Supplementary Materials of Enhanced Sparsification via Stimulative Training

Shengji Tang*[1], Weihao Lin*[1], Hancheng Ye[3],
Peng Ye[1], Chong Yu[2], Baopu Li[4], and Tao Chen**[1]

[1] School of Information Science and Technology, Fudan University, Shanghai, China
[2] Academy for Engineering and Technology, Fudan University, Shanghai, China
[3] Shanghai AI Laboratory, Shanghai, China
[4] Independent Researcher
eetchen@fudan.edu.cn

Appendix A: Implementation Details

In this section, we introduce the detailed training settings of experiments in the main manuscript. All experiments are implemented using Pytorch [11].

A1. CIFAR-100 implementation details

The CIFAR-100 [7] is a typical classification dataset with 100 categories, consisting of 50,000 training images and 10,000 testing images. For ResNet-50 [4] and MBV3 [5], we adopt the training settings of [17]. Specifically, the epoch number and batch size are 500 and 64, respectively. The SGD is chosen as the optimizer with a 0.05 initial learning rate and a 0.0003 weight decay. We use the cosine decay schedule to adjust the learning rate over the training process. For WRN28-10 [18], we adopt the training settings of [18]. Specifically, the epoch number and batch size are 200 and 128, respectively. The SGD is chosen as the optimizer with a 0.1 initial learning rate and a 0.0005 weight decay. The learning rate scheduler is also the cosine decay schedule. For ViT [15] and Swin Transformer [9], we use an image size of 32x32 and a patch size of 4. The epoch number and batch size are 200 and 128, respectively. The optimizer is AdamW [10] with an initial learning rate of 0.001/0.003 for Swin/ViT and a 0.05 weight decay. The learning rate is warmed up for 10 epochs. The data augmentations are the same as the ones in [8]. Different from CNNs, where we regard the channel numbers of convolutional and linear layers as the width dimension, to prune the width of Transformers, we take the head numbers of attention layers and the channel numbers of linear layers into account.

A2. Tiny ImageNet implementation details

The Tiny Imagenet dataset is inherited from the ImageNet dataset [2], containing 200 categories, 100,000 training images, and 10,000 testing images. For

* These authors contributed equally.
** Corresponding author

ResNet-50 [4] and MBV3 [5], the epoch number and batch size are 500 and 64, respectively. The optimizer is SGD with a 0.1 initial learning rate and a 0.0003 weight decay. We utilize a step-wise learning rate scheduler, downscaling the learning rate to 0.1 and 0.01 of the original one at the 250-th and 375-th epoch, respectively. For WRN28-10 [18], we adopt the training settings in [13]. The epoch number and batch size are 200 and 128, respectively. The optimizer is SGD with a 0.2 initial learning rate and a 0.0001 weight decay. We utilize a step-wise learning rate scheduler, downscaling the learning rate to 0.1 and 0.01 of the original one at the 100-th and 150-th epoch, respectively.

A3. ImageNet implementation details

The ImageNet dataset [2] is a widely used classification benchmark, containing 1000 categories, 1.2 million training images, and 50000 testing images. For the evaluated ResNet-50 [4], the epoch number and batch size are 200 and 512, respectively. We utilize SGD as the optimizer. The learning rate is initialized as 0.2 and is controlled by a cosine decay schedule. The weight decay is 0.0001. Besides, we apply the commonly used data augmentations according to [6, 14].

A4. Pruning framework details

To build a relatively more general and user-friendly pruning framework, we resort to the symbolic tracing of Pytorch, called FX tracing [12]. Given any FX-compatible model, we first extract the FX graph from it. Based on the graph, we try to extract dependencies groups, which are similar to the concepts in [1,3]. A dependency group represents a series of layers whose outputs are expected to be integrated, such as being added or concatenated in the dimension to prune. When pruned, all layers in the same group are expected to have the same number of remaining channels.

Given the inputs to the model, we run an interpreter [12] based on the extracted FX graph instead of using the traditional model forward. The interpreter propagates the inputs from the root node of the graph to the last node, executing all encountered nodes according to their recorded operations, promising the same result as the one produced by the traditional model forward while providing the freedom to intercept any intermediate operation for injecting necessary functionality of pruning. Benefiting from the interpreter, we can dynamically reduce the layer number or shrink channels based on the sampled architecture without altering the model structure or parameters. To realize the dynamic interpreter, we dispatch all commonly used operators, such as trivial binary operators (add, mul, div, etc.), Pytorch tensor operators, and “torch.nn" forward functions, in the FX graph to a pre-registered dynamic handler, which can be easily extended to support customized operations.

Besides dynamic forward, we closely integrate FLOPs/parameters estimation with the above pruning framework via running an interpreter with a proxy tensor. A proxy tensor does not require dense calculation, such as convolution or matrix multiplication, resulting in significant acceleration of the FLOPs/parameters

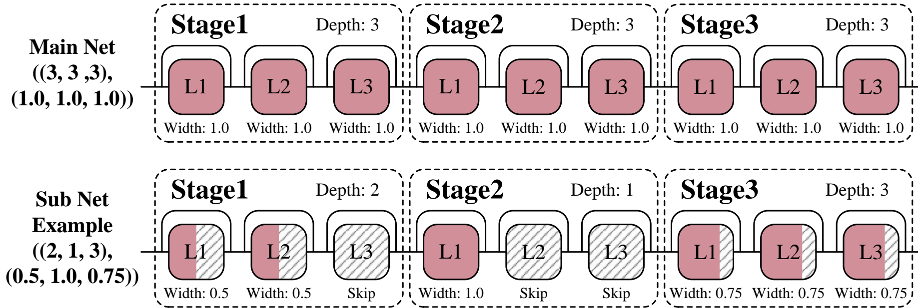

Fig. 1: The visualization of sampled architectures for an exemplary network. The exemplary network is composed of three stages. Each stage contains three layers that can be pruned, such as the convolutional layers.

estimation for dynamic models, which is the key to estimating the FLOPs of all architectures within the pool in a bearable time. During an interpreter run, a proxy tensor only contains the shape information of a tensor, and each encountered operation is required to infer the output shape of the tensor in addition to the FLOPs and parameters introduced by this operation. Similar to the dispatch of dynamic forward, we have registered the handlers of FLOPs/parameters estimation for all commonly used operators in the FX graph.

Appendix B: Pseudo Code of STP

To understand the detailed process of STP, the pseudo code of STP is shown in Alg. 1. Some expressions in Alg. 1 are different from the manuscript and they are explained as follows to avoid confusion.

In Alg. 1, we denote "$\leftarrow$" as the assignment operator and "$=$" as the equal comparison operator. Different from the sparsity ratio S_g, we utilize the target remaining FLOPs ratio r, which can achieve more precise control over the inference speed of the pruned network. $N_{shr} \leftarrow \left\lfloor \frac{k(N_p-1)}{T_{shr}} \right\rfloor$ is the removing number of architectures in each shrinking process, meaning that after shrinking $\frac{T_{shr}}{k}$ rounds, i.e., the T in Section 3.3 and Fig. 5 in the manuscript, the number of architectures in the pool $\mathcal{P}$ will reduce from N_p to 1. Notely, the rounding down of $\frac{k(N_p-1)}{T_{shr}}$ can cause that there are several architectures in $\mathcal{P}$; thus, we force to shrink the number of architecture in $\mathcal{P}$ into 1. The $\mathcal{N}_s$ and $\mathcal{N}_{sup}$ are the "Sub Network" and "Mut Network" in Fig. 5 respectively, while the $\mathcal{L}_{STC}$ and $\mathcal{L}_{SME}$ are the "KD_{sub}" and "KD_{mut}" in Fig. 5 respectively. β_1 and β_2 are the loss coefficients of $\mathcal{L}_{STS}$ and $\mathcal{L}_{SME}$ respectively.

Appendix C: Exemplary Sampled Subnets

Assuming an exemplary network with three stages, each consisting of three prunable layers, such as convolutional layers or linear layers, we encode the sampled

Algorithm 1: Stimulative training guided pruning (STP)

Input: A main network $\mathcal{N}_m$ with randomly initialized parameters θ_m; target remaining FLOPs ratio r; total training steps T_{total}; initial pool size N_p, the end steps of shrinking pool T_{shr}, pool refine interval k and shrinking number $N_{shr} \leftarrow \left\lfloor \frac{k(N_p-1)}{T_{shr}} \right\rfloor$ of each k steps; Input x and ground truth y of each minibatch;

Output: A pruned network $\mathcal{N}_s^*$ satisfying the target FLOPs ratio r with parameters θ_s^*;

1 $\triangleright$ Construct the architecture pool $\mathcal{P}$ by randomly sampling N_p subnets that satisfy target FLOPs ratio r; Initialize the training step $t \leftarrow 1$;
2 **while** $t \leq T_{total}$ **do**
3 $\triangleright$ forward N_m to obtain the output $Z_m(x;\theta_m)$ and compute the cross entropy loss with the ground truth $\mathcal{L}_{CE}$;
4 $\triangleright$ Sample a target subnet $\mathcal{N}_s$ from architecture pool $\mathcal{P}$, forward it and compute KL- loss $\mathcal{L}_{STS}$ with $Z_m(x;\theta_m)$ as Formulation 4; record $\mathcal{L}_{STS}$ and update the corresponding score in $\mathcal{P}$ as Formulation 8;
5 $\triangleright$ Mutate and expand the target subnet $\mathcal{N}_s$ to obtain $\mathcal{N}_{sup}$, forward $\mathcal{N}_{sup}$ and compute KL- loss $\mathcal{L}_{SME}$ with $Z_m(x;\theta_m)$ as Formulation 7;
6 $\triangleright$ Backward the total loss $\mathcal{L}_{total} \leftarrow \mathcal{L}_{CE} + \beta_1\mathcal{L}_{STS} + \beta_2\mathcal{L}_{SME}$ and update θ_m with by descending $\nabla_{\theta_m}\mathcal{L}_{total}$
7 **if** $t \ mod \ k = 0 \ and \ t \leq T_{shr}$ **then**
8 $\triangleright$ remove N_{shr} architectures with Top-N_{shr} score in the pool $\mathcal{P}$;
9 **end**
10 $\triangleright$ $t \leftarrow t + 1$;
11 **end**
12 $\triangleright$ There is only one architecture in the pool $\mathcal{P}$, extract it from $\mathcal{N}_m$ and regard it as the pruned network $\mathcal{N}_s^*$ with parameters θ_s^*.

Table 1: The remaining subnet architectures of ResNet-50 in the pool when the pool size is reduced to 10 under different FLOPs targets during the training on CIFAR-100. The Top-1 accuracy ("Acc (%)") of each architecture is measured after training. The architecture in bold is the final one remaining in the pool.

15% FLOPs Pool		35% FLOPs Pool		55% FLOPs Pool	
Architecture	Acc (%)	Architecture	Acc (%)	Architecture	Acc (%)
((1, 2, 5, 2), (0.5, 0.3, 0.3, 0.7))	79.51	((1, 2, 4, 3), (0.3, 0.7, 0.5, 1.0))	79.79	((3, 2, 5, 3), (0.3, 0.7, 0.9, 1.0))	80.16
((1, 3, 6, 2), (0.3, 0.3, 0.3, 0.7))	79.63	((2, 3, 3, 3), (0.3, 0.3, 0.7, 1.0))	79.86	((3, 2, 5, 3), (0.5, 0.5, 0.9, 1.0))	80.13
((1, 3, 4, 2), (0.5, 0.3, 0.3, 0.7))	79.63	((3, 4, 5, 3), (0.3, 0.3, 0.5, 1.0))	79.97	((3, 2, 6, 3), (0.3, 0.3, 0.9, 1.0))	80.21
((2, 3, 4, 2), (0.3, 0.3, 0.3, 0.7))	**79.64**	((3, 2, 6, 3), (0.3, 0.3, 0.5, 1.0))	79.74	((2, 2, 5, 3), (0.5, 0.5, 0.9, 1.0))	80.14
((1, 3, 5, 2), (0.3, 0.3, 0.3, 0.7))	79.48	((1, 2, 3, 3), (0.5, 0.3, 0.7, 1.0))	79.69	**((2, 4, 5, 3), (0.5, 0.5, 0.9, 0.9))**	**80.41**
((1, 4, 6, 2), (0.3, 0.3, 0.3, 0.7))	79.54	((3, 2, 4, 3), (0.5, 0.3, 0.5, 1.0))	79.56	((1, 4, 6, 3), (0.3, 0.3, 0.9, 1.0))	80.07
((1, 3, 5, 2), (0.5, 0.3, 0.3, 0.7))	79.39	((1, 2, 4, 3), (0.9, 0.3, 0.5, 1.0))	79.59	((2, 2, 6, 3), (0.3, 0.3, 0.9, 1.0))	80.08
((1, 2, 3, 3), (0.3, 0.3, 0.3, 0.7))	79.16	**((1, 4, 3, 3), (0.5, 0.3, 0.7, 1.0))**	**80.0**	((2, 3, 5, 3), (0.5, 0.5, 0.9, 1.0))	80.07
((1, 1, 4, 3), (0.3, 0.3, 0.3, 0.7))	79.31	((3, 2, 5, 3), (0.3, 0.5, 0.5, 1.0))	79.82	((1, 4, 5, 3), (0.5, 0.3, 1.0, 0.9))	80.05
((2, 1, 3, 3), (0.3, 0.3, 0.3, 0.7))	79.05	((1, 4, 3, 3), (0.3, 0.3, 0.7, 1.0))	79.78	((3, 2, 5, 3), (0.5, 0.7, 0.9, 0.9))	80.01

Table 2: The final remaining architecture in repeated experiments (“Exp") using different seeds for ResNet-50 on CIFAR100 with a 15% FLOPs target. The Top-1 accuracy (“Acc (%)"), FLOPs, and parameters (“Params") of the architecture are reported. For FLOPs and parameters, we use their relative fraction w.r.t. the ones of the main network as the metrics.

Exp	Architecture	Acc (%)	FLOPs (%)	Params (%)
#1	((2, 3, 5, 2), (0.3, 0.3, 0.3, 0.7))	79.47	14.88	22.28
#2	((1, 3, 6, 2), (0.3, 0.3, 0.3, 0.7))	79.49	14.69	22.36
#3	((1, 2, 5, 2), (0.5, 0.3, 0.3, 0.7))	79.55	15.22	22.33
#4	((2, 3, 4, 2), (0.3, 0.3, 0.3, 0.7))	79.64	14.89	21.94
#5	((2, 2, 6, 2), (0.3, 0.3, 0.3, 0.7))	79.57	15.23	22.93

architecture as a nested tuple as shown in Fig. 1. The first inner tuple denotes the remaining layers of each stage (the depth dimension), and each element in the second inner tuple denotes the layer-wise proportion of the remaining output channels to the total output channels in the corresponding stage (the width dimension).

Based on the nested tuple denotation, snapshots of the architecture pool are shown in Table 1. Given different FLOPs targets, i.e., 15%, 35%, and 55%, we conduct experiments on CIFAR-100 using ResNet-50 as the backbone. The ResNet-50 has four stages, containing 3, 4, 6, and 3 blocks, respectively. A block is composed of three convolutional layers. Slightly different from the former exemplary network, we pack the three convolutional layers and view them as a “layer" that will be skipped or retained together based on a given depth and will be pruned with the same width. The minimum and maximum widths are set to 0.3 and 1.0, respectively. The width granularity is 0.2, resulting in 5 different choices (0.3, 0.5, 0.7, 0.9, 1.0). It can be observed from Table 1 that 1) when the pool is about to converge (the pool size is reduced to 10), the remaining architectures are similar to each other, especially on the width dimension. For example, except for the first stage, where widths are either 0.3 or 0.5, the widths of the second, the third, and the final stage are consistently 0.3, 0.3, and 0.7, respectively; 2) From the aspect of depth, layers in the last two stages tend to be retained. These observations might help infer the relative importance of different stages; for example, the final stage seems to be the most important one because of relatively more remaining width and depth.

Appendix D: Robustness of Architecture Pool

To explore the robustness of the architecture pool, we choose 15% as the FLOPs target and conduct five experiments with different random seeds on CIFAR-100 using ResNet-50 as the backbone. The results are shown in Table 2. The final architectures in different experiments are highly similar in accordance with the observations in Appendix C. Besides, the Top-1 accuracy, FLOPs, and parame-

ters of the final architecture are relatively consistent with negligible deviations (within 0.2% for Top-1 accuracy deviations, within 1% for FLOPs and parameters deviations), verifying the robustness of the architecture pool, i.e., the converged architecture is relatively stable and its performance is guaranteed.

Appendix E: Theoretical Explanation and Analysis of Relative Sparsity Effect in ST

Due to the highly complex topology and non-linearity, explaining the relative sparsity effect in a deep neural network is difficult. Rather than giving completed and rigorous proof, we provide an explanation of a degenerate case for an intuitive understanding. We analyze the ST process, which transfers the capacity from the whole parameters to the chosen ones.

For simplicity, we consider a single-layer feed-forward network $\mathcal{N}$ with parameters $\theta \in \mathbb{R}^M$ and a sigmoid activation $sig(\cdot)$. We adopt mean-square error (MSE) as the distillation loss to transfer capacity:

$$\mathcal{L}_{KD} = (sig(\theta_s x_s + \theta_d x_d) - sig(\theta_s x_s))^2 \tag{1}$$

where $\theta_s \subseteq \theta$ is the chosen parameters, corresponding to the subnets in the manuscript; $\theta_d = \theta \setminus \theta_s$ is the unchosen parameters; x_s and x_d are the inputs cooperating with θ_s and θ_d, respectively. Note that, to avoid teacher degradation in vanilla KD, the teacher supervision, i.e., $sig(\theta_s x_s + \theta_d x_d)$, is detached from the computation graph, thus irrelevant to computing the partial gradient of θ_s:

$$\frac{\partial \mathcal{L}_{KD}}{\partial \theta_s} = -2[sig(\theta_s x_s + \theta_d x_d) - sig(\theta_s x_s)] \cdot sig(\theta_s x_s) \cdot [1 - sig(\theta_s x_s)] \cdot x_s. \tag{2}$$

This partial gradient consists of three components: 1) $sig(\theta_s x_s + \theta_d x_d) - sig(\theta_s x_s)$, 2) $sig(\theta_s x_s) \cdot [1 - sig(\theta_s x_s)]$, and 3) x_s. When the optimization converges, at least one component is deemed to approach zero. Since x_s is the input following the distribution of data, it is not guaranteed to approach zero. Similar to [16, 17], we apply Taylor expansion to analyze $[sig(\theta_s x_s + \theta_d x_d) - sig(\theta_s x_s)]$:

$$sig(\theta_s x_s + \theta_d x_d) - sig(\theta_s x_s) \approx sig(\theta_s x_s) \cdot [1 - sig(\theta_s x_s)] \cdot \theta_d x_d. \tag{3}$$

Note that $\theta_d x_d$ can be considered as a constant from two aspects: 1) θ_d keeps unaltered due to the absence of its gradient flow; 2) x_d is the input data and invariant to optimization. Consequently, the partial gradient $\partial \mathcal{L}_{KD} / \partial \theta_s$ is approximately determined by $sig(\theta_s x_s) \cdot [1 - sig(\theta_s x_s)]$. If the L_1 norm of θ_s increases dramatically, it is likely to make the partial gradient approach zero. To sum up, optimizing $\mathcal{L}_{KD}$ produces no influence on θ_d and enhances the L_1 norm of θ_s, which results in the relative sparsity effect. Since qualitative analyses and approximations are involved in the above mathematical proof, rigorousness is not guaranteed. This section merely provides an illustrative understanding of the relative sparsity effect in ST. Delving into rigorous proofs in more complex situations could become a future research direction, but this is beyond the scope of discussion in this manuscript.

Appendix F: Efficiency of STP

To explore the efficiency of STP, we record the training time of STP and other pruning methods. The experiments are conducted on CIFAR100, using ResNet50 as the backbone. As shown in Table 3, our method achieves superior performance with acceptable (the second shortest) training time. The efficiency of our method mainly lies in high parallelism. The forward/backward passes of the main network and subnets can be executed at the same time leveraging CUDA streams or multi-GPU model parallelism. Besides, our method is one-stage, eliminating sequential fine-tuning or iterative pruning, which can further reduce the training time. To ablate the influence of forward/backward counts, we extend the epochs of other methods three times to align our forward/backward counts. With 3x forward/backward, other methods struggle to improve further. It demonstrates that merely extending the training schedule cannot notably enhance pruning performance, suggesting the superiority of our method.

Table 3: Performance and training time of different methods and settings.

	RST-S	Depgraph	OTO v2	IMP-Refill	Ours
Top-1 Acc(1x training schedule)	75.02	49.07	77.04	75.12	79.64
Top-1 Acc(3x training schedule)	75.39	50.12	77.34	75.97	-
GPU time per epoch	46.58s	75.10s	82.22s	77.32s	61.92s

References

1. Chen, T., Liang, L., Tianyu, D., Zhu, Z., Zharkov, I.: Otov2: Automatic, generic, user-friendly. In: International Conference on Learning Representations (2023)
2. Deng, J., Dong, W., Socher, R., Li, L.J., Li, K., Fei-Fei, L.: Imagenet: A large-scale hierarchical image database. In: 2009 IEEE conference on computer vision and pattern recognition. pp. 248–255. Ieee (2009)
3. Fang, G., Ma, X., Song, M., Mi, M.B., Wang, X.: Depgraph: Towards any structural pruning. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. pp. 16091–16101 (2023)
4. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: Proceedings of the IEEE conference on computer vision and pattern recognition. pp. 770–778 (2016)
5. Howard, A., Sandler, M., Chu, G., Chen, L.C., Chen, B., Tan, M., Wang, W., Zhu, Y., Pang, R., Vasudevan, V., et al.: Searching for mobilenetv3. In: Proceedings of the IEEE/CVF international conference on computer vision. pp. 1314–1324 (2019)
6. Huang, G., Liu, Z., Van Der Maaten, L., Weinberger, K.Q.: Densely connected convolutional networks. In: Proceedings of the IEEE conference on computer vision and pattern recognition. pp. 4700–4708 (2017)
7. Krizhevsky, A., Hinton, G., et al.: Learning multiple layers of features from tiny images (2009)
8. Lee, S.H., Lee, S., Song, B.C.: Vision transformer for small-size datasets (2021)

9. Liu, Z., Lin, Y., Cao, Y., Hu, H., Wei, Y., Zhang, Z., Lin, S., Guo, B.: Swin transformer: Hierarchical vision transformer using shifted windows. In: Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV) (2021)
10. Loshchilov, I., Hutter, F.: Decoupled weight decay regularization (2019)
11. Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., Chintala, S.: Pytorch: An imperative style, high-performance deep learning library. In: Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., Garnett, R. (eds.) Advances in Neural Information Processing Systems. vol. 32. Curran Associates, Inc. (2019), https://proceedings.neurips.cc/paper_files/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf
12. Reed, J., DeVito, Z., He, H., Ussery, A., Ansel, J.: torch.fx: Practical program capture and transformation for deep learning in python. In: Marculescu, D., Chi, Y., Wu, C. (eds.) Proceedings of Machine Learning and Systems. vol. 4, pp. 638–651 (2022), https://proceedings.mlsys.org/paper_files/paper/2022/file/7c98f9c7ab2df90911da23f9ce72ed6e-Paper.pdf
13. Shen, Y., Xu, L., Yang, Y., Li, Y., Guo, Y.: Self-distillation from the last mini-batch for consistency regularization. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. pp. 11943–11952 (2022)
14. Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., Rabinovich, A.: Going deeper with convolutions. In: Proceedings of the IEEE conference on computer vision and pattern recognition. pp. 1–9 (2015)
15. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł., Polosukhin, I.: Attention is all you need. Advances in neural information processing systems **30** (2017)
16. Ye, P., He, T., Tang, S., Li, B., Chen, T., Bai, L., Ouyang, W.: Stimulative training++: Go beyond the performance limits of residual networks. arXiv preprint arXiv:2305.02507 (2023)
17. Ye, P., Tang, S., Li, B., Chen, T., Ouyang, W.: Stimulative training of residual networks: A social psychology perspective of loafing. Advances in Neural Information Processing Systems **35**, 3596–3608 (2022)
18. Zagoruyko, S., Komodakis, N.: Wide residual networks. In: British Machine Vision Conference 2016. British Machine Vision Association (2016)