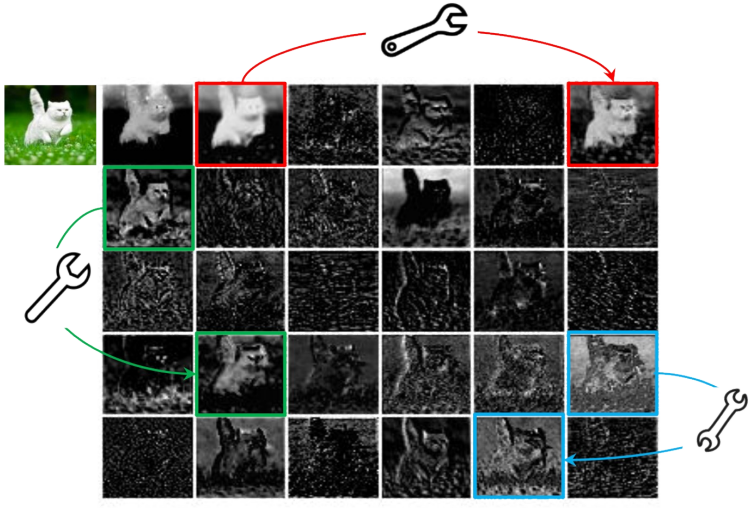
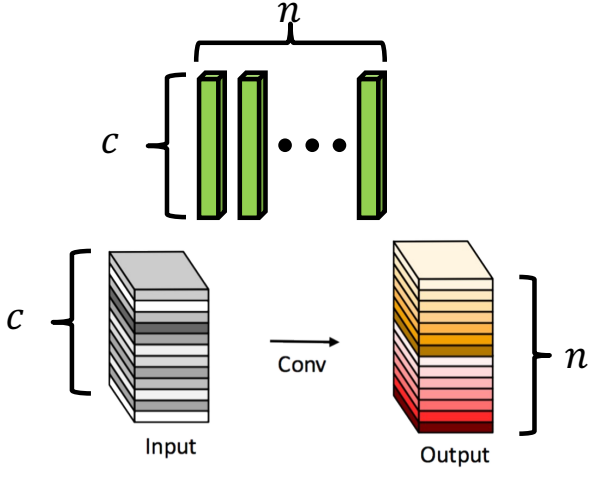
Extension

Ghost Filters

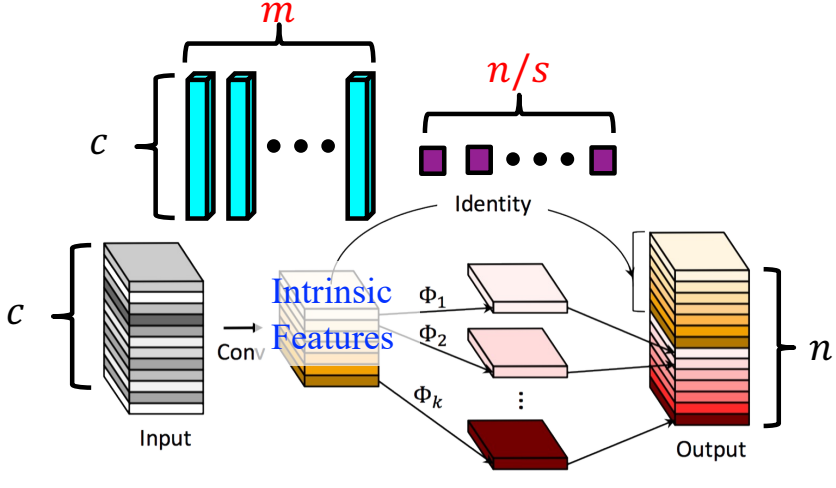
[CVPR 2020]



- Beyond 1×1 pointwise convolution, **the primary convolutions** can have customized kernel size;
- **The secondary convolutions** can be diverse to augment the features and increase the channels.
- **Identity mapping** is included to preserve the intrinsic feature maps.



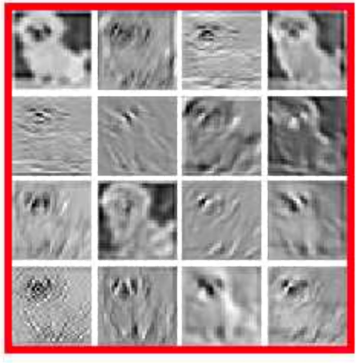
(a) The convolutional layer.



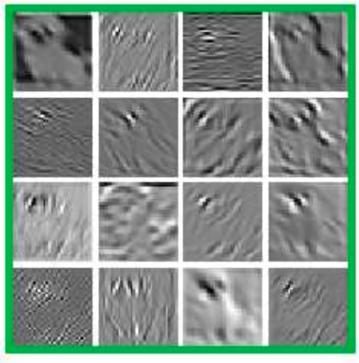
(b) The Ghost module.

Ghost Filters

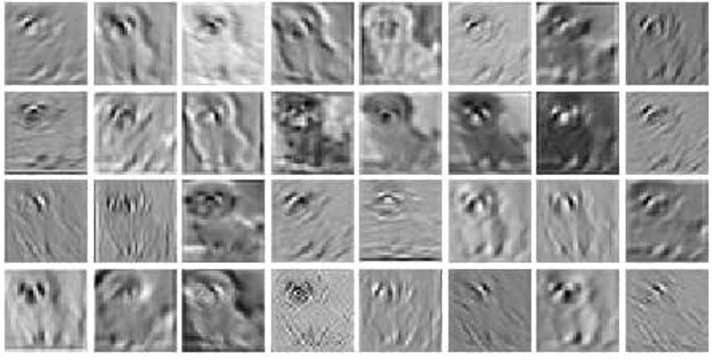
[CVPR 2020]



After primary convolution



After secondary convolution



The 2nd layer of Vanilla VGG16

Table 6. Comparison of state-of-the-art methods for compressing ResNet-50 on ImageNet dataset.

Model	Weights (M)	FLOPs (B)	Top-1 Acc. (%)	Top-5 Acc. (%)
ResNet-50 [16]	25.6	4.1	75.3	92.2
Thinet-ResNet-50 [39]	16.9	2.6	72.1	90.3
NISP-ResNet-50-B [59]	14.4	2.3	-	90.8
Versatile-ResNet-50 [49]	11.0	3.0	74.5	91.8
SSS-ResNet-50 [23]	-	2.8	74.2	91.9
Ghost-ResNet-50 (s=2)	13.0	2.2	**75.0**	**92.3**
Shift-ResNet-50 [53]	6.0	-	70.6	90.1
Taylor-FO-BN-ResNet-50 [41]	7.9	1.3	71.7	-
Slimmable-ResNet-50 0.5× [58]	6.9	1.1	72.1	-
MetaPruning-ResNet-50 [36]	-	1.0	73.4	-
Ghost-ResNet-50 (s=4)	6.5	1.2	**74.1**	**91.9**

Lego Filters

[ICML 2019]

Experience in network engineering:

2 principles

- VGGNets and ResNets **stack building blocks** of the same shape, which reduces the free choices of hyper-parameters.

- **Split-transform-merge** in Inception models - the input is split into a few embeddings of lower dimensionalities, transformed by a set of specialized filters, and merged by concatenation.

Whether these principles are applicable for re-designing ***Filter*** *- the basic unit in deep neural networks?*

(a) conv filters (b) Lego filters (c) stacked filters

Learning to stack Lego Filters:

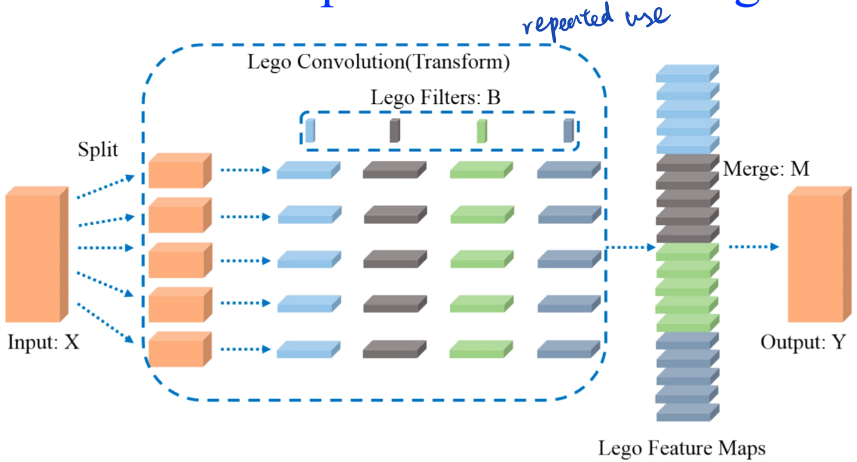
$$\min_{\mathbf{B},\mathbf{M}^j} \sum_{i=1}^{o} \frac{1}{2} ||\mathbf{Y}^j - \mathbf{X}_i^\top (\mathbf{B}\mathbf{M}_i^j)||_F^2,$$

$$s.t.\ \mathbf{M}_i^j \in \{0,1\}^{m\times 1},\ \ ||\mathbf{M}_i^j||_1 = 1, i = 1, ..., o.$$

Lego Filters

[ICML 2019]

An efficient implementation of Lego Filters



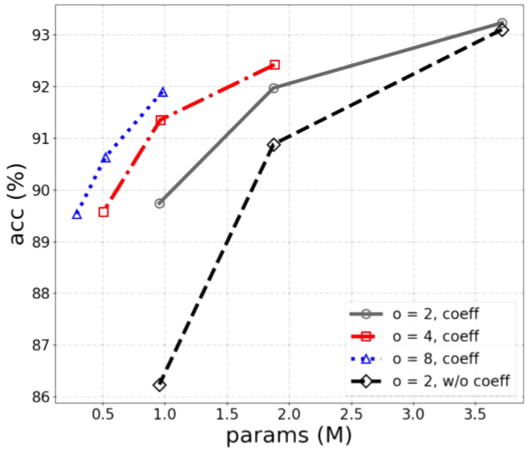


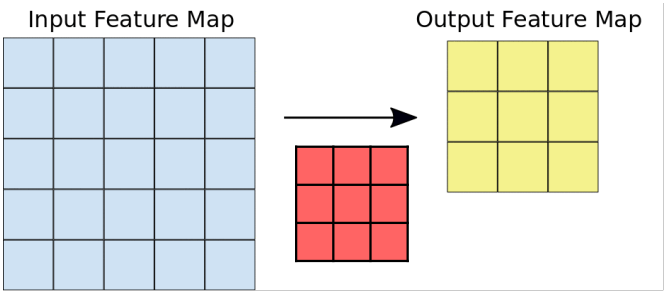
Table 2. Comparison results of different neural networks on the ILSVRC2012 datasets.

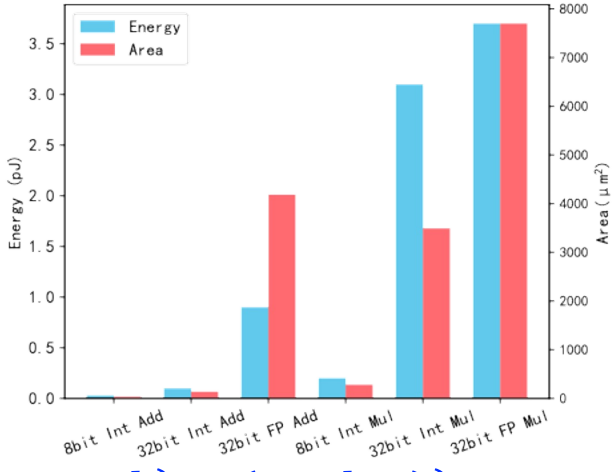
Model	Top-5 Acc(%)	Params(M)	Comp Ratio	FLOPs(B)	Speed Up
ResNet50 (He et al., 2016)	92.2	25.6	1.0×	4.1	1.0×
ThiNet-Res (Luo et al., 2017a)	88.3	8.7	2.9×	2.2	1.9×
Versatile (Wang et al., 2018a)	91.8	11.0	2.3×	3.0	1.4×
Lego-Res50(o=2,m=0.5)	89.7	8.1	3.2×	2.0	2.0×
Lego-Res50-w(o=2,m=0.5)	90.6	8.1	3.2×	2.0	2.0×
Lego-Res50-w(o=2,m=0.6)	91.3	9.3	2.8×	2.0	1.7×
VGGNet-16 (Simonyan and Zisserman, 2014)	90.1	138.0	1.0×	15.3	1.0×
ThiNet-VGG (Luo et al., 2017a)	90.3	38.0	3.6×	3.9	3.9×
Lego-VGGNet-16-w(o=2,m=0.5)	88.9	4.2	32.9×	7.7	2.0×
Lego-VGGNet-16-w(o=2,m=0.6)	89.2	5.0	27.6×	9.2	1.7×
MobileNet (Howard et al., 2017)	88.9	4.2	1.0×	0.6	1.0×
Lego-Mobile-w(o=2,m=0.9)	87.5	2.5	1.7×	0.5	1.1×
Lego-Mobile-w(o=2,m=1.5)	88.3	3.5	1.2×	0.6	1.0×

Adder Filters

[CVPR 2020]

correlation between filter & input element-wise multiplication → cross correlation





$$Y(m.n.t) = \sum_{i=0}^{d}\sum_{j=0}^{d}\sum_{k=0}^{c_{in}} S(X(m+i,n+j,k),F(i,j,k,t))$$

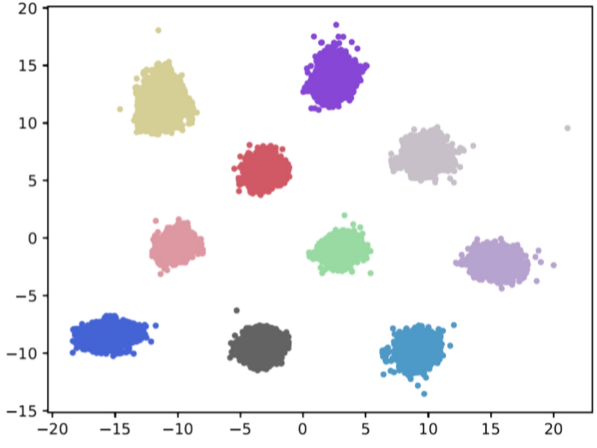
Convolution: $$Y(m.n.t) = \sum_{i=0}^{d}\sum_{j=0}^{d}\sum_{k=0}^{c_{in}} X(m+i,n+j,k)\times F(i,j,k,t)$$

input feature filter

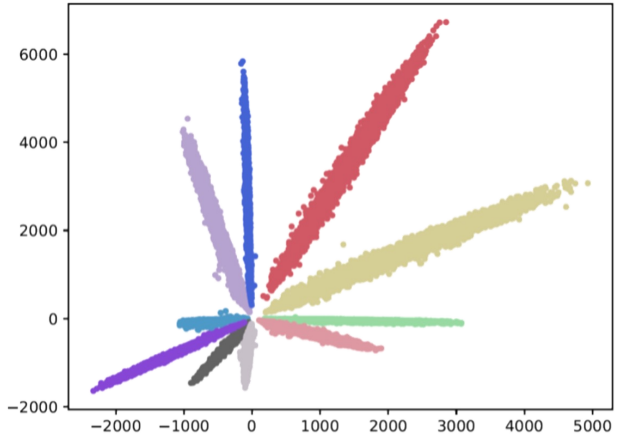
Adder: $$Y(m.n.t) = -\sum_{i=0}^{d}\sum_{j=0}^{d}\sum_{k=0}^{c_{in}} |X(m+i,n+j,k) - F(i,j,k,t)|$$

Adder Filters

[CVPR 2020]



(a) Visualization of features in AdderNets

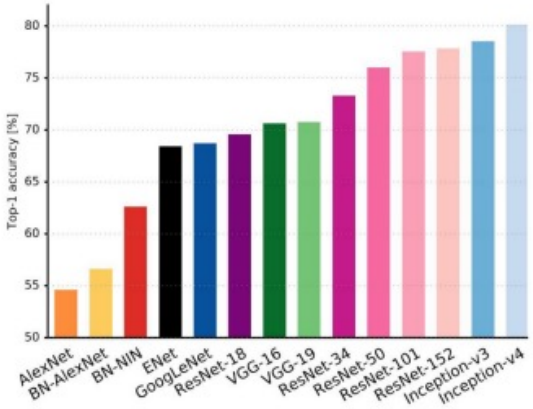


(b) Visualization of features in CNNs

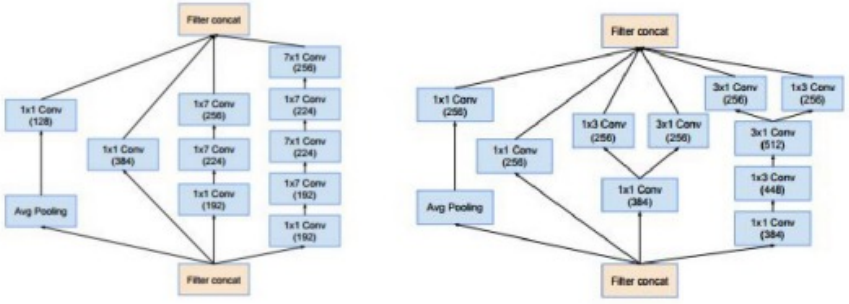
Table 2. Classification results on the CIFAR-10 and CIFAR-100 datasets.

Model	Method	#Mul.	#Add.	XNOR	CIFAR-10	CIFAR-100
VGG-small	BNN	0	0.65G	0.65G	89.80%	65.41%
	AddNN	0	1.30G	0	93.72%	72.64%
	CNN	0.65G	0.65G	0	93.80%	72.73%
ResNet-20	BNN	0	41.17M	41.17M	84.87%	54.14%
	AddNN	0	82.34M	0	91.84%	67.60%
	CNN	41.17M	41.17M	0	92.25%	68.14%
ResNet-32	BNN	0	69.12M	69.12M	86.74%	56.21%
	AddNN	0	138.24M	0	93.01%	69.02%
	CNN	69.12M	69.12M	0	93.29%	69.74%

Neural Architecture Search



Canziani et al (2017)

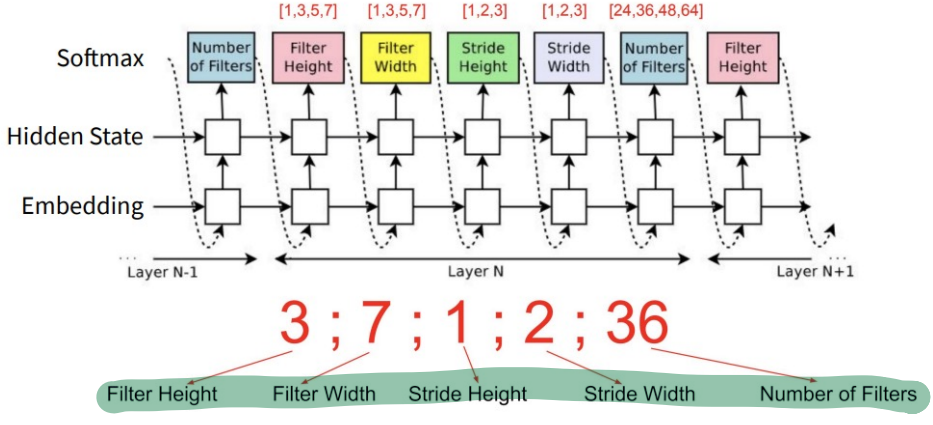


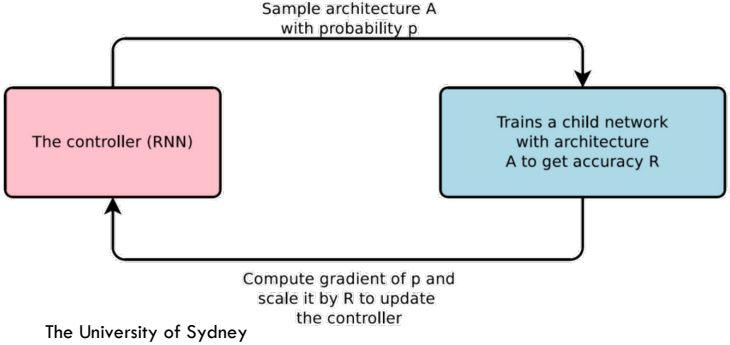
Complex hand-engineered layers from Inception-V4 (Szegedy et al., 2017)

Can we try and learn good architectures automatically?

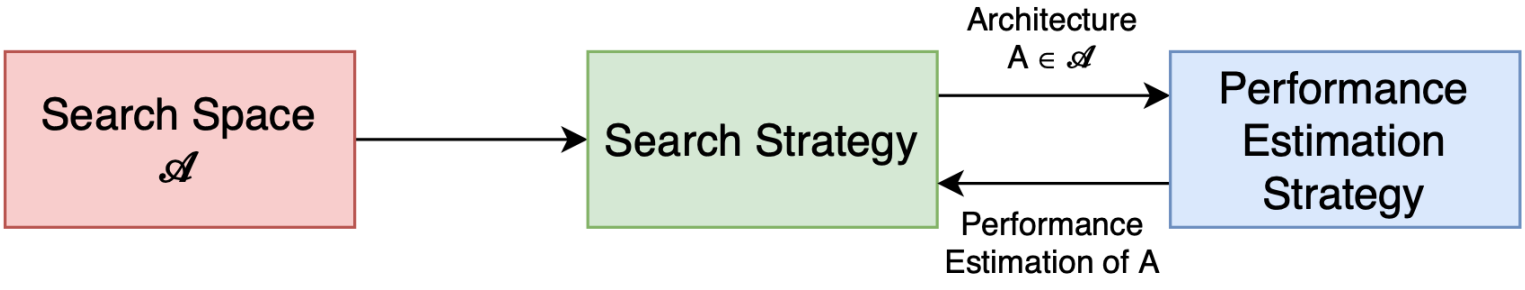
Neural Architecture Search with Reinforcement Learning

Barret Zoph,* Quoc V. Le
Google Brain

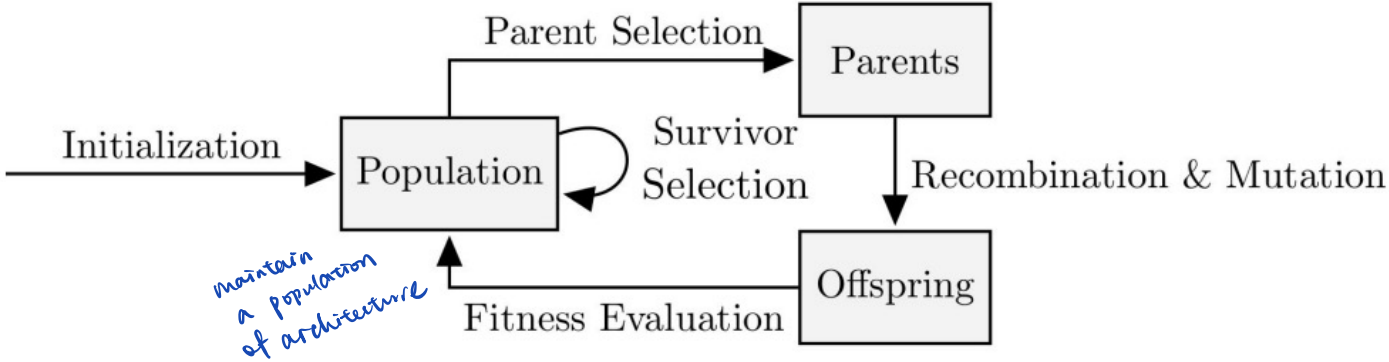


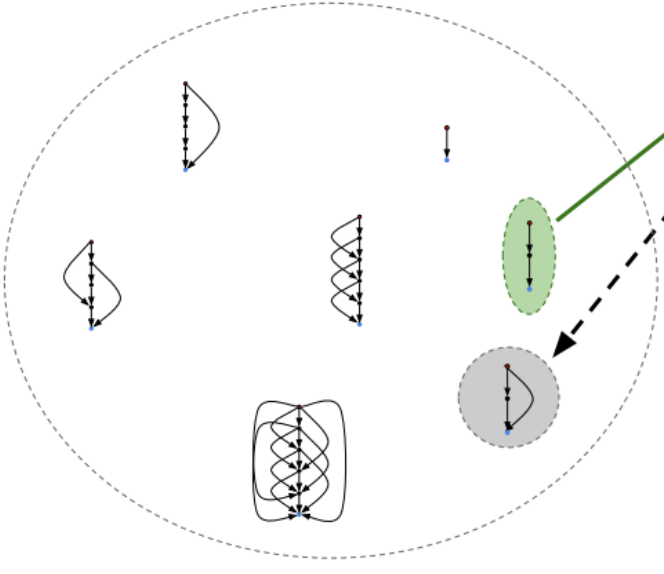


- Specify the structure and connectivity of a neural network by using a configuration string (e.g., [“Filter Width: 5”, “Filter Height: 3”, “Num Filters: 24”])
- Zoph and Le (2017): Use a RNN (“Controller”) to generate this string that specifies a neural network architecture •
- Train this architecture (“Child Network”) to see how well it performs on a validation set •
- Use reinforcement learning to update the parameters of the Controller model based on the accuracy of the child model


Search Space
$\mathscr{A}$
Search Strategy
Architecture
$A \in \mathscr{A}$
Performance
Estimation
Strategy
Performance
Estimation of A

Evolutionary controller



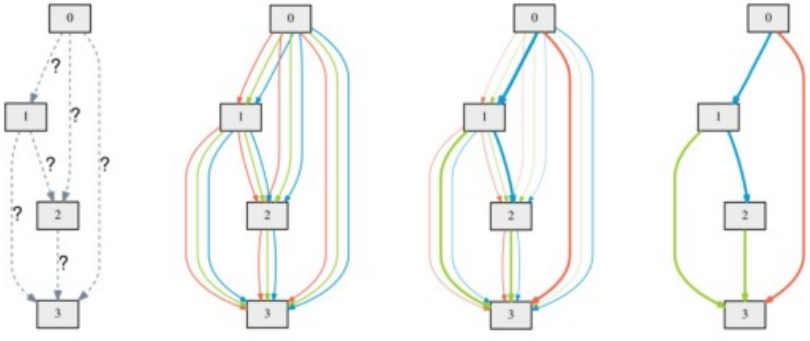


Worker

Possible Mutations

- Insert convolution
- Remove convolution
- Insert nonlinearity
- Remove nonlinearity
- Add-skip
- Remove skip
- Alter strides
- Alter number of channels
- Alter horizontal filter size
- Alter vertical filters size
- Alter Learning Rate
- Identity

Gradient-based NAS with Weight Sharing



DARTS (Liu et al., 2018)

Find encoding weights a^* that

$$\min_{\alpha} \quad \mathcal{L}_{val}(w^*(\alpha), \alpha)$$
$$\text{s.t.} \quad w^*(\alpha) = \text{argmin}_w \; \mathcal{L}_{train}(w, \alpha)$$

α: weights of operation
eg α_1: 1×1 conv
α_2: 3×3 conv. ...

Learn design of normal and reduction cells

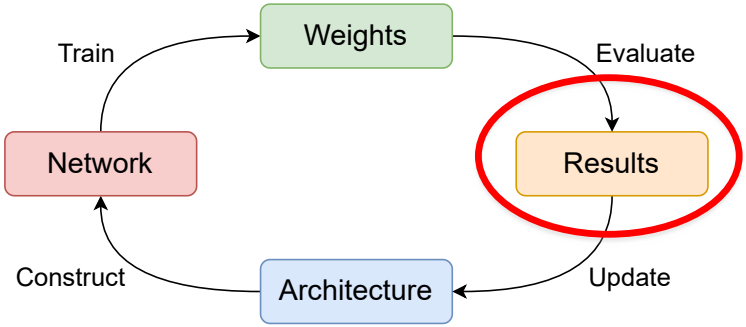
different connections between nodes
node → 1×1, 3×3, 5×5, ... → node

Create a mixed operation $\bar{o}^{(i,j)}$ parametrized by $\alpha^{(i,j)}$ for each edge (i, j)
while *not converged* **do**
1. Update architecture α by descending $\nabla_\alpha \mathcal{L}_{val}(w - \xi \nabla_w \mathcal{L}_{train}(w, \alpha), \alpha)$ ($\xi = 0$ if using first-order approximation)
2. Update weights w by descending $\nabla_w \mathcal{L}_{train}(w, \alpha)$

Derive the final architecture based on the learned α.

Neural Architecture Search in A Proxy Validation Loss Landscape

[ICML 2020]

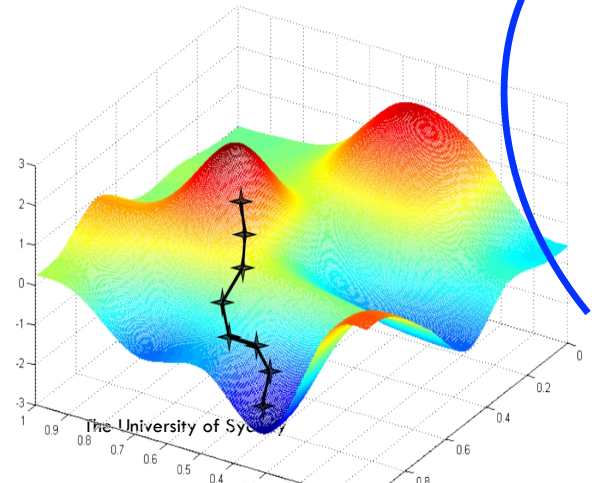


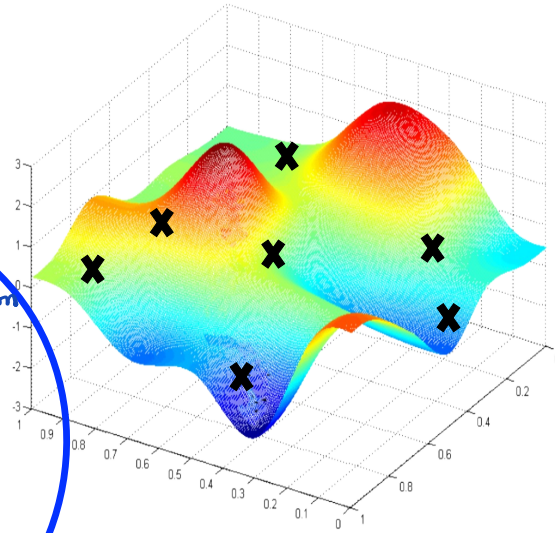
$$\min_{f} \sum_{i=1}^{n} \ell(f(\mathcal{N}_i), y_i)$$

architecture

evaluation

$$\min_{\mathcal{N}} f(\mathcal{N})$$





K-shot NAS: Learnable Weight-Sharing for NAS with K-shot Supernets

[ICML 2021]

sample small Neural network

customize weight for particular sub-network

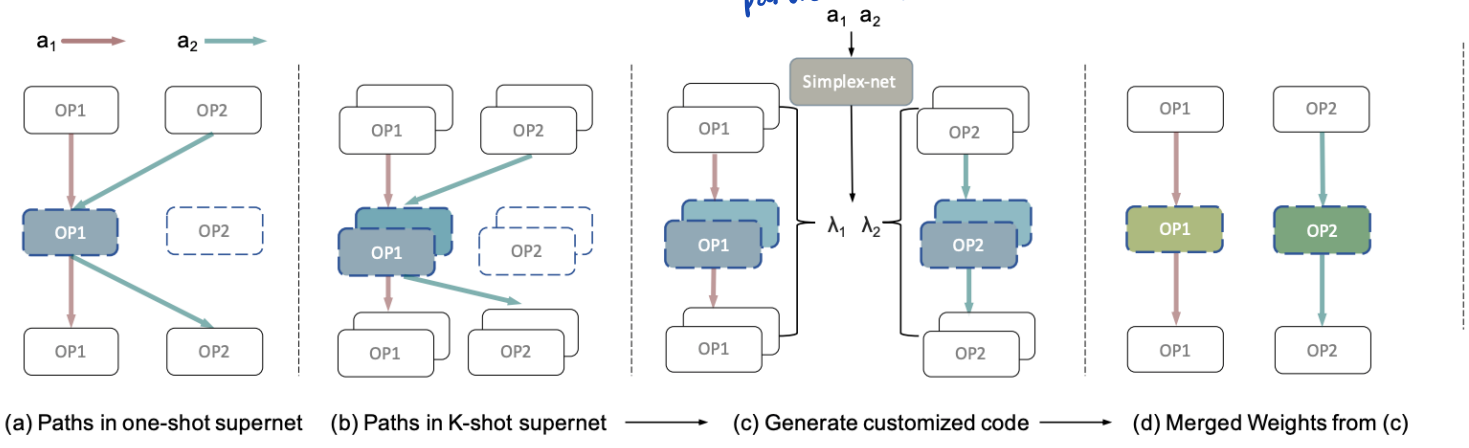


Figure 1. K-shot NAS directly learns the customized code (coefficient vector) $\boldsymbol{\lambda}$ for different paths (subnets or architectures) $\boldsymbol{a}$. Therefore, even with the same operation, *e.g.* OP1 in the second layer, the weights of each path is encouraged to be different from each other as in (d) and be a more accurate approximation for the stand-alone weights trained from scratch. However, as in (a), paths in one-shot supernet fully share a same set of weights even when architectures are different ($\boldsymbol{a}_1$ vs $\boldsymbol{a}_2$).

Prioritized Architecture Sampling with Monto-Carlo Tree Search

[CVPR 2021]

layer may influence its later layers

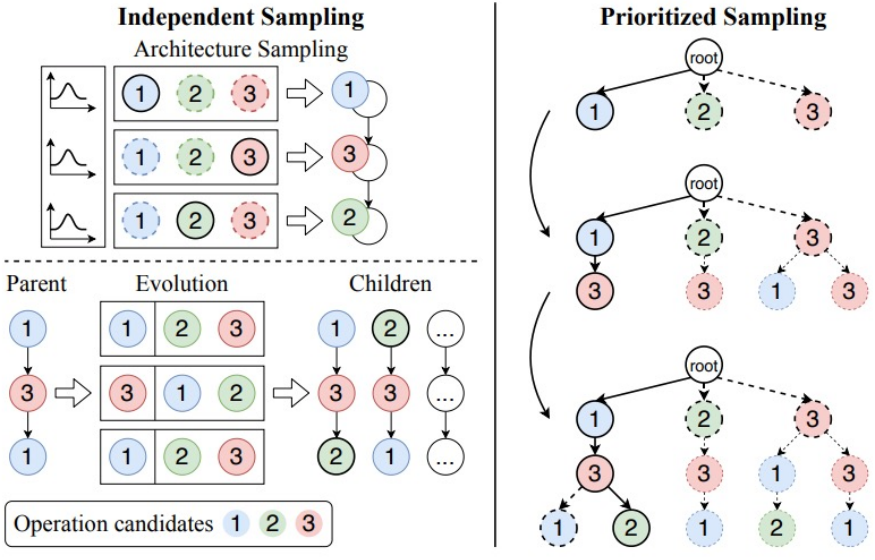


Figure 1. Comparison between existing methods (left) and our method (right). The existing method treats each layer independently in training (top-left) and search (bottom-left) stages, while our method models the search space with dependencies to a unified tree structure.

Existing one-shot NAS methods often consider each layer separately while ignoring the dependencies between the operation choices on different layers.
we introduce a sampling strategy based on Monte

Carlo tree search (MCTS) with the search space modelled as a Monte Carlo tree (MCT), which captures the dependency among layers.

Thank you!