What's Hidden in a Randomly Weighted Neural Network?

Vivek Ramanujan [*][†]   Mitchell Wortsman [*][‡]   Aniruddha Kembhavi [†][‡]

Ali Farhadi [‡]   Mohammad Rastegari [‡]

Abstract

Training a neural network is synonymous with learning the values of the weights. In contrast, we demonstrate that randomly weighted neural networks contain subnetworks which achieve impressive performance without ever modifying the weight values. Hidden in a randomly weighted Wide ResNet-50 [32] we find a subnetwork (with random weights) that is smaller than, but matches the performance of a ResNet-34 [9] trained on ImageNet [4]. Not only do these "untrained subnetworks" exist, but we provide an algorithm to effectively find them. We empirically show that as randomly weighted neural networks with fixed weights grow wider and deeper, an "untrained subnetwork" approaches a network with learned weights in accuracy. Our code and pretrained models are available at: https://github.com/allenai/hidden-networks.

1. Introduction

What lies hidden in an overparameterized neural network with random weights? If the distribution is properly scaled, then it contains a subnetwork which performs well without ever modifying the values of the weights (as illustrated by Figure 1).

The number of subnetworks is combinatorial in the size of the network, and modern neural networks contain millions or even billions of parameters [24]. We should expect that even a randomly weighted neural network contains a subnetwork that performs well on a given task. In this work, we provide an algorithm to find these subnetworks.

Finding subnetworks contrasts with the prevailing paradigm for neural network training – learning the values of the weights by stochastic gradient descent. Traditionally, the network structure is either fixed during training (*e.g.* ResNet [9] or MobileNet [10]), or optimized in conjunction with the weight values (*e.g.* Neural Architecture Search (NAS)). We instead optimize to find a good subnet-
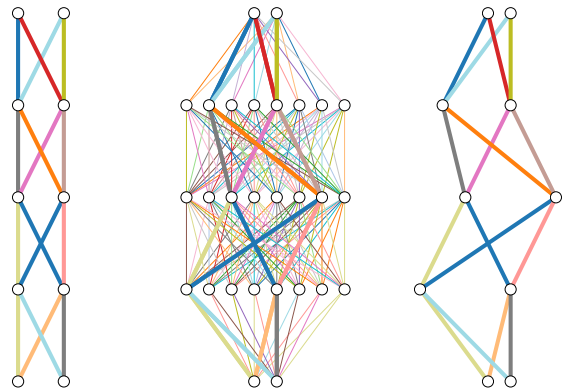
A neural network τ which achieves good performance — Randomly initialized neural network N — A subnetwork τ' of N

Figure 1. If a neural network with random weights (center) is sufficiently overparameterized, it will contain a subnetwork (right) that perform as well as a trained neural network (left) with the same number of parameters.

work within a fixed, randomly weighted network. We do not ever tune the value of any weights in the network, not even the batch norm [11] parameters or first or last layer.

In [5], Frankle and Carbin articulate *The Lottery Ticket Hypothesis*: neural networks contain sparse subnetworks that can be effectively trained from scratch when reset to their initialization. We offer a complimentary conjecture: within a sufficiently overparameterized neural network with random weights (*e.g.* at initialization), there exists a subnetwork that achieves competitive accuracy. Specifically, the test accuracy of the subnetwork is able to match the accuracy of a trained network with the same number of parameters.

This work is catalyzed by the recent advances of Zhou *et al.* [33]. By sampling subnetworks in the forward pass, they first demonstrate that subnetworks of randomly weighted neural networks can achieve impressive accuracy. However, we hypothesize that stochasticity may limit their performance. As the number of parameters in the network grows, they are likely to have a high variability in their sampled

networks.

To this end we propose the `edge-popup` algorithm for finding effective subnetworks within randomly weighted neural networks. We show a significant boost in performance and scale to ImageNet. For each fixed random weight in the network, we consider a positive real-valued score. To choose a subnetwork we take the weights with the top-$k\%$ highest scores. With a gradient estimator we optimize the scores via SGD. We are therefore able to find a good neural network without ever changing the values of the weights. We empirically demonstrate the efficacy of our algorithm and show that (under certain technical assumptions) the loss decreases on the mini-batch with each modification of the subnetwork.

We experiment on small and large scale datasets for image recognition, namely CIFAR-10 [13] and Imagenet [4]. On CIFAR-10 we empirically demonstrate that as networks grow wider and deeper, untrained subnetworks perform just as well as the dense network with learned weights. On ImageNet, we find a subnetwork of a randomly weighted Wide ResNet50 which is smaller than, but matches the performance of a trained ResNet-34. Moreover, a randomly weighted ResNet-101 [9] with fixed weights contains a subnetwork that is much smaller, but surpasses the performance of VGG-16 [27]. In short, we validate the unreasonable effectiveness of randomly weighted neural networks for image recognition.

2. Related Work

Lottery Tickets and Supermasks

In [5], Frankle and Carbin offer an intriguing hypothesis: neural networks contain sparse subnetworks that can be effectively trained from scratch when reset to their initialization. These so-called winning tickets have won the "initialization lottery". Frankle and Carbin find winning tickets by iteratively shrinking the size of the network, masking out weights which have the lowest magnitude at the end of each training run.

Follow up work by Zhou *et al.* [33] demonstrates that winning tickets achieve better than random performance without training. Motivated by this result they propose an algorithm to identify a "supermask" – a subnetwork of a randomly initialized neural network that achieves high accuracy without training. On CIFAR-10, they are able to find subnetworks of randomly initialized neural networks that achieve 65.4% accuracy.

The algorithm presented by Zhou *et al.* is as follows: for each weight w in the network they learn an associated probability p. On the forward pass they include weight w with probability p and otherwise zero it out. Equivalently, they use weight $\tilde{w} = wX$ where X is a Bernoulli(p) random variable (X is 1 with probability p and 0 otherwise). The probabilities p are the output of a sigmoid, and are learned

using stochastic gradient descent. The terminology super-mask" arises as finding a subnetwork is equivalent to learning a binary mask for the weights.

Our work builds upon Zhou *et al.*, though we recognize that the stochasticity of their algorithm may limit performance. In section 3.1 we provide more intuition for this claim. We show a significant boost in performance with an algorithm that does not sample supermasks on the forward pass. For the first time we are able to match the performance of a dense network with a supermask.

Neural Architecture Search (NAS)

The advent of modern neural networks has shifted the focus from feature engineering to feature learning. However, researchers may now find themselves manually engineering the architecture of the network. Methods of Neural Architecture Search (NAS) [34, 2, 19, 28] instead provide a mechanism for learning the architecture of neural network jointly with the weights. Models powered by NAS have recently obtained state of the art classification performance on ImageNet [29].

As highlighted by Xie *et al.* [31], the connectivity patterns in methods of NAS remain largely constrained. Surprisingly, Xie *et al.* establish that randomly wired neural networks can achieve competitive performance. Accordingly, Wortsman *et al.* [30] propose a method of Discovering Neural Wirings (DNW) – where the weights and structure are jointly optimized free from the typical constraints of NAS. We highlight DNW as we use a similar method of analysis and gradient estimator to optimize our supermasks. In DNW, however, the subnetwork is chosen by taking the weights with the highest magnitude. There is therefore no way to learn supermasks with DNW as the weights and connectivity are inextricably linked – there is no way to separate the weights and the structure.

Weight Agnostic Neural Networks

In Weight Agnostic Neural Networks (WANNs) [6], Gaier and Ha question if an architecture alone may encode the solution to a problem. They present a mechanism for building neural networks that achieve high performance when each weight in the network has the same shared value. Importantly, the performance of the network is agnostic to the value itself. They are able to obtain $\sim 92\%$ accuracy on MNIST [16].

We are quite inspired by WANNs, though we would like to highlight some important distinctions. Instead of each weight having the same value, we explore the setting where each weight has a random value. In Section A.2.2 of their appendix, Gaier and Ha mention that they were not successful in this setting. However, we find a good subnetwork for a given random initialization – the supermasks we find are not agnostic to the weights. Finally, Gaier and Ha construct their network architectures, while we look for supermasks within standard architectures.
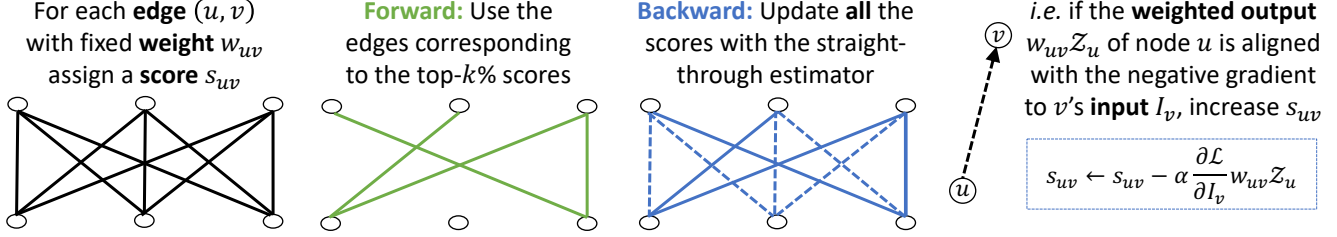
Figure 2. In the `edge-popup` Algorithm, we associate a score with each edge. On the forward pass we choose the top edges by score. On the backward pass we update the scores of all the edges with the straight-through estimator, allowing helpful edges that are "dead" to re-enter the subnetwork. We never update the value of any weight in the network, only the score associated with each weight.

Linear Classifiers and Pruning at Initialization

Linear classifiers on top of randomly weighted neural networks are often used as baselines in unsupervised learning [21, 3]. This work is different in motivation, we search for untrained subnetworks which achieve high performance without changing any weight values. This also differs from methods which prune at initialization and modify the weights of the discovered subnetwork [18, 17] or methods which modify a subset of the weights [25].

3. Method

In this section we present our optimization method for finding effective subnetworks within randomly weighted neural networks. We begin by building intuition in an unusual setting – the infinite width limit. Next we motivate and present our algorithm for finding effective subnetworks.

3.1. Intuition

The Existence of Good Subnetworks

Modern neural networks have a staggering number of possible subnetworks. Consequently, even at initialization, a neural network should contain a subnetwork which performs well.

To build intuition we will consider an extreme case – a neural network N in the infinite width limit (for a convolutional neural networks, the width of the network is the number of channels). As in Figure 1, let τ be a network with the same structure of N that achieves good accuracy. If the weights of N are initialized using any standard scaling of a normal distribution, *e.g.* xavier [7] or kaiming [8], then we may show there exists a subnetwork of N that achieves the same performance as τ without training. Let q be the probability that a given subnetwork of N has weights that are close enough to τ to obtain the same accuracy. This probability q is extremely small, but it is still nonzero. Therefore, the probability that no subnetwork of N is close enough to τ is effectively $(1 - q)^S$ where S is the number of subnetworks. S grows very quickly with the width of the network, and this probability becomes arbitrarily small.

How Should We Find A Good Subnetwork

Even if there are good subnetworks in randomly weighted neural networks, how should we find them?

Zhou *et al*. learn an associated probability p with each weight w in the network. On the forward pass they include weight w with probability p (where p is the output of a sigmoid) and otherwise zero it out. The infinite width limit provides intuition for a possible shortcoming of the algorithm presented by Zhou *et al*. [33]. Even if the parameters p are fixed, the algorithm will likely never observe the same subnetwork twice. As such, the gradient estimate becomes more unstable, and this in turn may make training difficult.

Our algorithm for finding a good subnetwork is illustrated by Figure 2. With each weight w in the neural network we learn a positive, real valued *popup* score s. The subnetwork is then chosen by selecting the weights in each layer corresponding to the top-$k\%$ highest scores. For simplicity we use the same value of k for all layers.

How should we update the score s_{uv}? Consider a single edge in a fully connected layer which connects neuron u to neuron v. Let w_{uv} be the weight of this edge, and s_{uv} the associated score. If this score is initially low then w_{uv} is not selected in the forward pass. But we would still like a way to update its score to allow it to pop back up. Informally, with backprop [26] we compute how the loss "wants" node v's input to change (*i.e.* the negative gradient). We then examine the weighted output of node u. If this weighted output is aligned with the negative gradient, then node u can take node v's output where the loss "wants" it to go. Accordingly, we should increase the score. If this alignment happens consistently, then the score will continue to increase and the edge will re-enter the chosen subnetwork (*i.e.* popup).

More formally, if $w_{uv}\mathcal{Z}_u$ denotes the weighted output of neuron u, and $\mathcal{I}_v$ denotes the input of neuron v, then we update s_{uv} as

$$s_{uv} \leftarrow s_{uv} - \alpha \frac{\partial \mathcal{L}}{\partial \mathcal{I}_v} \mathcal{Z}_u w_{uv}. \tag{1}$$

This argument and the analysis that follows is motivated and guided by the work of [30]. In their work, however, they do not consider a score and are instead directly updating the

weights. In the forward pass they use the top $k\%$ of edges by magnitude, and therefore there is no way of learning a subnetwork without learning the weights. Their goal is to train sparse neural networks, while we aim to showcase the efficacy of randomly weighted neural networks.

3.2. The `edge-popup` Algorithm and Analysis

We now formally detail the `edge-popup` algorithm.

For clarity, we first describe our algorithm for a fully connected neural network. In Section B.2 we provide the straightforward extension to convolutions along with code in PyTorch [23].

A fully connected neural network consists of layers $1, ..., L$ where layer ℓ has n_ℓ nodes $\mathcal{V}^{(\ell)} = \{v_1^{(\ell)}, ..., v_{n_\ell}^{(\ell)}\}$. We let $\mathcal{I}_v$ denote the input to node v and let $\mathcal{Z}_v$ denote the output, where $\mathcal{Z}_v = \sigma(\mathcal{I}_v)$ for some non-linear activation function σ (*e.g.* ReLU [14]). The input to neuron v in layer ℓ is a weighted sum of all neurons in the preceding layer. Accordingly, we write $\mathcal{I}_v$ as

$$\mathcal{I}_v = \sum_{u \in \mathcal{V}^{(\ell-1)}} w_{uv} \mathcal{Z}_u \qquad (2)$$

where w_{uv} are the network parameters for layer ℓ. The output of the network is taken from the final layer while the input data is given to the very first layer. Before training, the weights w_{uv} for layer ℓ are initialized by independently sampling from distribution $\mathcal{D}_\ell$. For example, if we are using kaiming normal initialization [8] with ReLU activations, then $\mathcal{D}_\ell = \mathcal{N}\left(0, \sqrt{2/n_{\ell-1}}\right)$ where $\mathcal{N}$ denotes the normal distribution.

Normally, the weights w_{uv} are optimized via stochastic gradient descent. In our `edge-popup` algorithm, we instead keep the weights at their random initialization, and optimize to find a subnetwork $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. We then compute the input of node v in layer ℓ as

$$\mathcal{I}_v = \sum_{(u,v) \in \mathcal{E}} w_{uv} \mathcal{Z}_u \qquad (3)$$

where $\mathcal{G}$ is a subgraph of the original fully connected network[1]. As mentioned above, for each weight w_{uv} in the original network we learn a *popup* score s_{uv}. We choose the subnetwork $\mathcal{G}$ by selecting the weights in each layer which have the top-$k\%$ highest scores. Equation 3 may therefore be written equivalently as

$$\mathcal{I}_v = \sum_{u \in \mathcal{V}^{(\ell-1)}} w_{uv} \mathcal{Z}_u h(s_{uv}) \qquad (4)$$

where $h(s_{uv}) = 1$ if s_{uv} is among the top $k\%$ highest scores in layer ℓ and $h(s_{uv}) = 0$ otherwise. Since the gradient of

---

[1]The original network has edges $\mathcal{E}_{\text{fc}} = \bigcup_{\ell=1}^{L-1} (\mathcal{V}_\ell \times \mathcal{V}_{\ell+1})$ where $\times$ denotes the cross-product.

h is 0 everywhere it is not possible to directly compute the gradient of the loss with respect to s_{uv}. We instead use the straight-through gradient estimator [1], in which h is treated as the identity in the backwards pass – the gradient goes "straight-through" h. Consequently, we approximate the gradient to s_{uv} as

$$\hat{g}_{s_{uv}} = \frac{\partial \mathcal{L}}{\partial \mathcal{I}_v} \frac{\partial \mathcal{I}_v}{\partial s_{uv}} = \frac{\partial \mathcal{L}}{\partial \mathcal{I}_v} w_{uv} \mathcal{Z}_u \qquad (5)$$

where $\mathcal{L}$ is the loss we are trying to minimize. The scores s_{uv} are then updated via stochastic gradient descent with learning rate α. If we ignore momentum and weight decay [15] then we update s_{uv} as

$$\tilde{s}_{uv} = s_{uv} - \alpha \frac{\partial \mathcal{L}}{\partial \mathcal{I}_v} w_{uv} \mathcal{Z}_u \qquad (6)$$

where $\tilde{s}_{uv}$ denotes the score after the gradient step[2].

As the scores change certain edges in the subnetwork will be replaced with others. Motivated by the analysis of [30] we show that when swapping does occur, the loss decreases for the mini-batch.

Theorem 1: When edge (i, ρ) replaces (j, ρ) and the rest of the subnetwork remains fixed then the loss decreases for the mini-batch (provided the loss is sufficiently smooth).

Proof. Let $\tilde{s}_{uv}$ denote the score of weight w_{uv} after the gradient update. If edge (i, ρ) replaces (j, ρ) then our algorithm dictates that $s_{i\rho} < s_{j\rho}$ but $\tilde{s}_{i\rho} > \tilde{s}_{j\rho}$. Accordingly,

$$\tilde{s}_{i\rho} - s_{i\rho} > \tilde{s}_{j\rho} - s_{j\rho} \qquad (7)$$

which implies that

$$-\alpha \frac{\partial \mathcal{L}}{\partial \mathcal{I}_\rho} w_{i\rho} \mathcal{Z}_i > -\alpha \frac{\partial \mathcal{L}}{\partial \mathcal{I}_\rho} w_{j\rho} \mathcal{Z}_j \qquad (8)$$

by the update rule given in Equation 6. Let $\tilde{\mathcal{I}}_\rho$ denote the input to node k after the swap is made and $\mathcal{I}_\rho$ denote the original input. Note that $\tilde{\mathcal{I}}_\rho - \mathcal{I}_\rho = w_{i\rho} \mathcal{Z}_i - w_{j\rho} \mathcal{Z}_j$ by Equation 3. We now wish to show that $\mathcal{L}(\tilde{\mathcal{I}}_\rho) < \mathcal{L}(\mathcal{I}_\rho)$.

If the loss is smooth and $\tilde{\mathcal{I}}_\rho$ is close to $\mathcal{I}_\rho$ and ignore second-order terms in a Taylor expansion:

$$\mathcal{L}\left(\tilde{\mathcal{I}}_\rho\right) = \mathcal{L}\left(\mathcal{I}_\rho + \left(\tilde{\mathcal{I}}_\rho - \mathcal{I}_\rho\right)\right) \qquad (9)$$

$$\approx \mathcal{L}\left(\mathcal{I}_\rho\right) + \frac{\partial \mathcal{L}}{\partial \mathcal{I}_\rho}\left(\tilde{\mathcal{I}}_\rho - \mathcal{I}_\rho\right) \qquad (10)$$

$$= \mathcal{L}\left(\mathcal{I}_\rho\right) + \frac{\partial \mathcal{L}}{\partial \mathcal{I}_\rho}(w_{i\rho} \mathcal{Z}_i - w_{j\rho} \mathcal{Z}_j) \qquad (11)$$

and from equation 8 we have that $\frac{\partial \mathcal{L}}{\partial \mathcal{I}_\rho}(w_{i\rho} \mathcal{Z}_i - w_{j\rho} \mathcal{Z}_j) < 0$ and so $\mathcal{L}(\tilde{\mathcal{I}}_\rho) < \mathcal{L}(\mathcal{I}_\rho)$ as needed. We examine a more general case of Theorem 1 in Section B.1 of the appendix.

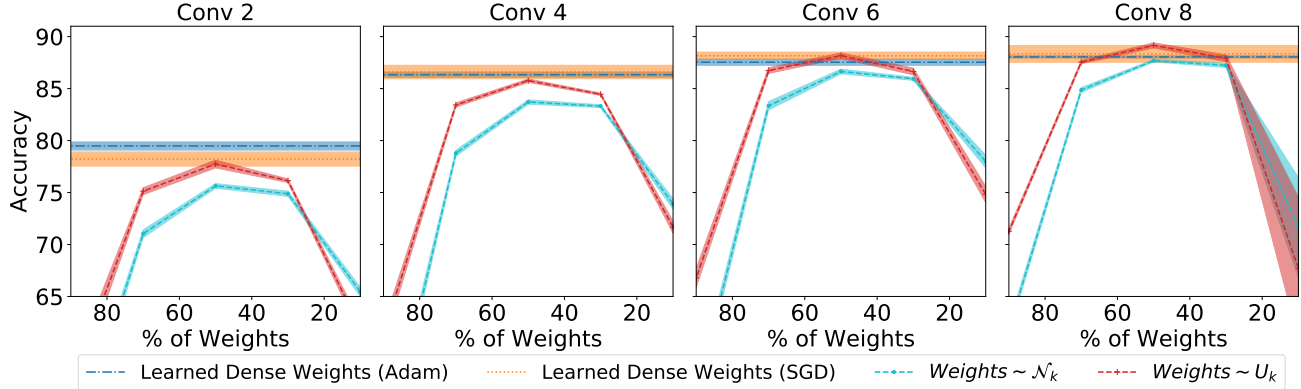[2]To ensure that the scores are positive we take the absolute value.

Figure 3. **Going Deeper:** Experimenting with shallow to deep neural networks on CIFAR-10 [13]. As the network becomes deeper, we are able to find subnetworks at initialization that perform as well as the dense original network when trained. The baselines are drawn as a horizontal line as we are not varying the % of weights. When we write *Weights ∼ D* we mean that the weights are randomly drawn from distribution $\mathcal{D}$ and are never tuned. Instead we find subnetworks with size (% of Weights)/100 * (Total # of Weights).

4. Experiments

We demonstrate the efficacy of randomly weighted neural networks for image recognition on standard benchmark datasets CIFAR-10 [13] and ImageNet [4]. This section is organized as follows: in Section 4.1 we discuss the experimental setup and hyperparameters. We perform a series of ablations at small scale: we examine the effect of k, the % of Weights which remain in the subnetwork, and the effect of width. In Section 4.4 we compare against the algorithm of Zhou *et al.*, followed by Section 4.5 in which we study the effect of the distribution used to sample the weights. We conclude with Section 4.6, where we optimize to find subnetworks of randomly weighted neural networks which achieve good performance on ImageNet.

4.1. Experimental Setup

We use two different distributions for the weights in our network:

- **Kaiming Normal** [8], which we denote $\mathcal{N}_k$. Following the notation in section 3.2 the Kaiming Normal distribution is defined as $\mathcal{N}_k = \mathcal{N}\left(0, \sqrt{2/n_{\ell-1}}\right)$ where $\mathcal{N}$ denotes the normal distribution.

- **Signed Kaiming Constant** which we denote U_k. Here we set each weight to be a constant and randomly choose its sign to be $+$ or $-$. The constant we choose is the standard deviation of Kaiming Normal, and as a result the variance is the same. We use the notation U_k as we are sampling uniformly from the set $\{-\sigma_k, \sigma_k\}$ where σ_k is the standard deviation for Kaiming Normal (*i.e.* $\sqrt{2/n_{\ell-1}}$).

In Section 4.5 we reflect on the importance of the random distribution and experiment with alternatives.

Model	Conv2	Conv4	Conv6	Conv8
Conv Layers	64, 64, pool	64, 64, pool 128, 128, pool	64, 64, pool 128, 128, pool 256, 256, pool	64, 64, pool 128, 128, pool 256, 256, pool 512, 512, pool
FC	256, 256, 10	256, 256, 10	256, 256, 10	256, 256, 10

Table 1. For completeness we provide the architecture of the simple VGG-like [27] architectures used for CIFAR-10 [13], which are identical to those used by Frankle and Carbin [5] and Zhou *et al.* [33]. However, the slightly deeper Conv8 does not appear in the previous work. Each model first performs convolutions followed by the fully connected (FC) layers, and pool denotes max-pooling.

On CIFAR-10 [13] we experiment with simple VGG-like architectures of varying depth. These architectures are also used by Frankle and Carbin [5] and Zhou *et al.* [33] and are provided in Table 1. On ImageNet we experiment with ResNet-50 and ResNet-101 [9], as well as their wide variants [32]. In every experiment (for all baselines, datasets, and our algorithm) we optimize for 100 epochs and report the last epoch accuracy on the validation set. When we optimize with Adam [12] we do not decay the learning rate. When we optimize with SGD we use cosine learning rate decay [20]. On CIFAR-10 [13] we train our models with weight decay 1e-4, momentum 0.9, batch size 128, and learning rate 0.1. We also often run both an Adam and SGD baseline where the weights are learned. The Adam baseline uses the same learning rate and batch size as in [5, 33][3]. For the SGD baseline we find that training does not converge with learning rate 0.1, and so we use 0.01. As standard we also use weight decay 1e-4, momentum 0.9, and batch size 128. For the ImageNet experiments we use the hyperparam-

---

[3] Batch size 60, learning rate 2e-4, 3e-4 and 3e-4 for Conv2, Conv4, and Conv6 respectively Conv8 is not tested in [5], though we use find that learning rate 3e-4 still performs well.
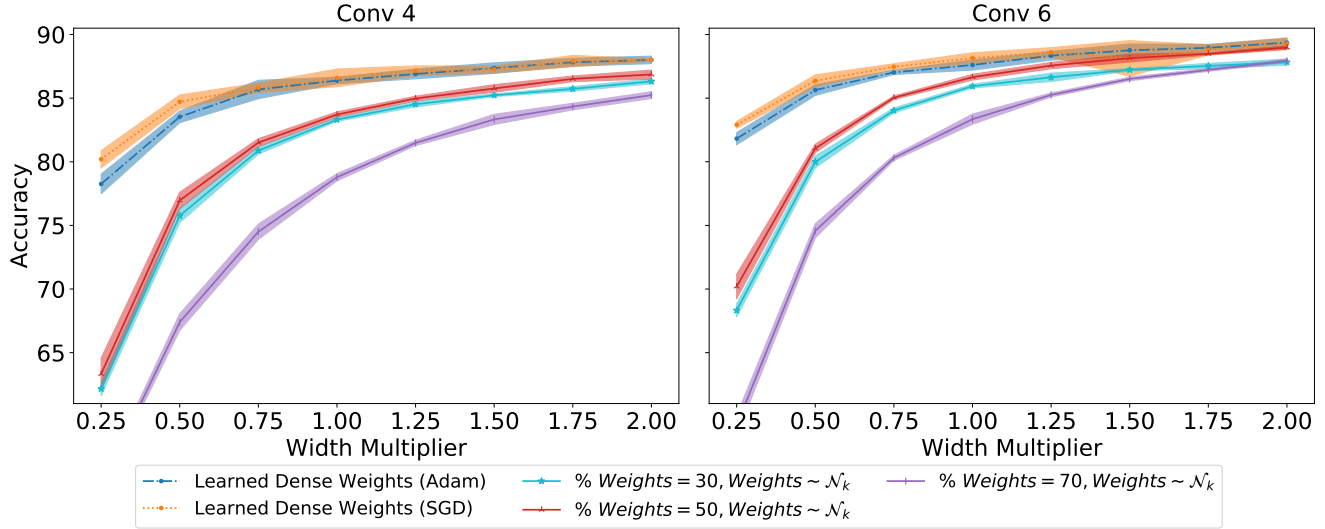
Figure 4. **Going Wider:** Varying the width (*i.e.* number of channels) of Conv4 and Conv6 for CIFAR-10 [13]. When Conv6 is wide enough, a subnetwork of the randomly weighted model (with %*Weights* = 50) performs just as well as the full model when it is trained.

eters found on NVIDIA's public github example repository for training ResNet [22]. For simplicity, our `edge-popup` algorithm does not modify batch norm parameters, they are frozen at their default initialization in PyTorch (*i.e.* bias 0, scale 1), and the scores are initialized Kaiming uniform [8].

This discussion has encompassed the extent of the hyperparameter tuning for our models. We do, however, perform hyperparameter tuning for the Zhou *et al*. [33] baseline and improve accuracy significantly. We include further discussion of this in Section 4.4.

In all experiments on CIFAR-10 [13] we use 5 different random seeds and plot the mean accuracy $\pm$ one standard deviation. Moreover, on all figures, *Learned Dense Weights* denotes the standard training the full model (all weights remaining).

4.2. Varying the % of Weights

Our algorithm has one associated parameter: the ==% of weights which remain in the subnetwork==, which we refer to as k. Figure 3 illustrates how the accuracy of the subnetwork we find varies with k, a trend which we will now dissect. We consider $k \in [10, 30, 50, 70, 90]$ and plot the dense model when it is trained as a horizontal line (as it has 100% of the weights).

We recieve the worst accuracy when k approaches 0 or 100. When k approaches 0, we are not able to perform well as our subnetwork has very few weights. On the other hand, when k approaches 100, our network outputs are random.

The best accuracy occurs when $k \in [30, 70]$, and we make a ==combinatorial argument for this trend==. We are choosing kn weights out of n, and there are $\binom{n}{kn}$ ways of doing so. ==The number of possible subnetworks is therefore maximized when $k \approx 0.5$, and at this value our search space==

==is at its largest.==

4.3. Varying the Width

Our intuition from Section 3.1 suggests that as the network gets wider, a subnetwork of a randomly weighted model should approach the trained model in accuracy. How wide is wide enough?
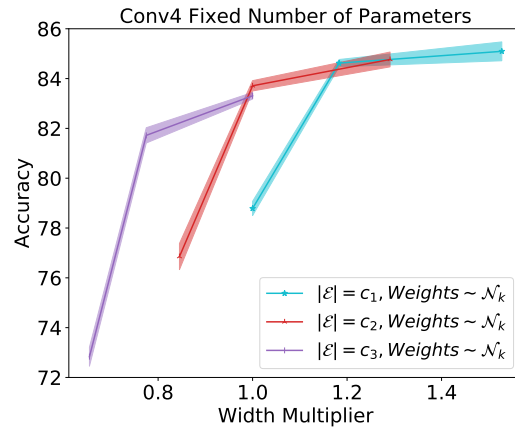

Figure 5. Varying the width of Conv4 on CIFAR-10 [13] while modifying k so that the # of Parameters is fixed along each curve. c_1, c_2, c_3 are constants which coincide with # of Parameters for $k = [30, 50, 70]$ for width multiplier 1.

In Figure 4 we vary the width of Conv4 and Conv6. The width of a linear layer is the number of "neurons", and the width of a convolution layer is the number of channels. The width multiplier is the factor by which the width of all layers is scaled. A width multiplier of 1 corresponds to the models tested in Figure 3.
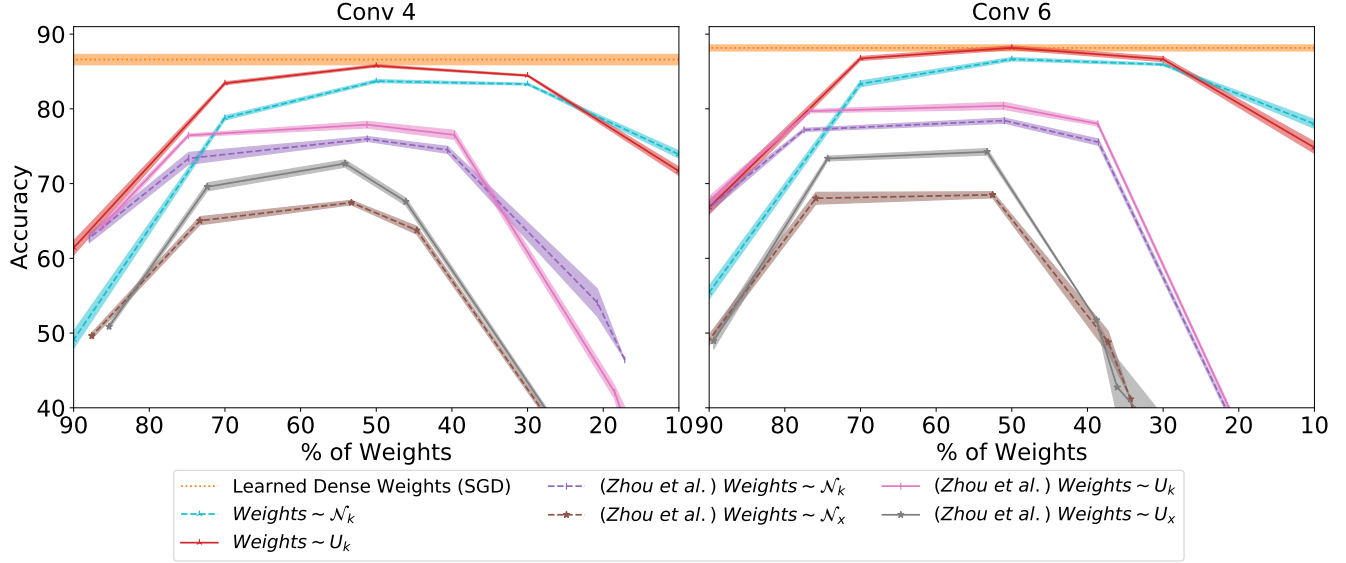
Figure 6. Comparing the performance of `edge-popup` with the algorithm presented by Zhou *et al.* [33] on CIFAR-10 [13].

As the width multiplier increases, the gap shrinks between the accuracy a subnetwork found with `edge-popup` and the dense model when it is trained. Notably, when Conv6 is wide enough, a subnetwork of the randomly weighted model (with $\%Weights = 50$) performs just as well as the dense model when it is trained.

Moreover, this boost in performance is not solely from the subnetwork having more parameters. Even when the # of parameters is fixed, increasing the width and therefore the search space leads to better performance. In Figure 5 we fix the number of parameters and while modifying k and the width multiplier. Specifically, we test $k \in [30, 50, 70]$ for subnetworks of constant size c_1, c_2 and c_3. On Figure 5 we use $|\mathcal{E}|$ denote the size of the subnetwork.

4.4. Comparing with Zhou *et al.* [33]

In Figure 6 we compare the performance of `edge-popup` with Zhou *et al.* Their work considers distributions $\mathcal{N}_x$ and U_x, which are identical to those presented in Section 4.1 but with xavier normal [7] instead of kaiming normal [8] – the factor of $\sqrt{2}$ is omitted from the standard deviation. By running their algorithm with $\mathcal{N}_k$ and U_k we witness a significant improvement. However, even the $\mathcal{N}_x$ and U_x results exceed those in the paper as we perform some hyperparameter tuning. As in our experiments on CIFAR-10, we use SGD with weight decay 1e-4, momentum 0.9, batch size 128, and a cosine scheduler [20]. We double the learning rate until we see the performance become worse, and settle on 200[4].

---

[4]A very high learning rate is required as mentioned in their work.
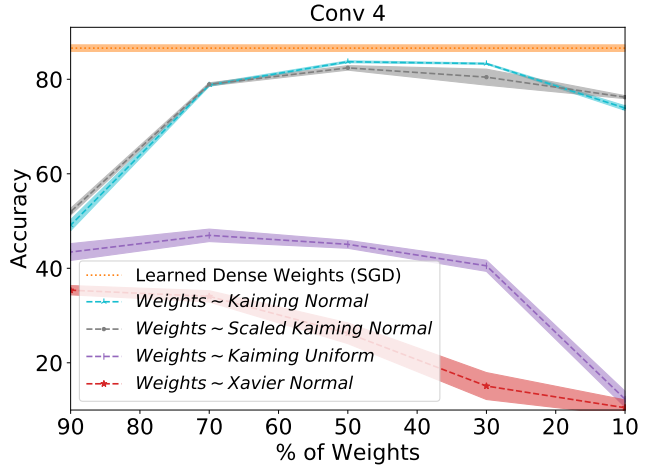

Figure 7. Testing different weight distributions on CIFAR-10 [13].

4.5. Effect of The Distribution

The distribution that the random weights are sampled from is very important. As illustrated by Figure 7, the performance of our algorithm vastly decreases when we switch to using xavier normal [7] or kaiming uniform [8].

Following the derivation in [8], the variance of the forward pass is not exactly 1 when we consider a subnetwork with only $k\%$ of the weights. To reconcile for this we could scale standard deviation by $\sqrt{1/k}$. This distribution is referred to as "Scaled Kaiming Normal" on Figure 7. We may also consider this scaling for the Signed Kaiming Constant distribution which is described in Section 4.1.
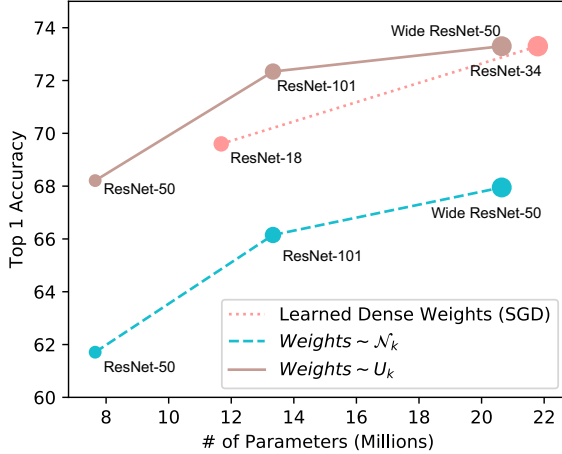
Figure 8. Testing our Algorithm on ImageNet [4]. We use a fixed $k = 30\%$, and find subnetworks within a randomly weighted ResNet-50 [9], Wide ResNet-50 [32], and ResNet-101. Notably, a randomly weighted Wide ResNet-50 contains a subnetwork which is smaller than, but matches the performance of ResNet-34. Note that for the non-dense models, # of Parameters denotes the size of the subnetwork.

4.6. ImageNet [4] Experiments

On ImageNet we observe similar trends to CIFAR-10. As ImageNet is a much harder dataset, ==computationally feasible models are not overparameterized to the same degree==. As a consequence, t==he performance of a randomly weighted subnetwork does not match the full model with learned weights==. However, we still witness a very encouraging trend – the performance increases with the width and depth of the network.

As illustrated by Figure 8, a randomly weighted Wide ResNet-50 contains a subnetwork that is smaller than, but matches the accuracy of ResNet-34 when trained on ImageNet [4]. As strongly suggested by our trends, better and larger "parent" networks would result in even stronger performance on ImageNet [4]. A table which reports the numbers in Figure 8 may be found in Section A of the appendix.

Figure 9 illustrates the effect of k, which follows an almost identical trend: $k \in [30, 70]$ performs best though 30 now provides the best performance. Figure 9 also demonstrates that we significantly outperform Zhou *et al.* at scale (in their original work they do not consider ImageNet). For Zhou *et al.* on ImageNet we report the best top-1 accuracy as we find their performance degrades towards the end of training. This is the only case where we do not report last epoch accuracy.

The choice of the random distribution matters more for ImageNet. The "Scaled" distribution we discuss in Section 4.5 did not show any discernable difference on CIFAR-10. However, Figure 10 illustrates that on ImageNet it is much
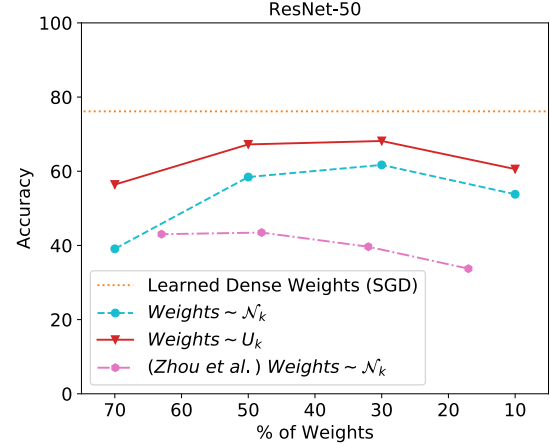

Figure 9. Examining the effect of $\%$ *weights* on ImageNet for `edge-popup` and the method of Zhou *et al.*.
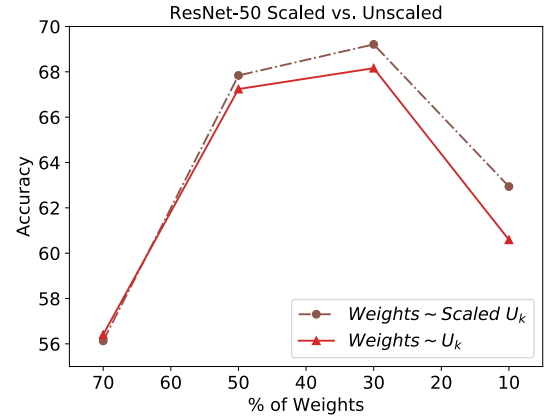

Figure 10. Examining the effect of using the "Scaled" initialization detailed in Section 4.5 on ImageNet.

better. Recall that the "Scaled" distribution adds a factor of $\sqrt{1/k}$, which has less of an effect when k approaches $100\% = 1$. ==This result highlights the possibility of finding better distributions which work better for this task==.

5. Conclusion

Hidden within randomly weighted neural networks we find subnetworks with compelling accuracy. This work provides an avenue for many areas of exploration. Finally, we hope that our findings serve as a useful step in the pursuit of understanding the optimization and initialization of neural networks.

References

[1] Yoshua Bengio, Nicholas Lonard, and Aaron Courville. Estimating or propagating gradients through stochastic neurons for conditional computation, 2013. 4

[2] Han Cai, Ligeng Zhu, and Song Han. ProxylessNAS: Direct neural architecture search on target task and hardware. In *ICLR*, 2019. 2

[3] Erik Cambria, Guang-Bin Huang, Liyanaarachchi Lekamalage Chamara Kasun, Hongming Zhou, Chi Man Vong, Jiarun Lin, Jianping Yin, Zhiping Cai, Qiang Liu, Kuan Li, et al. Extreme learning machines [trends & controversies]. *IEEE intelligent systems*, 28(6):30–59, 2013. 3

[4] 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, 5, 8, 11, 12, 13

[5] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *ICLR 2019*, 2019. 1, 2, 5, 12, 13

[6] Adam Gaier and David Ha. Weight agnostic neural networks. 2019. https://weightagnostic.github.io. 2

[7] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In Yee Whye Teh and Mike Titterington, editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256, Chia Laguna Resort, Sardinia, Italy, 13–15 May 2010. PMLR. 3, 7

[8] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the 2015 IEEE International Conference on Computer Vision (ICCV)*, ICCV '15, pages 1026–1034, Washington, DC, USA, 2015. IEEE Computer Society. 3, 4, 5, 6, 7

[9] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, 2016. 1, 2, 5, 8, 12, 13

[10] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *ArXiv*, abs/1704.04861, 2017. 1

[11] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *ICML*, 2015. 1, 11

[12] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014. 5

[13] Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, University of Toronto, 2009. 2, 5, 6, 7

[14] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012. 4

[15] Anders Krogh and John A. Hertz. A simple weight decay can improve generalization. In *NIPS*, 1991. 4

[16] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010. 2

[17] Namhoon Lee, Thalaiyasingam Ajanthan, Stephen Gould, and Philip HS Torr. A signal propagation perspective for pruning neural networks at initialization. *arXiv preprint arXiv:1906.06307*, 2019. 3

[18] Namhoon Lee, Thalaiyasingam Ajanthan, and Philip HS Torr. Snip: Single-shot network pruning based on connection sensitivity. *arXiv preprint arXiv:1810.02340*, 2018. 3

[19] Chenxi Liu, Barret Zoph, Maxim Neumann, Jonathon Shlens, Wei Hua, Li-Jia Li, Li Fei-Fei, Alan Yuille, Jonathan Huang, and Kevin Murphy. Progressive neural architecture search. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 19–34, 2018. 2

[20] Ilya Loshchilov and Frank Hutter. Sgdr: Stochastic gradient descent with warm restarts, 2016. 5, 7

[21] Mehdi Noroozi and Paolo Favaro. Unsupervised learning of visual representations by solving jigsaw puzzles, 2016. 3

[22] NVIDIA. Deeplearningexamples/pytorch/classification. https://github.com/NVIDIA/DeepLearningExamples/tree/master/PyTorch/Classification/RN50v1.5, 2019. 6, 12

[23] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary Devito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017. 4, 11, 12

[24] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *arXiv e-prints*, 2019. 1

[25] Amir Rosenfeld and John K Tsotsos. Intriguing properties of randomly weighted networks: Generalizing while learning next to nothing. In *2019 16th Conference on Computer and Robot Vision (CRV)*, pages 9–16. IEEE, 2019. 3

[26] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986. 3

[27] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. 2015. 2, 5

[28] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, and Quoc V. Le. Mnasnet: Platform-aware neural architecture search for mobile. *CoRR*, abs/1807.11626, 2018. 2

[29] Mingxing Tan and Quoc V. Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *ICML*, 2019. 2

[30] Mitchell Wortsman, Ali Farhadi, and Mohammad Rastegari. Discovering neural wirings. *ArXiv*, abs/1906.00586, 2019. 2, 3, 4

[31] Saining Xie, Alexander Kirillov, Ross Girshick, and Kaiming He. Exploring randomly wired neural networks for image recognition. *arXiv preprint arXiv:1904.01569*, 2019. 2

[32] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks, 2016. 1, 5, 8, 12, 13

[33] Hattie Zhou, Janice Lan, Rosanne Liu, and Jason Yosinski. Deconstructing lottery tickets: Zeros, signs, and the supermask, 2019. 1, 2, 3, 5, 6, 7

[34] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016. 2

A. Table of ImageNet Results

In Table 2 we provide a table of the results for image classification with ImageNet [4]. These results correspond exactly to Figure 8.

B. Additional Technical Details

In this section we first prove a more general case of Theorem 1 then provide an extension of `edge-popup` for convolutions along with code in PyTorch [23], found in Algorithm 1.

B.1. A More General Case of Theorem 1

Theorem 1 (more general): When a nonzero number of edges are swapped in one layer and the rest of the network remains fixed then the loss decreases for the mini-batch (provided the loss is sufficiently smooth).

Proof. As before, we let $\tilde{s}_{uv}$ denote the score of weight w_{uv} after the gradient update. Additionally, let $\tilde{\mathcal{I}}_v$ denote the input to node v after the gradient update whereas $\mathcal{I}_v$ is the input to node v before the update. Finally, let $i_1, ..., i_n$ denote the n nodes in layer $\ell - 1$ and $j_1, ..., j_m$ denote the m notes in layer ℓ. Our goal is to show that

$$\mathcal{L}\left(\tilde{\mathcal{I}}_{j_1}, ..., \tilde{\mathcal{I}}_{j_m}\right) < \mathcal{L}\left(\mathcal{I}_{j_1}, ..., \mathcal{I}_{j_m}\right) \tag{12}$$

where the loss is written as a function of layer ℓ's input for brevity. If the loss is smooth and $\tilde{\mathcal{I}}_{j_k}$ is close to $\mathcal{I}_{j_k}$ we may ignore second-order terms in a Taylor expansion:

$$\mathcal{L}\left(\tilde{\mathcal{I}}_{j_1}, ..., \tilde{\mathcal{I}}_{j_m}\right) \tag{13}$$

$$= \mathcal{L}\left(\mathcal{I}_{j_1} + \left(\tilde{\mathcal{I}}_{j_1} - \mathcal{I}_{j_1}\right), ..., \mathcal{I}_{j_m} + \left(\tilde{\mathcal{I}}_{j_m} - \mathcal{I}_{j_m}\right)\right) \tag{14}$$

$$= \mathcal{L}\left(\mathcal{I}_{j_1}, ..., \mathcal{I}_{j_m}\right) + \sum_{k=1}^{m} \frac{\partial \mathcal{L}}{\partial \mathcal{I}_{j_k}}\left(\tilde{\mathcal{I}}_{j_k} - \mathcal{I}_{j_k}\right) \tag{15}$$

And so, in order to show Equation 12 it suffices to show that

$$\sum_{k=1}^{m} \frac{\partial \mathcal{L}}{\partial \mathcal{I}_{j_k}}\left(\tilde{\mathcal{I}}_{j_k} - \mathcal{I}_{j_k}\right) < 0. \tag{16}$$

It is helpful to rewrite the sum to be over edges. Specifically, we will consider the sets $\mathcal{E}_{\text{old}}$ and $\mathcal{E}_{\text{new}}$ where $\mathcal{E}_{\text{new}}$ contains all edges that entered the network after the gradient update and $\mathcal{E}_{\text{old}}$ consists of edges which were previously in the subnetwork, but have now exited. As the total number of edges is conserved we know that $|\mathcal{E}_{\text{new}}| = |\mathcal{E}_{\text{old}}|$, and by assumption $|\mathcal{E}_{\text{new}}| > 0$.

Using the definition of $\mathcal{I}_k$ and $\tilde{\mathcal{I}}_k$ from Equation 3 we may rewrite Equation 16 as

$$\sum_{(i_a, j_b) \in \mathcal{E}_{\text{new}}} \frac{\partial \mathcal{L}}{\partial \mathcal{I}_{j_b}} w_{i_a j_b} Z_{i_a} - \sum_{(i_c, j_d) \in \mathcal{E}_{\text{old}}} \frac{\partial \mathcal{L}}{\partial \mathcal{I}_{j_d}} w_{i_c j_d} Z_{i_c} < 0 \tag{17}$$

which, by Equation 6 and factoring out $1/\alpha$ becomes

$$\sum_{(i_a, j_b) \in \mathcal{E}_{\text{new}}} \left(s_{i_a j_b} - \tilde{s}_{i_a j_b}\right) - \sum_{(i_c, j_d) \in \mathcal{E}_{\text{old}}} \left(s_{i_c j_d} - \tilde{s}_{i_c j_d}\right) < 0. \tag{18}$$

We now show that

$$\left(s_{i_a j_b} - \tilde{s}_{i_a j_b}\right) - \left(s_{i_c j_d} - \tilde{s}_{i_c j_d}\right) < 0 \tag{19}$$

for any pair of edges $(i_a, j_b) \in \mathcal{E}_{\text{new}}$ and $(i_c, j_d) \in \mathcal{E}_{\text{old}}$. Since $|\mathcal{E}_{\text{new}}| = |\mathcal{E}_{\text{old}}| > 0$ we are then able to conclude that Equation 18 holds.

As (i_a, j_b) was not in the edge set before the gradient update, but (i_c, j_d) was, we can conclude

$$s_{i_a j_b} - s_{i_c j_d} < 0. \tag{20}$$

Likewise, since (i_a, j_b) is in the edge set after the gradient update, but (i_c, j_d) isn't, we can conclude

$$\tilde{s}_{i_c j_d} - \tilde{s}_{i_a j_b} < 0. \tag{21}$$

By adding Equation 21 and Equation 20 we find that Equation 19 is satisfied as needed.

B.2. Extension to Convolutional Neural Networks

In order to show that our method extends to convolutional layers we recall that convolutions may be written in a form that resembles Equation 2. Let κ be the kernel size which we assume is odd for simplicity, then for $w \in \{1, ..., W\}$ and $h \in \{1, ..., H\}$ we have

$$\mathcal{I}_v^{w,h} = \sum_{u \in \mathcal{V}^{(\ell-1)}} \sum_{\kappa_1=1}^{\kappa} \sum_{\kappa_2=1}^{\kappa} w_{uv}^{(\kappa_1, \kappa_2)} \mathcal{Z}_u^{\left(w+\kappa_1-\lceil \frac{\kappa}{2} \rceil, h+\kappa_2-\lceil \frac{\kappa}{2} \rceil\right)} \tag{22}$$

where instead of "neurons", we now have "channels". The input $\mathcal{I}_v$ and output $\mathcal{Z}_v$ are now two dimensional and so $\mathcal{Z}_v^{(w,h)}$ is a scalar. As before, $\mathcal{Z}_v = \sigma(\mathcal{I}_v)$ where σ is a nonlinear function. However, in the convolutional case σ is often batch norm [11] followed by ReLU (and then implicitly followed by zero padding).

Instead of simply having weights w_{uv} we now have weights $w_{uv}^{(\kappa_1, \kappa_2)}$ for $\kappa_1 \in \{1, ..., \kappa\}$, $\kappa_2 \in \{1, ..., \kappa\}$. Likewise, in our `edge-popup` Algorithm we now consider scores $s_{uv}^{(\kappa_1, \kappa_2)}$ and again use the top $k\%$ in the forwards pass. As before, let $h\left(s_{uv}^{(\kappa_1, \kappa_2)}\right) = 1$ if $s_{uv}^{(\kappa_1, \kappa_2)}$ is among the top $k\%$ highest scores in the layer and $h\left(s_{uv}^{(\kappa_1, \kappa_2)}\right) = 0$ otherwise. Then in `edge-popup` we are performing a convolution as

$$\mathcal{I}_v^{w,h} = \sum_{u \in \mathcal{V}^{(\ell-1)}} \sum_{\kappa_1=1}^{\kappa} \sum_{\kappa_2=1}^{\kappa} w_{uv}^{(\kappa_1, \kappa_2)} \mathcal{Z}_u^{\left(w+\kappa_1-\lceil \frac{\kappa}{2} \rceil, h+\kappa_2-\lceil \frac{\kappa}{2} \rceil\right)} h\left(s_{uv}^{(\kappa_1, \kappa_2)}\right) \tag{23}$$

which mirrors the formulation of `edge-popup` in Equation 4. In fact, when $\kappa = W = H = 1$ (*i.e.* a 1x1 convolution on a 1x1 feature map) then Equation 23 and Equation 4 are equivalent.

The update for the scores is quite similar, though we must now sum over all spatial (*i.e.* w and h) locations as given below:

$$s_{uv}^{(\kappa_1, \kappa_2)} \leftarrow s_{uv}^{(\kappa_1, \kappa_2)}$$

$$- \alpha \sum_{w=1}^{W} \sum_{h=1}^{H} \frac{\partial \mathcal{L}}{\partial \mathcal{I}_v^{w,h}} w_{uv}^{(\kappa_1, \kappa_2)} \mathcal{Z}_u^{\left(w+\kappa_1-\lceil \frac{\kappa}{2} \rceil, h+\kappa_2-\lceil \frac{\kappa}{2} \rceil\right)} \tag{24}$$

Method	Model	Initialization	% of Weights	# of Parameters	Accuracy
Learned Dense Weights (SGD)	ResNet-34 [9]	-	-	21.8M	73.3%
	ResNet-50 [9]	-	-	25M	76.1%
	Wide ResNet-50 [32]	-	-	69M	78.1%
`edge-popup`	ResNet-50	Kaiming Normal	30%	7.6M	61.71%
	ResNet-101	Kaiming Normal	30%	13M	66.15%
	Wide ResNet-50	Kaiming Normal	30%	20.6M	67.95%
`edge-popup`	ResNet-50	Signed Kaiming Constant	30%	7.6M	68.6%
	ResNet-101	Signed Kaiming Constant	30%	13M	72.3%
	Wide ResNet-50	Signed Kaiming Constant	30%	20.6M	73.3%

Table 2. ImageNet [4] classification results corresponding to Figure 8. Note that for the non-dense models, # of Parameters denotes the size of the subnetwork.

Algorithm 1 PyTorch code for an `edge-popup` Conv.

```python
class GetSubnet(autograd.Function):
    @staticmethod
    def forward(ctx, scores, k):
        # Get the subnetwork by sorting the scores and
            using the top k%
        out = scores.clone()
        _, idx = scores.flatten().sort()
        j = int((1-k) * scores.numel())

        # flat_out and out access the same memory.
        flat_out = out.flatten()
        flat_out[idx[:j]] = 0
        flat_out[idx[j:]] = 1

        return out

    @staticmethod
    def backward(ctx, g):
        # send the gradient g straight-through on the
            backward pass.
        return g, None


class SubnetConv(nn.Conv2d):
    # self.k is the % of weights remaining, a real
        number in [0,1]
    # self.popup_scores is a Parameter which has the
        same shape as self.weight
    # Gradients to self.weight, self.bias have been
        turned off.
    def forward(self, x):
        # Get the subnetwork by sorting the scores.
        adj = GetSubnet.apply(
                self.popup_scores.abs(), self.k)
        # Use only the subnetwork in the forward pass.
        w = self.weight * adj
        x = F.conv2d(
            x, w, self.bias, self.stride, self.padding,
                self.dilation, self.groups
        )
        return x
```


Figure 11. Repeating the experiments from Figure 3 with ResNet18 on CIFAR.

C. Additional Experiments

C.1. Resnet18 on CIFAR-10

In figure 11 we experiment with a more advanced network architecture on CIFAR-10.

C.2. Are these subnetworks lottery tickets?

What happens when we train the weights of the subnetworks form Figure 8 and Table 2 on ImageNet? They do not train to the same accuracy as a dense network, and do not perform substantially better than training a random subnetwork. This suggests that the good performance of these subnetworks at initialization does not explain the lottery phenomena described in [5]. The results can be found in Table 3, where we again use the hyperparameters found on NVIDIA's public github example repository for training ResNet [22].

In summary, we now have κ^2 edges between each u and v. The PyTorch [23] code is given by Algorithm 1, where h is GetSubnet. The gradient goes straight through h in the backward pass, and PyTorch handles the implementation of these equations.
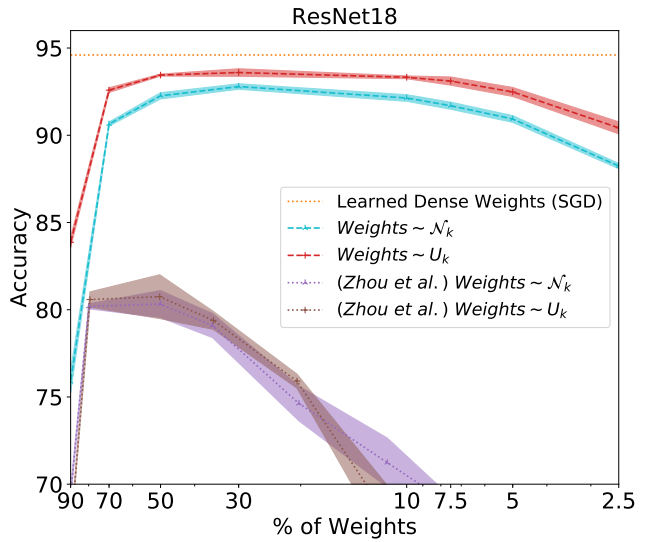
Subnetwork Type	Model	Initialization	% of Weights	# of Parameters	Top 1 Accuracy	Top 5 Accuracy
Learned	ResNet-50 [9]	Kaiming Normal	30%	7.6M	73.6%	91.6%
	ResNet-50 [9]	Signed Constant	30%	7.6M	73.7%	91.5%
	Wide ResNet-50 [32]	Kaiming Normal	30%	20.6M	76.8%	93.2%
	Wide ResNet-50 [32]	Signed Constant	30%	20.6M	76.9%	93.3%
Random	ResNet-50 [9]	Kaiming Normal	30%	7.6M	73.5%	91.5%
	Wide ResNet-50 [32]	Kaiming Normal	30%	20.6M	76.5%	93.1%
	ResNet-101 [9]	Kaiming Normal	30%	13M	76.1%	93.0%

Table 3. ImageNet [4] classification results after training the discovered subnetworks. Surprisingly, the accuracy is not substantially better than training a random subnetwork. This suggests that the good performance of these subnetworks does not explain the lottery phenomena described in [5].