

---

# Neural Plasticity Networks

---

**Yang Li**

Georgia State University,  
Atlanta, GA, USA  
yli93@student.gsu.edu

**Shihao Ji**

Georgia State University,  
Atlanta, GA, USA  
sji@gsu.edu

## Abstract

Neural plasticity is an important functionality of human brain, in which number of neurons and synapses can shrink or expand in response to stimuli throughout the span of life. We model this dynamic learning process as an  $L_0$ -norm regularized binary optimization problem, in which each unit of a neural network (e.g., weight, neuron or channel, etc.) is attached with a stochastic binary gate, whose parameters determine the level of activity of a unit in the network. At the beginning, only a small portion of binary gates (therefore the corresponding neurons) are activated, while the remaining neurons are in a hibernation mode. As the learning proceeds, some neurons might be activated or deactivated if doing so can be justified by the cost-benefit tradeoff measured by the  $L_0$ -norm regularized objective. As the training gets mature, the probability of transition between activation and deactivation will diminish until a final hardening stage. We demonstrate that all of these learning dynamics can be modulated by a single parameter  $k$  seamlessly. Our neural plasticity network (NPN) can prune or expand a network depending on the initial capacity of network provided by the user; it also unifies dropout (when  $k = 0$ ), traditional training of DNNs (when  $k = \infty$ ) and interpolates between these two. To the best of our knowledge, this is the first learning framework that unifies network sparsification and network expansion in an end-to-end training pipeline. Extensive experiments on synthetic dataset and multiple image classification benchmarks demonstrate the superior performance of NPN. We show that both network sparsification and network expansion can yield compact models of similar architectures and of similar predictive accuracies that are close to or sometimes even higher than baseline networks. We plan to release our code to facilitate the research in this area.

## 1 Introduction

Deep Neural Networks (DNNs) have achieved great success in a broad range of applications in image recognition [1], natural language processing [2], and game playing [3]. Along with this success is a paradigm shift from feature engineering to architecture design. Latest DNN architectures, such as ResNet [4], DenseNet [5] and Wide-ResNet [6], incorporate hundreds of millions of parameters to achieve state-of-the-art predictive performance. However, the expanding number of parameters not only increases the risk of overfitting, but also leads to high computational costs. A practical solution to this problem is network sparsification, by which weights, neurons or channels can be pruned or sparsified significantly with minor accuracy losses [7, 8], and sometimes sparsified networks can even achieve higher accuracies due to the regularization effects of the network sparsification algorithms [9, 10]. Driven by the widespread applications of DNNs in resource-limited embedded systems, there has been an increasing interest in sparsifying networks recently [7, 8, 11, 12, 13, 14, 9, 10].

A less explored alternative is network expansion, by which weights, neurons or channels can be gradually added to grow a small network to improve its predictive accuracy. This paradigm is in the

opposite to network sparsification, but might be more desirable because (1) we don't need to set an upper-bound on the network capacity (e.g., number of weights, neurons or channels, etc.) to start with, and the network can shrink or expand as needed for a given task; (2) it's computationally more efficient to train a small network and expand it to a larger one as redundant neurons are less likely to emerge during the whole training process; and (3) network expansion is more biologically plausible than network sparsification according to our current understanding to human brain development [15].

In this paper, we propose Neural Plasticity Networks (NPNs) that unify network sparsification and network expansion in an end-to-end training pipeline, in which number of neurons and synapses can shrink or expand as needed to solve a given learning task. We model this dynamic learning process as an  $L_0$ -norm regularized binary optimization problem, in which each unit of a neural network (e.g., weight, neuron or channel, etc.) is attached with a stochastic binary gate, whose parameters determine the level of activity of a unit in the whole network. The activation or deactivation of a unit is completely data-driven as long as doing so can be justified by the cost-benefit tradeoff measured by the  $L_0$ -norm regularized objective. As a result, given a network architecture either large or small, our NPN can automatically perform sparsification or expansion without human intervention until a suitable network capacity is reached.

NPN is built on top of the  $L_0$ -ARM algorithm of Li et al. [16]. However, the original  $L_0$ -ARM algorithm only explores network sparsification, in which it demonstrates state-of-the-art performance at pruning networks, while here we extend this framework to network expansion. On the algorithmic side, we further investigate the Augment-Reinforce-Merge (ARM) [17], a recently proposed unbiased gradient estimator for binary latent variable models. We show that due to the flexibility of ARM, many smooth or non-smooth parametric functions, such as scaled sigmoid or hard sigmoid, can be used to parameterize the  $L_0$ -norm regularized binary optimization problem and the unbiasedness of the ARM estimator is retained, while a closely related hard concrete estimator [13] has to rely on the hard sigmoid function for binary optimization. It is this difference that entails NPN the capability of shrinking or expanding network capacity as needed for a given task. We also introduce a learning stage scheduler for NPN and demonstrate that many training stages of network sparsification and expansion, such as pre-training, sparsification/expansion and fine-tuning, can be modulated by a single parameter  $k$  seamlessly; along the way, we also give a new interpretation of dropout [18]. Extensive experiments on synthetic dataset and multiple public datasets demonstrate the superior performance of NPNs for network sparsification and network expansion with fully connected layers and convolutional layers. Our experiments show that both network sparsification and network expansion can converge to similar network capacities with similar accuracies even though they are initialized with networks of different sizes. To the best of our knowledge, this is the first learning framework that unifies network sparsification and network expansion in an end-to-end training pipeline modulated by a single parameter.

The remainder of the paper is organized as follows. In Sec. 2 we describe the  $L_0$ -norm regularized empirical risk minimization for NPN and its solver  $L_0$ -ARM [16] for network sparsification. A new learning stage scheduler for NPN is introduced in Sec. 3. We then extend NPN to network expansion in Sec. 4, followed by the related work in Sec. 5. Experimental results are presented in Sec. 6. Conclusions and future work are discussed in Sec. 7.

## 2 Neural Plasticity Networks: Formulation

Our Neural Plasticity Network (NPN) is built on top of  $L_0$ -ARM [16], a recently proposed algorithm for network sparsification. We extend  $L_0$ -ARM to network expansion, and unify network sparsification and expansion in an end-to-end training pipeline. For the sake of clarity, we first introduce NPN in the context of network sparsification, and later extend it to network expansion. The formulation below largely follows that of  $L_0$ -ARM [16].

Given a training set  $D = \{(\mathbf{x}_i, y_i), i = 1, 2, \dots, N\}$ , where  $\mathbf{x}_i$  denotes the input and  $y_i$  denotes the target, a neural network is a function  $h(\mathbf{x}; \boldsymbol{\theta})$  parametrized by  $\boldsymbol{\theta}$  that fits to the training data  $D$  with the goal of achieving good generalization to unseen test data. To optimize  $\boldsymbol{\theta}$ , typically a regularized empirical risk is minimized, which contains two terms – a data loss over training data and a regularization loss over model parameters. Empirically, the regularization term can be weight decay or Lasso, i.e., the  $L_2$  or  $L_1$  norm of model parameters.

Intuitively, network sparsification or expansion is a model selection problem, in which a suitable model capacity is selected for a given learning task. In this problem, how to measure model complexity is a core issue. The Akaike Information Criterion (AIC) [19] and the Bayesian Information Criterion (BIC) [20], well-known model selection criteria, measure model complexity by counting number of non-zero parameters. Since the  $L_2$  or  $L_1$  norm only imposes shrinkage on large values of  $\theta$ , the resulting model parameters  $\theta$  are often manifested by smaller magnitudes but none of them are exact zero. Therefore, the  $L_2$  or  $L_1$  norm is not suitable for measuring model complexity. A more appealing alternative is the  $L_0$  norm as the  $L_0$ -norm of model parameters measures *explicitly* the number of non-zero parameters, which is the *exact* model complexity measured by AIC and BIC. With the  $L_0$  regularization, the empirical risk objective can be written as

$$\mathcal{R}(\theta) = \frac{1}{N} \sum_{i=1}^N \mathcal{L}(h(x_i; \theta), y_i) + \lambda \|\theta\|_0 \quad (1)$$

where  $\mathcal{L}(\cdot)$  denotes the data loss over training data  $D$ , such as the cross-entropy loss for classification or the mean squared error (MSE) for regression, and  $\|\theta\|_0$  denotes the  $L_0$ -norm over model parameters, i.e., the number of non-zero weights, and  $\lambda$  is a regularization hyperparameter that balances between data loss and model complexity. For network sparsification, minimizing of Eq. 1 will drive the redundant or insignificant weights to be exact zero and thus pruned away. For network expansion, adding additional neurons will increase model complexity (the second term) but potentially can reduce data loss (the first term) and therefore the total loss. Thus, we will use Eq. 1 as our guiding principle for sparsifying or expanding a network.

To represent a sparsified network, we attach a binary random variable  $z$  to each element of model parameters  $\theta$ . Therefore, we can reparameterize the model parameters  $\theta$  as an element-wise product of non-zero parameters  $\tilde{\theta}$  and binary random variables  $z$ :

$$\theta = \tilde{\theta} \odot z, \quad (2)$$

where  $z \in \{0, 1\}^{|\theta|}$ , and  $\odot$  denotes the element-wise product. As a result, Eq. 1 can be rewritten as:

$$\mathcal{R}(\tilde{\theta}, z) = \frac{1}{N} \sum_{i=1}^N \mathcal{L}(h(x_i; \tilde{\theta} \odot z), y_i) + \lambda \sum_{j=1}^{|\tilde{\theta}|} \mathbf{1}_{[z_j \neq 0]}, \quad (3)$$

where  $\mathbf{1}_{[c]}$  is an indicator function that is 1 if the condition  $c$  is satisfied, and 0 otherwise. Note that both the first term and the second term of Eq. 3 are not differentiable w.r.t.  $z$ . Therefore, further approximations need to be considered.

Fortunately, we can approximate Eq. 3 through an inequality from stochastic variational optimization [21]. Specifically, given any function  $\mathcal{F}(z)$  and any distribution  $q(z)$ , the following inequality holds

$$\min_z \mathcal{F}(z) \leq \mathbb{E}_{z \sim q(z)} [\mathcal{F}(z)], \quad (4)$$

i.e., the minimum of a function is upper bounded by the expectation of the function. With this result, we can derive an upper bound of Eq. 3 as follows.

Since  $z_j, \forall j \in \{1, \dots, |\theta|\}$  is a binary random variable, we assume  $z_j$  is subject to a Bernoulli distribution with parameter  $\pi_j \in [0, 1]$ , i.e.  $z_j \sim \text{Ber}(z; \pi_j)$ . Thus, we can upper bound  $\min_z \mathcal{R}(\tilde{\theta}, z)$  by the expectation

$$\begin{aligned} \hat{\mathcal{R}}(\tilde{\theta}, \pi) &= \mathbb{E}_{z \sim \text{Ber}(z; \pi)} \mathcal{R}(\tilde{\theta}, z) \\ &= \mathbb{E}_{z \sim \text{Ber}(z; \pi)} \left[ \frac{1}{N} \sum_{i=1}^N \mathcal{L}(h(x_i; \tilde{\theta} \odot z), y_i) \right] + \lambda \sum_{j=1}^{|\tilde{\theta}|} \pi_j. \end{aligned} \quad (5)$$

As we can see, now the second term is differentiable w.r.t. the new model parameters  $\pi$ , while the first term is still problematic since the expectation over a large number of binary random variables  $z \in \{0, 1\}^{|\theta|}$  is intractable, so is its gradient.

To minimize Eq. 5,  $L_0$ -ARM utilizes the Augment-Reinforce-Merge (ARM) [17], an unbiased gradient estimator, to this stochastic binary optimization problem. Specifically,

**Theorem 1** (ARM) [17]. For a vector of  $V$  binary random variables  $\mathbf{z} = (z_1, \dots, z_V)$ , the gradient of

$$\mathcal{E}(\phi) = \mathbb{E}_{\mathbf{z} \sim \prod_{v=1}^V \text{Ber}(z_v; g(\phi_v))} [f(\mathbf{z})] \quad (6)$$

w.r.t.  $\phi = (\phi_1, \dots, \phi_V)$ , the logits of the Bernoulli distribution parameters, can be expressed as

$$\nabla_{\phi} \mathcal{E}(\phi) = \mathbb{E}_{\mathbf{u} \sim \prod_{v=1}^V \text{Uniform}(u_v; 0, 1)} \left[ (f(\mathbf{1}_{[u > g(-\phi)]}) - f(\mathbf{1}_{[u < g(\phi)]})) (\mathbf{u} - 1/2) \right], \quad (7)$$

where  $\mathbf{1}_{[u > g(-\phi)]} := \mathbf{1}_{[u_1 > g(-\phi_1)]}, \dots, \mathbf{1}_{[u_V > g(-\phi_V)]}$  and  $g(\phi) = \sigma(\phi) = 1/(1 + \exp(-\phi))$  is the sigmoid function.

Parameterizing  $\pi_j \in [0, 1]$  as  $g(\phi_j)$ , we can rewrite Eq. 5 as

$$\begin{aligned} \hat{\mathcal{R}}(\tilde{\theta}, \phi) &= \mathbb{E}_{\mathbf{z} \sim \text{Ber}(\mathbf{z}; g(\phi))} [f(\mathbf{z})] + \lambda \sum_{j=1}^{|\tilde{\theta}|} g(\phi_j) \\ &= \mathbb{E}_{\mathbf{u} \sim \text{Uniform}(\mathbf{u}; 0, 1)} [f(\mathbf{1}_{[u < g(\phi)]})] + \lambda \sum_{j=1}^{|\tilde{\theta}|} g(\phi_j), \end{aligned} \quad (8)$$

where  $f(\mathbf{z}) = \frac{1}{N} \sum_{i=1}^N \mathcal{L}(h(\mathbf{x}_i; \tilde{\theta} \odot \mathbf{z}), y_i)$ . From Theorem 1, we can evaluate the gradient of Eq. 8 w.r.t.  $\phi$  by

$$\nabla_{\phi} \hat{\mathcal{R}}(\tilde{\theta}, \phi) = \mathbb{E}_{\mathbf{u} \sim \text{Uniform}(\mathbf{u}; 0, 1)} \left[ (f(\mathbf{1}_{[u > g(-\phi)]}) - f(\mathbf{1}_{[u < g(\phi)]})) (\mathbf{u} - 1/2) \right] + \lambda \sum_{j=1}^{|\tilde{\theta}|} \nabla_{\phi_j} g(\phi_j), \quad (9)$$

which is an unbiased and low variance estimator as demonstrated in [17].

**Choice of  $g(\phi)$**  Theorem 1 of ARM defines  $g(\phi) = \sigma(\phi)$ , where  $\sigma(\cdot)$  is the sigmoid function. For the purpose of network sparsification and expansion, we find that this parametric function isn't very effective due to its fixed rate of transition between values 0 and 1. Thanks to the flexibility of ARM, we have a large freedom to design this parametric function  $g(\phi)$ . Apparently, it's straightforward to generalize Theorem 1 for any parametric functions (smooth or non-smooth) as long as  $g: \mathcal{R} \rightarrow [0, 1]$  and  $g(-\phi) = 1 - g(\phi)$ <sup>1</sup>. Example parametric functions that work well in our experiments are the scaled sigmoid function

$$g_{\sigma_k}(\phi) = \sigma(k\phi) = \frac{1}{1 + \exp(-k\phi)}, \quad (10)$$

and the centered-scaled hard sigmoid

$$g_{\bar{\sigma}_k}(\phi) = \min(1, \max(0, \frac{k}{7}\phi + 0.5)), \quad (11)$$

where 7 is introduced such that  $g_{\bar{\sigma}_1}(\phi) \approx g_{\sigma_1}(\phi) = \sigma(\phi)$ . See Figure 1 for some example plots of  $g_{\sigma_k}(\phi)$  and  $g_{\bar{\sigma}_k}(\phi)$  with different  $k$ s. Empirically, we find that  $k = 7$  works well for network sparsification, and  $k = 0.5$  for network expansion. More on this will be discussed when we present results.

One important difference between the hard concrete estimator from Louizos et al. [10] and  $L_0$ -ARM is that the hard concrete estimator has to rely on the hard sigmoid gate to zero out some parameters during training (a.k.a. conditional computation [22]), while  $L_0$ -ARM achieves conditional computation naturally by sampling from the Bernoulli distribution, parameterized by  $g(\phi)$ , where  $g(\phi)$  can be any parametric function (smooth or non-smooth) as shown in Figure 1. The consequence of using the hard sigmoid gate is that once a unit is pruned, the corresponding gradient will be always zero due to the exact 0 gradient at the left tail of the hard sigmoid gate (see Figure 1) and therefore it can never be reactivated in the future. To mitigate this issue of the hard concrete estimator,  $L_0$ -ARM can utilize the scaled sigmoid gate (10), which has non-zero gradient everywhere  $(-\infty, \infty)$ , and therefore a unit can be activated or deactivated freely and thus be plastic.

<sup>1</sup>The second condition is not necessary. But for simplicity, we will impose this condition to select parametric function  $g(\phi)$  that is antithetic. Designing  $g(\phi)$  without this constraint could be a potential area that is worthy of further investigation.

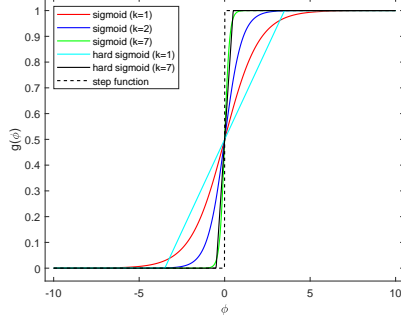


Figure 1: The plots of  $g(\phi)$  with different  $k$  for sigmoid and hard sigmoid functions. A large  $k$  tends to be more effective at sparsifying networks. Best viewed in color.

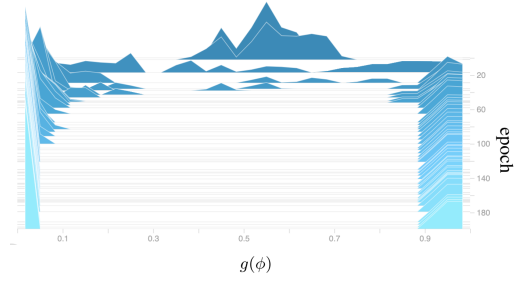


Figure 2: Evolution of the histogram of  $g(\phi)$  over training epochs. All  $g(\phi)$  are initialized by random samples from a normal distribution  $\mathcal{N}(0.5, 0.01)$ , which are split into two spikes during training.

## 2.1 Sparsifying Network Architectures for Inference

After training, we get model parameters  $\tilde{\theta}$  and  $\phi$ . At test time, we can use the expectation of  $\mathbf{z} \sim \text{Ber}(\mathbf{z}; g(\phi))$  as the mask  $\hat{\mathbf{z}}$  to get the final model parameters  $\hat{\theta}$ :

$$\hat{\mathbf{z}} = \mathbb{E}[\mathbf{z}] = g(\phi), \quad \hat{\theta} = \tilde{\theta} \odot \hat{\mathbf{z}}. \quad (12)$$

However, this will not yield a sparsified network for inference since none of the element of  $\hat{\mathbf{z}} = g(\phi)$  is exact zero (unless the hard sigmoid gate  $g_{\sigma_k}(\phi)$  is used). A simple approximation is to set the elements of  $\hat{\mathbf{z}}$  to zero if the corresponding values in  $g(\phi)$  are less than a threshold  $\tau$ , i.e.,

$$\bar{z}_j = \begin{cases} 0, & g(\phi_j) \leq \tau \\ g(\phi_j), & \text{otherwise} \end{cases} \quad j = 1, 2, \dots, |\mathbf{z}| \quad (13)$$

We find that this approximation is very effective in all of our experiments as the histogram of  $g(\phi)$  is widely split into two spikes around values of 0 and 1 after training because of the sharp transition of the scaled sigmoid function (10). See Figure 2 for a typical plot of the histograms of  $g(\phi)$  evolving during training process. We notice that our algorithm isn't very sensitive to  $\tau$ , tuning which incurs negligible impacts to model compactness and accuracies. Therefore, for all of our experiments we set  $\tau = 0.5$  by default. Apparently, better designed  $\tau$  is possible by considering the histogram of  $g(\phi)$ . However, we find this isn't very necessary for all of our experiments in the paper. Therefore, we will consider this histogram-dependent  $\tau$  as our future improvement.

## 3 Learning Stage Scheduler

As far as we know, all network sparsification algorithms either operate in a three-stage of pre-training, sparsification, and fine-tuning [8, 7, 11] or only have one sparsification stage from scratch [10]. It has been shown that the three-stage sparsification leads to better predictive accuracies than the one-stage alternatives [23]. To support this three-stage learning process, previous methods [8, 7, 11] however manage this tedious process manually. Thanks to the flexibility of NPN, we can modulate these learning stages by simply adjusting  $k$  of the  $g_k(\phi)$  function at different training stages. Along the way, we also discover a new interpretation of dropout [18].

### 3.1 Dropout as $k = 0$

When  $k = 0$ , it is readily to verify that  $g_{\sigma_k}(\phi) = g_{\bar{\sigma}_k}(\phi) = 0.5$ , and the objective function (8) is degenerated to

$$\mathcal{R}(\tilde{\theta}, \phi) = \mathbb{E}_{\mathbf{u} \sim \text{Uniform}(\mathbf{u}; 0, 1)} [f(\mathbf{1}_{[\mathbf{u} < 0.5]})] + \lambda |\tilde{\theta}|/2, \quad (14)$$

which is in fact the standard dropout with a dropout probability of 0.5 [18]. Note that the value of 0.5 is due to the artifact of the antithetic constraint on the parametric function  $g(\phi)$ . As we discussed in Sec. 2, this constraint isn't necessary and we have freedom of designing  $g(0) = c$  with  $c \in [0, 1]$ , which corresponds to any dropout probability of the standard dropout. From this point of view, dropout is just a special case of NPN when  $k = 0$ , and this is a new interpretation of dropout.

### 3.2 Pre-training as $k = \infty$ at the beginning of NPN training

At the beginning of NPN training, we initialize all  $\phi$ 's to some positive values, e.g.,  $\phi > 0.1$ . If we set  $k = \infty$ , then  $g_{\sigma_\infty}(\phi) = g_{\bar{\sigma}_\infty}(\phi) = 1$  and the objective function (8) becomes

$$\hat{\mathcal{R}}(\tilde{\theta}, \phi) = \mathbb{E}_{\mathbf{u} \sim \text{Uniform}(\mathbf{u}; 0, 1)} [f(\mathbf{1}_{[\mathbf{u} < 1]})] + \lambda |\tilde{\theta}|, \quad (15)$$

which corresponds to the standard training of DNNs with all neurons activated. Moreover, the gradient w.r.t.  $\phi$  is degenerated to

$$\nabla_\phi \hat{\mathcal{R}}(\tilde{\theta}, \phi) = \mathbb{E}_{\mathbf{u} \sim \text{Uniform}(\mathbf{u}; 0, 1)} \left[ (f(\mathbf{1}_{[\mathbf{u} > 0]}) - f(\mathbf{1}_{[\mathbf{u} < 1]}))(\mathbf{u} - 1/2) \right] + \lambda \sum_{j=1}^{|\tilde{\theta}|} \nabla_{\phi_j} 1 = 0, \quad (16)$$

such that  $\phi$  will not be updated during the training and the architecture is fixed. This corresponds to the pre-training of a network from scratch.

### 3.3 Fine-tuning as $k = \infty$ at the end of NPN training

At the end of NPN training, the histogram of  $g(\phi)$  is typically split to two spikes of values around 0 and 1 as shown in Figure 2. If we set  $k = \infty$ , then the values of  $g(\phi)$  will be exactly 0 or 1. In this case, the gradient w.r.t.  $\phi$  is zero, the neurons with  $g(\phi) = 1$  are activated and the neurons with  $g(\phi) = 0$  are deactivated. This corresponds to the case of fine-tuning a fixed architecture without the  $L_0$  regularization.

### 3.4 Modulating learning stages by $k$

As discussed above, we can now integrate the three-stage of pre-training, sparsification and fine-tuning into one end-to-end pipeline, modulated by a single parameter  $k$ . At the beginning of the training, we set  $k = \infty$  to pre-train a network from scratch. Upon convergence, we can set  $k$  to some small values (e.g.,  $k = 7$ ) to enable the  $L_0$  regularized optimization for network sparsification. After the convergence, we can set  $k = \infty$  again to fine-tune the final learned architecture without the  $L_0$  regularization. To the best of our knowledge, there is no other network sparsification algorithm that supports this three-stage training in a native end-to-end pipeline. As an analogy to the common learning rate scheduler, we call  $k$  as a learning stage scheduler. We will demonstrate this when we present results.

## 4 Network Expansion

So far we have described NPN in the context of network sparsification. Thanks to the flexibility of  $L_0$ -ARM, it is straightforward to extend it to network expansion. Instead of starting from a large network for pruning, we can expand a small network by adding neurons during training, an analogy of growing a brain from small to large. Specifically, given the level of activity of a neuron is determined by its  $\phi$ , a new neuron can be added to the network with a large  $\phi$  such that it will be activated in future training epochs. If this neuron is useful at reducing the  $L_0$ -regularized loss function (8), its  $\phi$  value will be increased such that it will be activated more often in the future; otherwise, it will be gradually deactivated and pruned away in the future. At each training epoch, we add a new neuron to a layer if (a) the  $L_0$ -regularized loss plateaus, and (b) that layer has no redundant neuron. This means that the network doesn't have enough capacity to reduce the loss further, therefore new neurons will be added to the layers that need more capacities. The network expansion will terminate when the  $L_0$ -regularized loss plateaus and no more neurons can be added to any layer of the network. This is the case when the network has a sufficient capacity since all layers start to prune neurons due to the  $L_0$ -norm regularization.

Thanks to the learning stage scheduler discussed in Sec. 3, we can simulate network expansion easily by manipulating  $k$ . Specifically, we can initialize a very large network to represent an upper-bound on network capacity. To start with a small network, we randomly select a small number of neurons and initialize the corresponding  $\phi$ s to large positive values and set all the remaining  $\phi$ 's to large negative values, such that only a small portion of the neurons will be activated while the remaining neurons are in a hibernation mode. Since they will not be activated, these hibernating neurons consume no

computation resources. To pretrain the initial small network, we can train NPN with  $k = \infty$ . Upon convergence, we can randomly activate a few hibernating neurons (under the conditions discussed above) and switch  $k$  to a small value (e.g.,  $k = 0.5$ ). Along with the original neurons, we can optimize the expanded network by reducing the  $L_0$ -regularized loss. In such a way, we can readily simulate network expansion. Upon the network expansion terminates, we can set  $k = \infty$  to fine-tune the final network architecture. In the experiments, we will resort to this approach to simulate network expansion.

## 5 Related Work

Our NPN has a built-in support to network sparsification, network expansion, and can automatically determine an appropriate network capacity for a given learning task. In this section, we will review multiple related researches in the area.

**Network Sparsification** Driven by the widespread applications of DNNs in resource-limited embedded systems, recently there has been an increasing interest in network sparsification [7, 8, 11, 12, 13, 14, 9, 10, 16]. One of the earliest sparsification methods is to prune the redundant weights based on the magnitudes [24], which is proved to be effective in modern CNNs [7]. Although weight sparsification is able to compress networks, it can barely improve computational efficiency due to unstructured sparsity [11]. Therefore, magnitude-based group sparsity is proposed [11, 12], which can prune networks while reducing computation cost significantly. These works are mainly based on the  $L_2$  or  $L_1$  regularization to penalize the magnitude of weights. A more appealing approach is based on the  $L_0$  regularization [10, 16] as this corresponds to the well-known model selection criteria such as AIC [19] and BIC [20]. Our NPN is built on top of  $L_0$ -ARM [16] and extend it for network expansion. In addition, as far as we know almost all the network sparsification algorithms [7, 8, 11, 12] usually proceed in three stages manually: pretrain a full network, prune the redundant weights or filters, and fine-tune the pruned network. In contrast, our NPN can support this three-stage training by simply adjusting the learning stage scheduler  $k$  at different stages in an end-to-end fashion.

**Neural Architecture Search** Another closely related area is neural architecture search [25, 26, 27] that searches for an optimal network architecture for a given learning task. It attempts to determine number of layers, types of layers, layer configurations, different activation functions, etc. Given the extremely large search space, typically reinforcement learning algorithms are utilized for efficient implementations. Our NPN can be categorized as a subset of neural architecture search in the sense that we start with a fixed architecture and aim to determine an optimal capacity (e.g., number of weights, neurons or channels) of a network.

**Dynamic Network Expansion** Compared to network sparsification, network expansion is a relatively less explored area. There are few existing works that can dynamically increase the capacity of network during training. For example, DNC [28] sequentially adds neurons one at a time to the hidden layers of network until the desired approximation accuracy is achieved. [29] proposes to train a denoising autoencoder (DAE) by adding in new neurons and later merging them with other neurons to prevent redundancy. For convolutional networks, [30] proposes to widen or deepen a pretrained network for better knowledge transfer. Recently, a boosting-style method named AdaNet [31] is used to adaptively grow the structure while learning the weights. However, all these approaches either only add neurons or add/remove neurons manually. In contrast, our NPN can add or remove (deactivate) neurons during training as needed without human intervention, and is an end-to-end unified framework for network sparsification and expansion.

## 6 Experimental Results

We evaluate the performance of NPNs on multiple public datasets with different network architectures for network sparsification and network expansion. Specifically, we illustrate how NPN evolves on a synthetic “moons” dataset [32] with a 2-hidden-layer MLP. We also demonstrate LeNet5-Caffe<sup>2</sup> on the MNIST dataset [33], and AlexNet [34] on the CIFAR-10 and CIFAR-100 datasets [35]. Similar to  $L_0$ -ARM [16] and  $L_0$ -HC [13], to achieve computational efficiency, only neuron-level (instead of

<sup>2</sup><https://github.com/BVLC/caffe/tree/master/examples/mnist>

weight-level) sparsification/expansion is considered, i.e., all weights of a neuron or filter are either pruned from or added to a network altogether. For the comparison to the state-of-the-art network sparsification algorithms [10, 14, 13], we refer the readers to  $L_0$ -ARM [16] for more details since NPN is built on top of  $L_0$ -ARM for sparsification.

As discussed in Sec. 2, each neuron in an NPN is attached with a Bernoulli random variable parameterized by  $g(\phi)$ . Therefore, the level of activity of a neuron is determined by the value of  $\phi$ . To initialize an NPN, in our experiments we activate a neuron by setting  $\phi = 3/k$ . Since  $g_{\sigma_k}(\phi) = \sigma(3) \approx 0.95$ , this means that the corresponding neuron has a 95% probability of being activated. Similarly, we set  $\phi = -3/k$  to deactivate a neuron with a 95% probability.

As discussed in Sec. 3, all of our experiments are performed in three stages: (1) pre-training, (2) sparsification/expansion, and (3) fine-tuning, in an end-to-end training pipeline modulated by parameter  $k$ . In pre-training and fine-tuning stages, we set  $k = 5000$  (as a close approximation to  $k = \infty$ ) to train an NPN with a fixed architecture. In sparsification/expansion stage, we set  $k$  to a small value to allow NPNs to search for a suitable network capacity freely.

The final architecture of a network is influenced significantly by two hyperparameters: (1) the regularization strength  $\lambda$ , and (2)  $k$  of the  $g_{\sigma_k}(\cdot)$  function, which determine how aggressively to sparsify or expand a network. Typically, a positive  $\lambda$  is used both for sparsification and expansion. However, in some expansion experiments, we notice that a small negative  $\lambda$  is beneficial because a negative  $\lambda$  essentially encourages more neurons to be activated, which is important for network expansion to achieve competitive accuracies. For the hyperparameter  $k$  used in stage 2, in all of our experiments we set  $k = 7$  for sparsification and  $k = 0.5$  for expansion. The reason that different  $k$ s are used is because for expansion we need to encourage the network to grow and a small  $k$  is more amenable to keep neurons activated.

We use the Adam optimizer [36] with an initial learning rate of 0.001. All of our experiments are performed on NVIDIA Titan-Xp GPUs.

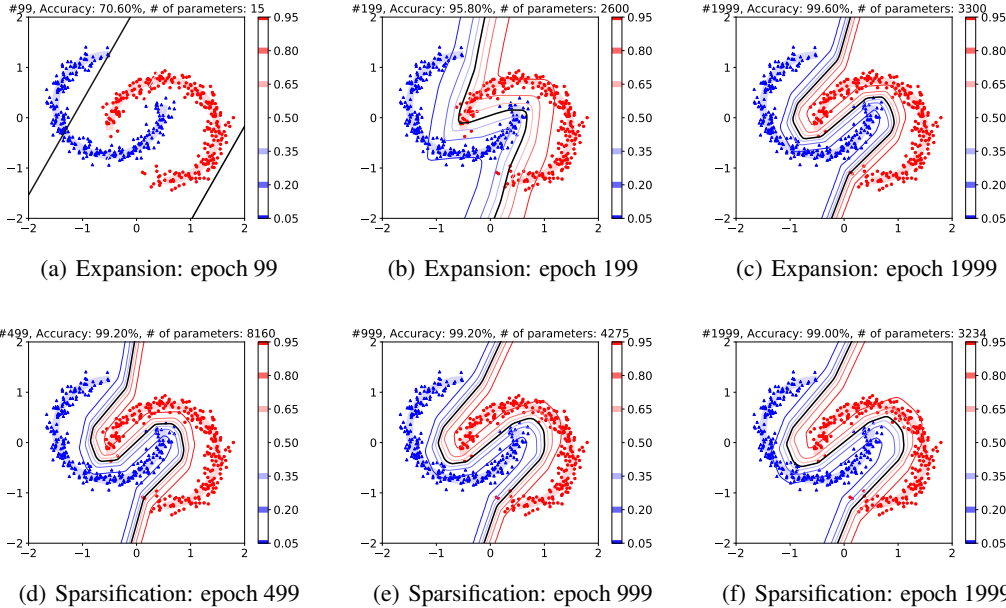


Figure 3: The evolution of the decision boundaries of NPNs for network expansion (a,b,c) and network sparsification (d,e,f).

## 6.1 Synthetic Dataset

To demonstrate that NPNs can adapt their capacities for a learning task, we visualize the learning process of NPNs on a synthetic ‘‘moons’’ dataset [32] for network sparsification and network expansion. The ‘‘moons’’ dataset contains 1000 data points distributed in two moon-shaped clusters



for binary classification. We randomly pick 500 data points for training and use the rest 500 data points for test. Two different MLP architectures are used for network sparsification and network expansion, respectively. For network sparsification, we train an MLP with 2 hidden layers of 100 and 80 neurons, respectively. The input layer has 2 neurons corresponding to the 2-dim coordinates of each data point, which is transformed to a 100-dim vector by a fixed matrix. The output layer has two neurons for binary classification. The overall architecture of the MLP is 2-100 (fixed)-80-2 with the first weight matrix fixed, and the overall number of trainable model parameters is 8160 (excluding biases for clarity). The binary gates are attached to the outputs of the two hidden layers for sparsification. For the expansion experiment, we start an MLP with a very small architecture of 2-100 (fixed)-3-2. Initially, only three neurons at each hidden layer are activated, and therefore the total number of trainable model parameters is 15 (excluding biases for clarity). Apparently, the first MLP is overparameterized for this synthetic binary classification task, while the second MLP is too small and doesn't have enough capacity to solve the classification task with a high accuracy.

To visualize the learning process of NPNs, in Figure 3 we plot the decision boundaries and confidence contours of the NPN-sparsified MLP and NPN-expanded MLP on the test set of "moons". We pick three snapshots from each experiment. The evolution of the decision boundaries of the NPN-expanded MLP is shown in Figure 3 (a, b, c). At the end of pre-training (a. epoch 99), the decision boundary is mostly linear and the capacity of the network is obviously not enough. During the stage 2 expansion (b. epoch 199), more neurons are added to the network and the decision boundary becomes more expressive as manifested by a piece-wise linear function, and at the same time the accuracy is significantly improved to about 96%. At the end of stage 3 fine-tuning (c. epoch 1999), the accuracy reaches 99.2% with more neurons being added. Similarly, Figure 3 (d, e, f) demonstrates the evolution of the decision boundaries of NPN-sparsified MLP on the test set. The model achieves an accuracy of 99.2% at the end of stage 1 pre-training (d. epoch 499). Then 47.6% of weights are pruned without any accuracy loss during stage 2 sparsification (e. epoch 999). The model finally prunes 60.4% of neurons at the end of stage 3 fine-tuning (f. epoch 1999). In this sparsification experiment, across different training stages, the shapes of decision boundaries are appropriately the same even though a large amount of neurons are pruned.

Interestingly, the final architectures achieved by network expansion and network sparsification are very similar (3300 vs. 3234), so are their accuracies (99.6% vs. 99.00%) even though the initial network capacities are quite different (15 vs. 8160). This experiment demonstrates that given an initial network architecture either large or small, NPNs can adapt their capacities to solve a learning task with high accuracies.

## 6.2 MNIST

In the second part of experiments, we run NPNs with LeNet5-Caffe on the MNIST dataset for network sparsification and expansion. LeNet5-Caffe consists of two convolutional layers of 20 and 50 neurons, respectively, interspersed with max pooling layers, followed by two fully-connected layers with 800 and 500 neurons. We start network sparsification from the full LeNet5 architecture (20-50-800-500, in short), while in the expansion experiment we start from a very small network with only 3 neurons in the first two convolutional layers, 48 and 3 neurons in the fully-connected layers (3-3-48-3, in short). We pre-train the NPNs for 100 epochs, followed by sparsification/expansion for 250 epochs and fine-tuning for 150 epochs. For both experiments, we use  $\lambda = (10, 0.5, 0.1, 10)/N$  where  $N$  is the number of training images.

The results are shown in Table 1 and Figure 4. For the sparsification experiment, the NPN achieves 99.36% accuracy at the end of pre-training, and yields a sparse architecture with a minor accuracy drop at the end of sparsification. With the fine-tuning at stage 3, the accuracy reaches 98.91% in the end. For the expansion experiment, the NPN achieves a low accuracy of 93.85% after pre-training due to insufficient capacity of the initial network, then it expands to a larger network and improves the accuracy to 96.71%. Finally, the accuracy reaches 98.31% after fine-tuning. Figure 4 demonstrate the learning processes of NPNs on MNIST for network sparsification and network expansion. It is interesting to note that both network sparsification and expansion reach the similar network capacity with similar classification accuracies at the end of the training even though they are started from two significantly different network architectures. It's worth emphasizing that both sparsification and expansion reach similar accuracies to the baseline model, while over 99% weights are pruned.

Table 1: The network sparsification and expansion with LeNet5 on MNIST. The architecture and accuracy at the end of each stage are shown in the table.

	Stage	Architecture (Number of Parameters)	Accuracy (%)
Baseline	-	20-50-800-500 (4.23e5)	<b>99.40</b>
Sparsification	stage 1	20-50-800-500 (4.23e5)	99.36
	stage 2	7-9-109-30 (5320)	98.90
	stage 3	7-9-109-30 (5320)	98.91
Expansion	stage 1	3-3-48-3 (474)	93.85
	stage 2	8-8-53-8 (2304)	96.71
	stage 3	8-8-53-8 ( <b>2304</b> )	98.31

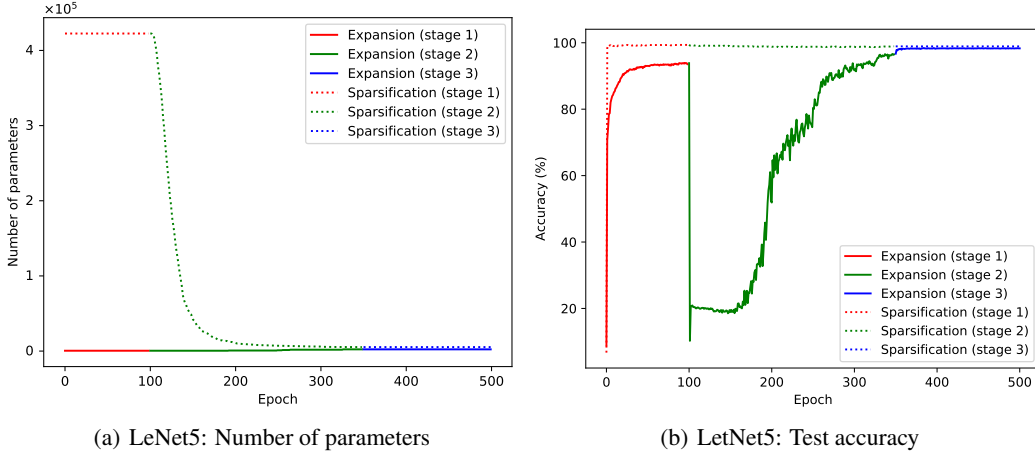


Figure 4: The evolution of network capacity and test accuracy as a function of epoch for NPN network sparsification and expansion with LetNet5 on MNIST.

### 6.3 CIFAR-10/100

In the final part of experiments, we train a modified AlexNet on the CIFAR-10 and CIFAR-100 datasets for network sparsification and network expansion. AlexNet was originally designed for the ImageNet classification. To accommodate the small image of CIFAR-10/100, we slightly modify the kernel sizes and strides of the AlexNet. Specifically, the modified AlexNet architecture still consists of five convolutional layers and three fully-connected layers, but the first convolutional layer has 96 kernels of size  $3 \times 3$  with a stride of 2; the following four convolutional layers have 256, 384, 384, and 256 kernels of size  $3 \times 3$  with a stride of 1, respectively; the three fully-connected layers have 1024, 4096 and 4096 neurons, respectively. Overall, the architecture is 96-256-384-384-256-1024-4096-4096 in short.

As before, the networks are training in three stages. In the sparsification experiment, we pre-train the full network for the first 80 epochs. The network is then sparsified in stage 2 with 420 epochs, and finally is fine-tuned in stage 3 until it converges. In the expansion experiment, we pretrain a small network of 48-63-128-120-120-120-1500 for the first 80 epochs. The network is then expanded in stage 2 until the training loss plateaus. Finally, we fine-tune the network in stage 3 until the training loss plateaus again. For the sparsification experiment on CIFAR-10, we use  $\lambda = 0.02/N$  for all layers, while on CIFAR-100 we use  $\lambda = (0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.02, 0.08)/N$  for different layers. For all the expansion experiments on CIFAR-10/100, we use  $\lambda = -0.01/N$  for all layers. As we discussed earlier, we note that a negative  $\lambda$  is beneficial for network expansion on the CIFAR datasets. The negative  $\lambda$  essentially encourages the network to grow in order to achieve a competitive accuracy.

The results on CIFAR-10 and CIFAR-100 are shown in Tables 2, 3 and Figure 5. Similar to the results on MNIST, both network sparsification and expansion reach the similar network capacities

Table 2: The network sparsification and expansion with AlexNet on CIFAR-10. The architecture and accuracy at the end of each stage are shown in the table.

	Stage	Architecture (Number of Parameters)	Accuracy (%)
Baseline	-	96-256-384-384-256-1024-4096-4096 (2.43e7)	84.76
Sparsification	stage 1	96-256-384-384-256-1024-4096-4096 (2.43e7)	83.96
	stage 2	89-161-131-91-57-92-299-3486 (1.58e6)	82.32
	stage 3	89-161-131-91-57-92-299-3486 (1.58e6)	<b>85.45</b>
Expansion	stage 1	48-63-128-120-120-120-120-1500 (5.78e5)	83.03
	stage 2	96-138-169-179-195-195-195-1335 (1.23e6)	78.81
	stage 3	96-138-169-179-195-195-195-1335 ( <b>1.23e6</b> )	85.42

Table 3: The network sparsification and expansion with AlexNet on CIFAR-100. The architecture and accuracy at the end of each stage are shown in the table.

	Stage	Architecture (Number of Parameters)	Accuracy (%)
Baseline	-	96-256-384-384-256-1024-4096-4096 (2.43e7)	56.1
Sparsification	stage 1	96-256-384-384-256-1024-4096-4096 (2.43e7)	53.03
	stage 2	92-196-215-196-126-332-130-3013 (1.61e6)	53.66
	stage 3	92-196-215-196-126-332-130-3013 (1.61e6)	<b>57.74</b>
Expansion	stage 1	48-63-128-120-120-120-120-1500 (5.78e5)	39.18
	stage 2	94-109-192-217-230-246-246-1314 (1.50e6)	48.10
	stage 3	94-109-192-217-230-246-246-1314 ( <b>1.50e6</b> )	56.30

with similar classification accuracies at the end of the training even though they are started from two significant different network architectures. Interestingly, on the CIFAR datasets both sparsification and expansion reach accuracies that are better than the baseline models, even though the final architectures are less than 10% of the baseline architectures.

## 7 Conclusion

We propose Neural Plasticity Networks (NPNs) for network sparsification and network expansion by attaching each unit of a network with a stochastic binary gate, whose parameters are jointly optimized with original network parameters. The activation or deactivation of a unit is completely data-driven and determined by an  $L_0$ -regularized objective. Our NPN unifies dropout (when  $k = 0$ ), traditional training of DNNs (when  $k = \infty$ ) and interpolate between these two. To the best of our knowledge, it is the first learning framework that unifies network sparsification and network expansion in an end-to-end training pipeline that supports pre-training, sparsification/expansion, and fine-tuning seamlessly. Along the way, we also give a new interpretation of dropout. Extensive experiments on multiple public datasets and multiple network architectures validate the effectiveness of NPNs for network sparsification and expansion in terms of model compactness and predictive accuracies.

As for future extensions, we plan to design better (possibly non-antithetic) parametric function  $g(\phi)$  to improve the compactness of learned networks. We also plan to extend the framework to prune or expand layers to further improve model compactness and accuracy altogether.

## References

- [1] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. Imagenet: A large-scale hierarchical image database. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2009.
- [2] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.
- [3] David Silver, Aja Huang, Christopher J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik

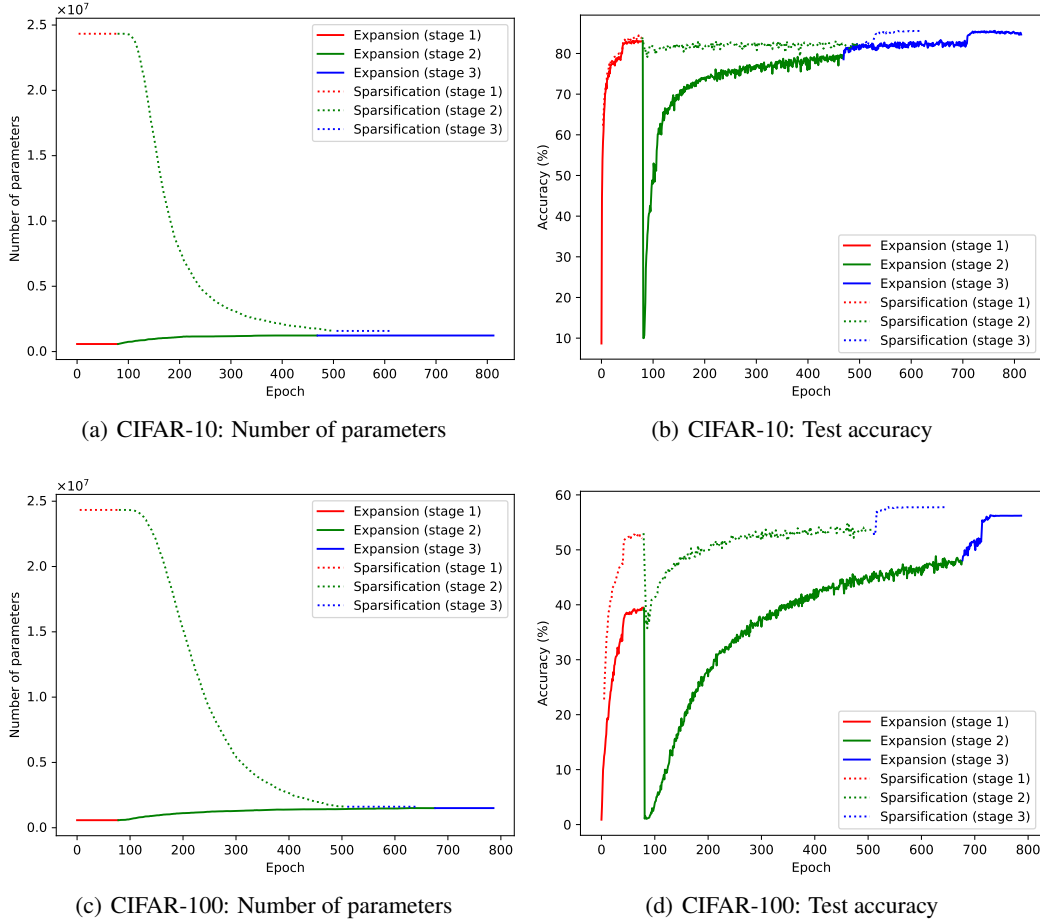


Figure 5: The evolution of network capacity and test accuracy as a function of epoch for NPN network sparsification and expansion with AlexNet on the CIFAR-10/100 dataset.

- Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–503, 2016.
- [4] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.
  - [5] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. Densely connected convolutional networks. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.
  - [6] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. In *The British Machine Vision Conference (BMVC)*, 2016.
  - [7] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *Advances in neural information processing systems*, pages 1135–1143, 2015.
  - [8] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. In *International Conference on Learning Representations (ICLR)*, 2016.
  - [9] Kirill Neklyudov, Dmitry Molchanov, Arsenii Ashukha, and Dmitry Vetrov. Structured bayesian pruning via log-normal multiplicative noise. In *Advances in Neural Information Processing Systems (NIPS)*, 2017.
  - [10] Christos Louizos, Max Welling, and Diederik P. Kingma. Learning sparse neural networks through  $l_0$  regularization. In *International Conference on Learning Representations (ICLR)*, 2018.
  - [11] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Learning structured sparsity in deep neural networks. In *Advances in Neural Information Processing Systems (NIPS)*, 2016.
  - [12] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. *arXiv preprint arXiv:1608.08710*, 2016.

- [13] Christos Louizos, Karen Ullrich, and Max Welling. Bayesian compression for deep learning. In *Advances in Neural Information Processing Systems*, pages 3288–3298, 2017.
- [14] Dmitry Molchanov, Arsenii Ashukha, and Dmitry Vetrov. Variational dropout sparsifies deep neural networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 2498–2507. JMLR. org, 2017.
- [15] Sandrine Thuret. You can grow new brain cells. here’s how. [https://www.youtube.com/watch?v=B\\_tjKYvEziI](https://www.youtube.com/watch?v=B_tjKYvEziI), 2015.
- [16] Yang Li and Shihao Ji. L0-ARM: Network sparsification via stochastic binary optimization. In *The European Conference on Machine Learning (ECML)*, 2019.
- [17] Mingzhang Yin and Mingyuan Zhou. Arm: Augment-REINFORCE-merge gradient for stochastic binary networks. In *International Conference on Learning Representations (ICLR)*, 2019.
- [18] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014.
- [19] Hirotugu Akaike. *Selected Papers of Hirotugu Akaike*, pages 199–213. Springer, 1998.
- [20] Gideon Schwarz. Estimating the dimension of a model. *The Annals of Statistics*, 6:461–464, 1978.
- [21] Thomas Bird, Julius Kunze, and David Barber. Stochastic variational optimization. *arXiv preprint arXiv:1809.04855*, 2018.
- [22] Yoshua Bengio, Nicholas Leonard, and Aaron Courville. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432*, 2013.
- [23] Juho Lee, Saehoon Kim, Jaehong Yoon, Hae Beom Lee, Eunho Yang, and Sung Ju Hwang. Adaptive network sparsification with dependent variational beta-bernoulli dropout. *arXiv preprint arXiv:1805.10896*, 2018.
- [24] Yann LeCun, John S Denker, and Sara A Solla. Optimal brain damage. In *Advances in neural information processing systems*, pages 598–605, 1990.
- [25] Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning. In *International Conference on Learning Representations (ICLR)*, 2017.
- [26] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. Learning transferable architectures for scalable image recognition. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.
- [27] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V. Le. Aging evolution for image classifier architecture search. In *AAAI*, 2019.
- [28] T. Ash. Dynamic node creation in backpropagation networks. *Connection Science*, 1:365–375, 1989.
- [29] Guanyu Zhou, Kihyuk Sohn, and Honglak Lee. Online incremental feature learning with denoising autoencoders. In *International Conference on Artificial Intelligence and Statistics (AISTats)*, page 1453–1461, 2012.
- [30] Yu-Xiong Wang, Deva Ramanan, and Martial Hebert. Growing a brain: Fine-tuning by increasing model capacity. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.
- [31] Corinna Cortes, Xavier Gonzalvo, Vitaly Kuznetsov, Mehryar Mohri, and Scott Yang. AdaNet: Adaptive structural learning of artificial neural networks. In *International Conference on Machine Learning (ICML)*, 2017.
- [32] Takeru Miyato, Shin-ichi Maeda, Shin Ishii, and Masanori Koyama. Virtual adversarial training: a regularization method for supervised and semi-supervised learning. *IEEE transactions on pattern analysis and machine intelligence*, 2018.
- [33] Yann Lecun, Leon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, pages 2278–2324, 1998.
- [34] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems (NIPS)*, pages 1097–1105, 2012.
- [35] Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, 2009.
- [36] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations (ICLR)*, 2015.