

ZipIt! Merging Models from Different Tasks *without Training*

George Stoica* Daniel Bolya* Jakob Bjorner Taylor Hearn Judy Hoffman
 Georgia Tech
 {gstoica3, dbolya, jakob_bjorner, thearn6, judy}@gatech.edu

Abstract

Typical deep visual recognition models are capable of performing the one task they were trained on. In this paper, we tackle the extremely difficult problem of combining completely distinct models with different initializations, each solving a separate task, into one multi-task model **without any additional training**. Prior work in model merging permutes one model to the space of the other then adds them together. While this works for models trained on the same task, we find that this fails to account for the differences in models trained on disjoint tasks. Thus, we introduce “ZipIt！”, a general method for merging two arbitrary models of the same architecture that incorporates two simple strategies. First, in order to account for features that aren’t shared between models, we expand the model merging problem to additionally allow for merging features within each model by defining a general “zip” operation. Second, we add support for partially zipping the models up until a specified layer, naturally creating a multi-head model. We find that these two changes combined account for a staggering 20-60% improvement over prior work, making the merging of models trained on disjoint tasks feasible.

1. Introduction

Ever since AlexNet [33] popularized deep learning in computer vision, the field has thrived under the reign of massive models with an ever increasing number of parameters. A large number of vision problems once considered difficult or impossible are now benchmark tasks: classification with tens of thousands of classes [12, 63, 19], accurate object detection [47, 53, 35], fast instance segmentation [22, 7], realistic image generation [28, 24, 48], and more.

However, while these deep models are extremely powerful, they suffer from a potentially debilitating issue: they can only perform the task they were trained on. If we want to expand an existing model’s capabilities, we run into many potential issues. If we try training the model on an addi-

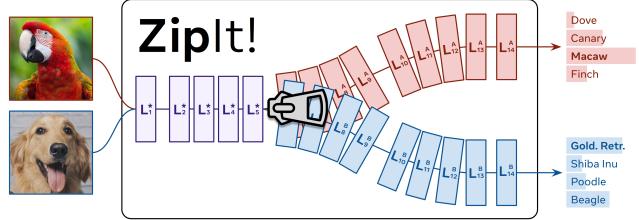


Figure 1: ZipIt! merges models trained on completely separate tasks *without any additional training* by identifying their shared features. Depending on the architecture and task, ZipIt! can nearly match their ensemble performance.

tional task, we face catastrophic forgetting [30, 37, 10]. If we evaluate the same model on different data without adaptation, we often find it doesn’t generalize to out of domain samples [5, 42, 57]. We can try so called “intervention” strategies [57, 10] to mitigate these effects, but these often require further training which can be expensive.

There are, of course, a plethora of existing, carefully tuned models out there for many tasks. But despite these models often sharing the same core architectural backbone (e.g., ResNet-50 [23]), no method yet exists that can easily combine models trained on disjoint tasks. We’re either stuck ensembling them, which requires evaluating each model individually, or jointly training a new model through distillation [34]—both of which can be prohibitively expensive, especially with the modern trend of ever increasing architecture and dataset scales [13, 62, 11]. Instead, it would be nice if we could simply “zip” these models together so that any redundant features between them only need to be computed once, *without any additional training*.

Recently, the idea of combining multiple models into one has started to gain traction in the vision community. Model Soups [59] can add multiple models finetuned from the same pretrained initialization to improve accuracy and robustness. Git Re-Basin [2] generalizes further to models trained on the same data but with different initializations, though with a significant accuracy drop. REPAIR [27] improves on Git Re-Basin by adding new parameters and adjusting model batch norms where applicable. However, all

*Equal Contribution. Code: <https://github.com/gstoica27/ZipIt>.

of these methods only combine models trained on the same task. In this paper, we take this line of work to its logical extreme: merging differently initialized models trained on *completely separate tasks* (see Fig. 2). While this is an incredibly challenging problem, we employ two simple strategies to make it feasible.

First, we note that prior work focuses on *permuting* one model to the other when merging them. This creates a 1-1 mapping between the two models, inherently assuming that most features *across* them are redundant. Since this isn’t necessarily the case for models trained on different tasks, we cannot rely on permutation alone. Instead, we exploit redundancy *within* each model as well. To do this, we generalize model merging to support “zipping” any combination of features *within* and *across* each model. We find that on some datasets, this alone improves accuracy **by up to 20%** vs. Git Re-basin [2] and an even stronger permutation baseline that we implement.

Second, existing methods merge *the entire network*. While this might work for extremely similar models trained in the same setting, the features of models trained on disjoint tasks become increasingly less correlated over the course of the network [31]. To solve this, we introduce *partial zipping*, where we only “zip” up to a specified layer. Afterwards, we feed the merged model’s intermediate outputs to the remaining unmerged layers of the original networks, naturally creating a multi-head model. Depending on the difficulty of each task, this can improve accuracy **by over 15%** while still keeping most of the layers merged.

Incorporating both of these strategies, we introduce ZipIt! (Fig. 1), a general method for “zipping” together any number of models trained on different tasks into a single multitask model *without additional training*. By deriving a general graph-based algorithm for merging and unmerging (Sec. 4), we can zip models of the same architecture together, merge features *within* each model, and partially zip them to create a multi-task model. We verify the effectiveness of our approach by merging models trained on completely disjoint sets of CIFAR [32], and ImageNet [12] categories, as well as merging models trained on completely independent datasets, significantly outperforming prior work in the process (Sec. 5). Finally, we ablate and analyze our method’s capabilities on these scenarios (Sec. 6).

2. Related Work

Model merging combines the weights of two or more models into a single set of weights in a useful way. In this work, we explicitly target models that have been trained on disjoint tasks (Fig. 2), which differs from prior work.

Merging Finetuned Models. If two models are finetuned from the same pretrained checkpoint, they often lie in the same error basin [43]. Several works [25, 26, 55, 60] have

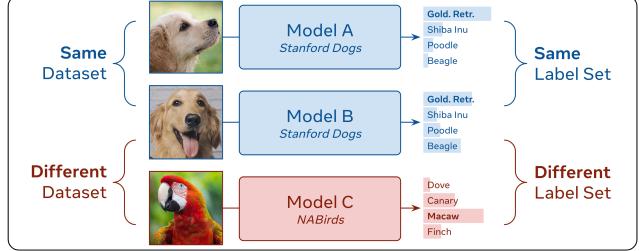


Figure 2: **Our Setting.** Prior work [59, 2, 27] focuses on merging models from the **same dataset** with the **same label sets**: e.g., merging two models both trained to classify dog breeds. In this work, we remove that restriction and “zip” models that can come from **different datasets** and have **different label sets**: e.g., merging a model that classifies dog breeds with one that classifies bird species.

exploited this property to average together the weights of a model while training. [51, 8, 20, 9, 4] use an “exponential moving average” of training checkpoints as a teacher for self-supervised learning. Other works merge models initialized from the same pretrained base, but that were fine-tuned independently, either by simply averaging their weights [41, 59], permuting one model to the other [3, 61, 56], or maximizing some objective [40]. Our setting differs, as we do not assume the same initialization.

Merging Differently Initialized Models. Merging models with different initializations is a much more challenging problem. Works in this space often rely on *mode connectivity* [17, 18, 14, 16], attempting to interpolate between models along a low loss path (e.g., [52, 50, 39]). The most popular approach follows the intuition, later formalized by [15], that models permuted to the same loss basin can be merged by averaging their weights. Most notably, Git Re-Basin [2] permutes models locally, primarily by comparing the similarity between their weights. REPAIR [27] improves the accuracy of Git Re-Basin by instead computing the similarity between their intermediate layer feature activations, and adding several batch norms to the network. [45] finds permutations using global rather than local optimization, though the method doesn’t generalize well to modern architectures. Some of these works (e.g., [50, 2, 39]) evaluate on a setting where each model sees varying numbers of instances per class. And one concurrent work [45] evaluates on a continual learning setting where models are given disjoint categories, but their method requires optimization and does not support skip connections. As far as we are aware, we present the first *general method* to successfully merge models trained on disjoint tasks *without training*.

3. Motivation

Our goal is to merge any models of the same architecture together *without additional training*. Unlike prior work

[59, 2, 27], we specifically target the extremely difficult setting of models that have different initializations and are trained on completely different tasks. In this section, we will motivate the need for a new approach by showing how prior work fails on this challenging setting.

What constitutes a task? The term “task” is often overloaded in machine learning. The broader vision community treats “tasks” as different problem statements (e.g., detection vs. segmentation [38]), while subfields like continual learning [10] define “tasks” as disjoint category splits of the same data. While we would ideally support any definition, we specifically focus on classification in this work. We define tasks as either disjoint category splits of the same data or as classification on different datasets entirely.

3.1. Background

Consider a model \mathcal{L} as a collection of layers $L_i \in \mathcal{L}$, each of which may have some parameters (e.g., W_i, b_i for a linear layer). We denote “merging” two models \mathcal{L}^A and \mathcal{L}^B as combining their parameters into a new model \mathcal{L}^* such that \mathcal{L}^* retains the accuracy of \mathcal{L}^A and \mathcal{L}^B on their original tasks.

If \mathcal{L}^A and \mathcal{L}^B are finetuned from the same checkpoint, several works (e.g., [25, 26, 59]) have found that merging them is as easy as averaging their weights. For instance, if L_i is a linear layer, the new weight matrix W_i^* is simply

$$W_i^* = \frac{1}{2} W_i^A + \frac{1}{2} W_i^B \quad (1)$$

However, if \mathcal{L}^A and \mathcal{L}^B were not finetuned from the same checkpoint, Eq. 1 typically results in random accuracy. To fix this, a line of work (most recently [2, 27]) has found that if you first permute the feature space of one model to align with the feature space of the other model before averaging them together, you can recover much of the lost accuracy. More concretely, let P_i be a permutation matrix that permutes the output of layer L_i^B to the space of L_i^A . Then for each layer, works such as Git Re-Basin [2] apply

$$W_i^* = \frac{1}{2} W_i^A + \frac{1}{2} P_i W_i^B P_{i-1}^T \quad (2)$$

Note that here we permute the output space of W_i^B , but we also need to permute its input space to undo the permutation from the previous layer (hence the use of P_{i-1}^T).

Problems with Permutation. While Eq. 2 works decently well for models trained on the same task, its underlying assumptions break down when the models are trained on *different* tasks. The idea of permutation-based model merging (e.g. Git Re-Basin [2]) stems from mode connectivity [15], where it has been conjectured that models with different initializations trained on the same data lie in the same *loss basin* (i.e., region of low loss or high accuracy) modulo permutation (as most neural networks can be permuted

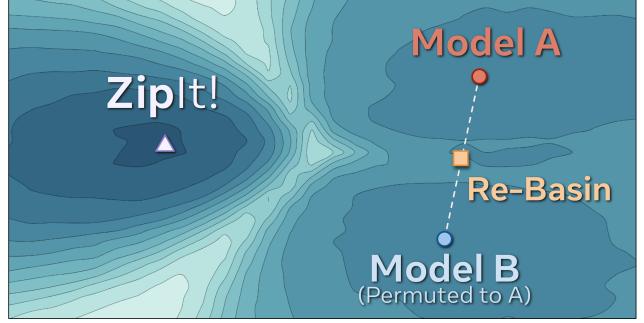


Figure 3: **Prior Work Fails** on merging models trained on *different tasks*. Git Re-Basin [2] assumes the two models lie in same loss basin *modulo permutation* and interpolates between them. However, that is not sufficient when the models are trained on *different tasks*, here shown for disjoint class sets of CIFAR-100. While **A** and the permuted **B** lie in similar basins, Git Re-Basin’s interpolation performs *worse* than the originals. In contrast, our ZipIt! merges them into an even better model in a completely different loss basin.

internally without affecting their outputs). As we show in Fig. 3, this does not hold in our setting where the two models are trained on different tasks. In this case, **Model B**’s optimal permutation lies in a *similar* yet distinct basin to **Model A**. Because the two models are actually in *different basins*, the interpolated result actually has a *lower* accuracy than either of the two original models. This motivates us to explore alternative methods for merging.

4. ZipIt!

In this work, we treat model merging as jointly combining the checkpoints (i.e., collection of weights) of two models into a single checkpoint that can perform all the tasks of its constituents. We accomplish this by merging each layer of one model with the corresponding layer in the other, *while modifying both* (in contrast to permutation-based merging, which only permutes one of the models).

For instance, if layer $L_i \in \mathcal{L}$ is a linear layer, it has parameters $W_i \in \mathbb{R}^{n_i \times m_i}, b_i \in \mathbb{R}^{n_i}$ and takes input $x \in \mathbb{R}^{m_i}$, with an output feature vector $f_i \in \mathbb{R}^{n_i}$ where

$$f_i = L_i(x) = W_i x + b_i \quad (3)$$

Then our goal is to take $L_i^A \in \mathcal{L}^A$ from **model A** and $L_i^B \in \mathcal{L}^B$ from **model B** and merge them into a layer L_i^* that combines their feature spaces such that information from both f_i^A and f_i^B is retained in f_i^* . Note that we consider activation and normalization to be distinct layers in the network as they are implemented in practice, rather than one combined unit.

How do we “combine” feature spaces? In order to construct the combined features f_i^* , we assume that there are

The Zip Operation

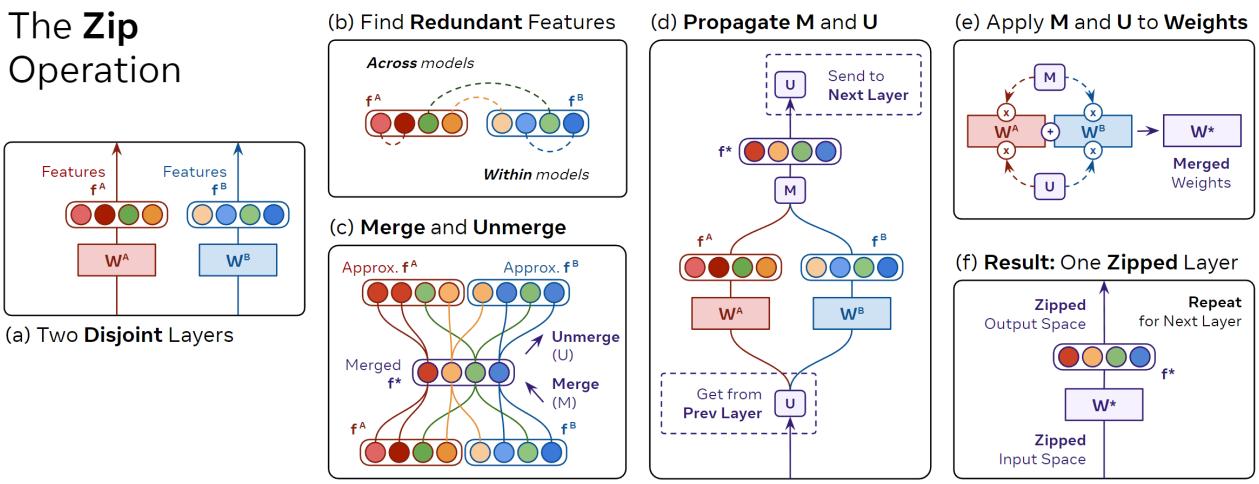


Figure 4: **ZipIt!** merges models layer-wise by exploiting redundancy in their features. (a) Starting with completely disjoint layers with weights W^A and W^B from models trained on different tasks, (b) we match redundant features by comparing their activations f^A and f^B . (c) We use this matching to produce a merge matrix M to combine f^A and f^B into a single shared feature space f^* and a corresponding unmerge matrix U that undoes this operation. (d) In order to align the input space of the next layer, we propagate U forward along network and at the same time receive a U matrix from the previous layer. (e) Once we have both an M for the output, and a U for the input, we can “zip” the layers together by applying Eq. 7. (f) The result is a single layer with a shared input and output space, and we can now repeat from (a) on the next layer.

some redundant features in f_i^A and f_i^B . That is, some elements of the two feature vectors are *highly correlated* over a sample of data. In this work, we consider correlations between features *within* the same model and *across* the two models. In practice we concatenate the two feature vectors into $f_i^A \parallel f_i^B \in \mathbb{R}^{2n_i}$ and consider correlations between each pair of elements in this concatenated vector, which differs from prior work [2, 27] that only consider correlations *across* the two models.

If two features are highly correlated, then we can average them without losing much information. Thus, if we can find a good pairing for each element of the concatenated $f_i^A \parallel f_i^B$ (leaving us with n_i pairs), we can construct a merged feature f_i^* that contains an efficiently compressed representation of f_i^A and f_i^B by averaging each pair of features. In practice, we define a merge matrix $M_i \in \mathbb{R}^{n_i \times 2n_i}$ s.t.

$$f_i^* = M_i (f_i^A \parallel f_i^B) \quad (4)$$

The resulting M_i is zero everywhere except for each pair with index p of matches (j, k) , $M_{i[p,j]} = M_{i[p,k]} = 1/2$. We find these matches greedily—an optimal algorithm exists but is very slow and only slightly more accurate (Tab. 4).

What about the next layer? When computing matches in the following layer, we now have the problem that the next layers, L_{i+1}^A , L_{i+1}^B , are not compatible with this merged representation. Instead, we need to *undo* the merge operation before passing the features to the next layer. Thus, we need to define an “unmerge” matrix $U_i \in \mathbb{R}^{2n_i \times n_i}$ s.t.

$$U_i f_i^* \approx f_i^A \parallel f_i^B \quad (5)$$

In the case of the matching from earlier, U_i is simply $2M_i^T$ and has the effect of “copying” the merged features back to their original locations. Note that in most cases, we can’t have a strict equality here because U_i isn’t full rank.

We can further split this unmerge matrix into $U_i^A, U_i^B \in \mathbb{R}^{n_i \times n_i}$ that act individually to produce f_i^A and f_i^B by splitting U_i in half along its rows. With this, we can evaluate the next layers using the merged features:

$$f_{i+1}^A = L_{i+1}^A (U_i^A f_i^*) \quad f_{i+1}^B = L_{i+1}^B (U_i^B f_i^*) \quad (6)$$

4.1. The “Zip” Operation

We now have all the necessary pieces, and can derive a general operation to merge L_i^A and L_i^B at an arbitrary point in the network (Fig. 4). First, we compute M_i and U_i by matching features between f_i^A and f_i^B . We then pass U_i to the next layer and receive U_{i-1} from the previous layer. Using M_i and U_{i-1} , we can now “fuse” the merge and unmerge operations into the layer’s parameters. For a linear layer:

$$W_i^* = M_i^A W_i^A U_{i-1}^A + M_i^B W_i^B U_{i-1}^B \quad (7)$$

where M_i^A and M_i^B are M_i split in half along its columns. The equation for b_i^* is similar but without the unmerge.

Note the similarity between Eq. 7 and Eq. 2. This isn’t a coincidence: if we only allowed merging *across* models and not *within* models, our “zip” operation would be identical to Git Re-Basin’s permute-then-average approach. Thus, Eq. 7 can be thought of as a generalization of prior work.

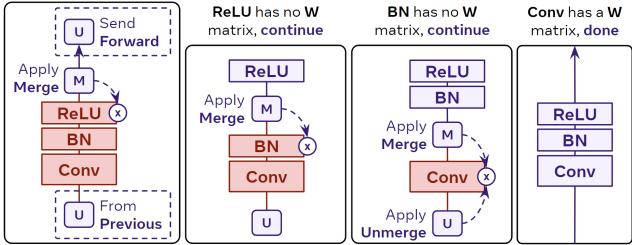


Figure 5: **Zip Propagation.** In practice, we compute M_i and U_i after activations (e.g., ReLU). Since we can’t apply Eq. 7 to a layer without a weight matrix, we have to propagate M_i backward until we hit such a layer, merging element-wise layers (e.g., BatchNorm) along the way.

4.2. Zip Propagation

However, most modern neural networks are not simply collections of linear layers stacked on top of each other. In practice, we cannot combine merge and unmerge matrices into every layer of the network, as a local zip (Eq. 7) expects the layer to have a weight *matrix*—i.e., the layer has to have separate input and output spaces so that we can unmerge the input space and merge the output space. Other layers (e.g., BatchNorm, ReLU) don’t have such a weight matrix.

Thus, we “propagate” M_i and U_i through these layers. For instance, in Fig. 5, we show a common stack of layers found in a typical ConvNet. Following [27], we compute M_i and U_i using the activations of the network (i.e., after each ReLU). We can’t fuse M_i with the ReLU layer, as it doesn’t have any parameters. Similarly, we can merge the parameters of the preceding BatchNorm layer (i.e., in the same way as bias). But it doesn’t have a weight matrix, so we also can’t fuse M_i into it. Only once we’ve reached the Conv layer can we fuse M_i and U_i into it using Eq. 7 (in this case, treating each kernel element as independent).

Similar care needs to be taken with skip connections, as every layer that takes input from or outputs to a skip connection shares the same feature space. However, this too can be dealt with during propagation—we just need to propagate M_i backward and U_i forward to each layer connected by the same skip connection. In general, we can define propagation rules to handle many different types of network modules (see Appendix B).

4.3. Extensions

Partial Zip. In many cases, we don’t want to zip every layer of the two networks, especially if their output spaces are incompatible, or if doing so would lose too much accuracy. In those cases, we can perform a *partial zip*. That is, we zip most of the layers together, but leave some of the later ones *unzipped* (Fig. 6).

Implementing this operation is simple in our framework: zip as normal until the specified layer i , then the remaining

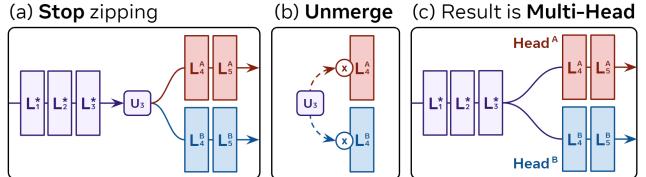


Figure 6: **Partial Zip.** If we stop zipping early (a), we can create a multi-head model that can perform multiple tasks. All we need to do is apply the last unmerge from zip propagation (Fig. 5) to the inputs of the first unmerged layer in each model (b), and we get a model with multiple heads (c).

unzipped layers will receive U_i through zip propagation. If we apply U_i^A to L_{i+1}^A and U_i^B to L_{i+1}^B , the remaining unzipped layers will form “heads” that expect merged features as input. We can then ensemble the heads or choose one to evaluate at runtime.

Repeated Matching (α). In some cases, we’d like to merge more than two models together. To do this, we allow “repeated matches”. That is, when two features are matched in our greedy algorithm, they are removed and replaced with the resulting merged feature. To ensure that one feature doesn’t get merged endlessly, we set the correlations of the new feature to be the minimum of the old features’ similarities weighted by $\alpha \in (0, 1]$. We find a small value of α typically works best.

Same-model Budget (β). To demonstrate the effectiveness of same-model merges, we introduce a “budget” parameter $\beta \in [0, 1]$ that denotes what percent of total merged features can come from models merging within themselves, with each model receiving an equal portion of this budget. Note that a budget of 0 results in Eq. 2, as in that case no features can be merged within models.

5. Results

We devise two types of experiments to benchmark disjoint task model merging (Fig. 2): (1) Merging models trained on disjoint category splits of the same dataset (i.e., *same dataset and different label sets*), and (2) merging models trained on completely different datasets (i.e., *different datasets and label sets*).

Experimental Details. For each experiment where we sample multiple disjoint splits of categories, we hold one split out for hyperparameter search and report mean and standard deviation on the rest. For experiments with models trained on different datasets, we subsample the validation set into a validation and test set to use for the same purpose. To compute activations, we use a portion of the training set for each dataset (see Appendix A). For a fair comparison, we reset the batch norms for *all* methods (including the original models) using the training data (following the recommenda-

Method	FLOPs (G)	Joint Acc (%)	Per-Task (%)		
			Task A	Task B	Avg
Model A	0.68	48.2 \pm 1.0	97.0 \pm 0.6	45.1 \pm 8.6	71.0 \pm 4.4
Model B	0.68	48.4 \pm 3.8	49.1 \pm 9.3	96.1 \pm 1.1	72.6 \pm 4.9
W. Avg (Eq. 1)	0.68	43.0 \pm 1.6	54.1 \pm 1.4	67.5 \pm 1.2	60.8 \pm 4.5
Git Re-Basin [2]	0.68	46.2 \pm 0.8	76.8 \pm 8.9	82.7 \pm 5.1	79.8 \pm 6.5
Permute (Eq. 2)	0.68	58.4 \pm 6.8	86.6 \pm 2.1	87.4 \pm 1.1	87.4 \pm 1.4
ZipIt! _{20/20}	0.68	79.1\pm1.1	92.9\pm1.1	91.2\pm1.4	92.1\pm1.0
Ensemble	1.37	87.4 \pm 2.6	97.0 \pm 0.6	96.1 \pm 1.1	96.6 \pm 0.4
ZipIt! _{13/20}	0.91	83.8\pm3.1	95.1\pm0.7	94.1\pm1.5	94.6\pm0.6

(a) **CIFAR-10 (5+5).** Using ResNet-20 ($4 \times$ width).

Table 1: **CIFAR Results.** ZipIt! vs. baselines on combining a model trained on half the classes (**Task A**) with one trained on the other half (**Task B**) *without extra training*. We report both joint (10/100-way) and per-task (5/50-way) accuracy. ZipIt! significantly outperforms its baseline and closes in on the upper bound (ensemble accuracy).

tion in [27]). For our method, ZipIt!_{n/m} indicates that n out of the m layers in the network have been zipped (Sec. 4.3).

Evaluation. For the setting with disjoint class splits of the same dataset, we evaluate performance in two ways: joint accuracy and per task accuracy. For joint accuracy, we evaluate each model over *all* classes in the combined dataset. For per task accuracy, we compute the accuracy of each task individually (i.e., supposing we had task labels at runtime) and then report the average. The former is similar to a continual learning setting where we want to augment the knowledge of the model, while the latter is akin to a multi-task setting where we know which task we’re using at test time. For the scenario where we merge models trained on different datasets, we use the per task accuracy metric, as the label spaces are not comparable.

Baselines. In addition to Git Re-Basin [2], we compare to two baselines: Weight Averaging (Eq. 1) and Permute (Eq. 2) implemented in our framework (i.e., we set M_i and U_i such that Eq. 7 is equivalent). For Permute, we use linear sum assignment to find optimal permutations (following [36]). Note that our Permute is a *strong* baseline we create and is more accurate than Git Re-Basin in our settings. It’s also similar to REPAIR [27], but without adding extra parameters to the model (which is out of scope for this work).

5.1. CIFAR-10 and CIFAR-100

We train 5 pairs of ResNet-20 [23] from scratch with different initializations on disjoint halves of the CIFAR-10 and CIFAR-100 classes [32]. While ZipIt! supports “partial zipping” to merge models with different outputs (in this case, disjoint label sets), prior methods do not. To make a fair comparison, we train these CIFAR models with a CLIP-style loss [46] using CLIP text encodings of the class names as targets. That way, both models output into the same space, despite predicting different sets of categories. Note that this means the models are capable of some amount of zero-shot classification. Thus, they get better than random

Method	FLOPs (G)	Joint Acc (%)	Per-Task (%)		
			Task A	Task B	Avg
Model A	2.72	41.6 \pm 0.3	82.9 \pm 0.7	24.8 \pm 0.4	53.9 \pm 0.5
Model B	2.72	41.6 \pm 0.2	25.1 \pm 1.2	82.8 \pm 0.2	54.0 \pm 0.6
W. Avg (Eq. 1)	2.72	17.0 \pm 1.7	23.8 \pm 6.9	24.8 \pm 5.9	24.3 \pm 1.9
Git Re-Basin [2]	2.72	40.9 \pm 0.2	57.3 \pm 1.5	56.7 \pm 0.7	57.0 \pm 0.8
Permute (Eq. 2)	2.72	42.8 \pm 0.7	61.6 \pm 1.4	60.5 \pm 0.5	61.0 \pm 0.8
ZipIt! _{20/20}	2.72	54.9\pm0.8	68.2\pm0.8	67.9\pm0.6	68.0\pm0.4
Ensemble	5.45	73.5 \pm 0.4	82.9 \pm 0.7	82.8 \pm 0.2	82.8 \pm 0.4
ZipIt! _{13/20}	3.63	70.2\pm0.4	80.3\pm0.8	80.1\pm0.7	80.2\pm0.6

(b) **CIFAR-100 (50+50).** Using ResNet-20 ($8 \times$ width).

accuracy on tasks they were not trained on.

CIFAR-10 (5+5). In Tab. 1a, we merge models trained on disjoint 5 class subsets of CIFAR-10 using ResNet-20 with a $4 \times$ width multiplier (denoted as ResNet-20 \times 4). In joint classification (i.e., 10-way), Git Re-Basin is unable to perform better than using either of the original models alone, while our Permute baseline performs slightly better. In stark contrast, our ZipIt! performs a staggering 32.9% better than Git Re-Basin and 20.7% better than our baseline. If we allow the last stage of the network to remain unzipped (i.e., zip up to 13 layers), our method obtains 83.8%, which is only 3.6% behind an ensemble of **Model A** and **Model B** (which is practically the upper bound for this setting).

CIFAR-100 (50+50). We find similar results on disjoint 50 class splits of CIFAR-100 in Tab. 1b, this time using an $8 \times$ width multiplier instead. Like with CIFAR-10, Git Re-Basin fails to outperform even the unmerged models themselves in joint classification (i.e., 100-way), and this time Permute is only 1.2% ahead. ZipIt! again significantly outperforms prior work with +14% accuracy over Git Re-Basin for all layers zipped, and a substantial +29.2% if zipping 13/20 layers. At this accuracy, ZipIt!_{13/20} is again only 3.3% behind the ensemble for joint accuracy and 2.6% behind for average per task accuracy, landing itself in an entirely different performance tier compared to prior work.

5.2. ImageNet-1k (200+200)

To test our method on the *much harder* setting of large-scale data, we train 5 differently initialized ResNet-50 models with cross entropy loss on disjoint 200 class subsets of ImageNet-1k [12]. To compare to prior work that doesn’t support partial zipping, we initialize the models with capacity for all 1k classes, but only train each on their subset.

In Tab. 2 we show results on exhaustively merging pairs from the 5 models. To compute joint (i.e., 400-way) accuracy, we softmax over each task’s classes individually (like in [1]), and take the argmax over the combined 400 class

Method	FLOPs (G)	Joint Acc (%)	Per-Task (%)		
			Task A	Task B	Avg
Model A	4.11	37.2 \pm 2.0	74.3 \pm 4.0	0.5 \pm 0.1	37.4 \pm 2.0
Model B	4.11	35.3 \pm 1.6	0.5 \pm 0.1	70.5 \pm 3.2	35.5 \pm 1.6
W. Avg (Eq. 1)	4.11	0.3 \pm 0.1	0.6 \pm 0.1	0.7 \pm 0.1	0.6 \pm 0.1
Git Re-Basin [2]	4.11	3.1 \pm 1.2	5.3 \pm 2.6	5.7 \pm 2.4	5.5 \pm 1.7
Permute (Eq. 2)	4.11	8.6 \pm 5.8	10.1 \pm 4.4	15.3 \pm 11.1	12.7 \pm 7.7
ZipIt! _{50/50}	4.11	8.6 \pm 4.7	12.4 \pm 5.9	14.7 \pm 7.8	13.5 \pm 6.6
Ensemble	8.22	63.3 \pm 4.9	74.3 \pm 4.0	70.5 \pm 3.2	72.4 \pm 2.5
ZipIt! _{22/50}	6.39	55.8 \pm 4.1	65.9 \pm 2.5	64.1 \pm 3.0	65.0 \pm 2.3
ZipIt! _{10/50}	7.43	60.9 \pm 4.1	70.7 \pm 3.0	69.0 \pm 2.9	69.9 \pm 1.9

Table 2: **ImageNet-1k (200+200) Results.** Merging ResNet-50 models trained from scratch on disjoint 200 category subsets (Task A and B) of ImageNet-1k. Prior work performs poorly, but ZipIt! makes this task feasible.

vector. On this extremely difficult task, Git Re-Basin only obtains 3.1% for joint accuracy (with random accuracy being 0.25%). Both the Permute baseline and ZipIt! with all layers zipped perform better, but with each at 8.6%, are still clearly lacking. Note that we find the same-model merging budget β to not matter for this set of models (see Fig. 8b), which suggests that there’s not a lot of redundant information *within* each model for this setting. Thus, ZipIt! chooses to merge mostly *across* models instead, performing similarly to the permute baseline. We find this same trend in CIFAR with smaller models (see Fig. 9), so this may be an artifact of model capacity.

To that end, the story changes when we increase the capacity of the merged model by partial zipping: ZipIt!_{10/50} is able to reach close to upper bound ensemble accuracy while saving on FLOPs, *on this extremely difficult task*.

5.3. Multi-Dataset Merging

In this experiment, we take disjoint task model merging one step further by merging ResNet-50 models with different initializations trained on *four completely separate datasets*, each with a different set of labels: Stanford Dogs [29], Oxford Pets [44], CUB200 [58], and NABirds [54]). In Tab. 3, we show the average per task accuracy for each dataset both if we exhaustively merge each pair and also the much more difficult setting of merging all four at once. We report the accuracy of our baselines by applying them up until the last layer, but we can’t compare to prior work as they don’t support this setting. Note that as in all our previous experiment we merge *without training*.

For pairs of models, ZipIt! slightly out performs our permute baseline across all tasks. And for merging all 4 models at once, we perform similarly to permuting. However, again, if we add additional capacity to the merged model through partial unzipping, we perform up to 25.6% better on merging pairs and a massive 35.4% better on merging all four models than the permute baseline. This shows that

Method	FLOPs (G)	Per-Task (%)				
		SD	OP	CUB	NAB	Avg
Merging Pairs						
W. Avg (Eq. 1)	4.11	15.1	23.8	11.8	2.1	13.2
Permute (Eq. 2)	4.11	51.3	64.7	36.7	15.5	42.1
ZipIt! _{49/50}	4.11	51.2	67.7	40.6	15.6	43.8
Ensemble	8.22	72.7	83.2	71.0	77.2	76.0
ZipIt! _{37/50}	4.92	56.8	73.8	54.6	37.9	55.8
ZipIt! _{22/50}	6.39	65.3	79.7	64.8	61.2	67.7
Merging All 4						
W. Avg (Eq. 1)	4.12	0.7	3.4	0.4	0.2	1.2
Permute (Eq. 2)	4.12	34.2	55.4	13.4	5.7	27.2
ZipIt! _{49/50}	4.12	32.1	55.3	14.7	6.9	27.3
Ensemble	16.44	72.7	83.2	71.0	77.2	76.0
ZipIt! _{37/50}	6.5	39.9	66.4	44.3	24.6	43.8
ZipIt! _{22/50}	11.0	58.2	78.5	58.6	55.1	62.6

Table 3: **Multi-Dataset Results.** Merging pairs of differently initialized ResNet-50 models trained on *completely different datasets*: Stanford Dogs (SD), Oxford Pets (OP), CUB200 (CUB), and NABirds (NAB). We report average per-task accuracy over all pairs (2-way merging) and per-task accuracy for each head (4-way merging). We compare to our strong baseline as [2] doesn’t support models with different outputs.

Algorithm	A \leftrightarrow A/B \leftrightarrow B?	Acc	Time
Identity (Eq. 1)	✗	43.0 \pm 3.1	1.8 ms
Permute (Eq. 2)	✗	58.4 \pm 1.3	28 ms
K-Means	✓	29.1 \pm 5.5	19 sec
Zip (Eq. 7)			
Optimal Match	✓	79.6 \pm 1.7	11 min
Greedy Match	✓	79.0 \pm 1.8	1.1 sec
Greedy, $\alpha=0.1$	✓	79.1 \pm 2.1	1.2 sec

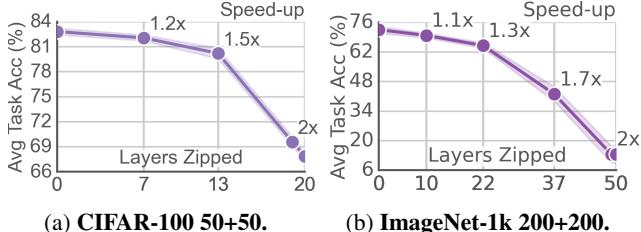
Table 4: **Matching Algorithm** to use for M_i . Permuting B \rightarrow A as in prior work (Eq. 2) performs poorly, thus we allow merging features *within* each model (Eq. 7). Our greedy approach is nearly as accurate as the optimal algorithm while being two orders of magnitude faster. “Acc” is CIFAR-10 (5+5) joint 10-way accuracy.

partial zipping is a significant factor in obtain strong performance when merging models together, especially as the number of models merged increases.

6. Analysis

Here, we analyze and ablate the performance of ZipIt! on the settings described in Sec. 5.

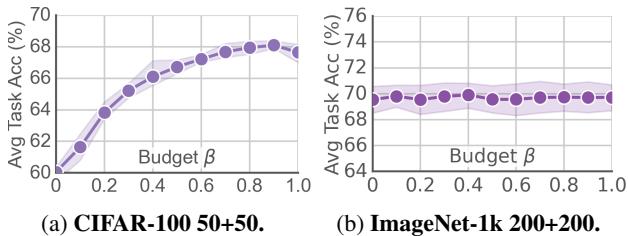
Matching Algorithm. In Tab. 4, we compare matching algorithms used to compute M_i in Eq. 7. Using either the identity (weight averaging) or a permutation (as in prior work) underperforms on CIFAR-10 (5+5) joint 10-way classification. In contrast, if we allow merging *within* models as well, then we obtain up to 21.2% higher accuracy than permuting alone. However, doing this optimally



(a) CIFAR-100 50+50.

(b) ImageNet-1k 200+200.

Figure 7: **Varying Partial Zip.** By leaving some layers unzipped (Sec. 4.3), we can recover a significant amount of performance while still merging most of the model.



(a) CIFAR-100 50+50.

(b) ImageNet-1k 200+200.

Figure 8: **Varying β .** We test the importance of same model matches by varying the budget β (Sec. 4.3). A budget of 0 means no same-model matches are allowed, while 1 places no restrictions. We find when the model has enough capacity for the task, a high budget improves performance.

is difficult, as the standard linear sum assignment algorithm assumes bipartite matches. We could use a graph-based solver (e.g., [21]) instead, but doing so is prohibitively slow (11 minutes to transform a ResNet-20 \times 4 model). Thus, we find matches greedily by repeatedly taking the most correlated pair of features without replacement. This performs almost as well, but is multiple orders of magnitude faster. If we allow repeated matches (Sec. 4.3), we obtain a slightly better result. Like [6], we also find that matching is better for merging features than clustering (e.g., K-Means).

Partial Zipping. In Fig. 6, we plot the average per task accuracy by the number of layers zipped in ResNet-20 \times 8 for CIFAR-100 (50+50) and ResNet-50 for ImageNet-1k (200+200). Note that to avoid adding extra unmerge modules into the network, our stopping point while unzipping has to be the end of a stage. Overall, we find partial zipping to be a simple yet effective technique to add capacity back to the merged model. For CIFAR-100, we can obtain near ensemble accuracies at a $1.5\times$ speed-up. Similarly on a difficult setting like ImageNet, partial zipping is *necessary* to obtain any reasonable accuracy.

Merging within Models. A critical piece of ZipIt! compared to prior work is the ability to merge *within* models, not just *across* models. In Sec. 4.3, we introduce a budget parameter β to limit the number of same-model merges, and here use CIFAR-100 (50+50) and ImageNet-1k (200+200)

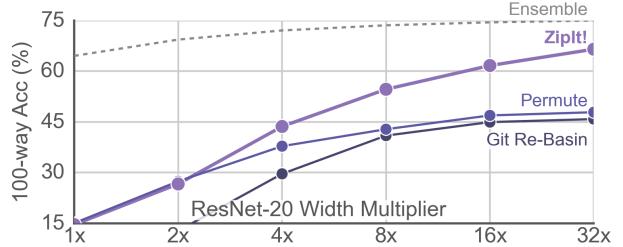


Figure 9: **Model Scale.** As we increase the width of the ResNet-20 models used for the CIFAR-100 (50+50) setting, ZipIt! makes effective use of that extra capacity, quickly approaching ensemble accuracy. Git Re-Basin [2] and Permute only slightly benefit from the extra scale.

to illustrate its effectiveness (Fig. 8). On CIFAR, same-model merges are very important, with the optimal budget being above 0.8, meaning 80% of merges are allowed to be within the same model. This is not the case, however, on ImageNet, where the difficulty of the task means there likely are much fewer redundant features *within* each model.

Model Scale. In Fig. 9, we test the effect of model scale directly by evaluating joint accuracy on our CIFAR-100 (50+50) setting with ResNet-20 models of increasing width. Here, we explicitly see that when the width of the models are too small for the task (e.g., < 4), ZipIt! and the Permute baseline perform identically (though both much better than Git Re-Basin). However, when the scale increases, ZipIt! trends toward the ensemble upper bound of 75%, while both the Permute baseline and Git Re-Basin plateau at around 45%. This indicates that our method uses the extra capacity of these models effectively, much better than prior work.

7. Conclusion

In this paper, we tackle the extremely difficult task of merging two or more models trained on completely disjoint tasks *without additional training*. We show experimentally how prior work falls short in this setting and posit that this is due to not merging features *within models* as well as merging *every* layer in the model at once. We then introduce ZipIt!, a generalized framework for merging models that deals with these issues and find it to significantly outperform both prior work [2] and our own strong baseline on a number of difficult model merging settings. We then analyze the behavior of our method and find that at smaller model capacities, it performs similarly to permutation-based methods, but can perform much better as the model capacity increases. We hope ZipIt! can serve as a strong starting point for practical applications of merging models trained on different tasks.

References

- [1] Hongjoon Ahn, Jihwan Kwak, Subin Lim, Hyeonsu Bang, Hyojun Kim, and Taesup Moon. Ss-il: Separated softmax for incremental learning. In *ICCV*, 2021. 6
- [2] Samuel K Ainsworth, Jonathan Hayase, and Siddhartha Srinivasa. Git re-basin: Merging models modulo permutation symmetries. *arXiv:2209.04836*, 2022. 1, 2, 3, 4, 6, 7, 8, 11, 12
- [3] Stephen Ashmore and Michael Gashler. A method for finding similarity between multi-layer perceptrons by forward bipartite alignment. In *IJCNN*, 2015. 2
- [4] Alexei Baevski, Wei-Ning Hsu, Qiantong Xu, Arun Babu, Jiatao Gu, and Michael Auli. Data2vec: A general framework for self-supervised learning in speech, vision and language. In *ICML*, 2022. 2
- [5] Gilles Blanchard, Gyemin Lee, and Clayton Scott. Generalizing from several related classification tasks to a new unlabeled sample. In *NeurIPS*, 2011. 1
- [6] Daniel Bulya, Cheng-Yang Fu, Xiaoliang Dai, Peizhao Zhang, Christoph Feichtenhofer, and Judy Hoffman. Token merging: Your vit but faster. *ICLR*, 2023. 8
- [7] Daniel Bulya, Chong Zhou, Fanyi Xiao, and Yong Jae Lee. Yolact: Real-time instance segmentation. In *ICCV*, 2019. 1
- [8] Zhaowei Cai, Avinash Ravichandran, Subhransu Maji, Charles Fowlkes, Zhuowen Tu, and Stefano Soatto. Exponential moving average normalization for self-supervised and semi-supervised learning. In *CVPR*, 2021. 2
- [9] Mathilde Caron, Hugo Touvron, Ishan Misra, Hervé Jégou, Julien Mairal, Piotr Bojanowski, and Armand Joulin. Emerging properties in self-supervised vision transformers. In *ICCV*, 2021. 2
- [10] Matthias De Lange, Rahaf Aljundi, Marc Masana, Sarah Parisot, Xu Jia, Aleš Leonardis, Gregory Slabaugh, and Tinne Tuytelaars. A continual learning survey: Defying forgetting in classification tasks. *TPAMI*, 2021. 1, 3
- [11] Mostafa Dehghani, Josip Djolonga, Basil Mustafa, Piotr Padlewski, Jonathan Heek, Justin Gilmer, Andreas Steiner, Mathilde Caron, Robert Geirhos, Ibrahim Alabdulmohsin, et al. Scaling vision transformers to 22 billion parameters. *arXiv:2302.05442*, 2023. 1
- [12] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *CVPR*, 2009. 1, 2, 6
- [13] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv:2010.11929*, 2020. 1
- [14] Felix Draxler, Kambis Veschgini, Manfred Salmhofer, and Fred Hamprecht. Essentially no barriers in neural network energy landscape. In *ICML*, 2018. 2
- [15] Rahim Entezari, Hanie Sedghi, Olga Saukh, and Behnam Neyshabur. The role of permutation invariance in linear mode connectivity of neural networks. *arXiv:2110.06296*, 2021. 2, 3
- [16] Jonathan Frankle, Gintare Karolina Dziugaite, Daniel Roy, and Michael Carbin. Linear mode connectivity and the lottery ticket hypothesis. In *ICML*, 2020. 2
- [17] C Daniel Freeman and Joan Bruna. Topology and geometry of half-rectified network optimization. *arXiv:1611.01540*, 2016. 2
- [18] Timur Garipov, Pavel Izmailov, Dmitrii Podoprikin, Dmitry P Vetrov, and Andrew G Wilson. Loss surfaces, mode connectivity, and fast ensembling of dnns. *NeurIPS*, 2018. 2
- [19] Jort F Gemmeke, Daniel PW Ellis, Dylan Freedman, Aren Jansen, Wade Lawrence, R Channing Moore, Manoj Plakal, and Marvin Ritter. Audio set: An ontology and human-labeled dataset for audio events. In *ICASSP*, 2017. 1
- [20] Jean-Bastien Grill, Florian Strub, Florent Altché, Corentin Tallec, Pierre Richemond, Elena Buchatskaya, Carl Doersch, Bernardo Avila Pires, Zhaohan Guo, Mohammad Gheshlaghi Azar, et al. Bootstrap your own latent-a new approach to self-supervised learning. *NeurIPS*, 2020. 2
- [21] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using networkx. In *Proceedings of the 7th Python in Science Conference*, 2008. 8
- [22] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. Mask r-cnn. In *ICCV*, 2017. 1
- [23] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. 2016. 1, 6
- [24] Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models. *NeurIPS*, 2020. 1
- [25] Gao Huang, Yixuan Li, Geoff Pleiss, Zhuang Liu, John E Hopcroft, and Kilian Q Weinberger. Snapshot ensembles: Train 1, get m for free. *arXiv:1704.00109*, 2017. 2, 3
- [26] Pavel Izmailov, Dmitrii Podoprikin, Timur Garipov, Dmitry Vetrov, and Andrew Gordon Wilson. Averaging weights leads to wider optima and better generalization. *UAI*, 2018. 2, 3
- [27] Keller Jordan, Hanie Sedghi, Olga Saukh, Rahim Entezari, and Behnam Neyshabur. Repair: Renormalizing permuted activations for interpolation repair. *arXiv:2211.08403*, 2022. 1, 2, 3, 4, 5, 6, 11
- [28] Tero Karras, Samuli Laine, and Timo Aila. A style-based generator architecture for generative adversarial networks. In *CVPR*, 2018. 1
- [29] Aditya Khosla, Nityananda Jayadevaprakash, Bangpeng Yao, and Fei-Fei Li. Novel dataset for fine-grained image categorization: Stanford dogs. In *CVPR Workshop on Fine-Grained Visual Categorization (FGVC)*, 2011. 7
- [30] James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, et al. Overcoming catastrophic forgetting in neural networks. *PNAS*, 2017. 1
- [31] Simon Kornblith, Mohammad Norouzi, Honglak Lee, and Geoffrey Hinton. Similarity of neural network representations revisited. In *ICML*, 2019. 2
- [32] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009. 2, 6

- [33] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 2017. 1
- [34] Wei-Hong Li and Hakan Bilen. Knowledge distillation for multi-task learning. In *ECCV*, 2020. 1
- [35] Yanghao Li, Hanzi Mao, Ross Girshick, and Kaiming He. Exploring plain vision transformer backbones for object detection. In *ECCV*, 2022. 1
- [36] Yixuan Li, Jason Yosinski, Jeff Clune, Hod Lipson, and John Hopcroft. Convergent Learning: Do different neural networks learn the same representations? *arXiv:1511.07543*, 2015. 6
- [37] Zhizhong Li and Derek Hoiem. Learning without forgetting. *TPAMI*, 2017. 1
- [38] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In *ECCV*, 2014. 3
- [39] Chang Liu, Chenfei Lou, Runzhong Wang, Alan Yuhan Xi, Li Shen, and Junchi Yan. Deep neural network fusion via graph matching with applications to model ensemble and federated learning. In *ICML*, 2022. 2
- [40] Michael Matena and Colin Raffel. Merging models with fisher-weighted averaging. *arXiv:2111.09832*, 2021. 2
- [41] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. Communication-efficient learning of deep networks from decentralized data. In *Artificial intelligence and statistics*. PMLR, 2017. 2
- [42] Krikamol Muandet, David Balduzzi, and Bernhard Schölkopf. Domain generalization via invariant feature representation. In *ICML*, 2013. 1
- [43] Behnam Neyshabur, Hanie Sedghi, and Chiyuan Zhang. What is being transferred in transfer learning? *NeurIPS*, 2020. 2
- [44] Omkar M. Parkhi, Andrea Vedaldi, Andrew Zisserman, and C. V. Jawahar. Cats and dogs. In *CVPR*, 2012. 7
- [45] Fidel A Guerrero Peña, Heitor Rapela Medeiros, Thomas Dubail, Masih Aminbeidokhti, Eric Granger, and Marco Pedersoli. Re-basin via implicit sinkhorn differentiation. *arXiv:2212.12042*, 2022. 2
- [46] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. Learning transferable visual models from natural language supervision. In *ICML*, 2021. 6, 11
- [47] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. *NeurIPS*, 2015. 1
- [48] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-resolution image synthesis with latent diffusion models. In *CVPR*, 2022. 1
- [49] Tamar Rott Shaham, Tali Dekel, and Tomer Michaeli. Singan: Learning a generative model from a single natural image. In *ICCV*, 2019. 13
- [50] Sidak Pal Singh and Martin Jaggi. Model fusion via optimal transport. *NeurIPS*, 2020. 2
- [51] Antti Tarvainen and Harri Valpola. Mean teachers are better role models: Weight-averaged consistency targets improve semi-supervised deep learning results. *NeurIPS*, 2017. 2
- [52] Norman Tatro, Pin-Yu Chen, Payel Das, Igor Melnyk, Prasanna Sattigeri, and Rongjie Lai. Optimizing mode connectivity via neuron alignment. *NeurIPS*, 2020. 2
- [53] Zhi Tian, Chunhua Shen, Hao Chen, and Tong He. Fcos: Fully convolutional one-stage object detection. In *ICCV*, 2019. 1
- [54] Grant Van Horn, Steve Branson, Ryan Farrell, Scott Haber, Jessie Barry, Panos Ipeirotis, Pietro Perona, and Serge Belongie. Building a bird recognition app and large scale dataset with citizen scientists: The fine print in fine-grained dataset collection. In *CVPR*, 2015. 7
- [55] Johannes Von Oswald, Sejin Kobayashi, Joao Sacramento, Alexander Meulemans, Christian Henning, and Benjamin F Grewe. Neural networks with late-phase weights. *arXiv:2007.12927*, 2020. 2
- [56] Hongyi Wang, Mikhail Yurochkin, Yuekai Sun, Dimitris Papailiopoulos, and Yasaman Khazaeni. Federated learning with matched averaging. *arXiv:2002.06440*, 2020. 2
- [57] Jindong Wang, Cuiling Lan, Chang Liu, Yidong Ouyang, Tao Qin, Wang Lu, Yiqiang Chen, Wenjun Zeng, and Philip Yu. Generalizing to unseen domains: A survey on domain generalization. *TKDE*, 2022. 1
- [58] P. Welinder, S. Branson, T. Mita, C. Wah, F. Schroff, S. Belongie, and P. Perona. Caltech-UCSD Birds 200. Technical Report CNS-TR-2010-001, California Institute of Technology, 2010. 7
- [59] Mitchell Wortsman, Gabriel Ilharco, Samir Ya Gadre, Rebecca Roelofs, Raphael Gontijo-Lopes, Ari S Morcos, Hongseok Namkoong, Ali Farhadi, Yair Carmon, Simon Kornblith, et al. Model soups: averaging weights of multiple fine-tuned models improves accuracy without increasing inference time. In *ICML*, 2022. 1, 2, 3
- [60] Mitchell Wortsman, Gabriel Ilharco, Jong Wook Kim, Mike Li, Simon Kornblith, Rebecca Roelofs, Raphael Gontijo Lopes, Hannaneh Hajishirzi, Ali Farhadi, Hongseok Namkoong, et al. Robust fine-tuning of zero-shot models. In *CVPR*, 2022. 2
- [61] Mikhail Yurochkin, Mayank Agarwal, Soumya Ghosh, Kristjan Greenewald, Nghia Hoang, and Yasaman Khazaeni. Bayesian nonparametric federated learning of neural networks. In *ICML*, 2019. 2
- [62] Xiaohua Zhai, Alexander Kolesnikov, Neil Houlsby, and Lucas Beyer. Scaling vision transformers. In *CVPR*, 2022. 1
- [63] Bolei Zhou, Hang Zhao, Xavier Puig, Sanja Fidler, Adela Barriuso, and Antonio Torralba. Scene parsing through ade20k dataset. In *CVPR*, 2017. 1

A. Data Usage

In our approach, we use a sample of the training set in order to compute activations and match features together. For the main paper, we used the full training set for CIFAR, 1% of the training set for ImageNet, and the number of images in the smallest training set for the Multi-Dataset experiment (so that we could use the same number of images from each dataset). In each case, we used the same data augmentations from training.

That begs the question: how much data do we actually need, and how necessary are data augmentations? In Fig. 10, we test how much data is actually necessary to obtain a good accuracy on CIFAR and ImageNet with or without data augmentation.

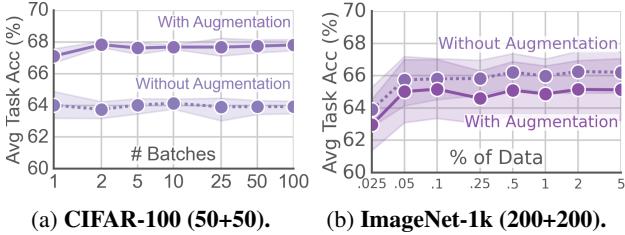


Figure 10: **Data Usage.** How much data do we need to use to compute activations? Here we ablate the amount of data used for our CIFAR-100 (50+50) ResNet-20 ($8 \times$ width) and ImageNet (200+200) Resnet-50 (2% layers) experiments. The batch size used is 500 for CIFAR and 16 for ImageNet. In both cases, we only need a few hundred images to obtain the best results. On the other hand, data augmentation is necessary for CIFAR but hurts for ImageNet. Our default for all experiments uses data augmentation and the full set for CIFAR (100 batches) and 1% of the data for ImageNet.

We find ultimately that the amount of data doesn't actually matter that much. In the main paper, we use the entire training set for CIFAR-100 with a batch size of 500 (100 batches, or 50,000 images), but it seems like as little as 2 batches (100 images) produces the same result. Similarly on ImageNet, using 0.05% of the data (640 images) produces the same result as 5% (64,048 images).

In fact, the main consideration is whether or not to use data augmentation. For less diverse datasets like CIFAR-100, data augmentation seems essential (giving an almost 4% boost in average task accuracy), and well above the variance of results without augmentation. However, for ImageNet, which has much more diverse images, data augmentation actually hurts slightly on average—though the two are within variance. Note that despite this result, for consistency we use data augmentation in *all* experiments.

B. Zip Propagation Details

In the main paper we described general rules for zip propagation—namely, propagate through layers until you reach a module with a weight matrix. Here, we describe rules more concretely for each layer type needed to define most convnets.

Linear. Apply M_i and U_i . Stop propagation.

Conv. Apply M_i and U_i to each kernel location (i.e., move the $k \times k$ kernel dimensions to the batch dimension). Stop propagation.

BatchNorm. Apply M_i to all parameters (weight, bias, mean, variance), squaring it for the variance term. Continue propagation. As [27] points out, we cannot compute the correct variance without knowing the covariance between the two models (which we don't have access to). Thus, we reset batch norms after merging to evaluate the variance correctly.

LayerNorm. Apply M_i to all parameters (weight, bias). Continue propagation. Since LayerNorm computes mean and standard deviation on the fly, we don't need to do anything special.

ReLU. Nothing to merge. Continue propagation. Note that passing the merge matrix unchanged through the ReLU layers is an approximation, since we're using a linear merge operation on nonlinear features. Addressing this issue could be an interesting topic for future work, as even the permute and add approach of prior work has this issue (ReLU is invariant to permutation, but certainly not adding).

Avg / Max Pool. Nothing to Merge. Continue propagation.

Skip Connection. Continue propagation through every input to the skip connection (using the same M_i and U_i for each).

C. CIFAR with Cross Entropy

In the main paper, we train our CIFAR models with a CLIP [46] loss (using CLIP embeddings of class names as targets). This ensures that the output spaces of the two models are aligned, which is necessary to get good accuracy for prior work that merge the entire model together. In Tab. 5, we show results for CIFAR-100 (50+50) where we train with the normal one-hot targets (i.e., like we did for ImageNet), instead.

Immediately, we see that accuracies of the merged models are much lower across the board, with no method able to outperform just using one of the original models when merging the entire network. In fact, Git Re-Basin [2] does almost no better than weight averaging, which gets close to random accuracy. While ZipIt! without partial zipping also performs worse than the original models, it still greatly outperforms all prior work. And with partial zipping, ZipIt! is

Method	FLOPs (G)	Joint Acc (%)	Per-Task (%)		
			Task A	Task B	Avg
Model A	10.88	37.9 \pm 0.2	74.15 \pm 0.6	1.7 \pm 0.3	36.4 \pm 0.7
Model B	10.88	36.7 \pm 1.2	2.2 \pm 0.5	75.2 \pm 2.3	38.7 \pm 1.1
W. Avg	10.88	2.7 \pm 0.1	5.0 \pm 3.3	4.9 \pm 1.2	4.9 \pm 0.1
Git Re-Basin [2]	10.88	3.1 \pm 0.8	5.8 \pm 0.9	5.3 \pm 0.8	5.6 \pm 0.8
Permute	10.88	20.0 \pm 1.3	30.8 \pm 3.3	32.8 \pm 1.6	31.8 \pm 1.7
ZipIt!_{20/20}	10.88	27.9\pm1.5	40.1\pm2.4	39.7\pm1.6	39.9\pm1.9
Ensemble	21.76	60.5 \pm 1.0	74.2 \pm 0.6	75.2 \pm 2.3	74.7 \pm 1.4
ZipIt!_{13/20}	14.52	38.6 \pm 2.4	51.8 \pm 1.6	52.0 \pm 3.5	51.9 \pm 2.4
ZipIt!_{7/20}	18.14	47.0\pm2.2	60.6\pm1.2	60.5\pm3.2	60.6\pm2.1

Table 5: **CIFAR-100 (50+50) with Cross Entropy.** ZipIt! vs. baselines using ResNet-20 ($16\times$ width). Merging the entire model as in prior work produces bad results when using cross-entropy, hence we use CLIP in the main draft. If we use partial zipping, we can recover a lot of the lost performance.

able to exceed the accuracy of the original models.

Thus, in the case of using cross-entropy loss, partial zipping is extremely important. Merging the entire model as in prior work fails, since the later layers of the model are incompatible with each other due to each model having a different output space. Partial zipping, on the other hand, can mitigate that issue.

D. CIFAR with VGG

In the main paper, we use ResNets for each experiment, since they are easy to train and produce strong results. However, in principle ZipIt! can work on any architecture. For completeness, in Tab. 6, we show results on the CIFAR-10 (5+5) setting with VGG11 ($1\times$ width). Note that this is a much smaller and weaker model than the ResNet-20s we use in the main paper, so its results on CIFAR-10 aren’t as strong. Furthermore, we conducted this experiment with a cross entropy loss, so merging the entire model performs worse than the original models.

Despite this, we observe a very similar trend to the ResNet-20 models in that ZipIt! outperforms all baselines and that partial zipping is important for reaching the accuracy of the original models (in this case, matching not exceeding). In fact, these results continue a more general trend in that ZipIt! greatly benefits from larger model scales, making effective use of the extra capacity. In this case, the scale of the model is quite small, so there is not as much room in the weights to store the potentially disjoint features of both models.

E. ImageNet with 1.5x Width

In the main paper, we show that ZipIt! scales very well with increased width of the model for the CIFAR-100 (50+50) setting. While CIFAR-100 is a challenging dataset

Method	FLOPs (G)	Joint Acc (%)	Per-Task (%)		
			Task A	Task B	Avg
Model A	0.15	44.6 \pm 1.0	89.2 \pm 2.0	21.0 \pm 1.5	55.1 \pm 1.5
Model B	0.15	44.0 \pm 1.4	23.1 \pm 4.5	88.1 \pm 2.8	55.6 \pm 3.4
W. Avg	0.15	10.2 \pm 0.2	20.8 \pm 1.2	20.9 \pm 0.7	20.9 \pm 0.7
Permute	0.15	25.4 \pm 1.1	47.2 \pm 1.3	48.5 \pm 12.3	47.8 \pm 5.8
ZipIt!_{22/22}	0.15	33.2\pm9.3	53.8\pm14.7	59.9\pm9.6	56.5\pm6.5
Ensemble	0.30	66.6 \pm 2.5	89.2 \pm 2.0	88.1 \pm 2.8	88.6 \pm 0.5
ZipIt!_{14/22}	0.17	35.2 \pm 6.0	56.7 \pm 13.7	60.2 \pm 15.9	58.4 \pm 5.9
ZipIt!_{7/22}	0.27	44.5\pm2.9	66.0\pm4.2	65.1\pm11.2	65.5\pm4.1

Table 6: **CIFAR-10 (5+5) CE with VGG.** ZipIt! vs. baselines using VGG11 ($1\times$ width) using Cross Entropy instead of CLIP loss. ZipIt! displays the same behavior here as it does for ResNet-20 with low width. Note that this is CIFAR-10 not CIFAR-100.

Method	FLOPs (G)	Joint Acc (%)	Per-Task (%)		
			Task A	Task B	Avg
$1\times$ Width					
Ensemble	8.22	63.3 \pm 4.9	74.3 \pm 4.0	70.5 \pm 3.2	72.4 \pm 2.5
ZipIt!_{50/50}	4.11	8.6\pm4.7	12.4\pm5.9	14.7\pm7.8	13.5\pm6.6
ZipIt!_{37/50}	4.92	33.1 \pm 5.9	41.8 \pm 5.3	42.3 \pm 8.2	42.0 \pm 6.2
ZipIt!_{22/50}	6.39	55.8\pm4.1	65.9\pm2.5	64.1\pm3.0	65.0\pm2.3
ZipIt!_{10/50}	7.43	60.9 \pm 4.1	70.7 \pm 3.0	69.0 \pm 2.9	69.9 \pm 1.9
$1.5\times$ Width					
Ensemble	32.6	67.8 \pm 3.5	76.7 \pm 4.1	72.6 \pm 2.8	74.7 \pm 2.5
ZipIt!_{50/50}	16.3	9.7\pm6.9	13.2\pm9.5	16.0\pm10.0	14.6\pm9.3
ZipIt!_{37/50}	19.5	49.0\pm2.5	56.2\pm4.2	56.7\pm2.1	56.4\pm2.8
ZipIt!_{22/50}	25.5	64.1\pm2.7	71.6\pm2.3	70.4\pm2.3	71.0\pm1.8
ZipIt!_{10/50}	29.7	66.8\pm3.2	74.9\pm3.5	72.1\pm2.3	73.5\pm2.1

Table 7: **ImageNet-1k (200+200) Width Comparison.** We show how ZipIt! is able to make use of the extra model width when merging models together. For instance, merging 37 layers goes from 33% joint accuracy with $1\times$ width to 49% with $1.5\times$, while the ensemble only improves by 4%. Because these models use cross-entropy, merging the entire network still results in poor performance.

on its own, the natural question is if that same trend occurs the much harder ImageNet-1k (200+200) setting.

In Tab. 7, we test this by comparing ZipIt! on the original $1\times$ width ResNet-50 in the main paper with a $1.5\times$ width one. In all cases, except for the fully zipped model (likely because of the Cross-Entropy loss), ZipIt! enjoys a large jump in performance from the extra width. For 37 layers, 33.1% joint accuracy becomes 49.0%. For 22 layers, 55.8% becomes 64.1%. And for 10 layers, 60.9% becomes 66.8%, now only 1% away from the ensemble.

Thus, even in this much more challenging setting, ZipIt! makes full use of the extra model capacity.

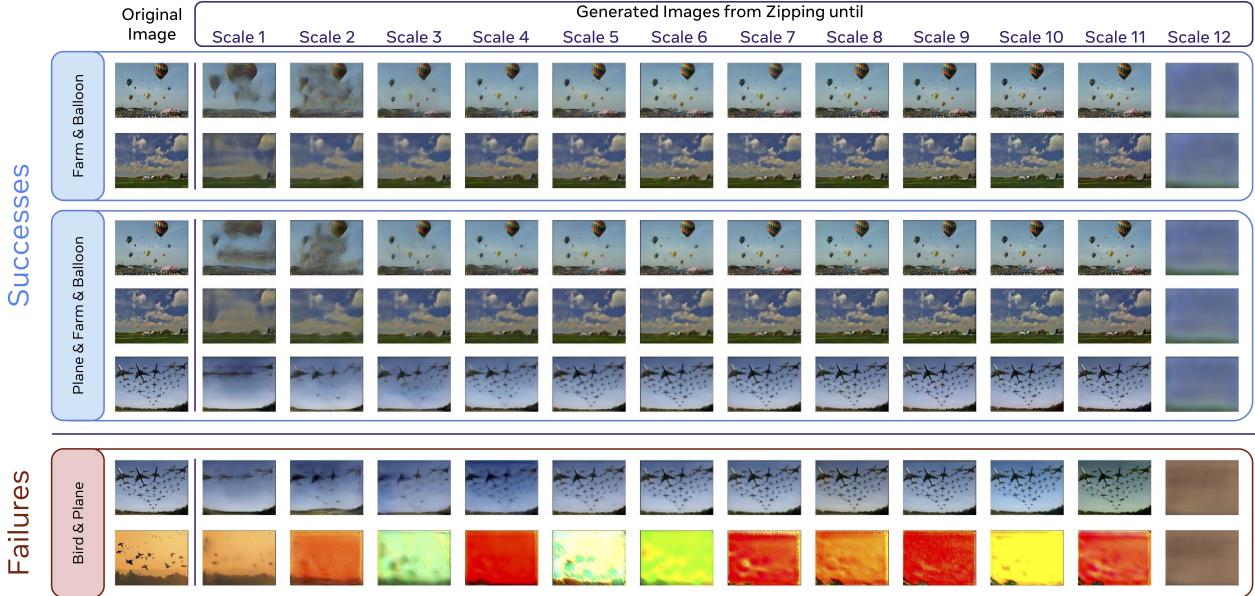


Figure 11: **Merging SinGAN Models.** Partially zipping several SinGAN [49] models with ZipIt! We visualize three experiments settings: in the first (top), we merge a SinGAN trained on a picture of hot air balloons and another trained on a picture of a farm. In the second (middle), we zip three SinGAN models (trained on balloons, a farm, and planes) together. Notably, the first two experiments partially zip SinGANs trained on images of *similar contexts*, as the background of each image is similar. In contrast, the third (bottom) image showcases the results when zipping SinGANs trained on images with *different contexts* (e.g., times of day). The leftmost image in each row is the training image, while the remaining images are the results from partially zipping up to the corresponding scale block (e.g., “Scale 7” denotes partially zipping until the 7th scaling block).

F. SinGAN

While we applied ZipIt! mostly to classification models so far, our method could in theory be applied to any domain. We test this on image generation by applying ZipIt! to merge several SinGAN [49] models trained on different images with different initializations.

To allow for sampling the ground truth image from all models at once, we set the ground truth noise sample to be the same for all models. After training, we remove each model’s discriminator, leaving only their generator networks—each consisting of 12 stacked scaling blocks that iteratively refine the generated output images from their predecessors. We merge two or more of these SinGANs by partially zipping (as defined in Section 4.3) up to a given block (scale 1 indicating that merging stops after the first scale and scale 12 that the entire network is merged). We then pass the ground truth noise to the model and display the images generated from each head.

Fig 11 visualizes our results. The first (top) experiment combines two SinGANs trained on an image of a farm and balloon respectively, the second experiment further adds a SinGAN trained on a plane image to the mix, while the third experiment visualizes the effect of zipping models trained

on images with *different contexts*. The leftmost column in each row shows the image used to train each respective SinGAN model, while the remaining images in a row show the resulting images generated from partial zipping.

Overall, we find that ZipIt! is capable of zipping nearly all parameters of SinGANs trained on images of *similar contexts* (e.g., similar backgrounds) while retaining their abilities to generate semantically similar images to their respective ground truths. In fact ZipIt! can merge up to 11/12 scaling blocks of three SinGANs while still being able to generate images semantically equivalent to the images each individual model was trained on. Notably, this *decreases* the effective number of scaling blocks required to generate distinct images by $\frac{(12 \times 3) - (11+3)}{12 \times 3} \times 100 = 61\%$!

Interestingly however, we find that shared context is critical to a successful merge. The third experiment (bottom) visualizes this observation: zipping two SinGANs trained on an image of planes in the day time, and an image of birds flying at dusk yield very poor results at *all* zipping locations. This is likely due to the rgb pixel values being too different for merging to conserve important features.