LayerMerge: Neural Network Depth Compression through Layer Pruning and Merging

Jinuk Kim [1 2] **Marwa El Halabi** [3] **Mingi Ji** [4] **Hyun Oh Song** [1 2]

Abstract

Recent works show that reducing the number of layers in a convolutional neural network can enhance efficiency while maintaining the performance of the network. Existing depth compression methods remove redundant non-linear activation functions and merge the consecutive convolution layers into a single layer. However, these methods suffer from a critical drawback; the kernel size of the merged layers becomes larger, significantly undermining the latency reduction gained from reducing the depth of the network. We show that this problem can be addressed by jointly pruning convolution layers and activation functions. To this end, we propose *LayerMerge*, a novel depth compression method that selects which activation layers and convolution layers to remove, to achieve a desired inference speedup while minimizing performance loss. Since the corresponding selection problem involves an exponential search space, we formulate a novel surrogate optimization problem and efficiently solve it via dynamic programming. Empirical results demonstrate that our method consistently outperforms existing depth compression and layer pruning methods on various network architectures, both on image classification and generation tasks. We release the code at https://github.com/snu-mllab/LayerMerge.

1. Introduction

Convolutional neural networks (CNNs) have shown remarkable performance in various vision-based tasks such as classification, segmentation, and object detection (Krizhevsky
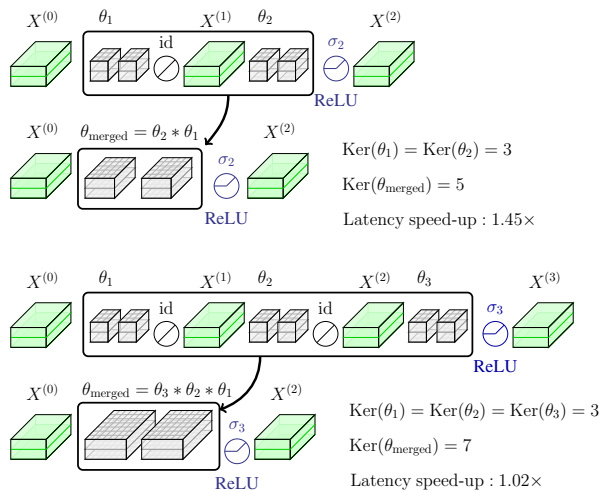

Figure 1: An illustration of the increase in kernel size significantly undermining the latency reduction in the depth compression framework. Here, θ_l denotes the l-th convolution parameter, $X^{(l)}$ denotes the l-th feature map, and $\mathrm{Ker}(\cdot)$ denotes the kernel size of the parameter. As the layers are merged, the kernel size of the merged layer continues to grow, impeding the latency reduction. The latency is measured for the depicted model, on RTX2080 Ti, with channel size 256, input resolution 56×56, and batch size 128.

et al., 2012; Chen et al., 2018a; Girshick, 2015). More recently, diffusion probabilistic models employing U-Net based architecture are demonstrating great performance in various high-quality image generation tasks (Ho et al., 2020; Ronneberger et al., 2015). However, the impressive capabilities of these models on complex vision tasks come at the cost of increasingly higher computational resources and inference latency as they are scaled up (Nichol & Dhariwal, 2021; Liu et al., 2022).

An effective approach to address this is *structured pruning*, which consists of removing redundant regular regions of weights, such as channels, filters, and entire layers, to make the model more efficient without requiring specialized hardware while preserving its performance. In particular, *channel pruning* methods remove redundant channels in CNNs (Molchanov et al., 2016; Li et al., 2017; Shen et al., 2022).
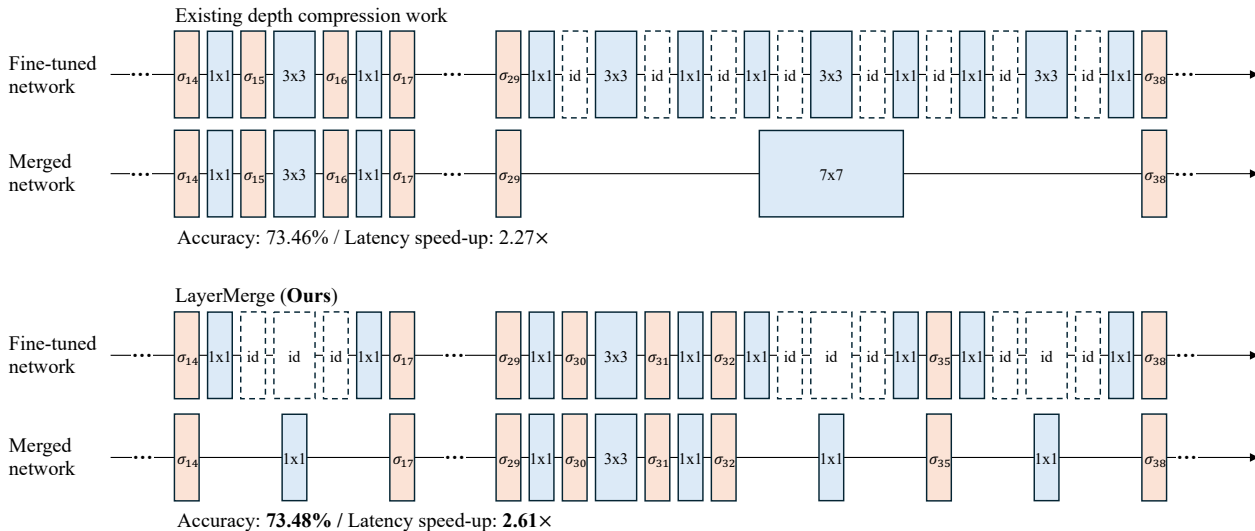
Figure 2: A qualitative example comparing our method to the depth compression baseline (Kim et al., 2023), applied to MobileNetV2-1.4 model on ImageNet dataset. Existing depth compression methods have limitations in reducing latency due to the inevitable increase in the kernel size of the merged layer. Our method effectively bypasses this challenge by jointly optimizing the selection of the convolution layers and the non-linear activation layers.

While these methods have shown significant improvements in accelerating models and reducing their complexity, they are less effective on architectures with a low number of channels compared to methods that reduce the number of layers in the network (Elkerdawy et al., 2020; Fu et al., 2022; Kim et al., 2023). One approach to reduce the number of layers is *layer pruning*, which removes entire layers in the network (Chen & Zhao, 2018; Elkerdawy et al., 2020). Although such methods achieve a larger speed-up factor, they tend to suffer from severe degradation in performance due to their aggressive nature in removing parameters (Fu et al., 2022).

To this end, a line of research called *depth compression* or *depth shrinking* proposes to remove redundant non-linear activation functions and to merge the resulting consecutive convolution layers into a single layer to achieve higher inference speed-up (Dror et al., 2022; Fu et al., 2022; Kim et al., 2023). However, these methods have a fundamental limitation; merging consecutive convolution layers leads to an increase in the kernel size of the merged layers, significantly hindering the latency reduction gained from reducing the depth of the network (Figure 1).

In this work, we argue that this critical drawback of existing depth compression methods can be addressed by extending the search space and jointly optimizing the selection of both *non-linear activation layers* and the *convolution layers* to remove (Figure 2). Since the corresponding optimization problem involves an exponential number of potential solutions, we propose a novel surrogate optimization problem that can be solved exactly via an efficient dynamic programming (DP) algorithm.

In particular, we make the following contributions:

- We propose a novel depth compression approach that introduces a new pruning modality; removing the convolution layers in addition to activation layers. This formulation encompasses the search space of existing depth compression methods and enables us to bypass the increase in the kernel size of the merged convolution layers (Figure 2).

- We propose a surrogate optimization problem that can be solved exactly via a DP algorithm. We further develop an efficient method to construct DP lookup tables, leveraging the inherent structure of the problem.

- We conduct extensive experiments and demonstrate the effectiveness of our method on image classification and generation tasks with different networks, including ResNet, MobileNetV2, and DDPM (He et al., 2016; Sandler et al., 2018; Ho et al., 2020).

2. Preliminaries

Let f_{θ_l} and σ_l denote the l-th convolution layer and the l-th activation layer, respectively. Here, θ_l denotes the parameters of the convolution layer f_{θ_l}. An L-layer CNN can be represented as $\bigcirc_{l=1}^{L}(\sigma_l \circ f_{\theta_l})$, where $\bigcirc$ denotes an iterated function composition, and the last activation function σ_L is set to identity.

Depth compression methods eliminate less important non-linear activation layers, then *merge* consecutive convolution layers into a single layer by applying a convolution opera-

tion to their parameters (Dror et al., 2022; Fu et al., 2022; Kim et al., 2023). This approach leverages the fact that the successive convolution layers $f_{\theta_j} \circ f_{\theta_{j-1}} \circ \cdots \circ f_{\theta_i}$ can be represented as single equivalent convolution layer $f_{\theta_j * \cdots * \theta_i}$, where $*$ denotes the convolution operation.

However, this method has a fundamental limitation; the kernel size of the merged layer increases as more layers are merged. This significantly undermines the latency speed-up achieved by reducing the number of layers (Figure 1). To illustrate, let us denote the convolution parameter of the merged layer as $\hat{\theta} := \theta_j * \cdots * \theta_i$ and assume that all convolution layers have a stride of 1. Then, the kernel size of this merged layer is given by

$$\operatorname{Ker}(\hat{\theta}) = 1 + \sum_{l=i}^{j} \left(\operatorname{Ker}(\theta_l) - 1 \right), \qquad (1)$$

where the $\operatorname{Ker}(\cdot)$ denotes the kernel size of the given convolution parameter. To address this, we propose to jointly remove unimportant convolution layers and non-linear activation layers (Figure 2).

3. LayerMerge

In this section, we present our proposed depth compression method *LayerMerge*, designed to address the increase in kernel size resulting from depth compression and find more efficient networks. We first formulate the NP-hard subset selection problem we aim to solve. Then, we present a surrogate optimization problem that can be exactly solved with an efficient dynamic programming algorithm. Afterwards, we examine the theoretical optimality, complexity, and practical cost of our approach.

3.1. Selection problem

We observe that if we selectively replace certain convolution layers with identity functions (id), we can effectively alleviate the problem of increasing kernel sizes resulting from merging layers. Indeed, an identity layer can be represented with a 1×1 depthwise convolution layer, where the parameter values are set to 1. We denote the corresponding convolution parameters as $\theta_{\mathrm{id}} \in \mathbb{R}^{1 \times C_{\mathrm{in}} \times 1 \times 1}$, where C_{in} denotes the number of input channels. Then, it is evident from Equation (1) that θ_{id} does not contribute to the expansion of the kernel size.

To this end, we propose to optimize two sets of layer indices: $A \subseteq [L-1] := \{1, \ldots, L-1\}$ for activation layers and $C \subseteq [L] = \{1, \ldots, L\}$ for convolution layers, where L represents the depth of the original network. The indices in A denote where we keep the original activation layers, while the indices in C correspond to where the original convolution layers are maintained. For any layer l not in A, we replace its activation function σ_l with the id function. Simi-

larly, if a layer l is not in C, we substitute its convolution layer f_{θ_l} with the identity layer $f_{\theta_{\mathrm{id}}}$.

It is worth noting that it is non-trivial to remove a convolution layer when the shapes of the input and output feature maps are different. To address this, we define a set of irreducible convolution layer indices R, where the input shape and the output shapes differ. Concretely, we define R as $R := \{l \in [L] : \operatorname{Shape}(X^{(l-1)}) \neq \operatorname{Shape}(X^{(l)})\}$, where $\operatorname{Shape}(\cdot)$ denotes the shape of a tensor and $X^{(l)}$ denotes the l-th layer feature map. We then restrict the choice of C to supersets of R, i.e., $R \subseteq C$.

Given a desired latency target $T_0 > 0$, our goal is to select A and C that maximize the performance of the resulting model after fine-tuning, while satisfying the latency target after merging. We formulate this problem as follows:

$$\max_{A \subseteq [L-1], C \subseteq [L]} \max_{\theta} \operatorname{Perf}\left(\bigcirc_{l=1}^{L} \left(\sigma_{A,l} \circ f_{C,\theta_l,l} \right) \right) \qquad (2)$$

subject to

$$R \subseteq C, \qquad \text{(irreducible conv)}$$

$$\sigma_{A,l} = \left(\mathbb{1}_A(l)\sigma_l + (1 - \mathbb{1}_A(l))\,\mathrm{id} \right), \qquad \text{(replaced act)}$$

$$f_{C,\theta_l,l} = \left(\mathbb{1}_C(l)f_{\theta_l} + (1 - \mathbb{1}_C(l))\,f_{\theta_{\mathrm{id}}} \right), \qquad \text{(replaced conv)}$$

$$\forall i \in [|A|+1] : \hat{\theta}_i = \underset{l=a_{i-1}+1}{\overset{a_i}{\circledast}} \left(\mathbb{1}_C(l)\theta_l + (1 - \mathbb{1}_C(l))\,\theta_{\mathrm{id}} \right),$$

$$\text{(merged parameters)}$$

$$T\left(\bigcirc_{i=1}^{|A|} \left(\sigma_{a_i} \circ f_{\hat{\theta}_i} \right) \right) < T_0, \qquad \text{(latency constraint)}$$

where $\circledast$ denotes an iterated convolution operation, $a_0 = 0$, $a_{|A|+1} = L$, and $(a_i)_{i=1}^{|A|}$ denotes the elements of the set A in ascending order. Here, $\operatorname{Perf}(\cdot)$ and $T(\cdot)$ denote the performance and latency of the network, respectively. The performance of the network is defined as a task-dependent metric: accuracy for classification tasks and negative diffusion loss for generation tasks (Ho et al., 2020). The indicator function $\mathbb{1}_X(x)$ is equal to 1 if $x \in X$, and 0 otherwise. We denote by $\sigma_{A,l}$ the l-th activation layer replaced according to set A, and by $f_{C,\theta_l,l}$ the l-th convolution layer replaced according to set C. The parameter $\hat{\theta}_i$ is the i-th convolution layer in the merged network.

Note that we used the pruned network before merging in the objective, while the latency constraint is applied to the merged network. Both networks represent the same function and yield the same output, and thus have the same performance objective. However, in practice, we observe that it is better to merge consecutive convolution layers only at inference time after fine-tuning is finished. We chose to use the network before merging in the objective to stress this.

3.2. Surrogate optimization problem

Solving Problem (2) in general is NP-hard. We propose to assign an importance value for each merged layer and optimize the sum of the importance values as a surrogate objective. This is a common approximation used in the literature (Shen et al., 2022; Frantar & Alistarh, 2022; Kim et al., 2023). We also approximate the overall latency of the merged network with the sum of the layer-wise latencies (Cai et al., 2019; Shen et al., 2022). The main challenge that we face then is the exponentially large number of potential combinations of the merged layers that arise from the joint optimization over C. To this end, we develop an efficient method for measuring latency and importance values, leveraging the inherent combinatorial structure of the problem. Subsequently, we compute the optimal solutions of the surrogate problem in polynomial-time using a dynamic programming algorithm.

Latency cost We construct a latency lookup table for all possible merged layers. A straightforward approach is to construct a table with entries $T[i, j, C]$ for all $i, j \in \{0, \ldots, L\}$, $i < j$, where each entry denotes the latency of the layer obtained by merging from the $(i + 1)$-th layer to the j-th layer after replacing the convolution layers according to C. However, this approach is not feasible because it requires measuring the latency for the exponential number of possible sets $C \cap (i, j]$.

To address this, we note that the choice of C only affects the latency of a merged layer via the size of its kernel, since the number of input and output channels is fixed. To this end, we propose to construct the latency table with entries $T[i, j, k]$, where the last index k denotes the kernel size of the merged layer, given by $k = 1 + \sum_{l \in C \cap (i,j]} (\text{Ker}(\theta_l) - 1)$.

Let K_{ij} be the set of possible merged kernel sizes that can appear after merging from the $(i + 1)$-th layer to the j-th layer. Note that $|K_{ij}| \le 1 + \sum_{l=i+1}^{j} (\text{Ker}(\theta_l) - 1)$. Therefore, constructing the latency table with entries $T[i, j, k]$ requires $O(L^2 K_0)$ latency measurements, where $K_0 \coloneqq \sum_l \text{Ker}(\theta_l)$ denotes the sum of the kernel sizes in the original network. This is significantly lower than the $O(L^2 2^L)$ measurements needed to construct the latency table with entries $T[i, j, C]$.

Importance value Similarly to the latency cost case, we construct the importance lookup table with entries $I[i, j, k]$. Each entry denotes the importance of the layer obtained by merging from the $(i + 1)$-th layer to the j-th layer, with the merged layer having a kernel size k.

We define the importance value of each merged layer as the change in the performance after replacing the corresponding part of the original network with the merged layer. However, multiple choices of C can yield the same kernel size k in

the merged layer, but will vary in performance.

We propose to keep the convolution layers with the largest parameters ℓ_1-norm among those resulting in the same merged kernel size k. This simple yet effective criterion is often used for channel and layer pruning (Li et al., 2017; Elkerdawy et al., 2020). Concretely, for any $i, j \in \{0, \ldots, L\}$, $i < j$, and $k \in K_{ij}$, we let

$$\widehat{C}_{ijk} \coloneqq \operatorname*{argmax}_{C_{ij} \subseteq (i,j]} \sum_{l \in C_{ij}} \|\theta_l\|_1 \tag{3}$$
$$\text{subject to } 1 + \sum_{l \in C_{ij}} (\text{Ker}(\theta_l) - 1) = k,$$
$$R \cap (i, j] \subseteq C_{ij}.$$
$$\widetilde{C}_{ijk} \coloneqq \{1, \ldots, i\} \cup \widehat{C}_{ijk} \cup \{j + 1, \ldots, L\}.$$
$$\widetilde{A}_{ij} \coloneqq \{1, \ldots, i\} \cup \{j, \ldots, L - 1\}.$$

Computing $\widetilde{C}_{ijk}$ has a negligible cost. We can now define the importance $I[i, j, k]$ as follows:

$$I[i, j, k] \coloneqq \exp\left(\max_\theta \text{Perf}\left(\bigcirc_{l=1}^{L} \left(\underbrace{\sigma_{\widetilde{A}_{ij}, l}}_{\text{Replaced act}} \circ \underbrace{f_{\widetilde{C}_{ijk}, \theta_l, l}}_{\text{Replaced conv}} \right) \right) \right.$$
$$\left. - \max_\theta \text{Perf}\left(\underbrace{\bigcirc_{l=1}^{L} \left(\sigma_l \circ f_{\theta_l} \right)}_{\text{Original network}} \right) \right). \tag{4}$$

We use $\exp(\cdot)$ to normalize the importance value. This choice is based on our empirical observation that using positive values for importance leads to better performance by favoring solutions with more activation layers. In practice, to estimate the first term, we measure the performance of the network after fine-tuning it for a few steps. For the second term, we use the performance of the pre-trained original network. Constructing the importance table requires $O(L^2 K_0)$ importance value evaluations, which is identical to the number of latency measurements needed.

Optimization problem After we pre-compute the latency and importance lookup tables, T and I, we maximize the sum of the importance values of merged layer under the constraint on the sum of the latency costs. This can be formulated as follows:

Algorithm 1 DP algorithm for Problem (5)

---

input Importance I, latency T, latency budget T_0, discretization level P

Initialize $M[0,t] \leftarrow 0$ for $t \geq 0$, $M[l,t] \leftarrow -\infty$ for $t < 0$, $A[0,t] \leftarrow \emptyset$, $C[0,t] \leftarrow \emptyset$

Discretize latency values in T

for $l = 1$ **to** L **do**

  for $t \in \{\frac{T_0}{P}, \frac{2T_0}{P} \dots, T_0\}$ **do**

    $l^*, k^* \leftarrow \underset{0 \leq l' < l, k \in K_{l'l}}{\operatorname{argmax}} (M[l', t - T[l', l, k]] + I[l', l, k])$

    $M[l,t] \leftarrow M[l^*, t - T[l^*, l, k^*]] + I[l^*, l, k^*]$

    $A[l,t] \leftarrow A[l^*, t - T[l^*, l, k^*]] \cup \{l^* : l^* > 0\}$

    $\widehat{C}_{l^* l k^*} \leftarrow$ compute via Equation (3)

    $C[l,t] \leftarrow C[l^*, t - T[l^*, l, k^*]] \cup \widehat{C}_{l^* l k^*}$

  end for

end for

$A^* \leftarrow A[L, T_0]$, $C^* \leftarrow C[L, T_0]$

$k_i^* \leftarrow 1 + \sum_{l \in C^* \cap (a_{i-1}^*, a_i^*]} (\operatorname{Ker}(\theta_l) - 1), \forall i \in [|A^*| + 1]$

output A^*, C^*, and $(k_i^*)_{i=1}^{|A^*|+1}$.

Algorithm 2 LayerMerge

---

input Input network f, latency budget T_0, descretization level P

for $i = 0$ **to** $L - 1$ **do**

  for $j = i + 1$ **to** L **do**

    for $k \in K_{ij}$ **do**

      $\widetilde{A}_{ij}, \widetilde{C}_{ijk} \leftarrow$ compute via Equation (3)

      $\hat{\theta}_{ij} = \circledast_{l=i+1}^{j} \left(\mathbb{1}_{\widetilde{C}_{ijk}}(l)\theta_l + \left(1 - \mathbb{1}_{\widetilde{C}_{ijk}}(l)\right)\theta_{\mathrm{id}} \right)$

      $I[i,j,k] \leftarrow$ compute via Equation (4)

      $T[i,j,k] \leftarrow T(f_{\hat{\theta}_{ij}})$

    end for

  end for

end for

$A^*, C^*, k^* \leftarrow$ Algorithm $1(I, T, T_0, P)$

Replace activation functions outside A^* by id and convolution layers outside C^* by $f_{\theta_{\mathrm{id}}}$

Fine-tune and merge the network

output Merged network

$$\max_{A \subseteq [L-1], k_i} \sum_{i=1}^{|A|+1} I[a_{i-1}, a_i, k_i] \quad (5)$$

$$\text{subject to} \quad \sum_{i=1}^{|A|+1} T[a_{i-1}, a_i, k_i] < T_0, \quad \text{(latency constraint)}$$

$$k_i \in K_{a_{i-1}a_i}, \quad \text{(merged kernel size)}$$

where $a_0 = 0$, $a_{|A|+1} = L$ as before. Given a solution $A^*, (k_i^*)_{i=1}^{|A^*|+1}$ of Problem (5), the corresponding set of convolution layers we keep is given by $C^* = \bigcup_i \widehat{C}_{a_{i-1}^*, a_i^*, k_i^*}$.

3.3. Dynamic programming algorithm

Once we construct the lookup tables T and I, we can obtain an exact solution of Problem (5) for discretized latency values, using dynamic programming (DP). In particular, we discretize latency values in the lookup table T by rounding them down to the closest values in $\{\frac{T_0}{P}, \frac{2T_0}{P} \dots, T_0\}$, where P is a large natural number that represents the discretization level.

Then, we consider a sub-problem of Problem (5) where we maximize over the first $l \in \{0, \dots, L\}$ layers with latency budget $t \in \{\frac{T_0}{P}, \frac{2T_0}{P} \dots, T_0\}$, as follows:

$$\max_{A \subseteq [l-1], k_i} \sum_{i=1}^{|A|+1} I[a_{i-1}, a_i, k_i] \quad (6)$$

$$\text{subject to} \quad \sum_{i=1}^{|A|+1} T[a_{i-1}, a_i, k_i] < t, \quad \text{(latency constraint)}$$

$$k_i \in K_{a_{i-1}a_i}, \quad \text{(merged kernel size)}$$

where $a_0 = 0$ and $a_{|A|+1} = l$. We define $M[l,t]$ as the corresponding maximum objective value achievable in the sub-problem (6). Then $M[L, T_0]$ gives the maximum objective value achievable in Problem (5). We initialize $M[0,t] = 0$ for $t \geq 0$, and $M[l,t] = -\infty$ for $t < 0$. Then, for $l > 0$, the recurrence of the DP algorithm can be written as follows:

$$M[l,t] = \max_{0 \leq l' < l, k \in K_{l'l}} \left(\underbrace{M[l', t - T[l', l, k]]}_{\text{Optimal importance sum until } l'\text{-layer}} + \underbrace{I[l', l, k]}_{\text{Importance value of the last compressed layer}} \right). \quad (7)$$

We present the DP algorithm for Problem (5) in Algorithm 1. Once we compute the optimal sets A^* and C^*, we fine-tune the network after replacing the layers accordingly. Then, we merge every convolution layer between a_{i-1}^* and a_i^* for all $i \in [|A| + 1]$ at inference time. We outline the overall procedure of LayerMerge in Algorithm 2.

3.4. Theoretical analysis

We show that the proposed DP algorithm optimally solves the surrogate optimization Problem (5). The proof is given in Appendix B.

Theorem 3.1. *The solution A^* and $(k_i^*)_{i=1}^{|A^*|+1}$ given by Algorithm 1 is an optimal solution of Problem (5).*

The time complexity of the DP algorithm is $O\left(L^2 P K_0\right)$. In practice, the DP algorithm is highly efficient, typically completing within a few seconds on CPU. Furthermore, our method efficiently computes the DP lookup tables, exploiting the structure of the problem. It is worth noting that this can be done in an embarrassingly parallel fashion. We report the wall-clock time for constructing the DP lookup tables in Appendix C.

4. Experiments

In this section, we provide experimental results demonstrating the effectiveness of our method across different network architectures and tasks. We apply our method on ResNet-34 and MobileNetV2 models (He et al., 2016; Sandler et al., 2018) for the image classification task, and on the DDPM model (Ho et al., 2020) for the image generation task. We present additional details on how we handle normalization layers, strided convolutions, padding, skip connections, and other network-specific implementation details of our method in Appendix A.

Baselines We compare our method to a depth compression method and a layer pruning method since both approaches rely, like our method, on reducing the number of layers of the network to accelerate it. We also include a comparison with a knowledge distillation method in Appendix E. For the depth compression baseline, we use the state-of-the-art work of Kim et al. (2023), which we denote as *Depth*.

Existing layer pruning methods do not directly take latency into consideration during pruning (Jordao et al., 2020; Chen & Zhao, 2018; Elkerdawy et al., 2020). To address this gap, we propose a variant of our method tailored specifically for layer pruning, which we use as our layer pruning baseline.

Specifically, we assign an importance value $I[l]$ and a latency cost $T[l]$ for each convolution layer $l \in [L]$. The importance value of the convolution layer is defined as the change in performance resulting from replacing the layer with the identity layer in the original network and then fine-tuning it for a few steps. As in *LayerMerge*, we approximate the overall latency and importance of the network by the sum of the layer-wise latencies and importance values, re-

spectively. We solve the following surrogate problem:

$$\max_{R \subseteq C \subseteq [L]} \sum_{l \in C} I[l] \tag{8}$$

$$\text{subject to } \sum_{l \in C} T[l] < T_0. \quad \text{(latency constraint)}$$

Problem (8) is a 0-1 knapsack problem that can be solved exactly for discretized latency values via a DP algorithm in $O(LP)$ time, where P denotes the discretization level. We denote this method as *LayerOnly*.

We additionally compare with a channel pruning baseline for each network. Note that channel pruning is an orthogonal approach to depth compression. Nonetheless, we include channel pruning results to study the effectiveness of reducing the number of layers compared to reducing the width in different types of networks. We compare with HALP (Shen et al., 2022) on ResNet-34, with AMC and MetaPruning (He et al., 2018; Liu et al., 2019) on MobileNetV2, and with Diff-Pruning (Fang et al., 2023) on DDPM.

For ResNet-34, we apply the depth compression and the channel pruning baselines using their publicly available code (Shen et al., 2022; Kim et al., 2023). We also use the pre-trained weights of the original network from Wightman et al. (2021) for all compression methods. For MobileNetV2, we report the results for the depth compression baseline from the original paper (Kim et al., 2023). For the channel pruning baselines, we prune channels of each layer using the same channel ratio of their optimized model from their open-sourced code (He et al., 2018; Liu et al., 2019). We use the pre-trained weights of the original network from the public code of Kim et al. (2023) to ensure a fair comparison. For DDPM, we apply the depth compression and the channel pruning baselines using their open-sourced code (Kim et al., 2023; Fang et al., 2023). We use the pre-trained weights of the original network from Song et al. (2021) for all compression methods in DDPM. In all networks, we compare the compression results in each table using an identical fine-tuning schedule for fair comparison.

Finally, we provide ablation studies on the importance of joint optimization on activation layers and convolution layers in our method, where we compare our method to sequentially applying Depth then LayerOnly. Throughout this section, we refer to each compressed model obtained by Depth, LayerOnly, and LayerMerge as Depth-$p\%$, LayerOnly-$p\%$, and LayerMerge-$p\%$, respectively. Here, $p\%$ is calculated as T_0/T_{orig}, where T_0 is the chosen latency budget and T_{orig} is the latency of the original model.

Experimental setup We construct the latency lookup table of each method on RTX2080 Ti GPU and report the wall-clock latency speed-up of the compressed networks measured on the same device. We provide the details on the

Table 1: Accuracy and latency speed-up of applying compression methods to ResNet-34 on ImageNet dataset. The latency speed-up is measured on RTX2080 Ti GPU at batch size 128.

Network	Acc (%) ↑	PyTorch Speed-up ↑	TensorRT Speed-up ↑
ResNet-34	74.42	1.00×	1.00×
HALP-80% (Shen et al., 2022)	73.98	1.23×	**1.25×**
Depth-78% (Kim et al., 2023)	73.49	1.24×	1.14×
LayerOnly-73% (Ours)	74.06	1.33×	1.24×
LayerMerge-71% (Ours)	**74.26**	**1.36×**	**1.25×**
HALP-65% (Shen et al., 2022)	73.36	1.48×	1.45×
Depth-68% (Kim et al., 2023)	73.35	1.40×	1.26×
LayerOnly-59% (Ours)	73.31	**1.65×**	1.48×
LayerMerge-60% (Ours)	**73.46**	1.56×	**1.50×**
HALP-55% (Shen et al., 2022)	72.69	1.69×	1.69×
Depth-63% (Kim et al., 2023)	72.33	1.43×	1.24×
LayerOnly-49% (Ours)	72.58	**1.82×**	1.64×
LayerMerge-50% (Ours)	**72.84**	1.79×	**1.72×**

Table 2: Accuracy and latency speed-up of applying compression methods to MobileNetV2-1.0 on ImageNet dataset. The latency speed-up is measured on RTX2080 Ti GPU at batch size 128.

Network	Acc (%) ↑	PyTorch Speed-up ↑	TensorRT Speed-up ↑
MobileNetV2-1.0	72.89	1.00×	1.00×
AMC-70% (He et al., 2018)	72.01	1.32×	1.34×
Depth-74% (Kim et al., 2023)	72.83	1.62×	**1.42×**
LayerOnly-73% (Ours)	69.66	1.30×	1.35×
LayerMerge-55% (Ours)	**72.99**	**1.63×**	**1.42×**
Depth-66% (Kim et al., 2023)	72.13	1.88×	1.57×
LayerMerge-46% (Ours)	**72.46**	**1.90×**	**1.65×**
Depth-59% (Kim et al., 2023)	71.44	2.07×	1.79×
LayerMerge-38% (Ours)	**71.74**	**2.18×**	**1.84×**
Depth-53% (Kim et al., 2023)	70.65	2.47×	1.97×
LayerMerge-33% (Ours)	**70.99**	**2.49×**	**2.05×**

measurement process in Appendix C. Notably, we measure the latency of the network in two different formats, PyTorch format and TensorRT compiled format (Paszke et al., 2017; Vanholder, 2016). When measuring latency speedup, we use a batch size of 128 for the ImageNet dataset and 64 for the CIFAR10 dataset, following the same measurement protocol from Kim et al. (2023); Shen et al. (2022); Fang et al. (2023). For ResNet-34, we fine-tune each pruned network for 90 epochs following the same fine-tuning recipe as HALP (Shen et al., 2022). For MobileNetV2, we fine-tune for 180 epochs, using the same fine-tuning recipe as Kim et al. (2023). For DDPM, we follow the fine-tuning and sampling recipe of Diff-Pruning (Fang et al., 2023), except for the learning rate which we set to 4×10^{-4} since it leads to better performance. We present a representative subset of the results in Tables 1 to 6. For additional results, see Appendix E.

4.1. Classification task results

In this section, we evaluate the performance of the different pruning methods on ResNet-34, MobileNetV2-1.0, and MobileNetV2-1.4 models, on the ImageNet dataset (He et al., 2016; Sandler et al., 2018; Russakovsky et al., 2015). We report the last top-1 accuracy of the compressed model after fine-tuning, evaluated on the validation set, and its corresponding latency speedup.

ResNet-34 Table 1 summarizes the different compression results on ResNet-34. HALP-p% refers to the pruned model obtained by HALP by setting the latency budget to be p% of the original model latency. LayerMerge outperforms existing channel pruning and depth compression baselines. Specifically, we achieve 1.10× speed-up in PyTorch with

0.77% point higher accuracy compared to the depth compression baseline (comparing LayerMerge-71% to Depth-78%). It is worth noting that the layer pruning variant of our method performs on par with LayerMerge on ResNet-34. This is mainly due to ResNet-34 being more suitable for layer pruning than depth compression. Indeed, LayerMerge frequently opts for pruning convolution layers over activation functions when applied to ResNet-34.

MobileNetV2 Table 2 presents the various compression results on MobileNetV2-1.0. AMC-p% denotes the pruned model obtained by AMC by setting the FLOPs budget to be p% of the original model FLOPs. LayerMerge surpasses existing channel pruning and depth compression baselines, as well as our layer pruning variant. In particular, we achieve 1.63× speed-up in PyTorch without losing accuracy from the original network (LayerMerge-55%). Table 3 further shows the compression results on MobileNetV2-1.4. LayerMerge outperforms existing methods, offering 0.23% point higher accuracy with a larger speed-up compared to the depth compression baseline (comparing LayerMerge-43% to Depth-62%). The gain in efficiency stems from its unique capability to jointly prune layers and merge them, as demonstrated in Figure 2.

4.2. Generation task results

In this section, we evaluate the performance of the different pruning methods on DDPM model (Ho et al., 2020) on the CIFAR10 dataset (Krizhevsky, 2009). We measure performance using the standard Frechet Inception Distance (FID) metric (Heusel et al., 2017). We report the last FID of the compressed model after fine-tuning, evaluated on the validation set, and its corresponding latency speedup.

Table 3: Accuracy and latency speed-up of applying compression methods to MobileNetV2-1.4 on ImageNet dataset. The latency speed-up is measured on RTX2080 Ti GPU at batch size 128.

Network	Acc (%) ↑	PyTorch Speed-up ↑	TensorRT Speed-up ↑
MobileNetV2-1.4	76.28	1.00×	1.00×
MetaPruning-1.0× (Liu et al., 2019)	73.69	1.59×	1.38×
Depth-62% (Kim et al., 2023)	74.68	1.93×	**1.61×**
LayerOnly-75% (Ours)	73.94	1.27×	1.28×
LayerMerge-43% (Ours)	**74.91**	**1.99×**	**1.61×**
Depth-60% (Kim et al., 2023)	74.19	1.99×	1.67×
LayerMerge-42% (Ours)	**74.48**	**2.07×**	**1.73×**
Depth-53% (Kim et al., 2023)	73.46	2.27×	1.85×
LayerMerge-35% (Ours)	**73.99**	**2.39×**	**1.93×**
Depth-46% (Kim et al., 2023)	72.57	2.41×	2.01×
LayerMerge-30% (Ours)	**73.29**	**2.72×**	**2.12×**

Table 4: FID metric and PyTorch latency speed-up of compression methods applied to DDPM on CIFAR10 dataset. The latency speed-up is measured on RTX2080 Ti GPU at batch size 64.

Network	FID ↓	PyTorch Speed-up ↑	Fine-tune Steps ↓
DDPM	4.18	1.00×	-
Depth-89% (Kim et al., 2023)	4.21	1.04×	100K
LayerOnly-77% (Ours)	4.64	1.09×	100K
LayerMerge-73% (Ours)	**4.16**	**1.13×**	100K
Depth-85% (Kim et al., 2023)	4.78	1.08×	100K
LayerOnly-69% (Ours)	5.52	1.14×	100K
LayerMerge-70% (Ours)	**4.55**	**1.16×**	100K
LayerOnly-54% (Ours)	6.23	1.26×	100K
LayerMerge-58% (Ours)	**5.61**	**1.27×**	100K

Table 5: FID metric and PyTorch latency speed-up of compression methods applied to channel pruned DDPM on CIFAR10 dataset. The latency speed-up is measured on RTX2080 Ti GPU at batch size 64. Diff-p% denotes applying Diff-Pruning with p% compression ratio.

Network	FID ↓	PyTorch Speed-up ↑	Fine-tune Steps ↓
DDPM	4.18	1.00×	-
Diff-30% (Fang et al., 2023)	4.85	1.40×	100K
Diff-60% (Fang et al., 2023)	7.90	2.33×	100K
Diff-70% (Fang et al., 2023)	9.89	2.57×	200K
Diff-60% → Depth-86% (Kim et al., 2023)	9.09	2.42×	200K
Diff-60% → LayerOnly-63% (Ours)	10.15	**2.59×**	200K
Diff-60% → LayerMerge-70% (Ours)	**8.92**	**2.59×**	200K

Table 6: Accuracy and corresponding latency speed-up compared to sequential optimization and our method evaluated with MobileNetV2-1.0 on ImageNet dataset.

Network	Acc (%) ↑	PyTorch Speed-up ↑	TensorRT Speed-up ↑
MobileNetV2-1.0	72.89	1.00×	1.00×
Depth-74% → LayerOnly-72%	71.72	2.09×	1.79×
LayerMerge-39% (Ours)	**71.89**	**2.15×**	**1.80×**
Depth-74% → LayerOnly-64%	70.14	2.33×	2.09×
LayerMerge-33% (Ours)	**70.81**	**2.47×**	**2.15×**

DDPM Table 4 reports the different compression results on DDPM. LayerMerge shows superior performance compared to the existing depth compression baseline and the layer pruning variant of our method. Specifically, we achieve 1.08× speed-up with a lower FID metric compared to the depth compression baseline (comparing LayerMerge-73% to Depth-89%). The channel pruning baseline Diff-Pruning shows superior performance than LayerMerge here (Table 5). This is likely due to DDPM having more channel redundancy than other models.

Channel pruned DDPM We note that channel pruning and depth compression methods can be jointly applied. We thus include results where we apply our method to the channel pruned DDPM model obtained by Diff-Pruning in Table 5 (Fang et al., 2023). Diff-p% denotes the pruned model obtained with Diff-Pruning by removing p% of the channels in each layer. The results show that combining our method

with Diff-Pruning achieves a larger speed-up than solely relying on Diff-Pruning, and our method consistently outperforms the depth compression and layer pruning baselines in this setting as well.

4.3. Ablation studies

Our method jointly optimizes the selection of the activation layers and convolution layers to prune. An alternative way to do this is to sequentially optimize the selection of each type of layer independently. We compare our method to sequentially applying Depth then LayerOnly on MobileNetV2 on the ImageNet dataset in Table 6. Our method outperforms the sequential optimization baseline, underlining the importance of our joint optimization approach. We provide the details in Appendix D.

5. Related Work

Unstructured pruning Unstructured pruning methods remove individual neurons in the network to achieve network sparsity (Han et al., 2015; Hubara et al., 2021; Benbaki et al., 2023; Frantar & Alistarh, 2023). The closest method to ours in this line of work is Frantar & Alistarh (2022), which proposes a dynamic programming algorithm that determines layer-wise sparsity under a given latency constraint. However, unstructured pruning methods require specialized

hardware to achieve computational savings.

Channel pruning   In contrast, structured pruning methods, which consist of removing redundant regular regions of weights, can yield computational savings on off-the-shelf hardware. Among such methods are channel pruning methods, which remove redundant channels in a convolutional neural network (Molchanov et al., 2016; 2019; Li et al., 2017). Aflalo et al. (2020) formulate this as a knapsack problem that maximizes the sum of channel importance values under a given FLOPs budget. Similarly, Shen et al. (2022) formulates another knapsack problem, which maximizes the sum of channel importance values under a latency constraint on a target device.

Layer pruning   Layer pruning methods aim to make a shallower network by entirely removing certain convolution layers (Jordao et al., 2020; Chen & Zhao, 2018; Elkerdawy et al., 2020). However, their aggressive nature in removing parameters results in a large performance degradation under high compression ratios (Fu et al., 2022).

Depth compression   Instead, depth compression methods focus on eliminating unimportant non-linear activation layers, then merging consecutive convolution layer to reduce the network's depth (Dror et al., 2022; Fu et al., 2022; Kim et al., 2023). In particular, Dror et al. (2022) and Fu et al. (2022) propose to train a soft parameter that controls the intensity of non-linearity of each layer with an additional loss that penalizes the absolute value of the soft parameter. More recently, Kim et al. (2023) propose to maximize the sum of the importance values of merged layers under a latency constraint, via a dynamic programming algorithm. However, this line of work suffers from a fundamental drawback as merging layers leads to an increase in the kernel size of the merged layers. Fu et al. (2022) sidestep this issue, as they only consider merging within inverted residual blocks. However, this restriction not only limits the applicability of their method to mobile-optimized CNNs, but also limits its performance. Indeed, Kim et al. (2023) have shown that their method outperforms that of (Fu et al., 2022).

6. Conclusion

We propose *LayerMerge*, a novel efficient depth compression method, which jointly prunes convolution layers and activation functions to achieve a desired target latency while minimizing the performance loss. Our method avoids the problem of increasing kernel size in merged layers, which existing depth compression methods suffer from. It consistently outperforms existing depth compression and layer pruning baselines in various settings.

Impact Statement

This work contributes to the area of NN compression, in particular to compressing pre-trained CNNs and diffusion models. As such, it helps reduce the energy consumption and computational resources of such models at inference. Hence, this work contributes to reducing the environmental impact of NNs and enabling their use on resource-constrained devices like mobile phones and for latency-critical applications such as self-driving cars. On the other hand, pruning has been shown to have a disparate impact on performance between different sub-groups of data, which amplifies existing algorithmic bias (Hooker et al., 2020; Paganini, 2020). There is an ongoing effort to mitigate this negative impact of pruning either using fairness-aware pruning (Lin et al., 2022) or by modifying the objective during fine-tuning of the pruned model (Tran et al., 2022). The latter approach can be applied to our pruning method (Hashemizadeh et al., 2024).

References

Aflalo, Y., Noy, A., Lin, M., Friedman, I., and Zelnik, L. Knapsack pruning with inner distillation. *arXiv preprint arXiv:2002.08258*, 2020.

Benbaki, R., Chen, W., Meng, X., Hazimeh, H., Ponomareva, N., Zhao, Z., and Mazumder, R. Fast as chita: Neural network pruning with combinatorial optimization. In *ICML*, 2023.

Cai, H., Zhu, L., and Han, S. Proxylessnas: Direct neural architecture search on target task and hardware, 2019.

Chen, L.-C., Zhu, Y., Papandreou, G., Schroff, F., and Adam, H. Encoder-decoder with atrous separable convolution for semantic image segmentation. In *ECCV*, 2018a.

Chen, S. and Zhao, Q. Shallowing deep networks: Layer-wise pruning based on feature representations. *IEEE*

transactions on pattern analysis and machine intelligence, 2018.

Chen, X., Mishra, N., Rohaninejad, M., and Abbeel, P. Pixelsnail: An improved autoregressive generative model. In *ICML*, 2018b.

Ding, X., Zhang, X., Ma, N., Han, J., Ding, G., and Sun, J. Repvgg: Making vgg-style convnets great again. In *CVPR*, 2021.

Dror, A. B., Zehngut, N., Raviv, A., Artyomov, E., Vitek, R., and Jevnisek, R. Layer folding: Neural network depth reduction using activation linearization. In *BMVC*, 2022.

Elkerdawy, S., Elhoushi, M., Singh, A., Zhang, H., and Ray, N. To filter prune, or to layer prune, that is the question. In *ACCV*, 2020.

Fang, G., Ma, X., and Wang, X. Structural pruning for diffusion models. In *NeurIPS*, 2023.

Frantar, E. and Alistarh, D. Spdy: Accurate pruning with speedup guarantees. In *ICML*, 2022.

Frantar, E. and Alistarh, D. SparseGPT: Massive language models can be accurately pruned in one-shot. In *ICML*, 2023.

Fu, Y., Yang, H., Yuan, J., Li, M., Wan, C., Krishnamoorthi, R., Chandra, V., and Lin, Y. Depthshrinker: A new compression paradigm towards boosting real-hardware efficiency of compact neural networks. In *ICML*, 2022.

Girshick, R. Fast r-cnn. In *ICCV*, 2015.

Han, S., Pool, J., Tran, J., and Dally, W. Learning both weights and connections for efficient neural network. In *NeurIPS*, 2015.

Hashemizadeh, M., Ramirez, J., Sukumaran, R., Farnadi, G., Lacoste-Julien, S., and Gallego-Posada, J. Balancing act: Constraining disparate impact in sparse models. In *ICLR*, 2024.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *CVPR*, 2016.

He, Y., Lin, J., Liu, Z., Wang, H., Li, L.-J., and Han, S. Amc: Automl for model compression and acceleration on mobile devices. In *ECCV*, 2018.

Heusel, M., Ramsauer, H., Unterthiner, T., Nessler, B., and Hochreiter, S. Gans trained by a two time-scale update rule converge to a local nash equilibrium. In *NeurIPS*, 2017.

Hinton, G., Vinyals, O., and Dean, J. Distilling the knowledge in a neural network. In *NeurIPS-W*, 2014.

Ho, J., Jain, A., and Abbeel, P. Denoising diffusion probabilistic models. In *NeurIPS*, 2020.

Hooker, S., Moorosi, N., Clark, G., Bengio, S., and Denton, E. Characterising bias in compressed models. *arXiv preprint arXiv: 2010.03058*, 2020.

Hubara, I., Chmiel, B., Island, M., Banner, R., Naor, J., and Soudry, D. Accelerated sparse neural training: A provable and efficient method to find n:m transposable masks. In *NeurIPS*, 2021.

Ioffe, S. and Szegedy, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *ICML*, 2015.

Jordao, A., Lie, M., and Schwartz, W. R. Discriminative layer pruning for convolutional neural networks. *IEEE Journal of Selected Topics in Signal Processing*, 2020.

Kim, J., Jeong, Y., Lee, D., and Song, H. O. Efficient latency-aware cnn depth compression via two-stage dynamic programming. In *ICML*, 2023.

Krizhevsky, A. Learning multiple layers of features from tiny images. *Master's thesis, Department of Computer Science, University of Toronto*, 2009.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. Imagenet classification with deep convolutional neural networks. In *NeurIPS*, 2012.

Li, H., Kadav, A., Durdanovic, I., Samet, H., and Graf, H. P. Pruning filters for efficient convnets. In *ICLR*, 2017.

Lin, X.-Z., Kim, S., and Joo, J. Fairgrape: Fairness-aware gradient pruning method for face attribute classification. In *ECCV*, 2022.

Liu, Z., Mu, H., Zhang, X., Guo, Z., Yang, X., Cheng, K., and Sun, J. Metapruning: Meta learning for automatic neural network channel pruning. In *ICCV*, 2019.

Liu, Z., Mao, H., Wu, C.-Y., Feichtenhofer, C., Darrell, T., and Xie, S. A convnet for the 2020s. In *CVPR*, 2022.

Molchanov, P., Tyree, S., Karras, T., Aila, T., and Kautz, J. Pruning convolutional neural networks for resource efficient inference. *arXiv preprint arXiv:1611.06440*, 2016.

Molchanov, P., Mallya, A., Tyree, S., Frosio, I., and Kautz, J. Importance estimation for neural network pruning. In *CVPR*, June 2019.

Nichol, A. Q. and Dhariwal, P. Improved denoising diffusion probabilistic models. In *ICML*, 2021.

Paganini, M. Prune responsibly. *arXiv preprint arXiv: 2009.09936*, 2020.

Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., and Lerer, A. Automatic differentiation in pytorch. In *NeurIPS-W*, 2017.

Ronneberger, O., Fischer, P., and Brox, T. U-net: Convolutional networks for biomedical image segmentation. In *MICCAI*, 2015.

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., et al. Imagenet large scale visual recognition challenge. In *IJCV*, 2015.

Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., and Chen, L.-C. Mobilenetv2: Inverted residuals and linear bottlenecks. In *CVPR*, 2018.

Shen, M., Yin, H., Molchanov, P., Mao, L., Liu, J., and Alvarez, J. Structural pruning via latency-saliency knapsack. In *NeurIPS*, 2022.

Song, J., Meng, C., and Ermon, S. Denoising diffusion implicit models. In *ICLR*, 2021.

Tran, C., Fioretto, F., Kim, J.-E., and Naidu, R. Pruning has a disparate impact on model accuracy. In *NeurIPS*, 2022.

Vanholder, H. Efficient inference with tensorrt. In *GTC*, 2016.

Wightman, R., Touvron, H., and Jégou, H. Resnet strikes back: An improved training procedure in timm. In *NeurIPS-W*, 2021.

Wu, Y. and He, K. Group normalization. In *ECCV*, 2018.

A. Implementation Details

In this section, we provide the implementation details regarding the different types of layers and networks.

Normalization layers   Both ResNet-34 and MobileNetV2 utilize a batch normalization layer for normalization (Ioffe & Szegedy, 2015). At inference time, we fuse the batch normalization layer with the convolution layer, before merging the network. On the other hand, DDPM employs a group normalization layer for normalization (Wu & He, 2018). Unlike batch normalization, this layer cannot be fused with the convolution layer at inference time because it uses test-time feature map statistics. To address this, before fine-tuning, we move any group normalization layer between successive convolution layers that will be merged, to after these convolution layers. If there are multiple such normalization layers, we only keep the last one. This adjustment allows us to merge the consecutive convolution layers at inference time.

Strided convolutions, depthwise convolutions, and padding   For convolution layers that have a stride larger than 1, we restrict merging them when the kernel size of the following convolution layer is larger than 1, as done in Kim et al. (2023). We thus restrict the choice of A to include the activation layers between such layers. This is because merging the convolution layer with a stride larger than 1 significantly increases the merged kernel size. Concretely, merging the consecutive convolution layers $f_{\theta_2} \circ f_{\theta_1}$ results in the merged kernel size of

$$\mathrm{Ker}(\theta_{\mathrm{merged}}) = (\mathrm{Ker}(\theta_2) - 1) \times \mathrm{Str}(f_{\theta_1}) + \mathrm{Ker}(\theta_1),$$

where $\mathrm{Str}(\cdot)$ denotes the stride of the convolution layer (Fu et al., 2022).

It is worth mentioning that merging two depthwise convolution layers results in a single depthwise convolution layer. To account for this in networks that include depthwise convolution layers, we add a binary variable to the lookup table to indicate whether a layer is a depthwise convolution layer. We integrate this variable into the dynamic programming algorithm when we implement our method.

As noted in Kim et al. (2023), consecutive convolution layers that will be merged should not have any padding in between to avoid discrepancies at the boundary of the output before and after merging. To address this, before fine-tuning, we reorder the padding of the network to be applied prior to each such block of consecutive convolution layers. as done in Kim et al. (2023).

Skip-connections   There are two different types of skip-connection in a CNN: Skip-addition and skip-concatenation. Skip-addition, employed by ResNet and MobileNetV2, adds the output of an earlier layer directly to the output of a later layer. We can fuse skip-addition into the convolution layer if every intermediate convolution layer is merged into a single layer (Ding et al., 2021). We thus can merge across a skip-addition only if every convolution layer in between is merged into a single layer. Conversely, skip-concatenation, used in DDPM, concatenates the output of an earlier layer directly to the output of a later layer. We do not merge layers across a skip-concatenation.

MobileNetV2   In MobileNetV2, it is notable that there is no non-linear activation layer following each Inverted Residual Block (Sandler et al., 2018). Prior works on depth compression suggest that the performance of a compressed network can be improved by adding a non-linear activation layer after the merged layers (Fu et al., 2022; Kim et al., 2023). We also adapt this trick in our implementation on MobileNetV2 architecture.

DDPM   The DDPM architecture uses an upsampling module to increase the spatial dimension of the feature map, and it further employs a self-attention layer at the 16×16 resolution between the residual blocks (Chen et al., 2018b; Ho et al., 2020). We do not merge convolution layers across the self-attention layer or the upsampling layer in the DDPM architecture. It is also worth mentioning that the DDPM architecture has a 3×3 convolution layer with stride 1 between the upsampling layer and the skip concatenation. We include these convolution layers as potential pruning candidates in our algorithm, as this leads to improved performance of the compressed network. When measuring $\mathrm{Perf}(\cdot)$ using negative diffusion loss in DDPM, we further divide this value by the diffusion loss of the pre-trained network since it leads to more stable result.

B. Proof

In this section, we prove Theorem 3.1, restated here for convenience.

Theorem 3.1. *The solution A^* and $(k_i^*)_{i=1}^{|A^*|+1}$ given by Algorithm 1 is an optimal solution of Problem (5).*

Proof. We prove this by induction. Suppose that for $l < l_0$ and $t < t_0$, $A[l, t]$ and $k_i^{(lt)} := C[l, t] \cap (a_{i-1}^{(lt)}, a_i^{(lt)}]$ are the optimal solution of Problem (6), where $(a_i^{(lt)})_{i=1}^{|A[l,t]|}$ denotes the element of $A[l, t]$ in ascending order.

Assume that $A[l_0, t_0]$ and $k_i^{(l_0 t_0)}$ from Equation (7) is not the optimal solution of Problem (6) when $l = l_0, t = t_0$. We denote the optimal solution of Problem (6) for $l = l_0, t = t_0$ as $\widehat{A}$ and $\hat{k}_i$.

$$\sum_{i=1}^{|\widehat{A}|+1} I\big[\hat{a}_{i-1}, \hat{a}_i, \hat{k}_i\big] > \sum_{i=1}^{|A[l_0,t_0]|+1} I\big[a_{i-1}^{(l_0 t_0)}, a_i^{(l_0 t_0)}, k_i^{(l_0 t_0)}\big], \tag{9}$$

$$\sum_{i=1}^{|\widehat{A}|+1} T\big[\hat{a}_{i-1}, \hat{a}_i, \hat{k}_i\big] < t_0. \tag{10}$$

We first show that $\widehat{A}$ is not empty because

$$\sum_{i=1}^{|\widehat{A}|+1} I\big[\hat{a}_{i-1}, \hat{a}_i, \hat{k}_i\big]$$

$$> \sum_{i=1}^{|A[l_0,t_0]|+1} I\big[a_{i-1}^{(l_0 t_0)}, a_i^{(l_0 t_0)}, k_i^{(l_0 t_0)}\big] = M[l_0, t_0] \qquad \text{(from the assumption \& definition of } A[l_0, t_0])$$

$$= \max_{0 \le l' < l_0, k \in K_{l' l_0}} \big(M[l', t - T[l', l_0, k]] + I[l', l_0, k]\big) \qquad \text{(from the recurrence relation Equation (7))}$$

$$\ge \max_{k \in K_{0 l_0}} I[0, l_0, k].$$

Now, let $\hat{l} := \hat{a}_{|\widehat{A}|}$ be the maximum value of $\widehat{A}$. Then,

$$\sum_{i=1}^{|\widehat{A}|+1} I\big[\hat{a}_{i-1}, \hat{a}_i, \hat{k}_i\big]$$

$$= \sum_{i=1}^{|\widehat{A}|} I\big[\hat{a}_{i-1}, \hat{a}_i, \hat{k}_i\big] + I[\hat{l}, l_0, \hat{k}_{|\widehat{A}|+1}]$$

$$\le M[\hat{l}, t_0 - T[\hat{l}, l_0, \hat{k}_{|\widehat{A}|+1}]] + I[\hat{l}, l_0, \hat{k}_{|\widehat{A}|+1}] \qquad \text{(from the optimality assumption)}$$

$$\le M[l_0, t_0] = \sum_{i=1}^{|A[l_0,t_0]|+1} I\big[a_{i-1}^{(l_0 t_0)}, a_i^{(l_0 t_0)}, k_i^{(l_0 t_0)}\big], \qquad \text{(from the recurrence relation Equation (7))}$$

which contradicts to Equation (9). Note that the inequality with from the optimality assumption holds because we assumed the optimality for $l = \hat{l} < l_0$ and $t = t_0 - T[\hat{l}, l_0, \hat{k}_{|\widehat{A}|+1}] < t_0$, and from Equation (10), we have

$$\sum_{i=1}^{|\widehat{A}|} T\big[\hat{a}_{i-1}, \hat{a}_i, \hat{k}_i\big] < t_0 - T[\hat{l}, l_0, \hat{k}_{|\widehat{A}|+1}],$$

which satisfies the constraint in Problem (6). Therefore, from the contradiction we have that $A[l_0, t_0]$ and $k_i^{(l_0 t_0)}$ are indeed optimal solutions of Problem (6) ($l = l_0$ and $t = t_0$).

For the base case $l = 0$ and $t = \frac{T_0}{P}$, $A[l, t] = \emptyset$ is indeed the solution of Problem (6).

Now, plugging in $l = L$ and $t = T_0$ proves the theorem. □

Table 7: Wall-clock time for constructing the importance and latency look-up tables. GPU hours for constructing the importance table is measured in RTX3090 and the latency table is measured in RTX2080 Ti.

Network	Dataset	Importance table (GPU Hours)	Latency table (GPU Hours)	# of table entries
ResNet-34	ImageNet	4.4 hours	25.9 minutes	150
MobileNetV2-1.0	ImageNet	13.2 hours	6.2 minutes	391
MobileNetV2-1.4	ImageNet	15.0 hours	10.6 minutes	391
DDPM	CIFAR10	2.5 hours	1.3 minutes	98

Table 8: Wall-clock time for constructing the importance look-up table for ResNet-34 and MobileNetV2-1.0 in different methods. GPU hours for constructing the importance table is measured in RTX3090.

(a) ResNet-34 on ImageNet dataset.

Method	GPU Hours	# of table entries
Depth (Kim et al., 2023)	25.8 hours	62
LayerOnly (Ours)	0.8 hours	29
LayerMerge (Ours)	4.4 hours	150

(b) MobileNetV2-1.0 on ImageNet dataset.

Method	GPU Hours	# of table entries
Depth (Kim et al., 2023)	126.0 hours	315
LayerOnly (Ours)	0.4 hours	13
LayerMerge (Ours)	13.2 hours	391

C. Details on Constructing Importance and Latency Tables

In this section, we provide details on how we measure the importance and latency values for the lookup tables I and T, along with their corresponding practical computation cost for different types of networks.

Importance measurements We defined importance values in Equation (4). Recall that the first term is estimated with the performance of the network after fine-tuning for a few steps. For that, we select a random subset of the training dataset for fine-tuning, then we evaluate performance on another separate subset, also drawn from the training dataset. For the second term, we evaluate the performance of the pre-trained network on the separate subset. In particular, we use a fine-tuning subset of size 4% of the total training dataset size for ImageNet, and 1% for CIFAR10. The separate subset is also the same size as the fine-tuning subset. We fine-tune the network for 1 epoch for ImageNet, and 50 epochs for CIFAR10 with the fine-tuning subset when we measure the importance.

We report the wall-clock time for constructing the importance look-up table in Table 7. It is worth mentioning that this computation can be done in an embarrassingly parallel fashion without any communication between GPUs, which allows for significant speedup if multiple GPUs are available. For instance, the importance table for MobileNetV2-1.0 only took 33 minutes with 24 GPUs.

In Table 8, we further compare the wall-clock time required to construct the importance look-up tables used in LayerMerge and LayerOnly to the one used in Depth (Kim et al., 2023). It is worth noting that the Depth baseline fine-tunes for one epoch over the full training dataset to evaluate each table entry. However, we observe that fine-tuning for one epoch using only a small random subset of the training dataset is sufficient for estimating the importance values. This significantly reduces the wall-clock time required to construct the look-up table in our method compared to the Depth baseline.

Latency measurements To measure each latency value in the lookup table T, we measure the inference time on PyTorch by first warming up the GPU for 300 forward passes, then averaging the inference time over the subsequent 200 forward passes, with a batch size of 128. Latency values are measured in milliseconds. We report the wall-clock time for constructing the latency look-up table in Table 7.

Recall that to solve the surrogate problem, we first need to discretize the latency values. We do that by multiplying the real valued latencies in the lookup table and T_0 by 10, then rounding them down to the nearest integer. Note that this is equivalent to choosing the discretization level as $P = 10T_0$.

Table 9: Accuracy and latency speed-up of applying compression methods to MobileNetV2-1.0 on ImageNet dataset with 90, 30, and 20 fine-tuning epochs. The latency speed-up is measured on RTX2080 Ti GPU at batch size 128.

(a) Results of fine-tuning for 90 epochs.

Network	Acc (%) ↑	PyTorch Speed-up ↑	TensorRT Speed-up ↑
MobileNetV2-1.0	72.89	1.00×	1.00×
AMC-70% (He et al., 2018)	71.66	1.32×	1.34×
Depth-74% (Kim et al., 2023)	72.49	1.62×	**1.42×**
LayerOnly-73% (Ours)	69.29	1.30×	1.35×
LayerMerge-55% (Ours)	**72.73**	**1.63×**	**1.42×**

(b) Results of fine-tuning for 30 epochs.

Network	Acc (%) ↑	PyTorch Speed-up ↑	TensorRT Speed-up ↑
MobileNetV2-1.0	72.89	1.00×	1.00×
AMC-70% (He et al., 2018)	71.05	1.32×	1.34×
Depth-74% (Kim et al., 2023)	71.59	1.62×	**1.42×**
LayerOnly-73% (Ours)	67.60	1.30×	1.35×
LayerMerge-55% (Ours)	**72.06**	**1.63×**	**1.42×**

(c) Results of fine-tuning for 20 epochs.

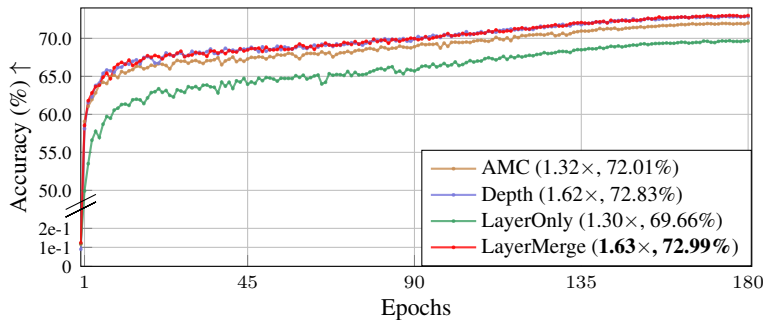| Network	Acc (%) ↑	PyTorch Speed-up ↑	TensorRT Speed-up ↑
MobileNetV2-1.0	72.89	1.00×	1.00×
AMC-70% (He et al., 2018)	70.64	1.32×	1.34×
Depth-74% (Kim et al., 2023)	71.09	1.62×	**1.42×**
LayerOnly-73% (Ours)	66.96	1.30×	1.35×
LayerMerge-55% (Ours)	**71.59**	**1.63×**	**1.42×**

Figure 3: Test accuracy recovery curve of different compression methods across fine-tuning epochs. We indicate the associated speed-up and accuracy after fine-tuning in the parentheses. The inference time is measured on RTX2080 Ti GPU at batch size 128 in PyTorch format.

D. Details on Ablation Studies

In this section, we outline the details of the sequential optimization baseline (*Depth → LayerOnly*) presented in Table 6. Recall that for MobileNetV2, we fine-tune every compressed network for 180 epochs, using the same fine-tuning recipe as Kim et al. (2023). For the sequential optimization baseline, we divide the fine-tuning epochs equally between the two pruning methods, i.e., we fine-tune for 90 epochs after each pruning method, again using the same fine-tuning recipe.

We fix the latency budget ratio $p\%$ for Depth to 74%, which yields a speed-up of $1.63\times$. Then we use two different values of p for LayerOnly: 72% and 64%. Note that p here corresponds to $T_0/T_{\text{depth-pruned}}$, where T_0 is the chosen latency budget of the final pruned model and $T_{\text{depth-pruned}}$ is the latency of the model pruned using Depth-74%. It is worth noting that the allocations of fine-tuning epochs and compression ratios between the two pruning methods are hyperparameters that need to be tuned. However, our method is free from these hyperparameters due to the joint optimization on the two types of pruning modalities.

E. Additional Experiments

Results with smaller fine-tuning epochs   In this section, we study the effect of fine-tuning for a shorter time. In particular, we present in Table 9 compression results on MobileNetV2-1.0 where we fine-tune all methods for 90, 30, and 20 epochs, using cosine learning rate decay. We further plot in Figure 3 the recovery curve of test accuracy across fine-tuning steps. Our method consistently outperforms baselines under these smaller fine-tuning budgets as well.

Comparison with knowledge distillation   In this section, we compare our method to the knowledge distillation method of Hinton et al. (2014). For that, we use a smaller version of MobileNetV2 (Sandler et al., 2018) than the one used for the pre-trained network as the student network and train it for the same number of epochs we use for fine-tuning in our method (180 epochs). We present the results in Table 10 and plot the recovery curve of test accuracy across fine-tuning steps in Figure 4. The key benefit of pruning methods like ours over knowledge distillation is that they only require fine-tuning the model, while knowledge distillation requires training the small model from scratch. This provides an advantage when both methods are compared under an identical training budget. Indeed our results show that our method outperforms the knowledge distillation method in this setting.

Table 10: Accuracy and latency speed-up comparison between knowledge distillation (Hinton et al., 2014) and our method with MobileNetV2-1.0 and MobileNetV2-1.4 on ImageNet dataset. The latency speed-up is measured on RTX2080 Ti GPU at batch size 128.

(a) MobileNetV2-1.0 on ImageNet dataset.

Network	Acc (%) ↑	PyTorch Speed-up ↑	TensorRT Speed-up ↑
MobileNetV2-1.0	72.89	1.00×	1.00×
Knowledge distillation (MobileNetV2-0.75)	69.69	1.17×	1.20×
LayerMerge-55% (Ours)	**72.99**	**1.63×**	**1.42×**

(b) MobileNetV2-1.4 on ImageNet dataset.

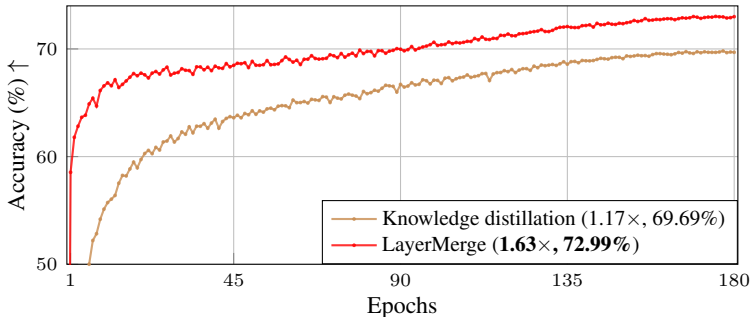| Network	Acc (%) ↑	PyTorch Speed-up ↑	TensorRT Speed-up ↑
MobileNetV2-1.4	76.28	1.00×	1.00×
Knowledge distillation (MobileNetV2-1.0)	72.30	1.51×	1.54×
LayerMerge-43% (Ours)	**74.91**	**1.99×**	**1.61×**

Figure 4: Test accuracy recovery curve of our method compared to knowledge distillation across fine-tuning epochs for MobileNetV2-1.0. We indicate the associated speed-up and the accuracy after fine-tuning in the parentheses. The inference time on RTX2080 Ti GPU at batch size 128 in PyTorch format.

Applying knowledge distillation during fine-tuning　It is worth noting that knowledge distillation methods can be jointly applied with pruning methods, by considering the pre-trained network as the teacher network and the pruned network as the student network to train. We present in Table 11 the results of applying the knowledge distillation method of Hinton et al. (2014) to different pruning methods. We observe that applying knowledge distillation further improves the accuracy of the pruned network (compared to Table 2), and our method consistently outperforms the baselines in this setting as well.

Additional compression results　In this section, we present additional results comparing various compression methods across different network architectures (ResNet-34, MobileNetV2-1.0, MobileNetV2-1.4, and DDPM) and compression ratios. We display the Pareto curves for each method on the different architectures in Figure 5. We report the latency speed-up measured in PyTorch format. For the ResNet and MobileNetV2 architectures, we plot accuracy against speed-up, and for the DDPM architecture, we plot the FID metric against the latency speed-up.

Table 11: Accuracy and latency speed-up of applying knowledge distillation (Hinton et al., 2014) with compression methods to MobileNetV2-1.0 on ImageNet dataset. The latency speed-up is measured on RTX2080 Ti GPU at batch size 128.

Network	Acc (%) ↑	PyTorch Speed-up ↑	TensorRT Speed-up ↑
MobileNetV2-1.0	72.89	1.00×	1.00×
AMC-70% (He et al., 2018)	72.04	1.32×	1.34×
Depth-74% (Kim et al., 2023)	72.99	1.62×	**1.42×**
LayerOnly-73% (Ours)	69.70	1.30×	1.35×
LayerMerge-55% (Ours)	**73.14**	**1.63×**	**1.42×**
Depth-66% (Kim et al., 2023)	72.31	1.88×	1.57×
LayerMerge-46% (Ours)	**72.56**	**1.90×**	**1.65×**
Depth-59% (Kim et al., 2023)	71.76	2.07×	1.79×
LayerMerge-38% (Ours)	**72.06**	**2.18×**	**1.84×**
Depth-53% (Kim et al., 2023)	70.81	2.47×	1.97×
LayerMerge-33% (Ours)	**71.32**	**2.49×**	**2.05×**

(a) ResNet-34 on ImageNet dataset.

(b) MobileNetV2-1.0 on ImageNet dataset.

(c) MobileNetV2-1.4 on ImageNet dataset.

(d) DDPM on CIFAR10 dataset.
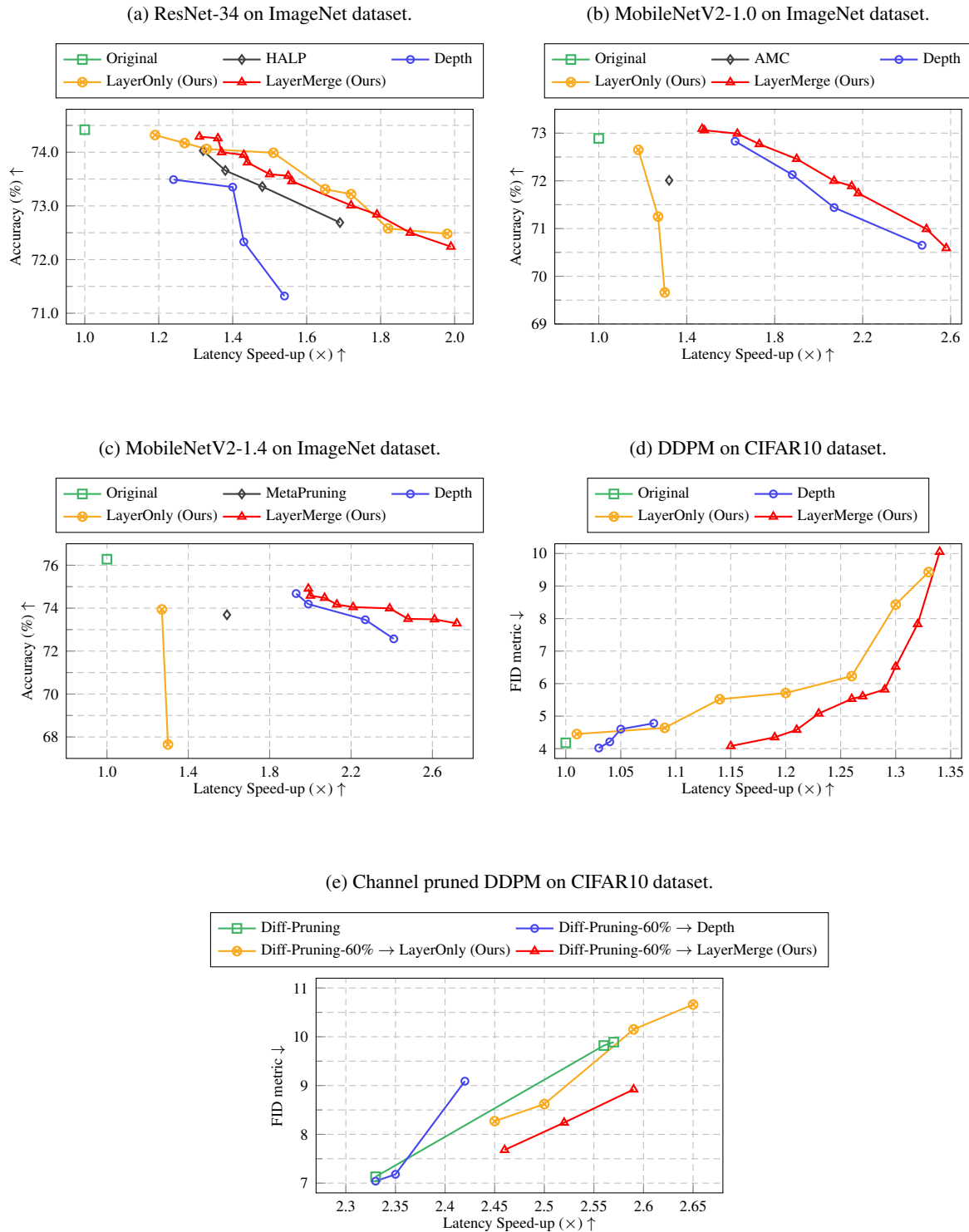
(e) Channel pruned DDPM on CIFAR10 dataset.

Figure 5: Pareto curve of each compression method applied to each network. The latency speed-up is measured on RTX2080 Ti GPU in PyTorch format, with batch size of 128 for ImageNet dataset and batch size of 64 for CIFAR10 dataset.