes a complicated nesting scheme in which weights of the matrix are trained as if belonging to different heads and

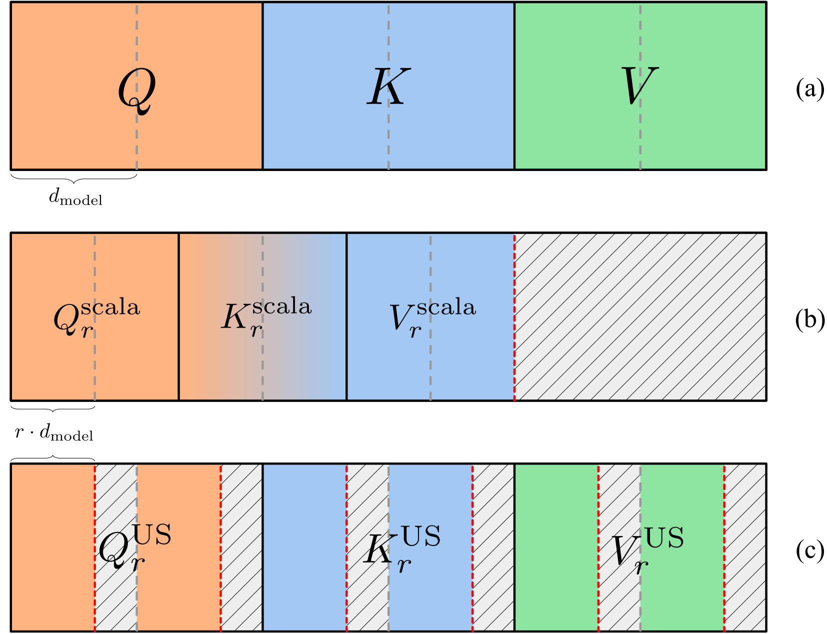

Figure 4.5: Illustration of different slicing methods applied to reduce the output dimension of Q,K,V projections. (a) shows the full weight matrix, obtained from concatenating the three distinct projections. In this example, each projection has $h = 2$ heads, drawn with a grey dashed line. The slicing ratio is $r = 2/3$. The point where channels are sliced is marked with a red dashed line. Part (b) illustrates the Scala-style slicing. Slicing the matrix as a whole unit and then partitioning the remaining channels between projections, despite effectively reducing each head's size, may cause some channels to be assigned to a different projection than the one they belonged to before slicing. Our method (c) slices each head independently and concatenates the remaining channels together to obtain a final matrix with modulated output size. This approach guarantees that channels are consistently mapped to their original projection matrix after slicing.

Code Listing 4.1: Simplified MHSA PyTorch-like implementation.

```
def MHSA(x):
    B, N, D = x.shape
    # batched across batch-size and sequence length
    qkv = linear(x, W_qkv)
    # results are concatenated together
    qkv = qkv.reshape(B, N, 3, h, d_model)
    # permute tensor:
    # (B, N, 3, h, d_model) -> (3, B, h, N, d_model)
    qkv = qkv.permute(2, 0, 3, 1, 4)
    q, k, v = qkv[0], qkv[1], qkv[2]
    # msha
    dots = matmul(q, k.transpose(1,2)) / scale
    dots = softmax(dots, dim=-1)
    attn = matmul(dots, v)
    # permute (B, h, N, d_model) -> (B, N, h, d_model)
    attn = attn.transpose(1,2)
    # concatenate heads (just flatten last two axes)
    attn = attn.reshape(B, N, D)
    # final projection
    out = linear(attn, W_o)
    return out
```

Code Listing 4.2: Scala slicing on the Attention layer.

```
B, N, D = x.shape
C_i_sliced = int(C_i * r)
C_o_sliced = int(C_o * r)
if r == s or r == max_r:
    W_qkv_slice = W_qkv[:C_o_sliced*3, :C_i_sliced]
else:
    W_qkv_slice = W_qkv[-C_o_sliced*3:, -C_i_sliced:]
qkv = linear(x, W_qkv_slice)
# D is already sliced accordingly,
# given the uniform slicing regime
qkv = qkv.reshape(B, N, 3, h, D)
# ...
if r == s or r == max_r:
    W_o_slice = W_o[:C_o_sliced*3, :C_i_sliced]
else:
    W_o_slice = W_o[-C_o_sliced*3:, -C_i_sliced:]
out = linear(attn, W_o_slice)
return out
```

Code Listing 4.3: Our slicing on the Attention layer.

```
B, N, d = x.shape
dim = int(d_model * r)
in_dim = int(D * r) # equal to "d"
# expose the underlying structure of W_qkv
W_qkv_slice = W_qkv.reshape(3, h, d_model, D)
# slice **each** head separately
W_qkv_slice = W_qkv_slice[:, :, :dim, :in_dim]
# stack back into a single matrix
W_qkv_slice = W_qkv_slice.reshape(-1, D)
qkv = F.linear(x, W_qkv_slice)
# ...
# in this case, only the input dim. is stacked
W_o_slice = W_o.reshape(D, h, -1)
W_o_slice = W_o_slice[:in_dim, :, :dim]
W_o_slice = W_o_slice.reshape(in_dim, -1)
#...
```

even to different projections, depending on the value of r. Motivated by the belief that such scheme might play a role in the interpolation-extrapolation ability of Vision Transformers highlighted in Scala's experiments, we propose a different slicing scheme implemented in Listing 4.3.

To evaluate our semantic-aware slicing strategy, we trained a small ViT variant (embedding size $D = 192$, $h = 8$ heads, $N = 6$ layers) from scratch on CIFAR-100 [17] for 80 epochs, using AdamW with a initial learning rate of 10^{-3} and a cosine decay scheduler with 10 epochs of linear warmup. To isolate the effect of Scala's slicing on Attention, we decided to keep a standard slicing approach in the MLP (without smallest ratio "isolation").

Results are shown in Figure 4.6. All models are evaluated at every possible value d_{model} can take: in this case, having $D = 192$ and $h = 8$, it means $d_{model} = D/h = 24$ possible widths. Despite having almost identical performance when evaluated on trained ratios, Scala consistently fails at unseen widths. On the other hand, nesting sub-networks following the structure imposed by the Attention operation's semantics as proposed by our approach shows the same extrapolation behaviour of CNNs.

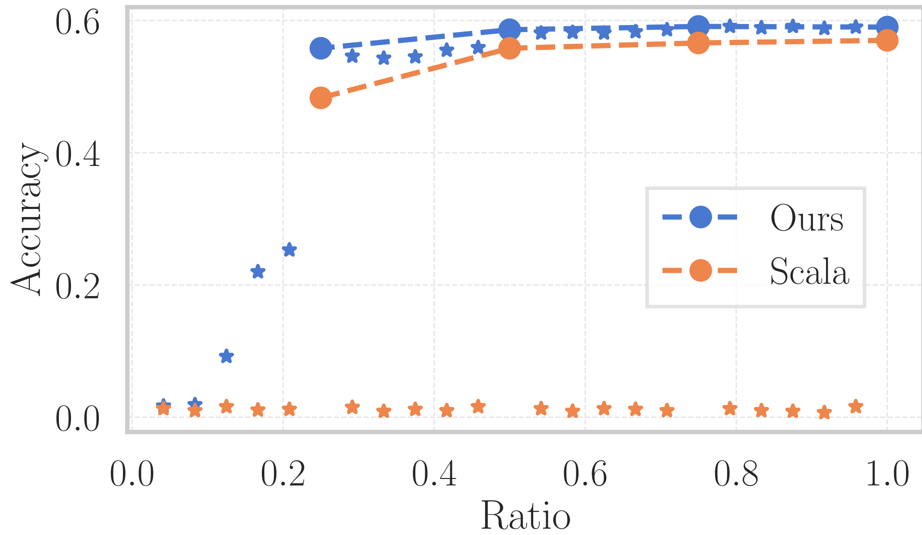

Figure 4.6: Evaluation of semantic-aware slicing of Attention-related weight matrices. In this case, Scala "isolation" is limited to the Attention projection matrices. Our proposed slicing scheme retains the interpolation-extrapolation capabilities typical of CNNs described in the original Scala paper.

4.2.2 Full Slimmable adaptation experiments

A series of experiments was conducted to evaluate the feasibility of adapting pre-trained ViTs into slimmable architectures. This experimental framework deviates from the convolutional approach detailed in Section 4.1 in several key aspects. The primary divergence lies in the discrepancy between the pre-training and adaptation datasets. Given that empirical evidence suggests Transformers necessitate larger datasets than CNNs for effective convergence, publicly available pre-trained models are predominantly trained on the ImageNet dataset. In contrast, the present study utilizes the CIFAR-100 [17] dataset for adaptation, selected to balance computational tractability with task complexity. With a tenfold increase in number of target classes compared to CIFAR-10, CIFAR-100 should provide a sufficiently rigorous benchmark for this evaluation.

Consequently, this knowledge transfer setting introduces a novel element to this context, as it was not addressed during the ResNet evaluations in Section 4.1. Note that ImageNet exhibits a significantly more complex data distribution than CIFAR-100, characterized not only by a higher

categorical dimensionality (1000 classes versus 100) but also by a different image spatial resolution (224×224 RGB images compared to the 32×32 format of CIFAR-100).

Mainly due to their higher computational requirements, we decided to limit our experiments to a ViT variant with a contained number of parameters. We elected ViT-Tiny/16 (patch size 16×16) as our target variant. As already mentioned, we work with the public pre-trained checkpoint tagged `vit_tiny_patch16_224` from the PyTorch Image Models `timm` [30] repository.

Baseline. Given the burden of training from scratch a Vision Transformer, we decided to instead obtain a baseline by fine-tuning the ImageNet pre-trained checkpoint on the CIFAR-100 dataset. As for the ResNet case, we trained four baselines in isolation, at ratios $r \in \{0.25, 0.5, 0.75, 1.0\}$. Since pre-trained weights where only available for the full configuration, we obtained the resized configurations by simply keeping the weights with lowest indices in parameter tensors, following the structure of our slicing strategy. We decided to not prepare the weights by applying permutations: the selected weights have not been selected accordingly to any additional property except for their initial (static) location in the weight tensors. Even if realistically such a simple scheme is sub-optimal, we believe it can still provide a simple comparison point for evaluating the performance of slimmable adaptation. Notice that, as illustrated in the previous chapter, slicing weight matrices without first applying permutation could result in the loss of salient weights; in this particular case, we notice that even if this fact is in practice true, the sliced networks retain a conditioning from pre-training that would be lost with random initialization of weights.

Regarding the classification head, we adapt it to work with classification task consisting of a different number of classes by following the same approach proposed in [6], where its parameters are zero-initialized prior to training. We train with AdamW for 30 epochs with global batch size 256; the learning rate is first increased linearly to a maximum of 0.001 with a 5 epochs warmup, then it is decayed with a cosine schedule to a final value of 10^{-5}. Weight decay is set to $5 \cdot 10^{-4}$. Data augmentation features random cropping and random horizontal flip (as adopted in the CIFAR-10 case). Since the original network was trained with 224×224 images, we resize images to match that shape via bicubic interpolation. Lastly, normalization is performed. Table 4.3 collects the results achieved by our baselines.

Ratio	Top-1 Accuracy (%)	Param. count
1.0	82.84	5.47M
0.75	80.20	4.14M
0.5	73.97	2.81M
0.25	63.95	1.48M

Table 4.3: ImageNet pre-trained ViT-Tiny/16 CIFAR-100 individual training. In order to resize the model, we modulate with a ratio r (**a**) the output size of the Attention projections Q,K,V; (**b**) the input dimension of the projection following MHSA; (**c**) the dimension between the two MLP layers. All other components, such as Layer Normalization, classifier head, patch embedding projection and class token are not resized.

Slimmable Adaptation with permutations. As in Section 4.1, we tested the performance of directly fine-tuning a pre-trained, non-slimmable network with slimmable training. The training setup is identical to the baseline illustrated in the previous section. The main difference with respect to the ResNet20 experiments is that for ViT we evaluate only two out of three permutation strategy: no permutation and ℓ_1-based permutation. The choice to exclude ThiNet was made after observing that the greedy iterative algorithm of ThiNet required substantial computational time to determine the permutation of the 768-D hidden layer of the MLP. Experiments involving ThiNet or about how to speedup this kind of computation could be an interesting avenue for future work.

Similarly to CNNs, permutations must produce a network that is functionally equivalent to the original one. For the MLP, identically to the two convolutional layers in our ResNet "bottleneck" block, the output channels' ordering of the first layer determines the ordering of the second layer's input channels. For Attention-related projections, the semantics of Self-Attention must be taken into account. For our discussion, we consider the projections corresponding to a single Attention head, since the same scheme must be applied to each head independently in order to guarantee functional equivalence. In the case of W_Q and W_K, the dot-product interaction between the resulting query and key tokens necessitates that their output channels follow an identical ordering; any independent permutation would misalign the element-wise correspondence necessary to preserve dot-products, altering the resulting Attention map. Thus we elected to base the permutation on the ℓ_1-norm of

W_Q and apply it identically to W_K. Similarly, the output channels of W_V must remain synchronized with the corresponding input channels of W_O: thus the ordering imposed on the former is mirrored on the latter.

The plot in Figure 4.7 shows that slimmable fine-tuning is competitive with the baseline chosen. At lower ratios, both US-adapted variants lag behind the individually fine-tuned networks; the situation is instead reversed above $r = 0.5$, where the slimmable variants achieve better final accuracy than the baseline. It seems that larger sub-networks benefit from well-performing nested sub-networks, but the latter incur in a degradation of accuracy compared to individual networks optimized in isolation. Regarding permutation strategies, ℓ_1 score guidance provides a fair advantage over the non-ordered counterpart across essentially the whole compression spectrum; only at full dimensionality the two setups are equivalent.

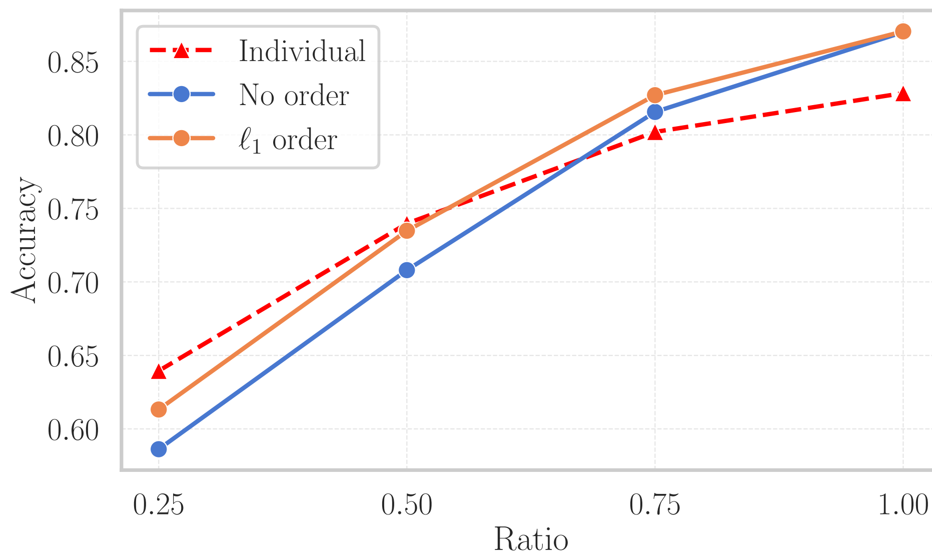

Figure 4.7: ViT-Tiny/16 slimmable adaptation with different ordering strategies.

4.2.3 Slimmable adaptation with LoRA

Just like in Section 4.1.2, we compressed the slimmable update matrices with LoRA and ran experiments to observe the impact of such compression on the resulting slimmable capabilities of the adapted networks. In order to simplify the implementation, we select rank dimensions

uniformly across all components; additionally, we always keep the coefficient α the same as ρ, so that $\rho/\alpha = 1$. We targeted with LoRA all weight matrices undergoing slimmable adaptation. For the unified Attention projection matrix, we decided to factorize it like a single entity, but in order to keep the parameter count consistent with the fact that there are actually three weight matrices we multiplied the rank by a factor of 3. Given our choice of slicing scheme, similarly to the ResNet case, only one between the LoRA matrices A and B has its non-rank dimension modulated for a given ratio r, depending on whether the output or input channel number is to be sliced. So for instance, since our slicing scheme mandates that only the output dimension of the Attention Queries, Keys, Values projection is to be resized, slicing is only performed on LoRA matrix B; similarly to modulate the intermediate dimension for the first and second fully-connected layers in the MLP, we slice matrix B in the former and matrix A in the latter. Lastly, when modulating the channels of Attention-related projections, we maintain the same semantic-aware slicing described in Figure 4.5.

Results. Without changing the training recipe from the previous section, we adapted the same ViT-Tiny/16 checkpoint using LoRA ranks $\rho \in \{8, 16, 32, 64\}$. Table 4.4 presents a breakdown of the number of parameters of LoRA components (summed across all Transformer layers) versus the original full rank model. Figure 4.8 illustrates the achieved accuracy of LoRA Slimmable adaptation compared to the full rank individually trained models and the full rank ℓ_1-ordered Slimmable adapted model from Figure 4.7. The graph shows a clear performance degradation with respect to both baselines for every rank value. Expectedly, the more extreme compression setups suffer higher loss of accuracy, especially at lower ratios (0.25 and 0.5). For larger ratios, the network appears to better retain accuracy even in the compressed scenario. The performance of the full network does not seem to be affected by LoRA compression. Clearly we have to interpret these results as just indicative, since results on such a small pre-trained model and fine-tuning dataset might not scale. Another interesting aspect to be studied might be the allocation of rank dimensions across various components. Since for instance the MLP represents the major part of the ViT model, even restricting LoRA compression only to its fully-connected layers could provide an interesting trade-off between adaptation cost and efficacy.

Table 4.4: Parameter size breakdown of full rank components in a ViT-Tiny/16 full versus their LoRA counterparts, at various rank dimensions. As already observed in ResNet20's case, uniformly setting ρ across components results in one configuration (marked with †) having more parameters than the full rank component; the final compound parameter count is still smaller than the original network for all considered values of ρ (rightmost column)

Rank (ρ)	**QKV**	**Proj**	**MLP**	**% of orig. (# params.)**
Original	1,327,104	442,368	3,538,944	100.00% (5.47M)
8	221,184	36,864	184,320	10.96% (0.60M)
16	442,368	73,728	368,640	19.06% (1.04M)
32	884,736	147,456	737,280	35.25% (1.93M)
64	1,769,472†	294,912	1,474,560	67.62% (3.70M)

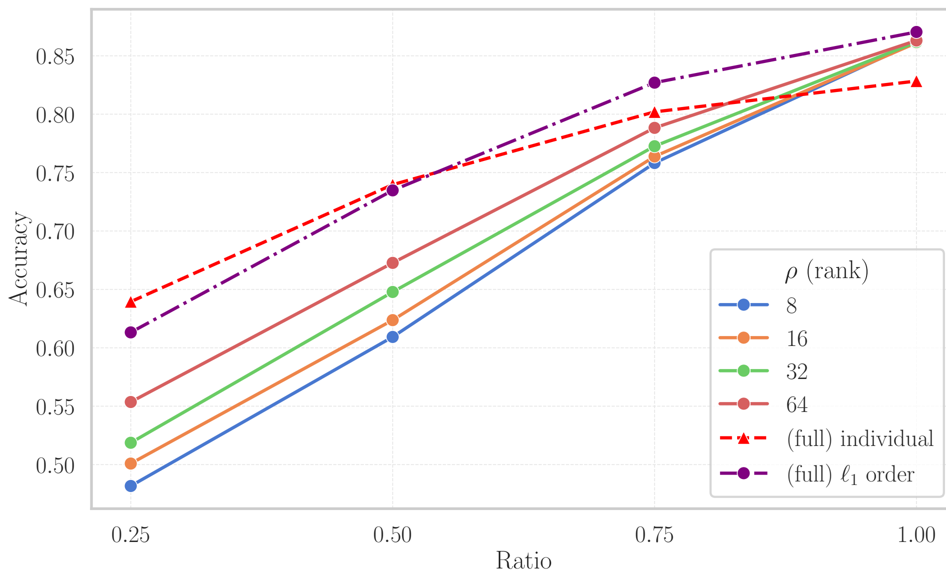

Figure 4.8: Slimmable adaptation accuracy for ViT-Tiny/16 with ℓ_1-based permutation on CIFAR-100, using LoRA. Performance is compared with two full rank baselines: individually trained networks at different ratios and ℓ_1 ordered entry of Figure 4.7

5. Conclusions

This work investigated the adaptation of pre-trained neural networks to support runtime flexibility in computational requirements. We explored two primary strategies: a generative approach utilizing hypernetworks and a direct adaptation approach using slimmable fine-tuning principles.

In the first part of our research, we extended the Neural Metamorphosis framework. We identified that the original approach faced scalability challenges when applied beyond small network subsets due to the high ratio of hypernetwork parameters to main network parameters. By reframing weight generation as a residual learning problem and incorporating data-driven permutations such as ThiNet, we improved the hypernetwork's ability to maintain performance at lower compression ratios. However, the optimization complexity and memory bottlenecks of secondary networks remained significant for full architectures.

The second part of the thesis focused on direct slimmable fine-tuning. We demonstrated that it is possible to "induce" a nested, slimmable structure into a standard pre-trained network. Our experiments indicated that initializing the network with sorted weight manifoldsâĂŤusing magnitude-based or data-driven scoringâĂŤprovides a stable starting point that helps preserve the model's original representations during adaptation. Furthermore, we found that limiting the fine-tuning to specific layers can maintain the network's functional capabilities while reducing the computational cost of the transition.

For the generative approach, we believe there are opportunities for further improvement, even in the restricted setting with simple networks and datasets that this work primarily targeted. Future work could, for instance, be focused on enhancing robustness of hypernetwork training and the generalization capabilities on unseen slicing widths. For direct Slimmable adaptation, one interesting direction for future studies would be to evaluate these adaptation methods on more complex models, such as larger Vision Transformers or even Large Language Models, where the efficiency of the adaptation process is even more critical. Regarding the choice of slicing ratio and the allocation of ranks in LoRA, there is space to experiment with more advanced allocation strategies for this quantities; for instance, evaluating criteria for tailored choice of such quantities in different layers.

Bibliography

[1] Saleh Ashkboos et al. "Slicegpt: Compress large language models by deleting rows and columns". In: *arXiv preprint arXiv:2401.15024* (2024).

[2] Maor Ashkenazi et al. *NeRN – Learning Neural Representations for Neural Networks*. arXiv:2212.13554 [cs]. Apr. 2023. DOI: 10.48550/arXiv.2212.13554. URL: http://arxiv.org/abs/2212.13554 (visited on 04/01/2025).

[3] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. "Layer normalization". In: *arXiv preprint arXiv:1607.06450* (2016).

[4] Yaofo Chen. *PyTorch CIFAR Models*. https://github.com/chenyaofo/pytorch-cifar-models. Accessed: 2025-5-17.

[5] Jacob Devlin et al. "Bert: Pre-training of deep bidirectional transformers for language understanding". In: *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*. 2019, pp. 4171–4186.

[6] Alexey Dosovitskiy et al. "An image is worth 16x16 words: Transformers for image recognition at scale". In: *arXiv preprint arXiv:2010.11929* (2020).

[7] Jonathan Frankle and Michael Carbin. "The lottery ticket hypothesis: Finding sparse, trainable neural networks". In: *arXiv preprint arXiv:1803.03635* (2018).

[8] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. "Deep sparse rectifier neural networks". In: *Proceedings of the fourteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings. 2011, pp. 315–323.

[9] David Ha, Andrew Dai, and Quoc V. Le. *HyperNetworks*. arXiv:1609.09106 [cs]. Dec. 2016. DOI: 10.48550/arXiv.1609.09106. URL: http://arxiv.org/abs/1609.09106 (visited on 02/11/2025).

[10] Kaiming He et al. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[11] Dan Hendrycks and Kevin Gimpel. "Gaussian error linear units (gelus)". In: *arXiv preprint arXiv:1606.08415* (2016).

[12] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. "Distilling the knowledge in a neural network". In: *arXiv preprint arXiv:1503.02531* (2015).

[13] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. "Multilayer feedforward networks are universal approximators". In: *Neural Networks* 2.5 (1989), pp. 359–366. ISSN: 0893-6080. DOI: https://doi.org/10.1016/0893-6080(89)90020-8. URL: https://www.sciencedirect.com/science/article/pii/0893608089900208.

[14] Edward J. Hu et al. *LoRA: Low-Rank Adaptation of Large Language Models*. arXiv:2106.09685 [cs]. Oct. 2021. DOI: 10.48550/arXiv.2106.09685. URL: http://arxiv.org/abs/2106.09685 (visited on 07/01/2025).

[15] Sergey Ioffe and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift". In: *International conference on machine learning*. pmlr. 2015, pp. 448–456.

[16] Eunwoo Kim, Chanho Ahn, and Songhwai Oh. "Nestednet: Learning nested sparse structures in deep neural networks". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2018, pp. 8669–8678.

[17] Alex Krizhevsky, Geoffrey Hinton, et al. "Learning multiple layers of features from tiny images". In: (2009).

[18] Yann Le Cun, John S. Denker, and Sara A. Solla. "Optimal Brain Damage". In: *Proceedings of the 3rd International Conference on Neural Information Processing Systems*. NIPS'89. Cambridge, MA, USA: MIT Press, 1989, pp. 598–605.

[19] Chunyuan Li et al. *Measuring the Intrinsic Dimension of Objective Landscapes*. arXiv:1804.08838 [cs]. Apr. 2018. DOI: 10.48550/arXiv.1804.08838. URL: http://arxiv.org/abs/1804.08838 (visited on 06/06/2025).

[20] Hao Li et al. *Pruning Filters for Efficient ConvNets*. arXiv:1608.08710 [cs]. Mar. 2017. DOI: 10.48550/arXiv.1608.08710. URL: http://arxiv.org/abs/1608.08710 (visited on 05/13/2025).

[21] Ilya Loshchilov and Frank Hutter. "Decoupled weight decay regularization". In: *arXiv preprint arXiv:1711.05101* (2017).

[22] Jian-Hao Luo, Jianxin Wu, and Weiyao Lin. *ThiNet: A Filter Level Pruning Method for Deep Neural Network Compression*. arXiv:1707.06342 [cs]. July 2017. DOI: 10.48550/arXiv.1707.06342. URL: http://arxiv.org/abs/1707.06342 (visited on 05/07/2025).

[23] Nasim Rahaman et al. "On the spectral bias of neural networks". In: *International conference on machine learning*. PMLR. 2019, pp. 5301–5310.

[24] Oren Rippel, Michael Gelbart, and Ryan Adams. "Learning ordered representations with nested dropout". In: *International Conference on Machine Learning*. PMLR. 2014, pp. 1746–1754.

[25] Karen Simonyan and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition". In: *arXiv preprint arXiv:1409.1556* (2014).

[26] Thomas Sommariva, Simone Calderara, and Angelo Porrello. "How to Train Your Metamorphic Deep Neural Network". In: *arXiv preprint arXiv:2505.05510* (2025).

[27] Christian Szegedy et al. "Going deeper with convolutions". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2015, pp. 1–9.

[28] Matthew Tancik et al. "Fourier features let networks learn high frequency functions in low dimensional domains". In: *Advances in neural information processing systems* 33 (2020), pp. 7537–7547.

[29] Ashish Vaswani et al. "Attention is all you need". In: *Advances in neural information processing systems* 30 (2017).

[30] Ross Wightman. *PyTorch Image Models*. https://github.com/rwightman/pytorch-image-models. 2019. DOI: 10.5281/zenodo.4414861.

[31] Xingyi Yang and Xinchao Wang. *Neural Metamorphosis*. arXiv:2410.11878 [cs]. Oct. 2024. DOI: 10.48550/arXiv.2410.11878. URL: http://arxiv.org/abs/2410.11878 (visited on 01/22/2025).

[32] Jiahui Yu and Thomas Huang. *Universally Slimmable Networks and Improved Training Techniques*. arXiv:1903.05134 [cs]. Oct. 2019. DOI: 10.48550/arXiv.1903.05134. URL: http://arxiv.org/abs/1903.05134 (visited on 05/07/2025).

[33] Jiahui Yu et al. *Slimmable Neural Networks*. arXiv:1812.08928. Dec. 2018. DOI: 10.48550/arXiv.1812.08928. URL: http://arxiv.org/abs/1812.08928 (visited on 11/14/2024).

[34] Sergey Zagoruyko and Nikos Komodakis. "Wide residual networks". In: *arXiv preprint arXiv:1605.07146* (2016).

[35] Yitian Zhang et al. "Slicing vision transformer for flexible inference". In: *Advances in Neural Information Processing Systems* 37 (2024), pp. 42649–42671.