A. Training Details

Our training receipt follows most of the techniques used in DeiT [32]. In Table 5, we provide the default setting for compression rate searching. In this case, we initialize the backbone with their official pre-trained models, fix the network parameters, and train the proposed Differentiable Discrete Proxy for 3 epochs. In the first epoch, we optionally set λ_f in Eq. (13) to 0 for warm-up. To fine-tune the compressed models, we fix the compression rate and update the network parameters for 30 epochs. The default settings for fine-tuning are provided in Table 6. During compression rate searching, we employ attention masking to simulate token dropping, while in the fine-tuning process, we directly drop the redundant tokens. Any differences from the default DeiT recipe are highlighted in **bold** in the tables.

Table 5: Compression rate searching training settings.

cofig	value
optimizer	AdamW
learning rate	**0.01**
minimal learning rate	**0.001**
learning rate schedule	cosine decay
weight decay	**0**
batch size	1024
training epochs	**3**
augmentation	RandAug(9, 0.5)
LabelSmoth	0.1
DropPath	0.1
Mixup	0.8
CutMix	1.0

Table 6: Fine-tuning training settings.

cofig	value
optimizer	AdamW
learning rate	**2e-5**
minimal learning rate	**1e-6**
learning rate schedule	cosine decay
weight decay	0.05
batch size	1024
training epochs	**30**
augmentation	RandAug(9, 0.5)
LabelSmoth	0.1
DropPath	0.1
Mixup	0.8
CutMix	1.0

B. Computational Cost Constraint Details

This section provides more details about the computational cost constraint of DiffRate, including traditional FLOPs, and hardware-aware metrics like latency and power consumption.

B.1. FLOPs Calculation

Algorithm 2 FLOPs Calculation.

Input: block-wise pruning compression rate $\boldsymbol{\alpha}_p = \{\alpha_p^l\}_{l=1}^L$ and merging compression rate $\boldsymbol{\alpha}_m = \{\alpha_m^l\}_{l=1}^L$, embedding size C.
Output: FLOPs $\mathcal{F}(\boldsymbol{\alpha}_p, \boldsymbol{\alpha}_m)$.

1: $\alpha^0 = 0$
2: $\mathcal{F}(\boldsymbol{\alpha}_p, \boldsymbol{\alpha}_m) = 0$ ▷ FLOPs
3: **for** l=1 to L **do**
4: $\quad \mathcal{F}(\boldsymbol{\alpha}_p, \boldsymbol{\alpha}_m)$ += $4NC^2 + 2N^2C$ ▷ Attention
5: $\quad \alpha^l = max(\alpha^{l-1}, \alpha_p^l, \alpha_m^l)$
6: $\quad N = N(1 - \alpha^l)$
7: $\quad \mathcal{F}(\boldsymbol{\alpha}_p, \boldsymbol{\alpha}_m)$ += $8NC^2$ ▷ MLP
8: **end for**
9: **return** $\mathcal{F}(\boldsymbol{\alpha}_p, \boldsymbol{\alpha}_m)$

By utilizing the compression rate obtained from our proposed DiffRate, we can calculate the corresponding FLOPs $\mathcal{F}(\boldsymbol{\alpha}_p, \boldsymbol{\alpha}_m)$ using Algorithm 2. The final FLOPs calculation also includes the patch embedding layer and classifier, which are excluded from Algorithm 2 for clarity. Additionally, we utilize the straight-through estimator (STE) for backpropagation in the max operation of Line 5.

Table 7: **Gemmini Search Space.** The Gemmini search space is determined by the number of tiles/meshes in each row and column, which indicates its computational resources, while the bank number/capacity of the scratchpad memory and accumulator determines its memory resources. The buswidth sets an upper limit on the communication speed between the scratchpad memory and computation modules.

parameters	type	search space
Tiles in a row	int	1,2,4,8
Tiles in a column	int	1,2,4,8
Meshes in a row	int	4,8,16,32
Meshes in a column	int	4,8,16,32
Buswidth (bit)	int	64,128,256,512
Bank number of scratchpad memory	int	1,2,4,8,16
Capacity of scratchpad memory (MB)	int	0.25,0.5,1,2,4
Capacity of accumulator (KB)	int	64,128,256,512,1024

B.2. Latency and Power Constraint

Apart from considering FLOPs, we can incorporate a differentiable method that accounts for hardware (HW) performance metrics, including latency and power consumption. Both latency and power consumption can be constrained in a similar manner. Therefore, we utilize the notation $\mathcal{L}_{hw}$

to represent both $\mathcal{L}_{la}$ and $\mathcal{L}_{pw}$ in Eq. (15) of the main text. The loss function of a hardware metric is given by:

$$\mathcal{L}_{hw} = \log(\cosh(E_{hw}(\boldsymbol{\alpha}_p, \boldsymbol{\alpha}_m, \boldsymbol{\beta}) - T_{hw})) \quad (16)$$

where $\mathcal{L}_{hw}$ represents the corresponding HW performance constraint loss, E_{hw} is the HW performance with HW parameters $\boldsymbol{\beta}$ and compression rates $\boldsymbol{\alpha}_p, \boldsymbol{\alpha}_m$, T_{hw} is the given target HW performance.

To find the optimal solution that considers both accuracy and HW performance metrics, we can co-explore the design space of compression ratio and HW simultaneously. We use Gemmini [10], a deep-learning accelerator generation framework that can produce a wide range of realistic accelerators from a flexible architecture template. The HW parameters $\boldsymbol{\beta}$ are provided in Table 7.

We adopt iterative training method to optimize compression rates and HW parameters. To be specific, we first optimize compression rate given HW parameters, and then optimize HW parameters given compression rates. To make the loss of hardware metric differentiable w.r.t. compression rates, we formulate E_{hw} as

$$E^{\alpha}_{hw} = \sum_{l=1}^{L} (\alpha^l + \mathrm{SG}(1 - \alpha^l)) \mathcal{F}'(\alpha^l, \boldsymbol{\beta}^*) \quad (17)$$

where $\alpha^l = max(\alpha^{l-1}, \alpha^l_p, \alpha^l_m)$, and $SG(\cdot)$ is the operator of stopping gradient. $\mathcal{F}'$ denotes the HW cost model, which can calculate a hardware metric given compression rate and HW parameters. '*' indicates that the HW parameters are fixed.

On the other hand, to make hardware metric differentiable w.r.t. compression rates, we formulate E_{hw} as

$$E^{\beta}_{hw} = \sum_{h=1}^{H} \sum_{l=1}^{L} (\beta_h + \mathrm{SG}(1 - \beta_h)) \mathcal{F}'(\alpha^{l*}, \boldsymbol{\beta}) \quad (18)$$

where $\beta_h = \mathrm{GS}(\pi_h)$, π_h is learnable parameter for h-th HW parameter, and GS represents Gumbel-Softmax function. Here '*' indicates that the compression rates are fixed.

With Eqn.(17) and Eqn.(18), we can update the compression rate and HW designs in a differentiable manner to satisfy the target latency and power consumption.

C. More Results

In this section, we present detailed results from our experiments. Firstly, we present the results obtained across different FLOPs constraints for the DeiT [32] in Sec. C.1 and for the MAE [13] in Sec. C.2. In Sec. C.3, we provide an detailed of the compression schedule that we searched. Additionally, we investigate the transferability of the searched compression schedules across different models in Sec. C.4. Finally, we demonstrate the effectiveness of the compressed models by training them from scratch with faster training speeds in Sec. C.5.

C.1. DeiT Models

Table 8 displays the comprehensive results of DeiT [32], encompassing both off-the-shelf models and fine-tuned models.

Table 8: **Full DeiT Results.** FT denotes fine-tuning the compressed model for 30 epochs. Gray denotes the official un-compressed pre-trained models (Baseline).

Model	FLOPs(G)	Acc.(%) w/o FT	Acc.(%) w/ FT
ViT-T (DeiT)	1.3	72.13	-
	0.6	70.36	71.11
	0.7	71.16	71.70
	0.8	71.74	72.18
	0.9	71.91	72.39
	1.0	72.12	72.46
ViT-S (DeiT)	4.6	79.82	-
	2.3	78.74	79.39
	2.5	79.09	79.61
	2.7	79.40	79.64
	2.9	79.58	79.83
	3.1	79.71	79.90
ViT-B (DeiT)	17.6	81.82	-
	8.7	78.98	80.61
	10.0	80.63	81.17
	10.4	80.97	81.30
	11.5	81.50	81.59
	12.5	81.75	81.80

C.2. MAE Models

Also, Table 9 presents the complete outcomes of MAE [13], encompassing both off-the-shelf models and fine-tuned models. We do not fine-tune ViT-H due to the constraints of computational resources.

C.3. Seached Compression Schedule

Table 10 presents the detailed compression schedule that we searched. Notably, in contrast to DeiT models, MAE models tend to retain more tokens in the deep block. This is because DeiT classifies solely based on the class token, while MAE classifies based on the average of all image tokens.

C.4. Compression Schedule Transfer

The number of blocks in ViT-T, ViT-S, and ViT-B is 12, facilitating the transferability of their block-wise compression rates. To investigate this, we transferred the compression rates obtained from ViT-T and ViT-B to ViT-S, as illustrated in Fig. 6. We can observe that the compression rate attained by ViT-S itself was optimal, whereas the transfer of compression rate from ViT-T to ViT-S resulted in

Table 9: **Full MAE Results.** FT denotes fine-tuning the compressed model for 30 epochs. Gray denotes the official un-compressed pre-trained models (Baseline).

Model	FLOPs(G)	Acc.(%) w/o FT	Acc.(%) w/ FT
ViT-B (MAE)	17.6	83.72	-
	8.7	79.96	81.89
	10.0	81.87	82.65
	10.4	82.07	82.83
	11.5	82.91	83.19
ViT-L (MAE)	61.6	85.95	-
	31.0	84.65	85.31
	34.7	85.19	85.45
	38.5	85.45	85.61
	42.3	85.56	85.63
	46.1	85.76	85.84
ViT-H (MAE)	167.4	86.88	-
	83.7	86.15	-
	93.2	86.40	-
	103.4	86.72	-
	124.5	86.77	-

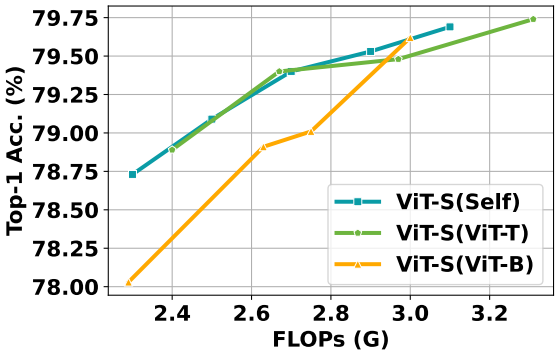

Figure 6: **Compression rate transfer**. Transferring the compression rate of ViT-B(DeiT) and ViT-T(DeiT) to ViT-S(DeiT). Self indicates the compression rate learn in ViT-S(DeiT).

a similar outcome. However, the transfer of compression rate from ViT-B to ViT-S leads to significantly poorer performance. This experiment verifies the ability of our proposed DiffRate to learn block-wise compression rates suitable for different network structures based on their features. Furthermore, it highlights that compression rates are somewhat transferable among similar network structures, such as transferring the compression rate from ViT-T to ViT-S.

C.5. Train from Scratch

The compressed model also has the capability to train from scratch using the searched compression rate. In this scenario, redundant tokens are directly eliminated, resulting in a faster training speed. As presented in Table 11, DiffRate yields a $1.4\times$ increase in training speed with only a -0.06% performance degradation.

D. More Visualization

We utilize the approach proposed by ToMe [1] to generate visualizations of the merging results. Specifically, we map each merged and pruned token back to its original input patch. To visualize merged tokens, we assign each input patch the average color of the merged tokens it belongs to and apply a random border color to distinguish tokens. For pruned tokens, we color their corresponding input patches black. Fig. 7 provides additional examples of token compression on images, extending the visualizations shown in Fig. 5. Our proposed DiffRate approach effectively identifies semantic objects, removes semantically irrelevant background tokens, and merges less-discriminative tokens in the foreground. By combining the advantages of pruning and merging through learnable compression rates, DiffRate can reduce the token count with minimal loss of information.

E. Overhead of DiffRate

In this section, we offer the analysis about computational overhead introduced by our proposed DiffRate framework. For a given number of input tokens, denoted by N, and a total of L blocks, the reparameterization trick introduces $2NL$ parameters and $\frac{(N^2+5N)L}{2}$ FLOPs. For example, we consider DeiT-S with $N = 196$ and $L = 12$, which results in only 4.7k additional parameters and 0.24M additional FLOPs. Furthermore, we evaluated the FLOPs of the DiffRate module within one block, taking into account the number of input tokens (N), the embedding size (C), and the number of merging tokens (N_m). The FLOPs of the DiffRate module within a block can be approximated as $N \log N + (N - N_m + 1)N_m C$. For instance, in the case of one DeiT-S block with $N = 196$, $C = 384$, and $N_m = 20$, the FLOPs of the DiffRate module amount to 1.36M. Overall, the overhead of DiffRate framework is negligible compared to the original 22M parameters and 4.6G FLOPs for DeiT-S.

F. Extending to Hierarchical Architecture

In addition to the standard ViT [7], various ViT variants have emerged, including EfficientFormer [22], Swin Transformer [26], CAFormer [41], among others. These modern ViT variants commonly employ a hierarchical architecture, dividing the network into multiple stages and progressively reducing the resolution of the feature map. Within hierarchical architectures, downsampling operations, such as convolution or pooling, are used to achieve the reduction in resolution. These operations rely on maintaining the spatial in-

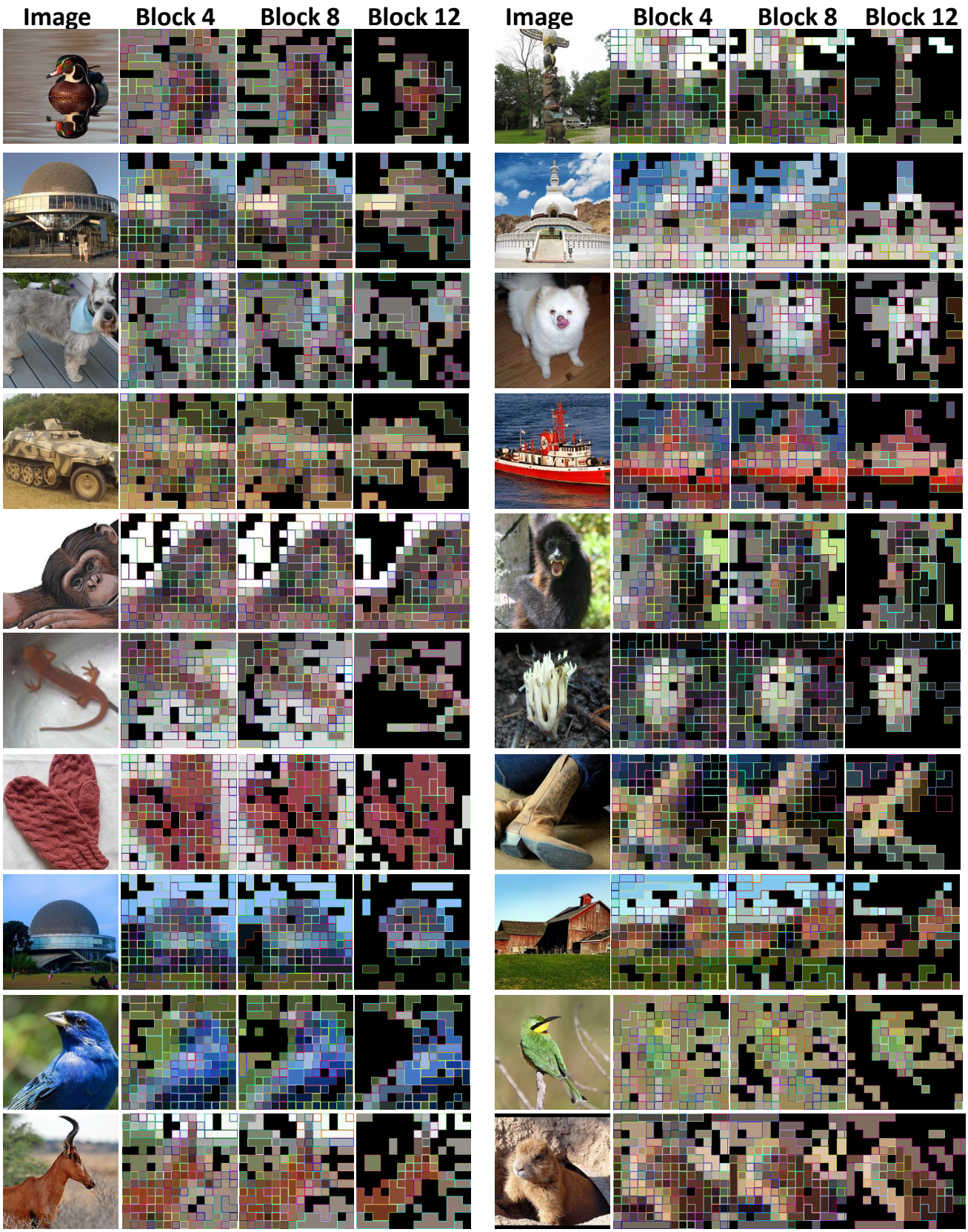

Figure 7: **More visualization.** Continuation of Fig 5.

Table 10: **Searched Compression Schedule.** We provide the block-wise kept tokens number for pruning and merging, respectively.

Model	FLOPs(G)	Compression Schedule Prune & Merge
ViT-T (DeiT)	0.6	[197,196,180,157,130,107,86,73,63,51,40,3] [197,187,167,139,114,90,76,66,57,45,37,3]
	0.7	[197,196,194,180,150,123,98,82,68,60,52,3] [197,196,192,158,133,103,88,72,64,58,49,3]
	0.8	[197,197,196,186,166,147,117,103,92,80,74,3] [197,196,190,172,154,125,107,96,84,78,70,3]
	0.9	[197,197,196,190,182,166,141,125,113,105,99,3] [197,196,194,188,172,147,129,115,107,103,96,3]
	1.0	[197,197,196,194,188,184,178,164,156,137,129,3] [197,197,196,190,186,154,148,156,145,135,125,3]
ViT-S (DeiT)	2.3	[197,192,168,143,121,105,92,74,62,45,33,3] [197,180,156,127,109,98,80,66,52,37,31,3]
	2.5	[197,192,174,156,131,115,111,99,76,50,39,3] [197,186,160,133,119,107,96,82,66,43,37,3]
	2.7	[197,196,182,160,141,127,115,105,88,66,49,3] [197,188,170,147,131,119,109,96,76,54,45,3]
	2.9	[197,196,190,168,150,139,129,117,99,78,58,3] [197,194,176,156,141,133,121,107,88,64,56,3]
	3.1	[197,197,194,180,160,147,137,129,115,92,76,3] [197,196,186,164,150,141,133,121,103,78,68,3]
ViT-B (DeiT)	8.7	[197,192,172,156,131,107,90,72,56,33,17,3] [197,178,162,141,115,96,78,60,47,21,13,3]
	10.0	[197,194,182,168,148,129,113,101,82,49,25,3] [197,186,174,156,137,117,105,90,66,31,19,3]
	10.4	[197,196,188,174,156,141,121,103,90,58,25,3] [197,190,184,164,145,131,107,94,74,31,21,3]
	11.5	[197,197,197,188,170,154,139,123,107,72,43,3] [197,197,192,178,158,145,127,111,94,50,35,3]
	12.5	[197,197,196,192,184,170,154,139,129,94,58,3] [197,196,194,190,176,160,145,131,115,68,47,3]
ViT-B (MAE)	8.7	[197,192,166,143,119,103,86,72,54,43,37,23] [197,176,150,127,107,92,76,58,47,41,23,23]
	10.0	[197,194,173,154,135,120,108,92,76,64,52,40] [197,181,160,141,122,116,95,80,67,60,40,40]
	10.4	[197,195,179,159,140,122,105,100,89,71,56,45] [197,187,166,145,125,109,102,97,75,64,45,45]
	11.5	[197,196,182,172,162,147,131,109,101,94,70,49] [197,192,178,164,150,133,115,101,96,82,49,49]

Table 11: **Train from scratch.** Train compressed ViT-S (DeiT) from scratch with the official DeiT training receipt.

Model	FLOPs	Throughput	Train Speed	Acc.
ViT-S (DeiT)	4.6	5039	1×	79.82
DiffRate	2.3	8901	1.8×	79.41
DiffRate	2.9	6744	1.4×	79.76

tegrity of the feature map. However, traditional token compression operations disrupt the spatial integrity by reducing the number of tokens in an unstructured manner. As a result, conventional token compression techniques cannot be directly applied to hierarchical architectures. To overcome this challenge, we propose a token uncompression module (see Fig.,8) that restores the compressed token sequence by copying tokens based on their relationships, inspired by the

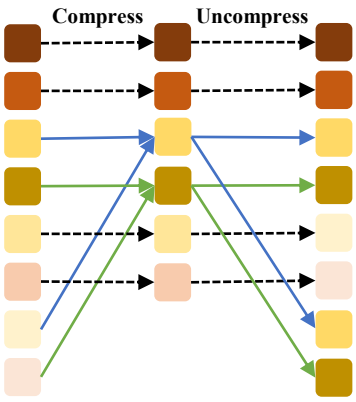

Figure 8: An example of token uncompression. We only consider token merging here.

Table 12: Appling DiffRate to CAFormer-S36 on ImageNet-1k.

Method	FLOPs(G)	Top-1 Acc.(%)	
		w/o tuning	w/ tuning
Baseline	8.0	84.45	-
DiffRate	6.0	84.21	84.32
	5.6	83.92	84.21
	5.2	83.49	84.03

Table 13: Downstream semantic segmentation task using Semantic FPN with CAFormer-S36 backbone on ADE20K dataset. FLOPs is calculated under the input scale of 512×704.

Backbone	FLOPs (G)	mIoU (%)	mAcc (%)
CAFormer-S36	77.2	41.05	51.50
w/ DiffRate	54.5	40.88	51.32

approaches in ToMeSD [2] and TCFormer [43]. By incorporating this module at the end of each stage, we can adapt our proposed DiffRate method to hierarchical architectures. Specifically, we apply DiffRate to the CAFormer [41], a state-of-the-art ViT variant with four stages. DiffRate is applied only to the third stage, as the first two stages consist of convolution blocks, and the last stage incurs minimal computational cost (6% in CAFormer-S36). As shown in Table 12, DiffRate achieves a significant 25% reduction in FLOPs with a marginal sacrifice of 0.24% in accuracy without additional fine-tuning. Moreover, when fine-tuned, DiffRate achieves a remarkable 35% reduction in FLOPs with a minimal accuracy drop of only 0.42%.

G. Extending to Downstream Tasks

By incorporating the introduced uncompression module in Fig. 8, DiffRate can also be applied to the model for downstream tasks. We begin by transferring the uncompressed pre-trained CAFormer-S36 model to ADE20K [44], following the settings of PVT [33]. Subsequently, DiffRate is applied to the third stage of the trained segmentation model without further fine-tuning. The results in Table,13 demonstrate that DiffRate achieves a significant 30% reduction in FLOPs with negligible performance degradation.