Learning to Prune Deep Neural Networks via Layer-wise Optimal Brain Surgeon

Xin Dong  
Nanyang Technological University  
n1503521a@e.ntu.edu.sg

Shangyu Chen  
Nanyang Technological University  
schen025@e.ntu.edu.sg

Sinno Jialin Pan  
Nanyang Technological University  
sinnopan@ntu.edu.sg

Abstract

How to develop slim and accurate deep neural networks has become crucial for real-world applications, especially for those employed in embedded systems. Though previous work along this research line has shown some promising results, most existing methods either fail to significantly compress a well-trained deep network or require a heavy retraining process for the pruned deep network to re-boost its prediction performance. In this paper, we propose a new layer-wise pruning method for deep neural networks. In our proposed method, parameters of each individual layer are pruned independently based on second order derivatives of a layer-wise error function with respect to the corresponding parameters. We prove that the final prediction performance drop after pruning is bounded by a linear combination of the reconstructed errors caused at each layer. Therefore, there is a guarantee that one only needs to perform a light retraining process on the pruned network to resume its original prediction performance. We conduct extensive experiments on benchmark datasets to demonstrate the effectiveness of our pruning method compared with several state-of-the-art baseline methods.

1 Introduction

Intuitively, deep neural networks [1] can approximate predictive functions of arbitrary complexity well when networks are of a huge amount of parameters, i.e., a lot of layers and neurons. In practice, the size of deep neural networks has been being tremendously increased, from LeNet-5 with less than 1M parameters [2] to VGG-16 with 133M parameters [3]. Such a large number of parameters not only make deep models memory intensive and computationally intensive, but also urge researchers to dig into redundancy of deep neural network. On one hand, in neuroscience, recent studies point out that there are significant redundant neurons in human brain, and memory may have relation with vanishment of specific synapses [4]. On the other hand, in machine learning, both theoretical analysis and empirical experiments have shown the evidence of redundancy in several deep models [5, 6]. Therefore, it is possible to compress deep neural networks without or with little loss in prediction by pruning parameters with carefully designed criteria.

However, finding an optimal pruning solution is NP-hard because the search space for pruning is exponential in terms of parameter size. Recent work mainly focuses on developing efficient algorithms to obtain a near-optimal pruning solution [7, 8, 9, 10, 11]. A common idea behind most exiting approaches is to select parameters for pruning based on certain criteria, such as increase in training error, magnitude of the parameter values, etc. As most of the existing pruning criteria are designed heuristically, there is no guarantee that prediction performance of a deep neural network
can be preserved after pruning. Therefore, a time-consuming retraining process is usually needed to boost the performance of the trimmed neural network.

Instead of consuming efforts on a whole deep network, a layer-wise pruning method, Net-Trim, was proposed to learn sparse parameters by minimizing reconstructed error for each individual layer [6]. A theoretical analysis is provided that the overall performance drop of the deep network is bounded by the sum of reconstructed errors for each layer. In this way, the pruned deep network has a theoretical guarantee on its performance. However, as Net-Trim adopts $\ell_1$-norm to induce sparsity for pruning, it fails to obtain high compression ratio compared with other methods [9,11].

In this paper, we propose a new layer-wise pruning method for deep neural networks, aiming to achieve the following three goals: 1) For each layer, parameters can be highly compressed after pruning, while the reconstructed error is small. 2) There is a theoretical guarantee on the overall prediction performance of the pruned deep neural network in terms of reconstructed errors for each layer. 3) After the deep network is pruned, only a light retraining process is required to resume its original prediction performance.

To achieve our first goal, we borrow an idea from some classic pruning approaches for shallow neural networks, such as optimal brain damage (OBD) [12] and optimal brain surgeon (OBS) [13]. These classic methods approximate a change in the error function via functional Taylor Series, and identify unimportant weights based on second order derivatives. Though these approaches have proven to be effective for shallow neural networks, it remains challenging to extend them for deep neural networks because of the high computational cost on computing second order derivatives, i.e., the inverse of the Hessian matrix over all the parameters. In this work, as we restrict the computation on second order derivatives w.r.t. the parameters of each individual layer only, i.e., the Hessian matrix is only over parameters for a specific layer, the computation becomes tractable. Moreover, we utilize characteristics of back-propagation for fully-connected layers in well-trained deep networks to further reduce computational complexity of the inverse operation of the Hessian matrix.

To achieve our second goal, based on the theoretical results in [6], we provide a proof on the bound of performance drop before and after pruning in terms of the reconstructed errors for each layer. With such a layer-wise pruning framework using second-order derivatives for trimming parameters for each layer, we empirically show that after significantly pruning parameters, there is only a little drop of prediction performance compared with that before pruning. Therefore, only a light retraining process is needed to resume the performance, which achieves our third goal.

The contributions of this paper are summarized as follows. 1) We propose a new layer-wise pruning method for deep neural networks, which is able to significantly trim networks and preserve the prediction performance of networks after pruning with a theoretical guarantee. In addition, with the proposed method, a time-consuming retraining process for re-boosting the performance of the pruned network is waived. 2) We conduct extensive experiments to verify the effectiveness of our proposed method compared with several state-of-the-art approaches.

2 Related Works

Pruning methods have been widely used for model compression in early neural networks [7] and modern deep neural networks [6,8,9,10,11]. In the past, with relatively small size of training data, pruning is crucial to avoid overfitting. Classical methods include OBD and OBS. These methods aim to prune parameters with the least increase of error approximated by second order derivatives. However, computation of the Hessian inverse over all the parameters is expensive. In OBD, the Hessian matrix is restricted to be a diagonal matrix to make it computationally tractable. However, this implicitly assume parameters have no interactions, which may hurt the pruning performance. Different from OBD, OBS makes use of the full Hessian matrix for pruning, which obtains better performance while is much more computational expensive. For example, using OBS on VGG-16 naturally requires to compute the Hessian matrix with a size of $133M \times 133M$.

Regarding pruning for modern deep models, Han et al. [9] proposed to delete unimportant parameters based on magnitude of their absolute values, and retrain the remaining ones to recover the original prediction performance. This method achieves considerable compression ratio in practice. However, as pointed out by pioneer research work [12,13], parameters with low magnitude of their absolute values can be necessary for low error. Therefore, magnitude-based approaches may eliminate wrong parameters, resulting in a big prediction performance drop right after pruning, and poor robustness before retraining [14]. Though some variants have tried to find better magnitude-based
We define the error function \( E \) where \( Y = X \| \cdot \| \). Therefore, to find parameters whose deletion (set to be zeros) minimize (1) can be translated to find \( \hat{\Theta} \) the goal of pruning is to set the values of some elements in \( \Theta \) to be zeros.

The following lemma shows that the layer-wise error is bounded by the error defined in (2).

\[
\delta E^l = \left( \frac{\partial E^l}{\partial \Theta_l} \right)^\top \delta \Theta_l + \frac{1}{2} \delta \Theta_l^\top H_l \delta \Theta_l + O \left( \| \delta \Theta_l \|^3 \right),
\]

3 Layer-wise Optimal Brain Surgeon

3.1 Problem Statement

Given a training set of \( n \) instances, \( \{(x_j, y_j)\}_{j=1}^n \), and a well-trained deep neural network of \( L \) layers (excluding the input layer). Denote the input and the output of the deep neural network by \( X = [x_1, \ldots, x_n] \in \mathbb{R}^{d \times n} \) and \( \hat{Y} \in \mathbb{R}^{L \times 1} \). For a layer \( l \), we denote the input and output of the layer by \( \hat{Y}^{l-1} = [y_1^{l-1}, \ldots, y_n^{l-1}] \in \mathbb{R}^{m_{l-1} \times n} \) and \( Y^l = [y_1^l, \ldots, y_n^l] \in \mathbb{R}^{m_l \times n} \), where \( y_i^l \) can be considered as a representation of \( x_i \) in layer \( l \), and \( Y_n^0 = X \), \( Y_L^L = Y \), and \( m_0 = d \). Using one forward-pass step, we have \( Y^l = \sigma(Z^l) \), where \( Z = W_l^\top Y^{l-1} \) with \( W_l \in \mathbb{R}^{m_{l-1} \times m_l} \) being the matrix of parameters for layer \( l \). For convenience in presentation and proof, we define nonlinear activation function \( \sigma(\cdot) \) as the rectified linear unit (ReLU) [19]. We further denote by \( \hat{\Theta}_l \in \mathbb{R}^{m_{l-1} \times m_l} \) the vectorization of \( W_l \).

3.2 Layer-Wise Error

During layer-wise pruning in layer \( l \), the input \( \hat{Y}^{l-1} \) is fixed as the same as the well-trained network.

Suppose we set the \( q \)-th element of \( \Theta_l \), denoted by \( \Theta_{lq} \), to be zero, and get a new parameter vector, denoted by \( \hat{\Theta}_l \). With \( \hat{Y}^{l-1} \), we obtain a new output for layer \( l \), denoted by \( \hat{Y}^l \). Consider the root of mean square error between \( Y^l \) and \( \hat{Y}^l \) over the whole training data as the layer-wise error:

\[
\epsilon^l = \sqrt{\frac{1}{n} \sum_{j=1}^n \left( \left( \hat{Y}^l_j - Y^l_j \right)^\top \left( \hat{Y}^l_j - Y^l_j \right) \right) = \frac{1}{\sqrt{n}} \| \hat{Y}^l - Y^l \|_F,}
\]

where \( \| \cdot \|_F \) is the Frobenius Norm. Note that for any single parameter pruning, one can compute its error \( \epsilon^l_q \), where \( 1 \leq q \leq m_{l-1} m_l \), and use it as a pruning criterion. This idea has been adopted by some methods [14]. However, in this way, for every parameter at every layer, one has to pass the whole training data once to compute its error measure, which is very computationally expensive. A more efficient approach is to make use of the second order derivatives of the error function to help identify importance of each parameter.

We define the error function \( E(\cdot) \) with respect to \( Z^l \) (outcome of weighted sum before performing the activation function ReLU) as

\[
E^l = E(\hat{Z}^l) = \frac{1}{n} \| \hat{Z}^l - Z^l \|_F^2.
\]

The following lemma shows that the layer-wise error is bounded by the error defined in (2).

**Lemma 3.1.** With the error function (2) and \( Y^l = \sigma(Z^l) \), the following holds: \( \epsilon^l \leq \sqrt{E(\hat{Z}^l)} \).

Therefore, to find parameters whose deletion (set to be zeros) minimize (1) can be translated to find parameters those deletion minimize the error function (2). Following [12, 13], the error function can be approximated by functional Taylor series as follows,

\[
\delta E^l = \left( \frac{\partial E^l}{\partial \Theta_l} \right)^\top \delta \Theta_l + \frac{1}{2} \delta \Theta_l^\top H_l \delta \Theta_l + O \left( \| \delta \Theta_l \|^3 \right),
\]

1 For simplicity in presentation, we suppose the neural network is a feed-forward (fully-connected) network. In Section 3.4, we will show how to extend our method to filter layers in Convolutional Neural Networks.
where \( \delta \) denotes a permutation of a corresponding variable, \( \mathbf{H}_t \equiv \partial^2 E^t / \partial \Theta_t \) is the Hessian matrix w.r.t. \( \Theta_t \), and \( O(||\delta \Theta_t||^3) \) is the third and all higher order terms, which can be ignored. For a well-trained network, gradient w.r.t. parameters is small enough to be ignored as well, i.e., \( \delta E^t / \delta \Theta_t \to 0 \).

Suppose every time we aim to find a parameter \( \Theta_{t[q]} \) to set to be zero such that the change \( \delta E^t \) is minimal. We can formulate it as the following optimization problem:

\[
\min_q \frac{1}{2} \delta \Theta_t^\top \mathbf{H}_t \delta \Theta_t, \quad \text{s.t.} \quad \mathbf{e}_q^\top \delta \Theta_t + \Theta_{t[q]} = 0,
\]

where \( \mathbf{e}_q \) is the unit selecting vector whose \( q \)-th element is 1 and otherwise 0. It can be shown that the optimization problem (4) can be solved by the Lagrange multipliers method [20]. The optimal parameter pruning and the resultant minimal change in the error function can be written as,

\[
\delta \Theta_t = -\frac{\Theta_{t[q]}}{[\mathbf{H}_t^{-1}]_{qq}} \mathbf{H}_t^{-1} \mathbf{e}_q, \quad \text{and} \quad L_q = \delta E^t = \frac{1}{2} (\Theta_{t[q]})^2,
\]

Here, \( L_q \) is referred to as the sensitivity of parameter \( \Theta_{t[q]} \). Then we select parameters to prune based on their sensitivity scores instead of their magnitudes. As mentioned in section 2, magnitude-based criteria which merely consider the numerator in (5) is a poor estimation of sensitivity of parameters. Moreover, in (5), as the inverse Hessian matrix over the training data is involved, it is able to capture data distribution when measuring sensitivities of parameters.

After pruning the parameter, \( \Theta_{t[q]} \), with the smallest sensitivity, the parameter vector is updated via \( \Theta_t^t = \Theta_t + \delta \Theta_t \). With Lemma 3.1 and (5), we have that the layer-wise error for layer \( l \) is bounded by

\[
\varepsilon_q^l \leq \sqrt{E(\hat{Z}^l) - E(Z^l)} \leq \sqrt{\delta E^t} \leq \sqrt{\frac{|\Theta_{t[q]}|}{2[H_t^{-1}]_{qq}}}. \tag{6}
\]

Note that the term \( E(Z^l) \) is canceled out in the second inequality because of the fact that \( E(Z^l) = 0 \). It is worth to mention that though we merely focus on layer \( l \), the Hessian matrix is still a square matrix with size of \( m_{l-1} m_l \times m_{l-1} m_l \). However, we will show how to significantly reduce the computation of \( \mathbf{H}^{-1} \) in Section 3.4.

### 3.3 Layer-Wise Error Propagation and Accumulation

By far, we have known how to prune parameters for each layer, and estimate their introduced errors independently. However, our aim is to control the consistence of the network’s final output \( Y^L \) before and after pruning. To do this, in the following, we show how the layer-wise errors propagate to final output layer, and the accumulated error over multiple layers will not explode.

**Theorem 3.2.** Given a pruned deep network via layer-wise pruning introduced in Section 3.2, each layer has its own layer-wise error \( \varepsilon^l \), \( 1 \leq l \leq L \), then the accumulated error of ultimate network output \( \varepsilon^L = \frac{1}{\sqrt{\pi}} ||\hat{Y}^l - Y^l||_F \) obeys:

\[
\varepsilon^L \leq \sum_{k=1}^{L-1} \left( \prod_{l=k+1}^{L} ||\hat{\Theta}_l||_F \sqrt{\delta E^k} \right) + \sqrt{\delta E^L}. \tag{7}
\]

where \( \hat{Y}^l \) denotes ‘accumulated pruned output’ of layer \( l \), \( \hat{Y}^l = \sigma(\hat{W}_l^T \hat{Y}^{l-1}) \) and \( \hat{Y}^1 = \sigma(\hat{W}_1^T X) \).

Theorem 3.2 shows that: 1) Layer-wise error for a layer \( l \) will be scaled by continued multiplication of parameters’ Frobenius Norm over layers after \( l \) when it propagates to final output; 2) The final error of ultimate network output is bounded by the weighted sum of layer-wise errors.

We prove Theorem 3.2 via induction. First, for \( l = 1 \), (7) holds as a special case of (6). Then suppose that Theorem 3.2 holds up to layer \( l \):

\[
\varepsilon^l \leq \sum_{h=1}^{l-1} \left( \prod_{k=h+1}^{l} ||\hat{\Theta}_k||_F \sqrt{\delta E^h} \right) + \sqrt{\delta E^l}. \tag{8}
\]

In order to show that (8) holds for layer \( l + 1 \) as well, we refer to \( \hat{Y}^{l+1} = \sigma(\hat{W}_l^T Y^l) \) as ‘layer-wise pruned output’, where the input \( Y^l \) is fixed as the same as the originally well-trained network not an accumulated input \( \hat{Y}^l \), and have the following theorem.
Theorem 3.3. Consider layer $l+1$ in a pruned deep network, the difference of its accumulated pruned output, $\hat{Y}^{l+1}$, and layer-wise pruned output, $\hat{Y}^{l+1}$ is bounded by:

$$\|\hat{Y}^{l+1} - \hat{Y}^{l+1}\|_F \leq \sqrt{n}\|\hat{\Theta}^{l}\|_F \varepsilon^l.\tag{9}$$

Proof sketch: Consider one arbitrary element of the layer-wise pruned output $\hat{Y}^{l+1}$:

$$\hat{y}^{l+1}_{ij} = \sigma(\hat{w}^T_i y_j + \hat{w}^T_i (y_j - \tilde{y}_j)) \leq \tilde{y}^{l+1}_{ij} + \sigma(\hat{w}^T_i (y_j - \tilde{y}_j)) \leq \hat{y}^{l+1}_{ij} + |\hat{w}^T_i (y_j - \tilde{y}_j)|,$$

where $\hat{w}_i$ is the $i$-th column of $\hat{W}_i$. Similarly, it holds for accumulated pruned output:

$$\hat{y}^{l+1}_{ij} \leq \hat{y}^{l+1}_{ij} + |\hat{w}^T_i (y_j - \tilde{y}_j)|.$$ 

By combining these two inequalities, we have $|\hat{y}^{l+1}_{ij} - \hat{y}^{l+1}_{ij}| \leq |\hat{w}^T_i (y_j - \tilde{y}_j)|$, and thus have the following inequality in a form of matrix,

$$\|\hat{Y}^{l+1} - \hat{Y}^{l+1}\|_F \leq \|\hat{W}_i(Y^l - \tilde{Y}^l)\|_F \leq \|\hat{\Theta}^{l}\|_F \|Y^l - \tilde{Y}^l\|_F$$

With (6), (9) and the triangle inequality, we are now able to extend (8) to layer $l + 1$:

$$\varepsilon^{l+1} = \frac{1}{\sqrt{n}}\|\hat{Y}^{l+1} - Y^{l+1}\|_F \leq \frac{1}{\sqrt{n}}\|\hat{Y}^{l+1} - \tilde{Y}^{l+1}\|_F + \frac{1}{\sqrt{n}}\|\tilde{Y}^{l+1} - Y^{l+1}\|_F$$

$$\leq \sum_{h=1}^{l} \left( \prod_{k=h+1}^{l+1} \|\hat{\Theta}_k\|_F \cdot \sqrt{\varepsilon^{k+1}} \right) + \sqrt{\varepsilon^{l+1}}.$$ 

Finally, we prove that (8) holds up for all layers, and Theorem 3.2 is a special case when $l = L$.

Consider a general case with (5) and (7); parameter $\Theta_{i[l]}$ who has the smallest sensitivity in layer $l$ is pruned by the $i$-th pruning operation, and this finally adds $\prod_{k=l+1}^{L} \|\hat{\Theta}^k\|_F \sqrt{\varepsilon^l}$ to the ultimate network output error. It is worth to mention that although it seems that the layer-wise error is scaled by a quite large product factor, $S_l = \prod_{k=l+1}^{L} \|\hat{\Theta}^k\|_F$ when it propagates to the final layer, this scaling is still tractable in practice because ultimate network output is also scaled by the same product factor compared with the output of layer $l$. For example, we can easily estimate the norm of ultimate network output via, $\|Y^L\|_F \approx \|\hat{S} \| Y^l\|_F$. If one pruning operation in the 1st layer causes the layer-wise error $\sqrt{\varepsilon^1}$, then the relative ultimate output error is $\xi^L = \|Y^L - \hat{Y}^L\|_F \|Y^L\|_F \approx \sqrt{\varepsilon^L} \|Y^L\|_F$. Thus, we can see that even $S_1$ may be quite large, the relative ultimate output error would still be about $\sqrt{\varepsilon^L}/\|Y^L\|_F$ which is controllable in practice especially when most of modern deep networks adopt maxout layer [21] as ultimate output. Actually, $S_0$ is called as network gain representing the ratio of the magnitude of the network output to the magnitude of the network input.

3.4 The Proposed Algorithm

Prune Fully-Connected Layer: To selectively prune parameters, our approach needs to compute the inverse Hessian matrix at each layer to measure the sensitivities of each parameter of the layer, which is still computationally expensive though tractable. In this section, we present an efficient algorithm that can reduce the size of the Hessian matrix and thus speed up computation on its inverse.

For each layer $l$, according to the definition of the error function used in Lemma 3.1, the first derivative of the error function with respect to $\hat{\Theta}_l$ is $\frac{\partial^2 E}{\partial \hat{\Theta}^l} = -\frac{1}{n} \sum_{j=1}^{n} \frac{\partial z_j}{\partial \hat{\Theta}^l} (\hat{z}_j - z_j)$, where $\hat{z}_j$ and $z_j$ are the $j$-th columns of the matrices $\hat{Z}$ and $Z$, respectively, and the Hessian matrix is defined as:

$$H = \frac{\partial^2 E}{\partial \Theta^l} = \frac{1}{n} \sum_{j=1}^{n} \left( \frac{\partial z_j}{\partial \Theta^l} \right)^T - \frac{\partial z_j}{\partial \Theta^l} (\hat{z}_j - z_j)^T.$$

Note that given a well-trained network, for most cases, $\hat{z}_j$ is quite close to $z_j$, we simply ignore the term containing $\hat{z}_j - z_j$. Even in the late-stage of pruning when this difference is not small, we can still ignore the corresponding term [13]. For layer $l$ that has $m_l$ output units, $z_j = [z_{j1}, \ldots, z_{jm_l}]$, the Hessian matrix can be calculated via

$$H = \frac{1}{n} \sum_{j=1}^{n} H_j = \frac{1}{n} \sum_{j=1}^{n} \sum_{k=1}^{m_l} \frac{\partial z_{ki}}{\partial \Theta^l} \left( \frac{\partial z_{kj}}{\partial \Theta^l} \right)^T,$$

where the Hessian matrix for a single instance, $H_j$, is a block diagonal square matrix of the size $m_{l-1} \times m_l$. Specifically, gradient of the first output unit $z_{1j}$ with respect to all parameters $\Theta_i$ is $\frac{\partial z_{1j}}{\partial \Theta_i} = \left[ \frac{\partial z_{1j}}{\partial \Theta^i}, \ldots, \frac{\partial z_{1j}}{\partial \Theta^i} \right]$, where $w_i$ is the $i$-th column of $W_i$. 

5
As \( z_{1j} \) is the layer output before activation function, its gradient is simply to calculate, and more importantly all output units’ gradients are equal to the layer input: \( \frac{\partial z_{1j}}{\partial w_{ij}} = y_{j}^{(l-1)} \), and \( \frac{\partial z_{1j}}{\partial w_{ij}} = 0 \) for \( i \neq j \).

It can be shown that the block diagonal square matrix \( H^{l} \)’s diagonal blocks \( H_{ii}^{l} \) are all equal to \( \psi^{l} = y_{j}^{(l-1)} (y_{j}^{(l-1)})^{-1} \), and the inverse Hessian matrix \( H^{-1} \) is also a block diagonal square matrix with its diagonal blocks being \( \frac{1}{\sum_{j=1}^{n} \psi^{j}} \). In addition, normally \( \psi = \frac{1}{n} \sum_{j=1}^{n} \psi^{j} \) is degenerate and its pseudo-inverse can be calculated recursively via standard matrix inversion: \( \psi_{j+1}^{-1} = \psi_{j}^{-1} - \frac{\psi_{j}^{-1} y_{j}^{(l-1)} (y_{j}^{(l-1)})^{-1} \psi_{j}^{-1}}{n + (y_{j}^{(l-1)})^{-1} \psi_{j}^{-1} (y_{j}^{(l-1)})^{-1}} \), where \( \psi_{k} = \frac{1}{n} \sum_{j=1}^{n} \psi^{j} \) with \( \psi^{(l)} = \alpha I, \alpha \in [10^{4}, 10^{8}] \), and \( \psi^{-1} = \psi^{1}^{-1} \). The size of \( \psi \) is then reduced to \( m_{l-1} \), and the computational complexity of calculating \( H^{-1} \) is \( O(m_{l-1}^{3}) \).

To make estimated minimal change of the error function optimal in \( \Theta \), two conditions have to be met: 1) the whole network is at a local minimum in error (gradient is small enough to be ignored), and 2) the Hessian matrix is exact. Since the Hessian matrix only depends on layer input, it is always able to be exact even after several pruning operations. The only parameter we have to control is the layer-wise error \( \epsilon \) to ensure gradient to be small enough after pruning. That is one of reasons why there is a “pruning inflection point” after which layer-wise error would drop dramatically. In practice, user can incrementally increase the size of pruned parameters based on sensitivity \( L_{q} \), and make a trade-off between pruning ratio and performance drop to set a proper tolerable error threshold or pruning ratio.

According to above analysis, the procedure of our pruning algorithm for a fully-connected layer \( l \) is as follows. Step 1: Get layer input \( y^{(l-1)} \) from a well-trained deep network. Step 2: Calculate the Hessian matrix \( H_{ii}^{l} \) and its pseudo-inverse over the dataset, and get the whole pseudo-inverse of the Hessian matrix. Step 3: Compute optimal parameter change \( \delta \Theta_{l} \) and the sensitivity \( L_{q} \) for each parameter in layer \( l \). Set tolerable error threshold \( \epsilon \). Step 4: Pick up parameters \( \Theta_{l_{q}} \)’s with the smallest sensitivity scores. Step 5: If \( \sqrt{L_{q}} < \epsilon \), prune the parameter \( \Theta_{l_{q}} \)’s and get new parameter values via \( \hat{\Theta} = \Theta + \delta \Theta \), then repeat Step 4; otherwise stop pruning.

**Prune Convolutional Layer:** It is straightforward to generalize our method to a convolutional layer and its variants if we vectorize filters of each channel and consider them as special fully-connected layers that have multiple inputs (patches) from a single instance. Consider a vectorized filter \( w_{i} \), where \( 1 \leq i \leq m_{l} \), it acts similarly to parameters which are connected to the same output unit in a fully-connected layer. However, the difference is that for a single input unit \( j \), every filter step of a sliding window across of it will extract a patch \( C_{jn} \) from the input volume. Similarly, each pixel \( z_{jn} \) in the 2-dimensional activation map that gives the response to each patch corresponds to one output unit in a fully-connected layer. Hence, in convolutional layer case, \( \Theta \) is generalized as \( H = \frac{1}{n} \sum_{j=1}^{n} \sum_{i=1}^{m_{l}} \sum_{jn} \frac{\partial z_{jn}}{\partial w_{ij}} \cdot w_{ij} \), where \( H \) is also a block diagonal square matrix whose diagonal blocks are all the same. Then, we can slightly revise the computation of the Hessian matrix, and extend the algorithm for fully-connected layers to convolutional layers.

Note that the accumulated error of ultimate network output can be linearly bounded by layer-wise error as long as the model is feed-forward. Thus, L-OBS is a general pruning method and friendly with most of feed-forward neural networks whose layer-wise Hessian can be computed expeditiously with slight modification. However, if models have sizable layers like ResNet-101, L-OBS may not be economical because of computational cost of Hessian, which will be studied in our future work.
4 Experiments

In this section, we verify the effectiveness of our proposed Layer-wise OBS (L-OBS) using various architectures of deep neural networks in terms of compression ratio (CR), error rate before retraining, and the number of iterations required for retraining to achieve satisfactory performance. Here, CR is defined as ratio of the number of preserved parameters to the number of original parameters. We conduct comparison results of L-OBS with the following pruning approaches: 1) Randomly prune, 2) OBD [12], 3) LWC [9], 4) DNS [11], and 5) Net-Trim [6]. The deep architectures used for experiments include: LeNet-300-100 [2] and LeNet-5 [2] on the MNIST dataset, CIFAR-Net [22] on the CIFAR-10 dataset, AlexNet [23] and VGG-16 [3] on the ImageNet ILSVRC-2012 dataset. For experiments, we first well-train the networks, and apply various pruning approaches on networks to evaluate their performance. The retraining batch size, crop method and other hyper-parameters are under the same setting as used in LWC. Note that to make comparisons fair, we do not adopt any other pruning related methods like Dropout or sparse regularizers on MNIST. In practice, L-OBS can work well along with them as shown on large scale dataset CIFAR-10 and ImageNet.

4.1 Overall Comparison Results

The overall comparison results are shown in Table 1. In the first set of experiments, we prune each layer of the well-trained LeNet-300-100 with compression ratios: 6.7%, 20% and 65%, achieving slightly better overall compression ratio (7%) than LWC. Under comparable compression ratio, L-OBS has quite less drop of performance and lighter retraining compared with LWC whose performance is almost ruined by pruning. Classic pruning approach OBD is also compared though we observe that Hessian matrices of most modern deep models are strongly non-diagonal in practice. Besides relative heavy cost to obtain the second derivatives via the chain rule, OBD suffers from drastic drop of performance when it is directly applied to modern deep models as shown in Table 1.

To properly prune each layer of LeNet-5, we increase tolerable error threshold $\epsilon$ from relative small initial value to incrementally prune more parameters, monitor model performance, stop pruning and set $\epsilon$ until encounter the “pruning inflection point” mentioned in Section 3.4. In practice, we prune each layer of LeNet-5 with compression ratio: 54%, 43%, 6% and 25% and retrain pruned model with mere milli iterations in other methods. As DNS retrain the pruned network after every pruning operation, we are not able to report its error rate of pruned network before retraining. However, as can be seen, similar to LWC, the total number of iterations used by DNS for rebooting the network is very large compared with L-OBS. Results of retraining iterations of DNS are reported from [11] and the other experiments are implemented based on TensorFlow [24]. In addition, in scenario requiring high pruning ratio, L-OBS can be quite flexibly adopted to its iterative version-doing pruning and light retraining alternately, to obtain higher pruning ratio with relative higher cost of pruning. With two iterations of pruning and retraining, L-OBS is able to achieve as the same pruning ratio as DNS with much lighter total retraining: 643 iterations on LeNet-300-100 and 841 iterations on LeNet-5.

Regarding comparison results with CIFAR-Net, our well-trained CIFAR-Net achieves testing error of 18.57% with Dropout and Batch-Normalization. Then we prune the well-trained network with LWC and L-OBS and get the similar results as experiments on other network architecture. We also observe that LWC and other retraining-required methods always require much smaller learning rate in retraining. This is because representation capability of the pruned networks which have much fewer parameters is damaged during pruning based on a principle that number of parameters is an important factor for representation capability. However, L-OBS can still adopt original learning rate to retrain the pruned networks. Under this consideration, L-OBS not only ensures a warm-start for retraining, but also finds important connections (parameters) and preserve capability of representation for the pruned network instead of ruining model with pruning.

For AlexNet, as with the previous experiments, L-OBS achieves overall compression ratio 11% without loss of accuracy taking 2.9 hours on 48 Intel Xeon(R) CPU E5-1650 to compute Hessians and 3.1 hours on NVIDIA Tian X GPU to retrain pruned model (i.e. 18.1K iterations). Cost of Hessian inverse’s computation in L-OBS is negligible compared with heavy retraining in other methods. This claim can also be supported by analysis of time complexity. As mentioned in Section 3.4, time complexity of calculating $H^{-1}$ is $O(nm^2_{l-1})$. Assume that neural networks are retrained via SGD, time complexity of retraining is approximately $O(IdM)$, where $d$ is the size of the mini-batch, $M$ and $I$ is total number of parameters and iterations. For $M \approx \sum_{l=1}^{L} (m^2_{l-1})$ and retraining in previous methods always requires millions of iterations $(Id \gg n)$, complexity of calculating Hessian (inverse) being a revised AlexNet for CIFAR-10 containing three convolutional layers and two fully connected layers.
Our Method

We have proposed a novel L-OBS pruning framework to prune parameters based on second order derivatives information of the layer-wise error function and provided a theoretical guarantee on the overall error in terms of the reconstructed errors for each layer. Our proposed L-OBS can prune considerable number of parameters with tiny drop of performance and reduce or even omit retraining. More importantly, it identifies and preserves the real important part of networks when pruning compared with previous methods, which may help to dive into nature of neural networks.
References

[1] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. Nature, 521(7553):436–444, 2015.
[2] 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.
[3] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. arXiv preprint [arXiv:1409.1556] 2014.
[4] Luisa de Vivo, Michele Bellesi, William Marshall, Eric A Bushong, Mark H Ellisman, Giulio Tononi, and Chiara Cirelli. Ultrastructural evidence for synaptic scaling across the wake/sleep cycle. Science, 355(6324):507–510, 2017.
[5] Misha Denil, Babak Shakibi, Laurent Dinh, Nando de Freitas, et al. Predicting parameters in deep learning. In Advances in Neural Information Processing Systems, pages 2148–2156, 2013.
[6] Nguyen N. Aghasi, A. and J. Romberg. Net-trim: A layer-wise convex pruning of deep neural networks. Journal of Machine Learning Research, 2016.
[7] Russell Reed. Pruning algorithms-a survey. IEEE transactions on Neural Networks, 4(5):740–747, 1993.
[8] Yunchao Gong, Liu Liu, Ming Yang, and Lubomir Bourdev. Compressing deep convolutional networks using vector quantization. arXiv preprint [arXiv:1412.6115] 2014.
[9] 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.
[10] Yi Sun, Xiaogang Wang, and Xiaou Tang. Sparsifying neural network connections for face recognition. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pages 4856–4864, 2016.
[11] Yiwen Guo, Anbang Yao, and Yurong Chen. Dynamic network surgery for efficient dnns. In Advances In Neural Information Processing Systems, pages 1379–1387, 2016.
[12] Yann LeCun, John S Denker, Sara A Solla, Richard E Howard, and Lawrence D Jackel. Optimal brain damage. In NIPs, volume 2, pages 598–605, 1989.
[13] Babak Hassibi, David G Stork, et al. Second order derivatives for network pruning: Optimal brain surgeon. Advances in neural information processing systems, pages 164–164, 1993.
[14] Nikolas Wolfe, Aditya Sharma, Lukas Drude, and Bhiksha Raj. The incredible shrinking neural network: New perspectives on learning representations through the lens of pruning. arXiv preprint [arXiv:1701.04465] 2017.
[15] Hengyuan Hu, Rui Peng, Yu-Wing Tai, and Chi-Keung Tang. Network trimming: A data-driven neuron pruning approach towards efficient deep architectures. arXiv preprint [arXiv:1607.03250] 2016.
[16] Hao Li, Asim Kadav, Igor Durandovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. arXiv preprint [arXiv:1608.08710] 2016.
[17] Cheng Tai, Tong Xiao, Yi Zhang, Xiaogang Wang, et al. Convolutional neural networks with low-rank regularization. arXiv preprint [arXiv:1511.06067] 2015.
[18] Baoyuan Liu, Min Wang, Hassan Foroosh, Marshall Tappen, and Marianna Pensky. Sparse convolutional neural networks. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pages 806–814, 2015.
[19] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In Aistats, volume 15, page 275, 2011.
[20] R Tyrrell Rockafellar. Convex analysis. princeton landmarks in mathematics, 1997.
[21] Ian J Goodfellow, David Warde-Farley, Mehdi Mirza, Aaron C Courville, and Yoshua Bengio. Maxout networks. ICML (3), 28:1319–1327, 2013.
[22] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. 2009.
[23] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In Advances in neural information processing systems, pages 1097–1105, 2012.

[24] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. arXiv preprint arXiv:1603.04467, 2016.