ON THE EFFICIENCY OF DEEP NEURAL NETWORKS

Anonymous authors
Paper under double-blind review

ABSTRACT

The efficiency of neural networks is essential in large-scale deployment scenarios such as mobile applications, internet of things, and edge computing. For a given performance requirement, an efficient neural network should use the simplest network architecture with a minimal number of parameters and connections. **In this paper, we introduce a framework to analyze and obtain efficient neural networks.** In summary, our main contributions are three-fold. **Our first contribution is the subnetwork hypothesis to address overfitting issues and help explain the effectiveness of several key techniques in training efficient networks:** 1) softmax normalization in output layers may be one major cause of overparameterization; 2) using log likelihood ratio representation in output layers can reduce overfitting; 3) weight decaying and structural regularization can also effectively reduce overfitting. The second contribution is **a simple and effective snapshot-based procedure to prune a well-trained network that minimizes overfitting** – pruning unimportant weights and connections first, and simply adjust remaining non-weight parameters using the backpropagation algorithm. Besides, the snapshot-based pruning method can also be used to evaluate the efficiency of trained networks. Finally, we hypothesize that there exist lower bounds of the total number of bits for representing parameters and connections regarding performance metrics for a given optimization problem. Rather than focusing on improving the sole accuracy metric with more complex network architectures, it is also important to explore the trade-offs between accuracy and the total number of representation bits, when comparing different network architectures and implementations.

1 INTRODUCTION

Deep learning has achieved tremendous success in large-scale machine learning systems, such as big-data analytics (Najafabadi et al., 2015), billion-parameter generative models for natural language processing (Brown et al., 2020; Radford et al., 2019), and computer vision for self-driving cars (Grigorescu et al., 2020). A general trend for recent success is the use of neural networks of ever-increasing model sizes and their exponentially increasing computation power requirement. Training these gigantic neural network models require tens of thousands parallel computing units inside dedicated computer clusters with extremely high transient storage capacity and data synchronization bandwidth. Consequently, some of the state-of-the-art models are only accessible to very few researchers in the machine learning community.

On the other hand, large-scale deployment of machine learning applications in low-power scenarios, such as mobile applications, internet-of-things (IoT), and edge computing, has put more stringent requirements on the efficiency of neural network models. For a given problem and performance metrics, efficient neural network models should have a minimal number of weights and connections, simple network topology and architecture suitable for low-power computing devices, and low data bandwidth and transient storage requirements. It is important to investigate the model efficiency problem to bridge the performance gap between petascale high-end models and low-power neural architectures for large-scale deployment. However, methods and principles for obtaining efficient deep neural networks have not yet been thoroughly studied.

In this paper, we introduce a framework to analyze and obtain efficient deep neural networks. Especially, we identify several key issues in training efficient deep neural networks and propose a new model compression procedure to prune redundant weights and connections. One important insight of our study is the high correlation between overfitting and model efficiency. Overfitting may improve training accuracy, but it can cause overparameterization. In Section 2, we show that softmax output layers can introduce non-deterministic effects to the backpropagation algorithm, yielding redundant subnetworks with exploding numbers of parameters. To solve this problem, we propose the log likelihood ratio (LLR) representation for output layers. We also investigate potential mechanisms for weight decaying and structural regularization to reduce overfitting. Furthermore, we propose a simple and effective snapshot-based pruning procedure to obtain efficient deep neural networks. We empirically validate this novel approach in Section 3 using various deep learning architectures including LeNet, ResNet, and DenseNet, on the MNIST, CIFAR-10, and CIFAR-100 datasets. Based on the empirical results, we further discuss the model efficiency regarding information cost of model representation. Section 4 reviews prior work in regularization, overfitting, and model compression, followed by Section 5 that concludes the paper.

2 Efficient Neural Networks

A fundamental assumption of our analysis is that a complex neural network can be decomposed into subnetworks that are responsible for different operation modes. In other words, the complex non-linear function of a neural network can be decomposed into groups of sub-functions. Each group of sub-functions represents one mode of operation. In this way, the efficiency of a neural network highly depends on the composition and correlation between these groups of sub-functions. Thus, overfitting may be viewed as forming redundant subnetworks that reduce the efficiency of trained networks.

In this section, we shows that a critical step for obtaining efficient neural networks is to eliminate redundant subnetworks by minimizing overfitting. We first analyze the overfitting issues caused by redundant subnetworks and describe potential mitigating mechanisms. Several hypotheses presented in this section will also be empirically validated using experiments in Section 3. Finally, we introduce a novel snapshot-based procedure to obtain efficient deep neural networks by pruning their unimportant weights and connections. This procedure is also used to analyze the efficiency of trained networks.

2.1 Softmax Normalization

The softmax function, a.k.a. softargmax, is a normalization function often used as the last activation function of a neural network (Bishop, 2006). Let $\boldsymbol{Z} = \{z_0, z_1, \cdots, z_i, \cdots\}$ represents the input vector, the softmax output vector $\boldsymbol{Q} = \{q_0, q_1, \cdots, q_i, \cdots\}$ is defined as

$$q_i = \frac{\exp(z_i)}{\sum_k \exp(z_k)} \tag{1}$$

where $q_i \in [0, 1]$ and $\sum_i q_i = 1$. Thus, the normalized output vector can be interpreted as marginal probabilities. The softmax output can be naturally combined with the cross entropy function $J = -\sum_i p_i \log q_i$, where p_i is the target probability. The derivative of J with respect to z_i takes a simple form of $q_i - p_i$ (Goodfellow et al., 2016). The simple probabilistic interpretation and derivative computation make the combination of softmax normalization and cross entropy loss a pervasive choice for multinomial classification problems. However, potential issues using softmax normalization with the backpropagation (BP) algorithm has not been fully investigated.

Suppose a neural network $\boldsymbol{G}$ can be decomposed into two or more smaller subnetworks $\boldsymbol{G} = \{\boldsymbol{G}_0, \boldsymbol{G}_1, \cdots, \boldsymbol{G}_m, \cdots\}$ with the same feature input $\boldsymbol{X}$. The final activation $\boldsymbol{Z}$ is the superposition of the subnetwork activation before the softmax normalization in the output layer

$$\boldsymbol{Z} = \sum_{m=0}^{M} \boldsymbol{Y}_m = \sum_{m=0}^{M} \boldsymbol{f}_m(\boldsymbol{X}) \tag{2}$$

where $\boldsymbol{f}_m$ is the non-linear function representing subnetwork $\boldsymbol{G}_m$. The decomposition is done according to the final activation without considering intermediate hidden layers. The softmax normalization operation has the following properties regarding the relationship between subnetwork activations (see Appendix A).

1. If the subnetwork activations are linear offset versions of each other, such that $\boldsymbol{Y}_0 = \boldsymbol{Y}_1 - \beta_1 \cdots = \boldsymbol{Y}_m - \beta_m \cdots$, the normalization result of the whole network is equivalent to applying the softmax function to the activation of any subnetwork scaled by M: $\mathbf{Q} =$ **softmax**$(M\boldsymbol{Y}_m)$. Note that the offset between subnetwork activation Y_m has no impact on the softmax output. **If the activations $\mathbf{Y}_m$ are linearly semi-correlated, the generalized softmax property is applicable, i.e., that $\mathbf{Q} \approx$ softmax$(M\boldsymbol{Y}_m)$.**
2. If the subnetwork activations are scaled versions of each other, such that $\boldsymbol{Y}_0 = \alpha_1 \boldsymbol{Y}_1 \cdots = \alpha_k \boldsymbol{Y}_k \cdots$ and $1 \geq \alpha_1 \geq \alpha_2 \geq \cdots \geq \alpha_k \cdots$, the normalization operation is equivalent to applying the softmax function to the scaled principal subnetwork: $\mathbf{Q} =$ **softmax**$(S\boldsymbol{Y}_0)$, where $S = 1 + \alpha_1 + \alpha_2 + \cdots$. The softmax normalization allows proportional integration of information. A single subnetwork that has very strong activation (higher prediction probabilities) can dominate over other subnetworks with weak activations. If there are no dominant subnetworks, the total number of contributing subnetworks may be large and the whole network tends to be overparameterized.

In short, the softmax function can act as a super combinator for different modes of the neural network, summing and amplifying weak subnetwork activations. This could partially explain why deep neural networks are so expressive that they are suitable for diverse types of problems. However, when there are redundant subnetworks that produce linearly correlated activations, the softmax normalization function make them indistinguishable from each other. The linearly correlated subnetworks potentially lead to overfitting and overparameterization. We have the following hypothesis regarding the effects of such redundant subnetworks:

Hypothesis 1: *For deep neural networks, the existence of redundant subnetworks combining with softmax normalization can lead to overfitting and overparameterization when training with the back-propagation algorithm.*

The derivative of the cross entropy loss is linear with regard to the softmax output $\boldsymbol{Q}$ and target $\boldsymbol{P}$, and softmax normalization makes it impossible to differentiate between the effects of different subnetworks, the BP algorithm thus will fine-tune all the parameters without penalizing any individual subnetwork. **Therefore, the initialization of weights may create redundant subnetworks that have non-deterministic effects on the training process. For example, Mishkin & Matas (2016) demonstrated that initialization of weights can affect test accuracy.** Such behaviors and the existence of redundant subnetworks will be validated from empirical results in Section 3.

2.2 LLR Representation

The softmax normalization is mainly used to convert neural network outputs to probabilities. However, the softmax normalization allows linearly correlated subnetwork activations and potentially introduces overfitting. Therefore, it is desirable to avoid softmax normalization in output layers. It turns out that using the log likelihood ratio (LLR) representation in output layers can avoid normalization and overfitting issues. Given a binary random variable X and $P_1(X) = \{\text{probability X is true}\}$, the LLR for X can be defined as

$$LLR(X) = \log \frac{P_1(X)}{1 - P_1(X)} \tag{3}$$

Since neural networks can model arbitrary non-linear functions, we can adopt LLR representation for each component of the outputs $\boldsymbol{Y}$ and target labels $\boldsymbol{T}$. For both multi-class and multi-label classification, the problem can be regarded as multiple binary regression problems adopting the LLR representation for each class. Therefore, output normalization across different classes is not needed, but loss functions need to be changed accordingly – we introduce the bipolar softplus (BSP) loss function as defined in Appendix B.1. We demonstrate that the LLR representation combined with the BSP loss function does not need normalization and avoids the introduction of redundant subnetworks. The choice of loss functions is not mandatory and there may be better alternative loss functions. The optimization of loss functions will be addressed in future study. In this paper, we use empirical results to demonstrate the effectiveness and behavior of this novel scheme. We introduce the following hypothesis regarding the LLR representation:

Hypothesis 2: *For classification problems with deep neural networks, using the LLR representation for the output layer and the target labels can reduce overfitting and avoid overparameterization compared with softmax normalization.*

It is worth emphasizing that the LLR representation has clear physical meanings, which could help the explainability of neural networks. LLR values are symmetrical and centered around zero, which can be regarded as a natural normalization point. Note that LLR values have range $(-\infty, +\infty)$ and a large magnitude means higher confidence in prediction. Thus, by controlling the LLR magnitude of the target labels, we can introduce regularization to network outputs.

2.3 WEIGHT DECAYING

In previous discussion, potential issues in normalization and representation are analyzed by decomposing the activation in the output layer. In a similar fashion, we can also decompose the weights and activation in each hidden layer as follows.

Suppose feature inputs of a layer can be represented as $\boldsymbol{X} = \sum_{m=0}^{M-1} \boldsymbol{X}_m$, and its weight matrix $\boldsymbol{W}$ can be decomposed as $\boldsymbol{W} = \sum_{n=0}^{N} \boldsymbol{W}_n$, where $\boldsymbol{W}_n$ is non-zero weight components, then the activation $\boldsymbol{Z}$ can be decomposed as

$$\boldsymbol{Z} = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} \boldsymbol{A}_{m,n} = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} (\boldsymbol{X}_m \boldsymbol{W}_n + \boldsymbol{B}) \tag{4}$$

where $\boldsymbol{B}$ is the bias vector. When the rectified linear unit (ReLU) non-linear function is adopted, only those activations larger than zero in Eq. 4 are effective. For a given feature input $\boldsymbol{X}_k$, if there are multiple $\boldsymbol{A}_{k,n}$ components that have all positive elements, the weight components $\boldsymbol{W}_n$ can be effectively combined to reduce the total number of parameters. The redundant weights components may be different for different input features. The existence of such redundant weight components may become the source of overfitting and overparameterization. The large ones of these redundant $\boldsymbol{A}_{k,n}$ components also tend to be working in the linear regime of the ReLU function, which effectively reduces the non-linear behavior of the network. To reduce overfitting and redundancy, the weights should have relatively small magnitudes working in the non-linear regime of the ReLU function, hence we have the following hypothesis regarding weight decaying (L2 regularization).

Hypothesis 3: *Limiting the magnitude of weights using weight decaying can reduce overfitting and overparameterization in deep neural networks when the ReLU activation function is used.*

This hypothesis could explain why regularizing weights is an effective technique to improve training performance. Weight decaying should also be separated from loss regularization, which was first discussed in Loshchilov & Hutter (2018). In our experiments, however, their AdamW algorithm turns out to improve the training accuracy by increasing overfitting and overparameterization **as shown in Section 3.2**.

2.4 STRUCTURAL REGULARIZATION

The underlying assumption in the analysis of weight decaying is that the outputs of fully-connected subnetworks can be freely superimposed with each other. Thus, if the combination of subnetworks are restricted, overfitting issues could be mitigated. A common technique for this purpose is structural restriction of the subnetworks. Some examples are listed in the following.

1. Structural pruning - Various techniques to selectively remove connections from the whole network in training have been proposed and shown to reduce overfitting, such as Wan et al. (2013). In a sense, stochastic gradient descent (SGD) can also be regarded as adopting random structural pruning.
2. Weight sharing - By sharing weights and forcing regular network structures, neural networks become more effective and easier to train. Convolutional neural network (CNN) can be regarded as a prominent type which is often used as feature extraction layers.
3. Micro-architectural design - By adopting certain topology patterns between or within neural network layers, the resulting networks are confined to subsets of fully-connected networks, hence their overfitting issues are mitigated. Skip connections, for example, have been show to improve training speed and performance (He et al., 2016; Huang et al., 2017).

Many existing optimization techniques for training neural networks could be partially explained and further analyzed using the subnetwork analysis. The underlying principle is that by reducing the initial functional space, the optimization problem becomes less difficult and easier to converge, which explains why micro-architecture design can have significant impact on the performance of neural networks. **In Section 3, the effects of structural regularization are partially demonstrated by comparing the efficiency of different network architectures on the same dataset.**

2.5 Snapshot-based Pruning

It is well known that neural networks can be made more efficient in terms of computation and storage requirements by pruning some of the unimportant weights. For deep neural networks, the iterative pruning and retraining procedure in Han et al. (2015) has been used for generating efficient neural networks for low-power applications. However, the iterative procedure requires extra computing power and processing time. **Furthermore, the iterative procedure often requires manually fine-tuning pruning thresholds.** We discuss two important aspects of pruning neural networks in the following.

1. ***Important weights*** - Deciding which weights are important is the first key issue. In general, weights with smaller magnitudes are considered unimportant and can be pruned, but this may not always be the case for different types of components in various network architectures. For example, shared weights in convolutional layers may be more important than weights in fully connected layers. Even the importance of weights in the same layer may not be correlated with their magnitudes.
2. ***Retrain requirement*** - After pruning its weights and connections, a pruned neural network usually needs to be adjusted. It is not clear which aspects of the network need to be modified. For the iterative pruning and retrain process, the weights and biases between initial and final iterations may be completely different so that it is hard to analyze the iterative retraining mechanism.

By analyzing experimental data from extensive empirical studies with different datasets and various architectures, we have the following observation regarding these two key aspects for pruning neural networks.

1. ***Weight distribution*** - If the neural network is well trained such that overfitting is minimized, the weight magnitude distribution correlates better with the weights' importance. In other words, weights with smaller magnitudes around zero can be effectively pruned. Different layers and types of components may need different pruning thresholds but they can be easily adjusted using macro network attributes.
2. ***Essential network*** - The important weights together with their corresponding connections define the essence of a trained network; therefore, they should be kept unchanged as a snapshot. Only the biases need to be adjusted. Other non-weight parameters, such as batch normalization parameters, may also need to be adjusted as well.

In short, iterative pruning and retraining may not be necessary when a neural network is well-trained. The first step in obtaining efficient neural networks is to adopt efficient training techniques, such as LLR representation, weight decaying, and structural regularization. After pruning unimportant weights and connections from trained snapshots, the next step is to simply adjust remaining non-weight parameters using the BP algorithm. For most architectures, because most of the parameters are connection weights, adjusting the non-weight parameters requires much fewer iterations than the initial training process – usually only a few epochs are enough.

Conversely, the efficiency and quality of a trained network can be evaluated with the effectiveness of parameter pruning. Refinement of optimization algorithms may also be further examined using this pruning procedure. **If a network is overparameterized, the performance of its pruned versions deteriorates dramatically as the total number of parameters is reduced. The key discovery here is the high correlation between overfitting and the efficiency of neural networks.**

If trained neural networks are not efficient enough initially, combining iterative techniques with the proposed snapshot-based pruning method could be beneficial. **For very large networks, it should be noted that using all the methods analyzed in this section may not be enough to yield efficient deep neural networks using a single-shot training procedure.**

3 Experiments

We empirically analyze the model efficiency trade-offs in deep neural networks as well as the overfitting issues in training neural networks to validate the subnetwork assumption and the analysis of various mitigating methods in previous section. We also demonstrate the effectiveness of the proposed snapshot-based pruning procedure in obtaining and evaluating efficient neural networks.

3.1 Methodology

The trade-offs between accuracy and total number of parameters are analyzed with various architectures and datasets using the following procedure: **1) the neural network is first trained using different hyper-parameter settings and output representation; 2) the trained networks are pruned using different threshold settings, and non-weight parameters are retrained using the BP algorithm; 3) test accuracy and total number of parameters of the pruned networks are averaged over at least 10 different experiment runs using the same pruning settings.**

The pruning thresholds are set according to the standard deviation of weights' magnitudes. **Two different thresholds are used for convolutional layers and linear layers, respectively. For example, the threshold value for convolutional layers is calculated by multiplying the pruning setting with the standard deviation of weights in all convolutional layers. Weights with magnitudes lower than the threshold value are pruned. In all cases, simple linearly-spaced pruning settings are used without further fine-tuning. However, optimization of the pruning settings is possible by taking into account the structural attributes of given network architectures.**

Other hyper-parameter settings and detailed analysis of the results are included in Appendix C. In the following, we focus on the efficiency trade-offs using different architectures on several datasets.

3.2 MNIST Classification

We first conduct experiments on the MNIST dataset (LeCun et al., 1998) using the LeNet-300-100 and LeNet-5 architectures (LeCun et al., 2015). In Figure 1 (left), the distribution of weights when training with LLR representation is compared with the case of softmax normalization. Using LLR representation yields a better distribution of weights – the probability of small weights around zero is higher and these weights can be pruned with less impact on the performance. Furthermore, weight decaying can push the weights aggressively towards zero as shown in Figure 1 (right). **While softmax normalization primarily affects output layers, the ReLU function may cause overfitting in all layers. Therefore, the effect of weight decaying is more prominent and effective as shown in Figure 1**. This observation is consistent with the analysis and hypotheses from Section 2.

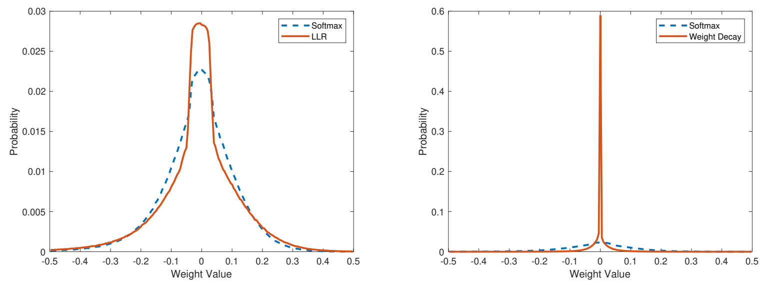


Figure 1: Effects of output representation and weight regularization on the distribution of weights using (left) LLR representation and (right) weight decaying on the MNIST dataset.

Figure 2 shows the trade-off curves for test errors vs. total number of effective parameters for all experiment results on the MNIST dataset. Each curve represents 20 trained networks with the same training settings, each point represents the average total number of weights and average top-1 errors of the pruned networks for each of the 10 different pruning settings. We can see that using LLR representation instead of softmax normalization can reduce the total number of parameters for the same accuracy requirement. Using weight decaying also significantly improve the efficiency of the trained networks. Using both methods yields the most efficient neural networks with better performance than the ones using the iterative pruning approach from Han et al. (2015), as shown in Table 1-2 in Appendix C.1. Compared with fully-connected networks, convolutional neural networks show better performance partially due to inherent structural regularization.

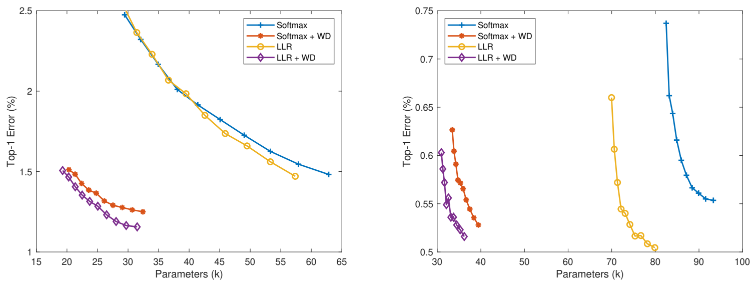


Figure 2: Test errors vs. total parameters trade-offs on the MNIST dataset using softmax normalization and LLR representation: (left) LeNet-300-100 and (right) LeNet5

We found that the AdamW optimizer with weight decaying may increase training accuracy by increasing overfitting and yield less efficient networks, as demonstrated in Figure 3. Compared with previous results, the optimal pruned model sizes are dramatically increased and using weight decaying does not improve the efficiency of trained networks.

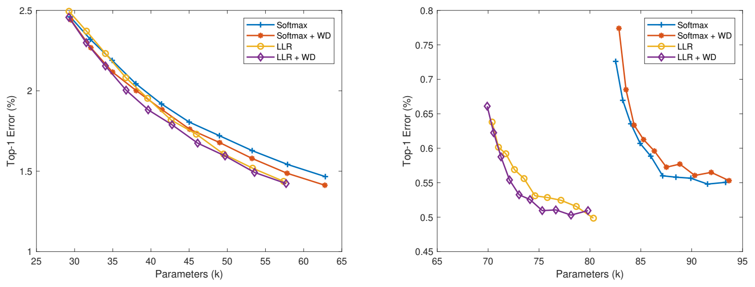


Figure 3: Test errors vs. total parameters trade-offs on the MNIST dataset using AdamW optimization with (left) LeNet-300-100 and (right) LeNet5 architectures

3.3 CIFAR-10 Classification

Several ResNet (He et al., 2016) and DenseNet (Huang et al., 2017) architectures are used for experiments on the CIFAR-10 dataset. We compare ResNet with 20/32/56 layers to DenseNet with 40/60/100 layers and a growth rate $k = 12$. To reduce the total number of parameters, bottleneck layers are enabled for DenseNet. **Further comparison and analysis of the overfitting issues are provided in Appendix C.2.**

Figure 4 summarizes the experiment results on the CIFAR-10 dataset. **A weight-decaying setting of $1e^{-4}$ is used with softmax normalization, which is the default setting to obtain the best accuracy results without pruning. For comparison purpose, experiments for LLR representation use a weight-decaying setting of $5e^{-4}$. Although using softmax yield the best training accuracy, the trained networks are overparameterized compared with those using LLR representation.**

Therefore, the trade-off curves can be used to judge the efficiency of trained networks. Curves closer to the bottom-left region on the figure represent more efficient networks. Both ResNet and DenseNet architectures show similar trends in terms of efficiency. **The trade-off curves for the same architecture with different initial model sizes seem to be bounded by a single theoretical curve.** In the energy efficient region with small number of parameters, the error rate goes down rapidly with a small increase in the number of parameters; while in the high accuracy region, small increases in accuracy require exponentially increasing numbers of parameters. In this regard, the ResNet-32 and the DenseNet-60 architectures may offer better alternative trade-offs in efficiency.

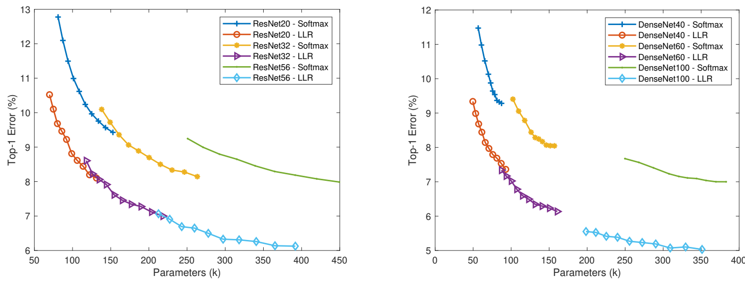


Figure 4: Test errors vs. total parameters trade-offs on the CIFAR-10 dataset using softmax normalization and LLR representation: (left) ResNet and (right) DenseNet

3.4 CIFAR-100 Classification

Figure 5 summarizes the experiment results using different ResNet and DenseNet architectures on the CIFAR-100 dataset. **For all experiments using either softmax normalization or LLR representation, the same weight-decaying settings are used.** In terms of efficiency trade-offs, we see a similar trend as before: linear increase in accuracy tends to require exponential increase in network capacity. **We notice that the difference between softmax normalization and LLR representation is less prominent for the DenseNet architecture. One possible reason is that the BSP loss function is not yet fully optimized for the DenseNet architecture. Another possible reason is that the effects of weight decaying are more prominent than softmax normalization for the DenseNet architecture with larger initial model sizes.**

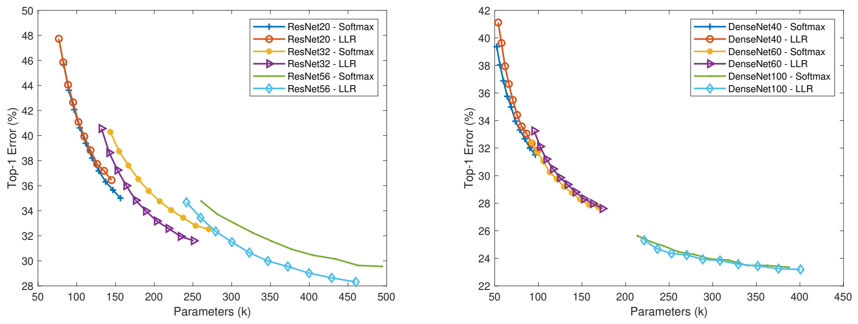


Figure 5: Test errors vs. total parameters trade-offs on the CIFAR-100 dataset using softmax normalization and LLR representation: (left) ResNet and (right) DenseNet

4 Related Work

Historically, deep neural networks using sigmoid or hyperbolic tangent activation functions were difficult to train using backpropagation (Rumerlhar, 1986) due to the vanishing gradient problem (Glorot & Bengio, 2010). The introduction of ReLU activation function (Nair & Hinton, 2010) greatly improves training speed for deep learning, yielding improved prediction accuracy in many new applications. However, using the ReLU activation function also tends to introduce overfitting issues as shown in this paper.

Regularization using modified loss functions can alleviate overfitting but with limited effects. Data augmentation is another method to reduce overfitting and improve generalization performance. Dropout, i.e., randomly selected units are dropped during training, was introduced in Hinton et al. (2012) and Srivastava et al. (2014) as an effective method to prevent overfitting. This idea was extended to randomly dropping connections in Wan et al. (2013). Batch normalization is another method to reduce overfitting and improve training speed. Nevertheless, it is not completely clear how in principle these methods work, and they still can not fully eliminate overfitting issues in deep neural networks.

Overfitting is also related to the size of a neural network. Excessively large networks tend to introduce overfitting, and vice versa. It is also desirable to minimize model sizes for processing speed and systematic scaling purposes. A straightforward way for compressing over-parameterized neural networks is to prune trivial weights and retain only important connections, which is similar to the development of mammalian brain (Rauschecker, 1984). Pruning unimportant weights and connections after training is a common way to obtain efficient neural networks. Early work in Hassibi & Stork (1993); Hassibi et al. (1994); LeCun et al. (1989) uses the statistics from backpropagation to trim trained networks.

Recently, Han et al. (2015) proposed an iterative pruning and re-training procedure for efficient model compression. Similarly, pruning filters were proposed for convolutional networks in Li et al. (2016). However, iterative pruning and re-training is generally difficult, requiring extra processing time and resources. Furthermore, the iterative pruning process is opaque and requires try-and-error in selecting pruning thresholds for parameters in different layers. The lottery ticket hypothesis from Frankle & Carbin (2018) tries to explain why the iterative pruning procedure can work, but the empirical results therein are not conclusive enough.

Alternatively, one-shot pruning techniques try to train sparse neural networks directly without iterative operations (Lee et al., 2019; Zhang & Stadie, 2019; Wang et al., 2020). However, Liu et al. (2019) observe that previous state-of-the-art pruning techniques may not provide better performance compared with randomly initialized networks. Their observations could be partially explained using our subnetwork analysis on structural regularization effects.

5 Discussion and Future Work

In this paper, we identify several important issues affecting overfitting in training deep neural networks. The key finding is that reducing overfitting is critical for obtaining efficient neural networks. It is demonstrated with several datasets and network architectures that a simple snapshot-based pruning procedure can generate efficient deep neural networks. However, more empirical validation results using other neural network architectures and larger datasets are required to further validate the proposed approach. Quantizing the parameters will further compress neural network models, which is not considered here for brevity but could be a natural extension in future work.

The snapshot-based retrain method can also be useful in real-world applications, where we only need to store pruned weights and connections, while biases and other optimization parameters can be restored using new datasets. This could be a very important optimization in cloud and edge computing applications. For transfer learning, neural networks trained with old datasets may be effectively retrained using new datasets, given that the underlying neural models are similar in nature.

We further analyze the efficiency trade-offs in training deep neural networks. For a given optimization problem with given objective and dataset, we should consider structural information, in additional to weights, as representation cost of trained networks. For the small-scale network architectures used in this study, few extra parameters are needed to specify the network topology and connections. However, for large-scale networks, the parameters for describing the network topology and connections should also be included in the representation cost of models. When we compare neural network performance, domain-specific knowledge for designing network architectures should be considered as additional information. We hypothesize that there exist lower-bounds of total number of bits for representing parameters and connections with regard to given performance metrics for an optimization problem. Rather than focusing on improving the sole accuracy metric with more complex network architectures, we should also explore the trade-offs between accuracy and total number of representation bits when comparing different network architectures and implementations.

Several hypotheses regarding training efficient deep neural networks are put forward and empirically validated with experiments. Although rigorous proofs are not provided, we hope that they will encourage further discussion and research efforts on the trade-offs between model performance and complexity.

6 Reproducibility Statement

The authors of this paper regard it critical to ensure all empirical results in this paper can be consistently reproduced. For each experiment case with different parameters and optimization settings, the results are generated with at least 10 runs with different random seed initialization. We also cross-check our results with different references. Furthermore, for some of the experiments, we have verified the results using several machine learning frameworks including PyTorch, TensorFlow, and Matlab Deep Learning Toolbox. Finally, we will publish the source codes for this work on GitHub and provide bug fixes and updates.

References

Christopher M Bishop. *Pattern recognition and machine learning*. springer, 2006.

Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. *arXiv preprint arXiv:1803.03635*, 2018.

Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pp. 249–256. JMLR Workshop and Conference Proceedings, 2010.

Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep learning*, volume 1. MIT press Cambridge, 2016.

Sorin Grigorescu, Bogdan Trasnea, Tiberiu Cocias, and Gigel Macesanu. A survey of deep learning techniques for autonomous driving. *Journal of Field Robotics*, 37(3):362–386, 2020.

Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. *Advances in Neural Information Processing Systems*, 28, 2015.

Babak Hassibi and David G Stork. *Second order derivatives for network pruning: Optimal brain surgeon*. Morgan Kaufmann, 1993.

Babak Hassibi, David G Stork, Gregory Wolff, and Takahiro Watanabe. Optimal brain surgeon: Extensions and performance comparison. *Advances in Neural Information Processing Systems*, 1994.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

Geoffrey E Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012.

Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 4700–4708, 2017.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. Technical report, University of Toronto, 2009.

Yann LeCun, John S Denker, Sara A Solla, Richard E Howard, and Lawrence D Jackel. Optimal brain damage. In *NIPs*, volume 2, pp. 598–605. Citeseer, 1989.

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

Yann LeCun et al. Lenet-5, convolutional neural networks. *URL: http://yann. lecun. com/exdb/lenet*, 20(5):14, 2015.

Namhoon Lee, Thalaiyasingam Ajanthan, and Philip Torr. SNIP: SINGLE-SHOT NETWORK PRUNING BASED ON CONNECTION SENSITIVITY. In *International Conference on Learning Representations*, 2019.

Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. *arXiv preprint arXiv:1608.08710*, 2016.

Zhuang Liu, Mingjie Sun, Tinghui Zhou, Gao Huang, and Trevor Darrell. Rethinking the value of network pruning. In *International Conference on Learning Representations*, 2019.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *International Conference on Learning Representations*, 2018.

Dmytro Mishkin and Jiri Matas. All you need is a good init. *arXiv preprint arXiv:1511.06422*, 2016.

Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Icml*, 2010.

Maryam M Najafabadi, Flavio Villanustre, Taghi M Khoshgoftaar, Naeem Seliya, Randall Wald, and Edin Muharemagic. Deep learning applications and challenges in big data analytics. *Journal of Big Data*, 2(1):1–21, 2015.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

JP Rauschecker. Neuronal mechanisms of developmental plasticity in the cat's visual system. *Human neurobiology*, 3(2):109–114, 1984.

DE Rumerlhar. Learning representation by back-propagating errors. *Nature*, 323:533–536, 1986.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

Li Wan, Matthew Zeiler, Sixin Zhang, Yann Le Cun, and Rob Fergus. Regularization of neural networks using dropconnect. In *International conference on machine learning*, pp. 1058–1066. PMLR, 2013.

Chaoqi Wang, Guodong Zhang, and Roger Grosse. Picking winning tickets before training by preserving gradient flow. In *International Conference on Learning Representations*, 2020.

Matthew Shunshi Zhang and Bradly Stadie. One-shot pruning of recurrent neural networks by jacobian spectrum evaluation. *arXiv preprint arXiv:1912.00120*, 2019.

A THE SOFTMAX PROPERTIES

A.1 PROOF OF THE SOFTMAX PROPERTY

Given two linearly correlated vectors $\boldsymbol{Y}_0 = (y_{0,0}, y_{0,1}, \cdots, y_{0,K-1})$, $\boldsymbol{Y}_1 = (y_{1,0}, y_{1,1}, \cdots, y_{1,K-1})$ of length K such that $\boldsymbol{Y}_1 = \boldsymbol{Y}_0 + \beta_1$, where β_1 is a scalar, then each component of the softmax normalization vector $\boldsymbol{Q} = \textbf{softmax}(\boldsymbol{Y}_0 + \boldsymbol{Y}_1)$ can be calculated as

$$\begin{aligned} q_n &= \frac{\exp(y_{0,n} + y_{1,n})}{\sum_k \exp(y_{0,k} + y_{1,k})} \\ &= \frac{\exp(2y_{0,n} + \beta_1)}{\sum_k \exp(2y_{0,k} + \beta_1)} \\ &= \frac{\exp(\beta_1)\exp(2y_{0,n})}{\exp(\beta_1)\sum_k \exp(2y_{0,k})} \\ &= \frac{\exp(2y_{0,n})}{\sum_k \exp(2y_{0,k})} \end{aligned}$$

If we add a third vectors $\boldsymbol{Y}_2 = (y_{2,0}, y_{2,1}, \cdots, y_{2,K-1})$ such that $\boldsymbol{Y}_2 = \boldsymbol{Y}_0 + \beta_2$, then each component of the softmax normalization vector $\boldsymbol{Q} = \textbf{softmax}(\boldsymbol{Y}_0 + \boldsymbol{Y}_1 + \boldsymbol{Y}_2)$ can be calculated as

$$\begin{aligned} q_n &= \frac{\exp(2y_{0,n} + \beta_1 + y_{2,n})}{\sum_k \exp(2y_{0,k} + \beta_1 + y_{2,k})} \\ &= \frac{\exp(3y_{0,n} + \beta_1 + \beta_2)}{\sum_k \exp(3y_{0,k} + \beta_1 + \beta_2)} \\ &= \frac{\exp(\beta_1 + \beta_2)\exp(3y_{0,n})}{\exp(\beta_1 + \beta_2)\sum_k \exp(3y_{0,k})} \\ &= \frac{\exp(3y_{0,n})}{\sum_k \exp(3y_{0,k})} \end{aligned}$$

Using generalization for $\boldsymbol{Z} = \sum_0^{M-1} \boldsymbol{Y}_m$, where $\boldsymbol{Y}_m$ are linear offset versions of each other, such that $\boldsymbol{Y}_0 = \boldsymbol{Y}_1 - \beta_1 = \boldsymbol{Y}_2 - \beta_2 = \cdots = \boldsymbol{Y}_{M-1} - \beta_{M-1}$, we have

$$\textbf{softmax}(\boldsymbol{Z}) = \textbf{softmax}(M\boldsymbol{Y}_0) = \textbf{softmax}(M\boldsymbol{Y}_m)$$

A.2 PROOF OF THE GENERALIZED SOFTMAX PROPERTY

Given two vectors $\boldsymbol{Y}_0 = (y_{0,0}, y_{0,1}, \cdots, y_{0,K-1})$, $\boldsymbol{Y}_1 = (y_{1,0}, y_{1,1}, \cdots, y_{1,K-1})$ of length K such that $\boldsymbol{Y}_1 = \boldsymbol{Y}_0 + \boldsymbol{\beta}_1$, where $\boldsymbol{\beta}_1 = (\beta_{1,0}, \beta_{1,1}, \cdots, \beta_{1,K-1})$. Without any loss of generality, we assume that $Y_{0,0} \le Y_{0,1} \le \cdots \le Y_{0,K-1} = Y_{max}$. Define the maximal variation of $\boldsymbol{\beta}_1$ as δ_1 such that $|\beta_{1,k} - \beta_{1,n}| \le \delta_1$ for any $k, n \in \mathbb{Z}$ and $0 \le k, n \le K-1$. If δ_1 is insignificant relative to $\boldsymbol{Y}_0$, i.e., that

$$\exp(\delta_1) = o(\exp(Y_{max})), \tag{5}$$

where $o(\cdot)$ is the little-o notation, we define $\boldsymbol{Y}_0$, $\boldsymbol{Y}_1$ as linearly semi-correlated vectors. Then, each component of the softmax normalization vector $\boldsymbol{Q} = \textbf{softmax}(\boldsymbol{Y}_0 + \boldsymbol{Y}_1)$ can be calculated as

$$\begin{aligned} q_n &= \frac{\exp(y_{0,n} + y_{1,n})}{\sum_k \exp(y_{0,k} + y_{1,k})} \\ &= \frac{\exp(2y_{0,n} + \beta_{1,n})}{\sum_k \exp(2y_{0,k} + \beta_{1,k})} \\ &= \frac{\exp(\beta_{1,n})\exp(2y_{0,n})}{\exp(\beta_{1,n})\sum_k \exp(2y_{0,k} + \beta_{1,k} - \beta_{1,n})} \\ &= \frac{\exp(2y_{0,n})}{\sum_k \exp(2y_{0,k} + \beta_{1,k} - \beta_{1,n})}. \end{aligned}$$

Note that the denominator of q_n is mainly determined by the largest components of $\boldsymbol{Y}_0$, and thus we have the following approximation

$$\begin{aligned} q_n &\approx \frac{\exp(2y_{0,n})}{\sum_k \exp(2y_{0,k} + \delta_1)} \\ &\approx \frac{\exp(2y_{0,n})}{\sum_k \exp(2y_{0,k})} \end{aligned}$$

If we add a third linearly semi-correlated vectors $\boldsymbol{Y}_2 = (y_{2,0}, y_{2,1}, \cdots, y_{2,K-1})$ such that $\boldsymbol{Y}_2 = \boldsymbol{Y}_0 + \boldsymbol{\beta}_2$, where $\boldsymbol{\beta}_2 = (\beta_{2,0}, \beta_{2,1}, \cdots, \beta_{2,K-1})$ and $|\beta_{2,k} - \beta_{2,n}| \leq \delta_2$ for any $k, n \in \mathbb{Z}$ and $0 \leq k, n \leq K-1$. Then each component of the softmax normalization vector $\boldsymbol{Q} = \textbf{softmax}(\boldsymbol{Y}_0 + \boldsymbol{Y}_1 + \boldsymbol{Y}_2)$ can be calculated as

$$\begin{aligned} q_n &= \frac{\exp(2y_{0,n} + \beta_{1,n} + y_{2,n})}{\sum_k \exp(2y_{0,k} + \beta_{1,k} + y_{2,k})} \\ &= \frac{\exp(3y_{0,n} + \beta_{1,n} + \beta_{2,n})}{\sum_k \exp(3y_{0,k} + \beta_{1,k} + \beta_{2,k})} \\ &= \frac{\exp(\beta_{1,n} + \beta_{2,n}) \exp(3y_{0,n})}{\exp(\beta_{1,n} + \beta_{2,n}) \sum_k \exp(3y_{0,k} + \beta_{1,k} - \beta_{1,n} + \beta_{2,k} - \beta_{2,n})} \\ &\approx \frac{\exp(3y_{0,n})}{\sum_k \exp(3y_{0,k} + \delta_1 + \delta_2)} \\ &\approx \frac{\exp(3y_{0,n})}{\sum_k \exp(3y_{0,k})} \end{aligned}$$

Using generalization for $\boldsymbol{Z} = \sum_0^{M-1} \boldsymbol{Y}_m$, where $\boldsymbol{Y_m}$ are linearly semi-correlated of each other and $\boldsymbol{Y}_0 = \boldsymbol{Y}_1 - \boldsymbol{\beta}_1 = \boldsymbol{Y}_2 - \boldsymbol{\beta}_2 = \cdots = \boldsymbol{Y}_{M-1} - \boldsymbol{\beta}_{M-1}$, we have

$$\textbf{softmax}(\boldsymbol{Z}) \approx \textbf{softmax}(M\boldsymbol{Y}_0) \approx \textbf{softmax}(M\boldsymbol{Y}_m)$$

Note that the above relation holds as long as the variations in $\boldsymbol{\beta}_m$ is insignificant according to (5), while the magnitudes of $\boldsymbol{\beta}_m$ do not matter.

B Loss Functions for LLR representation

B.1 Bipolar SoftPlus Loss

Given a set of N neural network outputs $\{Y_n\}$ and corresponding targets $\{T_n\}$, the bipolar softplus (BSP) loss is defined as

$$\text{BSP}(Y, T) = \frac{1}{\beta N} \sum_{n=0}^{N-1} \log(1 + e^{-\beta \text{sgn}(T_n) Y_n}) \tag{6}$$

where β is a constant and $\text{sgn}(x)$ returns the sign of x as

$$\text{sgn}(x) = \begin{cases} 1, & x > 0 \\ 0, & x = 0 \\ -1, & x < 0 \end{cases} \tag{7}$$

C Experiment Settings and Detailed Results

C.1 MNIST Classification

The MNIST dataset of handwritten digits has a training set of 60,000 examples and a test set of 10,000 examples. Each image contains 28×28 monochrome pixels for one digit. The pixel values are converted to range $(0, 1)$ with dataset normalization.

Two architectures are used in the experiments: 1) the LeNet-300-100 is a three-layer fully connected network with 300 and 100 hidden nodes, 2) the LeNet-5 architecture has two convolutional layers with 20 and 50 filters and two fully connected layers with 800 and 500 hidden nodes.

Data augmentation is used to randomly shift each image horizontally and vertically by 0 or 1 pixel. The batch size for training is set to 128, and the Adam optimizer (Kingma & Ba, 2014) is used with default parameters: $\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999$,and $\varepsilon = 10^{-8}$. A weight decaying setting of 4e-4 is used for both the LeNet-300-100 and LeNet-5 architectures in corresponding cases. At least 20 runs with random seeds are carried out for each experiment case.

Table 1 summarizes the top-1 error rates, total number of parameters after pruning, and compression rate for different methods with the LetNet-300-100 architecture. The output layer uses a fixed pruning setting of 0.75, and the hidden layers use pruning settings $\theta_k = 1.0 + 0.05 \times k$, where $k = 0, 1, \cdots, 9$. The total numbers of pruned parameters and compression ratio in Table 1 are obtained using the **largest** pruning threshold.

Compared with LLR representation, results using softmax normalization have higher training errors and test errors. This indicates that LLR representation can mitigate the overfitting issues and improve accuracy in both training and testing. Figure 6 (left) also compares the effects of overfitting between softmax normalization and LLR representation with the LetNet-300-100 architecture. Using both LLR representation and weight decaying can yield more efficient networks than the iterative method from Han et al. (2015).

Table 1: Comparing MNIST performance and weight pruning for LeNet-300-100

	Top-1 Error (%)				
Methods	Training	Test	Pruned	Parameters	Compression
Softmax	0.70 ± 0.04	1.36 ± 0.10	1.48 ± 0.10	62.8 ± 0.5K	$4.2\times$
LLR	0.64 ± 0.03	1.22 ± 0.08	1.47 ± 0.10	57.4 ± 0.6K	$4.6\times$
Softmax+WD	1.65 ± 0.05	1.48 ± 0.12	1.51 ± 0.09	20.3 ± 0.4K	$13.1\times$
LLR+WD	1.37 ± 0.04	1.30 ± 0.12	1.51 ± 0.08	**19.3 ± 0.3K**	**$13.8\times$**
Iterative Pruning		1.6	1.6	22K	$12\times$

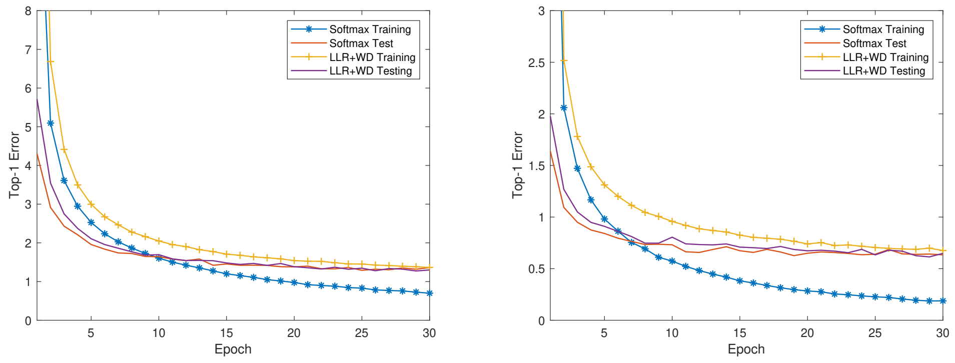


Figure 6: Training and test errors versus epochs using Softmax normalization and LLR representation: (left) LetNet-300-100 and (right) LeNet-5

Table 2 summarizes the top-1 error rates, total number of parameters after pruning, and compression rate for different methods with the LetNet-5 architecture. Fully-connected layers use a fixed pruning setting of 1.25. For convolutional layers, the pruning settings are set as $\theta_k = 0.5 + 0.1 \times k$, where $k = 0, 1, \cdots, 9$. The total numbers of pruned parameters and compression ratio in Table 2 are obtained using the **largest** pruning threshold. Weight sharing and inherent structural regularization of CNN further mitigate the overfitting issues in training. Using LLR representation and weight decaying, the accuracy of the pruned network is even better than the accuracy in training and testing. The snapshot-based method generates efficient networks with 31K parameters and the state-of-the-art performance, better than the ones using the iterative method from Han et al. (2015). Figure 6 (right) also compares the effects of overfitting between softmax normalization and LLR representation.

Table 2: Comparing MNIST performance and weight pruning for LeNet-5

	Top-1 Error (%)				
Methods	Training	Test	Pruned	Parameters	Compression
Softmax	0.19 ± 0.02	0.63 ± 0.09	0.74 ± 0.07	82.5 ± 1.0K	5.2×
LLR	0.14 ± 0.02	0.53 ± 0.07	0.66 ± 0.06	70.0 ± 1.9K	6.2×
Softmax+WD	0.65 ± 0.03	0.69 ± 0.10	0.63 ± 0.05	33.5 ± 1.2K	12.9×
LLR+WD	0.67 ± 0.02	0.64 ± 0.10	0.60 ± 0.06	$\mathbf{31.0 \pm 0.9}$**K**	**13.9**×
Iterative Pruning		0.8	0.8	36K	12×

For comparison purpose, the results using the AdamW algorithm from Loshchilov & Hutter (2018) are summarized in Table 3 and 4 for the LeNet-300-100 and LetNet-5 architectures, respectively. The accuracy differences between training and testing are always larger than previous results using weight decaying and the original ADAM algorithm. The results show that using the AdamW algorithm may generate overparameterized networks. Thus, the snapshot-based pruning method can be a valuable tool for evaluating optimization algorithms.

Table 3: Comparing MNIST performance using AdamW Optimizer for LeNet-300-100

	Top-1 Error (%)				
Methods	Training	Test	Pruned	Parameters	Compression
Softmax	0.71 ± 0.05	1.31 ± 0.08	1.47 ± 0.10	62.8 ± 0.3K	4.2×
LLR	0.65 ± 0.03	1.19 ± 0.08	1.44 ± 0.10	57.4 ± 0.9K	4.6×
Softmax+WD	0.70 ± 0.04	1.31 ± 0.12	1.41 ± 0.11	62.8 ± 0.4K	4.2×
LLR+WD	0.62 ± 0.03	1.22 ± 0.08	1.42 ± 0.07	57.7 ± 0.5K	4.6×

Table 4: Comparing MNIST performance using AdamW Optimizer for LeNet-5

	Top-1 Error (%)				
Methods	Training	Test	Pruned	Parameters	Compression
Softmax	0.18 ± 0.03	0.67 ± 0.08	0.55 ± 0.07	91.6 ± 1.0K	4.7×
LLR	0.15 ± 0.02	0.56 ± 0.07	0.50 ± 0.05	80.3 ± 3.1K	5.4×
Softmax+WD	0.18 ± 0.02	0.67 ± 0.09	0.55 ± 0.05	93.7 ± 0.9K	4.6×
LLR+WD	0.14 ± 0.02	0.57 ± 0.06	0.50 ± 0.04	78.1 ± 2.3K	5.5×

C.2 CIFAR-10 CLASSIFICATION

The CIFAR-10 datasets (Krizhevsky et al., 2009) consists of 50,000 images in the training set and 10,000 images in the test set. Each sample contains 32 × 32 color images drawn from 10 classes.

The data batch size of 128 is used for training. For data augmentation, the standard mirroring and shifting scheme is used. The SGD optimizer is used with initial learning rate of 0.05 and momentum of 0.9, the learning rate is multiplied by 0.1 after 100 and 150 epochs. Weight decaying settings of 6e-4 and 5e-4 are used for the ResNet and DenseNet architectures, respectively. Initial training runs use 200 epochs, while for non-weight parameter adjustment after pruning only 20 epochs are needed. At least 10 runs with random seeds are carried out for each experiment case.

The top-1 error rates, total number of parameters after pruning, and compression ratio for CIFAR-10 dataset with the ResNet architectures are summarized in Table 5. Fully-connected layers use a fixed pruning setting of 0.75. For convolutional layers, the pruning settings are set as $\theta_k = 0.5 + 0.05 \times k$, where $k = 0, 1, \cdots, 9$. The total numbers of pruned parameters and compression ratio in Table 5 are obtained using the **largest** pruning threshold. For all cases, using LLR representation yields better performance and less parameters after pruning. For the ResNet-56 case, using LLR representation with weight decaying reduces the total number of parameters to about 200K without significant loss of performance.

The results in Table 6 show better performance for DenseNet architectures as compared with the ResNet architectures. Fully-connected layers use a fixed pruning setting of 0.75. For convolutional layers, the pruning settings are set as $\theta_k = 0.5 + 0.05 \times k$, where $k = 0, 1, \cdots, 9$. The total numbers of pruned parameters and compression ratio in Table 6 are obtained using the **largest** pruning threshold.

For DenseNet-60 with less than 90K parameters, the performance is comparable to ResNet-56 with 200K parameters. Therefore, overfitting issues with the DenseNet architecture are less prominent than the ResNet architecture. Figure 7 summarize the efficiency trade-offs for both ResNet and DenseNet architectures. Compared with ResNet architecture, the initial DenseNet model sizes are larger, the effects of weight decaying are more prominent than the softmax normalization, and the difference between softmax normalization and LLR representation for DenseNet is smaller.

Table 5: Comparing CIFAR-10 performance and weight pruning for ResNet-20/32/56

		Top-1 Error (%)				Comp
Architecture	Methods	Training	Test	Pruned	Parameters	Ratio
ResNet-20	Softmax	0.53 ± 0.03	7.58 ± 0.19	10.0 ± 0.27	77.8 ± 0.3K	$3.5\times$
	LLR	1.45 ± 0.07	7.9 ± 0.12	10.5 ± 0.32	$\mathbf{69.6 \pm 0.5K}$	$\mathbf{3.9}\times$
ResNet-32	Softmax	0.17 ± 0.03	6.77 ± 0.19	8.32 ± 0.24	130.7 ± 1.0K	$3.6\times$
	LLR	0.50 ± 0.03	6.90 ± 0.17	8.62 ± 0.17	$\mathbf{118.2 \pm 1.0}$K	$\mathbf{4.0}\times$
ResNet-56	Softmax	0.07 ± 0.01	6.05 ± 0.19	7.08 ± 0.32	232.8 ± 1.0K	$3.7\times$
	LLR	0.17 ± 0.02	6.03 ± 0.16	7.06 ± 0.15	$\mathbf{213.3 \pm 1.7K}$	$\mathbf{4.0}\times$

Table 6: Comparing CIFAR-10 performance and weight pruning for DenseNet-40/60/100

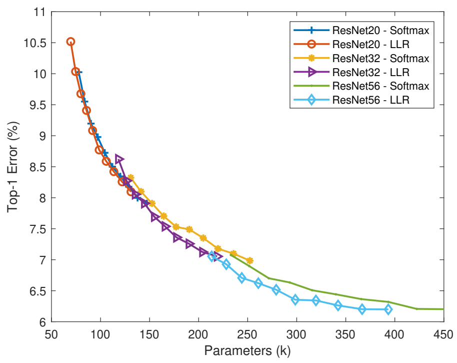
		Top-1 Error (%)				Comp.
Architecture	Methods	Training	Test	Pruned	Parameters	Ratio
DenseNet-40	Softmax	0.26 ± 0.04	7.01 ± 0.31	9.19 ± 0.26	49.7 ± 0.3K	$3.6\times$
	LLR	0.50 ± 0.05	7.14 ± 0.19	9.34 ± 0.38	49.6 ± 0.1K	$3.7\times$
DenseNet-60	Softmax	0.12 ± 0.03	5.87 ± 0.10	7.10 ± 0.26	87.5 ± 0.6K	$3.7\times$
	LLR	0.11 ± 0.02	6.00 ± 0.15	7.34 ± 0.23	87.3 ± 0.5K	$3.7\times$
DenseNet-100	Softmax	0.03 ± 0.01	4.96 ± 0.17	5.61 ± 0.12	198.1 ± 0.7K	$4.0\times$
	LLR	0.03 ± 0.01	4.93 ± 0.22	5.55 ± 0.17	198.5 ± 2.1K	$4.0\times$

C.3 CIFAR-100 CLASSIFICATION

The CIFAR-100 datasets (Krizhevsky et al., 2009) consists of 50,000 images in the training set and 10,000 images in the test set. Each sample contains 32×32 color images drawn from 100 classes. The 100 classes are grouped into 20 superclasses. The data batch size of 128 is used for training.

For data augmentation, the standard mirroring and shifting scheme is used. The SGD optimizer is used with initial learning rate of 0.05 and momentum of 0.9, the learning rate is multiplied by 0.1 after 100 and 150 epochs. A weight decaying setting of 5e-4 is used for both the ResNet and DenseNet architectures. Initial training runs use 200 epochs, while for non-weight parameter adjustment after pruning only 20 epochs are needed. At least 20 runs with random seeds are carried out for each experiment case.

The top-1 error rates for trained and pruned networks, total number of parameters after pruning, and compression ratio for CIFAR-100 dataset are summarized in Table 7 and 8. Fully-connected



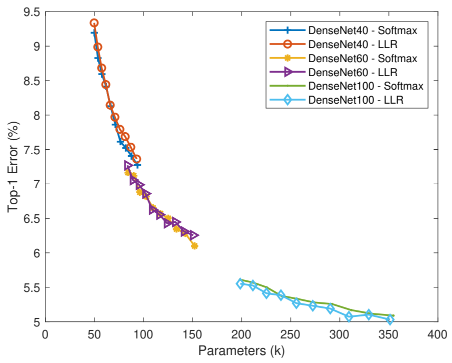


Figure 7: Test errors vs. total parameters trade-offs on the CIFAR-10 dataset using Softmax normalization and LLR representation: (left) ResNet and (right) DenseNet

layers use a fixed pruning setting of 0.75. For convolutional layers, the pruning settings are set as $\theta_k = 0.5 + 0.05 \times k$, where $k = 0, 1, \cdots, 9$. The total numbers of pruned parameters and compression ratio in Table 7 and 8 are obtained using the **median** pruning threshold.

Table 7: Comparing CIFAR-100 performance and weight pruning for ResNet-20/32/56

		Top-1 Error (%)				Comp
Architecture	Methods	Training	Test	Pruned	Parameters	Ratio
ResNet-20	Softmax	13.5 ± 0.1	33.0 ± 0.4	38.2 ± 0.4	120.0 ± 0.6K	$2.3\times$
	LLR	24.8 ± 0.3	35.7 ± 0.3	39.9 ± 0.7	$\mathbf{109.7 \pm 0.5}$**K**	$\mathbf{2.5}\times$
ResNet-32	Softmax	5.8 ± 0.2	31.3 ± 0.3	34.8 ± 0.5	207.0 ± 0.4K	$2.3\times$
	LLR	15.1 ± 0.3	32.4 ± 0.3	34.0 ± 0.6	$\mathbf{189.3 \pm 0.8}$**K**	$\mathbf{2.5}\times$
ResNet-56	Softmax	1.4 ± 0.1	28.7 ± 0.3	30.9 ± 0.3	377.5 ± 1.2K	$2.3\times$
	LLR	6.8 ± 0.2	30.2 ± 0.4	30.0 ± 0.3	$\mathbf{346.6 \pm 2.5}$**K**	$\mathbf{2.5}\times$

Table 8: Comparing CIFAR-100 performance and weight pruning for DenseNet-40/60/100

		Top-1 Error (%)				Comp.
Architecture	Methods	Training	Test	Pruned	Parameters	Ratio
DenseNet-40	Softmax	9.3 ± 0.4	29.9 ± 0.2	34.0 ± 0.6	73.7 ± 0.2K	$2.6\times$
	LLR	12.7 ± 0.2	30.6 ± 0.5	34.8 ± 0.4	76.2 ± 0.2K	$2.5\times$
DenseNet-60	Softmax	2.0 ± 0.1	27.2 ± 0.4	29.2 ± 1.2	129.5 ± 1.7K	$2.6\times$
	LLR	4.2 ± 0.2	27.8 ± 0.4	29.8 ± 1.1	134.3 ± 1.3K	$2.5\times$
DenseNet-100	Softmax	0.1 ± 0.0	23.4 ± 0.6	24.0 ± 0.5	297.5 ± 3.3K	$2.8\times$
	LLR	0.3 ± 0.1	24.0 ± 0.1	23.8 ± 0.3	308.4 ± 1.7K	$2.7\times$