Rare Gems: Finding Lottery Tickets at Initialization

Kartik Sreenivasan[*] [w], **Jy-yong Sohn**[*] [w], **Liu Yang**[w], **Matthew Grinde**[w]
Alliot Nagle[w], **Hongyi Wang**[c], **Eric Xing**[m,c,p], **Kangwook Lee**[w], **Dimitris Papailiopoulos**[w]

[c] Carnegie Mellon University [m] Mohamed Bin Zayed University of Artificial Intelligence
[p] Petuum, Inc. [w] University of Wisconsin-Madison

Abstract

Large neural networks can be pruned to a small fraction of their original size, with little loss in accuracy, by following a time-consuming "train, prune, re-train" approach. Frankle & Carbin [9] conjecture that we can avoid this by training *lottery tickets*, *i.e.*, special sparse subnetworks found *at initialization*, that can be trained to high accuracy. However, a subsequent line of work [11, 41] presents concrete evidence that current algorithms for finding trainable networks at initialization, fail simple baseline comparisons, *e.g.*, against training random sparse subnetworks. Finding lottery tickets that train to better accuracy compared to simple baselines remains an open problem. In this work, we resolve this open problem by proposing GEM-MINER which finds lottery tickets *at initialization* that beat current baselines. GEM-MINER finds lottery tickets trainable to accuracy competitive or better than Iterative Magnitude Pruning (IMP), and does so up to $19\times$ faster.

1 Introduction

A large body of research since the 1980s empirically observed that large neural networks can be compressed or sparsified to a small fraction of their original size while maintaining their predictive accuracy [14–16, 20, 23, 29, 45]. Although several pruning methods have been proposed during the past few decades, many of them follow the "*train, prune, re-train*" paradigm. Although the above methods result in very sparse, accurate models, they typically require several rounds of re-training, which is computationally intensive.

Frankle & Carbin [9] suggest that this computational burden may be avoidable. They conjecture that given a randomly initialized network, one can find *a sparse subnetwork that can be trained to accuracy comparable to that of its fully trained dense counterpart*. This trainable subnetwork found *at initialization* is referred to as a *lottery ticket*. The study above introduced iterative magnitude pruning (IMP) as a means of finding these lottery tickets. Their experimental findings laid the groundwork for what is now known as the *Lottery Ticket Hypothesis* (LTH).

Although Frankle & Carbin [9] establish that the LTH is true for tasks like image classification on MNIST, they were not able to get satisfactory results for more complex datasets like CIFAR-10 and ImageNet when using deeper networks, such as VGG and ResNets [10]. In fact, subsequent work brought the effectiveness of IMP into question. Su et al. [41] showed that even randomly sampled sparse subnetworks at initialization can beat lottery tickets found by IMP as long as the layerwise sparsities are chosen carefully. Gale et al. [12] showed that methods like IMP which train tickets from initialization cannot compete with the accuracy of a model trained with pruning as part of the optimization process.

Frankle et al. [10] explain the failures of IMP using the concept of *linear mode connectivity* which measures the stability of these subnetworks to SGD noise. Extensive follow-up studies propose

[*] Authors contributed equally to this paper.

36th Conference on Neural Information Processing Systems (NeurIPS 2022).

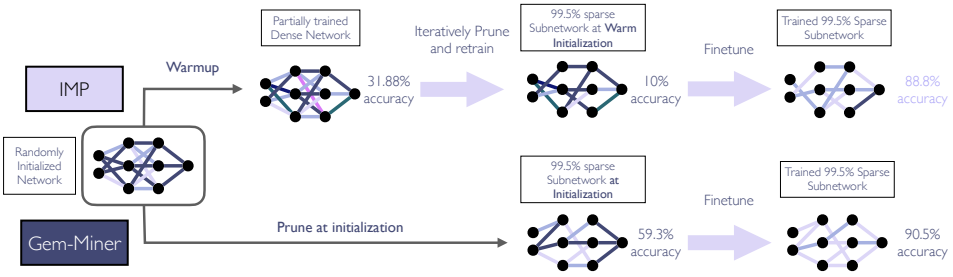

Figure 1: Conceptual visualization of GEM-MINER vs IMP with warmup. The accuracies listed are on a 99.5% sparse VGG-16 trained on CIFAR-10. Given a randomly initialized network, both methods output a subnetwork which is then finetuned. IMP requires warmup *i.e.*, few epochs of training before it can find a sparse subnetwork. GEM-MINER finds a *rare gem*, a subnetwork *at initialization* that achieves high accuracy both before and after weight training.

several heuristics for finding trainable sparse subnetworks at initialization [24, 42, 43]. However, subsequent work by Frankle et al. [11], Su et al. [41] show experimentally that all of these methods fail simple sanity checks. Most methods seem to merely identify good sparsities at each layer, but given those, random sparse subnetworks can be trained to similar or better accuracy.

Frankle et al. [10] show that with a small modification, IMP can beat these sanity checks; the caveat is that it no longer finds these subnetworks at initialization, but after a few epochs of *warm-up* training. Since these subnetworks are found *after initialization*, **IMP with warmup does not find lottery tickets**.

As noted in the original work by Frankle & Carbin [9], the importance of finding trainable subnetworks at initialization is computational efficiency. It is far preferable to train a sparse model from scratch, rather than having to deal with a large dense model, even if that is for a few epochs (which is what IMP with warmup does). To the best of our knowledge, the empirical validity of the *Lottery Ticket Hypothesis*, *i.e.*, the hunt for subnetworks at initialization trainable to SOTA accuracy, remains an open problem.

Our Contributions. We resolve this open problem by developing GEM-MINER, an algorithm that finds sparse subnetworks *at initialization*, trainable to accuracy comparable or better than IMP *with warm-up*. GEM-MINER does so by first discovering *rare gems*. Rare gems are subnetworks at initialization that attain accuracy far above random guessing, even before training. Rare gems can then be *refined* to achieve near state-of-the-art accuracy. Simply put, rare gems are lottery tickets that also have high accuracy at initialization.

High accuracy at initialization is not a requirement for a network to be defined as a lottery ticket. However, if our end goal is high accuracy after training, then having high accuracy at initialization likely helps.

Rare gems found by GEM-MINER are the first lottery tickets to beat all baselines in [11, 41]. In Fig. 1 we give a sketch of how GEM-MINER compares with IMP with warm start. GEM-MINER finds subnetworks *at initialization* and is up to 19× faster than IMP which needs warmup.

2 Related Work

Lottery ticket hypothesis. Following the pioneering work of Frankle & Carbin [9], the search for lottery tickets has grown across several applications, such as language tasks, graph neural networks and federated learning [3, 4, 13, 25]. Savarese et al. [37] propose an alternative to IMP which is significantly faster given enough parallel computing resources. While the LTH itself has yet to be proven mathematically, the so-called strong LTH has been derived which shows that any target network can be approximated by pruning a randomly initialized network with minimal overparameterization [28, 30, 32]. Recently, it has been shown that for such approximation results it suffices to prune a random binary network with slightly larger overparameterization [6, 40].

Pruning at initialization. While network pruning has been studied since the 1980s, finding sparse subnetworks at initialization is a more recently explored approach. Lee et al. [24] propose SNIP,

Table 1: We compare the different popular pruning methods in the literature on whether they prune at initialization, are finetunable and pass sanity checks. We also list the amount of computation they need to find a 98.6% sparse subnetwork on ResNet-20, CIFAR-10. For consistency, we do not include the time required to finetune this subnetwork to full accuracy as it would be equal for all methods. For single-shot pruning method we list it as 1 epoch but this depends on the choice of batch-size. Learning Rate Rewinding which we label Renda et al. [34] is a pruning after training algorithm and just outputs a high accuracy subnetwork and hence the sanity checks do not apply to it.

Pruning Method	Prunes at initialization	Finetunable	Passes sanity checks	Computation to reach 98.6% sparsity
IMP [9]	✗	✓	✓	2850 epochs
SNIP [24]	✓	✓	✗	1 epoch
GraSP [43]	✓	✓	✗	1 epoch
SynFlow [42]	✓	✓	✗	1 epoch
Edge-popup [33]	✓	✗	✗	150 epochs
Smart Ratio [41]	✓	✓	–	$\mathcal{O}(1)$
Learning Rate Rewinding [34]	✗	–	–	3000 epochs
GEM-MINER	✓	✓	✓	**150 epochs**

which prunes based on a heuristic that approximates the importance of a connection. Tanaka et al. [42] propose SynFlow which prunes the network to a target sparsity without ever looking at the data. Wang et al. [43] propose GraSP which computes the importance of a weight based on the Hessian gradient product. Patil & Dovrolis [31] propose PHEW which is based on the decomposition of the Neural Tangent Kernel. Lubana & Dick [26] create an interesting theoretical framework based on gradient flow that justifies the successes and failures of several of these algorithms. The goal of these algorithms is to find a subnetwork that can be trained to high accuracy. Ramanujan et al. [33] propose Edge-Popup (EP) which finds a subnetwork at initialization that has high accuracy to begin with. Unfortunately, they also note that in most cases, these subnetworks are not conducive to further finetuning.

The above algorithms are all based on the idea that one can assign a "score" to each weight to measure its importance. Once such a score is assigned, one simply keeps the top fraction of these scores based on the desired target sparsity. This may be done by sorting the scores layer-wise or globally across the network. Additionally, this can be done in *one-shot* (SNIP, GraSP) or *iteratively* (SynFlow). Note that IMP can also be fit into the above framework by defining the "score" to be the *magnitude* of the weights and then pruning globally across the network iteratively.

More recently, Alizadeh et al. [1] propose ProsPr which utilizes the idea of *meta-gradients* through the first few steps of optimization to determine which weights to prune. Their intuition is that this will lead to masks at initialization that are more amenable to training to high accuracy. While it finds high accuracy subnetworks, we show in Section 4.2 that it fails to pass the sanity checks in [11, 41].

Sanity checks for lottery tickets. A natural question that arises with pruning at initialization is whether these algorithms are truly finding interesting and nontrivial subnetworks, or if their performance after finetuning can be matched by simply training equally sparse, yet random subnetworks. Ma et al. [27] propose more rigorous definitions of winning tickets and study IMP under several settings with careful tuning of hyperparameters. Frankle et al. [11] and Su et al. [41] introduce several sanity checks (i) Random shuffling (ii) Weight reinitialization (iii) Score inversion and (iv) Random Tickets. Even at their best performance, they show that SNIP, GraSP and SynFlow merely find a good sparsity ratio in each layer and fail to surpass, in term of accuracy, fully trained randomly selected subnetworks, whose sparsity per layer is similarly tuned. Frankle et al. [11] show through extensive experiments that none of these methods show accuracy deterioration after random reshuffling. We explain the sanity checks in detail in Section 4 and use them as baselines to test our own algorithm.

Pruning during/after training. While the above algorithms prune at/near initialization, there exists a rich literature on algorithms which prune during/after training. Unlike IMP, algorithms in this category do not rewind the weights. They continue training and pruning iteratively. Frankle et al. [11] and Gale et al. [12] show that pruning at initialization cannot hope to compete with these algorithms. While they do not find lottery tickets, they do find high accuracy sparse networks. Zhu & Gupta [45] propose a gradual pruning schedule where the smallest fraction of weights are pruned at a predefined frequency. They show that this results in models up to 95% sparsity with negligible loss in performance on language as well as image processing tasks. Gale et al. [12] and Frankle et al. [11] also study this as a baseline under the name *magnitude pruning after training*. Renda et al. [34] show that rewinding the learning rate as opposed to weights(like in IMP) leads to the best performing

sparse networks. The closest among these to GEM-MINER is Movement Pruning by Sanh et al. [36] which also computes the mask as a quantized version of the scores in its soft variant. However, it is important to remark that these algorithms do not find Lottery Tickets, merely high accuracy sparse networks. We contrast these different methods in Table 1 in terms of whether they prune at initialization, their finetunability, whether they pass sanity checks as well as their computational costs.

Finally, we note that identifying a good pruning mask can be thought of as training a binary network where the loss is computed over the element-wise product of the original network with the mask. This has been explored in the quantization during training literature [5, 19, 38].

3 GEM-MINER: Discovering Rare Gems

Setting and notation. Let $S = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^n$ be a given training dataset for a k-classification problem, where $\boldsymbol{x}_i \in \mathbb{R}^{d_0}$ denotes a feature vector and label $y_i \in \{1, \ldots, k\}$ denotes its label. Typically, we wish to train a neural network classifier $f(\boldsymbol{w}; \boldsymbol{x}) : \mathbb{R}^{d_0} \to \{1, \ldots, k\}$, where $\boldsymbol{w} \in \mathbb{R}^d$ denotes the set of weight parameters of this neural network. The goal of a pruning algorithm is to extract a mask $\boldsymbol{m} = \{0, 1\}^d$, so that the pruned network is denoted by $f(\boldsymbol{w} \odot \boldsymbol{m}; \boldsymbol{x})$, where $\odot$ denotes the element-wise product. We define the *sparsity* of this network to be the fraction of weights that have been pruned: $s = (1 - \|\boldsymbol{m}\|_0/d)$ following the convention set by Frankle et al. [11]. The loss of a classifier on a single sample $(\boldsymbol{x}, y)$ is denoted by $\ell(f(\boldsymbol{w} \odot \boldsymbol{m}; \boldsymbol{x}), y)$, which captures a measure of discrepancy between prediction and reality. In what follows, we will denote by $\boldsymbol{w}_0 \in \mathbb{R}^d$ to be the set of random initial weights. The type of randomness will be explicitly mentioned when necessary.

On the path to rare gems; first stop: Maximize pre-training accuracy. A rare gem needs to satisfy three conditions: (i) sparsity, (ii) non-trivial pre-training accuracy, and (iii) that it can be finetuned to achieve accuracy close to that of the fully trained dense network. This is not an easy task as we have two different objectives in terms of accuracy (pre-training and post-training), and it is unclear if a good subnetwork for one objective is also good for the other. However, since pre-training accuracy serves as a lower bound on the final performace, we focus on maximizing that first, and then attempt to further improve it by finetuning.

Our algorithm is inspired by Edge-Popup (EP) [33]. EP successfully finds subnetworks with high pre-training accuracy but it has two major limitations: (i) it does not work well in the high sparsity regime (*e.g.*, $> 95\%$), and (ii) most importantly, the subnetworks it finds are typically not conducive to further finetuning in that the final accuracy does not approach the performance of IMP.

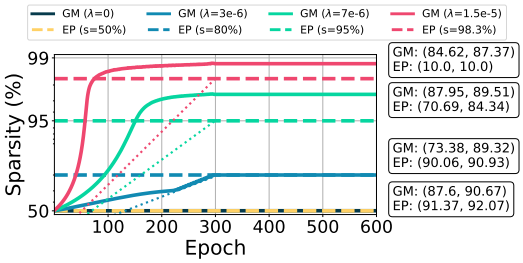

Figure 2: The sparsity of intermediate results, the accuracy of the final output, and the accuracy after finetuning on MobileNet-V2, CIFAR-10. For GEM-MINER (GM), we also visualize the sparsity upper bounds as dotted lines. As λ increases, note that the sparsity of GEM-MINER's output increases. For$\lambda = 3 \cdot 10^{-6}$, the iterative freezing algorithm kicks in around epoch 220, regularizing the sparsity thereafter. The gem found by GEM-MINER($\lambda = 1.5 \cdot 10^{-5}$) achieves an accuracy of 84.62% before finetuning and 87.37% after finetuning, while EP is unable to achieve non-trivial accuracy before or after finetuning at 98.3% sparsity.

In the following, we take GEM-MINER apart and describe the components that allow it to surpass these issues.

GEM-MINER without sparsity control. Much like EP, GEM-MINER employs a form of backpropagation, and works as follows. Each of the random weights $[\mathbf{w}_0]_i$ in the original network is associated with a normalized score $p_i \in [0, 1]$. These normalized scores become our optimization variables and are responsible for computing the *supermask* $\mathbf{m}$, *i.e.*, the pruning pattern of the network at initialization.

For a given set of weights $\boldsymbol{w}$ and scores $\boldsymbol{p}$, GEM-MINER sets the effective weights as $\boldsymbol{w}_{\text{eff}} = \boldsymbol{w} \odot r(\boldsymbol{p})$, where $\mathrm{r}(\cdot)$ is an element-wise rounding function, and $\mathbf{m} = r(\boldsymbol{p})$ is the resulting supermask. The rounding function can be changed, *e.g.*, r can perform randomized rounding, in which case p_i would be the probability of keeping weight w_i in $\mathbf{m}$. In our case, we found that simple deterministic rounding, *i.e.*, $r(p_i) = \mathbf{1}_{p_i \geq 0.5}$ works well.

At every iteration GEM-MINER samples a batch of training data and performs backpropagation on the loss of the effective weights, with respect to the scores $\boldsymbol{p}$, while projecting back to $[0, 1]$ when needed. During the forward pass, due to the rounding function, the effective network used is indeed a subnetwork of the given network. Here, since $r(\boldsymbol{p})$ is a non-differentiable operation we use the Straight Through Estimator (STE) [2] which backpropagates through the indicator function as though it were the identity function. Therefore, even pruned scores can receive non-zero gradients which allows them to revive over the course of training.

Note that this vanilla version of GEM-MINER is unable to exercise control over the final sparsity of the model. For reasons that will become evident in below, we will call this version of our algorithm GEM-MINER(0). There is already a stark difference from EP: GEM-MINER(0) will automatically find the optimal sparsity, while EP requires the target sparsity s as an input parameter.

However, at the same time, this also significantly limits the applicability of GEM-MINER(0) as one cannot obtain a highly sparse gem. Shown as a dark blue curve in Fig. 2 is the sparsity of GEM-MINER(0). Here, we run GEM-MINER with a randomly initialized MobileNet-V2 network on CIFAR-10. Note that the sparsity stays around 50% throughout the run, which is consistent with the observation by Ramanujan et al. [33] that accuracy of subnetworks at initialization is maximized at around 50% sparsity.

Algorithm 1: GEM-MINER

Input: Dataset $D = \{(\boldsymbol{x}_i, y_i)\}$, learning rate η, rounding function $r(\cdot)$, number of epochs E, freezing period T, target sparsity $s \in [0, 1]$

Output: Mask $\boldsymbol{m} = r(\boldsymbol{p}) \odot \boldsymbol{q} \in \{0, 1\}^d$

$c \leftarrow \frac{1}{E} \ln\left(\frac{1}{1-s}\right)$, $\boldsymbol{q} \leftarrow \mathbf{1}_d$
$\boldsymbol{w}, \boldsymbol{p} \leftarrow$ random vector in $\mathbb{R}^d$,
  random vector in $[0, 1]^d$
for j *in* $1, 2, \ldots, E$ **do**
  for $(\boldsymbol{x}_i, y_i) \in D$ **do**
    $\boldsymbol{w}_{\text{eff}} \leftarrow (\boldsymbol{w} \odot \boldsymbol{q}) \odot r(\boldsymbol{p})$
    $\boldsymbol{p} \leftarrow \boldsymbol{p} - \eta \nabla_{\boldsymbol{p}} \ell(f(\boldsymbol{w}_{\text{eff}}; \boldsymbol{x}_i), y_i)$
    /* STE */
    $\boldsymbol{p} \leftarrow \text{proj}_{[0,1]^d}\, \boldsymbol{p}$
  if $\text{mod}(j, T) = 0$ **then**
    $I_1 \leftarrow \{i : q_i = 1\}$
    $\boldsymbol{p}_{sorted} \leftarrow \text{sort}(\boldsymbol{p}_{i \in I_1})$
    $\boldsymbol{p}_{bottom} \leftarrow$ Bottom-$(1 - e^{-cj})$ fraction of $\boldsymbol{p}_{sorted}$
    $q \leftarrow q \odot \mathbb{1}_{p_i \notin \boldsymbol{p}_{bottom}}$

Regularization and Iterative freezing. GEM-MINER(0) is a good baseline algorithm for finding accurate subnetworks at initialization, but it cannot be used to find *rare gems*, which need to be sparse and trainable. To overcome this limitation, we apply a standard trick – we add a regularization term to encourage sparsity. Thus, in addition to the task loss computed with the effective weights, we also compute the L_2 or L_1 norm of the score vector $\boldsymbol{p}$ and optimize over the total regularized loss. More formally, we minimize $\ell := \ell_{\text{task}} + \lambda \ell_{\text{reg}}$, where λ is the hyperparameter and ℓ_{reg} is either L_2 or L_1 norm of the score vector $\boldsymbol{p}$.

We call this variant GEM-MINER(λ), where λ denotes the regularization weight. This naming convention should explain why we called the initial version GEM-MINER(0).

The experimental results in Fig. 2 show that this simple modification indeed allows us to control the sparsity of the solution. We chose to use the L_2 regularizer, however preliminary experiments showed that L_1 performs almost identically. By varying λ from $\lambda = 0$ to $\lambda = 7 \cdot 10^{-6}$ and $\lambda = 1.5 \cdot 10^{-5}$, the final sparsity of the gem found by GEM-MINER(λ) becomes 97.5% and 98.6%, respectively.

One drawback of this regularization approach is that it only indirectly controls the sparsity. If we have a target sparsity s, then there is no easy way of finding the appropriate value of λ such that the resulting subnetwork is s-sparse. If we choose λ to be too large, then it will give us a gem that is way too sparse; too small a λ and we will end up with a denser gem than what is needed. As a simple heuristic, we employ *iterative freezing*, which is widely used in several existing pruning algorithms, including IMP [9, 12, 45]. More specifically, we can design an exponential function $\overline{s}(j) = 1 - e^{-cj}$ for some $c > 0$, which will serve as the upper bound on the sparsity. If the total number of epochs is E and the target sparsity is s, we have $\overline{s}(E) = 1 - e^{-cE} = s$. Thus, we have $c = \frac{1}{E} \ln\left(\frac{1}{1-s}\right)$.

Once this sparsity upper bound is designed, the iterative freezing mechanism regularly checks the current sparsity to see if the lower bound is violated or not. If the sparsity bound is violated, it finds the smallest scores, zeros them out, and freezes their values thereafter. By doing so, we can guarantee the final sparsity even when λ was not sufficiently large. To see this freezing mechanism in action, refer the blue curve in Fig. 2. Here, the sparsity lower bounds (decreasing exponential functions) are visualized as dotted lines. Note that for the case of $\lambda = 3 \cdot 10^{-6}$, the sparsity of the network does

not decay as fast as desired, so it touches the sparsity lower bound around epoch 220. The iterative freezing scheme kicks in here, and the sparsity decay is controlled by the lower bound thereafter, achieving the specified target sparsity at the end.

The full pseudocode of GEM-MINER is provided in Algorithm 1. There are two minor implementation details which differ from the explanation above: (i) we impose the iterative freezing every T epochs, not every epoch and (ii) iterative freezing is imposed even when the sparsity bound is not violated.

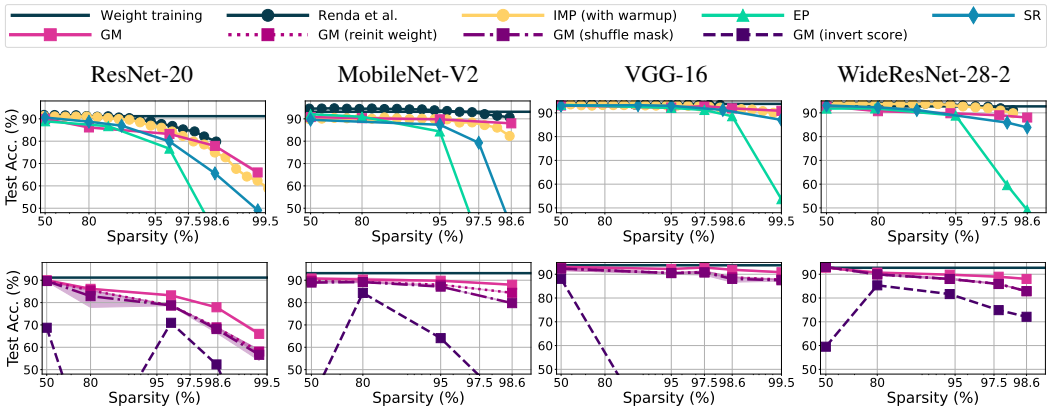

Figure 3: Performance of different pruning algorithms on CIFAR-10 for benchmark networks. Top: post-finetune accuracy; Bottom: sanity check methods suggested in Frankle et al. [11] applied on GEM-MINER (GM). Note that GM achieves similar post-finetune accuracy as IMP, and typically outperforms it in the sparse regime. GM has higher post-finetune accuracy than EP and Smart Ratio (SR). GM also passes the sanity checks suggested in Frankle et al. [11]. Finally, GM (which prunes *at* init) nearly achieves the performance of Renda et al. (which is a pruning after training method) in the sparse regime, e.g., 98.6% sparsity in ResNet-20.

4 Experiments

In this section, we present the experimental results[2] for the performance of GEM-MINER across various tasks.

Tasks. We evaluate our algorithm on **(Task 1)** CIFAR-10 classification, on ResNet-20, MobileNet-V2, VGG-16, and WideResNet-28-2, **(Task 2)** TinyImageNet classification on ResNet-18 and ResNet-50, **(Task 3)** Finetuning on the Caltech-101 [7] dataset using a ResNet-50 pretrained on ImageNet, and **(Task 4)** CIFAR-100 classification using ResNet-32. The detailed description of the datasets, networks and hyperparameters can be found in Section A of the Appendix.

Proposed scheme. We run GEM-MINER with an L_2 regularizer. If a network reaches its best accuracy after E epochs of dense training, then we run GEM-MINER for E epochs from random init to get a sparse subnetwork *at initialization*, and then run weight training on the sparse subnetwork for another E epochs.

Comparisons. We tested our method against the following baselines: dense weight training and four pruning algorithms: (i) IMP [10], (ii) Learning rate rewinding [34], denoted by Renda et al., (iii) Edge-Popup (EP) [33], and (iv) Smart-Ratio (SR) which is the random pruning method proposed by Su et al. [41].

We also ran the following sanity checks, proposed by Frankle et al. [11]: (i) (Random shuffling): To test if the algorithm prunes specific connections, we randomly shuffle the mask at every layer. (ii) (Weight reinitialization): To test if the final mask is specific to the weight initialization, we reinitialize the weights from the original distribution. (iii) (Score inversion): Since most pruning algorithms use a heuristic/score function as a proxy to measure the importance of different weights, we invert the scoring function to check whether it is a valid proxy. More precisely, this test involves pruning the weights which have the *smallest* scores rather than the largest. In all of the above tests, if the accuracy after finetuning the new subnetwork does not deteriorate significantly, then the algorithm is merely identifying optimal layerwise sparsities.

[2]Our codebase can be found at `https://github.com/ksreenivasan/pruning_is_enough`.

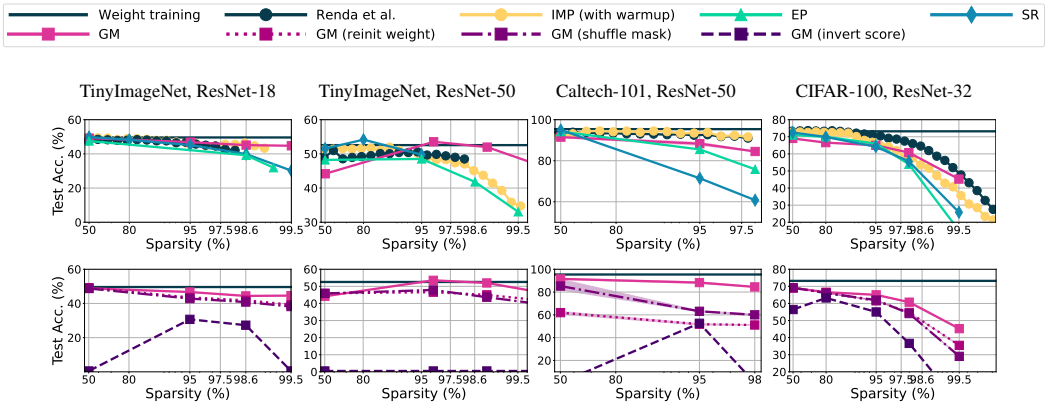

Figure 4: Accuracy on image classification tasks on TinyImageNet, Caltech-101 and CIFAR-100. For Caltech-101, we pruned a pre-trained ImageNet model (ResNet-50). Top: post-finetune accuracy, bottom: sanity check methods suggested in Frankle et al. [11] applied on GEM-MINER.

4.1 Rare gems obtained by GEM-MINER

Task 1. Fig. 3 shows the sparsity-accuracy tradeoff for various pruning methods trained on CIFAR-10 using ResNet-20, MobileNet-V2, VGG-16 and WideResNet-28-2. For each column (network), we compare IMP, IMP with learning rate rewinding (Renda et al.), GEM-MINER, EP, and SR in two performance metrics: the top row shows the accuracy of the subnetwork after weight training and bottom row shows the result of the sanity checks on GEM-MINER.

As shown in the top row of Fig. 3, GEM-MINER finds a lottery ticket *at* initialization. It reaches accuracy similar to IMP after weight training. Moreover, in the sparse regime (e.g., above 98.6% for ResNet-20 and MobileNet-V2), GEM-MINER outperforms IMP in terms of post-finetune accuracy. The bottom row of Fig. 3 shows that GEM-MINER passes the sanity check methods. For all networks, the performance in the sparse regime (98.6% sparsity or above) shows that the suggested GEM-MINER algorithm enjoys 3–10% accuracy gap with the best performance among variants. The results in the top row show that GEM-MINER far outperforms the random network with smart ratio (SR).

Tasks 2–4. Fig. 4 shows the sparsity-accuracy tradeoff for Tasks 2–4. Similar to Fig. 3, the top row reports the accuracy *after* weight training, and the bottom row contains the results of the sanity checks.

As shown in Fig. 4a and Fig. 4b, the results for **Task 2** show that (i) GEM-MINER achieves accuracy comparable to IMP as well as Renda et al. (IMP with learning rate rewinding) even in the sparse regime, (ii) GEM-MINER has non-trivial accuracy before finetuning (iii) GEM-MINER passes all the sanity checks, and (iv) GEM-MINER outperforms EP and SR. These results show that GEM-MINER successfully finds rare gems even in the sparse regime for **Task 2**.

Fig. 4c shows the result for **Task 3**. Unlike other tasks, GEM-MINER does not reach the post-finetune accuracy of IMP, but GEM-MINER enjoys over an 8% accuracy gap compared with EP and SR. Moreover, the bottom row shows that GEM-MINER has over 20% higher accuracy than the sanity checks above 95% sparsity showing that the subnetwork found by GEM-MINER is unique in this sparse regime.

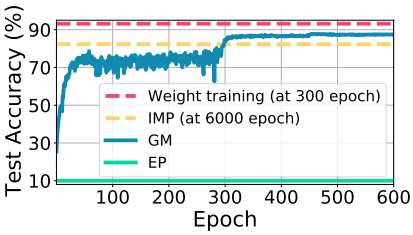

Figure 5: Convergence plot for CIFAR-10, MobileNet-V2 experiments, where we apply GEM-MINER for 300 epochs and then finetune the sparse model for another 300 epochs, to reach 98.6% sparse model. We include the accuracy of dense weight training, IMP and EP (98.6% sparse model) as references. Note that the comparison is with 300 epochs of weight training, and IMP using 20 rounds of iterative pruning, i.e., 300 $\times$ 20 = 6000 epochs, to reach 98.6% sparsity. GEM-MINER achieves a higher accuracy than IMP despite its 19$\times$ shorter runtime to find a sparse subnetwork.

Fig. 4d shows the result for **Task 4** where once again, GEM-MINER is comparable to IMP throughout and outperforms it in the sparse regime.

Table 2: We compare ProsPr [1] vs GEM-MINER on ResNet-20, CIFAR-10 and run the random shuffling as well as the weight reinit sanity checks. Note that GEM-MINER produces a subnetwork that is higher accuracy despite being more sparse. Moreover, ProsPr does not show significant decay in performance after the sanity checks while GEM-MINER does. Therefore, it is likely that ProsPr is merely identifying good layerwise sparsity ratios.

Algorithm	Sparsity	Accuracy after finetune	Accuracy after Random shuffling	Accuracy after Weight reinitialization
ProsPr	95%	82.67%	82.15%	81.64%
GEM-MINER	**96.28%**	**83.4%**	**78.73%**	**78.6%**

4.2 Comparison to ProsPr

Alizadeh et al. [1] recently proposed a pruning at init method called ProsPr which utilizes meta-gradients through the first few steps of optimization to determine which weights to prune, thereby accounting for the "trainability" of the resulting subnetwork. In Table 2 we compare it against GEM-MINER on ResNet-20, CIFAR-10 and also run the (i) Random shuffling and (ii) Weight reinitialization sanity checks from Frankle et al. [11]. We were unable to get ProsPr using their publicly available codebase to generate subnetworks at sparsity above 95% and therefore chose that sparsity. Note that GEM-MINER produces a subnetwork that is higher accuracy despite being more sparse. After finetuning for 150 epochs, our subnetwork reaches 83.4% accuracy while the subnetwork found by ProsPr only reaches 82.67% after training for 200 epochs. More importantly, ProsPr does not show significant decay in performance after the random reshuffling or weight reinitialization sanity checks. Therefore, as Frankle et al. [11] remark, it is likely that it is identifying good layerwise sparsity ratios, rather than a mask specific to the initialized weights.

4.3 Observations on GEM-MINER

Convergence of accuracy and sparsity. Fig. 5 shows how the accuracy of GEM-MINER improves as training progresses, for MobileNet-V2 on CIFAR-10 at sparsity 98.6%. This shows that GEM-MINER, reaches high accuracy even early in training, and can be finetuned to accuracy higher than that of IMP (which requires $19\times$ the runtime than our algorithm). EP remains at random-guessing throughout as it typically does not work well in the sparse regime.

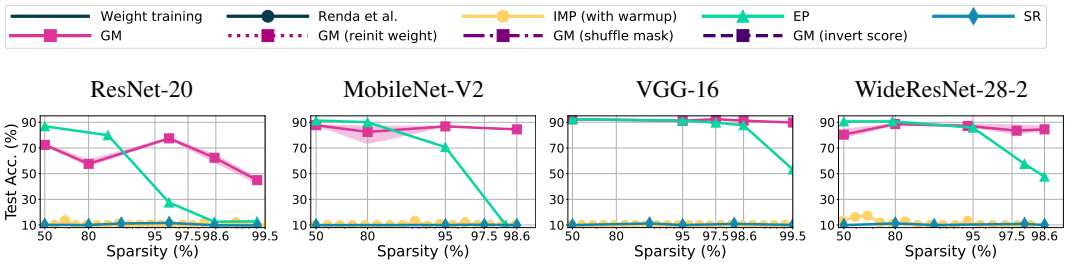

Figure 6: Performance of different pruning algorithms before finetuning on CIFAR-10 for benchmark networks. GEM-MINER finds subnetworks that already have reasonably high accuracy even before weight training. Note that, while IMP and SR have scarcely better than random guessing at initialization, subnetworks found by GEM-MINER typically perform even better than EP, especially in the sparse regime.

High pre-finetune accuracy. As shown in Fig. 6, GEM-MINER finds subnetworks at initialization that have a reasonably high accuracy even before the weight training, e.g., above 90% accuracy for 98.6% sparsity in VGG-16, and 85% accuracy for 98.6% sparsity in MobileNet-V2. Note that, in contrast, IMP and SR have accuracy scarcely better than random guessing at initialization. Clearly, GEM-MINER fulfills its objective in maximizing accuracy before finetuning and therefore finds rare gems – lottery tickets at initialization which already have high accuracy.

Limitations of GEM-MINER. We observed that in the dense regime (50% sparsity, 80% sparsity), GEM-MINER sometimes performs worse than IMP. While we believe that this can be resolved by appropriately tuning the hyperparameters, we chose to focus our attention on the sparse regime. We would also like to remark that GEM-MINER is fairly sensitive to the choice of hyperparameters and for some models, we had to choose different hyperparameters for each sparsity to ensure optimal performance. Though this occurs rarely, we also find that an extremely aggressive choice of λ can

Table 3: We construct different variants of EP and compare their performance with GEM-MINER, for ResNet-20, CIFAR-10, 99.41% sparsity. We establish that having a global score metric and gradually pruning is key to improved performance.

Pruning Method	EP	Global EP	Global EP with Gradual Pruning	Global EP with Gradual Pruning and Regularization	GEM-MINER
Pre-finetune acc (%)	19.57	22.22	31.56	19.67	**45.30**
Post-finetune acc (%)	24.47	34.42	63.54	63.72	**66.15**

lead to *layer-collapse* where one or more layers gets pruned completely. This happens when all the scores $\boldsymbol{p}$ of that layer drop below 0.5.

Layer-wise sparsity. We compare the layer-wise sparsities of different algorithms for ResNet-20 on CIFAR-10 in Fig. 7. Both GEM-MINER and IMP spend most of their sparsity budget on the first and last layers. SR assigns 70% sparsity to the last layer and the sparsity increases smoothly across the others. EP maintains the target sparsity ratio at each layer and therefore is always a horizontal line.

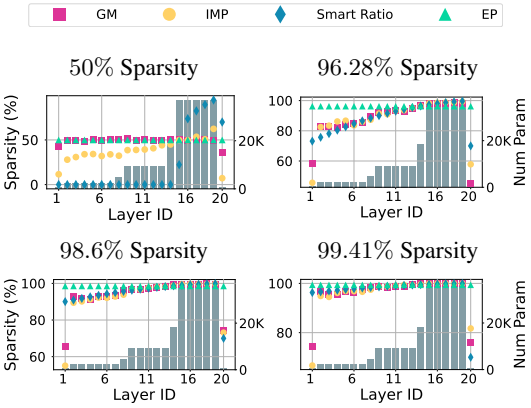

Figure 7: The layerwise sparsity for ResNet-20 pruned by GEM-MINER, IMP, Smart Ratio, and EP. The dark bar is the layerwise number of parameters. Both GEM-MINER and IMP utilize most of the sparsity budget for the first and last layers.

How does GEM-MINER resolve EP's failings? An open problem from Ramanujan et al. [33] is why most subnetworks found by EP are not fine-tunable. While GEM-MINER is significantly different from EP, it is reasonable to ask which modification allowed it to find lottery tickets without forgoing high accuracy at initialization. Table 3 explores this question for ResNet-20, CIFAR-10 at 99.41% sparsity. Here, we compare EP, GEM-MINER, as well as three EP variants that we construct. (i) *(Global EP)* is a modification where the bottom-k scores are pruned globally, not layer-wise. This allows the algorithm to trade-off sparsity in one layer for another. (ii) *(Gradual pruning)* reduces the parameter k gradually as opposed to setting it to the target sparsity from the beginning. (iii) *(Regularization)*: we add an L_2 term on the score p of the weights to encourage sparsity. The results indicate that *global pruning* and *gradual pruning* significantly improve both the pre and post-finetune accuracies of EP. Adding regularization does not improve the performance significantly. Finally, adding all three features to EP allows it to achieve 63.72% accuracy, while GEM-MINER reaches 66.15% accuracy. It is important to note that even with all three features, EP is inherently different from GEM-MINER in how it computes the *supermask* based on the scores. But we conjecture that aggressive, layerwise pruning is the key reason for EP's failings.

Ablation study on GEM-MINER. To better understand the relative importance of the different components of the algorithm, we do a more thorough ablation study in the same vein as the above analysis. We consider the setting of ResNet-20, CIFAR-10 at a sparsity of 98.56% and evaluate the performance of (i) GEM-MINER, (ii) GEM-MINER without regularization and (iii) GEM-MINER without regularization and without global pruning. As shown in Table 4, GEM-MINER outperforms all of its ablated versions. In fact, GEM-MINER without regularization works extremely poorly, and the final variant is barely above random guessing. Note that when we ablate global pruning, the algorithm chooses the bottom$-k$ weights in each layer much like EP.

Table 4: Ablation study of GEM-MINER on ResNet-20, CIFAR-10 at 98.56% sparsity. GM outperforms all of its variants and in fact, when we ablate regularization and global pruning, the performance is barely above random guessing. (GM - Regularization) denotes GM without regularization.

GEM-MINER variant	Accuracy before FT (%)	Accuracy after FT (%)
GM	**61.23**	**77.12**
GM - Regularization	10.18	27.41
GM - Regularization - Global Pruning	10.08	11.6

Table 5: Comparison of GEM-MINER and its longer version, for ResNet-20, CIFAR-10 at 98.6% sparsity. LONG GM, when given the same number of epochs rivals the performance of Renda et al. [34]

Method	GM (cold)	Long GM (cold)	IMP (warm)	Renda et al. (pruning after training)
Number of Epochs	300	3000	3000	3000
Accuracy (%)	77.89	79.50	74.52	80.21

Applying GEM-MINER for longer periods. Recall that GEM-MINER uses 19 $\times$ fewer epochs than methods like IMP [10] and Learning rate rewinding (Renda et al. [34]), to find a subnetwork at 98.6% sparsity which can then be trained to high accuracy. Here, we consider a long version of GEM-MINER to see if it can benefit if it is allowed to run for longer. Table 5 shows the comparison of post-finetune accuracy for GEM-MINER, LONG GEM-MINER, IMP and Renda et al. [34] tested on ResNet-20, CIFAR-10 at 98.5% sparsity. Regular GEM-MINER, applies iterative freezing every 5 epochs to arrive at the target sparsity in 150 epochs. LONG GEM-MINER instead prunes every 150 epochs and therefore reaches the target sparsity in 3000 epochs. We find that applying GEM-MINER for longer periods improves the post-finetune accuracy in this regime by 1.5%. This shows that given equal number of epochs, GEM-MINER, which prunes at initialization, can close the gap to Learning rate rewinding [34] which is a prune-after-training method.

5 Conclusion

In this work, we resolve the open problem of pruning at initialization by proposing GEM-MINER, an algorithm that finds *rare gems* – lottery tickets *at initialization* that have non-trivial accuracy even before finetuning, and accuracy rivaling prune-after-train methods after finetuning. Unlike other methods, subnetworks found by GEM-MINER pass all known sanity checks and baselines. Moreover, we show that GEM-MINER is competitive with IMP despite not using warmup and up to 19$\times$ faster.

Acknowledgements

The authors would like to thank Jeff Linderoth for early discussions on viewing pruning as an integer programming problem. This research was supported by ONR Grant No. N00014-21-1-2806 and NSF/Intel Partnership on Machine Learning for Wireless Networking Program under Grant No. CNS-2003129.

References

[1] Alizadeh, M., Tailor, S. A., Zintgraf, L. M., van Amersfoort, J., Farquhar, S., Lane, N. D., and Gal, Y. Prospect pruning: Finding trainable weights at initialization using meta-gradients. In *International Conference on Learning Representations*, 2021. 3, 8

[2] Bengio, Y., Léonard, N., and Courville, A. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432*, 2013. 5

[3] Chen, T., Frankle, J., Chang, S., Liu, S., Zhang, Y., Wang, Z., and Carbin, M. The lottery ticket hypothesis for pre-trained bert networks. *arXiv preprint arXiv:2007.12223*, 2020. 2

[4] Chen, T., Sui, Y., Chen, X., Zhang, A., and Wang, Z. A unified lottery ticket hypothesis for graph neural networks. In *International Conference on Machine Learning*, pp. 1695–1706. PMLR, 2021. 2

[5] Courbariaux, M., Bengio, Y., and David, J.-P. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in neural information processing systems*, pp. 3123–3131, 2015. 4

[6] Diffenderfer, J. and Kailkhura, B. Multi-prize lottery ticket hypothesis: Finding accurate binary neural networks by pruning a randomly weighted network. In *International Conference on Learning Representations*, 2020. 2

[7] Fei-Fei, L., Fergus, R., and Perona, P. Learning generative visual models from few training examples: An incremental bayesian approach tested on 101 object categories. In *2004 conference on computer vision and pattern recognition workshop*, pp. 178–178. IEEE, 2004. 6, 14

[8] Fischer, J. and Burkholz, R. Towards strong pruning for lottery tickets with non-zero biases. *arXiv preprint arXiv:2110.11150*, 2021. 15

[9] Frankle, J. and Carbin, M. The lottery ticket hypothesis: Finding sparse, trainable neural networks. *arXiv preprint arXiv:1803.03635*, 2018. 1, 2, 3, 5

[10] Frankle, J., Dziugaite, G. K., Roy, D., and Carbin, M. Linear mode connectivity and the lottery ticket hypothesis. In *International Conference on Machine Learning*, pp. 3259–3269. PMLR, 2020. 1, 2, 6, 10, 16, 22

[11] Frankle, J., Dziugaite, G. K., Roy, D. M., and Carbin, M. Pruning neural networks at initialization: Why are we missing the mark? *arXiv preprint arXiv:2009.08576*, 2020. 1, 2, 3, 4, 6, 7, 8, 16, 19, 20

[12] Gale, T., Elsen, E., and Hooker, S. The state of sparsity in deep neural networks. *arXiv preprint arXiv:1902.09574*, 2019. 1, 3, 5

[13] Girish, S., Maiya, S. R., Gupta, K., Chen, H., Davis, L. S., and Shrivastava, A. The lottery ticket hypothesis for object recognition. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 762–771, 2021. 2

[14] Han, S., Pool, J., Tran, J., and Dally, W. Learning both weights and connections for efficient neural network. In *Advances in neural information processing systems*, pp. 1135–1143, 2015. 1

[15] Han, S., Mao, H., and Dally, W. J. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. *arXiv:1510.00149 [cs]*, February 2016. URL `http://arxiv.org/abs/1510.00149`.

[16] Hassibi, B. and Stork, D. G. Second order derivatives for network pruning: Optimal Brain Surgeon. In Hanson, S. J., Cowan, J. D., and Giles, C. L. (eds.), *Advances in Neural Information Processing Systems 5*, pp. 164–171. Morgan-Kaufmann, 1993. 1

[17] He, K., Zhang, X., Ren, S., and Sun, J. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pp. 1026–1034, 2015. 15

[18] He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016. 15

[19] Hubara, I., Courbariaux, M., Soudry, D., El-Yaniv, R., and Bengio, Y. Binarized neural networks. In *Advances in neural information processing systems*, pp. 4107–4115, 2016. 4

[20] Hubara, I., Courbariaux, M., Soudry, D., El-Yaniv, R., and Bengio, Y. Quantized neural networks: Training neural networks with low precision weights and activations. *The Journal of Machine Learning Research*, 18(1):6869–6898, 2017. 1

[21] Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014. 16

[22] Krizhevsky, A. et al. Learning multiple layers of features from tiny images. 2009. 14, 15

[23] LeCun, Y., Denker, J. S., and Solla, S. A. Optimal Brain Damage. In Touretzky, D. S. (ed.), *Advances in Neural Information Processing Systems 2*, pp. 598–605. Morgan-Kaufmann, 1990. URL `http://papers.nips.cc/paper/250-optimal-brain-damage.pdf`. 1

[24] Lee, N., Ajanthan, T., and Torr, P. H. Snip: Single-shot network pruning based on connection sensitivity. *arXiv preprint arXiv:1810.02340*, 2018. 2, 3

[25] Li, A., Sun, J., Wang, B., Duan, L., Li, S., Chen, Y., and Li, H. Lotteryfl: Personalized and communication-efficient federated learning with lottery ticket hypothesis on non-iid datasets. *arXiv preprint arXiv:2008.03371*, 2020. 2

[26] Lubana, E. S. and Dick, R. A gradient flow framework for analyzing network pruning. In *International Conference on Learning Representations*, 2020. 3

[27] Ma, X., Yuan, G., Shen, X., Chen, T., Chen, X., Chen, X., Liu, N., Qin, M., Liu, S., Wang, Z., et al. Sanity checks for lottery tickets: Does your winning ticket really win the jackpot? *Advances in Neural Information Processing Systems*, 34, 2021. 3

[28] Malach, E., Yehudai, G., Shalev-Schwartz, S., and Shamir, O. Proving the lottery ticket hypothesis: Pruning is all you need. In *International Conference on Machine Learning*, pp. 6682–6691. PMLR, 2020. 2

[29] Mozer, M. C. and Smolensky, P. Skeletonization: A Technique for Trimming the Fat from a Network via Relevance Assessment. In Touretzky, D. S. (ed.), *Advances in Neural Information Processing Systems 1*, pp. 107–115. Morgan-Kaufmann, 1989. 1

[30] Orseau, L., Hutter, M., and Rivasplata, O. Logarithmic pruning is all you need. *Advances in Neural Information Processing Systems*, 33, 2020. 2

[31] Patil, S. M. and Dovrolis, C. Phew: Constructing sparse networks that learn fast and generalize well without training data. In *International Conference on Machine Learning*, pp. 8432–8442. PMLR, 2021. 3

[32] Pensia, A., Rajput, S., Nagle, A., Vishwakarma, H., and Papailiopoulos, D. Optimal lottery tickets via subsetsum: Logarithmic over-parameterization is sufficient. *Advances in neural information processing systems*, 2020. 2

[33] Ramanujan, V., Wortsman, M., Kembhavi, A., Farhadi, A., and Rastegari, M. What's hidden in a randomly weighted neural network? In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 11893–11902, 2020. 3, 4, 5, 6, 9, 15, 21

[34] Renda, A., Frankle, J., and Carbin, M. Comparing rewinding and fine-tuning in neural network pruning. *arXiv preprint arXiv:2003.02389*, 2020. 3, 6, 10

[35] Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., and Chen, L.-C. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 4510–4520, 2018. 16

[36] Sanh, V., Wolf, T., and Rush, A. Movement pruning: Adaptive sparsity by fine-tuning. *Advances in Neural Information Processing Systems*, 33:20378–20389, 2020. 4

[37] Savarese, P., Silva, H., and Maire, M. Winning the lottery with continuous sparsification. *Advances in Neural Information Processing Systems*, 33:11380–11390, 2020. 2

[38] Simons, T. and Lee, D.-J. A review of binarized neural networks. *Electronics*, 8(6):661, 2019. 4

[39] Simonyan, K. and Zisserman, A. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014. 16

[40] Sreenivasan, K., Rajput, S., Sohn, J.-y., and Papailiopoulos, D. Finding everything within random binary networks. *arXiv preprint arXiv:2110.08996*, 2021. 2

[41] Su, J., Chen, Y., Cai, T., Wu, T., Gao, R., Wang, L., and Lee, J. D. Sanity-checking pruning methods: Random tickets can win the jackpot. *arXiv preprint arXiv:2009.11094*, 2020. 1, 2, 3, 6, 19, 20, 21

[42] Tanaka, H., Kunin, D., Yamins, D. L., and Ganguli, S. Pruning neural networks without any data by iteratively conserving synaptic flow. *Advances in Neural Information Processing Systems*, 33, 2020. 2, 3

[43] Wang, C., Zhang, G., and Grosse, R. Picking winning tickets before training by preserving gradient flow. In *International Conference on Learning Representations*, 2019. 2, 3

[44] Zagoruyko, S. and Komodakis, N. Wide residual networks. *arXiv preprint arXiv:1605.07146*, 2016. 15

[45] Zhu, M. and Gupta, S. To prune, or not to prune: exploring the efficacy of pruning for model compression. *arXiv preprint arXiv:1710.01878*, 2017. 1, 3, 5

Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes]
   (b) Did you describe the limitations of your work? [Yes] Refer Section 4.3
   (c) Did you discuss any potential negative societal impacts of your work? [N/A]
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [N/A]
   (b) Did you include complete proofs of all theoretical results? [N/A]
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes] Our codebase can be found at `https://github.com/ksreenivasan/pruning_is_enough`
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes] Refer the appendix
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [Yes] Wherever it was computationally feasible, we ran 3 trials and report the plots with an error bar
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes] Refer the appendix
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [Yes] We cite the original papers that introduced ResNets, VGGs etc.

(b) Did you mention the license of the assets? [Yes] We cite the original papers that introduced CIFAR-10, TinyImagenet etc.

(c) Did you include any new assets either in the supplemental material or as a URL? [No]

(d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A]

(e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A]

5. If you used crowdsourcing or conducted research with human subjects...

(a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]

(b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]

(c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]

Contents of the Appendix

A Experimental Setup **14**

A.1 Dataset 14

A.2 Model 15

A.3 Hyper-Parameter Configuration 16

A.3.1 GEM-MINER Training 16

A.3.2 Finetuning the Rare Gems 17

B Additional Experiments **19**

A Experimental Setup

In this section, we introduce the datasets (A.1) and models (A.2) that we used in the experiments. We also report the detailed hyperparameter choices (A.3) of GEM-MINER each network and sparsity level. For competing methods, we used hyperparameters used by the original authors whenever possible. In other cases, we tried SGD (with momentum) and Adam optimizers, initial learning rate (LR) of η, 0.1η, 10η, and cosine/multi- step LR decay, where η is the best LR for weight training. All of our experiments are run using PyTorch 1.11 on Nvidia 3090 TIs and Nvidia V100s.

A.1 Dataset

In the experiments, we demonstrate the performance of GEM-MINER across various datasets. For each dataset, we optimize the training loss, and tune hyperparameters based on the validation accuracy. The test accuracy is reported for the best model chosen based on the validation accuracy.

CIFAR-10. CIFAR-10 consists of $60,000$ images from 10 classes, each with size 32×32, of which 50000 images are used for training, and $10,000$ images are for testing [22]. For data processing, we follow the standard augmentation: normalize channel-wise, randomly horizontally flip, and random cropping. For hyperparameter tuning we randomly split the train set into 45000 train images and retain 5000 images as the validation set. Once the hyperparameters are chosen, we retrain on the full train set and report test accuracy.

TinyImageNet. TinyImageNet contains 100000 images of 200 classes (500 each class), which are downsized to 64×64 colored images. Each class has 500 training images, 50 validation images and 50 test images. Augmentation includes normalizing, random rotation and random flip. Train set, validation set, and test set are provided.

Caltech-101. Caltech-101 contains figures of objects from 101 categories. There are around 40 to 800 images per category, and most categories have about 50 images [7]. The size of each image is roughly 300×200 pixels. When processing the image, we resize each figure to 224×224, and normalize it across channels. We split 20% of the data to be test set, and in the remaining training set, we retain 25% as the validation set, giving us train/val/test = $60\%/20\%/20\%$ split.

CIFAR-100. The CIFAR-100 dataset is just like CIFAR-10 except that it has 100 classes containing 600 images each. Therefore, it has $60,000$ images from 100 classes, each with size 32×32, of which 50000 images are used for training, and $10,000$ images are for testing [22]. For data processing, we follow the standard augmentation: normalize channel-wise, randomly horizontally flip, and random cropping. For hyperparameter tuning we randomly split the train set into 45000 train images and retain 5000 images as the validation set. Once the hyperparameters are chosen, we retrain on the full train set and report test accuracy.

A.2 Model

Unless otherwise specified, in all of our experiments experiments, we use Non-Affine BatchNorm, and disable bias for all the convolution and linear layers. We find that most implementations of pruning algorithms instead use them and merely ignore them while pruning and while computing sparsity. While they do not alter sparsity by much (since there are few parameters when compared to weights), we still find this to be inaccurate. Moreover, it is not obvious how to prune biases – should they be treated as weight? Should they be treated as a different set of parameters? In order to make sure we compare all the baselines on the same platform, we decided that eliminating them was the fair choice (As [8] describe, pruning with biases is an interesting problem but needs to be handled slightly carefully). We use uniform initialization for scores, signed constant initialization [33] for weight parameters for GEM-MINER while dense training initializes the weights using the standard Kaiming normal initializaiton [17].

The networks that we used in our experiments are summarized as follows.

ResNet-18, ResNet-20, ResNet-32 and ResNet-50 [18]. We follow the standard ResNet architecture. ResNet-20 is designed for CIFAR-10, ResNet-32 is for CIFAR-100, while ResNet-18 and ResNet-50 are for ImageNet. In Table 6, we use the convention [kernel $\times$ kernel, output]$\times$(times repeated) for the convolution layers found in ResNet blocks. We base our implementation on the following GitHub repository[3].

WideResNet-28-2 [44]. We base our implementation on the following GitHub repository[4]. Architecture details can be found in Table 6.

Table 6: ResNet architecture used in our experiments. The output layer of the network is changed according to the dataset. For example, ResNet50 is used for both TinyImageNet and ImageNet so we changed the output dimension to 200 and 1000 respectively.

Layer	ResNet-20	ResNet-18	ResNet-50	WideResNet-28-2
Conv 1	3×3, 16 padding 1 stride 1	3×3, 64 padding 3 stride 2	7×7, 64 padding 3 stride 2	3×3, 16 padding 1 stride 1
		Max Pool, kernel size 3, stride 2, padding 1		
Layer stack 1	$\begin{bmatrix} 3\times3, 16 \\ 3\times3, 16 \end{bmatrix}\times3$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 3\times3, 32 \\ 3\times3, 32 \end{bmatrix}\times4$
Layer stack 2	$\begin{bmatrix} 3\times3, 32 \\ 3\times3, 32 \end{bmatrix}\times3$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times4$
Layer stack 3	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$
Layer stack 4	-	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	-
FC	Avg Pool, kernel size 8 $64 \times$ N_CLASSES	Adaptive Avg Pool, output size $(1,1)$ $512 \times$ N_CLASSES	$2048 \times$ N_CLASSES	Avg Pool, kernel size 8 $128 \times$ N_CLASSES

[3] https://github.com/akamaster/pytorch_resnet_cifar10/blob/master/resnet.py
[4] https://github.com/xternalz/WideResNet-pytorch/blob/master/wideresnet.py

VGG-16 [39]. In the original VGG-16 network, there are 13 convolution layers and 3 FC layers (including the last linear classification layer). We follow the VGG-16 architectures used in [10, 11] to remove the first two FC layers while keeping the last linear classification layer. This finally leads to a 14-layer architecture, but we still call it VGG-16 as it is modified from the original VGG-16 architectural design. Detailed architecture is shown in Table 7. We base our implementation on the GitHub repository[5].

Table 7: Detailed architecture of the VGG-16 architecture used in our experiments. We have a non-affine batchnnorm layer followed by a ReLU activation after each convolutional layer (omitted in the table). The shapes for convolution layers follow (c_{in}, c_{out}, k, k).

Parameter	Shape	Layer hyper-parameter
layer1.conv1.weight	$3 \times 64 \times 3 \times 3$	stride:1;padding:1
layer2.conv2.weight	$64 \times 64 \times 3 \times 3$	stride:1;padding:1
pooling.max	N/A	kernel size:2;stride:2
layer3.conv3.weight	$64 \times 128 \times 3 \times 3$	stride:1;padding:1
layer4.conv4.weight	$128 \times 128 \times 3 \times 3$	stride:1;padding:1
pooling.max	N/A	kernel size:2;stride:2
layer5.conv5.weight	$128 \times 256 \times 3 \times 3$	stride:1;padding:1
layer6.conv6.weight	$256 \times 256 \times 3 \times 3$	stride:1;padding:1
layer7.conv7.weight	$256 \times 256 \times 3 \times 3$	stride:1;padding:1
pooling.max	N/A	kernel size:2;stride:2
layer8.conv9.weight	$256 \times 512 \times 3 \times 3$	stride:1;padding:1
layer9.conv10.weight	$512 \times 512 \times 3 \times 3$	stride:1;padding:1
layer10.conv11.weight	$512 \times 512 \times 3 \times 3$	stride:1;padding:1
pooling.max	N/A	kernel size:2;stride:2
layer11.conv11.weight	$512 \times 512 \times 3 \times 3$	stride:1;padding:1
layer12.conv12.weight	$512 \times 512 \times 3 \times 3$	stride:1;padding:1
layer13.conv13.weight	$512 \times 512 \times 3 \times 3$	stride:1;padding:1
pooling.max	N/A	kernel size:2;stride:2
pooling.avg	N/A	kernel size:1;stride:1
layer14.conv14.weight	$512 \times 10 \times 1 \times 1$	stride:1;padding:1

MobileNet-V2 [35]. We base our implementation on the GitHub repository.[6] Details of the architecture is shown in Table 8.

A.3 Hyper-Parameter Configuration

In this section, we will state the hyperparameter configuration for GEM-MINER and finetuning lottery tickets. For each dataset, model and different target sparsity, we tuned our hyperparameters for GEM-MINER by trying out different values of learning rate and L2 regularization weight λ. We also test different pruning periods of 5, 8, and 10 epochs. Finally, we also tried ADAM [21] and SGD. While SGD usually comes out on top, there were some settings where ADAM performed better.

A.3.1 GEM-MINER Training

We tested the CIFAR-10 dataset on the following architectures: i) ResNet-20 ii) MobileNet-V2 iii) VGG-16 iv) WideResNet-28-2. For TinyImageNet, we test on the architectures: i) ResNet-18 ii) ResNet-50. We tested the transfer learning on pretrained ImageNet model, where the target task is classification on Caltech-101 dataset with 101 classes. We first loaded the ResNet-50 model pretrained for ImageNet[7] and changed the last layer to a single fully-connected network having size 2048×101. To match the performance of the pretrained model, we used Affine BatchNorm. For

[5] https://github.com/kuangliu/pytorch-cifar/blob/master/models/vgg.py
[6] https://github.com/kuangliu/pytorch-cifar/blob/master/models/mobilenetv2.py
[7] https://pytorch.org/vision/stable/models.html

Table 8: The MobileNet-V2 structure that we use. Each layer consists of 3 total SubnetConv layer which correspond to the respective matrix. Inside the matrix is [kernal×kernal, C_{out}]×(number of times to repeat).

Layer Name	MobileNet-V2
Conv1	3×3, 32, stride 1, padding 1
Conv2	$\begin{bmatrix} 1\times1, 32 \\ 3\times3, 32 \\ 1\times1, 16 \end{bmatrix}\times1$
Conv3	$\begin{bmatrix} 1\times1, 96 \\ 3\times3, 96 \\ 1\times1, 24 \end{bmatrix}\times1$
Conv4	$\begin{bmatrix} 1\times1, 144 \\ 3\times3, 144 \\ 1\times1, 24 \end{bmatrix}\times1$
Conv5	$\begin{bmatrix} 1\times1, 144 \\ 3\times3, 144 \\ 1\times1, 32 \end{bmatrix}\times1$
Conv6	$\begin{bmatrix} 1\times1, 192 \\ 3\times3, 192 \\ 1\times1, 32 \end{bmatrix}\times2$
Conv7	$\begin{bmatrix} 1\times1, 192 \\ 3\times3, 192 \\ 1\times1, 64 \end{bmatrix}\times1$
Conv8	$\begin{bmatrix} 1\times1, 384 \\ 3\times3, 384 \\ 1\times1, 64 \end{bmatrix}\times3$
Conv9	$\begin{bmatrix} 1\times1, 384 \\ 3\times3, 384 \\ 1\times1, 96 \end{bmatrix}\times1$
Conv10	$\begin{bmatrix} 1\times1, 576 \\ 3\times3, 576 \\ 1\times1, 96 \end{bmatrix}\times2$
Conv11	$\begin{bmatrix} 1\times1, 576 \\ 3\times3, 576 \\ 1\times1, 160 \end{bmatrix}\times1$
Conv12	$\begin{bmatrix} 1\times1, 960 \\ 3\times3, 960 \\ 1\times1, 160 \end{bmatrix}\times2$
Conv13	$\begin{bmatrix} 1\times1, 960 \\ 3\times3, 960 \\ 1\times1, 320 \end{bmatrix}\times1$
FC1	320×1280
FC2	1280×10

CIFAR-100, we test on ResNet-32. The hyperparameter choices for each network, dataset and their corresponding sparsities are listed in Tables (9, 10, 11, 12, 13, 14, 15, 16)

A.3.2 Finetuning the Rare Gems

The details of the hyperparameter we used in finetuning the rare gems we find is shown in Table 17.

Table 9: Hyper Parameters used for different sparsities for GEM-MINER on ResNet-20 on CIFAR-10.

Network/Dataset	Sparsity	Pruning Period	Optimizer	LR	Lambda
ResNet-20 CIFAR-10	50%	8	SGD	0.05	10^{-8}
	86.26%	5	SGD	0.1	10^{-5}
	96.27%	5	SGD	0.1	3×10^{-5}
	98.56%	5	SGD	0.1	10^{-4}
	99.41%	5	SGD	0.1	10^{-4}

Table 10: Hyper Parameters used for different sparsities for GEM-MINER on MobileNet-V2 on CIFAR-10.

Network/Dataset	Sparsity	Pruning Period	Optimizer	LR	Lambda
MobileNet-V2 CIFAR-10	50%	5	SGD	0.05	0
	80%	5	SGD	0.1	3×10^{-6}
	95%	5	SGD	0.1	7×10^{-6}
	98.56%	5	SGD	0.1	10^{-4}

Table 11: Hyper Parameters used for different sparsities for GEM-MINER on VGG-16 on CIFAR-10.

Network/Dataset	Sparsity	Pruning Period	Optimizer	LR	Lambda
VGG-16 CIFAR-10	50%	5	SGD	0.01	0
	95%	5	SGD	0.01	10^{-6}
	97.5%	5	SGD	0.01	10^{-6}
	98.6%	5	SGD	0.01	10^{-6}
	99.5%	5	SGD	0.01	10^{-6}

Table 12: Hyper Parameters used for different sparsities for GEM-MINER on WideResNet-28-2 on CIFAR-10.

Network/Dataset	Sparsity	Pruning Period	Optimizer	LR	Lambda
WideResNet-28-2 CIFAR-10	50%	5	SGD	0.1	0
	80%	10	SGD	0.1	10^{-5}
	95%	10	SGD	0.1	10^{-5}
	98.56%	10	SGD	0.1	10^{-5}
	99.5%	10	SGD	0.1	10^{-5}

Table 13: Hyper Parameters used for different sparsities for GEM-MINER on ResNet-18 on TinyImageNet.

Network/Dataset	Sparsity	Pruning Period	Optimizer	LR	Lambda
ResNet-18 TinyImageNet	50%	10	SGD	0.1	0
	95%	5	SGD	0.001	8×10^{-6}
	98.6%	5	SGD	0.001	5×10^{-6}
	99.5%	5	SGD	0.001	10^{-5}

Table 14: Hyper Parameters used for different sparsities for GEM-MINER on ResNet-50 on TinyImageNet.

Network/Dataset	Sparsity	Pruning Period	Optimizer	LR	Lambda
ResNet-50 TinyImageNet	50%	5	ADAM	0.01	0
	95%	5	ADAM	0.01	10^{-6}
	98.6%	5	ADAM	0.01	10^{-6}
	99.5%	5	ADAM	0.01	10^{-6}

Table 15: Hyper Parameters used for different sparsities for GEM-MINER on ResNet-50 on Caltech101.

Network/Dataset	Sparsity	Pruning Period	Optimizer	LR	Lambda
ResNet-50 Caltech101	50%	5	ADAM	0.01	0
	95%	5	ADAM	0.01	10^{-6}
	98.6%	5	ADAM	0.01	10^{-6}
	99.5%	5	ADAM	0.01	10^{-6}

Table 16: Hyper Parameters used for different sparsities for GEM-MINER on ResNet-32 on CIFAR-100.

Network/Dataset	Sparsity	Pruning Period	Optimizer	LR	Lambda
ResNet-32 CIFAR-100	50%	5	SGD	0.1	10^{-7}
	80%	5	SGD	0.1	10^{-6}
	95%	5	SGD	0.1	10^{-5}
	98%	5	SGD	0.1	10^{-5}
	99.5%	5	SGD	0.1	5×10^{-5}

Table 17: Hyperparameters used for finetuning. We use the same number of epochs for GEM-MINER and finetuning.

Model	Dataset	Sparsity	Epochs	Batch Size	LR	LR-Schedule Milestones
ResNet-20	CIFAR-10	50%	150	128	0.1	[80, 120]
	CIFAR-10	others	150	128	0.01	[80, 120]
MobileNet-V2	CIFAR-10	all	300	128	0.1	[150, 250]
VGG-16	CIFAR-10	all	200	128	0.05	[100, 150]
WideResNet-28-2	CIFAR-10	50%	150	128	0.1	[80, 120]
	CIFAR-10	others	150	128	0.01	[80, 120]
ResNet-18	TinyImageNet	all	200	256	0.1	[100, 150]
ResNet-50	TinyImageNet	all	150	256	0.1	[80, 120]
ResNet-50 (Pretrained)	Caltech-101	all	50	16	0.0001	[20, 40]
ResNet-32	CIFAR-100	all	150	64	0.01	[80, 120]

B Additional Experiments

We repeated the comparison of GEM-MINER with the baselines on MobileNet-V2 on TinyImagenet as well as ResNet-18 on CIFAR-10. We show the results in Fig. 8. Similar to our earlier experiments, we have the following observations. Note that GEM-MINER outperforms IMP (with warmup) in the sparse regime. Also, as is expected, GEM-MINER has non-trivial accuracy before finetune, which is higher than both EP and significantly higher than IMP. GEM-MINER shows a significant deterioration in performance when subjected to the sanity checks suggested in Frankle et al. [11]. Therefore, GEM-MINER is considered to pass the sanity checks. Finally GEM-MINER (*at initialization*) nearly achieves the performance of **Renda et al.** (Learning rate rewinding) in the sparse regime. This is particularly impressive given that **Renda et al.** is a pruning-after-training method *i.e.*, it prunes and trains iteratively, never finding a subnetwork at or even near initiailization.

How far can we take random pruning? Recall that subnetworks identified by GEM-MINER outperform random subnetworks found by sampling based on smart ratio (SR) [41]. However, SR performs surprisingly well given that it is still random pruning. Therefore, we tried improving SR to see how far we could take random pruning. For simplicity, we call these different versions of SR starting with **v1** which is the original algorithm suggested by Su et al. [41]. Table 18 compares the post-finetune accuracy of subnetworks found by training randomly initialized subnetworks at initialization, each using "smart ratios" found by increasingly sophisticated means. Given L-layer network, each version of SR uses its own sparsity pattern $\boldsymbol{p} = [p_1, \cdots, p_L]$ to generate random subnetwork, where each layer randomly picks $0 < p_l \leq 1$ fraction of weights to use. The sparsity patterns are identified as follows:

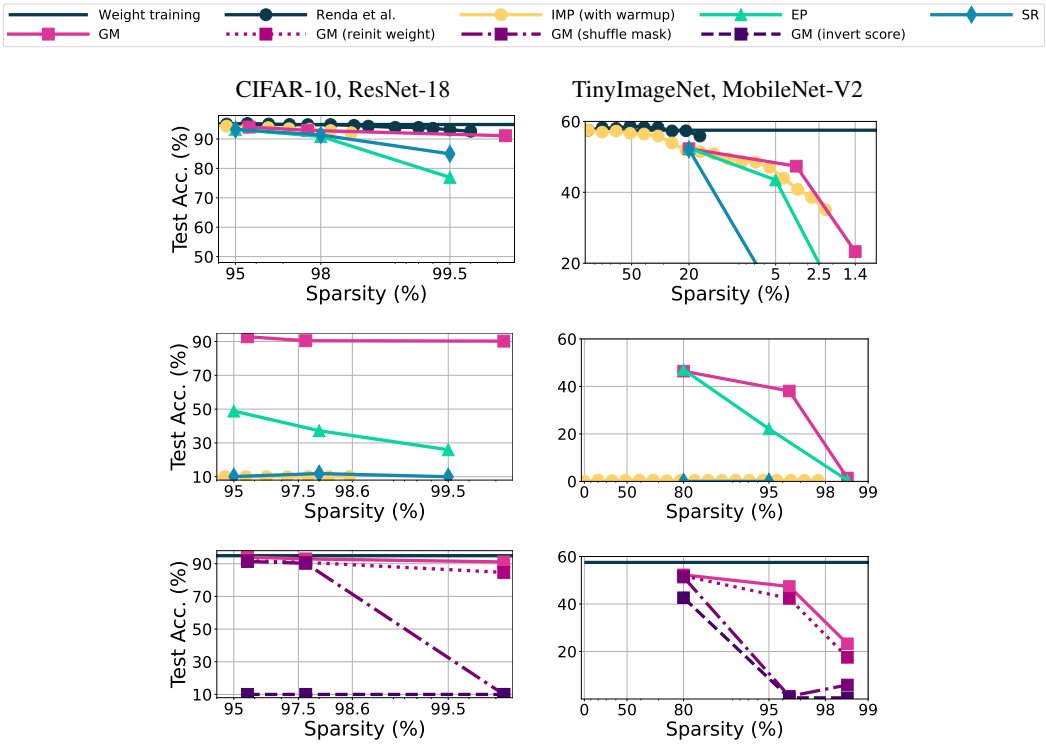

Figure 8: Additional experimental results on comparing different pruning algorithms. Top: post-finetune accuracy, Middle: pre-finetune accuracy, Bottom: sanity check methods suggested in Frankle et al. [11] applied on GEM-MINER. Similar to earlier experiments, we find that GEM-MINER outperforms IMP (with warmup) in the sparse regime in terms of both post finetune as well as pre-finetune accuracy. We also note that GEM-MINER passes all of the sanity checks.

- *SR-v1*: vanilla SR suggested in [41]
- *SR-v2*: set $p_l = p_l^{\text{SR}}$ for $2 \leq l \leq L-1$ and $p_l = p_l^{\text{GM}}$ for $l \in \{1, L\}$
- *SR-v3*: set $p_l = p_l^{\text{SR}}$ for $2 \leq l \leq L-1$ and $p_l = 1$ for $l \in \{1, L\}$
- *SR-v4*: start with $\boldsymbol{p}^{\text{IMP}}$ and search sparsity patterns in a small ball around it.
- *SR-v5*: start with the sparsity pattern of v2, and tune p_l using Eq (1)
- *SR-v6*: start with the sparsity pattern of v4, and tune p_l using Eq (1)

The vanilla SR proposed by Su et al. [41] is denoted by *SR-v1*. Motivated by the fact that (i) GEM-MINER outperforms SR and (ii) GEM-MINER and SR primarily differ in the layerwise sparsity p_l for the first and the last layers (refer Figure 7), we construct *SR-v2*. It uses the layerwise sparsity of *SR-v1* for intermediate layers ($2 \leq l \leq L$) and layerwise sparsity of GEM-MINER for $l \in \{1, L\}$. As a simple variant of *SR-v2*, we also considered using full dense layer ($p_l = 1$) for the first and the last layer, which is denoted by *SR-v3*. This was because we observed that the first and last layers found by GEM-MINER were relatively dense compared to *SR-v1*. *SR-v4* searches a few points around the sparsity pattern of IMP and chooses the best option. We chose only a few options based on intuition to make the search computationally tractable.

Finally, we tried to formulate finding the optimal random pruning method by writing it as an optimization problem in p_l as follows:

$$\min_{p_l} \sum_{(\boldsymbol{x},y)\in S} \ell(f(\boldsymbol{w}_l \odot \text{Bern}(p_l); \boldsymbol{x}), y) \tag{1}$$

Intuitively, this is equivalent to choosing p_l such that the loss of the random subnetwork generated by sampling the mask of layer l with probability p_l is minimized.

Table 18: Performance comparison of GEM-MINER and variants of Smart Ratio (SR), for ResNet-20 trained for CIFAR-10 classification task, for target sparsity 98.56%. We denoted the vanilla SR in [41] as *SR-v1* and tested five additional variants, from *SR-v2* to *SR-v6*. The detailed description of the variants are given in Paragraph B.

Schemes	Gem-Miner	SR (v1) [41]	SR (v2)	SR (v3)	SR (v4)	SR (v5)	SR (v6)
Sparsity (%)	98.56	98.56	98.53	98.25	98.56	98.47	98.53
Accuracy (%)	**77.89**	65.61	68.59	69.78	69.92	69.01	69.08

In order to make the problem more tractable, we set the output of previous versions as the initial value of p_l. Choosing $p_l^{(0)}$ as the $p_l^{\text{SR-v2}}$ and then applying SGD on Eq 1 gives us *SR-v5*. Repeating this with $p_l^{(0)} = p_l^{\text{SR-v4}}$ results in *SR-v6*.

The results of Table 18 show that different strategies in random pruning can improve the performance of random pruning, but GEM-MINER still has an 8% accuracy gap with the best random network we found for ResNet-20, CIFAR-10 classification, when the target sparsity is 98.56%.

Relationship between pre-finetune accuracy and post-finetune accuracy. Recall that rare gems need to have not only high *post-finetune* accuracy but also non-trivial *pre-finetune* accuracy.

Since the latter is a lower bound on *post-finetune* accuracy, we design GEM-MINER to just maximize the accuracy at initialization. However, it is not clear that this actually maximizes post-finetune accuracy. In fact, the performance of EP and IMP clearly show that it is neither a necessary nor sufficient condition. Fig. 9 shows both pre and post finetune accuracies for GEM-MINER (GM), edge-popup (EP), and smart ratio (SR), at both 98.6% and 99.5% sparsity, for CIFAR-10 classification using VGG-16. For GEM-MINER, we show the results of using four different learning rates $\eta = 0.01, 0.001, 0.0001, 0.00001$ for 200 epochs using lighter colors to indicate lower learning rates. It turns out that $\eta = 0.01$ has the best pre and post-finetune accuracies for both sparsities. Moreover, it shows that there is some correlation between pre-finetune and post-finetune accuracy *i.e.*, subnetworks that have higher pre-finetune accuracies typically have higher accuracy after finetuning as well. However, comparing GEM-MINER's results for 99.5% sparsity with $\eta = 0.0001$ and $\eta = 0.00001$ shows that this pattern does not always hold. With $\eta = 0.00001$, GEM-MINER achieves a higher pre-finetune accuracy but ends up with lower accuracy after finetuning. Therefore, we conclude that while pre-finetune accuracy is a reasonable proxy for accuracy after finetuning, it does not guarantee it in any way. Note that both points for EP have "post-finetune accuracy" $\simeq$ "pre-finetune accuracy", which confirms the observation by Ramanujan et al. [33] that subnetworks found by EP are not lottery tickets *i.e.*, they are not conducive to further training.

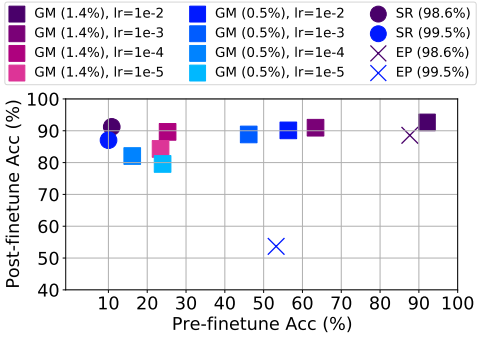

Figure 9: Relationship between pre-finetune accuracy and post-finetune accuracy, on CIFAR-10, VGG-16, for GEM-MINER (GM), edge-popup (EP), and smart ratio (SR). For both 98.6% and 99.5% sparsity, GEM-MINER typically shows higher post-finetune accuracy when it has higher pre-finetune acuracy. As noted by Ramanujan et al. [33], subnetworks found by EP are not finetunable.

Discussions on the need of warmup for IMP. For completeness, we also tried different variants of IMP. We refer to it as "cold" IMP when the weights are rewound to initialization, while "warm" IMP rewinds to some early iteration, *i.e.*, after training for a few epochs. Further, we classify it depending on the number of epochs per magnitude pruning. We say IMP is "short" if it only trains for a few epochs (*e.g.*, 8 epochs for ResNet-20 on CIFAR-10) before pruning, and "long" if it takes a considerably larger number of epochs before pruning (*e.g.*, 160 epochs for ResNet-20 on CIFAR-10). Regardless of "long" or "short", the number of epochs to finetune the pruned model are the same.

With these informal definitions, we can categorize IMP into four different versions: a) *short-cold* IMP, b) *short-warm* IMP, c) *long-cold* IMP, and d) *long-warm* IMP.

In the literature, only *long* variants have been studied thoroughly. Frankle et al. [10] noted that for large networks and difficult tasks, long-cold IMP fails to find lottery tickets, which is why they introduce long-warm IMP.

Somewhat surprisingly, as shown in Fig. 10, we find that short versions of IMP can achieve the same or an even better accuracy-sparsity tradeoff especially in the sparse regime. In particular, *short-cold* IMP matches the performance of *long-warm* one without any warmup, *i.e.*, *short-cold* IMP can find lottery-tickets at initialization. However, note that *short-cold* IMP only finds lottery tickets, **not** rare gems in that the subnetworks it finds have accuracy close to that of random guessing.

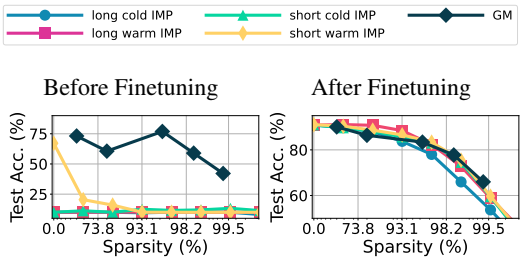

Figure 10: Test accuracy before (*left*) and after (*right*) finetuning for ResNet-20 on CIFAR-10. The "short" version of IMP achieves the same accuracy-sparsity tradeoff as *long-warm* IMP in the sparse regime. However, comparing the before finetuning accuracy (*left*) shows that GEM-MINER is capable of finding rare gems at initialization, whereas *short-cold* IMP can only find lottery tickets.