Supplementary Materials for "Pareto-aware Neural Architecture Generation for Diverse Computational Budgets"

Yong Guo, Yaofo Chen, Yin Zheng, Qi Chen, Peilin Zhao, Junzhou Huang, Jian Chen, Mingkui Tan*
South China University of Technology, Tencent AILab, University of Adelaide
{guo.yong, sechenyaofo}@mail.scut.edu.cn, yzheng3xg@gmail.com, qi.chen04@adelaide.edu.au,
{masonzhao, joehhuang}@tencent.com, {ellachen, mingkuitan}@scut.edu.cn

In the supplementary, we provide more discussions, more implementation details, and more experimental results of the proposed Pareto-aware Neural Architecture Generator (PNAG). We organize the supplementary as follows.

- In Section A, we give more details of learning the architecture generator with entropy regularization. Moreover, we also provide the derivations of the gradient *w.r.t.* the objective function $J(\theta)$.
- In Section B, we provide more discussions on the proposed Pareto dominance rule $d\big(\beta_1, \beta_2, B\big)$.
- In Section C, we provide more discussions on the pairwise ranking loss $L(w)$.
- In Section D, we depict more details on the model design of the proposed PNAG, including both the architecture generator $f(B;\theta)$ and the architecture evaluator $R(\cdot|B;w)$.
- In Section E, we provide more implementation details of the proposed PNAG.
- In Section F, we show the convergence curves of the architecture generator and the architecture evaluator.
- In Section G, we show the latency histograms of the architectures generated by PNAG under diverse budgets.
- In Sections H and I, we provide more search results on CPU and GPU devices, respectively.
- In Section J, we visualize all the generated architectures on three hardware devices.

A. Learning Architecture Generator with Entropy Regularization

To solve the problem (2), we use a policy gradient method to learn the architecture generator. To encourage exploration, we introduce an entropy regularization term $H(\cdot)$ to measure the entropy of the policy. Thus, the objective becomes

$$J(\theta) = \mathbb{E}_{B\sim\mathcal{B}}\left[\mathbb{E}_{\alpha_B\sim\pi(\cdot|B;\theta)}\left[R\left(\alpha_B|B;w\right)\right] + \lambda H\big(\pi(\cdot|B;\theta)\big)\right], \tag{A}$$

where λ is a hyper-parameter. In each iteration, we first sample $\{B_k\}_{k=1}^K$ from the distribution $\mathcal{B}$, and then sample N architectures $\{\alpha_{B_k}^{(i)}\}_{i=1}^N$ for each budget B_k. Thus, the gradient of the objective for the generator w.r.t. θ becomes

$$\begin{aligned}\nabla_\theta J(\theta) \approx \frac{1}{KN}\sum_{k=1}^{K}\sum_{i=1}^{N}\Big[&\nabla_\theta \log\pi(\alpha_{B_k}^{(i)}|B_k;\theta)R(\alpha_{B_k}^{(i)}|B_k;w) \\ &+ \lambda\nabla_\theta H(\pi(\cdot|B_k;\theta))\Big].\end{aligned} \tag{B}$$

---

*Corresponding author.

Proof of the Relation in Eqn. (B). The objective function of the architecture generator in PNAG can be formulated as

$$
\begin{aligned}
J(\theta) &= \mathbb{E}_{B\sim\mathcal{B}}\left[\mathbb{E}_{\alpha_{B}\sim\pi(\cdot|B;\theta)}\left[R\left(\alpha_{B}|B;w\right)\right] + \lambda H\big(\pi(\cdot|B;\theta)\big)\right] \\
&= \sum_{B} p(B)\left[\sum_{\alpha_{B}}\pi(\alpha_{B}|B;\theta)R\left(\alpha_{B}|B;w\right) + \lambda H\big(\pi(\cdot|B;\theta)\big)\right].
\end{aligned} \tag{C}
$$

Let $p(B)$ be the probability to sample a specific latency B from the distribution $\mathcal{B}$. The gradient of the objective function w.r.t. θ can be computed by

$$
\begin{aligned}
\nabla_{\theta}J(\theta) &= \sum_{B} p(B)\left[\sum_{\alpha_{B}}\nabla_{\theta}\pi(\alpha_{B}|T;\theta)R\left(\alpha_{B}|B;w\right) + \lambda\nabla_{\theta}H\big(\pi(\cdot|B;\theta)\big)\right] \\
&= \sum_{B} p(B)\left[\sum_{\alpha_{B}}\pi(\alpha_{B}|B;\theta)\nabla_{\theta}\log\pi(\alpha_{B}|B;\theta)R\left(\alpha_{B}|B;w\right) + \lambda\nabla_{\theta}H\big(\pi(\cdot|B;\theta)\big)\right] \\
&= \mathbb{E}_{\alpha_{B}\sim\pi(\cdot|B;\theta),B\sim\mathcal{B}}\left[\nabla_{\theta}\log\pi(\alpha_{B}|B;\theta)R\left(\alpha_{B}|B;w\right) + \lambda\nabla_{\theta}H\big(\pi(\cdot|T;\theta)\big)\right] \\
&\approx \frac{1}{KN}\sum_{k=1}^{K}\sum_{i=1}^{N}\left[\nabla_{\theta}\log\pi(\alpha_{B_k}^{(i)}|B_k;\theta)R(\alpha_{B_k}^{(i)}|B_k;w) + \lambda\nabla_{\theta}H(\pi(\cdot|B_k;\theta))\right].
\end{aligned} \tag{D}
$$

□

B. More Discussions on the Pareto Dominance Rule

In this section, we discuss the proposed Pareto dominance rule, which determines whether an architecture is better than another under diverse budgets. Let B be a computation budget. Given any two architectures β_1, β_2, if both of them satisfy the budget constraints (namely $c(\beta_1) \leq B$ and $c(\beta_2) \leq B$), then β_1 dominates β_2 if $\mathrm{Acc}(\beta_1) \geq \mathrm{Acc}(\beta_2)$. Moreover, when at least one of β_1, β_2 violates the budget constraint, clearly we have that β_1 dominates β_2 if $c(\beta_1) \leq c(\beta_2)$. Formally, we define the Pareto dominance function $d\big(\beta_1, \beta_2, B\big)$ to reflect the above rules:

$$
d\big(\beta_1,\beta_2,B\big) = \begin{cases}
1, & \text{if } \ (c(\beta_1) \leq B \wedge c(\beta_2) \leq B) \\
 & \quad \wedge\, (\mathrm{Acc}(\beta_1) \geq \mathrm{Acc}(\beta_2)); \\
-1, & \text{else if } \ (c(\beta_1) \leq B \wedge c(\beta_2) \leq B) \\
 & \quad \wedge\, (\mathrm{Acc}(\beta_1) < \mathrm{Acc}(\beta_2)); \\
1, & \text{else if } \ c(\beta_1) \leq \ c(\beta_2); \\
-1, & \text{otherwise.}
\end{cases} \tag{E}
$$

Based on Eqn. (E), we have $d(\beta_1, \beta_2, B) = -d(\beta_2, \beta_1, B)$ if $\beta_1 \neq \beta_2$. If $\beta_1 = \beta_2$ and they have exactly the same accuracy and computational cost. We will have $h(\beta_1, \beta_2, B) = h(\beta_2, \beta_1, B) = 1$, which implies that we cannot determine which architecture is better. In this case, these comparisons would inevitably influence the training of the architecture evaluator. To avoid this, we directly omit the architecture pairs with the same architectures in the training.

It is worth noting that the accuracy constraint $\mathrm{Acc}(\beta_1) \geq \mathrm{Acc}(\beta_2)$ plays an important role in the proposed Pareto dominance function $d\big(\beta_1, \beta_2, B\big)$. Without the accuracy constraint, we may easily find the architectures with very low computational cost and poor performance (See results in Table 2 of the main paper).

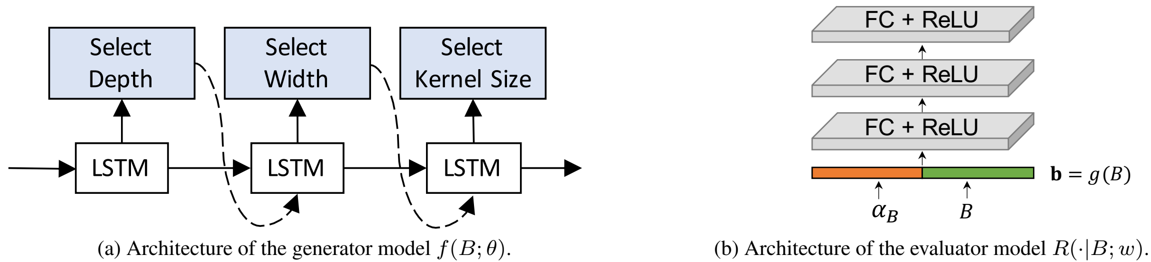
C. More Discussions on the Pairwise Ranking Loss

In this section, we provide more discussions on the pairwise ranking loss to train the architecture evaluator. Based on the Pareto dominance rule, we learn an architecture evaluator to predict the reward $R(\cdot|B;w)$ to guide the search process. To this end, we randomly sample M architectures from the search space Ω and construct $M(M-1)$ architecture pairs after

omitting the pairs with the same architectures. Given K different budgets, we train the architecture evaluator by minimizing the following loss function

$$L(w) = \frac{1}{KM(M-1)} \sum_{k=1}^{K} \sum_{i=1}^{M} \sum_{j=1, j\neq i}^{M} \phi\big(\left(R(\beta_i|B_k;w) - R(\beta_j|B_k;w)\right) \cdot d(\beta_i, \beta_j, B_k)\big), \tag{F}$$

where $\phi(z) = \max(0, 1 - z)$ is the hinge loss function. The goal of minimizing Eqn. (F) is to make the architecture evaluator $R(\beta_i|B_k;w)$ rank different architectures under the budget w.r.t. B_k. To this end, given any two architectures β_i and β_j, we use the hinge loss $\phi(\cdot)$ to force the predicted ranking result $R(\beta_i|B_k;w) - R(\beta_j|B_k;w)$ to be consistent with the ranking result $d(\beta_i, \beta_j, B_k)$ obtained by the Pareto dominance rule. Based on the pretrained evaluator, we are able to evaluate architectures under any given budget.



(a) Architecture of the generator model $f(B;\theta)$.

(b) Architecture of the evaluator model $R(\cdot|B;w)$.

Figure A. Architectures of the generator and the evaluator models in our PNAG. (a) We produce an architecture by sequentially decide the *depth*, *width* and *kernel size* via the generator $f(B;\theta)$ with a LSTM model. (b) The evaluator $R(\cdot|B;w)$, consisted of three fully-connected layers, takes an architecture β and a budget B as inputs and outputs the predicted performance.

D. *Architectures* of the Generator Model $f(B;\theta)$ and the Evaluator Model $R(\cdot|B;w)$

In this section, we depict the detailed model design of the **architecture generator** and the **architecture evaluator**.

***Architecture* of the Generator Model $f(B;\theta)$.** Following [9, 15], we represent an architecture as a sequence of tokens (each token denotes a setting of a specific layer, *e.g.*, width or kernel size). Thus, the architecture generation problem can be formulated as a sequential decision making problem, *i.e.*, sequentially predicting the tokens. To make sequential decisions, we build our generator model $f(B;\theta)$ with a LSTM model (See Fig. A). To be more specific, our PNAG takes a budget B as input and generates architectures by sequentially predicting the token sequences, including depth, width, and kernel size.

***Architecture* of the Evaluator Model $R(\cdot|B;w)$.** To learn PNAG, we need to evaluate architectures under diverse budgets to provide the reward signals. To this end, we build an architecture evaluator model which takes an architecture β and a budget B as inputs and predict the performance $R(\cdot|B;w)$ of β under the budget B. In practice, we build the architecture evaluator $R(\cdot|B;w)$ with a three-layer fully connected network and each of them is followed by a ReLU [8] activation layer. We set the number of intermediate neurons to 512.

E. More Implementation Details

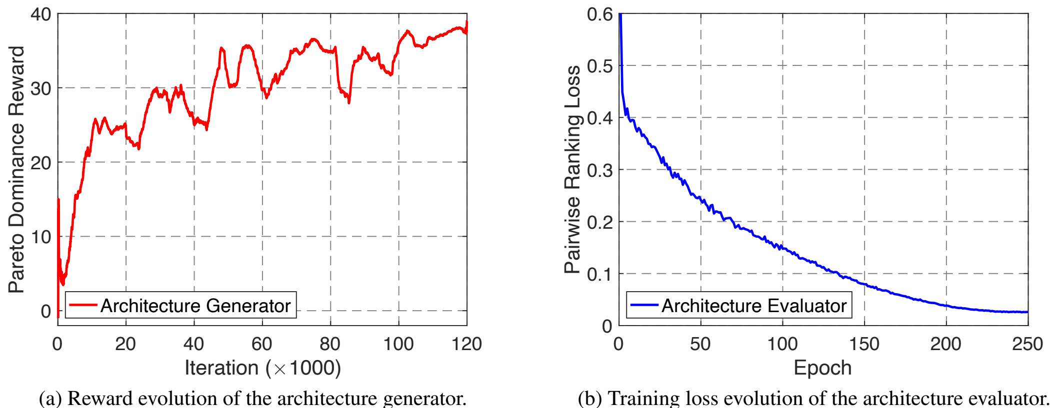
Search space. Following [1], we use MobileNetV3 [4] as the backbone to build the search space [1, 5]. We divide a network into several units. To find promising architectures, we allow each unit to have 1) any numbers of layers (*i.e.*, depth) chosen from $\{2, 3, 4\}$, 2) any width expansion ratios in each layer (*i.e.*, width) chosen from $\{3, 4, 6\}$, and 3) any kernel sizes chosen from $\{3, 5, 7\}$. We build the model with 5 units. Thus, there are 3×3 combinations of widths and kernel sizes for each layer.
Training the supernet. To accelerate the training of supernet, we follow [14] to randomly choose 100 classes from original 1000 classes in ImageNet for training and train the supernet with progressive shrinking strategy [1] for 90 epochs. We treat 80% of these data as the training set to train the supernet and the rest 20% as the validation set to measure the validation accuracy of candidate architectures (we report such validation accuracy in Figs. 1 and 4 of the main paper). We consider the original ImageNet validation set as the test data and report the test accuracy of candidate architectures on them in all the other tables and figures. Based on a NVIDIA V100 GPU, the training process of the supernet takes around *15 GPU hours* (*i.e.*, 0.6 GPU days).
Training architecture evaluator. We collect M=16,000 architectures by uniformly sampling architectures from the search space Ω (See Fig. 5 in the main paper) following [1] and obtain the latency ranges on three hardware devices. We deploy

these architectures to different devices and measure the latency over a batch of images. Specifically, we measure the latency on mobile and CPU devices with a batch size of 1. Since the inference on GPU is too fast to obtain the accurate latency, we measure the latency with a batch size of 64 on NVIDIA TITAN X. We compute the accuracy $\mathrm{Acc}(\cdot)$ on our validation set (*i.e.*, 20% samples of 100 selected classes in ImageNet). We train the architecture evaluator for 250 epochs. The learning rate is initialized to 0.1 and decreased to $1{\times}10^{-3}$ with a cosine annealing. Following [1], we train two predictors to predict the latency and accuracy, respectively. We set the dimension of the budget embedding to 64. We emphasize that training the architecture evaluator is very efficient and only takes less than *0.2 GPU hours*.

Training architecture generator. We train the model for 120k iterations using an Adam optimizer with a learning rate of 3×10^{-4}. Following ENAS [9], we sample $N{=}1$ architecture at each iteration and find it works well in practice. We select $K{=}10$ latency budgets by evenly dividing the range. We add an entropy regularization term to the reward weighted by $1{\times}10^{-3}$. Note that training the architecture generator approximately takes *2 GPU hours*. When evaluating the searched architectures, following [1, 7], we first obtain the parameters from the OFA full network and then finetune them for 75 epochs to obtain the final performance.

F. Convergence Curves of the Architecture Generator and the Architecture Evaluator

In this section, we show the convergence curves of the generator model and the evaluator model in Fig. B. As shown in Fig. Ba, by minimizing the pairwise ranking loss, the architecture evaluator is able to converge very fast. As for the architecture evaluator, based on the proposed Pareto dominance rule, the architecture evaluator would gradually reduce the pairwise ranking loss (See Fig. Bb).



(a) Reward evolution of the architecture generator. (b) Training loss evolution of the architecture evaluator.

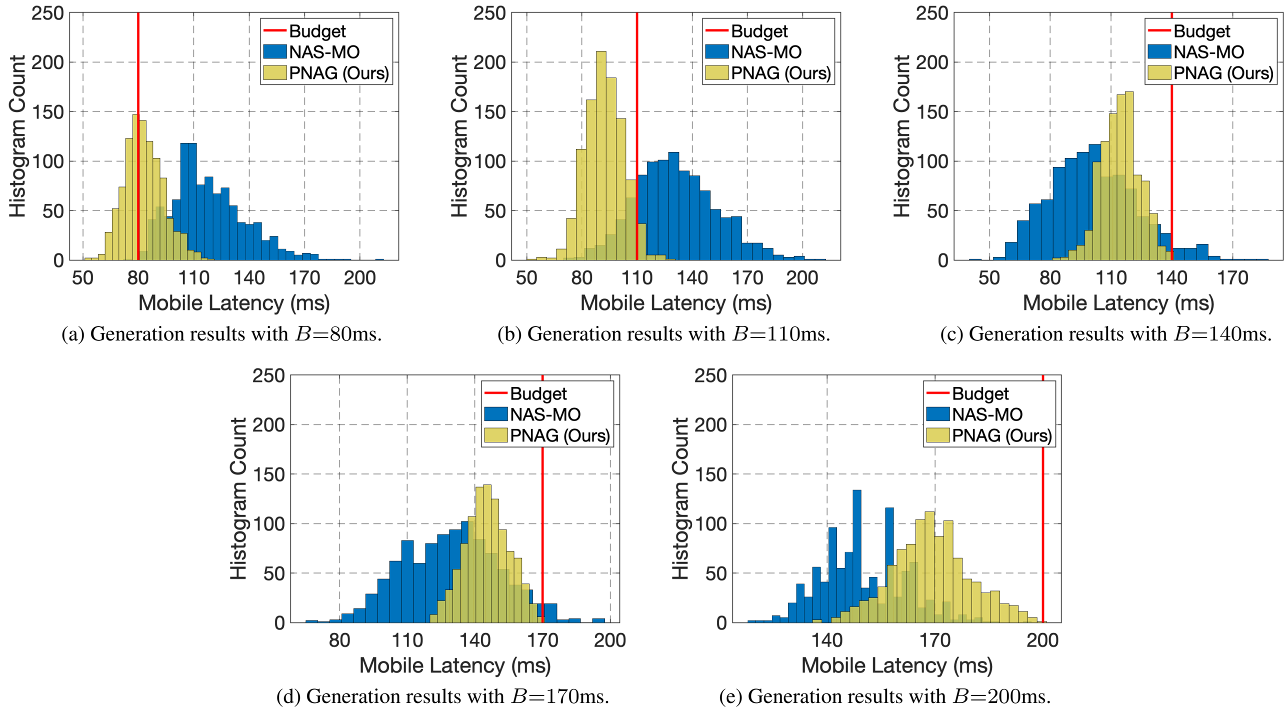
Figure B. Convergence curves of the architecture generator and the architecture evaluator.

G. Latency Histograms of the Generated Architectures under Diverse Budgets

In this section, we show the latency histograms of the generated architectures by OFA-MO [12] and our PNAG under 5 latency budgets. We show the results in Fig. C. From these figures, our PNAG is able to produce the architectures that satisfy the corresponding budget with a higher probability than OFA-MO under different budgets. As mentioned in Section 4.3 in the paper, even if there exist only a few architectures whose latency is lower than 80ms, we still produce a lot of architectures satisfying the budget constraint. These results demonstrate the effectiveness of the proposed method.

H. Architecture Search for CPU Devices

We further exploit our PNAG to generate architectures under the latency budgets evaluated on a CPU device (Core i5-7400). Similar to the experiments for mobile devices, we evaluate our PNAG under 5 latency budgets, *i.e.*, {30ms, 35ms, 40ms, 45ms, 50ms}. As shown in Fig. D, our PNAG yields a large performance improvement over the considered two variants, *i.e.*, EVO and NAS-MO, under diverse budgets. Moreover, our PNAG also outperforms popular NAS based (MnasNet, OFA*) and manually designed architectures (MobileNetV2, MobileNetV3, and EfficientNet). As for the quantitative comparisons, in Table 1, our PNAG consistently yields the best results across all the considered latency budgets. To be specific, given a small latency budget $B{=}35$ms, our PNAG-35 yields better accuracy than the compared NAS methods with much



(a) Generation results with B=80ms.

(b) Generation results with B=110ms.

(c) Generation results with B=140ms.

(d) Generation results with B=170ms.

(e) Generation results with B=200ms.

Figure C. Comparisons of the architecture generation results given different resource budgets on mobile device.

lower search cost. Given a relatively large budget B=50ms, our PNAG-50 yields the same accuracy (80.5%) as the best result on mobile devices (*i.e.*, PNAG-200). This indicates that our PNAG generalizes well across the latency budgets based on different hardwares. Overall, these results demonstrate that our PNAG is able to generate very competitive architectures while satisfying diverse latency budgets.

I. Architecture Search for GPU Devices

Besides the mobile and CPU devices, we also consider GPUs and adopt the latency on them as the computational budget. Since the inference speed on GPU is much faster than mobile processor and CPU, we measure the latency of deep models on a NVIDIA TITAN X GPU with a batch size of 64. In this experiments, we compare different architecture design/search methods under the budgets of {90ms, 115ms, 140ms, 165ms, 190ms}. As shown in Fig. E, similar to the results on mobile and CPU devices, our PNAG outperforms existing methods and the constructed variants by a large margin. We also reported the detailed comparisons in terms of accuracy and computational cost in Table 2. Again, compared with both the hand-crafted methods (*e.g.*, MobileNetV2 [11] and EfficientNet [13]) and NAS methods (*e.g.*, ENAS [9] and MnasNet [12]), our PNAG consistently produces better architectures under diverse budgets. These results further emphasize the generalization ability of our PNAG to the latency budgets evaluated on different hardware devices.

J. Visualization of the Generated Architectures

In this section, we visualize the architectures generated by PNAG under different budgets. We show the generated architectures on mobile phone, CPU, and GPU in Figures F, G, and H, respectively. For convenience, we use "Architecture-T-Hardware" to represent the generated architecture under the budget w.r.t. T on a specific hardware platform, *e.g.*, PNAG-80-Mobile. From these figures, our PNAG tends to produce the architectures with larger depth, width, and kernel size under a larger budget constraint. More critically, from Tables 1, 2, 3, the resultant architectures often have the latencies very close to the considered budget. These results show that our method is able to sufficiently exploit the given resource budget to produce promising architectures.

Table 1. Comparisons with state-of-the-art architectures on Intel Core i5-7400 CPU. * denotes the best architecture reported in the original paper. "-" denotes the results that are not reported. All the models are evaluated on 224×224 images of ImageNet.

Architecture	Latency (ms)	Test Accuracy (%)		#Params (M)	#MAdds (M)	Search Cost (GPU Days)
		Top-1	Top-5			
MobileNetV2 (1.0×) [11]	28.6	72.0	-	3.4	300	-
MobileNetV3-Large (1.0×) [4]	22.6	75.2	-	5.4	219	-
FBNet-C [14]	25.7	74.9	-	5.5	375	9.0
SGNAS-B [6]	-	76.8	-	-	326	0.3
EVO-30	29.1	77.9	93.8	7.9	385	0.7
NAS-MO-30	29.7	77.5	93.7	6.6	353	0.7
PNAG-30 (Ours)	29.7	**78.3**	**94.1**	7.6	335	0.7
ProxylessNAS-CPU [2]	34.6	75.3	-	4.4	438	8.3
MnasNet-A1 (1.4×) [12]	34.6	77.2	93.5	6.1	592	∼3792
EVO-35	34.5	78.5	94.3	8.2	354	0.7
NAS-MO-35	34.7	78.3	94.0	7.9	478	0.7
PNAG-35 (Ours)	34.5	**79.4**	**94.5**	8.4	431	0.7
ResNet-18 [3]	38.6	69.8	90.1	11.7	1814	-
EfficientNet B0 [13]	39.1	77.3	93.5	5.3	390	-
EVO-40	36.3	78.8	94.6	8.4	388	0.7
NAS-MO-40	39.3	78.6	94.3	8.3	491	0.7
PNAG-40 (Ours)	39.6	**79.8**	**94.9**	9.4	502	0.7
MobileNetV2 (1.4×) [11]	42.6	74.7	-	6.9	585	-
EVO-45	43.2	79.1	94.6	9.1	481	51.7
NAS-MO-45	43.7	78.8	94.4	9.3	626	0.7
PNAG-45 (Ours)	44.7	**80.2**	**95.0**	10.4	620	0.7
PONAS-C [5]	52.2	75.2	-	5.6	376	8.8
OFA* [1]	53.7	80.2	95.1	9.1	743	51.7
EVO-50	47.4	79.3	94.7	9.1	511	0.7
NAS-MO-50	46.7	78.9	94.4	9.1	632	0.7
PNAG-50 (Ours)	48.9	**80.5**	**95.1**	10.5	682	0.7

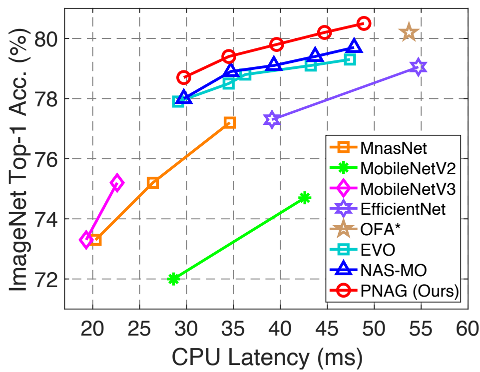


Figure D. Comparisons of the architectures obtained by different methods on a Core i5-7400 CPU.

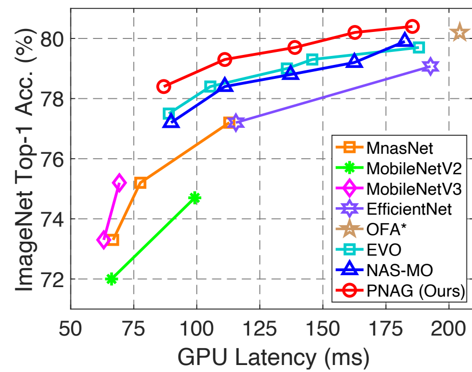
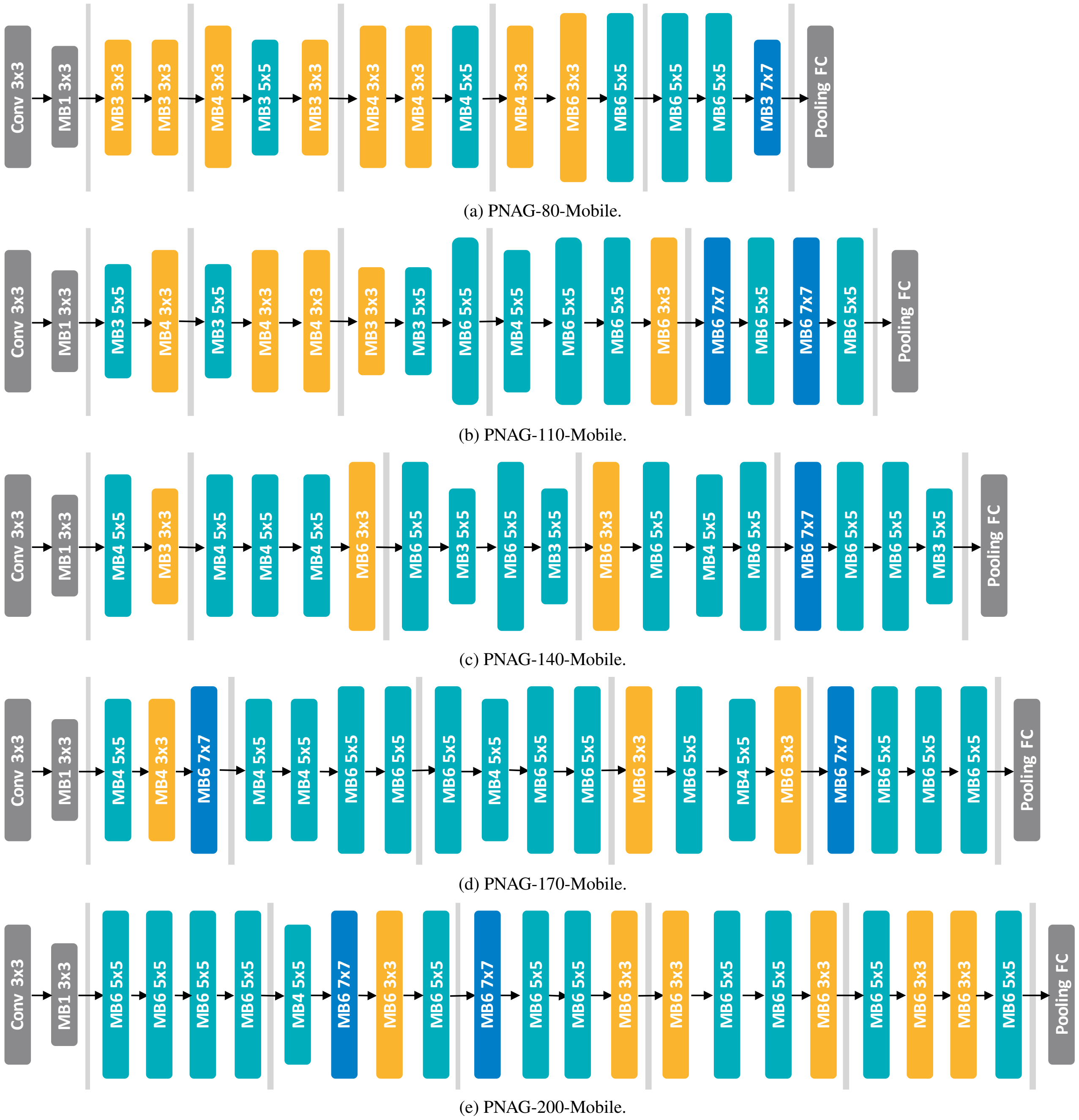


Figure E. Comparisons of the architectures obtained by different methods on a NVIDIA TITAN X GPU.

Table 2. Comparisons with state-of-the-art architectures on NVIDIA TITAN X GPU. * denotes the best architecture reported in the original paper. "-" denotes the results that are not reported. All the models are evaluated on 224×224 images of ImageNet.

Architecture	Latency (ms)	Test Accuracy (%)		#Params (M)	#MAdds (M)	Search Cost (GPU Days)
		Top-1	Top-5			
ProxylessNAS-GPU [2]	84.7	75.1	-	7.1	463	8.3
MobileNetV2 (1.0×) [11]	71.6	72.0	-	3.4	300	-
NAGO [10]	-	76.8	93.4	5.7	-	20.0
EVO-90	88.9	77.3	93.1	5.9	332	0.7
NAS-MO-90	89.8	75.4	92.4	4.9	266	0.7
PNAG-90 (Ours)	86.9	**78.3**	**94.0**	5.7	310	0.7
MnasNet-A1 (1.4×) [12]	112.9	77.2	93.5	6.1	592	∼3792
EfficientNet B0 [13]	115.5	77.3	93.5	5.3	390	-
ENAS [9]	110.8	73.8	91.7	5.6	607	0.5
EVO-115	105.4	78.4	94.1	8.4	388	51.7
NAS-MO-115	111.2	78.1	94.0	8.8	431	0.7
PNAG-115 (Ours)	111.2	**79.3**	**94.6**	8.9	411	0.7
EVO-140	135.7	78.9	94.4	9.1	481	0.7
NAS-MO-140	137.2	78.4	94.1	8.8	470	0.7
PNAG-140 (Ours)	138.9	**79.7**	**94.9**	9.7	510	0.7
ResNet-50 [3]	159.8	76.2	92.9	25.6	4087	-
EVO-165	164.1	79.1	94.5	10.7	597	51.7
NAS-MO-165	162.6	78.8	94.4	10.5	583	0.7
PNAG-165 (Ours)	162.7	**80.3**	**95.0**	10.5	582	0.7
NASNet-A [16]	162.3	74.0	91.6	5.3	564	∼3
PONAS [5]	182.4	75.2	-	5.6	376	8.8
EfficientNet B1 [13]	192.7	79.2	94.5	7.8	700	-
OFA* [1]	204.3	80.2	95.1	9.1	743	51.7
EVO-190	188.1	79.5	94.8	11.3	687	0.7
NAS-MO-190	183.2	78.8	94.5	10.7	652	0.7
PNAG-190 (Ours)	185.5	**80.4**	**95.0**	10.4	640	0.7



(a) PNAG-80-Mobile.

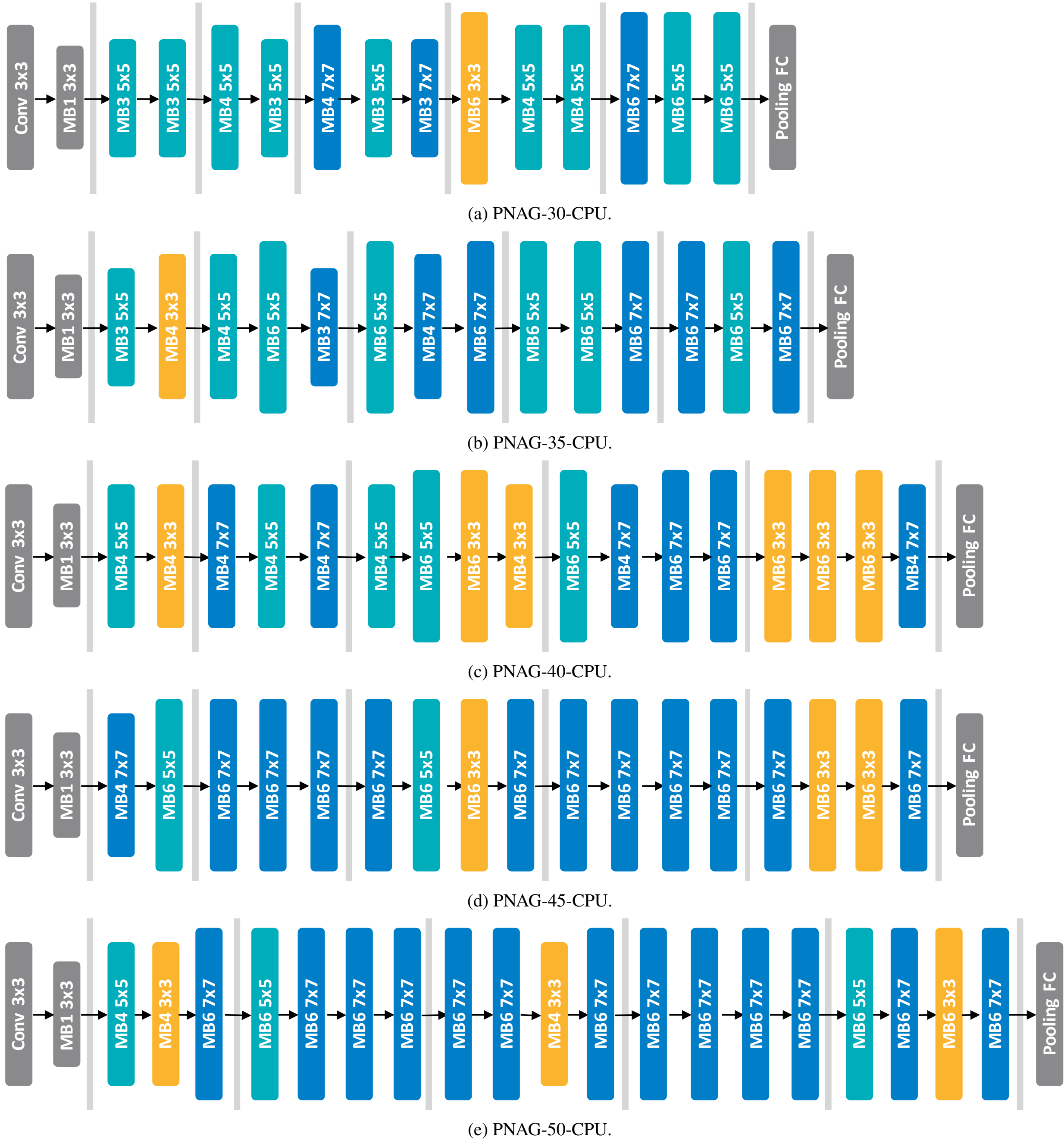
(b) PNAG-110-Mobile.

(c) PNAG-140-Mobile.

(d) PNAG-170-Mobile.

(e) PNAG-200-Mobile.

Figure F. Architectures generated by PNAG on Google Pixel1 phone.



(a) PNAG-30-CPU.

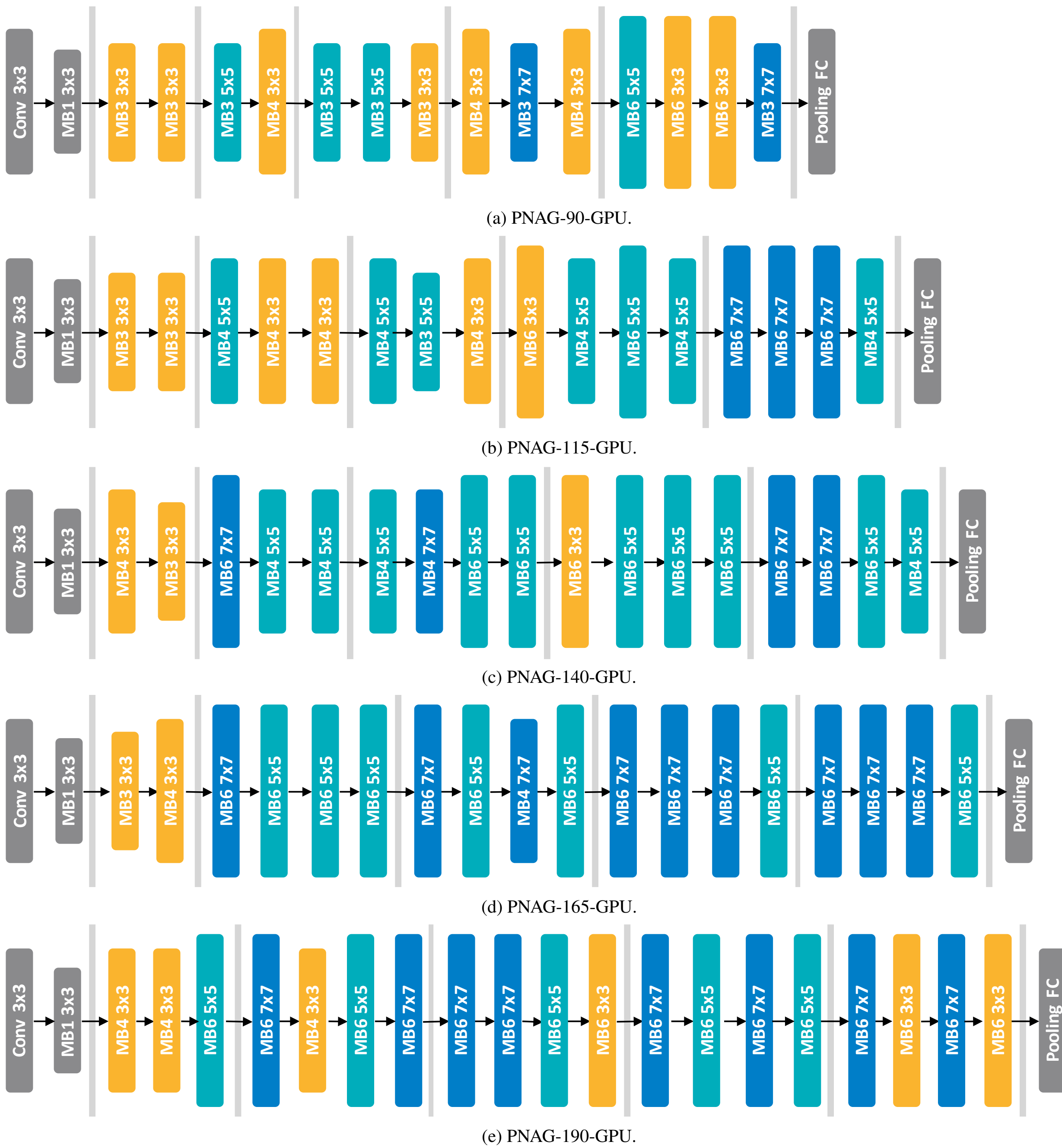
(b) PNAG-35-CPU.

(c) PNAG-40-CPU.

(d) PNAG-45-CPU.

(e) PNAG-50-CPU.

Figure G. Architectures generated by PNAG on Intel Core i5-7400 CPU.



(a) PNAG-90-GPU.

(b) PNAG-115-GPU.

(c) PNAG-140-GPU.

(d) PNAG-165-GPU.

(e) PNAG-190-GPU.

Figure H. Architectures generated by PNAG on NVIDIA TITAN X GPU.

References

[1] Han Cai, Chuang Gan, and Song Han. Once for all: Train one network and specialize it for efficient deployment. In *International Conference on Learning Representations*, 2020. 3, 4, 6, 7

[2] Han Cai, Ligeng Zhu, and Song Han. ProxylessNAS: Direct neural architecture search on target task and hardware. In *International Conference on Learning Representations*, 2019. 6, 7

[3] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016. 6, 7

[4] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, et al. Searching for mobilenetv3. In *IEEE International Conference on Computer Vision*, pages 1314–1324, 2019. 3, 6

[5] Sian-Yao Huang and Wei-Ta Chu. Ponas: Progressive one-shot neural architecture search for very efficient deployment. *arXiv preprint arXiv:2003.05112*, 2020. 3, 6, 7

[6] Sian-Yao Huang and Wei-Ta Chu. Searching by generating: Flexible and efficient one-shot nas with architecture generator. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 983–992, 2021. 6

[7] Zhichao Lu, Gautam Sreekumar, Erik Goodman, Wolfgang Banzhaf, Kalyanmoy Deb, and Vishnu Naresh Boddeti. Neural architecture transfer. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 43(9):2971–2989, 2021. 4

[8] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *International Conference on Machine Learning*, pages 807–814, 2010. 3

[9] Hieu Pham, Melody Guan, Barret Zoph, Quoc Le, and Jeff Dean. Efficient neural architecture search via parameter sharing. In *International Conference on Machine Learning*, pages 4095–4104, 2018. 3, 4, 5, 7

[10] Binxin Ru, Pedro Esperanca, and Fabio Carlucci. Neural architecture generator optimization. In *Advances in Neural Information Processing Systems*, 2020. 7

[11] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 4510–4520, 2018. 5, 6, 7

[12] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V Le. Mnasnet: Platform-aware neural architecture search for mobile. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 2820–2828, 2019. 4, 5, 6, 7

[13] Mingxing Tan and Quoc V. Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International Conference on Machine Learning*, pages 6105–6114, 2019. 5, 6, 7

[14] Bichen Wu, Xiaoliang Dai, Peizhao Zhang, Yanghan Wang, Fei Sun, Yiming Wu, Yuandong Tian, Peter Vajda, Yangqing Jia, and Kurt Keutzer. Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 10734–10742, 2019. 3, 6

[15] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. In *International Conference on Learning Representations*, 2017. 3

[16] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning transferable architectures for scalable image recognition. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 8697–8710, 2018. 7