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:

$$\begin{aligned}
&\mathcal{L}\left(\tilde{\mathcal{I}}_{j_1}, ..., \tilde{\mathcal{I}}_{j_m}\right) && (13)\\
&= \mathcal{L}\left(\mathcal{I}_{j_1} + \left(\tilde{\mathcal{I}}_{j_1} - \mathcal{I}_{j_1}\right), ..., \mathcal{I}_{j_m} + \left(\tilde{\mathcal{I}}_{j_m} - \mathcal{I}_{j_m}\right)\right) && (14)\\
&= \mathcal{L}\left(\mathcal{I}_{j_1}, ..., \mathcal{I}_{j_m}\right) + \sum_{k=1}^{m} \frac{\partial \mathcal{L}}{\partial \mathcal{I}_{j_k}} \left(\tilde{\mathcal{I}}_{j_k} - \mathcal{I}_{j_k}\right) && (15)
\end{aligned}$$

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

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

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, $\mathcal{Z}_v = \sigma\left(\mathcal{I}_v\right)$ 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

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

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:

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

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.

```
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
```

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.

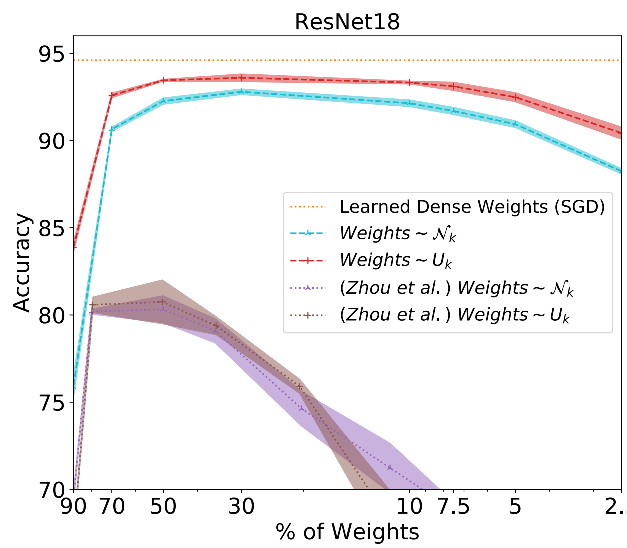

Figure 11. Repeating the experiments from Figure 3 with ResNet18 on CIFAR.

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].

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].