Discovering NeuralWirings

Mitchell Wortsman\textsuperscript{1,2}, Ali Farhadi\textsuperscript{1,2,3}, Mohammad Rastegari\textsuperscript{1,3}
\textsuperscript{1}PRIOR @ Allen Institute for AI, \textsuperscript{2}University of Washington, \textsuperscript{3}XNOR.AI
mitchnw@cs.washington.edu, \{ali, mohammad\}@xnor.ai

Abstract

The success of neural networks has driven a shift in focus from feature engineering to architecture engineering. However, successful networks today are constructed using a small and manually defined set of building blocks. Even in methods of neural architecture search (NAS) the network connectivity patterns are largely constrained. In this work we propose a method for discovering neural wirings. We relax the typical notion of layers and instead enable channels to form connections independent of each other. This allows for a much larger space of possible networks. The wiring of our network is not fixed during training – as we learn the network parameters we also learn the structure itself. Our experiments demonstrate that our learned connectivity outperforms hand engineered and randomly wired networks. By learning the connectivity of MobileNetV1 \cite{12} we boost the ImageNet accuracy by 10\% at \sim 41M FLOPs. Moreover, we show that our method generalizes to recurrent and continuous time networks. Our work may also be regarded as unifying core aspects of the neural architecture search problem with sparse neural network learning. As NAS becomes more fine grained, finding a good architecture is akin to finding a sparse subnetwork of the complete graph. Accordingly, DNW provides an effective mechanism for discovering sparse subnetworks of predefined architectures in a single training run. Though we only ever use a small percentage of the weights during the forward pass, we still play the so-called initialization lottery \cite{8} with a combinatorial number of subnetworks. Code and pretrained models are available at \url{https://github.com/allenai/dnw} while additional visualizations may be found at \url{https://mitchellnw.github.io/blog/2019/dnw/}.

1 Introduction

Deep neural networks have shifted the prevailing paradigm from feature engineering to feature learning. The architecture of deep neural networks, however, must still be hand designed in a process known as architecture engineering. A myriad of recent efforts attempt to automate the process of the architecture design by searching among a set of smaller well-known building blocks \cite{30,54,57,15,20}. While methods of search range from reinforcement learning to gradient based approaches \cite{14,20}, the space of possible connectivity patterns is still largely constrained. NAS methods explore wirings between predefined blocks, and \cite{28} learns the recurrent structure of CNNs. We believe that more efficient solutions may arrive from searching the space of wirings at a more fine grained level, i.e. single channels.

In this work, we consider an unconstrained set of possible wirings by allowing channels to form connections independent of each other. This enables us to discover a wide variety of operations (e.g. depthwise separable convs \cite{12}, channel shuffle and split \cite{56}, and more). Formally, we treat the network as a large neural graph where each each node processes a single channel.

One key challenge lies in searching the space of all possible wirings – the number of possible sub-graphs is combinatorial in nature. When considering thousands of nodes, traditional search methods are either prohibitive or offer approximate solutions. In this paper we introduce a simple
and efficient algorithm for discovering neural wirings (DNW). Our method searches the space of all possible wirings with a simple modification of the backwards pass.

Recent work in randomly wired neural networks \cite{35} aims to explore the space of novel neural network wirings. Intriguingly, they show that constructing neural networks with random graph algorithms often outperforms a manually engineered architecture. However, these wirings are fixed at training.

Our method for discovering neural wirings is as follows: First, we consider the sole constraint that the total number of edges in the neural graph is fixed to be $k$. Initially we randomly assign a weight to each edge. We then choose the weighted edges with the highest magnitude and refer to the remaining edges as hallucinated. As we train, we modify the weights of all edges according to a specified update rule. Accordingly, a hallucinated edge may strengthen to a point it replaces a real edge. We tailor the update rule so that when swapping does occur, it is beneficial.

We consider the application of DNW for static and dynamic neural graphs. In the static regime each node has a single output and the graphical structure is acyclic. In the case of a dynamic neural graph we allow the state of a node to vary with time. Dynamic neural graphs may contain cycles and express popular sequential models such as LSTMs \cite{11}. As dynamic neural graphs are strictly more expressive than static neural graphs, they can also express feed-forward networks (as in Figure 1).

Our work may also be regarded as a unification between the problem of neural architecture search and sparse neural network learning. As NAS becomes less restrictive and more fine grained, finding a good architecture is akin to finding a sparse sub-network of the complete graph. Accordingly, DNW provides an effective mechanism for discovering sparse networks in a single training run.

The Lottery Ticket Hypothesis \cite{8,9} demonstrates that dense feed-forward neural networks contain so-called winning-tickets. These winning-tickets are sparse subnetworks which, when reset to their initialization and trained in isolation, reach an accuracy comparable to their dense counterparts. This hypothesis articulate an advantage of overparameterization during training – having more parameters increases the chance of winning the initialization lottery. We leverage this idea to train a sparse neural network without retraining or fine-tuning. Though we only ever use a small percentage of the weights during the forward pass, we still play the lottery with a combinatorial number of sub-networks.

We demonstrate the efficacy of DNW on small and large scale data-sets, and for feed-forward, recurrent, continuous, and sparse networks. Notably, we augment MobileNetV1 \cite{12} with DNW to achieve a 10% improvement on ImageNet \cite{5} from the hand engineered MobileNetV1 at $\sim$ 41M FLOPs.

2 Discovering Neural Wirings

In this section we describe our method for jointly discovering the structure and learning the parameters of a neural network. We first consider the algorithm in a familiar setting, a feed-forward neural network, which we abstract as a static neural graph. We then present a more expressive dynamic neural graph which extends to discrete and continuous time and generalizes feed-forward, recurrent, and continuous time neural networks.

\footnote{We follow \cite{36,22} and define FLOPS as the number of Multiply Adds.}
2.1 Static Neural Graph

A static neural graph is a directed acyclic graph \( G = (\mathcal{V}, \mathcal{E}) \) consisting of nodes \( \mathcal{V} \) and edges \( \mathcal{E} \subseteq \mathcal{V} \times \mathcal{V} \). The state of a node \( v \in \mathcal{V} \) is given by the random variable \( Z_v \). At each node \( v \) we apply a function \( f_v \) and with each edge \((u, v)\) we associate a weight \( w_{uv} \). In the case of a multi-layer perceptron, \( f \) is simply a parameter-free non-linear activation like ReLU [17].

For any set \( A \subseteq \mathcal{V} \) we let \( Z_A \) denote \((Z_v)_{v \in A}\) and so \( Z_{\mathcal{V}} \) is the state of all nodes in the network.

\[ Z_v = \begin{cases} f_v \left( \sum_{(u,v) \in \mathcal{E}} w_{uv} Z_u \right) & v \in \mathcal{V} \setminus \mathcal{V}_0 \\ g_\phi (\mathcal{X}) & v \in \mathcal{V}_0. \end{cases} \] (1)

For brevity, we let \( \mathcal{I}_v \) denote the "input" to node \( v \), where \( \mathcal{I}_v \) may be expressed as:

\[ \mathcal{I}_v = \sum_{(u,v) \in \mathcal{E}} w_{uv} Z_u. \] (2)

In this work we consider the case where the input and output of each node is a two-dimensional matrix, commonly referred to as a channel. Each node performs a non-linear activation followed by normalization and convolution (which may be strided to reduce the spatial resolution). As in [35], we no longer conform to the traditional notion of "layers" in a deep network.

The combination of a separate \( 3 \times 3 \) convolution for each channel (depthwise convolution) followed by a \( 1 \times 1 \) convolution (pointwise convolution) is often referred to as a depthwise separable convolution, and is essential in efficient network design [12, 22]. With a static neural graph this process may be interpreted equivalently as a \( 3 \times 3 \) convolution at each node followed by information flow on a complete bipartite graph.

2.2 Discovering a \( k \)-Edge neural graph

We now outline our method for discovering the edges of a static neural graph subject to the constraint that the total number of edges must not exceed \( k \).

We consider a set of real edges \( \mathcal{E} \) and a set of hallucinated edges \( \mathcal{E}_{\text{hal}} = \mathcal{V} \times \mathcal{V} \setminus \mathcal{E} \). The real edge set is comprised of the \( k \)-edges which have the largest magnitude weight. As we allow the magnitude of the weights in both sets to change throughout training the edges in \( \mathcal{E}_{\text{hal}} \) may replace those in \( \mathcal{E} \).

Consider a hallucinated edge \((u, v) \notin \mathcal{E}\). If the gradient is pushing \( \mathcal{I}_v \) in a direction which aligns with \( Z_u \), then our update rule strengthens the magnitude of the weight \( w_{uv} \). If this alignment happens consistently then \( w_{uv} \) will be eventually be strong enough to enter the real edge set \( \mathcal{E} \). As the total...
We first consider the simple case where the state of the nodes are fixed and so the rest of the parameters update as in traditional SGD & Backprop. This gradient flow is important, we do not allow the gradient to flow through the hallucinated edges (as in Algorithm 1).

![Diagram](image)

**Figure 3:** Gradient flow: On the forward pass we use only on the real edges. On the backwards pass we allow the gradient to flow to but not through the hallucinated edges (as in Algorithm 1).

In practice we may also include a momentum and weight decay term in the weight update rule (line 10 in Algorithm 1). In fact, the weight update rule looks nearly identical to that in traditional SGD & Backprop but for one key difference: we allow the gradient to flow to edges which did not exist during the forward pass. Importantly, we do not allow the gradient to flow through these edges and so the rest of the parameters update as in traditional SGD & Backprop. This gradient flow is illustrated in Figure 3.

Under certain conditions we formally show that swapping an edge from $E_{\text{hal}}$ to $E$ decreases the loss $\mathcal{L}$. We first consider the simple case where the hallucinated edge $(i, k)$ replaces $(j, k) \in E$. In Section 3 we discuss the proof to a more general case.

We let $\tilde{w}$ to denote the weight $w$ after the weight update rule $\tilde{w}_{uv} = w_{uv} + \left( Z_u, -\alpha \frac{\partial \mathcal{L}}{\partial Z_u} \right)$. We assume that $\alpha$ is small enough so that $\text{sign}(\tilde{w}) = \text{sign}(w)$.

**Claim:** Assume $\mathcal{L}$ is Lipschitz continuous. There exists a learning rate $\alpha^* > 0$ such that for $\alpha \in (0, \alpha^*)$ the process of swapping $(i, k)$ for $(j, k)$ will decrease the loss on the mini-batch when the state of the nodes are fixed and $|w_{ik}| < |w_{jk}|$ but $|\tilde{w}_{ik}| > |\tilde{w}_{jk}|$.

---

**Algorithm 1** DNW-Train($\mathcal{V}, \mathcal{V}_0, \mathcal{Y}, g, h, \{ f_{\theta_v} \}_{v \in \mathcal{V}}, p_{xy}, k, \mathcal{L}$)

1: for each pair of nodes $(u, v)$ such that $u < v$ do
2: Initialize $w_{uv}$ by independently sampling from a uniform distribution.
3: for each training iteration do
4: Sample mini batch of data and labels $(\mathcal{X}, \mathcal{Y}) = \{(x_i, y_i)\}$ using $p_{xy}$
5: $E \leftarrow \{(u, v) : |w_{uv}| \geq \tau\}$ where $\tau$ is chosen so that $|E| = k$
6: $Z_v \leftarrow \left\{ \sum_{(u, v) \in E} w_{uv} Z_u \right\} \quad \forall v \in \mathcal{V} \setminus \mathcal{V}_0$
7: $\hat{Y} = h_{\phi}(\{Z_v\}_{v \in \mathcal{V}})$
8: Update $\phi, \{ \theta_v \}_{v \in \mathcal{V}}, \psi$ via SGD & Backprop using loss $\mathcal{L}(\hat{Y}, \mathcal{Y})$
9: for each pair of nodes $(u, v)$ such that $u < v$ do
10: $w_{uv} \leftarrow w_{uv} + \left( Z_u, -\alpha \frac{\partial \mathcal{L}}{\partial Z_u} \right)$
11: Recall $\mathcal{T}_v = \sum_{(u, v) \in E} w_{uv} Z_u$

---

4 Weight decay [18] may in fact be very helpful for eliminating dead ends.
We refer to this model as a dynamic neural graph. In Section E of the appendix we discuss the effect of \( \theta \) on \( w_{uv} \). In Section F of the Appendix, we show that the update rule is equivalent to a straight-through estimator \( \hat{A} \).

### 2.3 Dynamic Neural Graph

We now consider a more general setting where the state of each node \( Z_v(t) \) may vary through time. We refer to this model as a dynamic neural graph.

The initial conditions of a dynamic neural graph are given by

\[
Z_v(0) = \begin{cases} 
   g_v^{(v)}(\mathcal{X}) & v \in \mathcal{V}_0 \\
   0 & v \notin \mathcal{V} \setminus \mathcal{V}_0
\end{cases}
\]

(10)

where \( \mathcal{V}_0 \) is a designated set of input nodes, which may now have parents.

**Discrete Time Dynamics:** For a discrete time neural graph we consider times \( \ell \in \{0, 1, \ldots, L\} \). The dynamics are then given by

\[
Z_v(\ell + 1) = f_\theta_v \left( \sum_{(u,v) \in \mathcal{E}} w_{uv} Z_u(\ell), \, \ell \right)
\]

(11)

and the network output is \( \hat{Y} = h_\theta(Z_{\mathcal{V}_f}(L)) \). We may express equation (11) more succinctly as

\[
Z_{\mathcal{V}}(\ell + 1) = f_\theta \left( A_{\mathcal{G}} Z_{\mathcal{V}}(\ell), \, \ell \right)
\]

(12)
With this model we may jointly learn the structure of the graph along with the parameters on ImageNet. We may also apply Algorithm 1 to learn the structure of dynamic neural graphs. One may use where

\[ Z_v(\ell) = (Z_v(\ell))_{\ell \in V}, f_\theta(z, \ell) = (f_\theta(z_v, \ell))_{z \in V}, \] 

and \( A_G \) is the weighted adjacency matrix for graph \( G \). Equation 12 suggests the following interpretation: At each time step we send information through the edges using \( A_G \) then apply a function at each node.

\[ \nabla Z_v(t) = f_\theta(A_G Z_v(t), t). \] 

(13)

Interestingly, if \( V_0 \) is a strict subset of \( V \) we uncover an Augmented Neural ODE [7].

The discrete time case is unifying in the sense that it may also express any static neural graph. In Figure 1 we illustrate that an MLP may also be expressed by a discrete time neural graph. Additionally, the discrete time dynamics are able to capture sequential models such as LSTMs, as long as we allow input to flow into \( V_0 \) at any time.

In continuous time it is not immediately obvious how to incorporate strided convolutions. One approach is to keep the same spatial resolution throughout and pad with zeros after applying strided convolutions. This design is illustrated by Figure 2.

We may also apply Algorithm 1 to learn the structure of dynamic neural graphs. One may use backpropagation through time [3] and the adjoint-sensitivity method [3] for optimization in the discrete and continuous time settings respectively. In Section 3.1, we demonstrate empirically that our method performs better than a random graph, though we do not formally justify the application of our algorithm in this setting.

2.4 Implementation details for Large Scale Experiments

For large scale experiments we do not consider the dynamic case as optimization is too expensive. Accordingly, we now present our method for constructing a large and efficient static neural graph. With this model we may jointly learn the structure of the graph along with the parameters on ImageNet [5]. As illustrated by Table 5, our model closely follows the structure of MobileNetV1 [12], and so we refer to it as MobileNetV1-DNW. We consider a separate neural graph for each spatial resolution – the output of graph \( G_i \) is the input of graph \( G_{i+1} \). For width multiplier \( d \) and spatial resolution \( s \times s \) we constrain MobileNetV1-DNW to have the same number of edges for resolution \( s \times s \) as the corresponding MobileNetV1 \( \times d \). We use a slightly smaller width multiplier to obtain a model with similar FLOPs as we do not explicitly reduce the number of depthwise convolutions in MobileNetV1-DNW. However, we do find that neurons often die (have no output) and we may then skip the depthwise convolution during inference. Note that if we interpret a pointwise convolution with \( c_1 \) input channels and \( c_2 \) output channels as a complete bipartite graph then the number of edges is simply \( c_1 \times c_2 \).

We also constrain the longest path in graph \( G \) to be equivalent to the number of layers of the corresponding MobileNetV1. We do so by partitioning the nodes \( V \) into blocks \( B = \{B_0, ..., B_{L-1}\} \) where \( B_i \) is the input nodes \( V_0 \), \( B_{j-1} \) is output nodes \( V_{E} \), and we only allow edges between nodes in \( B_i \) and \( B_j \) if \( i < j \). The longest path in a graph with \( L \) blocks is then \( L - 1 \). Splitting the graph into blocks also improves efficiency as we may operate on one block at a time. The structure of MobileNetV1 may be recovered by considering a complete bipartite graph between adjacent blocks.

The operation \( f_{\theta} \) at each non-output node is a batch-norm [13] (2 parameters), ReLU [12], \( 3 \times 3 \) convolution (9 parameters) triplet. There are no operations at the output nodes. When the spatial resolution decreases in MobileNetV1 we change the convolutional stride of the input nodes to 2.

In models denoted MobileNetV1-DNW-Small \( \times d \) we also limit the last fully connected (FC) layer to have the same number of edges as the FC layer in MobileNetV1 \( \times d \). In the normal setting of MobileNetV1-DNW we do not modify the last FC layer.

3 Experiments

In this section we demonstrate the effectiveness of DNW for image classification in small and large scale settings. We begin by comparing our method with a random wiring on a small scale dataset.
We begin by training tiny classifiers for the CIFAR-10 dataset. As standard, we have divided the results of Table 3 to consider models which have similar FLOPs. Although our network follows the simple structure of MobileNetV1 (recurrent and continuous time networks are more expensive to optimize due to lack of parallelization). The image is first downsampled and we may therefore skip many operations. More FLOPs as nearly all depthwise convolutions must be performed. DNW allows neurons to die and model. This allows us to experiment in static, discrete time, and continuous settings. Next we explore the use of DNW at scale with experiments on ImageNet and compare DNW with other methods of discovering network structures. Finally we use our algorithm to effectively train sparse neural networks without retraining or finetuning.

Throughout this section we let RG denote our primary baseline – a randomly wired graph. To construct a randomly wired graph with k-edges we assign a uniform random weight to each edge then pick the k edges with the largest magnitude weights. As shown in random graphs often outperform manually designed networks.

### 3.1 Small Scale Experiments For Static and Dynamic Neural Graphs

We begin by training tiny classifiers for the CIFAR-10 dataset. Our initial aim is not to achieve state of the art performance but instead to explore DNW in the static, discrete, and continuous time settings. As illustrated by Table 1, our method outperforms a random graph by a large margin.

The image is first downsampled then each channel is given as input to a node in a neural graph. The static graph uses 5 blocks and the discrete time graph uses 5 time steps. For the continuous case we backprop through the operation of an adaptive ODE solver. The models have 41k parameters. At each node we perform Instance Normalization, ReLU, and a 3 × 3 single channel convolution.

The models have 41k parameters. At each node we perform Instance Normalization, ReLU, and a 3 × 3 single channel convolution.

### 3.2 ImageNet Classification

For large scale experiments on ImageNet we are limited to exploring DNW in the static case (recurrent and continuous time networks are more expensive to optimize due to lack of parallelization). Although our network follows the simple structure of MobileNetV1 we are able to achieve higher accuracy than modern networks which are more advanced and optimized. Notably, MobileNetV2 extends MobileNetV1 by adding residual connections and linear bottlenecks and ShuffleNet introduces channel splits and channel shuffles. The results of the large scale experiments may be found in Table 5.

As standard, we have divided the results of Table 3 to consider models which have similar FLOPs. In the more sparse case (~ 41M FLOPs) we are able to use DNW to boost the performance of MobileNetV1 by 10%. Though random graphs perform extremely well we still observe a 7% boost in performance. In each experiment we train for 250 epochs using Cosine Annealing as the learning rate scheduler with initial learning rate 0.1, as in. Models using random graphs have considerably more FLOPs as nearly all depthwise convolutions must be performed. DNW allows neurons to die and we may therefore skip many operations.

<table>
<thead>
<tr>
<th>Model</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static (RG)</td>
<td>76.1 ± 0.5%</td>
</tr>
<tr>
<td>Static (DNW)</td>
<td>80.9 ± 0.6%</td>
</tr>
<tr>
<td>Discrete Time (RG)</td>
<td>77.3 ± 0.7%</td>
</tr>
<tr>
<td>Discrete Time (DNW)</td>
<td>82.3 ± 0.6%</td>
</tr>
<tr>
<td>Continuous (RG)</td>
<td>78.5 ± 1.2%</td>
</tr>
<tr>
<td>Continuous (DNW)</td>
<td>83.1 ± 0.3%</td>
</tr>
</tbody>
</table>

Table 2: Other methods for discovering wirings (using the architecture described in Table 3) tested on CIFAR-10 shown as mean and std over 5 runs. Models with † first require the complete graph to be trained.

<table>
<thead>
<tr>
<th>Model</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>MobileNetV1 (×0.25)</td>
<td>86.3 ± 0.2%</td>
</tr>
<tr>
<td>MobileNetV1-RG(×0.225)</td>
<td>87.2 ± 0.1%</td>
</tr>
<tr>
<td>No Update Rule</td>
<td>86.7 ± 0.5%</td>
</tr>
<tr>
<td>L1 + Anneal</td>
<td>84.3 ± 0.6%</td>
</tr>
<tr>
<td>TD ρ = 0.95</td>
<td>89.2 ± 0.4%</td>
</tr>
<tr>
<td>Lottery Ticket (one-shot)†</td>
<td>87.9 ± 0.3%</td>
</tr>
<tr>
<td>Fine Tune α = 0.1†</td>
<td>89.4 ± 0.2%</td>
</tr>
<tr>
<td>Fine Tune α = 0.01†</td>
<td>89.7 ± 0.1%</td>
</tr>
<tr>
<td>Fine Tune α = 0.001†</td>
<td>88.7 ± 0.2%</td>
</tr>
<tr>
<td>MobileNetV1-DNW(×0.225)</td>
<td>89.7 ± 0.2%</td>
</tr>
</tbody>
</table>

3We use two 3 × 3 strided convolutions. The first is standard while the second is depthwise-separable.
4We use a 5th order Runge-Kutta method as implemented by (from t = 0 to 1 with tolerance 0.001).
Table 3: ImageNet Experiments (see Section 2.4 for more details). Models with * use the implementations of [22]. Models with multiples asterisks use different image resolutions so that the FLOPs is comparable (see Table 8 in [22] for more details).

<table>
<thead>
<tr>
<th>Model</th>
<th>Params</th>
<th>FLOPs</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>MobileNetV1 (×0.25) [24]</td>
<td>0.5M</td>
<td>41M</td>
<td>50.6%</td>
</tr>
<tr>
<td>X-4 MobileNetV1 [25]</td>
<td>—</td>
<td>&gt;50M</td>
<td>54.0%</td>
</tr>
<tr>
<td>MobileNetV2 (×0.15)* [27]</td>
<td>—</td>
<td>39M</td>
<td>44.9%</td>
</tr>
<tr>
<td>MobileNetV2 (×0.4)**</td>
<td>—</td>
<td>43M</td>
<td>56.6%</td>
</tr>
<tr>
<td>DenseNet (×0.5)* [13]</td>
<td>—</td>
<td>42M</td>
<td>41.1%</td>
</tr>
<tr>
<td>Xception (×0.5)* [21]</td>
<td>—</td>
<td>40M</td>
<td>55.1%</td>
</tr>
<tr>
<td>ShuffleNetV1 (×0.5, g = 3)</td>
<td>—</td>
<td>38M</td>
<td>56.8%</td>
</tr>
<tr>
<td>ShuffleNetV2 (×0.5) [22]</td>
<td>1.4M</td>
<td>41M</td>
<td>60.3%</td>
</tr>
<tr>
<td>MobileNetV1-RG(×0.225)</td>
<td>1.2M</td>
<td>55.7M</td>
<td>53.3%</td>
</tr>
<tr>
<td>MobileNetV1-DNW-Small (×0.15)</td>
<td>0.24M</td>
<td>22.1M</td>
<td>50.3%</td>
</tr>
<tr>
<td>MobileNetV1-DNW (×0.225)</td>
<td>0.4M</td>
<td>41.2M</td>
<td>59.9%</td>
</tr>
<tr>
<td>MobileNetV1-DNW-Small (×0.225)</td>
<td>1.1M</td>
<td>42.1M</td>
<td>60.9%</td>
</tr>
<tr>
<td>MnasNet-search1 [30]</td>
<td>1.9M</td>
<td>65M</td>
<td>64.9%</td>
</tr>
<tr>
<td>MobileNetV1-DNW(×0.3)</td>
<td>1.3M</td>
<td>66.7M</td>
<td>65.0%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Model</th>
<th>Params</th>
<th>FLOPs</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>MobileNetV1 (×0.5)</td>
<td>1.3M</td>
<td>149M</td>
<td>63.7%</td>
</tr>
<tr>
<td>MobileNetV2 (×0.6)*</td>
<td>—</td>
<td>141M</td>
<td>66.6%</td>
</tr>
<tr>
<td>MobileNetV2 (×0.75)**</td>
<td>—</td>
<td>145M</td>
<td>67.9%</td>
</tr>
<tr>
<td>DenseNet (×1)*</td>
<td>—</td>
<td>142M</td>
<td>54.8%</td>
</tr>
<tr>
<td>Xception (×1)*</td>
<td>—</td>
<td>145M</td>
<td>65.9%</td>
</tr>
<tr>
<td>ShuffleNetV1 (×1, g = 3)</td>
<td>—</td>
<td>140M</td>
<td>67.4%</td>
</tr>
<tr>
<td>ShuffleNetV2 (×1)</td>
<td>2.3M</td>
<td>146M</td>
<td>69.4%</td>
</tr>
<tr>
<td>MobileNetV1-RG(×0.49)</td>
<td>1.8M</td>
<td>170M</td>
<td>64.1%</td>
</tr>
<tr>
<td>MobileNetV1-DNW(×0.49)</td>
<td>1.8M</td>
<td>154M</td>
<td>70.4%</td>
</tr>
</tbody>
</table>

3.3 Related Methods

We compare DNW with various methods for discovering neural wirings. In Table 2 we use the structure of MobileNetV1-DNW but try other methods which find k-edge sub-networks. The experiments in Table 2 are conducted using CIFAR-10 [16]. We train for 160 epochs using Cosine Annealing as the learning rate scheduler with initial learning rate α = 0.1 unless otherwise noted.

The Lottery Ticket Hypothesis: The authors of [35] [3] offer an intriguing hypothesis: sparse sub-networks may be trained in isolation when reset to their initialization. However, their method for finding so-called winning tickets is quite expensive as it requires training the full graph from scratch. We compare with one-shot pruning from [2]. One-shot pruning is more comparable in training FLOPs than iterative pruning [8], though both methods are more expensive in training FLOPs than DNW. After training the full network $G_{\text{full}}$ (i.e. no edges pruned) the optimal sub-network $G_k$ with k-edges is chosen by taking the weights with the highest magnitude. In the row denoted Lottery Ticket we retrain $G_k$ using the initialization of $G_{\text{full}}$. We also initialize $G_k$ with the weights of $G_{\text{full}}$ after training – denoted by FT for fine-tune (we try different initial learning rates α). Though these experiments perform comparably with DNW, their training is more expensive as the full graph must initially be trained.

Exploring Randomly Wired Networks for Image Recognition: The authors of [35] explore “a more diverse set of connectivity patterns through the lens of randomly wired neural networks.” They achieve impressive performance on ImageNet [5] using random graph algorithms to generate the structure of a neural network. Their network connectivity, however, is fixed during training. Throughout this section we have a random graph (denoted RG) as our primary baseline – as in [35] we have seen that random graphs outperform hand-designed networks.

No Update Rule: In this ablation on DNW we do not apply the update rule to the hallucinated edges. An edge may only leave the hallucinated edge set if the magnitude of a real edge is sufficiently decreased. This experiment demonstrates the importance of the update rule.

L1 + Anneal: We experiment with a simple pruning technique – start with a fully connected graph and remove edges by magnitude throughout training until there are only $k$ remaining. We found that accuracy was much better if we added an L1 regularization term.
Table 4: Training a tuned version of ResNet50 on ImageNet with modern optimization techniques, as in Appendix C of 
For All Layers Sparse, every layer has a fixed sparsity. In contrast, we leave the very first convolution dense for First Layer Dense. The parameters in the first layer constitute only 0.04% of the total network.

<table>
<thead>
<tr>
<th>Method</th>
<th>Weights (%)</th>
<th>Top-1 Accuracy</th>
<th>Top-5 Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sparse Networks from Scratch [6]</td>
<td>10%</td>
<td>72.9%</td>
<td>91.5%</td>
</tr>
<tr>
<td>Ours - All Layers Sparse</td>
<td>10%</td>
<td>74.0%</td>
<td>92.0%</td>
</tr>
<tr>
<td>Ours - First Layer Dense</td>
<td>10%</td>
<td>75.0%</td>
<td>92.5%</td>
</tr>
<tr>
<td>Sparse Networks from Scratch [6]</td>
<td>20%</td>
<td>74.9%</td>
<td>92.5%</td>
</tr>
<tr>
<td>Ours - All Layers Sparse</td>
<td>20%</td>
<td>76.2%</td>
<td>93.0%</td>
</tr>
<tr>
<td>Ours - First Layer Dense</td>
<td>20%</td>
<td>76.6%</td>
<td>93.4%</td>
</tr>
<tr>
<td>Sparse Networks from Scratch [6]</td>
<td>30%</td>
<td>75.9%</td>
<td>92.9%</td>
</tr>
<tr>
<td>Ours - All Layers Sparse</td>
<td>30%</td>
<td>76.9%</td>
<td>93.4%</td>
</tr>
<tr>
<td>Ours - First Layer Dense</td>
<td>30%</td>
<td>77.1%</td>
<td>93.5%</td>
</tr>
<tr>
<td>Sparse Networks from Scratch [6]</td>
<td>100%</td>
<td>77.0%</td>
<td>93.5%</td>
</tr>
<tr>
<td>Ours - Dense Baseline</td>
<td>100%</td>
<td>77.5%</td>
<td>93.7%</td>
</tr>
</tbody>
</table>

Targeted Dropout: The authors of [10] present a simple and effective method for training a network which is robust to subsequent pruning. Their method outperforms variational dropout [23] and L0 pruning [21]. We compare with Weight Dropout/Pruning from [10], which we denote as TD. Section B of the Appendix contains more information, experimental details, and hyperparameter trials for the Targeted Dropout experiments, though we provide the best result in Table 2.

Neural Architecture Search: As illustrated by Table 3, our network (with a very simple MobileNetV1 like structure) is able to achieve comparable accuracy to an expensive method which performs neural architecture search using reinforcement learning [30].

3.4 Training Sparse Neural Networks

We may apply our algorithm for Discovering Neural Wirings to the task of training sparse neural networks. Importantly, our method requires no fine-tuning or retraining to discover a sparse subnetworks – the sparsity is maintained throughout training. This perspective was guided by the work of Dettmers and Zettelmoyer in [6], though we would like to highlight some differences. Their work enables faster training, though our backwards pass is still dense. Moreover, their work allows for a redistribution of parameters across layers whereas we consider a fixed sparsity per layer.

Our algorithm for training a sparse neural network is similar to Algorithm [1], though we implicitly treat each convolution as a separate graph where each parameter is an edge. For each convolutional layer on the forwards pass, we use the top \( k \% \) of the parameters chosen by magnitude. On the backwards pass we allow the gradient to flow to, but not through, all weights that were zeroed out on the forwards pass. All weights receive gradients as if they existed on the forwards pass, regardless of if they were zeroed out.

As in [6] we leave the biases and batchnorm dense. We compare with the result in Appendix C of [6], as we also use a tuned version of a ResNet50 that uses modern optimization techniques such as cosine learning rate scheduling and warmup[1]. We train for 100 epochs and showcase our results in Table 4.

4 Conclusion

We present a novel method for discovering neural wirings. With a simple algorithm we demonstrate a significant boost in accuracy over randomly wired networks. We benefit from overparameterization during training even when the resulting model is sparse. Just as in [25], our networks are free from the typical constraints of NAS. This work suggests exciting directions for more complex and efficient methods of discovering neural wirings.

[1] We adapt the code from https://github.com/NVIDIA/DeepLearningExamples/tree/master/PyTorch/Classification/RN50v1.5 using the exact same hyperparameters but training for 100 epochs.
Acknowledgments

We thank Sarah Pratt, Mark Yatskar and the Beaker team. We also thank Tim Dettmers for his assistance and guidance in the experiments regarding sparse networks. This work is in part supported by DARPA N66001-19-2-4031, NSF IIS-165205, NSF IIS-1637479, NSF IIS-1703166, Sloan Fellowship, NVIDIA Artificial Intelligence Lab, the Allen Institute for Artificial Intelligence, and the AI2 fellowship for AI. Computations on beaker.org were supported in part by credits from Google Cloud.

References


