A APPENDIX

A.1 CONVERGENCE RESULTS

The CE method aims to choose the sampling probability density $q(.)$, from the parametric families of densities $\{p(.;\theta)\}$ such that the Kullback-Leibler (KL)-divergence between the optimal importance sampling probability density $q^*(.,\alpha;\bar{\lambda})$, given by (2), and $p(.;\theta)$ is minimized. Thus, the *CE-Optimal solution* θ^* is obtained as

$$\begin{aligned}\theta^* &= \arg\min_{\theta} KL\left(q^*(.,\alpha;\bar{\lambda})||p(.;\theta)\right) = \arg\max_{\theta} \mathbb{E}_{\bar{\lambda}}\left[\mathbb{I}_{S(X)\geq\alpha}\ln p(X;\theta)\right] \\ &= \arg\max_{\theta} \mathbb{E}_{\tilde{\theta}}\left[\mathbb{I}_{S(X)\geq\alpha}\ln p(X;\theta)\frac{p(X;\bar{\lambda})}{p(X;\tilde{\theta})}\right], \end{aligned} \tag{3}$$

for any prior parameter $\tilde{\theta} \in \Theta$. Let $x_1,\ldots,x_N$ be i.i.d samples from $p(.;\tilde{\theta})$. Then the empirical estimate of θ^* is given by

$$\hat{\theta^*} = \arg\max_{\theta} \frac{1}{N}\sum_{k=1}^{N}\mathbb{I}_{S(x_k)\geq\alpha}\frac{p(x_k;\bar{\lambda})}{p(x_k;\tilde{\theta})}\ln p(x_k;\theta). \tag{4}$$

In the case of rare events, where $l(\alpha) \leq 10^{-6}$, solving the optimization problems (3) and (4) does not help us estimate the probability of the rare event (see (Homem-de Mello & Rubinstein, 2002)). Algorithm 3 presents the *multi-stage* CE approach proposed in (Homem-de Mello & Rubinstein, 2002) to overcome this problem. In this multi-stage algorithm, an auxiliary threshold sequence γ_t, $t \geq 0$, is introduced and the algorithm iterates between updating γ_t and θ_t. More precisely, initially, choose ρ and $\gamma(\bar{\lambda},\rho)$ so that $\gamma(\bar{\lambda},\rho)$ is the $(1-\rho)$-quantile of $S(X)$ under density $p(.;\bar{\lambda})$. And, in general, let $\gamma(\theta_t,\rho_t)$ be the $(1-\rho_t)$-quantile of $S(X)$ under the probability density $p(X;\theta_t)$. Furthermore, the prior parameter $\tilde{\theta}$ introduced in (3) and (4) is replaced by the optimum parameter obtained in iteration $t-1$, i.e., θ_{t-1}.

Algorithm 3 CE Method for rare-event estimation

1: $t \leftarrow 1$, $\rho_0 \leftarrow \rho$, $\theta_0 \leftarrow \bar{\lambda}$
2: **while** $\gamma(\theta_{t-1},\rho_{t-1}) < \alpha$ **do**
3: $\quad\gamma_{t-1} \leftarrow \min(\alpha,\gamma(\theta_{t-1},\rho_{t-1}))$
4: $\quad\theta_t \in \arg\max_{\theta\in\Theta}\mathbb{E}_{\bar{\lambda}}\left[\mathbb{I}_{S(X)\geq\gamma_{t-1}}\ln p(X;\theta)\right] = \arg\min_{\theta\in\Theta} KL(q^*(.,\gamma_{t-1};\bar{\lambda})||p(.;\theta))$
5: $\quad$Set ρ_t such that $\gamma(\theta_t,\rho_t) \geq \min(\alpha,\gamma(\theta_{t-1},\rho_{t-1})+\delta)$ for some fix $\delta > 0$.
6: $\quad t \leftarrow t+1$.

From (3), we notice that Step 3 in Algorithm 3 is equivalent to minimizing the KL-divergence between the following two densities, $q(x,\gamma_{t-1};\bar{\lambda}) = c^{-1}\mathbb{I}_{S(x)\geq\gamma_{t-1}}p(x;\bar{\lambda})$ and $p(x;\theta)$, where $c = \int \mathbb{I}_{S(x)\geq\gamma_{t-1}}p(x;\bar{\lambda})dx$, $\gamma_{t-1} = \min(\alpha,\gamma(\theta_{t-1},\rho_{t-1}))$. We will show later that one can always choose a small δ in Step 5, which will determine the number of times the while loop is executed. The CE method in Algorithm 3 essentially creates a sequence of probability densities $p(.;\theta_1), p(,;\theta_2),\ldots$ that are steered toward the direction of the theoretically optimal density $q^*(.,\alpha;\bar{\lambda})$ in an iterative manner.

Suppose that Algorithm 3 terminates with an estimate θ_T of θ^*. Letting $x_1,\ldots,x_N$ be independent and identically distributed (i.i.d.) samples from the probability density $p(.,\theta_T)$, the *unbiased empirical estimator* of l can be obtained as

$$\hat{l} = \frac{1}{N}\sum_{k=1}^{N}\mathbb{I}_{S(x_k)\geq\alpha}\frac{p(x_k;\bar{\lambda})}{p(x_k,\theta_k)}. \tag{5}$$

In order to use Algorithm 3 in an optimization framework where the goal is to maximize $S(X)$ and there is no specific level parameter α, one can use any desired stopping criteria to terminate the while loop in Algorithm 3 (see (Botev et al., 2013)). For example, one can stop the algorithm when

no change in the value of $S(X)$ is observed after a certain number of iterations (see (Rubinstein & Kroese, 2013) page 134).

The following theorem which was first proved by Homem-de Mello & Rubinstein (2002) asserts that Algorithm 3 terminates. We provide a proof of Theorem A.1 in the following.

Theorem A.1. *Assume that $l > 0$. Then, Algorithm 3 converges to a CE-optimal solution in a finite number of iterations.*

Proof. Let t be an arbitrary iteration of the algorithm, and let $\rho_\alpha := P(S(X) \geq \alpha; \theta_t)$. It can be shown by induction that $\rho_\alpha > 0$. Note that this is also true if $p(x;\theta) > 0$ for all θ, x. Next, we show that for every $\rho_t \in (0, \rho_\alpha)$, $\gamma(\theta_t, \rho_t) \geq \alpha$. By the definition of γ, we have

$$\begin{aligned} &P(S(X) \geq \gamma(\theta_t, \rho_t); \theta) \geq \rho_t, \\ &P(S(X) \leq \gamma(\theta_t, \rho_t); \theta) \geq 1 - \rho_t > 1 - \rho_\alpha. \end{aligned} \tag{6}$$

Suppose by contradiction, that $\gamma(\theta_t, \rho_t) < \alpha$. Then, we get

$$P(S(X) \leq \gamma(\theta_t, \rho_t); \theta^*) \leq P(S(X) \leq \alpha; \theta_t) = 1 - \rho_\alpha,$$

which contradicts (6). Thus, $\gamma(\theta_t, \rho_t) \geq \alpha$. This implies that Step 5 can always be carried out for any $\delta > 0$. Consequently, Algorithm 3 terminates after T iterations with $T \leq \lceil \alpha/\delta \rceil$. Thus at time T, we have

$$\theta_T \in \arg\max_{\theta \in \Theta} \mathbb{E}_{\bar{\lambda}} \left[\mathbb{I}_{S(X) \geq \alpha} \ln P(X; \theta) \right],$$

which implies that θ_T is a CE-Optimal solution. Therefore, Algorithm 3 converges to a CE-Optimal solution in a finite number of iterations. □

Note that we can re-write the maximization step of the Algorithm 3, as follows

$$\theta_t \in \arg\max_{\theta \in \Theta} \mathbb{E}_{\theta_{t-1}} \left[\mathbb{I}_{S(X) \geq \min(\alpha, \gamma(\theta_{t-1}, \rho_{t-1}))} \frac{p(X; \bar{\lambda})}{p(X; \theta_{t-1})} \ln p(X; \theta) \right]. \tag{7}$$

In practice, we calculate the expectation of line 4 of Algorithm 3 with sample averages. Therefore, letting $x_1, \ldots, x_N$ be a sample from the probability density obtained in iteration $t-1$, i.e., $p(x; \hat{\theta}_{t-1})$, $\hat{\gamma}(X, \rho_{t-1})$ be the the sample $(1-\rho_{t-1})$-quantile of $S(x_1), \ldots, S(x_N)$ and using equation (7) above, we replace line 4 of Algorithm 3 with

$$\hat{\theta}_t \in \arg\max_{\theta \in \Theta} \frac{1}{N} \sum_{i=1}^{N} \mathbb{I}_{S(x_i) \geq \hat{\gamma}_N(X, \rho_{t-1})} \frac{p(x_i; \bar{\lambda})}{p(x_i; \theta_{t-1})} \ln p(x_i; \theta). \tag{8}$$

Calculating the expectation line 4 of Algorithm 3 with sample averages might prevent satisfying the stopping criterion of the algorithm. For example, the samples drawn at two consecutive iterations might be the same. Indeed, the following theorem proved by Homem-de Mello & Rubinstein (2002), shows that replacing line 4 of Algorithm 3 with (8), the stopping criterion is still reachable and the while loop in Algorithm 3 terminates, when N is large enough.

Theorem A.2. *Suppose the assumption in Theorem A.1 hold. Let $\theta \in \Theta$ and $X_1, X_2, \ldots$ be i.i.d samples with probability density $p(x;\theta)$. Then there exists $\rho_\alpha > 0$ and $N_\alpha > 0$ such that for all $\rho \in (0, \rho_\alpha)$ and all $N \geq N_\alpha$, we have $\hat{\gamma}_N(X, \rho) \geq \alpha$ with probability one. Moreover, the probability that $\hat{\gamma}(X, \rho) \geq \alpha$ for a given N goes to one exponentially fast with N.*

Proof. Let the function $g(X, \zeta)$ be as

$$g(X, \zeta) = \begin{cases} (1-\rho)(X - \zeta) & \text{if } \zeta \leq X, \\ \rho(\zeta - X) & \text{if } \zeta \geq X. \end{cases} \tag{9}$$

We claim that ζ^* is the $(1-\rho)$-quantile of a random variable X if and only if $\zeta^* \in \arg\min_\zeta \mathbb{E}[g(X, \zeta)]$. Indeed, noting that the subdifferential of $\mathbb{E}[g(X, \zeta)]$ is

$$\partial_\zeta \mathbb{E}[\rho - P(X \geq \zeta), -(1-\rho) + P(X \leq \zeta)].$$

Then, ζ^* is an optimal solution if and only if $0 \in \partial_\zeta \mathbb{E}\left[g(X,\zeta)\right]$. This implies that ζ^* is an optimal solution if and only if

$$\begin{aligned} &\rho - P(X \geq \zeta^*) \leq 0, \\ &-(1-\rho) + P(X \leq \zeta^*) \geq 0, \end{aligned} \tag{10}$$

which means that ζ^* is a $(1-\rho)$-quantile of X. Let $\hat{\zeta}^*$ be the $(1-\rho)$-quantile of a sample $X_1, \ldots, X_N$. Following the similar argument as above, one can see that $\hat{\zeta}^*$ is the solution to the Sample Average Approximation (SAA) problem $\min_\zeta \frac{1}{N}\sum_{i=1}^N g(X_i,\zeta)$. Furthermore, from the results by Rubinstein & Shapiro (1993) and Shapiro et al. (2014), $|\zeta^* - \hat{\zeta}^*| \to 0$ ans $N \to \infty$. Let $X_1, \ldots, X_N$ be i.i.d samples with probability density $p(x;\theta)$. Let $\hat{\gamma}_N(X,\rho^*)$ be the $(1-\rho^*)$-quantile of $S(X_1), \ldots, S(X_N)$. Given a threshold α, we consider two cases, i.e., $P(S(X) > \alpha) > 0$ and $P(S(X) > \alpha) = 0$. For the case $P(S(X) > \alpha) > 0$, set $\rho_\alpha = P(S(X) > \alpha) > 0$. Then following the same argument as in the proof of Theorem A.1, for every $\rho^* \in (0, \rho_\alpha)$, we get $\gamma\left(\theta,\rho^*\right) > 0$. As mentioned earlier, we have $|\gamma\left(\theta,\rho^*\right) - \hat{\gamma}_N(X,\rho^*)| \to 0$ ans $N \to \infty$. Furthermore, following the results by Kaniovski et al. (1995) and Shapiro et al. (2014), the probability that $\hat{\gamma}_N\left(X,\rho^*\right) > \alpha$ goes to one exponentially fast. Now, we consider the other case, where we assume that α is the maximum value achieved by $S(X)$, i.e., $P(S(X) > \alpha;\theta) = 0$. By the assumption of the theorem, setting $\rho_\alpha := P(S(X) = \alpha;\theta)$, $\rho_\alpha > 0$ and for every $\rho^* \in (0,\rho_\alpha)$, we have $\gamma(\theta,\rho^*) = \alpha$. Now considering the random variable $Y := \alpha \mathbb{I}_{S(X)=\alpha}$, $\gamma(\theta,\rho^*) = \alpha$ is a unique $(1-\rho^*)$-quantile of the random variable Y. Setting $\hat{\gamma}_{N,\alpha} = \alpha \mathbb{I}_{\hat{\gamma}(X,\rho^*)=\alpha}$, and $Y_i = \alpha \mathbb{I}_{S(X_i)=\alpha}$, for $i \in \{1,\ldots,N\}$, $\hat{\gamma}_{N,\alpha}$ is a sample $(1-\rho^*)$-quantile of $Y_1, \ldots, Y_N$. Now noting that the random variable Y has a finite support and using the results by Shapiro & Homem-de Mello (2000), we have $\hat{\gamma}_{N,\alpha} = \gamma(\theta,\rho^*) = \alpha$ with probability one with large enough N. Furthermore, the probability that $\hat{\gamma}_{N,\alpha} = \gamma(\theta,\rho^*) = \alpha$ for a given N goes to one exponentially fast. The fact that $\hat{\gamma}_{N,\alpha} = \alpha$ if and only if $\hat{\gamma}_N(X,\rho^*) = \alpha$ completes the proof. □

Consider the Jensen-Shannon (JS) divergence between $q(x,\gamma_{t-1};\bar{\lambda}) = c^{-1}\mathbb{I}_{S(x)\geq\gamma_{t-1}}p(x;\bar{\lambda})$ and $p(x;\theta)$, where $c = \int \mathbb{I}_{S(x)\geq\gamma_{t-1}}p(x;\bar{\lambda})dx$, $\gamma_{t-1} = \min(\alpha, \gamma(\theta_{t-1},\rho_{t-1}))$, which is

$$\begin{aligned} &JS(q(x,\gamma_{t-1};\bar{\lambda})||p(x;\theta)) = \\ &\frac{1}{2}KL\left(c^{-1}p(x;\bar{\lambda})\mathbb{I}_{S(X)\geq\gamma_{t-1}} \parallel \frac{1}{2}\left(c^{-1}p(x;\bar{\lambda})\mathbb{I}_{S(X)\geq\gamma_{t-1}} + p(x;\theta)\right)\right) + \\ &\frac{1}{2}KL\left(p(x;\theta) \parallel \frac{1}{2}\left(c^{-1}p(x;\bar{\lambda})\mathbb{I}_{S(X)\geq\gamma_{t-1}} + p(x;\theta)\right)\right). \end{aligned} \tag{11}$$

Motivated by the well-known fact that the symmetric JS divergence is more robust than the asymmetric KL divergence (Goodfellow, 2016; Arjovsky & Bottou, 2017), we modify Algorithm 3 by minimizing the JS-divergence in (11) instead of the KL-divergence of (2) and (3) to obtain Algorithm 1, which naturally leads to the proposed GA-NAS scheme. In fact, the differences between GANs (relying on JS-divergence) and Variational Auto Encoders (VAEs, which rely on KL divergence), in terms of comparing objectives, are comprehensively discussed on page 14 of (Goodfellow, 2016) as well as page 2 of (Arjovsky & Bottou, 2017). Let the real and generated data are from the probability distribution $P_{real}(.)$ and $P_{gen}(.)$, respectively. According to the discussion in (Arjovsky & Bottou, 2017), asymmetric KL-divergence fails to work properly in two extreme cases. In the case $P_{real}(x) > P_{gen}(x)$, $P_{real}(x) > 0$, and $P_{gen}(x)$ goes to zero, the generator "does not cover parts of data", and in the case $P_{gen}(x) > P_{real}(x)$, $P_{gen}(x) > 0$, and $P_{real}(x)$ goes to zero, the generator generates "fake looking samples". Therefore, it is a general belief in the literature that the symmetry of JS-divergence with respect to $P_{real}(x)$ and $P_{gen}(x)$ causes GANs to generate samples of better quality than VAEs.

We say θ^* is a *JS-optimal solution*, if

$$\theta^* \in \arg\min_{\theta\in\Theta} JS(q(x,\gamma_{t-1};\bar{\lambda})||p(x;\theta)). \tag{12}$$

Then we have the following corollary.

Corollary A.1. *Assuming the condition in Theorem A.1 holds, Algorithm 1 converges to a JS-optimal solution.*

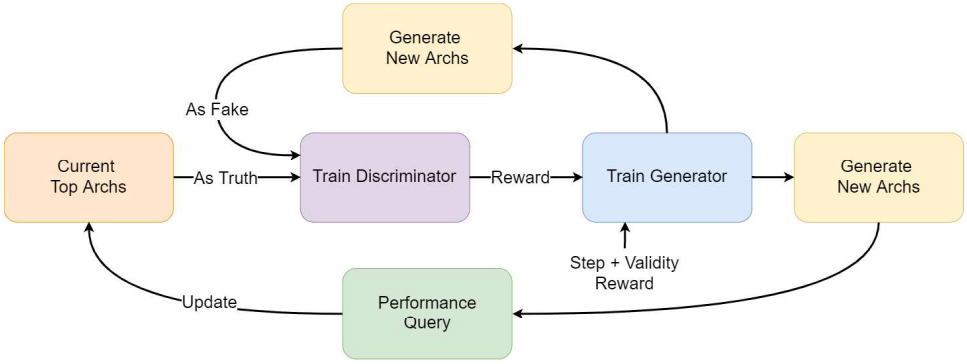


Figure 2: An illustration of the flow of the proposed GA-NAS algorithm.

Proof. First, we assume that we have been able to minimize the JS-divergence between $q(x, \gamma_{t-1}; \bar{\lambda})$ and $p(x; \theta_{t-1})$ exactly by the end of iteration $t-1$. Then, following the same arguments in the proof of Theorem A.1, the assertion of Theorem A.1 still holds. Furthermore, drawing samples from $p(x; \theta)$ as in Theorem A.2, the arguments in the proof of Theorem A.2 still hold and Algorithm 1 converges to a JS-optimal solution. □

A.2 Model Details

A.2.1 GA-NAS Algorithm Flow

Figure 2 provides an illustrative view of the components and steps in the proposed GA-NAS algorithm.

A.2.2 Pairwise Architecture Discriminator

For ease of presentation, we focus on a discriminator for cell-based micro search, i.e., we search for a cell architecture $\mathcal{C}$ where each node is an operator like Conv3x3, Maxpooling, etc, and edges direct the flow of information between nodes, although the model can easily be generalized to macro search. Under this definition, the input to D is a pair of cells $(\mathcal{C}^{true}, \mathcal{C}')$, where $\mathcal{C}^{true}$ denotes a cell from the current truth data, while $\mathcal{C}'$ is a cell from either the truth data or the generator's learned distribution. We transform each cell in the pair into a node embedding vector E using a shared k-GNN model. E has a size of $N \times M$ with N being the number of nodes in the cell and M being the dimension of the node embedding. The GNN encodes each node, corresponding to a row in E, by incorporating features from neighboring nodes. The graph embeddings for $\mathcal{C}^{true}$ and $\mathcal{C}'$ are denoted by the last rows of $E_{\mathcal{C}^{true}}$ and $E_{\mathcal{C}'}$, respectively, as they encompass the features of the whole graph architectures. The graph embedding vectors of the two input cells are then concatenated and passed to an MLP for classification. The output of D is a two-dimensional logits vector describing the probability that $\mathcal{C}'$ is from the truth data distribution (1) or not (0). We train D in a supervised fashion and minimize the cross-entropy loss between the target labels and the predictions.

To create the training data for D, we first sample $\mathcal{T}$ unique cells from the current truth data. We create positive training samples by pairing each truth cell against every other truth cell, hence, creating $|\mathcal{T}|^2$ pairs with a label of 1. We then let the generator generate $|\mathcal{T}|$ unique and valid cells as fake data. We pair each truth cell against all the generated cells and assign them a label of 0. This ensures balanced positive and negative pairs.

We want to emphasize that the same concept for D can be easily transferred to macro search, where now each input cell $\mathcal{C}$ to the discriminator is actually a macro network of blocks, e.g., a single-path network.

A.2.3 Architecture Generator

A complete architecture is constructed in an auto-regressive fashion. We define the state at the t-th time step as $\mathcal{C}_t$, which is a partially constructed graph. Given the state $\mathcal{C}_{t-1}$, the actor inserts a new node. The action a_t is composed of an operator type for this new node selected from a predefined

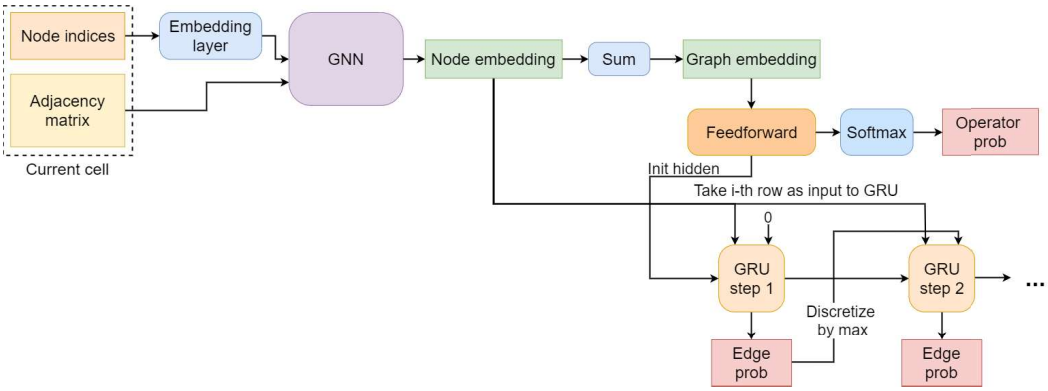


Figure 3: The structure of a GNN architecture generator for cell-based micro search.

operation space, including the *output* node, as well as its connections to other previous nodes. We set the common starting state $\mathcal{C}_0$ as a cell that consists of only the input node(s). We define an episode by a trajectory τ of length N as the state transitions from $\mathcal{C}_0$ to $\mathcal{C}_{N-1}$, i.e., $\tau = \{\mathcal{C}_0, \mathcal{C}_1, \ldots, \mathcal{C}_{N-1}\}$, where N is an upper bound that limits the number of steps allowed in graph construction.

Specifically, at time step t, the actor takes the cell state $\mathcal{C}_{t-1}$ as an input and outputs a probability distribution over all actions using an Encoder-Decoder framework. Similar to the discriminator, the encoder is a multi-layer k-GNN (Morris et al., 2019). The decoder consists of a Feedforward-Softmax setup that outputs the operator probability distribution and a uni-directional Gated Recurrent Unit (Chung et al., 2014) that recursively determines the edge connections to all previous nodes. The actor samples from this distribution to decide a new operator and its connections to all the previously chosen operators and update the state to $\mathcal{C}_t$. The architecture construction terminates when the actor generates a terminal *output* node or the final state $\mathcal{C}_{N-1}$ is reached.

Figure 3 provides an illustration of an architecture generator for cell-based micro search, which is used in our NAS-Bench-101 and 201 experiments. The encoder is a multi-layer k-GNN (Morris et al., 2019) that transforms an input cell $\mathcal{C}$ with N nodes into an embedding vector $E_{\mathcal{C}}^{node}$ of size $N \times M$. Then, a graph embedding $E_{\mathcal{C}}^{graph}$ of size $1 \times M$ is derived for $\mathcal{C}$ by summing the first dimension. In the decoder, a Feedforward + Softmax setup takes in the graph embedding and outputs the probability distribution for the new node. For the edge probabilities, we employ a uni-GRU. Each step examines one of the existing nodes in B, starting from the latest one, and determines the probability of a connection. Aside from the decoder, we also supply $E_{\mathcal{C}}^{graph}$ to another Feedforward layer to implement the value function network $V(\mathcal{C}_t)$, which is required for PPO training. The output of $V(\mathcal{C}_t)$ is a scalar value for the state $\mathcal{C}_t$.

A.2.4 Training Procedure of the Architecture Generator

The state transition is captured by a policy $\pi_\theta(a_t|\mathcal{C}_t)$, where the action a_t includes predictions on a new node type and its connections to previous nodes. To learn $\pi_\theta(a_t|\mathcal{C}_t)$ in this discrete action space, we adopt the Proximal Policy Optimization (PPO) (Schulman et al., 2017) algorithm with generalized advantage estimation. The actor is trained by maximizing the cumulative expected reward of the trajectory. For a trajectory τ with a maximum allowed length of N, this objective translates to

$$\max E[R(\tau)] = \max E[R_{step}(\tau)] + E[R_{final}(\tau)] \quad \text{s.t. } |\tau| \leq N, \tag{13}$$

where R_{step} and R_{final} correspond the per-step reward and the final reward, respectively, which will be decribed in the following.

In the context of cell-based micro search, there is a step reward R_{step}, which is given to the actor after each action, and a final reward R_{final}, which is only assigned at the end of a multi-step generation episode. For R_{step}, we assign the generator a step reward of 0 if a_t is valid. Otherwise, we assign -0.1 and terminate the episode immediately.

R_{final} consists of two parts. The first part is a validity score R_v. For a completed cell $\mathcal{C}_{gen}$, if it is a valid DAG and contains exactly one output node, and there is at least one path from every

other node to the output, then the actor receives a validity score of $R_v(\mathcal{C}_{gen}) = 0$. Otherwise, the validity score will be -0.1 multiplied by the number of validity violations. In our search space, we define four possible violations: 1) There is no output node; 2) There is a node with no incoming edges; 3) There is a node with no outgoing edges; 4) There is a node with no incoming or outgoing edges. The second part of R_{final} is $R_D(\mathcal{C}_{gen})$, which represents the probability that the discriminator classifies $\mathcal{C}_{gen}$ as a cell from the truth data distribution $p_{data}(x)$. In order to receive $R_D(\mathcal{C}_{gen})$ from the discriminator, $\mathcal{C}_{gen}$ must have a validity score of 0 and $\mathcal{C}_{gen}$ cannot be one of the current truth cells $\{\mathcal{C}^j_{true}|j = 1, 2, ...K\}$.

Formally, we express the final reward R_{final} for a generated architecture $\mathcal{C}_{gen}$ as

$$R_{final}(\mathcal{C}_{gen}) = \begin{cases} R_v(\mathcal{C}_{gen}) & \text{if } R_v(\mathcal{C}_{gen}) < 0 \text{ or } \mathcal{C}_{gen} \in \{\mathcal{C}^j_{true}|j = 1, 2, ...K\}, \\ R_v(\mathcal{C}_{gen}) + R_D(\mathcal{C}_{gen}) & \text{otherwise.} \end{cases} \tag{14}$$

We compute $R_D(\mathcal{C}_{gen})$ by conducting pairwise comparisons against the current truth cells, then take the maximum probability that the discriminator will predict a 1, i.e., $R_D = \max_j P(\mathcal{C}_{gen} \in p_{data}(x)|\mathcal{C}_{gen}, \mathcal{C}^j_{true}; D),$ for $j = 1, 2, ...K$, as the discriminator reward.

Maintaining the right balance of exploration/exploitation is crucial for a NAS algorithm. In GA-NAS, the architecture generator and discriminator provide an efficient way to utilize learned knowledge, i.e., exploitation. For exploration, we make sure that the generator always have some uncertainties in its actions by tuning the multiplier for the entropy loss in the PPO learning objective (Schulman et al., 2017). The entropy loss determines the amount of randomness, hence, variations in the generated actions. Increasing its multiplier would increase the impact of the entropy loss term, which results in more exploration. In our experiments, we have tuned this multiplier extensively and found that a value of 0.1 works well for the tested search spaces.

Last but not least, it is worth mentioning that the above formulation of the reward function also works for the single-path macro search scenario, such as EfficientNet and ProxylessNAS search spaces, in which we just need to modify R_{step} and $R_D(\mathcal{C}_{gen})$ according to the definitions of the new search space.

A.3 Experimentation

In this section, we present ablation studies and the results of additional experiments. We also provide more details on our experimental setup. We implement GA-NAS using PyTorch 1.3 (Paszke et al., 2017). We also use the PyTorch Geometric library (Fey & Lenssen, 2019) for the implementation of k-GNN.

A.3.1 Ablation Studies

We report the results of ablation studies to show the effect of different components of GA-NAS on its performance.

Usefulness of the Discriminator: We are interested in how much the discriminator in GA-NAS contributes to the superior search performance. Therefore, we perform an ablation study on NAS-Bench-101 by creating a *RL-NAS* algorithm for comparison. RL-NAS removes the discriminator in GA-NAS and directly queries the accuracy of a generated block from NAS-Bench-101 as the reward for training. We test the performance of RL-NAS under two setups that differ in the total number of queries made to NAS-Bench-101. Table 7 reports the results.

Algorithm	Mean Acc	Mean Rank	Average #Q
RL-NAS-1	94.144 ± 0.100	20.8	7093.2 ± 3903.8
RL-NAS-2	93.781 ± 0.141	919.0	314.0 ± 300.0
GA-NAS-Setup1	94.221± 4.45e-5	2.9	647.5 ± 433.4
GA-NAS-Setup2	94.227± 7.43e-5	2.5	1561.8 ± 802.1

Table 7: Results of ablation study on NAS-Bench-101 by removing the discriminator and directly queries the benchmark for reward.

Compared to GA-NAS-Setup2, RL-NAS-1 makes 3× more queries to NAS-Bench-101 yet it is still unable to outperform both GA-NAS setups. If we instead limits the number of queries as in RL-NAS-2 then the search performance deteriorates significantly. Therefore, we conclude that the discriminator in GA-NAS is crucial for

reducing the number of queries, i.e. number of evaluations in real-world search problems, as well as for finding architectures with better performance.

Standard versus Pairwise Discriminator: In order to investigate the effect of pairwise discriminator in GA-NAS explained in section 3, we perform the same experiments as in Table 2, with the standard discriminator where each architecture $\in \mathcal{T}$ is compared against a generated architecture $\in \mathcal{F}$. The results presented in Table 8 indicate that using pairwise discriminator leads to a better performance compared to using standard discriminator.

Algorithm	Mean Acc	Mean Rank	Average #Q
GA-NAS-Setup1 with pairwise discriminator	**94.221**± 4.45e-5	2.9	**647.5** ± 433.43
GA-NAS-Setup1 without pairwise discriminator	94.22± 0.0	3	771.8 ± 427.51
GA-NAS-Setup2 with pairwise discriminator	**94.227**± 7.43e-5	2.5	1561.8 ± 802.13
GA-NAS-Setup2 without pairwise discriminator	94.22± 0.0	3	897 ± 465.20

Table 8: The mean accuracy and rank, and average number of queries over 10 runs.

Uniform versus linear sample size increase with fixed number of evaluation budget: Once the generator in Algorithm 2 is trained, we sample $|\mathcal{X}_t|$ architectures $\mathcal{X}_t$. Intuitively, as the algorithm progresses, G becomes more and more accurate, thus, increasing the size of $\mathcal{X}_t$ over the iterations should prove advantageous. We provide some evidence that this is the case. More precisely, we perform the same experiments as in Table 2; however, keeping the total number of generated cell architectures during the 10 iterations of the algorithm the same as that in the setup of Table 2, we generate a constant number of cell architectures of 225 and 450 in each iteration of the algorithm in setup1 and setup2, respectively. The results presented in Table 9 indicate that a linear increase in the size of generated cell architectures leads to a better performance.

Algorithm	Mean Acc	Mean Rank	Average #Q
GA-NAS-Setup1 ($\|\mathcal{X}_t\| = \|\mathcal{X}_{t-1}\| + 50,\ \forall t \geq 2$)	**94.221** ± 4.45e-5	**2.9**	**647.5** ± 433.43
GA-NAS-Setup1 ($\|\mathcal{X}_t\| = 225,\ \forall t \geq 2$)	94.21± 0.000262	3.4	987.7 ± 394.79
GA-NAS-Setup2 ($\|\mathcal{X}_t\| = \|\mathcal{X}_{t-1}\| + 50,\ \forall t \geq 2$)	**94.227**± 7.43e-5	**2.5**	1561.8 ± 802.13
GA-NAS-Setup2 ($\|\mathcal{X}_t\| = 450,\ \forall t \geq 2$)	94.22± 0.0	3	1127.6 ± 363.75

Table 9: The mean accuracy and rank, and average number of queries over 10 runs.

A.3.2 Pareto Front Search Results on NAS-Bench-101

In addition to constrained search, we search through Pareto frontiers to further illustrate our algorithm's ability to learn any given truth set. We consider test accuracy vs. normalized training time and found that the truth Pareto front of NAS-Bench-101 contains 41 cells. To reduce variance, we always initialize with the worst 50% of cells in terms of accuracy and training time, which amounts to 82,329 cells. We modify GA-NAS to take a Pareto neighborhood of size 4 in each iteration, which is defined as iteratively removing the cells in the current Pareto front and finding a new front using the remaining cells, until we have collected cells from 4 Pareto fronts. We run GA-NAS for 10 iterations and compare with a random search baseline. GA-NAS issued 2,869 unique queries (#Q) to the benchmark, so we set the #Q for random search to 3,000 for a fair comparison. Figure 4 showcases the effectiveness of GA-NAS in uncovering the Pareto front. While random search also finds a Pareto front that is close to the truth, a small gap is still visible. In comparison, GA-NAS discovers a better Pareto front with a smaller number of queries. GA-NAS finds 10 of the 41 cells on the truth Pareto front, and random search only finds 2.

A.3.3 Hyper-parameter Setup

Table 10 reports the key hyper-parameters used in our NAS-Bench-101 and 201 experiments, which includes (1) best-accuracy search by querying the true accuracy (Acc), (2) best-accuracy search using the supernet (Acc-WS), (3) constrained best-accuracy search (Acc-Cons), and (4) Pareto front search (Pareto).

Here, we include a brief description for some parameters in Table 10.

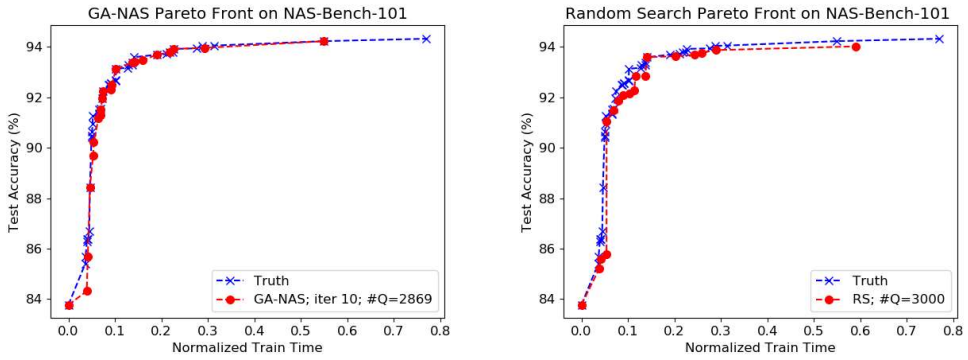


Figure 4: Pareto Front search on NAS-Bench-101. Observe that GA-NAS nearly recovers the truth Pareto Front while Random Search (RS) only discovers one close to the truth. #Q represents the total number of unique queries made to the benchmark. Each marker represents a cell on the Pareto front.

Table 10: Key hyper-parameters used by our NAS-Bench-101 and 201 experiments. Among multiple runs of an experiment, the same hyper-parameters are used and only the random seed differs.

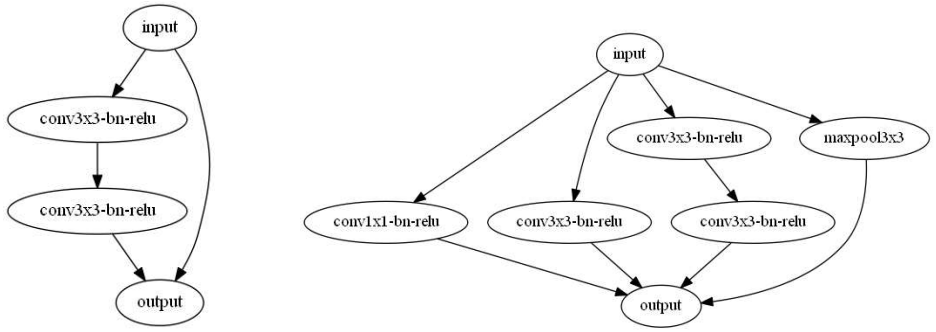
	NAS-Bench-101				**NAS-Bench-201**
Parameter	**Acc (2 setups)**	**Acc-WS**	**Acc-Cons**	**Pareto**	**Acc (3 datasets)**
G optimizer	Adam(Kingma & Ba, 2014)	Adam	Adam	Adam	Adam
G learn rate	0.0001	0.0001	0.0001	0.0001	0.0001
D optimizer	Adam	Adam	Adam	Adam	Adam
D learn rate	0.001	0.001	0.001	0.001	0.001
# GNN layers	2	2	2	2	2
Iterations (T)	10	5	5	10	5
# Eval base ($\|\mathcal{X}_1\|$)	100	100	200	500	60
# Eval inc ($\|\mathcal{X}_t\| - \|\mathcal{X}_{t-1}\|$)	50, 100	100	200	100	10
Init method	Random	Random	Random	Worst 50%	Random
Init size ($\|\mathcal{X}_0\|$)	50, 100	100	100	82329	60
Truth set size ($\|\mathcal{T}\|$)	25, 50	50	50	4 fronts	40

- *# Eval base ($|\mathcal{X}_1|$)* is the number of unique, valid cells to be generated by G after the first iteration of D and G training, i.e., last step of an iteration of Algorithm. 2.
- *# Eval inc ($|\mathcal{X}_t| - |\mathcal{X}_{t-1}|$)* is the increment to the number of generated cells after completing an iteration. From the CE method and importance sampling described in Sec. 3.1, the early generator distribution is not close enough to the well-performing cell architecture. To lower the number of queries to the benchmark without sacrificing the performance, we propose an incremental increase in the number of generated cell architectures at each iteration.
- *Init method* describes how to choose the initial set of cells, from which we create the initial truth set. For most experiments, we randomly choose $|\mathcal{X}_0|$ number of cells as the initial set. For the Pareto front search, we initialize with cells that rank in the lower 50% in terms of both test accuracy and training time (which constitutes 82,329 cells in NAS-Bench-101).
- *Truth set size ($\mathcal{T}$)* controls the number of truth cells for training D. For best-accuracy searches, we take the top most accurate cells found so far. For Pareto front searches, we iteratively collect the cells on the current Pareto front, then remove them from the current pool, then find a new Pareto front, until we visit the desired number of Pareto fronts.

For the complete set of hyper-parameters, please check out our code.

A.3.4 SUPERNET TRAINING AND USAGE

To train a supernet for NAS-Bench-101, we first set up a macro-network in the same way as the evaluation network of NAS-Bench-101. Our supernet has 3 stacks with 3 supercells per stack. A downsampling layer consisting of a Maxpooling 2-by-2 and a Convolution 1-by-1 is inserted after



(a) ResNet cell (b) Inception cell

Figure 5: Structures of the ResNet and Inception cells, which are considered hand-crafted architectures in our constrained best-accuracy search.

the first and second stack to halve the input width/height and double the channels. Each supercell contains 5 searchable nodes. In a NAS-Bench-101 cell, the output node performs concatenation of the input features. However, this is not easy to handle in a supercell. Therefore, we replace the concatenation with summation and do not split channels between different nodes in a cell.

We train the supernet on 40,000 randomly sampled CIFAR-10 training data and leave the other 10,000 as validation data. We set an initial channel size of 64 and a training batch size of 128. We adopt a uniformly random strategy for training, i.e., for every batch of training data, we first uniformly sample an edge topology, then, we uniformly sample one operator type for every searchable node in the topology. Following this strategy, we train for 500 epochs using the Adam optimizer and an initial learning rate of 0.001. We change the learning rate to 0.0001 when the training accuracy do not improve for 50 consecutive epochs.

During search, after we acquire the set of cells ($\mathcal{X}$) to evaluate, we first fine-tune the current supernet for another 50 epochs. The main difference here compared to training a supernet from scratch is that for a batch of training data, we randomly sample a cell from $\mathcal{X}$ instead of from the complete search space. We use the Adam optimizer with a learning rate of 0.0001. Then, we get the accuracy of every cell in $\mathcal{X}$ by testing on the validation data and by inheriting the corresponding weights of the supernet.

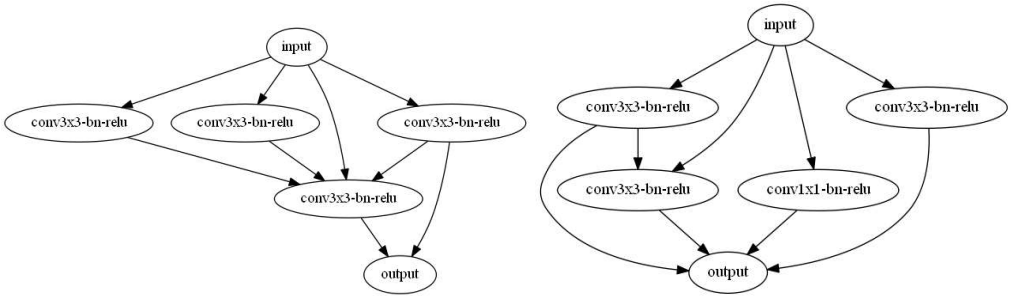
A.3.5 ResNet and Inception Cells in NAS-Bench-101

Figure 5 illustrates the structures of the ResNet and Inception cells used in our constrained best-acc search. Note that both cells are taken from the NAS-Bench-101 database as is, there might be small differences in their structures compared to the original definitions. Table 5 reports that the ResNet cell has a lot more weights than the inception cell, even though it contains fewer operators. This is because NAS-Bench-101 performs channel splitting so that each operator in a branched path will have a much fewer number of trainable weights.

We then present the two best cell structures found by GA-NAS that are better than the ResNet and Inception cells, respectively, in Figure 6. Both cells are also in the NAS-Bench-101 database. Observe that both cells contain multiple branches coming from the input node.

A.3.6 More on NAS-Bench-101 and NAS-Bench-201

We summarize essential statistical information about NAS-Bench-101 and NAS-Bench-201 in Table 11. The purpose is to establish a common ground for comparisons. Future NAS works who wish to compare to our experimental results directly can check Table 11 to ensure a matching benchmark setting. There are 3 candidate operators for NAS-Bench-101, (1) A sequence of convolution 3-by-3, batch normalization (BN), and ReLU activation (Conv3$\times$3-BN-ReLU), (2) Conv1$\times$1-BN-ReLU, (3) Maxpool3$\times$3. NAS-Bench-201 defines 5 operator choices: (1) zeroize, (2) skip connection, (3) ReLU-Conv1$\times$1-BN, (4) ReLU-Conv3$\times$3-BN, (5) Averagepool3$\times$3. We want the re-emphasize that for each cell in NAS-Bench-101, we take the average of the final test accuracy at epoch 108 over



(a) ResNet cell replacement. (b) Inception cell replacement.

Figure 6: The structures of the best two cells found by GA-NAS that are better than the ResNet and Inception cells in terms of test accuracy, training time and number of weights.

Table 11: Key information about the NAS-Bench-101 and NAS-Bench-201 benchmarks used in our experiments.

	NAS-Bench-101	**NAS-Bench-201**		
Dataset	CIFAR-10	CIFAR-10	CIFAR-100	ImageNet-16-120
# Cells	423,624	15,625	15,625	15,625
Highest test acc	94.32	94.37	73.51	47.31
Lowest test acc	9.98	10.0	1.0	0.83
Mean test acc	89.68	87.06	61.39	33.57
# Operator choices	3	5	5	5
# Searchable nodes per cell	5	6	6	6
Max # edges per cell	9	-	-	-

3 runs as its true test accuracy. For NAS-Bench-201, we take the labeled accuracy on the test sets. Since there is a single edge topology for all cells in NAS-Bench-201, there is no need to predict the edge connections; hence, we remove the GRU in the decoder and only predict the node types of 6 searchable nodes.

A.3.7 Clarification on Cell Ranking

We would like to clarify that all ranking results reported in the paper are based on the *un-rounded* true accuracy. In addition, if two cells have the same accuracy, we randomly rank one before the other, i.e. no two cells will have the same ranking.

A.3.8 Conversion from DARTS-like Cells to the NAS-Bench-101 Format

DARTS-like cells, where an edge represents a searchable operator, and a node represents a feature map that is the sum of multiple edge operations, can be transformed into the format of NAS-Bench-101 cells, where nodes represent searchable operators and edges determine data flow. For a unique, discrete cell, we assume that each edge in a DARTS-like cell can adopt a single unique operator. We achieve this transformation by first converting each edge in a DARTS-like cell to a NAS-Bench-101 node. Next, we construct the dependency between NAS-Bench-101 nodes from the DARTS nodes, which enables us to complete the edge topology in a new NAS-Bench-101 cell. Figure 7 shows a DARTS-like cell defined by NAS-Bench-201 and the transformed NAS-Bench-101 cell. This transformation is a necessary first step to make GA-NAS compatible with NAS-Bench-201 and NAS-Bench-301. Notice that every DARTS-like cell has the same edge topology in the NAS-Bench-101 format, which alleviates the need for a dedicated edge predictor in the decoder of G.

A.3.9 EfficientNet and ProxylessNAS Search Space

For EfficientNet experiment, we take the EfficientNet-B0 network structure as the backbone and define 7 searchable locations, as indicated by the TBS symbol in Table 12. We run GA-NAS to select a type of mobile inverted bottleneck MBConv (Sandler et al., 2018) block. We search for different

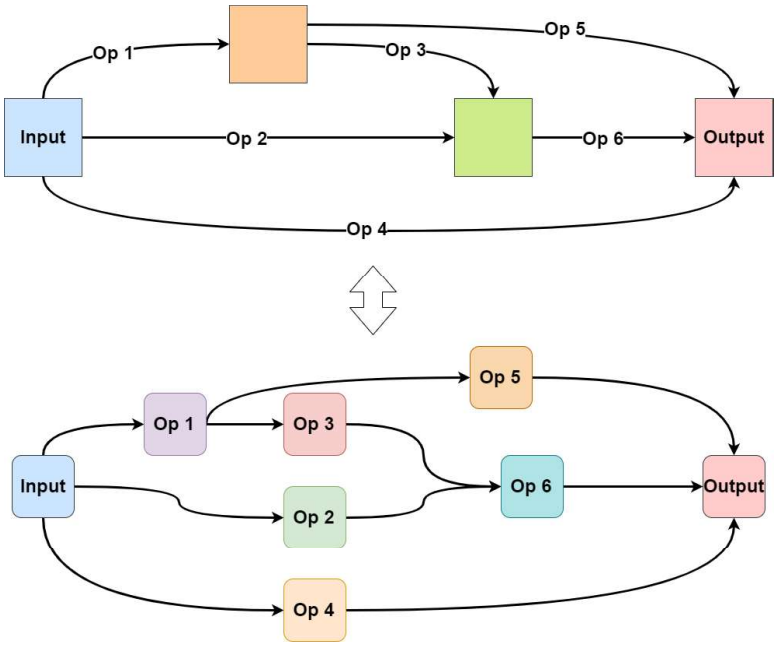


Figure 7: Illustration of how to transform a DARTS-like cell (top) in NAS-Bench-201 to the NAS-Bench-101 format (bottom).

expansion ratios {1, 3 ,6} and kernel size {3, 5} combinations, which results in 6 candidate MBConv blocks per TBS location.

We conduct 2 GA-NAS searches with different performance estimation methods. In setup 1 we estimate the performance of a candidate network by training it on CIFAR-10 for 20 epochs then compute the test accuracy. In setup 2 we first train a weight-sharing Supernet that has the same structure as the backbone in Table 12, for 500 epochs, and using a ImageNet-224-120 dataset that is subsampled the same way as NAS-Bench-201. The estimated performance in this case is the validation accuracy a candidate network could achieve on ImageNet-224-120 by inheriting weights from the Supernet. For setup 2, training the supernet takes about 20 GPU days on Tesla V100 GPUs, and search takes another GPU day, making a total of 21 GPU days.

Figure 8 presents a visualization on the three single-path networks found by GA-NAS on the EfficientNet search space. Compared to EfficientNet-B0, GA-NAS-ENet-1 significantly reduces the number of trainable weights while maintaining an acceptable accuracy. GA-NAS-ENet-2 improves the accuracy while also reducing the model size. GA-NAS-ENet-3 improves the accuracy further.

Operator	**Resolution**	**#C**	**#Layers**
Conv3x3	224×224	32	1
TBS	112×112	16	1
TBS	112×112	24	2
TBS	56×56	40	2
TBS	28×28	80	3
TBS	14×14	112	3
TBS	14×14	192	4
TBS	7×7	320	1
Conv1x1 & Pooling & FC	7×7	1280	1

Table 12: Macro search backbone network for our EfficientNet experiment. TBS denotes a cell position to be searched, which can be a type of MBConv block. Output denotes a Conv1x1 & Pooling & Fully-connected layer.

For ProxylessNAS experiment, we take the ProxylessNAS network structure as the backbone and define 21 searchable locations, as indicated by the TBS symbol in Table 13. We run GA-NAS to search for MBConv blocks with different expansion ratios {3 ,6} and kernel size {3, 5, 7} combinations, which results in 6 candidate MBConv blocks per TBS location.

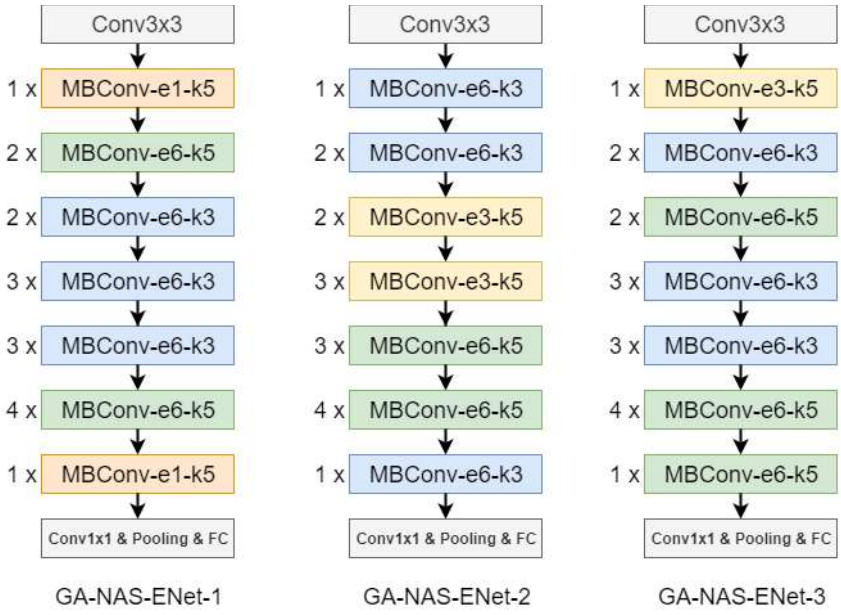


Figure 8: Structures of the single-path networks found by GA-NAS on the EfficientNet search space. In each MBConv block, e denotes expansion ratio and k stands for kernel size.

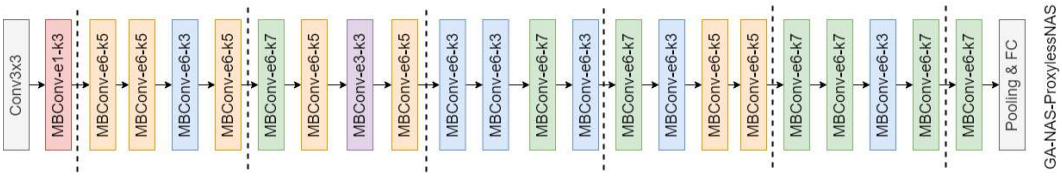


Figure 9: Structures of the single-path networks found by GA-NAS on the ProxylessNAS search space.

We train a supernet on ImageNet (Russakovsky et al., 2015) for 160 epochs (which is the same number of epochs performed by ProxylessNAS weight-sharing search (Cai et al., 2018)) for around 20 GPU days, and conduct an unconstrained search using GA-NAS for around 38 hours on 8 Tesla V100 GPUs in the search space of ProxylessNAS (Cai et al., 2018), a major portion out of which, i.e., 29 hours is spent on querying the supernet for architecture performance. Figure 9 presents a visualization of the best architecture found by GA-NAS-Proxyless with better top-1 accuracy and a comparable the number of trainable weights compared to ProxylessNAS-GPU.

Operator	Resolution	#C	Identity
Conv3x3	224×224	32	No
MBConv-e1-k3	112×112	16	No
TBS	112×112	24	No
TBS	56×56	24	Yes
TBS	56×56	24	Yes
TBS	56×56	24	Yes
TBS	56×56	40	No
TBS	28×28	40	Yes
TBS	28×28	40	Yes
TBS	28×28	40	Yes
TBS	28×28	80	No
TBS	14×14	80	Yes
TBS	14×14	80	Yes
TBS	14×14	80	Yes
TBS	14×14	96	No
TBS	14×14	96	Yes
TBS	14×14	96	Yes
TBS	14×14	96	Yes
TBS	14×14	192	No
TBS	7×7	192	Yes
TBS	7×7	192	Yes
TBS	7×7	192	Yes
TBS	7×7	320	Yes
Avg. Pooling	7×7	1280	1
FC	1×1	1000	1

Table 13: Macro search backbone network for our ProxyLessNAS experiment. TBS denotes a cell position to be searched, which can be a type of MBConv block. Identity denotes if an identity shortcut is enabled.