Supplemental Material for "Rethinking Group Fisher Pruning for Efficient Label-Free Network Compression"

Jong-Ryul Lee[1]
jongryul.lee@etri.re.kr
Yong-Hyuk Moon[†1,2]
yhmoon@etri.re.kr

[1] Artificial Intelligence Research Laboratory
Electronics and Telecommunications Research Institute (ETRI)
Daejeon, Korea

[2] University of Science and Technology (UST)
Daejeon, Korea

1 Layer Grouping Algorithm

This section describes our algorithm for finding convolution layers that share coupled output channels.

1.1 Background

Graph Structure. One can consider a simple directed acyclic graph to represent a neural network model where a node represents a layer, and an edge represents the connection between two layers. However, such a graph cannot support a general neural network model because a layer can be shared like the parameter sharing of ALBERT in [1]. To correctly handle such a general neural network model, we need to design a graph structure more carefully.

Definition 1.1 (Layer-Level Graph)**.** *Given a neural network $\mathcal{N}$, the layer-level graph is defined to be a directed graph $\mathcal{G} = (V, E)$ where V is the set of nodes and E is the set of directed edges. Each node $v \in V$ is identified by tuple (L, i) where L is a neural layer in $\mathcal{N}$ and i represents the i-th visit to L during the inference of $\mathcal{N}$. Let us denote L and i by $layer(v)$ and $visit(v)$. Each edge $e \in E$ is identified by tuple (u, v) where u is a source node and v is a target node.*

From the definition of the layer-level graph, an edge (u, v) represents the data flow from u to v. Note that since a layer can be visited multiple times in the inference of a neural network model, we distinguish each visit to the same layer by introducing i in the definition. It is easy to see that this gives us the following proposition.

© 2022. The copyright of this document resides with its authors.
It may be distributed unchanged freely in print or electronic forms.
† Corresponding author.

Proposition 1.1. *For any neural network model $\mathcal{N}$, the layer-level graph for $\mathcal{N}$ is acyclic.*

Note that a layer may have multiple different output tensors, but we omit to define them for notational clarity.

Delayed Depth-First Search. We need a new graph traversal algorithm beyond Depth-First Search (DFS) to correctly understand and explore a graph structure standing for a complex neural network model. It should be noticed that DFS itself cannot mimic the data flow of a neural network model from its inputs to its outputs. For example, consider a neural network model having concatenation, and suppose that DFS visits a concatenation layer C. Then, after visiting C, DFS will directly move to the next layer, which is connected from C. The problem in this situation is that we cannot figure out the shape of the output tensor of C and its successors because there is no guarantee that DFS has visited all the inbound layers to C. Such information is essential for transforming a neural network model like channel pruning. If the shape of the output tensor of C is unknown, we cannot update the weight tensor of convolution layers connected to C.

To remedy the limitation of DFS, we devise a different graph traversal algorithm named Delayed Depth-First Search (D-DFS). D-DFS starts from nodes corresponding to the input layers. This algorithm acts like DFS except that it waits to traverse v until all the inbound neighbors of v are visited. We denote a set of the inbound neighbors of v by $\mathbf{N}_{\text{in}}(v)$. Since we can check whether all nodes in $\mathbf{N}_{\text{in}}(v)$ are visited in $O(1)$ with a binary vector of size $|\mathbf{N}_{\text{in}}(v)|$ and $\mathcal{G}$ is acyclic, the time complexity of D-DFS is $O(|E|)$.

One can see that D-DFS correctly emulates the inference of a neural network. Thus, D-DFS can be used to search and manipulate the topology of $\mathcal{N}$ with consideration of the actual inference flow of $\mathcal{N}$.

1.2 Layer Grouping

Let us introduce an algorithm to find layers that have coupled in/out channels by residual connections. In this section, a node corresponding to a convolutional layer is called a convolution node for topological interpretation.

Definition 1.2 (Affecting Node)**.** *For any convolution node u, it is called an affecting node of another node v, if there is no path from u to v containing another convolution node. The layer corresponding to u is called an affecting layer (node).*

The concept of the affecting node is used to formulate channel-level coupling. Suppose that a convolution node u is an affecting node of another node v. Then, the output channels of $layer(u)$ are coupled with the input channels of $layer(v)$. Suppose that a node w is also an affecting node of v. It is easy to see that the output channels of $layer(u)$ are also coupled with those of $layer(w)$. Based on this property, we devise an efficient algorithm to find groups of convolution nodes sharing the coupled output channels, which is described in Algorithm 1.

This algorithm first defines a map Ω to store affecting layers of the nodes in V. It fills Ω by running D-DFS starting from the input nodes in V_{start}. During D-DFS, for each visited edge $(u,v) \in E$, $transfer(u,v,\Omega)$, described in Algorithm 2 is called. This function is basically designed to transfer affecting layers from u to v except in two cases. The first case is that u is a convolution node. In this case, u is an affecting layer of v itself. The other is that v is concatenation. Suppose that the outputs of two convolutional layers a and b are merged by concatenation, and the output goes to the input of a convolutional layer c. Because the input channels of c connected to a and b are not overlapped, the output channels of a and b

Algorithm 1: Layer Grouping

```
Input: 𝒢: the layer-level graph
Output: 𝔾: The minimal groups
begin
    Initialize a map Ω from V;
    foreach v ∈ V do
        Ω[v] := {};
    V_start := {v ∈ V | |N_in(v)| = 0};
    D-DFS(𝒢, Ω) with applying transfer for each visited edge;
    𝔾 := {};
    foreach v ∈ V do
        candidates := [];
        foreach G ∈ 𝔾 do
            if |G ∩ Ω[v]| > 0 then
                Push G to candidates;
        if |candidates| = 0 then
            Push Ω[v] to 𝔾;
        else if |candidates| = 1 ∧ candidates[0] = Ω[v] then
            continue;
        else
            new := {};
            foreach G ∈ candidates do
                new := new ∪ G;
                Remove G from 𝔾;
            new := new ∪ Ω[v];
            Push new to 𝔾;
    return 𝔾;
```

Algorithm 2: *transfer*(u, v, Ω)

```
Input: (u,v) ∈ E: the visited edge (u,v) during D-DFS, Ω: the affecting layer map
begin
    if u is convolution then
        Ω[v] := Ω[v] ∪ {u};
    else if v is concatenation then
        Ω[v] := {};
    else
        Ω[v] := Ω[v] ∪ Ω[u]
```

are not coupled. Therefore, we do not need to transfer the affecting nodes of u to v, so that $\Omega[v]$ is assigned to the empty set.

After running D-DFS to get the affecting layers of every node, the algorithm finds layers sharing coupled output channels through Ω as follows. Let us define a sharing group as a group of layers sharing coupled output channels. A set of the affecting layers of a node can

be a sharing group because the output channels of affecting layers are coupled. In addition, if there are two sharing groups overlapping with each other, we do not need to keep the smaller group. Based on this property, the minimal sharing groups are computed by checking all the affecting layers. The detailed procedure is described in Algorithm 1.

In Lines 2-5 of Algorithm 1, the algorithm initializes variables for running D-DFS to fill Ω. In Line 6, it computes Ω with D-DFS. In Lines 8-23, the algorithm computes the minimal sharing groups as follows. For each node in V, it finds sharing groups in $\mathbb{G}$ overlapped with $\Omega[v]$. If there is no such sharing group, $\Omega[v]$ is added into $\mathbb{G}$ in Lines 13-14. If there is only one sharing group $\mathbf{G}$ and $\mathbf{G} = \Omega[v]$, then the algorithm does nothing in Lines 15-16. Otherwise, all sharing groups overlapped with $\Omega[v]$ and $\Omega[v]$ are merged into a single sharing group in Lines 17-23. In this way, we can efficiently find minimal sharing groups.

2 The Computation of *t-rank*

For a layer l in a neural network model, to compute *t-rank*(l), we need to know the value of *t-rank* for all inbound layers of l. Recall that D-DFS waits to traverse l until all the inbound layers of it are visited. Thus, we can compute *t-rank*(l) via D-DFS with the guarantee that for any inbound layer u of l, *t-rank*(u) is already computed.

The value of *t-rank* for all layers can be computed via running D-DFS from input layers to outputs once. Thus, the computation algorithm for *t-rank* totally takes $O(m)$ where m is the number of connections between the layers, which equals the time complexity of DFS.

Table 1: The Effect of Distillation

Method	ENetB0		RNet50		DNet121	
	Top-1	#F (B)	Top-1	#F (B)	Top-1	#F (B)
GF+D	81.38	0.22	**81.05**	1.85	82.43	1.68
GF+Label	81.49	0.22	80.27	1.85	83.11	1.68
GF+D+Label	**82.06**	0.22	79.69	1.85	**83.13**	1.68
CURL+D	82.47	0.22	**80.76**	2.10	82.13	1.65
CURL+Label	83.40	0.22	80.32	2.10	82.62	1.65
CURL+D+Label	**83.59**	0.22	80.27	2.10	**82.71**	1.65
BTS_{FCD}+D	81.79	0.22	**81.01**	2.10	82.91	1.65
BTS_{FCD}+D+Label	**82.16**	0.22	80.42	2.10	**83.13**	1.65

3 Distillation Effect

To understand the effect of our distillation method, we conduct more experiments. The results are described in Table 1. In this table, 'D' represents training with our distillation method, and 'Label' represents training with ground truth labels. For example, CURL_{D} is equivalent to CURL+D.

The results show that our distillation method has similar effectiveness compared to using only the task loss with the labels. For EfficientNet and DenseNet, it is quite reasonable that the best strategy in terms of accuracy is to use our distillation method and the ground truth labels together. On the other hand, the best strategy for ResNet is to use our distillation method solely. This is somewhat strange, but consistently happens over GF, CURL, and BTS_{FCD}. Since all the hyperparameters for training EfficientNet, ResNet, and DenseNet are

the same, we need to do more experiments with different options for the hyperparameters. Such experiments would be a good starting point for further study.

References

[1] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. Albert: A lite bert for self-supervised learning of language representations. In *ICLR*, 2020.