

RRR-Net: Reusing, Reducing, and Recycling a Deep Backbone Network

Haozhe Sun ¹ Isabelle Guyon ^{1,2} Felix Mohr ³ Hedi Tabia ⁴

¹ LISN/CNRS/INRIA, Université Paris-Saclay, France

² ChaLearn, USA

³ Universidad de La Sabana, Colombia

⁴ IBISC, Univ. Evry, Université Paris-Saclay, France
Email: haozhe.sun@universite-paris-saclay.fr

Abstract

It has become mainstream in computer vision and other machine learning domains to reuse backbone networks pre-trained on large datasets as preprocessors. Typically, the last layer is replaced by a shallow learning machine of sorts; the newly-added classification head and (optionally) deeper layers are fine-tuned on a new task. Due to its strong performance and simplicity, a common pre-trained backbone network is ResNet152. However, ResNet152 is relatively large and induces inference latency. In many cases, a compact and efficient backbone with similar performance would be preferable over a larger, slower one. This paper investigates techniques to reuse a pre-trained backbone with the objective of creating a smaller and faster model. Starting from a large ResNet152 backbone pre-trained on ImageNet, we first reduce it from 51 blocks to 5 blocks, reducing its number of parameters and FLOPs by more than 6 times, without significant performance degradation. Then, we split the model after 3 blocks into several branches, while preserving the same number of parameters and FLOPs, to create an ensemble of sub-networks to improve performance. Our experiments on a large benchmark of 40 image classification datasets from various domains suggest that our techniques match the performance (if not better) of “classical backbone fine-tuning” while achieving a smaller model size and faster inference speed.

1 Background and motivations

Over the last decade, Deep Learning has set new standards in computer vision. Tasks in this area include the recognition of street signs, placards, and living beings. While it has achieved state-of-the-art in various academic and industrial fields, training deep networks from scratch requires massive amounts of data and hours of GPU training, which prevents it from being deployed in data-scarce and resource-scarce scenarios.

This limitation has been mainly addressed through the notion of *Transfer learning* [1]. Here, knowledge is transferred from a *source domain* (typically learned from a large dataset) to one or several *target domains* (typically with less available data). A common transfer learning approach is last-layer fine-tuning [2], in which a considerable part (the *backbone*) of a pre-trained deep network is reused; only the last layer is replaced with a new classifier and trained to the new task at hand. Depending on the distribution shift between the source domain and target domains, more layers may be fine-tuned. Pre-trained networks that have been used for fine-tuning range from the historical AlexNet [3] to various ResNets [4,5].

Modern neural networks are thought of as being “the bigger, the better” as big networks keep beating large benchmarks (such as ImageNet [6]). However, they are considerably over-parameterized when applied to smaller tasks. There is evidence that low-complexity models can, in some conditions, lead to comparably good or better performance [7]. Our charter is to elaborate on the basic “Reuse” methodology described above (replacing the last layer with a new classifier) by applying the three classical resource-saving “precepts” to the greatest possible extent: “**R**euse, **R**educe, and **R**ecycle” [8]. For simplicity of demonstration, we limit ourselves to the popular ResNet152 model [4], which provides us with enough flexibility to carry out our analyses. In this architecture, the “Reduce” step means to reduce the number of residual blocks in the network, and “Recycle” is done by creating a voting ensemble from existing layers. We leverage the reusability of a pre-trained backbone in part.

Our experimental results show that, for many tasks, a pre-trained ResNet152 can be significantly reduced in order to benefit from faster inference speed while achieving similar (if not better) performance. Our evaluation

Phases	output size	Blocks		Num. parameters		FLOPs (10^6)	
		Baseline	Ours (conv4_x, conv5_x: each branch)	Baseline	Ours	Baseline	Ours
conv1	32×32	$7 \times 7, 64, \text{conv (stride 2)}$ $3 \times 3, \text{max-pool (stride 2)}$	$7 \times 7, 64, \text{conv (stride 2)}$ $3 \times 3, \text{max-pool (stride 2)}$	9.5k	9.5k	40	40
conv2_x	32×32	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 1$	70k (75k)	70k (75k)	73 (78)	73 (78)
conv3_x	16×16	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 8$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 1$	280k (379k)	280k (379k)	72 (123)	72 (123)
conv4_x	8×8	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 36$	$\begin{bmatrix} 1 \times 1, 256/\sqrt{b} - a \\ 3 \times 3, 256/\sqrt{b} - a \\ 1 \times 1, (256/\sqrt{b} - a) \times 4 \end{bmatrix} \times 1$	1.12M (1.51M)	$136k (333k) \times 8$	72 (122)	$9 (30) \times 8$
conv5_x	4×4	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512/\sqrt{b} - a \\ 3 \times 3, 512/\sqrt{b} - a \\ 1 \times 1, (512/\sqrt{b} - a) \times 4 \end{bmatrix} \times 1$	4.46M (6.04M)	$553k (745k) \times 8$	72 (122)	$9 (15) \times 8$
Total				58.23M	9.09 M	3799	601

Figure 1: **The structure and approximated statistics of models.** We call “Baseline” the original ResNet152 [4]. We call “Ours” the reduced and recycled architecture. Reduction means removing blocks from each phase (until one block in each phase). Recycling means splitting blocks in the conv4_x and conv5_x phases into multiple branches while preserving the total number of parameters and FLOPs. This table assumes the input size is 128×128 . Building blocks are shown in brackets, with the numbers of blocks stacked. For conv4_x and conv5_x in the “Ours” column, blocks are split into branches, where b denotes the number of branches, and a is a scalar to adjust the model size. The “Num. parameters” and “FLOPs” columns assume $b = 8$. The values in parentheses indicate the statistics of the first block of each phase, which is responsible for downsampling. $k = 10^3$, $M = 10^6$. FLOPs mean floating-point operations or multiply-adds; they are measured with respect to a single $128 \times 128 \times 3$ image using the open source software Pytorch-OpCounter [27].

is quite extensive, being based on 40 datasets from various domains [9]. The positive results obtained with the ResNet152 model suggest generalizing our techniques to other architectures.

2 Related work

This paper combines several ideas: *reusing* a pre-trained backbone network, *reducing* it, and splitting the model into several branches to use ensemble techniques (*recycling*). We briefly review related work in these three domains.

Given the availability of neural networks trained on large datasets, the vanilla fine-tuning approach [2] remains the method of choice for transfer learning with neural networks. Guo *et al.* [10] propose to adaptively fine-tune pre-trained models on a per-instance basis. Wortsman *et al.* [11] and Liu *et al.* [12] propose methods to merge knowledge from multiple pre-trained models. Our method can be seen as performing neural architecture searches in the search space defined by ResNet152 [4], which allows *reusing* its pre-trained parameters.

Model pruning is one way to reduce the storage/computing resource requirements of neural networks. It removes redundant parts (parameters, channels, etc.) that do not significantly contribute to the performance. Model pruning can happen at different granularities, depending on the topology constraints of the removed parts [13–17]. The *reduction* step of our methodology (Section 3.2) can be categorized as a coarse-grained (block-wise) pruning approach, as opposed to finer-grained (element/channel-wise) pruning approaches. We compare our *reduction* step and finer-grained pruning approaches in Section 4.3.1.

Ensemble methods [18] combine multiple models to improve generalization and robustness. Neural network ensemble methods include bagging [19], Snapshot Ensemble [20], and Fast Geometric Ensemble [21]. Several works [22–25] investigate tree-structured neural networks. The regularization technique dropout [26] can also be interpreted as an implicit ensemble technique. We use ensembling to *recycle* parts of the pre-trained network in Section 3.3.

This paper introduces a method to enhance the compactness and efficiency of neural network architectures. It highlights the importance of considering the recycling of pre-trained models. The paper is not limited to merely

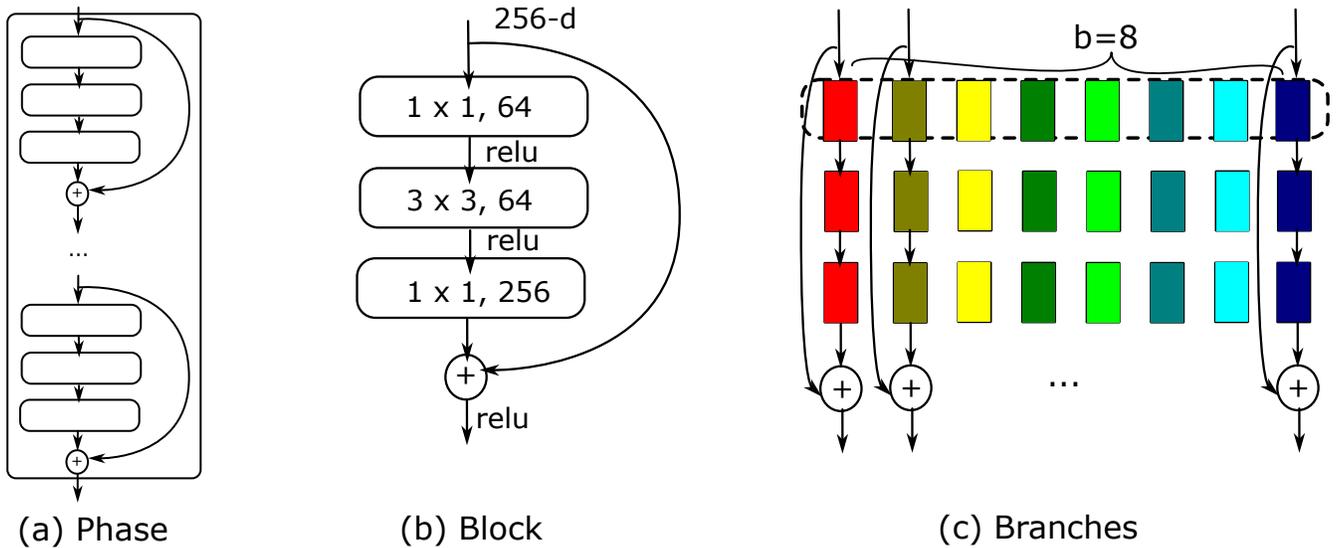


Figure 2: Illustration of phase, block, and branches. (a) A phase represents a group of blocks. (b) A block, also called bottleneck block, stacks 3 convolutional layers with a skip-connection. (c) A block can be split into b branches.

proposing a model compression technique but also emphasizes how efficiently one could reuse existing knowledge in pre-trained models.

3 The RRR Principle for Image Classification

When creating an artifact based on the RRR principle [8], one selects an object of interest for *reuse*, then *reduces* its complexity, and finally *recycles* parts of it for some innovative modification. Adopting the RRR principle, our suggestion to arrive at a high-quality neural network for image classification is as follows:

- **Reuse:** Due to its great performance and simplicity, we choose a ResNet152 pre-trained on ImageNet [6] as the basis (backbone).
- **Reduction:** We reduce the backbone to its basic version by eliminating potentially dispensable blocks.
- **Recycling:** We keep the first few blocks (stump) as the feature extractor and split the remaining blocks into multiple branches to create a voting ensemble.

We next describe each of these points in more detail.

3.1 Reuse: Description of the pre-trained backbone

We reuse a ResNet152 [4] model pre-trained on ImageNet [6]. This model consists of five groups of blocks, which we call *phases* (Figure 1): conv1, conv2_x, conv3_x, conv4_x, conv5_x. The conv1 phase contains a convolutional layer and a max-pooling layer; this can be seen as a special block. All other phases include multiple *bottleneck blocks* (Figure 2 (a)). A bottleneck block (Figure 2 (b)) consists of 3 convolutional layers and a skip-connection path (directly connecting input and output). All bottleneck blocks within a phase are identical, with the exception of the first one, which includes an additional convolutional layer in its skip-connection path. Feature map downsampling is performed by conv1, conv3_1, conv4_1, and conv5_1. As seen in Figure 1, ResNet152 repeats the same blocks many times in each phase.

While our study focuses on ResNet152, the reduction step described in Section 3.2 can, in principle, also be applied to other ResNet architectures. The important properties are the skip-connections and that we have phases consisting of identical blocks. In this case, we can *configure* the phases by deciding upon the number of blocks they contain; the initial ResNet152 serves as a template.

3.2 Reducing: Model block pruning

The ResNet architecture has the benefit of a modular structure consisting of identical blocks with skip-connections. It allows us to perform a neural architecture search that aims to find the optimal number of blocks in each phase. Since the goal here is to *reduce* the network, the maximum number of blocks in each phase is determined by the original ResNet152 architecture. To ease the notation, we will name the architectures as ResNet_ x_1 - x_2 - x_3 - x_4 , where x_i is the number of blocks used in phase i , the conv1 phase is phase 0. Following this notation, the original ResNet152 is written as ResNet_3.8_36_3; the smallest possible network in this regime is ResNet_1.1_1.1, which contains only one block per phase.

For ResNet152, the search space consists of 2592 possible architectures. Extensively evaluating all of them once on one dataset could take several weeks using GPU, which is computationally expensive. Thus, we resort to a greedy forward selection approach to narrow down the search space. This approach, termed forward block selection v1, starts with the simplest architecture ResNet_1.1_1.1 and successively adds one block at a time until no improvement can be observed. Within this logic, we fill one phase before moving to the next phase until we get ResNet152 (ResNet_3.8_36.3). The pseudocode of the procedure is given in Appendix A (Algorithm 1). This method reuses parameters pre-trained on ImageNet as initialization.

An alternative approach would be to add blocks in a cyclic way *i.e.*, always add the block to the phase with the fewest blocks and where blocks may still be added. However, we found in preliminary experiments that these two approaches lead to the same results, so we do not present results for different approaches.

3.3 Recycling: Ensembling network branches

One way to recycle a given pre-trained backbone is to split a part of it into an *ensemble of branches*. It means, starting from a certain layer, partitioning convolutional filters and adjusting layer connections so that all subsequent layers are influenced by exactly one of the filter sets in the partition. It creates one sub-network (branch) for each filter set in the partition (Figure 2 (c)). Each branch ends with a branch-specific output layer with softmax activation. The branches are merged by averaging the probability distributions predicted by different branches. In this sense, the branches can be seen as ensemble members whose votes are aggregated with an average.

We seek to do the branching in such a way that the overall model size and FLOPs (multiply-adds) are preserved. Each branch is constructed by adjusting the number of output channels of convolutional layers. For each branch, the number of output channels C_b is computed as follows:

$$C_b = \left\lfloor \frac{C_o}{\sqrt{b}} \right\rfloor - a \quad (1)$$

where C_o is the number of output channels in the original pre-trained convolutional layer, $\lfloor \cdot \rfloor$ is the floor function, and b is the number of branches to create. The variable a in Equation 1 is useful to ensure that the total number of parameters in the branches is reduced or equal to the original model’s parameter count. The value of a relies on: the number of branches b , the starting point from which to split the model, the number of classes of the downstream task, and the model structure (*e.g.*, ResNet_1.1_1.1 or ResNet152). a can be empirically computed by finding the smallest value that satisfies the total parameter count constraint. This computation results in each branch having C_b output channels as shown in Figure 1 (“Ours” columns). Each branch is initialized with a part of pre-trained kernels (pseudocode is given in Appendix B Algorithm 2).

In our design, we keep the conv1, conv2_x, and conv3_x phases as the shared stump (preprocessor), and we create the branches starting from conv4.1 *i.e.*, the first block of the conv4_x phase. This way, conv4_x and conv5_x effectively constitute an ensemble of independent sub-networks (branches). The reason for branching at the last two phases is that these two phases include the most number of parameters and output channels, allowing for more branches to be made. By maintaining the total parameter count, the more branches are created, the fewer parameters each branch will have.

There can be many methods to train this ensemble of branches. One method is to train branches as if they were individual models without using specific ensemble training techniques. We call this training approach **naive ensemble**. Other choices include bagging [19], Snapshot Ensemble [20] and Fast Geometric Ensemble [21]. All ensemble training methods can be accompanied by random data augmentation to create diverse ensemble members. The stump (conv1, conv2_x, conv3_x) shared by branches is frozen; it keeps the pre-trained parameters.

4 Experiments

We carry out comprehensive experiments in this section. We evaluate our methodology’s reduction step compared to finer-grained pruning approaches. We perform model selection and evaluation for our methodology’s reduction and recycling steps. Finally, we demonstrate the performance of the proposed model on a large benchmark. We also report results from a multi-criteria comparison, including accuracy, inference time, and the number of parameters.

4.1 Datasets

We conduct experiments on Meta-Album micro [9], which is a large image classification benchmark of 40 datasets. These datasets come from various domains such as ecology (fauna and flora), manufacturing (textures, vehicles), human actions, and optical character recognition. The datasets contain images with different scales, ranging from microscopic to human-sized, to images taken from remote sensing. These images represent a diverse and challenging set of data for our models to learn. Each dataset contains 128×128 images, and the goal is to classify each image into one of 20 different classes. Each class in each dataset has 20 images for the training and 20 images for the testing (*i.e.*, both the training set and the test set contain 400 images each). The choice of using this benchmark for our experiments is due to its diversity of problems and because it demonstrates low-resource scenarios (*i.e.*, few training examples available). These scenarios are common in real-world applications and highlight challenges faced when developing learning models with limited data.

4.2 Implementation Details

We formatted an extra held-out dataset to perform model selection and hyper-parameter validation: we carved out a dataset from an OCR dataset of images of alphanumeric characters in the wild [28]. This dataset is dimensioned similarly to the Meta-Album benchmark. It has 20 classes, 15 images per class for training, 5 images per class for validation, and 20 images per class for the test. We call this dataset ICDAR-micro (named after its original source).

In the spirit of Automated Machine Learning, we optimize all the hyper-parameters of our model with the ICDAR-micro dataset.¹ This allows us to use a simple train/test split when we evaluate our method on the Meta-Album benchmark since we do not need validation splits to select any setting. The identified hyper-parameter values are summarized in Appendix C.

We also used the ICDAR-micro dataset to tune the reduction and recycling procedure for our low-resource regime: (1) how much pruning of the original backbone we could do (to reduce inference time and storage); (2) into how many branches we should split the pre-trained backbone (to gain performance by ensembling). These experiments are detailed in Section 4.3.2 and 4.4.

All experimental results are averaged over repeated runs. 95% confidence intervals are computed over repeated runs using t-distribution.

4.3 Results from the reduction procedure

In this section, we summarize two results observed in the reduction procedure. In Section 4.3.1, we compare the inference speed of our methodology’s *reduction* step with finer-grained pruning approaches that could be applied to reduce the network. Second, we study the extent to which we could reduce the network without significantly degrading performance on the ICDAR-micro dataset in Section 4.3.2.

4.3.1 Comparing pruning granularities

This section investigates the potential gain in hardware inference speed of different pruning granularities; here, we ignore the classification performance. We compare 3 levels of pruning granularity: (1) element-wise pruning, where individual parameters are removed; (2) channel-wise pruning, where channels in a convolutional layer are removed; (3) block-wise pruning, where one ResNet block (Figure 2, one block contains three layers) is removed at a time; the latter is our approach. Element-wise pruning implementation is the official pruning package of PyTorch [29], where pruned individual parameters are set to zero. Channel-wise pruning is the Torch-Pruning implementation [30], where parameter tensors are effectively slimmed to remove pruned channels. Block-wise pruning skips pruned blocks.

¹Admittedly, we could have used more datasets, but, as it turns out, we already obtained quite good results with this strategy.

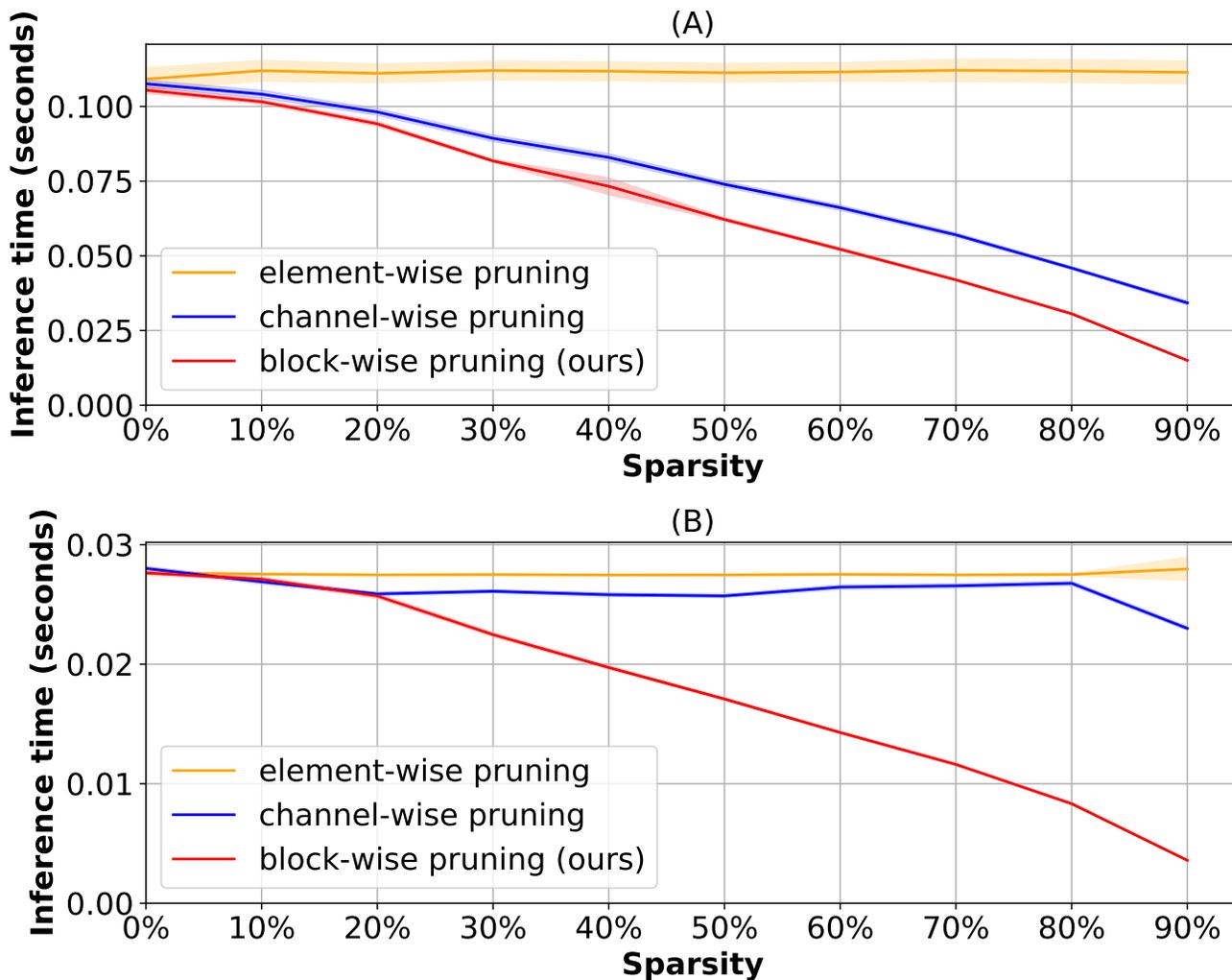


Figure 3: **Inference time of the pruned model.** The upper figure (A) shows the evaluation on CPU; the lower figure (B) shows the evaluation on GPU. Vertical axes show the number of seconds required to run one forward pass for one image; the lower, the better. Horizontal axes denote sparsity. 0% means no pruning, 90% means 90% of parameters are removed. Curves and 95% confidence interval are computed with at least 200 independent runs.

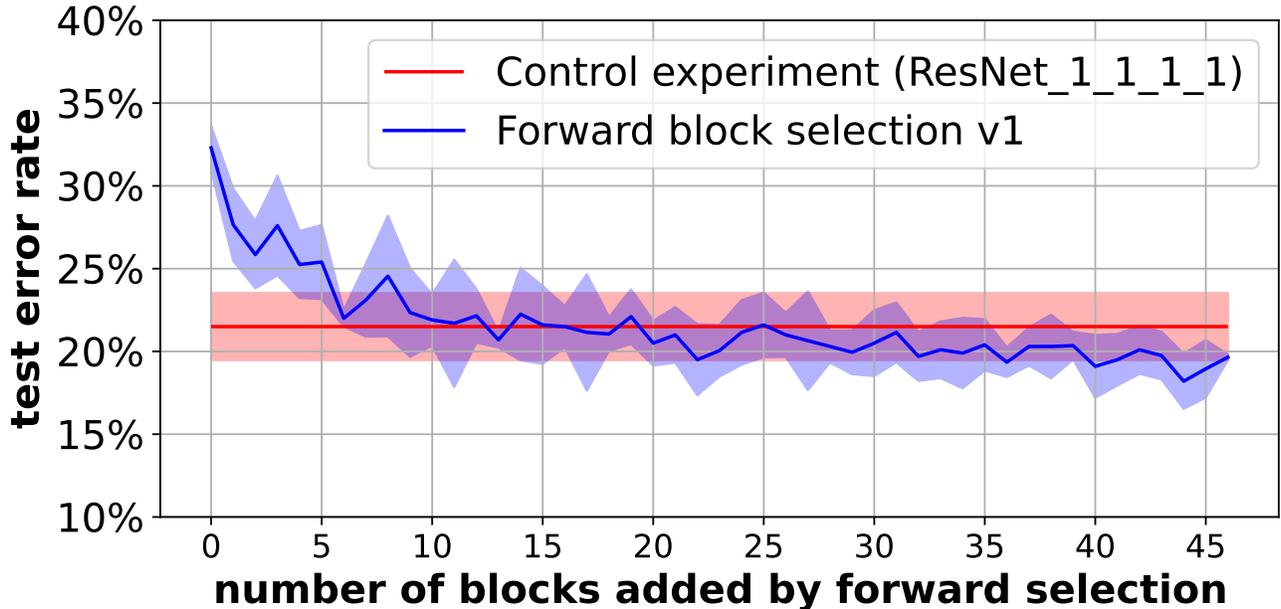


Figure 4: **Optimizing the number of blocks.** Experiments on ICDAR-micro data, applying the forward selection algorithm presented in Appendix A Algorithm 1 (blue curve). The red horizontal line indicates the final performance of the control experiment. The vertical axis denotes the error rate (1 - accuracy) on the test set; the lower, the better. Curves and 95% confidence intervals are computed with 5 independent runs (5 random seeds).

We compare all pruning approaches on the same backbone network: ResNet152 [4]. To measure the potential gain in hardware inference speed, we report inference time in seconds, both on CPU (Intel Xeon Gold 6126) and GPU (GeForce RTX 2080 Ti), across different sparsity levels (Figure 3). Sparsity is the fraction of parameters that are removed compared to the full ResNet152. In our experiments, individual parameters or convolutional channels are removed randomly for element-wise pruning and channel-wise pruning, and the sparsity is approximately uniformly distributed among layers; block-wise pruning removes blocks following the block order (Appendix A Algorithm 1).

The results show that element-wise pruning (yellow curves in Figure 3) provides limited speedup, which is expected because the number of parameters does not effectively change, tensor sparsity is not exploited; curve fluctuation corresponds to noise in evaluation. Channel-wise pruning (blue curves in Figure 3) provides limited speedup on GPU but effectively accelerates the inference on CPU. Our block-wise pruning strategy (red curves in Figure 3) results in the most significant computational gain on both CPU and GPU, particularly when sparsity increases. The advantage of block-wise pruning over channel-wise pruning and the poor result of channel-wise pruning on GPU can be explained by the use of parallel computing in modern CPUs and GPUs. The computation within each layer is parallelized because it involves matrix multiplications [31, 32]. The effect of channel-wise pruning boils down to reducing the parameter tensor size within each layer. Its benefit diminishes when there is more parallelism in the hardware processor, which is the case of GPUs [33, 34]. On the other hand, ResNet152, like many other modern neural networks, consists of sequentially concatenated layers. Computation between layers is blocking; the execution of one layer needs to wait for the result of the previous layer. Removing blocks (hence layers) reduces the number of sequentially blocking computations.

4.3.2 Reducing the number of blocks

We ran the forward selection algorithm, presented in Appendix A (Algorithm 1), on ICDAR-micro to determine the smallest number of blocks we could keep in our low-resource regime without significantly degrading performance. The results are shown in Figure 4. The blue curve is the learning curve of the forward selection algorithm, where each point in the curve is obtained by training the candidate architecture for 300 epochs. In this plot, the stopping criterion is deactivated, so the forward selection algorithm added 47 blocks in the end, which corresponds to 14100

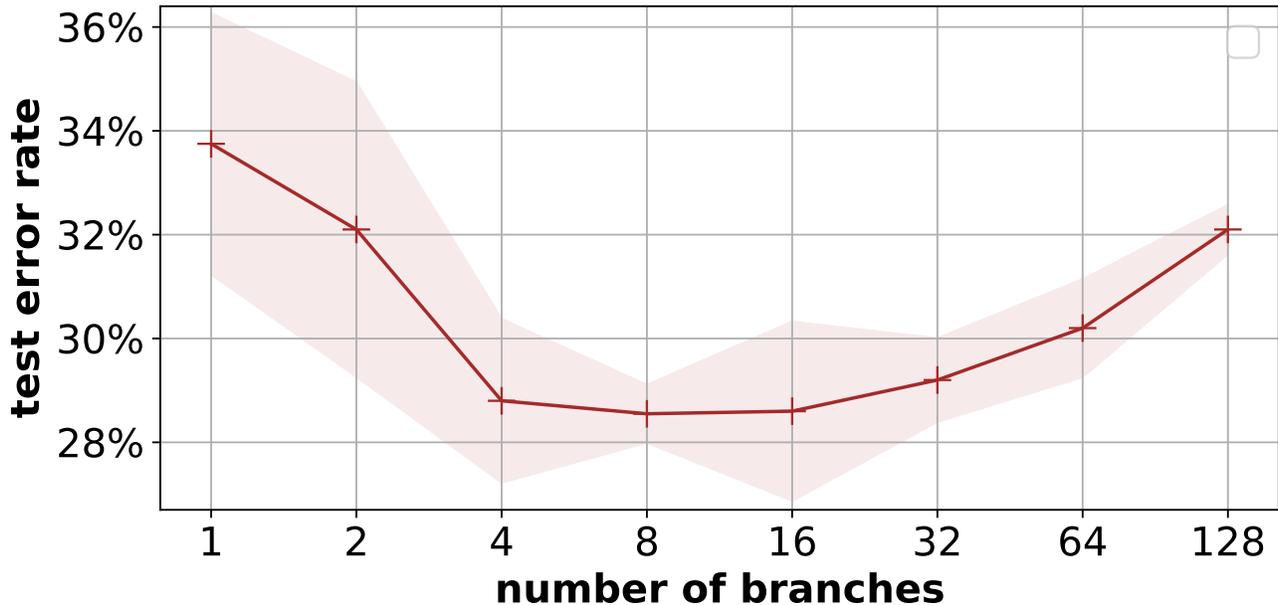


Figure 5: **Optimizing the number of branches.** Experiments on IDCAR-micro data, varying the number of branches. Branches are trained by the naive ensemble approach. In this plot, the number of branches is doubled each time, starting from 1 (ResNet_{1.1.1.1} without splitting) up to 128. The optimum (lowest error rate) is at 8 branches. Curves and 95% confidence intervals are computed with 5 independent runs (5 random seeds).

epochs in total. As a control experiment, we compare the performance of the forward selection algorithm with training ResNet_{1.1.1.1} for the same overall budget of epochs; the red horizontal line shows the final performance of training ResNet_{1.1.1.1} for 14100 epochs. We observe that the final point of the blue curve does not significantly improve upon the red horizontal line.

Hence, these results showed that the model ResNet_{1.1.1.1} is sufficient for the given type of tasks and budget, and adding more blocks does not result in significant improvement. Therefore, the recycling/ensembling technique will be applied to ResNet_{1.1.1.1} in subsequent experiments.

4.4 Results from the recycling procedure (varying the number of branches)

Since our approach preserves the total parameter count in the Recycling step, the number of branches provides a trade-off between the number of ensemble members and the capacity of each member. We used ICDAR-micro again to determine an optimal number of branches for our low-resource regime.

In Figure 5, we vary the number of branches recycled from our backbone (trained by the naive ensemble approach). The results show that “8 branches” is optimal (performances are quite insensitive to the exact number of branches around that point). In what follows, we use ResNet_{1.1.1.1} with 8 branches (splitting starts from conv4.1), denoted as ResNet_{1.1.1.1}-8-branch or RRR-Net.

We also tried alternative ensembling methods, including bagging [19], Snapshot Ensemble [20], and Fast Geometric Ensemble (FGE) [21], to train branches on the ICDAR-micro data. However, the experiments showed that the naive ensemble approach leads to better performances than the alternative ensembling methods. For this reason, we adopt the naive ensemble approach to train branches.

4.5 Results on the Meta-Album benchmark

The results on the Meta-Album benchmark [9] are shown in Figure 6. We compare 5 models:

- [A] blue Training the last layer of the pre-trained ResNet152

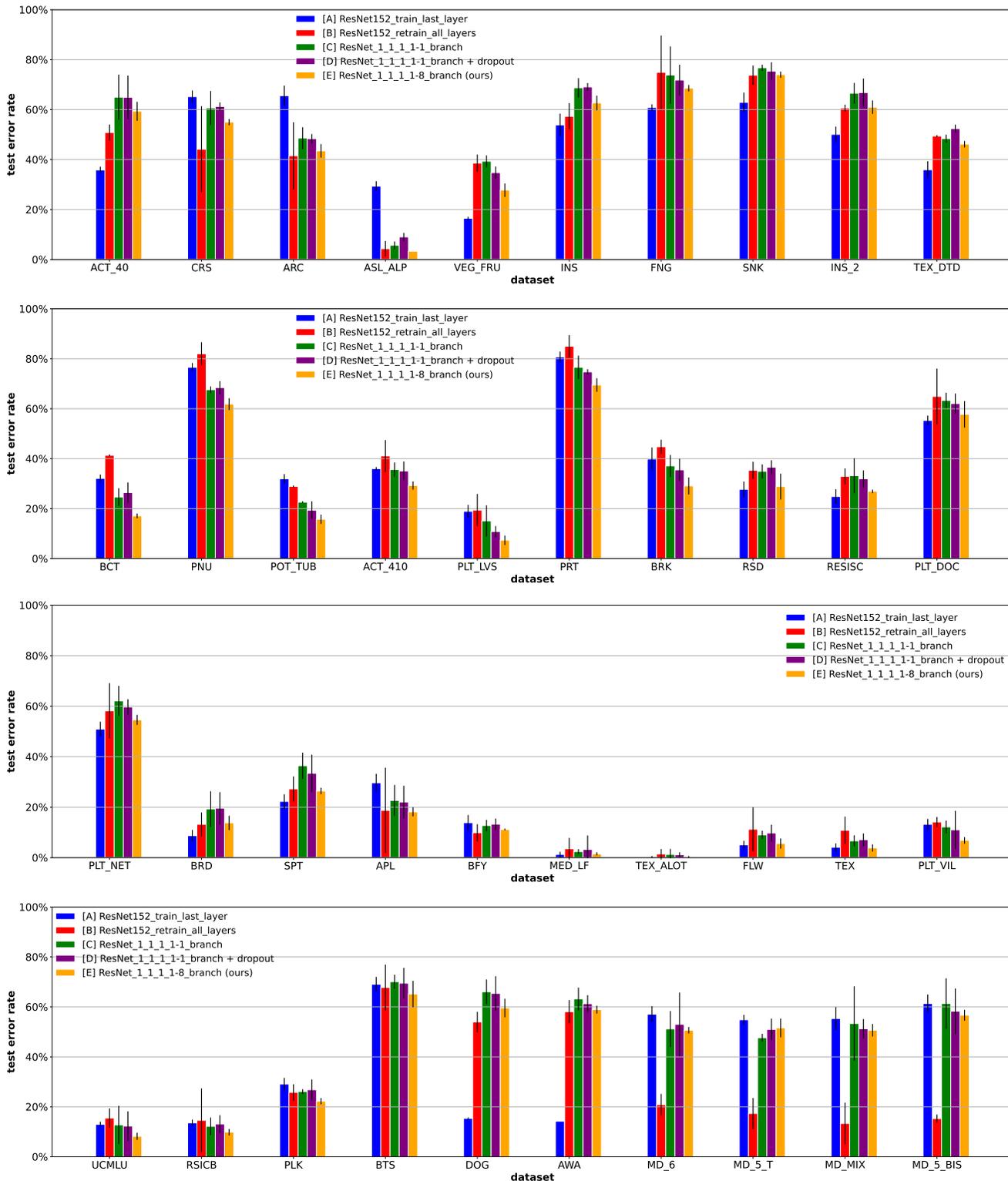


Figure 6: **Results on the 40 datasets of Meta-Album micro.** The vertical axis is the error rate on the test set; the lower, the better. Five models are shown for each dataset: [A] training the last layer (classification head) of the pre-trained ResNet152; [B] retraining all layers of the pre-trained ResNet152; [C] ResNet.1.1.1.1 without splitting; [D] ResNet.1.1.1.1 without splitting but we apply dropout on the last two blocks; [E] ResNet.1.1.1.1-8_branch (RRR-Net) trained by the naive ensemble approach. The 95% confidence intervals are computed with 3 independent runs (3 random seeds).

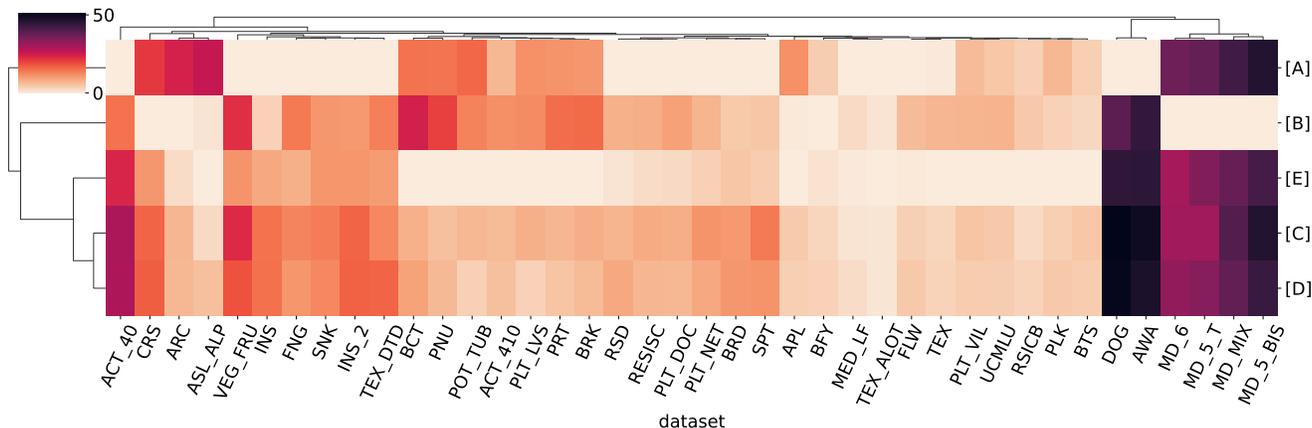


Figure 7: **Hierarchically-clustered heatmap of the model-dataset matrix.** Each column is one dataset. Each row is one model, [E] is our proposed model. The heatmap values are the error rate difference with respect to the best-performing model on the same dataset; the lower, the better.

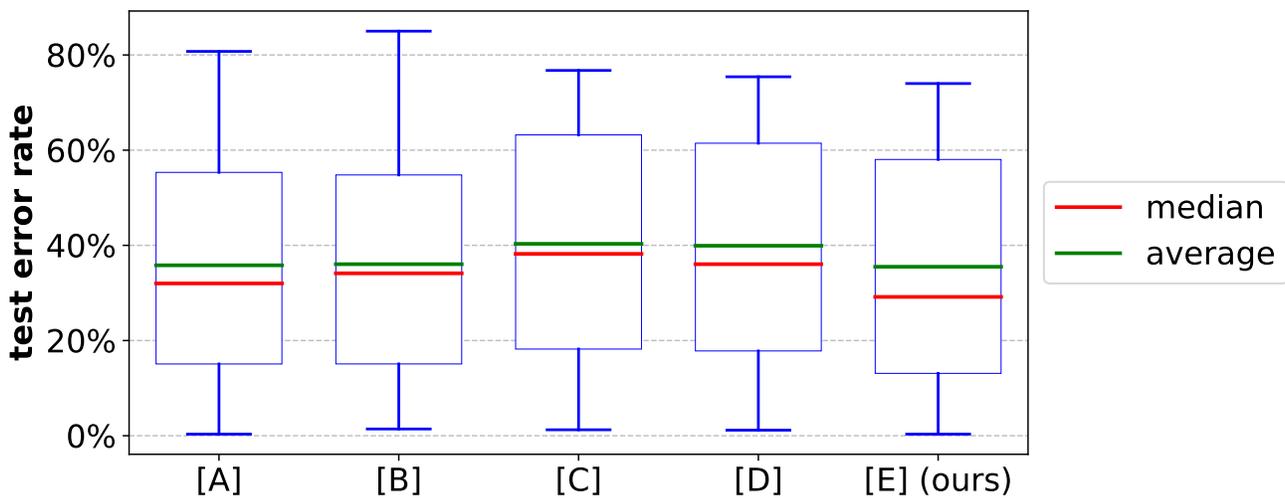


Figure 8: **Box plot of test error rates for each model.** The vertical axis shows the error rate; the lower, the better. Our model [E] has the best average, median, 25% quantile, and worst-case (upper whiskers) error rates. Baselines [A] and [B] have better 75% quantiles.

- [B] red Retraining all layers of the pre-trained ResNet152
- [C] green ResNet_1.1.1.1 without splitting
- [D] purple ResNet_1.1.1.1 without splitting but we apply dropout [26] on the last two blocks
- [E] yellow ResNet_1.1.1.1-8.branch (RRR-Net) trained by the naive ensemble approach.

We use models [A] and [B] as baselines. Depending on the distribution shift between the pre-training dataset (ImageNet) and the target dataset, models [A] and [B] can have different relative rankings. Model [E] is our proposed model using ensembling with 8 branches. Models [C] and [D] are controls for [E]. Model [C] is the reduced network without splitting and ensembling; it only has one branch. Model [D] replaces the recycling step of model [E] with dropout [26]: in the last two blocks, we inserted dropout after each activation function (except for the softmax activation). Of several tried dropout rates (0.05, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6), we report the most favorable result for model [D] (dropout rate = 0.05).

The Meta-Album benchmark [9] is challenging since it covers a wide diversity of domains and scales. As shown in Figure 7, the ranking of different models varies with respect to each dataset. None of the models consistently outperforms all others in all datasets. For example, the overall best baseline model [A] failed on datasets such as ARC and ASL_ALP. The degree of similarity between the target dataset and the pre-training dataset (ImageNet) affects the models’ relative performance. This pattern is particularly visible for datasets of macroscopic animals (DOG, AWA), for which last-layer fine-tuning ([A]) clearly outperforms the other models. Pre-trained features are readily available for these datasets, except for the last layer. On the other hand, the character recognition datasets (MD_6, MD_5_T, MD_MIX, MD_5_BIS) are very dissimilar to ImageNet; retraining all layers ([B]) clearly outperforms the other models in these character recognition datasets. In contrast, the proposed model [E] outperforms the other models on a wide variety of datasets, including datasets that are quite different from ImageNet *e.g.*, BCT, PNU, and POT_TUB from the microscopy domain; BRK, TEX, and TEX_ALOT from the texture domain; UCMLU and RSICB from the remote sensing domain.

The diversity of the Meta-Album benchmark [9] made us consider the overall performance of the models under examination. We calculated the average error rates of these 5 models by averaging over the 40 datasets in Meta-Album. The average error rates for these 5 models are 35.8%, 36.0%, 40.3%, 39.9%, 35.5%. Similarly, the median error rates of these 5 models are 32.0%, 34.1%, 38.2%, 36.0%, 29.2%. Our proposed model [E] is found to have the best average and median error rates, as shown in Figure 8. To further validate these findings, we conducted Wilcoxon signed-rank tests (one-tailed) [35]. These statistical tests indicated that model [E] significantly outperforms model [D] with a p-value of 3×10^{-12} and significantly outperforms model [C] with a p-value of 8×10^{-11} . Model [E] is also found to be significantly better than model [B] with a p-value of 0.036. On the other hand, model [E] performs comparably well to model [A] with a p-value of 0.16. These results indicate that the proposed model [E] performs well on this benchmark with 40 datasets.

Furthermore, we compared the overall best baseline model [A] and our proposed model [E] with respect to five evaluation criteria. These criteria include (1) the average error rate across all 40 datasets in Meta-Album; (2) the number of parameters; (3) the FLOPs (multiply-adds), which indicate the theoretical inference speed; (4) the inference time on CPU; and (5) the inference time on GPU. We represent the comparison as a spider graph depicted in Figure 9. Here the baseline error rate (35.8%) corresponds to the average of the blue bars ([A]) in Figure 6, which is the strongest baseline on this benchmark. “ours” refers to model [E] ResNet_1.1.1.1-8.branch, whose error rate (35.5%) corresponds to the average of the yellow bars ([E]) in Figure 6. We observe that model [E] uses significantly fewer parameters and FLOPs (multiply-adds) and achieves faster inference speed on both CPU and GPU. In summary, Figure 9 supports the advantage of our model; it indicates that our model ([E]) performs similarly well to the overall best baseline while being significantly more frugal.

5 Conclusion

In this paper, we studied the benefits of adjusting pre-trained neural network architectures for image classification in terms of compactness and efficiency. While focusing on the ResNet152 architecture, the paper goes beyond just proposing a model compression technique. It highlights the efficient utilization of pre-trained model knowledge following a “Reuse, Reduce, and Recycle” principle detailed in Section 3. We conducted experiments on a large benchmark of 40 image classification datasets from various domains. We empirically demonstrated that (1) Reducing a pre-trained ResNet backbone to a basic 5-block version yields significant savings in resources (computation and storage) without much loss in classification accuracy, and (2) Recycling the last two blocks by splitting them into

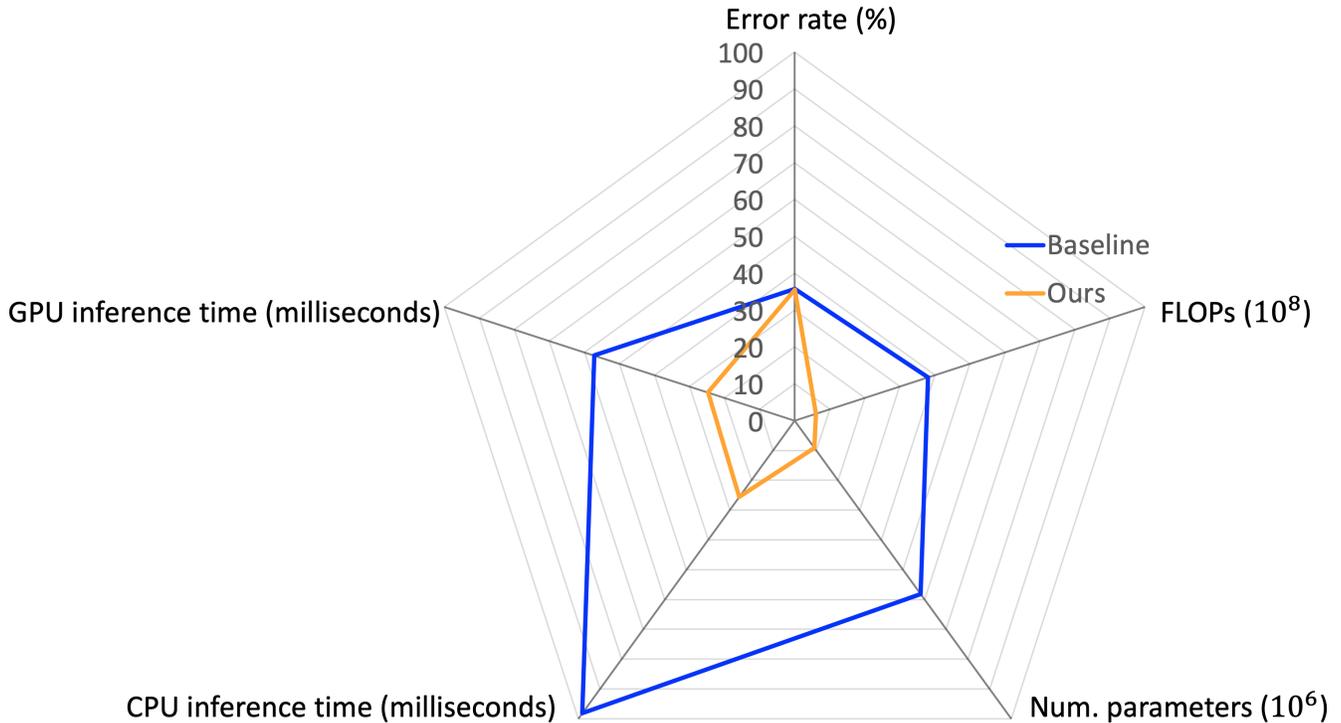


Figure 9: **Multi-criteria comparison.** Our model [E] performs similarly to the overall best baseline [A] while being significantly smaller and faster. Hardware inference time (for one image) is measured with Intel Xeon Gold 6126 and GeForce RTX 2080 Ti.

multiple branches improves performance, while preserving the same amount of computation and storage. Overall, the resulting model matches the performance of the overall best baseline (if not better) while being smaller and faster. The successful results obtained with the ResNet152 starting backbone suggest that the proposed techniques could be applied to other types of modular architectures. Future work might also include improving the ensemble methods, which could possibly lead to better accuracy. The ensemble nature of the proposed recycling technique lends itself to predictive uncertainty computation.

Acknowledgments

We gratefully acknowledge constructive feedback from Dustin Carrión-Ojeda and Romain Egele. This work was supported by the TAU team and the ANR (Agence Nationale de la Recherche, National Agency for Research) under AI chair of excellence HUMANIA, grant number ANR-19-CHIA-0022.

References

- [1] S.J. Pan and Q. Yang. A Survey on Transfer Learning. *IEEE Transactions on Knowledge and Data Engineering*, 22(10):1345–1359, 2010.
- [2] G. E. Hinton and R. R. Salakhutdinov. Reducing the Dimensionality of Data with Neural Networks. *Science*, 313(5786):504–507, 2006.
- [3] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems*, volume 25, 2012.
- [4] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.

- [5] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. *arXiv preprint arXiv:1605.07146*, 2016.
- [6] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. ImageNet Large Scale Visual Recognition Challenge. In *International Journal of Computer Vision*, volume 115, pages 211–252. Springer, 2015.
- [7] Lorenzo Brigato and Luca Iocchi. A close look at deep learning with small data. In *2020 25th International Conference on Pattern Recognition (ICPR)*, pages 2490–2497. IEEE, 2021.
- [8] Bernard Koch, Emily Denton, Alex Hanna, and Jacob Gates Foster. Reduced, reused and recycled: The life of a dataset in machine learning research. In *Thirty-Fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*, 2021.
- [9] Ihsan Ullah, Dustin Carrion, Sergio Escalera, Isabelle M Guyon, Mike Huisman, Felix Mohr, Jan N. van Rijn, Haozhe Sun, Joaquin Vanschoren, and Phan Anh Vu. Meta-album: Multi-domain meta-dataset for few-shot image classification. In *Thirty-Sixth Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2022.
- [10] Yunhui Guo, Honghui Shi, Abhishek Kumar, Kristen Grauman, Tajana Rosing, and Rogerio Feris. Spottune: Transfer learning through adaptive fine-tuning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4805–4814, 2019.
- [11] Mitchell Wortsman, Gabriel Ilharco, Samir Yitzhak Gadre, Rebecca Roelofs, Raphael Gontijo-Lopes, Ari S. Morcos, Hongseok Namkoong, Ali Farhadi, Yair Carmon, Simon Kornblith, and Ludwig Schmidt. Model soups: Averaging weights of multiple fine-tuned models improves accuracy without increasing inference time. *arXiv:2203.05482 [cs]*, March 2022.
- [12] Iou-Jen Liu, Jian Peng, and Alexander Schwing. Knowledge flow: Improve upon your teachers. In *International Conference on Learning Representations*, 2019.
- [13] Yann LeCun, John Denker, and Sara Solla. Optimal brain damage. *Advances in neural information processing systems*, 2, 1989.
- [14] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 28. Curran Associates, Inc., 2015.
- [15] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Learning Structured Sparsity in Deep Neural Networks. In *Advances in Neural Information Processing Systems*, volume 29. Curran Associates, Inc., 2016.
- [16] Zhuang Liu, Jianguo Li, Zhiqiang Shen, Gao Huang, Shoumeng Yan, and Changshui Zhang. Learning Efficient Convolutional Networks through Network Slimming. In *ICCV*, 2017.
- [17] Hao Li, Hanan Samet, Asim Kadav, Igor Durdanovic, and Hans Peter Graf. Pruning filters for efficient convnets. In *ICLR*, 2017.
- [18] Cheng Ju, Aurélien Bibaut, and Mark van der Laan. The relative performance of ensemble methods with deep convolutional neural networks for image classification. *Journal of Applied Statistics*, 45(15):2800–2818, 2018.
- [19] Leo Breiman. Bagging predictors. *Machine learning*, 24(2):123–140, 1996.
- [20] Gao Huang, Yixuan Li, Geoff Pleiss, Zhuang Liu, John E Hopcroft, and Kilian Q Weinberger. Snapshot ensembles: Train 1, get m for free. In *ICLR*, 2017.
- [21] Timur Garipov, Pavel Izmailov, Dmitrii Podoprikin, Dmitry P Vetrov, and Andrew G Wilson. Loss surfaces, mode connectivity, and fast ensembling of dnns. In *Advances in Neural Information Processing Systems*, volume 31, 2018.
- [22] Changxing Ding and Dacheng Tao. Trunk-branch ensemble convolutional neural networks for video-based face recognition. *IEEE transactions on pattern analysis and machine intelligence*, 40(4):1002–1014, 2017.

- [23] Juyong Kim, Yookoon Park, Gunhee Kim, and Sung Ju Hwang. SplitNet: Learning to Semantically Split Deep Networks for Parameter Reduction and Model Parallelization. In *Proceedings of the 34th International Conference on Machine Learning*, pages 1866–1874. PMLR, July 2017.
- [24] Michael Opitz, Horst Possegger, and Horst Bischof. Efficient model averaging for deep neural networks. In *Asian Conference on Computer Vision*, pages 205–220. Springer, 2016.
- [25] Emad MalekHosseini, Mohsen Hajabdollahi, Nader Karimi, Shadrokh Samavi, and Shahram Shirani. Splitting Convolutional Neural Network Structures for Efficient Inference, February 2020.
- [26] Jonathan Tompson, Ross Goroshin, Arjun Jain, Yann LeCun, and Christoph Bregler. Efficient object localization using convolutional networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 648–656, 2015.
- [27] Ligeng Zhu. Lyken17/pytorch-OpCounter: Count the MACs / FLOPs of your PyTorch model. <https://github.com/Lyken17/pytorch-OpCounter>, 2020.
- [28] S.M. Lucas, A. Panaretos, L. Sosa, A. Tang, S. Wong, and R. Young. ICDAR 2003 robust reading competitions. In *Seventh International Conference on Document Analysis and Recognition, 2003. Proceedings.*, pages 682–687, 2003.
- [29] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d’Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.
- [30] Gongfan Fang. Torch-Pruning, 7 2022.
- [31] Kumar Chellapilla, Sidd Puri, and Patrice Simard. High performance convolutional neural networks for document processing. In *Tenth International Workshop on Frontiers in Handwriting Recognition*. Suvisoft, 2006.
- [32] Yangqing Jia. *Learning Semantic Image Representations at a Large Scale*. PhD thesis, University of California, Berkeley, 2014.
- [33] Cristóbal A. Navarro, Nancy Hitschfeld-Kahler, and Luis Mateu. A survey on parallel computing and its applications in data-parallel problems using GPU architectures. *Communications in Computational Physics*, 15(2):285–329, 2014.
- [34] David Kirk et al. NVIDIA CUDA software and GPU parallel computing architecture. In *ISMM*, volume 7, pages 103–104, 2007.
- [35] Frank Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin*, 1(6):80–83, 1945.

Appendix

A. The block selection Algorithm

Algorithm 1 The forward block selection v1 algorithm for reducing ResNet.

Input: ResNet152, Task={TrainSet, TestSet}

Output: MiniNet

```
1: Initialize
2: MiniNet = ResNet_1.1.1.1 + new_classification_layer
3: MiniNet.train_all_layers(TrainSet, epochNum)
4: Accuracy* = MiniNet.evaluate(TestSet)
5: OldAccuracy = Accuracy*
6: # Traverse phases
7: for  $P = 2 : 5$  do
8:   # Traverse blocks
9:   #  $maxBlock(2) = 3, maxBlock(3) = 8$ 
10:  #  $maxBlock(4) = 36, maxBlock(5) = 3$ 
11:  for  $i = 2 : maxBlock(P)$  do
12:    # Add block  $i$  in phase  $P$ .
13:    MiniNet = MiniNet + convP_i
14:    # Continue training, no re-initialization.
15:    MiniNet.train_all_layers(TrainSet, epochNum)
16:    Accuracy = MiniNet.evaluate(TestSet)
17:    if  $(Accuracy - OldAccuracy) / Accuracy^* \geq \epsilon$  then
18:      OldAccuracy = Accuracy
19:      continue
20:    else
21:      return MiniNet - convP_i
22:    end if
23:  end for
24: end for
```

B. Splitting pre-trained kernels into branches

We describe in Algorithm 2 the splitting process of pre-trained layers into a set of branches. The input to Algorithm 2 is the parameter tensors P of the pre-trained model and the number of desired branches N . The output of this algorithm is the parameter tensors T of the branches. Here tensor indices are assumed to start at 1. At layer k , the shape of the 4-dimensional parameter tensor in a convolutional layer is $(C^{k,out}, C^{k,in}, s^h, s^w)$. $C^{k,out}$ means the number of output channels, $C^{k,in}$ means the number of input channels, s^h and s^w are height and width of convolutional kernels. For the pre-trained model, we add the subscript o ; for the branch, we add the subscript b .

C. Results of Hyper-parameter Tuning

The cross-entropy loss and the AdamW optimizer have been chosen. The learning rate equals 10^{-3} ; the weight decay equals 0.01, the batch size equals 32, and the model parameters are initialized with parameters pre-trained on ImageNet [6] (as opposed to training from scratch). The number of training epochs for each model is 300. Except for the pruning experiments, we perform model exponential moving average during the last 60 epochs for all compared methods (if applicable). Data augmentation includes rotation, translation, scaling, shear, random brightness/contrast/color change, sharpening, image inverting, gaussian noise, motion blur, Jpeg compression, posterization, histogram equalization, and solarization. Two transformations are drawn uniformly at random with replacement and then sequentially applied to each training example.

Algorithm 2 The algorithm describing how to split pre-trained kernels into branches.

Require: P the parameter tensors of the pre-trained model

Require: N denotes the number of branches

```

1: # Traverse pre-trained layers to split
2: for  $k = 1 : \text{size}(P)$  do
3:   # Traverse input channels at layer  $k$ 
4:   for  $x = 1 : C_b^{k,in}$  do
5:     # Compute output channel indices  $I$ 
6:     if  $C_o^{k,out} \geq (C_b^{k,out} \times N)$  then
7:        $I = 1 : (C_b^{k,out} \times N)$ 
8:     else
9:        $I = 1 : C_o^{k,out}$ 
10:      while  $\text{size}(I) < (C_b^{k,out} \times N)$  do
11:         $I = \text{concatenate}(I, \text{shuffled}(1 : C_o^{k,out}))$ 
12:      end while
13:       $I = I[: (C_b^{k,out} \times N)]$ 
14:    end if
15:     $start = 1, end = C_b^{k,out}$ 
16:    for  $i = 1 : N$  do
17:       $T_k[i][:, x, :, :] = P_k[I[start : end], x, :, :]$ 
18:       $start = start + C_b^{k,out}, end = end + C_b^{k,out}$ 
19:    end for
20:  end for
21: end for

```
