Supplementary Materials for *Auto-GAS: Automated Proxy Discovery for Training-free Generative Architecture Search*

Lujun Li[1], Haosen Sun[1], Shiwen Li[2], Peijie Dong[3], Wenhan Luo[1], Wei Xue[1], Qifeng Liu[1✉], and Yike Guo[1✉]

[1] The Hong Kong University of Science and Technology
[2] Xidian University
[3] The Hong Kong University of Science and Technology (Guangzhou)
lilujunai@gmail.com;hsunas@connect.ust.hk;shiwenlisw@gmail.com;
pdong212@connect.hkust-gz.edu.cn;whluo.china@gmail.com;weixue@ust.hk;
liuqifeng@ust.hk;yikeguo@ust.hk

In this supplemental material, we provide additional details on the proposed Auto-GAS method to enhance understanding through thoughtful extra explanations, analyses, and technical specifics omitted from the main paper due to length constraints.

A More discussions

A.1 Analyzing Different Proxy Formations

To demonstrate the superiority of the feature calculation method of our Auto-GAS, we conduct further experiments to facilitate a broader comparison. Specifically, we employ four different correlation calculation methods on the CIFAR-10 dataset to measure the correlation between sample features:

- **Hessian Trace [21]:** This method measures the average trace of the second derivative of the loss function with respect to network weights. This slight difference in performance may be attributed to the fact that Hessian Trace method focuses on the loss landscape's curvature, which provides insights for evaluating the stability and convergence behavior of the candidate generator architectures.
- **L_2 norm:** The L_2 norm represents the Euclidean length of sample features, serving as a metric for the overall size and distribution of features. The results are inferior to Auto-GAS, with an IS of 7.56 and FID of 24.62. Their discrepancy might be because the L_2 norm overlooks higher-order statistics and interactions between features.
- **Entropy:** Entropy measures the diversity and uncertainty of sample features. The generated images yield a slightly higher IS compared to the L2 norm (IS=7.7, FID=25.43).

✉ Corresponding authors.

- **Variance:** Variance represents the dispersion of sample features, providing a measure of feature stability and consistency. The results are worse than Auto-GAS, with an IS of 7.67 and FID of 23.75. This discrepancy may be due to the fact that variance only considers the spread of feature values without accounting for feature interactions and their impact on GAN performance.

A.2 Discussions

About Limitations and Future Work. While our proposed Auto-GAS approach achieves promising results, some limitations remain. Consistent with previous neural architecture search methods [1,6,7,10], Auto-GAS incurs additional overhead from the search process. However, we aim to address this limitation by carefully selecting search operators and algorithms that provide considerable speedups, such as our lightweight mating and mutation mechanisms. Still, further optimizations to the search procedure could help reduce its computational cost. Additionally, our search space currently focuses on CNN architectures and hyperparameters but does not consider model paradigms beyond CNNs. In future work, we will continue improving the efficiency of Auto-GAS and expanding its search space to incorporate more diverse types of neural networks with advanced methods [3–5, 12–20, 22, 24, 26, 27]. We aim to advance the general field of automated deep learning and make neural architecture design increasingly accessible.

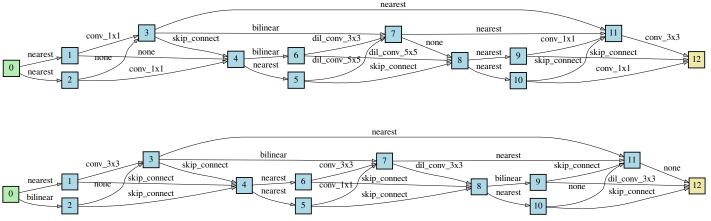

Figure 1: The architecture of the searched generator on CIFAR-10 (top) and STL-10 (bottom).

B Searched Architecture Analysis

Searched Generator. We perform generator architecture search on both CIFAR-10 and STL-10 datasets, as shown in Figure 1. For CIFAR-10, the searched G structure (size=9.9 MB) consists of a total of 12 operations, including 2 upsampling (nearest and bilinear), 4 convolutions, 3 'none', and 3 'skip-connect' operations. As for STL-10, the searched G structure (size=10.96 MB) consists of 17 operations, including 3 upsampling, 3 convolutions, 3 'none', and 7 'skip-connect'

Table 1: The unary operations in the search space. "UOP" denotes the unary operations, and ' The type of input and output can be scalar or matrix. "no_op" denotes that we do not perform any operation, and allows for the sparsity of the computation graph. Not all operations below are available or mathematically sound.

OP ID	OP Name	Input Args	Output Args	Description
UOP00	no_op	–	–	–
UOP01	element_wise_abs	a / scalar,matrix	b / scalar,matrix	$x_b = \|x_a\|$
UOP02	element_wise_tanh	a / scalar,matrix	b / scalar,matrix	$x_b = \tanh(x_a)$
UOP03	element_wise_pow	a / scalar,matrix	b / scalar,matrix	$x_b = x_a^2$
UOP04	element_wise_exp	a / scalar,matrix	b / scalar,matrix	$x_b = e^{x_a}$
UOP05	element_wise_log	a / scalar,matrix	b / scalar,matrix	$x_b = \ln x_a$
UOP06	element_wise_relu	a / scalar,matrix	b / scalar,matrix	$x_b = \max(0, x_a)$
UOP07	element_wise_leaky_relu	a / scalar,matrix	b / scalar,matrix	$x_b = \max(0.1x_a, x_a)$
UOP08	element_wise_swish	a / scalar,matrix	b / scalar,matrix	$x_b = x_a \times \text{sigmoid}(x_a)$
UOP09	element_wise_mish	a / scalar,matrix	b / scalar,matrix	$x_b = x_a \times \tanh(\ln 1 + \exp(x_a))$
UOP10	element_wise_invert	a / scalar,matrix	b / scalar,matrix	$x_b = 1/x_a$
UOP11	element_wise_normalized_sum	a / scalar,matrix	b / scalar,matrix	$x_b = \frac{\sum x_a}{\text{numel}(x_a)+\epsilon}$
UOP12	normalize	a / scalar,matrix	b / scalar,matrix	$x_b = \frac{x_a - \text{mean}(x_a)}{\text{std}(x_a)}$
UOP13	sigmoid	a / scalar,matrix	b / scalar,matrix	$x_b = \frac{1}{1+e^{-x_a}}$
UOP14	logsoftmax	a / scalar,matrix	b / scalar,matrix	$x_b = \ln \frac{e^{x_a}}{\sum_{i=1}^{n} e^{s_i}}$
UOP15	softmax	a / scalar,matrix	b / scalar,matrix	$x_b = \frac{e^{x_a}}{\sum_{i=1}^{n} e^{s_i}}$
UOP16	element_wise_sqrt	a / scalar,matrix	b / scalar,matrix	$x_b = \sqrt{x_a}$
UOP17	element_wise_revert	a / scalar,matrix	b / scalar,matrix	$x_b = -x_a$
UOP18	frobenius_norm	a / scalar,matrix	b / scalar,matrix	$x_b = \sqrt{\sum_{i=1}^{n} s_i^2}$
UOP19	element_wise_abslog	a / scalar,matrix	b / scalar,matrix	$x_b = \|\ln x_a\|$
UOP20	l1_norm	a / scalar,matrix	b / scalar,matrix	$x_b = \frac{\sum_{i=1}^{n} \|s_i\|}{\text{numel}(x_a)}$
UOP21	min_max_normalize	a / scalar,matrix	b / scalar,matrix	$x_b = \frac{x_a - \min(x_a)}{\max(x_a) - \min(x_a)}$
UOP22	to_mean_scalar	a / scalar,matrix	b / scalar	$x_b = \frac{x_a}{n}$
UOP23	to_std_scalar	a / scalar,matrix	b / scalar	$x_b = \sqrt{\frac{\sum_{i=1}^{n}(s_i - \bar{s})^2}{n}}$

operations. Unlike CIFAR-10, STL-10 applies nearest and bilinear upsampling operations on input nodes '0' and '1'. The choice of nearest upsampling allows for faster computation, while bilinear upsampling provides better smoothing, preserving details in the images. This design not only improves search efficiency but also enhances the expressive power of the generator.

C Detailed Experiment Settings

We conduct experiments using the CIFAR-10 [11] and STL-10 [2] datasets, which are in line with the EAGAN [25] and other training-based NAS-GANs [8, 9, 23]. We search and evaluate on both CIFAR-10 and STL-10. The CIFAR-10 dataset contains 50,000 training images and 10,000 test images, each with a resolution of 32×32 per image. The STL-10 dataset contains 100,500 images with a higher resolution of 48×48, which we resize to 48×48. The search process contain 150 epochs, and the search is performed every 10 epochs. During each round of search, a population of $P = 32$ individuals is trained and evolved, where the number of selected individuals is 8. The batch sizes of the generator and discriminator are set to 80 and 40, respectively. In the evaluation of both stages,

we use 16 randomly generated Gaussian noise images to compute the proxy score, Fréchet Inception Distance (FID) and Inception Score (IS) . This reduces the evaluation set and greatly shortens the evaluation time while maintaining the performance of the searched architecture. In the discriminator search process, we first select the generator with the best proxy score from the overall generators searched, and once the best generator has been identified, we continue to search for discriminator architectures that are complementary to the selected generator. The computational resources required are 0.07 and 0.05 GPU days, respectively, making the entire search process efficient and feasible. After the two-stage search, we employ the generator and discriminator architectures with the best proxy scores to perform a full training of the GAN. The fully-training consists of 500 epochs. For the CIFAR-10 dataset, the batch size and learning rate for both the generator and discriminator remain the same as in the search stage, specifically, the batch size is set to 80, and the initial learning rate is 0.0002. As for the STL-10 dataset, the generator and discriminator have different learning rates of 0.0003 and 0.0002, respectively. Additionally, the batch sizes for the generator and discriminator are adjusted to 256 and 128, respectively. During the evaluation, we follow the approach of previous NAS-GAN works [8,9,25], generating 50,000 images for calculating the IS and FID metrics.

D Details of Search Space

Our proxy search space incorporates transform, encoding, reduction, and augment operators to construct diverse proxy functions. Table 1 presents the formalization of some operations.

D.1 Details of Transform Operations

Transform operations enhance the raw features x to represent architectural potentials, formulated as $f(x;)$. Common transformations include nonlinear activations τ such as ReLU ($\tau = \mathcal{R}(x) = \max(0, x)$), SiLU ($\tau = \mathcal{S}(x) = x \cdot \sigma(x)$), and Mish ($\tau = \mathcal{M}(x) = x \cdot \tanh(softplus(x))$). Additional operators aim to process features, for example scale normalization ($\tau = \mathcal{SN}(x)$), etc. We consider the normalization operation on $\{N, C, HW\}$ dimensions:

$$\mathcal{T}_{norm_{\{N,C,HW\}}} = (f_{\{N,C,HW\}} - \mu_{\{N,C,HW\}})/\sigma_{\{N,C,HW\}} \tag{1}$$

where μ and σ represent the mean and standard deviation. Their detailed implementations are presented in Listing 1.

D.2 Details of Encoding Operations

Encoding functions Φ estimate the mutual correlation information between samples. Inspired by coding theory and manifold learning, we select binary encodings like sign ($\phi = \mathcal{S}(x)$) and above zero/median thresholding, as well as matrix-based encodings including the Gram ($\phi = \mathcal{G}(x)$) and Pearson's R ($\phi = \mathcal{P}(x)$) matrices. Their detailed implementations are presented in Listing 2.

D.3 Details of Reduction Operations

Reduction operators ρ extract meaningful matrix information. Formulated as $\rho(\Phi(f(x;)))$, choices involve matrix measures like log determinant ($\rho = \log\det$), trace ($\rho = \mathrm{tr}\cdot$), and eigenvalues ($\rho = \lambda$). Their detailed implementations are presented in Listing 3.

D.4 Details of Augment Operations

Augment operators a allow proxy values $p = \rho(\Phi(f(x;)))$ to better predict performance. Operators a encompass transformations like absolute value, powers, L1-norms, logs, etc. Their detailed implementations are presented in Listing 4.

Listing 1.1: The PyTorch implementation of Transform operations.

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from einops import rearrange, reduce

def trans_scale(f):
    return reduce(f, 'b c (h1 h2) (w1 w2) -> b c h1 w1', 'max', h2=2,
        w2=2)

def trans_local(f):
    return rearrange(f, 'b c (h hp) (w wp) -> b (c h w) hp wp', hp=2,
        wp=2)

def trans_mask(f, threshold=0.65):
    N, C, H, W = f.shape
    device = f.device
    mat = torch.rand((N, 1, H, W)).to(device)
    mat = torch.where(mat > 1 - threshold, 0, 1).to(device)
    return torch.mul(f, mat)


def trans_att(f, T=0.5):
    N, C, H, W = f.shape
    value = torch.abs(f)
    fea_map = value.mean(axis=1, keepdim=True)
    S_attention = (H * W * F.softmax(
        (fea_map / T).view(N, -1), dim=1)).view(N, H, W)
    return S_attention.unsqueeze(dim=-1)


def trans_nop(f):
    return f

def trans_norm_HW(f):
```

```
    return F.normalize(f, p=2, dim=(2, 3))


def trans_norm_C(f):
    return F.normalize(f, p=2, dim=1)


def trans_norm_N(f):
    return F.normalize(f, p=2, dim=0)


def trans_softmax_N(f):
    return F.softmax(f, dim=0)


def trans_softmax_C(f):
    return F.softmax(f, dim=1)


def trans_softmax_HW(f):
    return F.softmax(f, dim=2)


def trans_logsoftmax_N(f):
    return F.log_softmax(f, dim=1)


def trans_logsoftmax_C(f):
    return F.log_softmax(f, dim=1)


def trans_logsoftmax_HW(f):
    return F.log_softmax(f, dim=2)


def trans_sqrt(f):
    """transform with sqrt"""
    f = torch.clamp(f, min=0.0)
    return torch.sqrt(torch.abs(f))


def trans_log(f):
    return torch.sign(f) * torch.log(torch.abs(f) + 1e-9)


def trans_min_max_normalize(f):
    A_min, A_max = f.min(), f.max()
    return (f - A_min) / (A_max - A_min + 1e-9)

def trans_abs(f):
```

```
    return torch.abs(f)

def trans_sigmoid(f):
    return torch.sigmoid(f)

def trans_swish(f):
    """transform with swish"""
    return f * torch.sigmoid(f)

def trans_tanh(f):
    return torch.tanh(f)

def trans_relu(f):
    return F.relu(f)


def trans_leaky_relu(f):
    return F.leaky_relu(f)

def trans_mish(f):
    return f * torch.tanh(F.softplus(f))


def trans_exp(f):
    return torch.exp(f)
```

Listing 1.2: The PyTorch implementation of Encoding operations.

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from einops import rearrange, reduce

def above_zero_op(x: Tensor):
    return (x > 0).float()

def above_mean_op(x: Tensor):
    return (x > x.mean()).float()

def above_median_op(x: Tensor):
    return (x > x.median()).float()

def sign_op(x: Tensor):
    return torch.sign(x).float()


def gram_matrix_op(x: Tensor):
    return x @ x.T
```

```
def corref_op(x: Tensor):
    return torch.corrcoef(x)


def trace_op(x: Matrix) -> Scalar:
    if isinstance(x, torch.Tensor):
        return torch.trace(x)
    elif isinstance(x, np.ndarray):
        return np.trace(x)
    else:
        raise ValueError(f'x should be a matrix, but got {type(x)}')

def hamming_op(x: ALLTYPE) -> Scalar:
    if isinstance(x, (list, tuple)):
        K_list = []
        for x_item in x:
            if isinstance(x_item, torch.Tensor):
                K_list.append(x_item @ x_item.t())
                K_list.append((1. - x_item) @ (1. - x_item.t()))
            elif isinstance(x_item, np.ndarray):
                K_list.append(x_item @ x_item.T)
                K_list.append((1. - x_item) @ (1. - x_item.T))
        return sum(K_list)
    else:
        if isinstance(x, torch.Tensor):
            return x @ x.t() + (1. - x) @ (1. - x.t())
        elif isinstance(x, np.ndarray):
            return x @ x.T + (1. - x) @ (1. - x.T)
        else:
            raise ValueError(
                f'x should be a tensor or ndarray, but got {type(x)}')
```

Listing 1.3: The PyTorch implementation of Reduction operations.

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from einops import rearrange, reduce


def slogdet_op(x: Matrix) -> Scalar:
    if isinstance(x, torch.Tensor):
        sign, value = torch.linalg.slogdet(x)
        return value
    elif isinstance(x, np.ndarray):
        sign, value = np.linalg.slogdet(x)
        return value
    else:
        raise ValueError(f'x should be a matrix, but got {type(x)}')
```

```python
def eigenvalue_op(x: Matrix) -> Vector:
    if isinstance(x, torch.Tensor):
        assert len(x.shape) == 2 and x.shape[0] == x.shape[1]
        return torch.linalg.eig(x)[0]
    elif isinstance(x, np.ndarray):
        assert len(
            x.shape) == 2 and x.shape[0] == x.shape[1], f'x.shape:
                {x.shape}'
        return np.linalg.eigvals(x)[0]
    else:
        raise ValueError(f'x should be a matrix, but got {type(x)}')


def element_wise_abslog_op(x: ALLTYPE) -> ALLTYPE:
    if isinstance(x, torch.Tensor):
        x[x == 0] = 1
        x = torch.abs(x)
        return torch.log(x)
    elif isinstance(x, np.ndarray):
        x[x == 0] = 1
        x = np.abs(x)
        return np.log(x)
    else:
        raise ValueError(f'x should be a tensor or ndarray, but got
            {type(x)}')
```

Listing 1.4: The PyTorch implementation of Augment operations.

```python
import torch
import torch.nn as nn
import torch.nn.functional as F
from einops import rearrange, reduce



def no_op(x: ALLTYPE) -> ALLTYPE:
    """ No operation. """
    return x


def element_wise_log_op(x: ALLTYPE) -> ALLTYPE:
    if isinstance(x, torch.Tensor):
        x[x <= 0] = 1
        return torch.log(x)
    elif isinstance(x, np.ndarray):
        x[x <= 0] = 1
        return np.log(x)
```

```
    else:
        raise ValueError(f'x should be a tensor or ndarray, but got
            {type(x)}')


def element_wise_abslog_op(x: ALLTYPE) -> ALLTYPE:
    if isinstance(x, torch.Tensor):
        x[x == 0] = 1
        x = torch.abs(x)
        return torch.log(x)
    elif isinstance(x, np.ndarray):
        x[x == 0] = 1
        x = np.abs(x)
        return np.log(x)
    else:
        raise ValueError(f'x should be a tensor or ndarray, but got
            {type(x)}')


def element_wise_abs_op(x: ALLTYPE) -> ALLTYPE:
    if isinstance(x, torch.Tensor):
        return torch.abs(x)
    elif isinstance(x, np.ndarray):
        return np.abs(x)
    else:
        raise ValueError(f'x should be a tensor or ndarray, but got
            {type(x)}')


def element_wise_pow_op(x: ALLTYPE) -> ALLTYPE:
    if isinstance(x, torch.Tensor):
        return torch.pow(x, 2)
    elif isinstance(x, np.ndarray):
        return np.power(x, 2)
    else:
        raise ValueError(f'x should be a tensor or ndarray, but got
            {type(x)}')


def element_wise_exp_op(x: ALLTYPE) -> ALLTYPE:
    if isinstance(x, torch.Tensor):
        return torch.exp(x)
    elif isinstance(x, np.ndarray):
        return np.exp(x)
    else:
        raise ValueError(f'x should be a tensor or ndarray, but got
            {type(x)}')


def normalize_op(x: ALLTYPE) -> ALLTYPE:
```

```
    if isinstance(x, torch.Tensor):
        m = torch.mean(x)
        s = torch.std(x)
        C = (x - m) / s
        C[C != C] = 0
        return C
    elif isinstance(x, np.ndarray):
        m = np.mean(x)
        s = np.std(x)
        C = (x - m) / (s + 1e-6)
        C[np.isnan(C)] = 0
        return C
    else:
        raise ValueError(f'x should be a tensor or ndarray, but got
            {type(x)}')


def frobenius_norm_op(x: Matrix) -> Scalar:
    if isinstance(x, torch.Tensor):
        return torch.norm(x, p='fro')
    elif isinstance(x, np.ndarray):
        return np.linalg.norm(x, ord='fro')
    else:
        raise ValueError(f'x should be a matrix, but got {type(x)}')


def element_wise_normalized_sum_op(x: ALLTYPE) -> Scalar:
    if isinstance(x, torch.Tensor):
        return torch.sum(x) / x.numel()
    elif isinstance(x, np.ndarray):
        return np.sum(x) / x.size
    else:
        raise ValueError(f'x should be a tensor or ndarray, but got
            {type(x)}')


def l1_norm_op(x: ALLTYPE) -> Scalar:
    if isinstance(x, torch.Tensor):
        return torch.sum(torch.abs(x)) / x.numel()
    elif isinstance(x, np.ndarray):
        return np.sum(np.abs(x)) / x.size
    else:
        raise ValueError(f'x should be a tensor or ndarray, but got
            {type(x)}')


def softmax_op(x: Vector) -> Vector:
    if isinstance(x, torch.Tensor):
        return F.softmax(x, dim=0)
    elif isinstance(x, np.ndarray):
```

```
        e_x = np.exp(x - np.max(x))
        return e_x / e_x.sum(axis=0)
    else:
        raise ValueError(f'x should be a vector, but got {type(x)}')


def logsoftmax_op(x: Vector) -> Vector:
    if isinstance(x, torch.Tensor):
        return F.log_softmax(x, dim=0)
    elif isinstance(x, np.ndarray):
        return np.log(softmax_op(x))
    else:
        raise ValueError(f'x should be a vector, but got {type(x)}')


def element_wise_sigmoid_op(x: ALLTYPE) -> ALLTYPE:
    if isinstance(x, torch.Tensor):
        return torch.sigmoid(x)
    elif isinstance(x, np.ndarray):
        return 1 / (1 + np.exp(-x))
    else:
        raise ValueError(f'x should be a tensor or ndarray, but got
            {type(x)}')


def element_wise_tanh_op(x: ALLTYPE) -> ALLTYPE:
    if isinstance(x, torch.Tensor):
        return torch.tanh(x)
    elif isinstance(x, np.ndarray):
        return np.tanh(x)
    else:
        raise ValueError(f'x should be a tensor or ndarray, but got
            {type(x)}')


def element_wise_sigmoid_op(x: ALLTYPE) -> ALLTYPE:
    if isinstance(x, torch.Tensor):
        return torch.sigmoid(x)
    elif isinstance(x, np.ndarray):
        return 1 / (1 + np.exp(-x))
    else:
        raise ValueError(f'x should be a tensor or ndarray, but got
            {type(x)}')


def element_wise_tanh_op(x: ALLTYPE) -> ALLTYPE:
    if isinstance(x, torch.Tensor):
        return torch.tanh(x)
    elif isinstance(x, np.ndarray):
        return np.tanh(x)
```

```
    else:
        raise ValueError(f'x should be a tensor or ndarray, but got
            {type(x)}')


def element_wise_sqrt_op(x: ALLTYPE) -> ALLTYPE:
    if isinstance(x, torch.Tensor):
        # Ensuring non-negative values for sqrt
        x = torch.clamp(x, min=0)
        return torch.sqrt(x)
    elif isinstance(x, np.ndarray):
        # Ensuring non-negative values for sqrt
        x = np.clip(x, 0, None)
        return np.sqrt(x)
    else:
        raise ValueError(f'x should be a tensor or ndarray, but got
            {type(x)}')
```

References

1. Chen, K., Yang, L., Chen, Y., Chen, K., Xu, Y., Li, L.: Gp-nas-ensemble: a model for the nas performance prediction. In: CVPRW (2022)
2. Coates, A., Ng, A., Lee, H.: An analysis of single-layer networks in unsupervised feature learning. In: Fourteenth International Conference on Artificial Intelligence and Statistics (2011)
3. Dong, P., Li, L., Tang, Z., Liu, X., Pan, X., Wang, Q., Chu, X.: Pruner-zero: Evolving symbolic pruning metric from scratch for large language models. In: ICML (2024)
4. Dong, P., Li, L., Wei, Z.: Diswot: Student architecture search for distillation without training. In: CVPR (2023)
5. Dong, P., Li, L., Wei, Z., Niu, X., Tian, Z., Pan, H.: Emq: Evolving training-free proxies for automated mixed precision quantization. In: ICCV (2023)
6. Dong, P., Niu, X., Li, L., Tian, Z., Wang, X., Wei, Z., Pan, H., Li, D.: Rd-nas: Enhancing one-shot supernet ranking ability via ranking distillation from zero-cost proxies. arXiv preprint arXiv:2301.09850 (2023)
7. Dong, P., Niu, X., Li, L., Xie, L., Zou, W., Ye, T., Wei, Z., Pan, H.: Prior-guided one-shot neural architecture search. arXiv preprint arXiv:2206.13329 (2022)
8. Gao, C., Chen, Y., Liu, S., Tan, Z., Yan, S.: Adversarialnas: Adversarial neural architecture search for gans. In: CVPR (2020)
9. Gong, X., Chang, S., Jiang, Y., Wang, Z.: Autogan: Neural architecture search for generative adversarial networks. ICCV (2019)
10. Hu, Y., Wang, X., Li, L., Gu, Q.: Improving one-shot nas with shrinking-and-expanding supernet. Pattern Recognition (2021)
11. Krizhevsky, A., Nair, V., Hinton, G.: The cifar-10 dataset. online: http://www. cs. toronto. edu/kriz/cifar. html (2014)
12. Li, L.: Self-regulated feature learning via teacher-free feature distillation. In: ECCV (2022)

13. Li, L., Bao, Y., Dong, P., Yang, C., Li, A., Luo, W., Liu, Q., Xue, W., Guo, Y.: Detkds: Knowledge distillation search for object detectors. In: ICML (2024)
14. Li, L., Dong, P., Li, A., Wei, Z., Yang, Y.: Kd-zero: Evolving knowledge distiller for any teacher-student pairs. NeuIPS (2024)
15. Li, L., Dong, P., Wei, Z., Yang, Y.: Automated knowledge distillation via monte carlo tree search. In: ICCV (2023)
16. Li, L., Jin, Z.: Shadow knowledge distillation: Bridging offline and online knowledge transfer. In: NeuIPS (2022)
17. Li, L., Li, A.: A2-aug: Adaptive automated data augmentation. In: CVPRW (2023)
18. Li, L., Shiuan-Ni, L., Yang, Y., Jin, Z.: Boosting online feature transfer via separable feature fusion. In: IJCNN (2022)
19. Li, L., Shiuan-Ni, L., Yang, Y., Jin, Z.: Teacher-free distillation via regularizing intermediate representation. In: IJCNN (2022)
20. Li, L., Wang, Y., Yao, A., Qian, Y., Zhou, X., He, K.: Explicit connection distillation. In: ICLR (2020)
21. Liu, Y., Yu, S., Lin, T.: Hessian regularization of deep neural networks: A novel approach based on stochastic estimators of hessian trace. Neurocomputing **536**, 13–20 (2023)
22. Shao, S., Dai, X., Yin, S., Li, L., Chen, H., Hu, Y.: Catch-up distillation: You only need to train once for accelerating sampling. arXiv preprint arXiv:2305.10769 (2023)
23. Wang, C., Xu, C., Yao, X., Tao, D.: Evolutionary generative adversarial networks. IEEE Transactions on Evolutionary Computation **23**(6), 921–934 (2019)
24. Wei, Z., Pan, H., Li, L.L., Lu, M., Niu, X., Dong, P., Li, D.: Convformer: Closing the gap between cnn and vision transformers. arXiv preprint arXiv:2209.07738 (2022)
25. Ying, G., He, X., Gao, B., Han, B., Chu, X.: Eagan: Efficient two-stage evolutionary architecture search for gans. In: ECCV. Springer (2022)
26. Zhu, C., Li, L., Wu, Y., Sun, Z.: Saswot: Real-time semantic segmentation architecture search without training. In: AAAI (2024)
27. Zimian Wei, Z., Li, L.L., Dong, P., Hui, Z., Li, A., Lu, M., Pan, H., Li, D.: Auto-prox: Training-free vision transformer architecture search via automatic proxy discovery. In: AAAI (2024)