NAT: Neural Architecture Transformer for Accurate and Compact Architectures

Yong Guo[1*], **Yin Zheng**[2*], **Mingkui Tan**[1*†], **Qi Chen**[1],
Jian Chen[1†], **Peilin Zhao**[3], **Junzhou Huang**[3,4]
[1]South China University of Technology, [2]Weixin Group, Tencent,
[3]Tencent AI Lab, [4]University of Texas at Arlington
{guo.yong, sechenqi}@mail.scut.edu.cn, {mingkuitan, ellachen}@scut.edu.cn,
{yinzheng, masonzhao}@tencent.com, jzhuang@uta.edu

Abstract

Designing effective architectures is one of the key factors behind the success of deep neural networks. Existing deep architectures are either manually designed or automatically searched by some Neural Architecture Search (NAS) methods. However, even a well-searched architecture may still contain many non-significant or redundant modules or operations (*e.g.*, convolution or pooling), which may not only incur substantial memory consumption and computation cost but also deteriorate the performance. Thus, it is necessary to optimize the operations inside an architecture to improve the performance without introducing extra computation cost. Unfortunately, such a constrained optimization problem is NP-hard. To make the problem feasible, we cast the optimization problem into a Markov decision process (MDP) and seek to learn a Neural Architecture Transformer (NAT) to replace the redundant operations with the more computationally efficient ones (*e.g.*, skip connection or directly removing the connection). Based on MDP, we learn NAT by exploiting reinforcement learning to obtain the optimization policies w.r.t. different architectures. To verify the effectiveness of the proposed strategies, we apply NAT on both hand-crafted architectures and NAS based architectures. Extensive experiments on two benchmark datasets, *i.e.*, CIFAR-10 and ImageNet, demonstrate that the transformed architecture by NAT significantly outperforms both its original form and those architectures optimized by existing methods.

1 Introduction

Deep neural networks (DNNs) [25] have been producing state-of-the-art results in many challenging tasks including image classification [12, 23, 43, 60, 61, 18, 11, 56], face recognition [39, 44, 59], brain signal processing [34, 35], video analysis [48, 53, 52] and many other areas [58, 57, 24, 3, 10, 9, 4]. One of the key factors behind the success lies in the innovation of neural architectures, such as VGG [41] and ResNet[13]. However, designing effective neural architectures is often labor-intensive and relies heavily on substantial human expertise. Moreover, the human-designed process cannot fully explore the whole architecture space and thus the designed architectures may not be optimal. Hence, there is a growing interest to replace the manual process of architecture design with Neural Architecture Search (NAS).

Recently, substantial studies [30, 36, 64] have shown that the automatically discovered architectures are able to achieve highly competitive performance compared to existing hand-crafted architectures.

[*] Authors contributed equally.

[†] Corresponding author.

[2] This work is done when Yong Guo works as an intern in Tencent AI Lab.

33rd Conference on Neural Information Processing Systems (NeurIPS 2019), Vancouver, Canada.

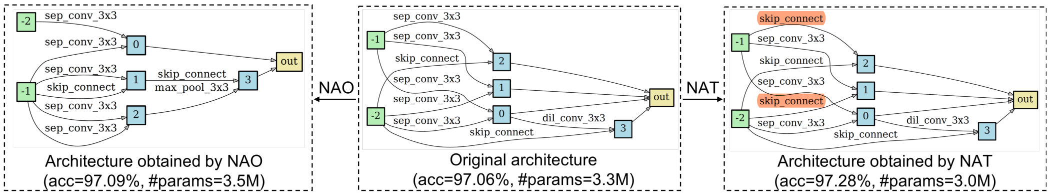


Figure 1: Comparison between Neural Architecture Optimization (NAO) [32] and our Neural Architecture Transformer (NAT). Green blocks denote the two input nodes of the cell and blue blocks denote the intermediate nodes. Red blocks denote the connections that are changed by NAT. The accuracy and the number of parameters are evaluated on CIFAR-10 models.

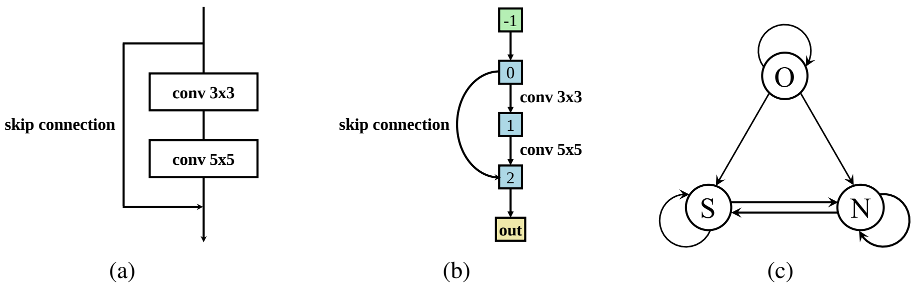
However, there are some limitations in NAS based architecture design methods. In fact, since there is an extremely large search space [36, 64] (*e.g.*, billions of candidate architectures), these methods often produce sub-optimal architectures, leading to limited representation performance or substantial computation cost. Thus, even for a well-designed model, it is necessary yet important to optimize its architecture (*e.g.*, removing the redundant operations) to achieve better performance and/or reduce the computation cost.

To optimize the architectures, Luo *et al.* recently proposed a neural architecture optimization (NAO) method [32]. Specifically, NAO first encodes an architecture into an embedding in continuous space and then conducts gradient descent to obtain a better embedding. After that, it uses a decoder to map the embedding back to obtain an optimized architecture. However, NAO comes with its own set of limitations. First, NAO often produces a totally different architecture from the input one and may introduce extra parameters or additional computation cost. Second, similar to the NAS based methods, NAO has a huge search space, which, however, may not be necessary for the task of architecture optimization and may make the optimization problem very expensive to solve. An illustrative comparison between our method and NAO can be found in Figure 1.

Unlike existing methods that design neural architectures, we seek to design an architecture optimization method, called Neural Architecture Transformer (NAT), to optimize neural architectures. Since the optimization problem is non-trivial to solve, we cast it into a Markov decision process (MDP). Thus, the architecture optimization process is reduced to a series of decision making problems. Based on MDP, we seek to replace the expensive operations or redundant modules in the architecture with more computationally efficient ones. Specifically, NAT shall remove the redundant modules or replace these modules with skip connections. In this way, the search space can be significantly reduced. Thus, the training complexity to learn an architecture optimizer is smaller than those NAS based methods, *e.g.*, NAO. Last, it is worth mentioning that our NAT model can be used as a general architecture optimizer which takes any architecture as the input and output an optimized one. In experiments, we apply NAT to both hand-crafted and NAS based architectures and demonstrate the performance on two benchmark datasets, namely CIFAR-10 [22] and ImageNet [8].

The main contributions of this paper are summarized as follows.

- We propose a novel architecture optimization method, called Neural Architecture Transformer (NAT), to optimize arbitrary architectures in order to achieve better performance and/or reduce computation cost. To this end, NAT either removes the redundant paths or replaces the original operation with skip connection to improve the architecture design.
- We cast the architecture optimization problem into a Markov decision process (MDP), in which we seek to solve a series of decision making problems to optimize the operations. We then solve the MDP problem with policy gradient. To better exploit the adjacency information of operations in an architecture, we propose to exploit graph convolution network (GCN) to build the architecture optimization model.
- Extensive experiments demonstrate the effectiveness of our NAT on both hand-crafted and NAS based architectures. Specifically, for hand-crafted models (*e.g.,* VGG), our NAT automatically introduces additional skip connections into the plain network and results in 2.75% improvement in terms of Top-1 accuracy on ImageNet. For NAS based models (*e.g.,* DARTS [30]), NAT reduces 30% parameters and achieves 1.31% improvement in terms of Top-1 accuracy on ImageNet.



(a) (b) (c)

Figure 2: An example of the graph representation of a residual block and the diagram of operation transformations. (a) a residual block [13]; (b) a graph view of residual block; (c) transformations among three kinds of operations. N denotes a null operation without any computation, S denotes a skip connection, and O denotes some computational modules other than null and skip connections.

2 Related Work

Hand-crafted architecture design. Many studies focus on architecture design and propose a series of deep neural architectures, such as Network-in-network [28], VGG [41], GoogLeNet [45] and so on. Unlike these plain networks that only contain a stack of convolutions, He *et al.* propose the residual network (ResNet) [13] by introducing residual shortcuts between different layers. However, the human-designed process often requires substantial human effort and cannot fully explore the whole architecture space, making the hand-crafted architectures often not optimal.

Neural architecture search. Recently, neural architecture search (NAS) methods have been proposed to automate the process of architecture design [64, 65, 36, 1, 62, 30, 2, 47, 42]. Some researchers conduct architecture search by modeling the architecture as a graph [54, 20]. Unlike these methods, DSO-NAS [55] finds the optimal architectures by starting from a fully connected block and then imposing sparse regularization [17, 46] to prune useless connections. Besides, Jin *et al.* propose a Bayesian optimization approach [19] to morph the deep architectures by inserting additional layers, adding more filters or introducing additional skip connections. More recently, Luo *et al.* propose the neural architecture optimization (NAO) [32] method to perform the architecture search on continuous space by exploiting the encoding-decoding technique. However, NAO is essentially designed for architecture search and often obtains very different architectures from the input architectures and may introduce extra parameters. Unlike these methods, our method is able to optimize architectures without introducing extra computation cost (See the detailed comparison in Figure 1).

Architecture adaptation and model compression. Several methods [51, 26, 7, 6] have been proposed to obtain compact architectures by learning the optimal settings of each convolution, including kernel size, stride and the number of filters. To obtain compact models, model compression methods [27, 15, 31, 63] detect and remove the redundant channels from the original models. However, these methods only change the settings of convolution but ignore the fact that adjusting the connections in the architecture could be more critical. Recently, Cao *et al.* propose an automatic architecture compression method [5]. However, this method has to learn a compressed model for each given pre-trained model and thus has limited generalization ability to different architectures. Unlike these methods, we seek to learn a general optimizer for any arbitrary architecture.

3 Neural Architecture Transformer

3.1 Problem Definition

Given an architecture space Ω, we can represent an architecture α as a directed acyclic graph (DAG), *i.e.*, $\alpha = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is a set of nodes that denote the feature maps in DNNs and $\mathcal{E}$ is an edge set [64, 36, 30], as shown in Figure 2. Here, the directed edge $e_{ij} \in \mathcal{E}$ denotes some operation (*e.g.,* convolution or max pooling) that transforms the feature map from node v_i to v_j. For convenience, we divide the edges in $\mathcal{E}$ into three categories, namely, S, N, O, as shown in Figure 2(c). Here, S denotes the skip connection, N denotes the null connection (*i.e.*, no edge between two nodes), and O denotes the operations other than skip connection or null connection (*e.g.*, convolution or max pooling). Note that different operations have different costs. Specifically, let $c(\cdot)$ be a function to evaluate the computation cost. Obviously, we have $c(O) > c(S) > c(N)$.

In this paper, we seek to design an architecture optimization method, called Neural Architecture Transformer (NAT), to optimize any given architecture into a better one with the improved performance and/or less computation cost. To achieve this, an intuitive way is to make the original operation with less computation cost, *e.g.*, using the skip connection to replace convolution or using the null connection to replace skip connection. Although the skip connection has slightly higher cost than the null connection, it often can significantly improve the performance [13, 14]. Thus, we enable the transition from null connection to skip connection to increase the representation ability of deep networks. In summary, we constrain the possible transitions among O, S, and N in Figure 2(c) in order to reduce the computation cost.

Note that the architecture optimization on an entire network is still very computationally expensive. Moreover, we hope to learn a general architecture optimizer. Given these two concerns, we consider learning a computational cell as the building block of the final architecture. To build a cell, we follow the same settings as that in ENAS [36]. Specifically, each cell has two input nodes, *i.e.*, v_{-2} and v_{-1}, which denote the outputs of the second nearest and the nearest cell in front of the current one, respectively. Each intermediate node (marked as the blue box in Figure 1) also takes two previous nodes in this cell as inputs. Last, based on the learned cell, we are able to form any final network.

3.2 Markov Decision Process for Architecture Optimization

In this paper, we seek to learn a general architecture optimizer $\alpha = \mathrm{NAT}(\beta;\theta)$, which transforms any β into an optimized α and is parameterized by θ. Here, we assume β follows some distribution $p(\cdot)$, *e.g.*, multivariate uniformly discrete distribution. Let w_α and w_β be the well-learned model parameters of architectures α and β, respectively. We measure the performance of α and β by some metric $R(\alpha, w_\alpha)$ and $R(\beta, w_\beta)$, *e.g.*, the accuracy on validation data. For convenience, we define the performance improvement between α and β by $R(\alpha|\beta) = R(\alpha, w_\alpha) - R(\beta, w_\beta)$.

To learn a good transformer $\alpha = \mathrm{NAT}(\beta;\theta)$ to optimize arbitrary β, we can maximize the expectation of performance improvement $R(\alpha|\beta)$ over the distribution of β under a constraint of computation cost $c(\alpha) \leq \kappa$, where $c(\alpha)$ measures the cost of α and κ is an upper bound of the cost. Then, the optimization problem can be written as

$$\max_{\theta} \ \mathbb{E}_{\beta \sim p(\cdot)} \left[R\left(\alpha|\beta\right) \right], \ \text{s.t.} \ \ c(\alpha) \leq \kappa. \tag{1}$$

Unfortunately, it is non-trivial to directly obtain the optimal α given different β. Nevertheless, following [64, 36], given any architecture β, we instead sample α from some well learned policy, denoted by $\pi(\cdot|\beta;\theta)$, namely $\alpha \sim \pi(\cdot|\beta;\theta)$. In other words, NAT first learns the policy and then conducts sampling from it to obtain the optimized architecture. In this sense, the parameters to be learned only exist in $\pi(\cdot|\beta;\theta)$. To learn the policy, we solve the following optimization problem:

$$\max_{\theta} \ \mathbb{E}_{\beta \sim p(\cdot)} \left[\mathbb{E}_{\alpha \sim \pi(\cdot|\beta;\theta)} \ R\left(\alpha|\beta\right) \right], \ \text{s.t.} \ \ c(\alpha) \leq \kappa, \ \alpha \sim \pi(\cdot|\beta;\theta), \tag{2}$$

where $\mathbb{E}_{\beta \sim p(\cdot)} \left[\cdot\right]$ and $\mathbb{E}_{\alpha \sim \pi(\cdot|\beta;\theta)} \left[\cdot\right]$ denote the expectation operation over β and α, respectively.

This problem, however, is still very challenging to solve. **First**, the computation cost of deep networks can be evaluated by many metrics, such as the number of multiply-adds (MAdds), latency, and energy consumption, making it hard to find a comprehensive measure to accurately evaluate the cost. **Second**, the upper bound of computation cost κ in Eqn. (1) may vary for different cases and thereby is hard to determine. Even if there already exists a specific upper bound, dealing with the constrained optimization problem is still a typical NP-hard problem. **Third**, how to compute $\mathbb{E}_{\beta \sim p(\cdot)} \left[\mathbb{E}_{\alpha \sim \pi(\cdot|\beta;\theta)} \ R\left(\alpha|\beta\right) \right]$ remains a question.

To address the above challenges, we cast the optimization problem into an architecture transformation problem and reformulate it as a Markov decision process (MDP). Specifically, we optimize architectures by making a series of decisions to alternate the types of different operations. Following the transition graph in Figure 2(c), as $c(O) > c(S) > c(N)$, we can naturally obtain more compact architectures than the given ones. In this sense, we can achieve the goal to optimize arbitrary architecture without introducing extra cost into the architecture. Thus, for the first two challenges, we do not have to evaluate the cost $c(\alpha)$ or determine the upper bound κ. For the third challenge, we estimate the expectation value by sampling architectures from $p(\cdot)$ and $\pi(\cdot|\beta;\theta)$ (See details in Section 3.4).

MDP formulation details. A typical MDP [40] is defined by a tuple $(\mathcal{S}, \mathcal{A}, P, R, q, \gamma)$, where $\mathcal{S}$ is a finite set of states, $\mathcal{A}$ is a finite set of actions, $P : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow \mathbb{R}$ is the state transition distribution, $R : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ is the reward function, $q : \mathcal{S} \rightarrow [0, 1]$ is the distribution of initial state, and $\gamma \in [0, 1]$ is a discount factor. Here, we define an architecture as a state, a transformation mapping $\beta \rightarrow \alpha$ as an action. Here, we use the accuracy improvement on the validation set as the reward. Since the problem is a one-step MDP, we can omit the discount factor γ. Based on the problem definition, we transform any β into an optimized architecture α with the policy $\pi(\cdot|\beta;\theta)$. Then, the main challenge becomes how to learn an optimal policy $\pi(\cdot|\beta;\theta)$. Here, we exploit reinforcement learning [49] to solve the problem and propose an efficient policy learning algorithm.

Search space of α over a cell structure. For a cell structure with B nodes and three states for each edge, there are $(B{-}2){\times}2$ edges and the size of the search space is $|\Omega_\beta| = 3^{2(B-2)}$. Thus, NAT has a much smaller search space than NAS methods [36, 64], *i.e.*, $k^{2(B-2)}(B-2)!$, where k denotes the number of candidate operations (*e.g.*, $k = 5$ in ENAS [36] and $k = 8$ in DARTS [30]).

3.3 Policy Learning by Graph Convolutional Neural Networks

To learn the optimal policy $\pi(\cdot|\beta;\theta)$ w.r.t. an arbitrary architecture β, we propose an effective learning method to optimize the operations inside the architecture. Specifically, we take an arbitrary architecture graph β as the input and output the optimization policy w.r.t β. Such a policy is used to optimize the operations of the given architecture. Since the choice of operation on an edge depends on the adjacent nodes and edges, we have to consider the attributes of both the current edge and its neighbors. For this reason, we employ a graph convolution networks (GCN) [21] to exploit the adjacency information of the operations in the architecture. Here, an architecture graph can be represented by a data pair $(\mathbf{A}, \mathbf{X})$, where $\mathbf{A}$ denotes the adjacency matrix of the graph and $\mathbf{X}$ denotes the attributes of the nodes together with their two input edges[3]. We consider a two-layer GCN and formulate the model as:

$$\mathbf{Z} = f(\mathbf{X}, \mathbf{A}) = \mathrm{Softmax}\left(\mathbf{A}\sigma\left(\mathbf{A}\mathbf{X}\mathbf{W}^{(0)}\right)\mathbf{W}^{(1)}\mathbf{W}^{\mathrm{FC}}\right), \tag{3}$$

where $\mathbf{W}^{(0)}$ and $\mathbf{W}^{(1)}$ denote the weights of two graph convolution layers, $\mathbf{W}^{\mathrm{FC}}$ denotes the weight of the fully-connected layer, σ is a non-linear activation function (*e.g.*, the Rectified Linear Unit (ReLU) [33]), and $\mathbf{Z}$ refers to the probability distribution of different candidate operations on the edges, *i.e.*, the learned policy $\pi(\cdot|\beta;\theta)$. For convenience, we denote $\theta = \{\mathbf{W}^{(0)}, \mathbf{W}^{(1)}, \mathbf{W}^{\mathrm{FC}}\}$ as the parameters of the architecture transformer. To cover all possible architectures, we randomly sample architectures from the whole architecture space and use them to train our model.

Differences with LSTM. The architecture graph can also be processed by the long short-term memory (LSTM) [16], which is a common practice in NAS methods [32, 64, 36]. In these methods, LSTM first treats the graph as a sequence of tokens and then learns the information from the sequence. However, turning a graph into a sequence of tokens may lose some connectivity information of the graph, leading to limited performance. On the contrary, our GCN model can better exploit the information from the graph and yield superior performance (See results in Section 4.4).

3.4 Training and Inference of NAT

We apply the policy gradient [49] to train our model. The overall scheme is shown in Algorithm 1, which employs an alternating manner. Specifically, in each training epoch, we first train the model parameters w with fixed transformer parameters θ. Then, we train the transformer parameters θ by fixing the model parameters w.

Training the model parameters w. Given any θ, we need to update the model parameters w based on the training data. Here, to accelerate the training process, we adopt the parameter sharing technique [36], *i.e.*, we construct a large computational graph, where each subgraph represents a neural network architecture, hence forcing all architectures to share the parameters. Thus, we can use the shared parameters w to represent the parameters for different architectures. For any architecture $\beta \sim p(\cdot)$, let $\mathcal{L}(\beta, w)$ be the loss function on the training data, *e.g.*, the cross-entropy loss. Then, given any m sampled architectures, the updating rule for w with parameter sharing can be given by $w \leftarrow w - \eta \frac{1}{m} \sum_{i=1}^{m} \nabla_w \mathcal{L}(\beta_i, w)$, where η is the learning rate.

[3] Due to the page limit, we put the detailed representation methods in the supplementary.

Algorithm 1 Training method for Neural Architecture Transformer (NAT).

```
Require: The number of sampled input architectures in an iteration $m$, the number of sampled optimized architectures for each input architecture $n$, learning rate $\eta$, regularizer parameter $\lambda$ in Eqn. (4), input architecture distribution $p(\cdot)$, shared model parameters $w$, transformer parameters $\theta$.
Initiate $w$ and $\theta$.
while not convergent do
    for each iteration on training data do
        // Fix $\theta$ and update $w$.
        Sample $\beta_i \sim p(\cdot)$ to construct a batch $\{\beta_i\}_{i=1}^m$.
        Update the model parameters $w$ by descending the gradient:
            $w \leftarrow w - \eta \frac{1}{m} \sum_{i=1}^m \nabla_w \mathcal{L}(\beta_i, w)$.
    end for
    for each iteration on validation data do
        // Fix $w$ and update $\theta$.
        Sample $\beta_i \sim p(\cdot)$ to construct a batch $\{\beta_i\}_{i=1}^m$.
        Obtain $\{\alpha_j\}_{j=1}^n$ according to the policy learned by GCN.
        Update the transformer parameters $\theta$ by ascending the gradient:
            $\theta \leftarrow \theta + \eta \frac{1}{mn} \sum_{i=1}^m \sum_{j=1}^n \left[ \nabla_\theta \log \pi(\alpha_j|\beta_i;\theta) \big( R(\alpha_j, w) - R(\beta_i, w) \big) + \lambda \nabla_\theta H\big(\pi\left(\cdot|\beta_i;\theta\right)\big) \right]$.
    end for
end while
```

Training the transformer parameters θ. We train the transformer model with policy gradient [49]. To encourage exploration, we introduce an entropy regularization term into the objective to prevent the transformer from converging to a local optimum too quickly [65], *e.g.*, selecting the "original" option for all the operations. Given the shared parameters w, the objective can be formulated as

$$\begin{aligned} J(\theta) &= \mathbb{E}_{\beta \sim p(\cdot)} \left[\mathbb{E}_{\alpha \sim \pi(\cdot|\beta;\theta)} \left[R\left(\alpha, w\right) - R\left(\beta, w\right) \right] + \lambda H\big(\pi(\cdot|\beta;\theta)\big) \right] \\ &= \sum_{\beta} p(\beta) \left[\sum_{\alpha} \pi(\alpha|\beta;\theta) \big(R\left(\alpha, w\right) - R\left(\beta, w\right) \big) + \lambda H\big(\pi(\cdot|\beta;\theta)\big) \right]. \end{aligned} \tag{4}$$

where $p(\beta)$ is the probability to sample some architecture β from the distribution $p(\cdot)$, $\pi(\alpha|\beta;\theta)$ is the probability to sample some architecture α from the distribution $\pi(\cdot|\beta;\theta)$, $H(\cdot)$ evaluates the entropy of the policy, and λ controls the strength of the entropy regularization term. For each input architecture, we sample n optimized architectures $\{\alpha_j\}_{j=1}^n$ from the distribution $\pi(\cdot|\beta;\theta)$ in each iteration. Thus, the gradient of Eqn. (4) w.r.t. θ becomes[4]

$$\nabla_\theta J(\theta) \approx \frac{1}{mn} \sum_{i=1}^m \sum_{j=1}^n \left[\nabla_\theta \log \pi(\alpha_j|\beta_i;\theta) \big(R(\alpha_j, w) - R(\beta_i, w) \big) + \lambda \nabla_\theta H\big(\pi(\cdot|\beta_i;\theta)\big) \right]. \tag{5}$$

The regularization term $H\big(\pi(\cdot|\beta_i;\theta)\big)$ encourages the distribution $\pi(\cdot|\beta;\theta)$ to have high entropy, *i.e.*, high diversity in the decisions on the edges. Thus, the decisions for some operations would be encouraged to choose the "identity" or "null" operations during training. As a result, NAT is able to sufficiently explore the whole search space to find the optimal architecture.

Inference the optimized architecture. We do not explicitly obtain the optimized architecture via $\alpha = \mathrm{NAT}(\beta;\varpi)$. Instead, we conduct sampling according to the learned probability distribution. Specifically, we first sample several candidate optimized architectures from the learned policy $\pi(\cdot|\beta;\theta)$ and then select the architecture with the highest validation accuracy. Note that we can also obtain the optimized architecture by selecting the operation with the maximum probability, which, however, tends to reach a local optimum and yields worse results than the sampling based method (See comparisons in Section 4.4).

4 Experiments

In this section, we apply NAT on both hand-crafted and NAS based architectures, and conduct experiments on two image classification benchmark datasets, *i.e.*, CIFAR-10 [22] and ImageNet [8]. All implementations are based on PyTorch.[5]

[4] We put the derivations of Eqn. (5) in the supplementary.

[5] The source code of NAT is available at https://github.com/guoyongcs/NAT.

Table 1: Performance comparisons of the optimized architectures obtained by different methods based on hand-crafted architectures. "/" denotes the original models that are not changed by architecture optimization methods.

CIFAR-10					ImageNet					
Model	Method	#Params (M)	#MAdds (M)	Acc. (%)	Model	Method	#Params (M)	#MAdds (M)	Acc. (%) Top-1	Acc. (%) Top-5
VGG16	/	15.2	313	93.56	VGG16	/	138.4	15620	71.6	90.4
	NAO[32]	19.5	548	95.72		NAO [32]	147.7	18896	72.9	91.3
	NAT	15.2	315	**96.04**		NAT	138.4	15693	**74.3**	**92.0**
ResNet20	/	0.3	41	91.37	ResNet18	/	11.7	1580	69.8	89.1
	NAO [32]	0.4	61	92.44		NAO [32]	17.9	2246	70.8	89.7
	NAT	0.3	42	**92.95**		NAT	11.7	1588	**71.1**	**90.0**
ResNet56	/	0.9	127	93.21	ResNet50	/	25.6	3530	76.2	92.9
	NAO [32]	1.3	199	95.27		NAO [32]	34.8	4505	77.4	93.2
	NAT	0.9	129	**95.40**		NAT	25.6	3547	**77.7**	**93.5**
MobileNetV2	/	2.3	91	94.47	MobileNetV2	/	3.4	300	72.0	90.3
	NAO [32]	2.9	131	94.75		NAO [32]	4.5	513	72.2	90.6
	NAT	2.3	92	**95.17**		NAT	3.4	302	**72.5**	**91.0**

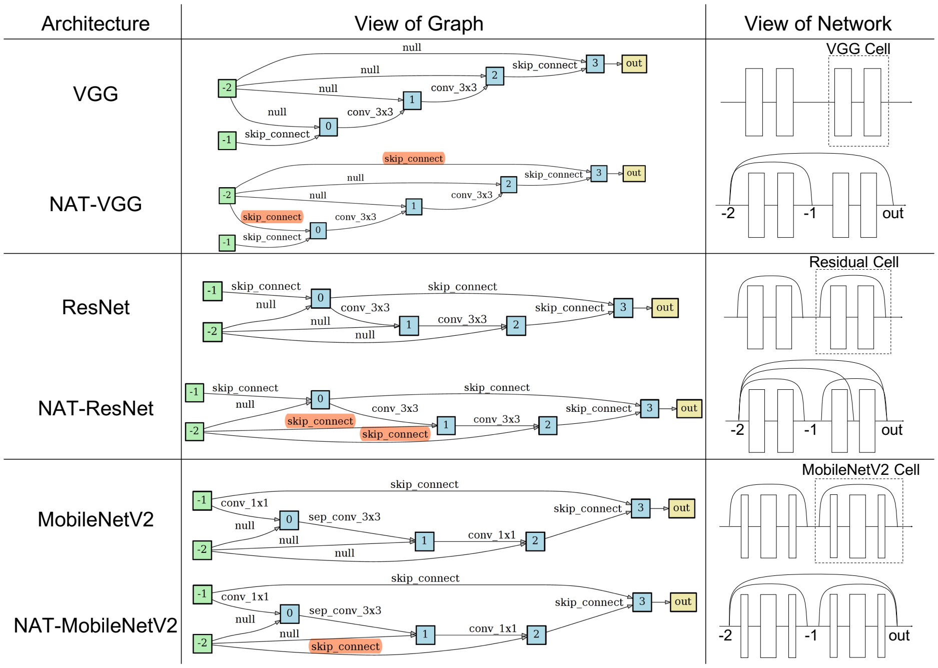


Figure 3: Architecture optimization results of hand-crafted architectures. We provide both the views of graph (left) and network (right) to show the differences in architecture.

4.1 Implementation Details

We consider two kinds of cells in a deep network, including the normal cell and the reduction cell. The normal cell preserves the same spatial size as inputs while the reduction cell reduces the spatial size by $2\times$. Both the normal and reduction cells contain 2 input nodes and a number of intermediate nodes. During training, we build the deep network by stacking 8 basic cells and train the transformer for 100 epochs. We set $m = 1$, $n = 1$, and $\lambda = 0.003$ in the training. We split CIFAR-10 training set into 40% and 60% slices to train the model parameters w and the transformer parameters θ, respectively. As for the evaluation of the networks with different architectures, we replace the original cell with the optimized one and train the model from scratch. Please see more details in the supplementary. For all the considered architectures, we follow the same settings of the original papers. In the experiments, we only apply cutout to the NAS based architectures on CIFAR-10.

4.2 Results on Hand-crafted Architectures

In this experiment, we apply NAT on three popular hand-crafted models, *i.e.*, VGG [41], ResNet [13], and MobileNet [38]. To make all architectures share the same graph representation method defined in Section 3.2, we add null connections into the hand-crafted architectures to ensure that each node has

Table 2: Comparisons of the optimized architectures obtained by different methods based on NAS based architectures. "-" denotes that the results are not reported. "/" denotes the original models that are not changed by architecture optimization methods. † denotes the models trained with cutout.

CIFAR-10					ImageNet					
Model	Method	#Params (M)	#MAdds (M)	Acc. (%)	Model	Method	#Params (M)	#MAdds (M)	Acc. (%)	
									Top-1	Top-5
AmoebaNet† [37]	/	3.2	-	96.73	AmoebaNet [37]	/	5.1	555	74.5	92.0
PNAS† [29]		3.2	-	96.67	PNAS [29]		5.1	588	74.2	91.9
SNAS† [50]		2.9	-	97.08	SNAS [50]		4.3	522	72.7	90.8
GHN† [54]		5.7	-	97.22	GHN [54]		6.1	569	73.0	91.3
ENAS† [36]	/	4.6	804	97.11	ENAS [36]	/	5.6	679	73.8	91.7
	NAO [32]	4.5	763	97.05		NAO [32]	5.5	656	73.7	91.7
	NAT	4.6	804	**97.24**		NAT	5.6	679	**73.9**	**91.8**
DARTS† [30]	/	3.3	533	97.06	DARTS [30]	/	5.9	595	73.1	91.0
	NAO [32]	3.5	577	97.09		NAO [32]	6.1	627	73.3	91.1
	NAT	3.0	483	**97.28**		NAT	3.9	515	**74.4**	**92.2**
NAONet† [32]	/	128	66016	97.89	NAONet [32]	/	11.35	1360	74.3	91.8
	NAO [32]	143	73705	97.91		NAO [32]	11.83	1417	74.5	92.0
	NAT	113	58326	**98.01**		NAT	8.36	1025	**74.8**	**92.3**

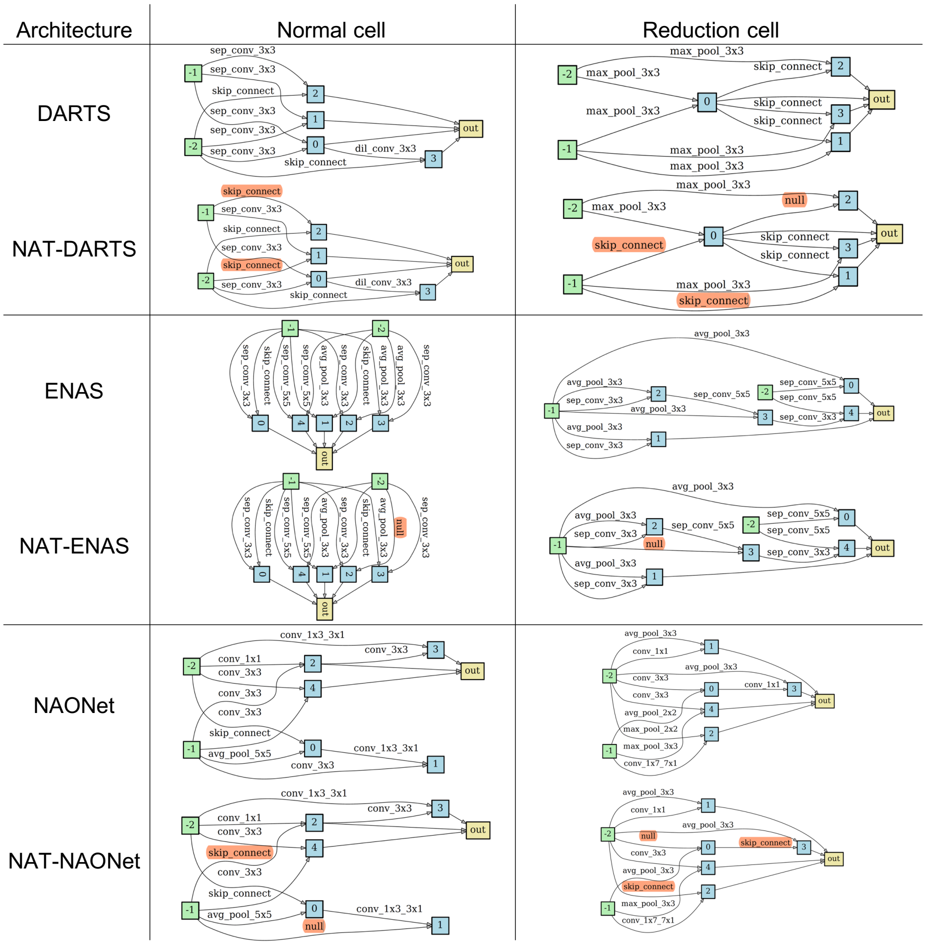


Figure 4: Architecture optimization results on the architectures of NAS based architectures.

two input nodes (See examples in Figure 3). For a fair comparison, we build deep networks using the original and optimized architectures while keeping the same depth and number of channels as the original models. We compare NAT with a strong baseline method Neural Architecture Optimization (NAO) [32]. We show the results in Table 1 and the corresponding architectures in Figure 3. From Table 1, although the models with NAO yield better performance than the original ones, they often have more parameters and higher computation cost. By contrast, our NAT based models consistently outperform the original models by a large margin with approximately the same computation cost.

Table 3: Performance comparisons of the architectures obtained by different methods on CIFAR-10. The reported accuracy (%) is the average performance of five runs with different random seeds. "/" denotes the original models that are not changed by architecture optimization methods. † denotes the models trained with cutout.

Method	VGG16	ResNet20	MobileNetV2	ENAS†	DARTS†	NAONet†
/	93.56	91.37	94.47	97.11	97.06	97.89
Random Search	93.17	91.56	94.38	96.58	95.17	96.31
LSTM	94.45	92.19	95.01	97.05	97.05	97.93
Maximum-GCN	94.37	92.57	94.87	96.92	97.00	97.90
Sampling-GCN (Ours)	**95.93**	**92.97**	**95.13**	**97.21**	**97.26**	**97.99**

4.3 Results on NAS Based Architectures

For the automatically searched architectures, we evaluate the proposed NAT on three state-of-the-art NAS based architectures, *i.e.*, DARTS [30], NAONet [32], and ENAS [36]. Moreover, we also compare our optimized architectures with other NAS based architectures, including AmoebaNet [37], PNAS [29], SNAS [50] and GHN [54]. From Table 2, all the NAT based architectures yield higher accuracy than their baseline models and the models optimized by NAO on CIFAR-10 and ImageNet. Compared with other NAS based architectures, our NAT-DARTS performs the best on CIFAR-10 and achieves the competitive performance compared to the best architecture (*i.e.*, AmoebaNet) on ImageNet with less computation cost and fewer number of parameters. We also visualize the architectures of the original and optimized cell in Figure 4. As for DARTS and NAONet, NAT replaces several redundant operations with the skip connections or directly removes the connection, leading to fewer number of parameters. While optimizing ENAS, NAT removes the average pooling operation and improves the performance without introducing extra computations.

4.4 Comparisons of Different Policy Learners

In this experiment, we compare the performance of different policy learners, including Random Search, LSTM, and the GCN method. For the Random Search method, we perform random transitions among O, S, and N on the input architectures. For the GCN method, we consider two variants which infer the optimized architecture by sampling from the learned policy (denoted by Sampling-GCN) or by selecting the operation with the maximum probability (denoted by Maximum-GCN). From Table 3, our Sampling-GCN method outperforms all the considered policies on different architectures. These results demonstrate the superiority of the proposed GCN method as the policy learner.

4.5 Effect of Different Graph Representations on Hand-crafted Architectures

In this experiment, we investigate the effect of different graph representations on hand-crafted architectures. Note that an architecture may correspond to many different topological graphs, especially for the hand-crafted architectures, *e.g.*, VGG and ResNet, where the number of nodes is smaller than that of our basic cell. For convenience, we study three different graphs for VGG and ResNet20, respectively. The average accuracy of NAT-VGG is 95.83% and outperforms the baseline VGG with the accuracy of 93.56%. Similarly, our NAT-ResNet20 yields the average accuracy of 92.48%, which is also better than the original model. We put the architecture and the performance of each possible representation in the supplementary. In practice, the graph representation may influence the result of NAT and how to alleviate its effect still remains an open question.

5 Conclusion

In this paper, we have proposed a novel Neural Architecture Transformer (NAT) for the task of architecture optimization. To solve this problem, we cast it into a Markov decision process (MDP) by making a series of decisions to optimize existing operations with more computationally efficient operations, including skip connection and null operation. To show the effectiveness of NAT, we apply it to both hand-crafted architectures and Neural Architecture Search (NAS) based architectures. Extensive experiments on CIFAR-10 and ImageNet datasets demonstrate the effectiveness of the proposed method in improving the accuracy and the compactness of neural architectures.

Acknowledgments

This work was partially supported by Guangdong Provincial Scientific and Technological Funds under Grants 2018B010107001, National Natural Science Foundation of China (NSFC) (No. 61602185), key project of NSFC (No. 61836003), Fundamental Research Funds for the Central Universities (No. D2191240), Program for Guangdong Introducing Innovative and Enterpreneurial Teams 2017ZT07X183, Tencent AI Lab Rhino-Bird Focused Research Program (No. JR201902), Guangdong Special Branch Plans Young Talent with Scientific and Technological Innovation (No. 2016TQ03X445), Guangzhou Science and Technology Planning Project (No. 201904010197), and Microsoft Research Asia (MSRA Collaborative Research Program). We last thank Tencent AI Lab.

References

[1] B. Baker, O. Gupta, N. Naik, and R. Raskar. Designing neural network architectures using reinforcement learning. In *International Conference on Learning Representations*, 2017.

[2] H. Cai, L. Zhu, and S. Han. ProxylessNAS: Direct neural architecture search on target task and hardware. In *International Conference on Learning Representations*, 2019.

[3] J. Cao, Y. Guo, Q. Wu, C. Shen, J. Huang, and M. Tan. Adversarial learning with local coordinate coding. In *International Conference on Machine Learning*, pages 706–714, 2018.

[4] J. Cao, L. Mo, Y. Zhang, K. Jia, C. Shen, and M. Tan. Multi-marginal wasserstein gan. In *Advances in Neural Information Processing Systems*, 2019.

[5] S. Cao, X. Wang, and K. M. Kitani. Learnable embedding space for efficient neural architecture compression. In *International Conference on Learning Representations*, 2019.

[6] T. Chen, I. Goodfellow, and J. Shlens. Net2net: Accelerating learning via knowledge transfer. In *International Conference on Learning Representations*, 2016.

[7] X. Dai, P. Zhang, B. Wu, H. Yin, F. Sun, Y. Wang, M. Dukhan, Y. Hu, Y. Wu, Y. Jia, et al. Chamnet: Towards efficient network design through platform-aware model adaptation. In *The IEEE Conference on Computer Vision and Pattern Recognition*, pages 11398–11407, 2019.

[8] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. Imagenet: A large-scale hierarchical image database. In *The IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009.

[9] Y. Guo, Q. Chen, J. Chen, J. Huang, Y. Xu, J. Cao, P. Zhao, and M. Tan. Dual reconstruction nets for image super-resolution with gradient sensitive loss. *arXiv preprint arXiv:1809.07099*, 2018.

[10] Y. Guo, Q. Chen, J. Chen, Q. Wu, Q. Shi, and M. Tan. Auto-embedding generative adversarial networks for high resolution image synthesis. *IEEE Transactions on Multimedia*, 2019.

[11] Y. Guo, M. Tan, Q. Wu, J. Chen, A. V. D. Hengel, and Q. Shi. The shallow end: Empowering shallower deep-convolutional networks through auxiliary outputs. *arXiv preprint arXiv:1611.01773*, 2016.

[12] Y. Guo, Q. Wu, C. Deng, J. Chen, and M. Tan. Double forward propagation for memorized batch normalization. In *AAAI Conference on Artificial Intelligence*, pages 3134–3141, 2018.

[13] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *The IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.

[14] K. He, X. Zhang, S. Ren, and J. Sun. Identity mappings in deep residual networks. In *The European Conference on Computer Vision*, pages 630–645, 2016.

[15] Y. He, X. Zhang, and J. Sun. Channel pruning for accelerating very deep neural networks. In *The IEEE International Conference on Computer Vision*, pages 1398–1406, 2017.

[16] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[17] Z. Huang and N. Wang. Data-driven sparse structure selection for deep neural networks. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 304–320, 2018.

[18] Z. Jiang, Y. Zheng, H. Tan, B. Tang, and H. Zhou. Variational deep embedding: An unsupervised and generative approach to clustering. In *International Joint Conference on Artificial Intelligence*, pages 1965–1972, 2017.

[19] H. Jin, Q. Song, and X. Hu. Auto-keras: Efficient neural architecture search with network morphism. *arXiv preprint arXiv:1806.10282*, 2018.

[20] W. Jin, K. Yang, R. Barzilay, and T. Jaakkola. Learning multimodal graph-to-graph translation for molecular optimization. In *International Conference on Learning Representations*, 2019.

[21] T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations*, 2016.

[22] A. Krizhevsky and G. Hinton. Learning multiple layers of features from tiny images. Technical report, Citeseer, 2009.

[23] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 1097–1105, 2012.

[24] S. Lauly, Y. Zheng, A. Allauzen, and H. Larochelle. Document neural autoregressive distribution estimation. *The Journal of Machine Learning Research*, 18(1):4046–4069, 2017.

[25] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation Applied to Handwritten zip Code Recognition. *Neural Computation*, 1(4):541–551, 1989.

[26] C. Lemaire, A. Achkar, and P.-M. Jodoin. Structured pruning of neural networks with budget-aware regularization. In *The IEEE Conference on Computer Vision and Pattern Recognition*, pages 9108–9116, 2019.

[27] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf. Pruning filters for efficient convnets. In *International Conference on Learning Representations*, 2017.

[28] M. Lin, Q. Chen, and S. Yan. Network in network. In *International Conference on Learning Representations*, 2014.

[29] C. Liu, B. Zoph, M. Neumann, J. Shlens, W. Hua, L.-J. Li, L. Fei-Fei, A. Yuille, J. Huang, and K. Murphy. Progressive neural architecture search. In *The European Conference on Computer Vision*, pages 19–34, 2018.

[30] H. Liu, K. Simonyan, and Y. Yang. Darts: Differentiable architecture search. In *International Conference on Learning Representations*, 2019.

[31] J.-H. Luo, J. Wu, and W. Lin. Thinet: A filter level pruning method for deep neural network compression. In *The IEEE International Conference on Computer Vision*, pages 5058–5066, 2017.

[32] R. Luo, F. Tian, T. Qin, E. Chen, and T.-Y. Liu. Neural architecture optimization. In *Advances in Neural Information Processing Systems*, pages 7816–7827, 2018.

[33] V. Nair and G. E. Hinton. Rectified linear units improve restricted boltzmann machines. In *International Conference on Machine Learning*, pages 807–814, 2010.

[34] C. S. Nam, A. Nijholt, and F. Lotte. *Brain–computer interfaces handbook: technological and theoretical advances*. CRC Press, 2018.

[35] J. Pan, Y. Li, and J. Wang. An eeg-based brain-computer interface for emotion recognition. In *2016 international joint conference on neural networks (IJCNN)*, pages 2063–2067. IEEE, 2016.

[36] H. Pham, M. Guan, B. Zoph, Q. Le, and J. Dean. Efficient neural architecture search via parameter sharing. In *International Conference on Machine Learning*, pages 4095–4104, 2018.

[37] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le. Regularized evolution for image classifier architecture search. In *AAAI Conference on Artificial Intelligence*, volume 33, pages 4780–4789, 2019.

[38] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *The IEEE Conference on Computer Vision and Pattern Recognition*, pages 4510–4520, 2018.

[39] F. Schroff, D. Kalenichenko, and J. Philbin. Facenet: A Unified Embedding for Face Recognition and Clustering. In *The IEEE Conference on Computer Vision and Pattern Recognition*, pages 815–823, 2015.

[40] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz. Trust region policy optimization. In *International Conference on Machine Learning*, pages 1889–1897, 2015.

[41] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. In *International Conference on Learning Representations*, 2015.

[42] D. R. So, C. Liang, and Q. V. Le. The evolved transformer. In *International Conference on Machine Learning*, 2019.

[43] R. K. Srivastava, K. Greff, and J. Schmidhuber. Training Very Deep Networks. In *Advances in Neural Information Processing Systems*, pages 2377–2385, 2015.

[44] Y. Sun, X. Wang, and X. Tang. Deeply Learned Face Representations are Sparse, Selective, and Robust. In *The IEEE Conference on Computer Vision and Pattern Recognition*, pages 2892–2900, 2015.

[45] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *The IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–9, 2015.

[46] M. Tan, I. W. Tsang, and L. Wang. Towards ultrahigh dimensional feature selection for big data. *The Journal of Machine Learning Research*, 15(1):1371–1429, 2014.

[47] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, pages 5998–6008, 2017.

[48] L. Wang, Y. Xiong, Z. Wang, Y. Qiao, D. Lin, X. Tang, and L. Van Gool. Temporal segment networks: Towards good practices for deep action recognition. In *The European Conference on Computer Vision*, pages 20–36, 2016.

[49] R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3-4):229–256, 1992.

[50] S. Xie, H. Zheng, C. Liu, and L. Lin. Snas: Stochastic neural architecture search. In *International Conference on Learning Representations*, 2019.

[51] T.-J. Yang, A. Howard, B. Chen, X. Zhang, A. Go, M. Sandler, V. Sze, and H. Adam. Netadapt: Platform-aware neural network adaptation for mobile applications. In *The European Conference on Computer Vision*, pages 285–300, 2018.

[52] R. Zeng, C. Gan, P. Chen, W. Huang, Q. Wu, and M. Tan. Breaking winner-takes-all: Iterative-winners-out networks for weakly supervised temporal action localization. *IEEE Transactions on Image Processing*, 28(12):5797–5808, 2019.

[53] R. Zeng, W. Huang, M. Tan, Y. Rong, P. Zhao, J. Huang, and C. Gan. Graph convolutional networks for temporal action localization. In *The IEEE International Conference on Computer Vision*, Oct 2019.

[54] C. Zhang, M. Ren, and R. Urtasun. Graph hypernetworks for neural architecture search. In *International Conference on Learning Representations*, 2019.

[55] X. Zhang, Z. Huang, and N. Wang. You only search once: Single shot neural architecture search via direct sparse optimization. *arXiv preprint arXiv:1811.01567*, 2018.

[56] Y. Zhang, H. Chen, Y. Wei, P. Zhao, J. Cao, X. Fan, X. Lou, H. Liu, J. Hou, X. Han, et al. From whole slide imaging to microscopy: Deep microscopy adaptation network for histopathology cancer image classification. In *International Conference on Medical Image Computing and Computer-Assisted Intervention*, pages 360–368. Springer, 2019.

[57] Y. Zheng, C. Liu, B. Tang, and H. Zhou. Neural autoregressive collaborative filtering for implicit feedback. In *Proceedings of the 1st Workshop on Deep Learning for Recommender Systems*, pages 2–6. ACM, 2016.

[58] Y. Zheng, B. Tang, W. Ding, and H. Zhou. A neural autoregressive approach to collaborative filtering. In *International Conference on Machine Learning*, pages 764–773, 2016.

[59] Y. Zheng, R. S. Zemel, Y.-J. Zhang, and H. Larochelle. A neural autoregressive approach to attention-based recognition. *International Journal of Computer Vision*, 113(1):67–79, 2015.

[60] Y. Zheng, Y.-J. Zhang, and H. Larochelle. Topic modeling of multimodal data: An autoregressive approach. In *The IEEE Conference on Computer Vision and Pattern Recognition*, pages 1370–1377, 2014.

[61] Y. Zheng, Y.-J. Zhang, and H. Larochelle. A deep and autoregressive approach for topic modeling of multimodal data. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 38(6):1056–1069, 2015.

[62] Z. Zhong, J. Yan, W. Wu, J. Shao, and C.-L. Liu. Practical block-wise neural network architecture generation. In *The IEEE Conference on Computer Vision and Pattern Recognition*, pages 2423–2432, 2018.

[63] Z. Zhuang, M. Tan, B. Zhuang, J. Liu, Y. Guo, Q. Wu, J. Huang, and J. Zhu. Discrimination-aware channel pruning for deep neural networks. In *Advances in Neural Information Processing Systems*, pages 875–886, 2018.

[64] B. Zoph and Q. V. Le. Neural architecture search with reinforcement learning. In *International Conference on Learning Representations*, 2017.

[65] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le. Learning transferable architectures for scalable image recognition. In *The IEEE Conference on Computer Vision and Pattern Recognition*, pages 8697–8710, 2018.