École Polytechnique

Promotion X2016

HUANG Zhexuan

RAPPORT DE STAGE DE RECHERCHE

Improved Gradient-Based Neural Architecture Search

RAPPORT NON CONFIDENTIEL

Départment de Mathémathiques Appliquées

Champ: MAP 594

Enseignant référent: E. Bacry

Tuteur de stage dans l'organisme: B. Conche

Dates du stage: 25/03/2019 - 16/08/2019

Adresse de l'organisme:

Total, Site Nano-INNOV

91120 Palaiseau

France

*Déclaration

Je soussigné **HUANG Zhexuan** certife sur l'honneur:

1. Que les résultats décrits dans ce rapport sont l'aboutissement de mon travail.
2. Que je suis l'auteur de ce rapport.
3. Que je n'ai pas utilisé des sources ou résultats tiers sans clairement les citer et les référencer selon les règles bibliographiques préconisées.

Mention à recopier

Je déclare que ce travail ne peut être suspecté de plagiat.

Date ..... Signature

Abstract

Neural Architecture Search (NAS) aims to design a neural network architecture for a given task automatically. Most existing approaches in NAS define a search space which contains all candidate neural network architectures. To search for the optimal neural network architecture among all neural networks in search space, different strategies are proposed in recent approaches. Among all searching strategies in NAS, gradient-based search strategies which estimate gradient of the expectation of loss function prove to be efficient and provide fast and economical solutions. Despite its low-computational-consuming characteristic, architectures found using gradient-based strategies also achieve the validation accuracy in state-of-the-art level. For example, most of the gradient-based NAS strategies can achieve $2\% \sim 3\%$ error rates in the dataset of CIFAR-10. However, estimating gradient of the expactation of loss function is a challenging problem, as it requires to make evaluation of loss function over several Monte-Carlo samples, which needs to go through the entire forward pass and is computational expensive. To estimate the gradient, some of the existing strategies use biased estimators while others use unbiased estimators but with high-variance. In this work, we propose several architecture search strategies using unbiased and low-variance gradient estimators.We explore and compare different gradient-based strategies using different gradient estimators.

Contents

1 Introduction

Deep neural networks have been much success in many challenges such as image classification, speech recognition and machine translation in the last few years. Though designing a state-of-the-art neural network architecture usually requires trying different combinations over layers and different connected patterns, thus needs enormous effort of human experts. Recently, several automatically Neural Architecture Search (NAS) strategies have been proposed and achieved state-of-the-art performance in image classification tasks. However, many of these strategies consume too much computational time. For example, [Zoph and Le, 2016] uses reinforcement learning (RL) and 1800 GPU days to get 2.65% of error rates over CIFAR-10. Meanwhile, [Kandasamy et al., 2019] use evolution approach and 3150 GPU days to get a similar result. Based on the ideal of NAS [Zoph and Le, 2016], several searching strategies have been proposed to accelerate the searching process, especially by sharing parameters across multiple architecture ENAS[Pham et al., 2018], SMOB [Jin and Hu, 2019] or Bayesian optimization [Real et al., 2019]. However, these approaches are still inefficient since they formulate NAS problem into a black-box optimization problem over a discrete search space, which requires a great amount of architecture evaluations (i.e. training architectures to converge) and take long time. Instead of black-box optimization strategies, DARTS [Liu et al., 2019] reformulate the problem by defining the search space as an over-parameterized parent network containing all candidate paths with importance weights. Following this setting, neural architecture search problem becomes a problem to optimize importance weight of candidate path in parent network. To perform gradient-descent algorithm for parameters which represent importance of candidate paths in parent network, DARTS relaxes the discrete search space to a continuous one and performs gradient-based optimization over continuous search space. One the other hand, DARTS also reduce the computational cost of NAS to just a few GPU days. However, DARTS introduces bias to loss function according to [Xie et al., 2019]. In order to correct bias in loss function and keep the same differentiable structure of search space as in DARTS, SNAS [Xie et al., 2019] reformulate NAS with a new stochastic modeling and relaxes the search space with *concrete distribution* [Maddison et al., 2017]. Nevertheless, bias is still introduced in SNAS due to continuous relaxation of *concrete distribution*.

In this work, we use the same stochastic modeling as in SNAS and we propose several architecture search strategies using unbiased and low-variance gradient estimators. To estimate the loss function and the gradient of the loss function unbiasly, We first review several techniques to estimate gradients in stochastic computation graph. And then we propose searching strategies using unbiased estimators which correct the bias issue in SNAS. Finally we compare results obtained using different gradient estimators with other state-of-the-art networks. The best architecture found by our methods achieve 2.64% error rates on test set over CIFAR-10, outperform other similar methods like 2.89% from ENAS, 2.83% from DARTS and 2.85% from SNAS.

2 Search Space and Reformulate of problem

We use the same search space as in SNAS. Following [Zoph and Le, 2016, Liu et al., 2018, Pham et al., 2018, Liu et al., 2019, Xie et al., 2019], to reduce computational costs, instead of searching the entire architecture structure, we search for optimal computation cells and build our final architecture by stacking several computation cells together. Two kinds of computation cells are considered: the normal cell and the reduction cell, where the normal cell is obtained by taking all convolution operations with stride $= 1$ and the reduction cell is obtained by taking all convolution operations adjacent to input nodes of cell with stride $= 2$.

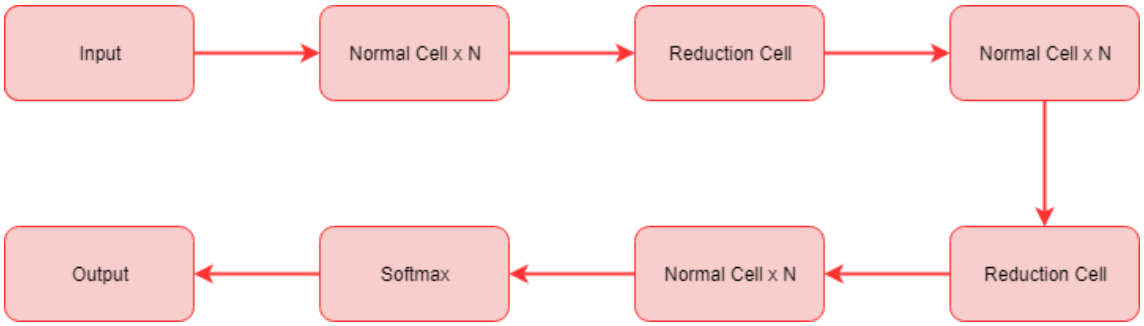


Figure 2.1: Global network consists of stacking normal cell and reduction cell together

The search space (i.e. in our case, computation cells) is represented by a directed acyclic graph (DAG) which contains N nodes, where N is given in priority. Each node x_i in this DAG are feature map and each edge (i, j) is associated with some operation $\tilde{O}_{i,j}$ which transforms x_i into $\tilde{O}_{i,j}(x_i)$. Each intermediate node is computed during forward pass as following:

$$x_j = \sum_{i<j} \tilde{O}_{i,j}(x_i). \tag{2.1}$$

where $\tilde{O}_{i,j}$ represent operation selected for edge (i, j). Thus each intermediate node is calculated by its previous nodes and the operations selected. Similar to ENAS, DARTS and SNAS, zero operation is introduced as one of the possible operations for $\tilde{O}_{i,j}$, inputs of each cell are outputs of previous two cells and the output of each cell is obtained by a concatenation of all intermediate nodes. Here we fixe $N = 7$ (thus 2 input nodes, 4 intermediates nodes and 1 output node in each cell) and we search for two cells, a normal cell and a reduce cell simultaneously.

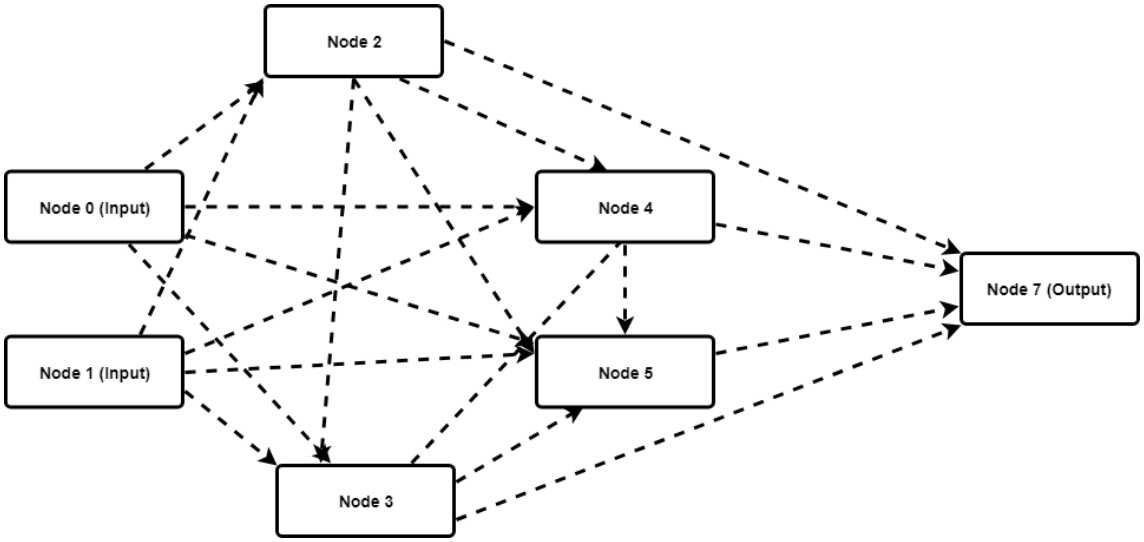


Figure 2.2: A conceptual of a normal cell as well as reduction cell containing 7 nodes.

As described in SNAS, one can reformulate the problem with a stochastic modeling. Let $O_{i,j} = [O^1_{i,j}, \ldots, O^A_{i,j}]^T$ be the vector of candidate operations for each edge (i, j) (e.g. convolution, skip-connection, zero, etc). Each operation is selected from a discrete distribution $\mathbb{P}_\alpha$, where $\alpha = (\alpha^k_{i,j})_{i<j,k=1,\ldots,A}$ and

$$\tilde{O}_{i,j} = \begin{cases} O^1_{i,j} & \textbf{with probability } p^1_{i,j} = \frac{\exp(\alpha^1_{i,j})}{\sum_{k=1}^A \exp(\alpha^k_{i,j})} \\ \ldots & \\ O^A_{i,j} & \textbf{with probability } p^A_{i,j} = \frac{\exp(\alpha^A_{i,j})}{\sum_{k=1}^A \exp(\alpha^k_{i,j})}, \end{cases}$$

Thus $\tilde{O}_{i,j}$ can be represented as $\tilde{O}_{i,j} = Z_{i,j} O_{i,j}$ where

$$Z_{i,j} = \begin{cases} [1, 0, \ldots, 0] & \textbf{with probability } p^1_{i,j} = \frac{\exp(\alpha^1_{i,j})}{\sum_{k=1}^A \exp(\alpha^k_{i,j})} \\ \ldots & \\ [0, 0, \ldots, 1] & \textbf{with probability } p^A_{i,j} = \frac{\exp(\alpha^A_{i,j})}{\sum_{k=1}^A \exp(\alpha^k_{i,j})}, \end{cases}$$

$Z = (Z_{i,j})_{i<j}$ are some independent one-hot random variables whose joint distribution is denoted here by $p_\alpha(Z)$. In the following, we denote $Z_{i,j} = [Z^1_{i,j}, \ldots, Z^A_{i,j}]^T$. With this reformulation, each intermediate node x_j becomes now a random variable and it is calculated by:

$$x_j = \sum_{i<j} \tilde{O}_{i,j}(x_i) = \sum_{i<j} Z_{i,j} O_{i,j}(x_i). \tag{2.2}$$

Following the setting of SNAS, our objective is to minimize the expected loss of architectures sampled with distribution $p_\alpha(Z)$:

$$\min_{\alpha,\theta} \mathbb{E}_{Z\sim p_\alpha(Z)}[L_\theta(Z)], \tag{2.3}$$

where θ represent parameters in all candidate operations. Generally, the expected loss can not be calculated due to large quantities of candidate architectures. However it can be estimated using Monte-Carlo sampling.

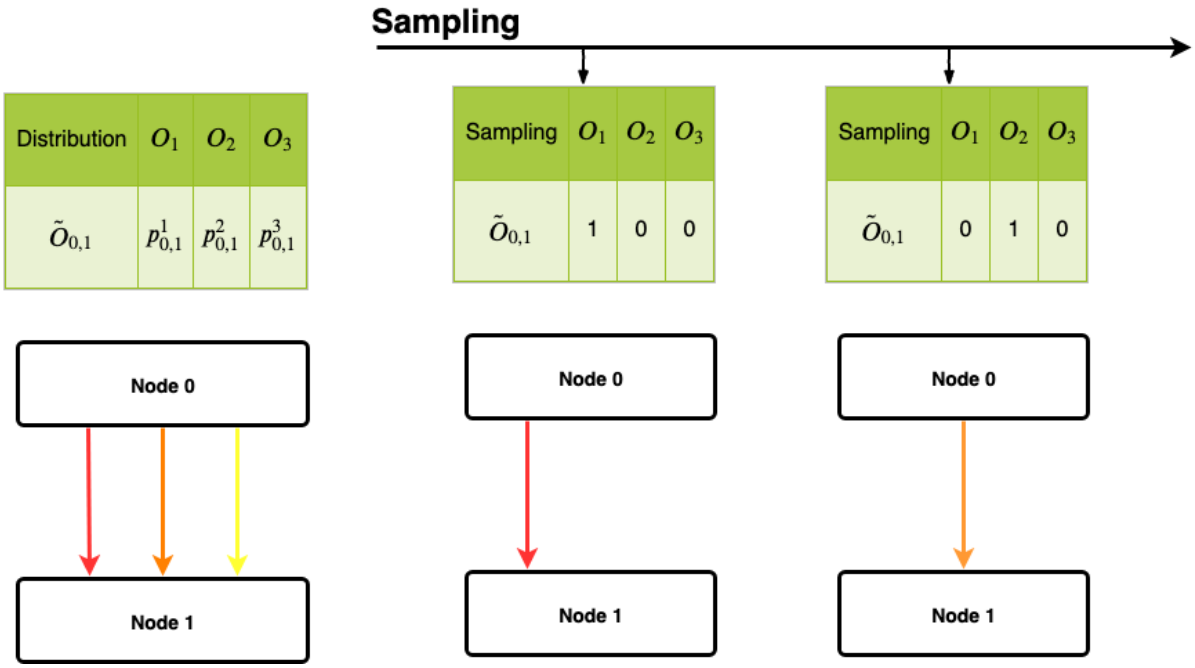


Figure 2.3: A conceptual for a forward pass with 2-nodes-DAG and 3 candidate operations.

3 Gradient estimators

The objective of (2.3) can be optimized with gradient-descent algorithm in which the gradients of $\mathbb{E}_{Z\sim p_\alpha(Z)}[L_\theta(Z)]$ w.r.t α and θ should be calculated or estimated. Calculation of exact gradients is generally unfeasible as it requires evaluations over all possible architectures. Thus the remaining challenge is to estimate gradients more efficiently with lower variances.

Estimating gradient w.r.t. θ is simple since the distribution of Z does not depend on θ and L_θ is differentiable w.r.t θ that can be represented as:

$$\nabla_\theta \mathbb{E}_{Z\sim p_\alpha(Z)}[L_\theta(Z)] = \mathbb{E}_{Z\sim p_\alpha(Z)}[\nabla_\theta L_\theta(Z)]. \tag{3.1}$$

Thus one can sample several $Z \sim p_\alpha(Z)$ and calculate mean value of gradients over these samples.

Unlike estimation of gradient w.r.t. θ, finding a good estimation of gradient w.r.t. α is difficult as Z is a discrete random variable whose distribution depends on α. In the following subsection we review several gradient estimation techniques for stochastic graph.

3.1 Score Function Estimators

The most common estimator of $\nabla_\alpha \mathbb{E}_{Z\sim p_\alpha(Z)}[L_\theta(Z)]$ is the score function (SF) estimator, it is also known as REINFORCE [J.Williams, 1992]. The score function estimator is based on the identity:

$$\nabla_\alpha \mathbb{E}_{Z\sim p_\alpha(Z)}[L_\theta(Z)] = \mathbb{E}_{Z\sim p_\alpha(Z)}[L_\theta(Z)\nabla_\alpha \log p_\alpha(Z)]. \tag{3.2}$$

Estimation is done by using naive Monte Carlo method:

$$\tilde{\nabla}_\alpha \mathbb{E}_{Z\sim p_\alpha(Z)}[L_\theta(Z)] = \frac{1}{S}\sum_{i=1}^{S} L_\theta(Z_i)\nabla_\alpha \log p_\alpha(Z_i), \tag{3.3}$$

where $Z_i \sim p_\alpha(Z)$ i.i.d.

Score function estimator is an unbiased estimator which does not require to calculate back-propagation of $L_\theta(Z)$ and thus can be applied directly to stochastic graph. However, SF estimator suffers from extremely high variance as it depends only on the final result $L_\theta(Z_i)$, making gradient descent algorithm very slow to converge. We will discuss several techniques to reduce variance for original SF estimator later.

3.2 Reparameterization Trick

In cases where Z is reparameterizable (i.e. one can rewrite $Z = g(\alpha, \epsilon)$ as a function of parameter α and a random variable $\epsilon \sim p_\epsilon$), the gradient can be computed with chain rule:

$$\nabla_\alpha \mathbb{E}_{Z\sim p_\alpha(Z)}[L_\theta(Z)] = \nabla_\alpha \mathbb{E}_{\epsilon\sim p_\epsilon}[L_\theta(g(\alpha,\epsilon))] = \mathbb{E}_{\epsilon\sim p_\epsilon}[L'_\theta(g(\alpha,\epsilon))\nabla_\alpha g(\alpha,\epsilon)]. \tag{3.4}$$

A simple example is that if we consider a Gaussian distribution $X \sim \mathcal{N}(\mu, \sigma^2)$, then X can be reparameterize as $X = \mu + \sigma \cdot \mathcal{N}(0, 1)$. With this reparameterization, instead of sampling X directly, one can sample $\epsilon \sim \mathcal{N}(0, 1)$ and use $X = \mu + \sigma\epsilon$ to calculate X, then calculate the gradient w.r.t μ and σ explicitly, making the problem of calculating the gradient w.r.t. parameters of a distribution to the problem of calculating the gradient w.r.t. parameters of a given function.

Reparameterization Trick provides an unbiased estimator with lower variance and it is often used when it is applicable. The reason why it is better than SF estimator is that the estimation depends on the derivative of L, which exposes the dependence of L on Z.

3.2.1 Gumbel-Softmax Estimator and SNAS

Though reparameterization trick provides a powerful tool to estimate gradient with lower variance, discrete random variable Z in our case is not reparameterizable. To overcome this difficulty, SNAS uses Gumbel-Softmax [Maddison et al., 2017, Jang et al., 2017] to

relax the discrete random variable $Z \sim p_\alpha(Z)$ to become a continuous reparameterizable random variable $\tilde{Z} = (\tilde{Z}_{i,j})_{i<j}$ and $\tilde{Z}_{i,j} = [\tilde{Z}^1_{i,j}, \ldots, \tilde{Z}^A_{i,j}]^T$, with reparameterization:

$$\begin{aligned} \tilde{Z}^k_{i,j} &= g_{i,j}(\alpha, U) \\ &= \frac{\exp((\alpha^k_{i,j} - \log(-\log(U^k_{i,j})))/\lambda)}{\sum_{l=1}^{A} \exp((\alpha^l_{i,j} - \log(-\log(U^l_{i,j})))/\lambda)}, \end{aligned} \tag{3.5}$$

where $U = \{U^k_{i,j}\}_{i,j,k}$ are some independent uniform random variables, $G^k_{i,j} = -\log(-\log(U^k_{i,j}))$ is Gumbel random variable and λ is the temperature of the Gumbel softmax, which is annealed to zero in SNAS.

According to [Maddison et al., 2017], $\tilde{Z}^k_{i,j}$ has following important properties:

Proposition 1. $\mathbb{P}(\tilde{Z}^k_{i,j} > \tilde{Z}^l_{i,j} \text{ for } l \neq k) = \frac{exp(\alpha^k_{i,j})}{\sum_{l=1}^{A} exp(\alpha^l_{i,j})}$.

Proposition 2. $\mathbb{P}(\lim_{\lambda \to 0} \tilde{Z}^k_{i,j} = 1) = \frac{exp(\alpha^k_{i,j})}{\sum_{l=1}^{A} exp(\alpha^l_{i,j})}$.

Proposition 1 shows that the original random variable $Z^k_{i,j}$ can be obtained by taking $\arg\max$ operation, i.e. $Z^k_{i,j} = 1$ if and only if $\tilde{Z}^k_{i,j}$ is the biggest element among all $\tilde{Z}^l_{i,j}$, $l = 1, \ldots, A$, otherwise $Z^k_{i,j} = 0$:

$$Z^k_{i,j} = \begin{cases} 1 & \textbf{if } k = \arg\max_{l=1,\ldots,A}\{\tilde{Z}^l_{i,j}\} \\ 0 & \textbf{otherwise} \end{cases} \tag{3.6}$$

This shows why we can regard $\tilde{Z}^k_{i,j}$ as a continuous relaxation of $Z^k_{i,j}$.

$$\tilde{Z}_{i,j} = [0.1, 0.3, 0.1, 0.4, 0.05, 0.05] \longrightarrow Z_{i,j} = [0, 0, 0, 1, 0, 0] \tag{3.7}$$

1: An example of how to obtain $Z_{i,j}$ from $\tilde{Z}_{i,j}$

Proposition 2 proves that as $\lambda \to 0$, $\tilde{Z}^k_{i,j}$ becomes sharper and closer to discrete random variable. More precisely, $\tilde{Z}^k_{i,j}$ converges to $Z^k_{i,j}$ in distribution as the temperature $\lambda \to 0$.

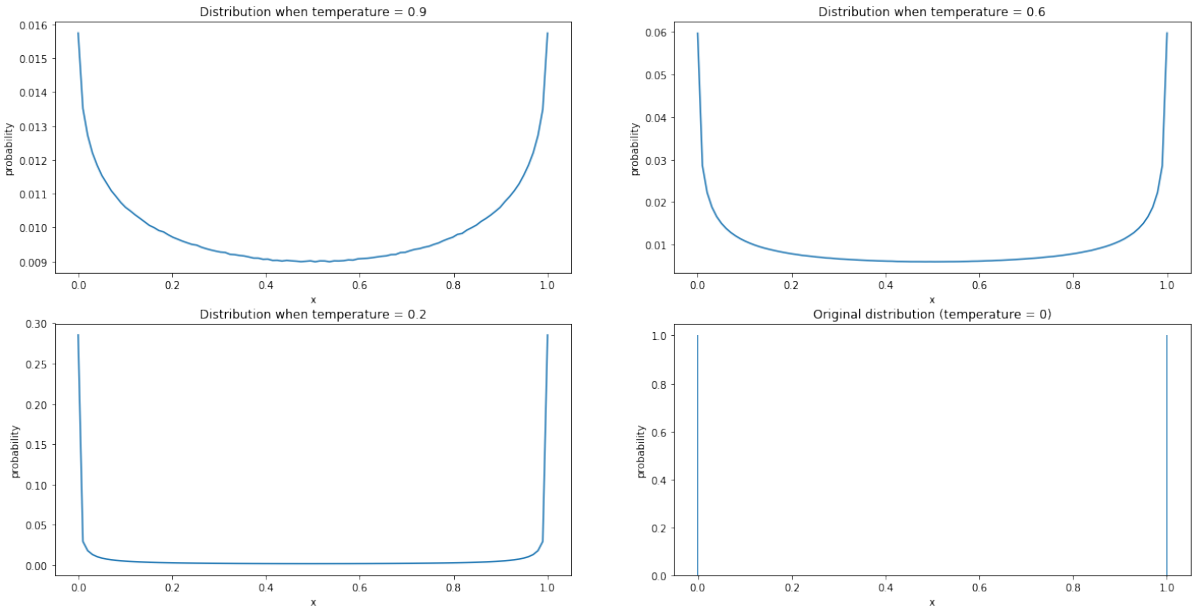


Figure 3.1: A visualization of distributions of Gumbel Softmax with different temperatures. Temperature $\lambda = 0$ correspond to original discrete distribution without relaxation. $\tilde{Z}^k_{i,j}$ becomes sharper and converge to discrete distribution as $\lambda \to 0$.

With this approximation, objective now becomes:

$$\min_{\alpha,\theta} \mathbb{E}_{\tilde{Z}\sim\tilde{p}_\alpha(\tilde{Z})}[L_\theta(\tilde{Z})] = \min_{\alpha,\theta} \mathbb{E}_U[L_\theta(g(\alpha, U))], \tag{3.8}$$

where $g(\alpha, U) = (g_{i,j}(\alpha, U))_{i,j}$. Since new objective in equation (3.8) is reparameterized by $g(\alpha, U)$, then base on equation (3.4), the gradients can be estimated by calculating the gradients over several Monte Carlo samples with automatic differentiation libraries. However, estimators become biased due to changes of objective function.

3.3 Score Function Estimators with Control Variates

As we have described in Section 3.1, Score Function (SF) estimator suffers from extremely high variance. Though its variance can be reduced by using control variates. A control variate is a function $c(Z)$ that in each sampling, we subtract c(Z) from the loss function $L_\theta(Z)$ and then manually adding back the mean value $\mathbb{E}_{Z\sim p_\alpha(Z)}[c(Z)\nabla_\alpha \log p_\alpha(Z)]$, gives

us a new estimator without bias since:

$$\begin{aligned}\nabla_\alpha \mathbb{E}_{Z\sim p_\alpha(Z)}[L_\theta(Z)] &= \mathbb{E}_{Z\sim p_\alpha(Z)}[L_\theta(Z)\nabla_\alpha \log p_\alpha(Z)] \\ &= \mathbb{E}_{Z\sim p_\alpha(Z)}[(L_\theta(Z) - c(Z))\nabla_\alpha \log p_\alpha(Z)] + \mathbb{E}_{Z\sim p_\alpha(Z)}[c(Z)\nabla_\alpha \log p_\alpha(Z)].\end{aligned} \tag{3.9}$$

Thus the new estimator is:

$$\hat{\nabla}_\alpha \mathbb{E}_{Z\sim p_\alpha(Z)}[L_\theta(Z)] = \frac{1}{S}\sum_{i=1}^{S}(L_\theta(Z_i) - c(Z_i))\nabla_\alpha \log p_\alpha(Z_i) + \mathbb{E}_{Z\sim p_\alpha(Z)}[c(Z)\nabla_\alpha \log p_\alpha(Z)], \tag{3.10}$$

it has lower variance if $c(Z)$ is positively correlated with $L_\theta(Z)\nabla_\alpha \log p_\alpha(Z)$. Score function estimators with control variates require knowledge of the exact value of expectation $\mathbb{E}_{Z\sim p_\alpha(Z)}[c(Z)\nabla_\alpha \log p_\alpha(Z)]$, though sometimes one can approximate it using various techniques.

3.3.1 Score Function with Constant Variates

If $c(Z) = c$ is constant that does not depend on Z, then the estimator becomes:

$$\hat{\nabla}_\alpha \mathbb{E}_{Z\sim p_\alpha(Z)}[L_\theta(Z)] = \frac{1}{S}\sum_{i=1}^{S}(L_\theta(Z_i) - c)\nabla_\alpha \log p_\alpha(Z_i). \tag{3.11}$$

A common used technique in REINFORCE [J.Williams, 1992] is taking $c = \mathbb{E}_{Z\sim p_\alpha(Z)}[L_\theta(Z)]$ and approximating it by moving average of $L_\theta(Z)$, which can significantly reduce the variance with little additional computational cost in practice. We implement Score function with constant variates in our experiments.

3.3.2 RELAX Estimators

Despite that constant variates have little computational cost, in some situation a more precise estimation is desired. Several techniques have been proposed to trade increased computational cost for decreased variance with non-constant variates $c(Z)$ [Mnih and Gregor, 2014, Gregor et al., 2014, Gu et al., 2016, Mnih and Rezende, 2016]. Besides, [Grathwohl et al., 2018] proposes an unbiased estimator which they call RELAX, to combine the score function estimator, reparameterization trick and control variates.

The idea behind RELAX estimator is to use a neural network ϕ to calculate the control variates $c_\phi(Z)$. Equation (3.9) can be written as:

$$\begin{aligned}\nabla_\alpha \mathbb{E}_{Z\sim p_\alpha(Z)}[L_\theta(Z)] &= \mathbb{E}_{Z\sim p_\alpha(Z)}[(L_\theta(Z) - c_\phi(Z))\nabla_\alpha \log p_\alpha(Z)] + \nabla_\alpha \mathbb{E}_{Z\sim p_\alpha(Z)}[c_\phi(Z)] \\ &= \mathbb{E}_{Z\sim p_\alpha(Z)}[(L_\theta(Z) - c_\phi(Z))\nabla_\alpha \log p_\alpha(Z)] + \mathbb{E}_{\epsilon\sim p_\epsilon(Z)}[c'_\phi(g(\alpha,\epsilon))\nabla_\alpha g(\alpha,\epsilon)],\end{aligned} \tag{3.12}$$

since

$$\nabla_\alpha \mathbb{E}_{Z\sim p_\alpha(Z)}[c_\phi(Z)] = \mathbb{E}_{\epsilon\sim p_\epsilon}(Z)[c'_\phi(g(\alpha,\epsilon))\nabla_\alpha g(\alpha,\epsilon)], \tag{3.13}$$

when Z is reparameterizable and reparameterized by $Z = g(\alpha, \epsilon)$.

Then ϕ is trained during each iteration in order to minimize the variance of the SF estimator

$$\mathrm{Var}[(L_\theta(Z) - c_\phi(Z))\nabla_\alpha \log p_\alpha(Z) + c'_\phi(g(\alpha,\epsilon))\nabla_\alpha g(\alpha,\epsilon)], \tag{3.14}$$

or alternatively its second moment

$$\mathbb{E}[((L_\theta(Z) - c_\phi(Z))\nabla_\alpha \log p_\alpha(Z) + c'_\phi(g(\alpha,\epsilon))\nabla_\alpha g(\alpha,\epsilon))^2], \tag{3.15}$$

which can be estimated using the same Monte-Carlo samples. The Entire algorithm summarized in the following pseudo-code.

Algorithm 1 LAX Estimator

Require: Loss function $L_\theta(\cdot)$; Control variate neural network $c_\phi(\cdot)$; Reparameterize function $g(\cdot,\cdot)$; $\log p_\alpha(\cdot)$; Step size $l_\alpha, l_\phi, l_\theta$.

1: **for** number of iterations **do**
2: sample ϵ
3: calculate $Z = g(\alpha, \epsilon)$
4: calculate gradient estimation $g_\alpha = [L_\theta(Z) - c_\phi(Z)]\nabla_\alpha \log p_\alpha(Z) + \nabla_\alpha c_\phi(Z)$
5: calculate gradient of second moment $g_\phi = \nabla_\phi(g_\alpha)^2$
6: calculate gradient of θ: $g_\theta = \nabla_\theta L_\theta(Z)$
7: update α: $\alpha = \alpha - l_\alpha g_\alpha$
8: update ϕ: $\phi = \phi - l_\phi g_\phi$
9: update θ: $\theta = \theta - l_\theta g_\theta$
10: **end for**

However, in our case Z is not reparameterizable thus equation (3.13) no longer holds. An alternative solution is to use continuous relaxation $\tilde{Z}$ of discrete random variable Z,

reparameterize $\tilde{Z}$ by $\tilde{Z} = g(\alpha, U)$ and rewrite equation (3.12) as

$$\begin{aligned}
&\nabla_\alpha \mathbb{E}_{Z\sim p_\alpha(Z)}[L_\theta(Z)] \\
&= \mathbb{E}_{Z\sim p_\alpha(Z)}[L_\theta(Z)\nabla_\alpha \log p_\alpha(Z)] - \mathbb{E}_{\tilde{Z}\sim p_\alpha(\tilde{Z})}[c_\phi(\tilde{Z})\nabla_\alpha \log p_\alpha(\tilde{Z})] + \nabla_\alpha \mathbb{E}_{\tilde{Z}\sim p_\alpha(\tilde{Z})}[c_\phi(\tilde{Z})] \\
&= \mathbb{E}_U[L_\theta(Z)\nabla_\alpha \log p_\alpha(Z)] - \mathbb{E}_U[c_\phi(\tilde{Z})\nabla_\alpha \log p_\alpha(\tilde{Z})] + \mathbb{E}_U[\nabla_\alpha c_\phi(\tilde{Z})],
\end{aligned} \tag{3.16}$$

where U is uniform random variable, $\tilde{Z} = g(\alpha, U)$ and Z is obtained from $\tilde{Z}$ by taking argmax operaion (3.6). This is an unbiased estimator which can serve as an alternative version when Z is not reparameterizable, in [Grathwohl et al., 2018] it is called LAX estimator for discrete random variables.

[Grathwohl et al., 2018] also proposes an advanced version called RELAX estimator which is based on the identity:

$$\nabla_\alpha \mathbb{E}_{Z\sim p_\alpha(Z)}[L_\theta(Z)] = \mathbb{E}_{U,\tilde{Z}_{cond}}[(L_\theta(Z) - c_\phi(\tilde{Z}_{cond}))\nabla_\alpha \log p_\alpha(Z)] + \mathbb{E}_{U,\tilde{Z}_{cond}}[\nabla_\alpha (c_\phi(\tilde{Z}) - c_\phi(\tilde{Z}_{cond}))], \tag{3.17}$$

where

- U is uniform random variable
- $\tilde{Z} = g(\alpha, U)$ which represent the continuous relaxation of Z
- Z is obtained from $\tilde{Z}$ by taking operaion (3.6)
- Finally $\tilde{Z}_{cond}$ is random variable of $\tilde{Z}$ conditions on Z, i.e. $\tilde{Z}_{cond} \sim \tilde{Z}|Z$.

Details of how to sample $\tilde{Z}_{cond}$ from Z is given by [Tucker et al., 2017, Grathwohl et al., 2018].

Algorithm 2 RELAX Estimator

Require: Loss function $L_\theta(\cdot)$; Control variate neural network $c_\phi(\cdot)$; Reparameterize function $g(\cdot,\cdot)$; $\log p_\alpha(\cdot)$; Step size $l_\alpha, l_\phi, l_\theta$.

1: **for** number of iterations **do**
2: sample ϵ
3: calculate $\tilde{Z} = g(\alpha, \epsilon)$
4: calculate $Z = argmax(\tilde{Z})$
5: sample $\tilde{Z}_{cond} \sim \tilde{Z}|Z$
6: calculate gradient estimation $g_\alpha = [L_\theta(Z) - c_\phi(\tilde{Z}_{cond})]\nabla_\alpha \log p_\alpha(Z) + \nabla_\alpha(c_\phi(\tilde{Z}) - c_\phi(\tilde{Z}_{cond}))$
7: calculate gradient of second moment $g_\phi = \nabla_\phi(g_\alpha)^2$
8: calculate gradient of θ: $g_\theta = \nabla_\theta L_\theta(Z)$
9: update α: $\alpha = \alpha - l_\alpha g_\alpha$
10: update ϕ: $\phi = \phi - l_\phi g_\phi$
11: update θ: $\theta = \theta - l_\theta g_\theta$
12: **end for**

Note that all of the calculations of gradient in RELAX estimator can be done by automatic differentiation libraries.

3.4 Finite-Difference Estimator

In this subsection, we review the reparameterization and marginalization (RAM) estimator introduced by [Tokui and sato, 2016, Andriyash et al., 2018]. Let us consider a single binary stochastic variable (i.e. a DAG with only two nodes: input node and output node, with two candidate operations: $\{O^1, O^2\}$). We denote $e_1 = [1, 0]$ and $e_2 = [0, 1]$ the ith entry vector. Then we can calculate the expected loss function as:

$$\mathbb{E}_{Z\sim p_\alpha(Z)}[L_\theta(Z)] = p(e_2)L_\theta(e_2) + (1 - p(e_2))L_\theta(e_1), \tag{3.18}$$

where $p(e_1) = \mathbb{P}(Z = e_1)$ and $p(e_2) = \mathbb{P}(Z = e_2)$, as well as its gradient w.r.t. α:

$$\nabla_\alpha \mathbb{E}_{Z\sim p_\alpha(Z)}[L_\theta(Z)] = \nabla_\alpha p(e_2)(L_\theta(e_2) - L_\theta(e_1)). \tag{3.19}$$

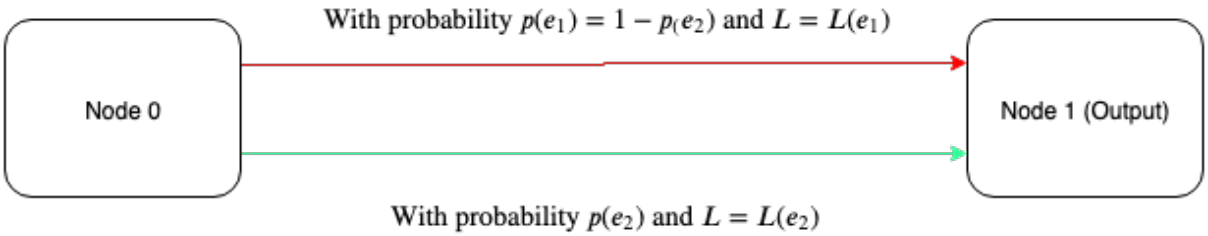


Figure 3.2: Forward pass for DAG with two nodes and two candidate operations.

Similarly, for a DAG with multi-node and two candidate operations, if we denote E its set of edges, then the gradient is:

$$\begin{aligned}&\nabla_\alpha \mathbb{E}_{Z\sim p_\alpha(Z)}[L_\theta(Z)] \\ &= \sum_{(i,j)\in E} \nabla_\alpha p_{i,j}(e_1) \sum_{Z_{\setminus(i,j)}} p_{\setminus(i,j)}(Z_{\setminus(i,j)})(L_\theta(Z_{i,j} = e_1, Z_{\setminus(i,j)}) - L_\theta(Z_{i,j} = e_0, Z_{\setminus(i,j)})),\end{aligned} \tag{3.20}$$

where $p_{i,j}$ is the marginal probability of $Z_{i,j}$, $Z_{\setminus(i,j)}$ represent Z without $Z_{i,j}$ and

$$p_{\setminus(i,j)}(Z_{\setminus(i,j)}) = \prod_{(i',j')\in E,(i',j')\neq(i,j)} p_{i',j'}(Z_{i',j'}). \tag{3.21}$$

This estimation can also be extended to multi-candidate operation case. We denote e_i the ith entry vector, then RAM estimator in multi-node and multi-candidate operation case can be represented as:

$$\begin{aligned}\nabla_\alpha \mathbb{E}_{Z\sim p_\alpha(Z)}[L_\theta(Z)] = \sum_{(i,j)\in E} \sum_{Z_{\setminus(i,j)}} p_{\setminus(i,j)}(Z_{\setminus(i,j)}) \sum_{e_a,e_b} (\nabla_\alpha l^a_{i,j}) p_{i,j}(e_a) p_{i,j}(e_b)[L_\theta(Z_{i,j} = e_a, Z_{\setminus(i,j)}) \\ - L_\theta(Z_{i,j} = e_b, Z_{\setminus(i,j)})]\end{aligned} \tag{3.22}$$

Here $(l^1_{i,j}, \ldots, l^A_{i,j}) = \text{logit}(p^{e_1}_{i,j}, \ldots, p^{e_A}_{i,j})$ and $(p^{e_1}_{i,j}, \ldots, p^{e_A}_{i,j}) = \text{softmax}(l^1_{i,j}, \ldots, l^A_{i,j})$. In our case, $(l^1_{i,j}, \ldots, l^A_{i,j}) = (\alpha^1_{i,j}, \ldots, \alpha^A_{i,j})$, makes it trivial to calculate $\nabla_\alpha l^a_{i,j}$.

Suppose now the number of edge is $|E|$ and the number of operations is A, then according to (3.22), we have two sampling strategies:

1. For each sample Z, we calculate

$$\sum_{e_a,e_b}(\nabla_\alpha l^a_{i,j})p_{i,j}(e_a)p_{i,j}(e_b)[L_\theta(Z_{i,j}=e_a,Z_{\setminus(i,j)})-L_\theta(Z_{i,j}=e_b,Z_{\setminus(i,j)})] \tag{3.23}$$

   for each edge $(i,j)\in E$. This requires $|E|A$ evaluations (i.e. calculate loss for $|E|A$ architecture structures), which could be done by parallelization.

2. Alternatively, for a given sample Z and for each edge $(i,j)\in E$, one can estimate equation (3.23) by sampling e_a and e_b independently with distribution $p_{i,j}(Z_{i,j})$. Then estimate (3.23) by:

$$(\nabla_\alpha l^a_{i,j})[L_\theta(Z_{i,j}=e_a,Z_{\setminus(i,j)})-L_\theta(Z_{i,j}=e_b,Z_{\setminus(i,j)})]. \tag{3.24}$$

   This method reduces the number of evaluations needed from $|E|A$ to $2\times|E|$, whereas it increases the variance.

RAM estimator has relatively low-variance due to $L_\theta(Z_{i,j}=e_a,Z_{\setminus(i,j)})-L_\theta(Z_{i,j}=e_b,Z_{\setminus(i,j)})$ are evaluated at the same $Z_{\setminus(i,j)}$. [Tokui and sato, 2016] has proved that it is better than any other Score Function estimator with constant variates.

Following the setting of ENAS, DARTS, SNAS, we search for computation cells which contains 7 nodes (2 input nodes, 4 intermediate nodes and 1 output node) and 8 possible operations, thus 14 connections in normal cell as well as reduction cell. In this case, we have $|E|=28$ and $A=8$. First strategy requires $28\times 8=224$ evaluations in each iteration whereas second strategy requires $28\times 2=56$ evaluations. To get a smaller variance, we perform the first strategy in our experiment.

4 EXPERIMENTS

We perform experiments using CIFAR-10 dataset, which consists of 60000 images of size 32×32 classified into 10 classes. CIFAR-10 is devided into training set consists of 50000 as well as test set consists of 10000 images. Our experiment consists of two stages, architecture search stage and architecture evaluation stage. In architecture search stage, the motivation is to search optimal convolutional cells in a small network on CIFAR-10. Then in architecture evaluation stage, a fully-sized network is built by stacking the convolutional cells learned in architecture search stage and retrained on CIFAR-10. Several techniques are used during the architecture search stage to find the convolutional cells (SF estimator with constant variates, RELAX estimator and RAM estimator) and we compare their validation accuracy with other state-of-the-art methods.

4.1 Architecture Search using CIFAR-10

4.1.1 Search space

Following the same setting as in DARTS, 8 operations are considered as candidate operations: None, Identity, 3×3 max pooling, 3×3 average pooling, 3×3 and 5×5 separable convolutions, 3×3 and 5×5 dilated separable convolutions. Convolutional cells consist of $N = 7$ nodes, where the first node and the second node in cell k represent the input nodes, they are set equal to the outputs of cells $k - 2$ and $k - 1$. Output of each cell is defined as the concatenation of all the intermediates nodes in this cell. The whole network is designed by stacking several cells together. To reduce the spatial resolution of feature maps, two reduction cells are used by setting strides $= 2$ for all operations adjacent to input nodes of the cell. They are located at the 1/3 and 2/3 of the total depth of the network. Other cells are called Normal cells. Thus the architecture parameters can be represented as $\alpha = (\alpha_{normal}, \alpha_{reduction})$, where α_{normal} represent parameters shared by all normal cells and $\alpha_{reduction}$ represent parameters shared by all reduction cells.

4.1.2 Training process

During architecture search stage, we train a small network consists of 8 cells for 150 epochs. Thus 3th and 6th cell are reduction cells. The initial number of channels is set to 16 whereas the batch size is determined in order to fit into a single GPU. We use momentum SGD with initial learning weight $l_\theta = 0.025$ and momentum 0.9 to optimize parameters θ. Learing weight l_θ is annealed to zero following a cosine schedule with weight decay 0.0003. To train the architecture parameters α, we use Adam optimizer with learning rate $l_\alpha = 0.03$ and momentum $\beta = (0.5, 0.999)$. During the training process, the training set for CIFAR-10 is divided into $(50\%, 50\%)$, where the first 50% is used to train the parameters θ whereas the second 50% is used to train the architecture parameters α. The training process for architecture search stage is represented as following:

Algorithm 3 Training process for architecture search

Require: Loss function $L_\theta(\cdot)$; Step size l_α, l_θ. Training set for α and θ: S_α, S_θ

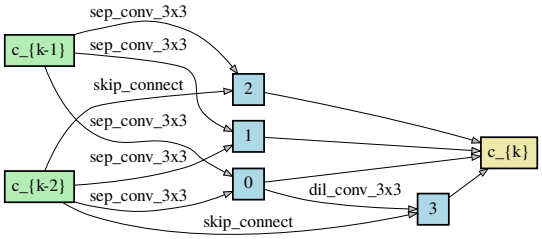
1: **for** number of epochs **do**
2: Estimate gradient $\tilde{\nabla}_\theta$ from S_θ
3: Estimate gradient $\tilde{\nabla}_\alpha$ from S_α
4: Update $\alpha = \alpha - l_\alpha \tilde{\nabla}_\alpha$
5: Update $\theta = \theta - l_\theta \tilde{\nabla}_\theta$
6: **end for**

4.1.3 Searching Results

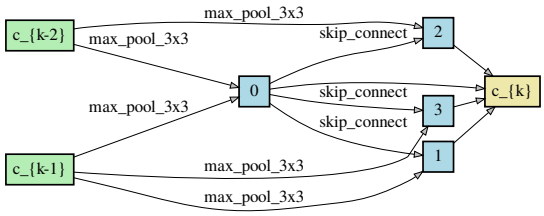
Once the training process is finished. The final operations in each cells are determined by takinig argmax operations over $\alpha_{i,j} = (\alpha_{i,j}^k)_{k=1,\dots,A}$ for each $(i,j) \in E$, i.e.

$$\tilde{O}_{i,j} = \begin{cases} O_{i,j}^1 & \textbf{if } \alpha_{i,j}^1 > \alpha_{i,j}^k \textbf{ for each } k \\ \cdots & \\ O_{i,j}^A & \textbf{if } \alpha_{i,j}^A > \alpha_{i,j}^k \textbf{ for each } k. \end{cases}$$

Here are the convolutional cells found using DARTS and SNAS:

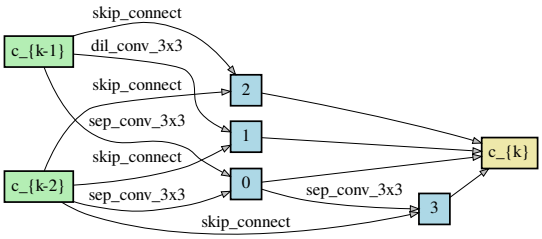


(a) Normal cell found by DARTS

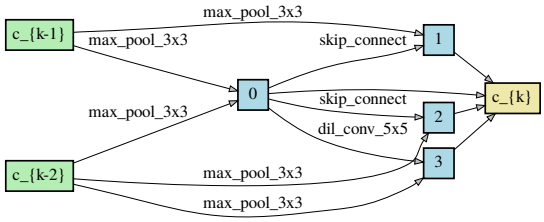


(b) Reduction cell found by DARTS

Figure 4.1: Normal cell and reduction cell (child graph) found by DARTS on CIFAR-10 [Liu et al., 2019]. (a) Normal cell. (b) Reduction cell.



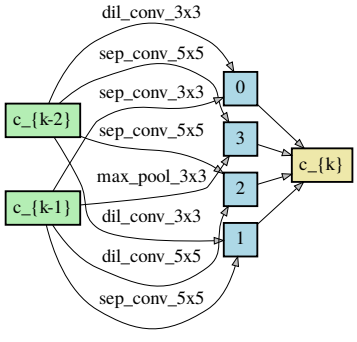
(a) Normal cell found by SNAS



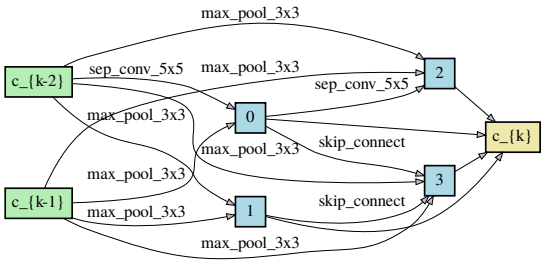
(b) Reduction cell found by SNAS

Figure 4.2: Normal cell and reduction cell (child graph) found by SNAS on CIFAR-10 [Xie et al., 2019] (a) Normal cell. (b) Reduction cell.

As the designed scheme in ENAS and DARTS, we restrain each node to take only two input edges with the largest probabilities. Here are some convolutional cells found using RAM and SF (before restrain and after restrain):

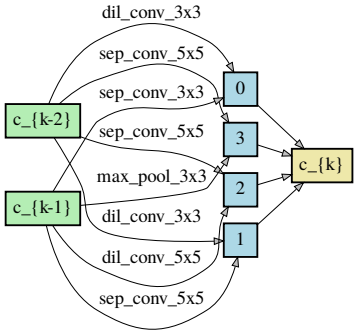


(a) Normal cell found by RAM before restrain

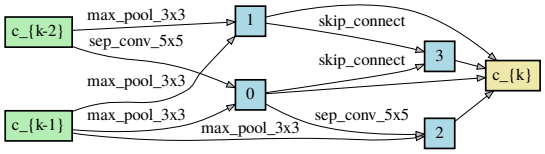


(b) Reduction cell found by RAM before restrain

Figure 4.3: Normal cell and reduction cell (child graph) found by RAM on CIFAR-10. (a) Normal cell before restrain. (b) Reduction cell after restrain.

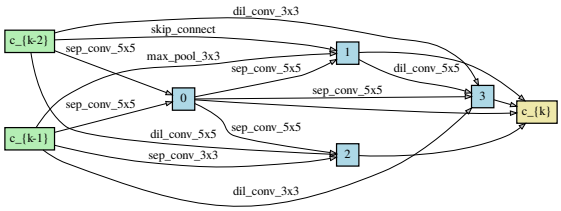


(a) Normal cell found by RAM after restrain

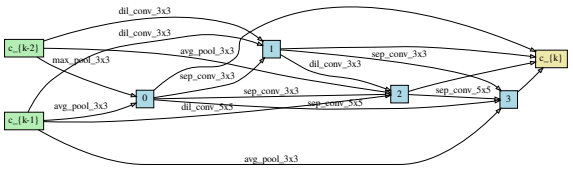


(b) Reduction cell found by RAM after restrain

Figure 4.4: Normal cell and reduction cell (child graph) found by RAM on CIFAR-10. (a) Normal cell after restrain. (b) Reduction cell after restrain.

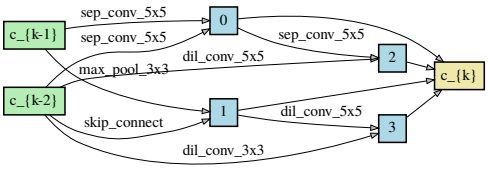


(a) Normal cell found by SF before restrain

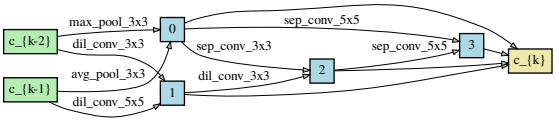


(b) Reduction cell found by SF before restrain

Figure 4.5: Normal cell and reduction cell (child graph) found by SF on CIFAR-10. (a) Normal cell before restrain. (b) Reduction cell after restrain.

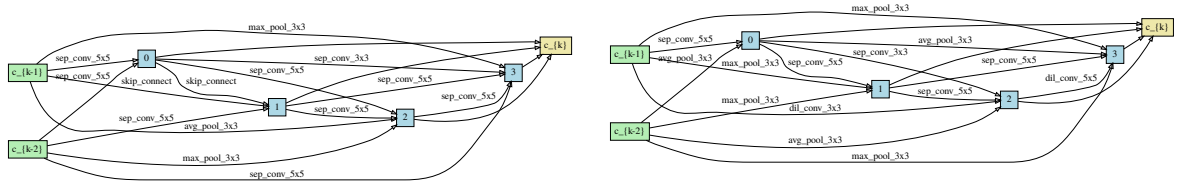


(a) Normal cell found by SF after restrain



(b) Reduction cell found by SF after restrain

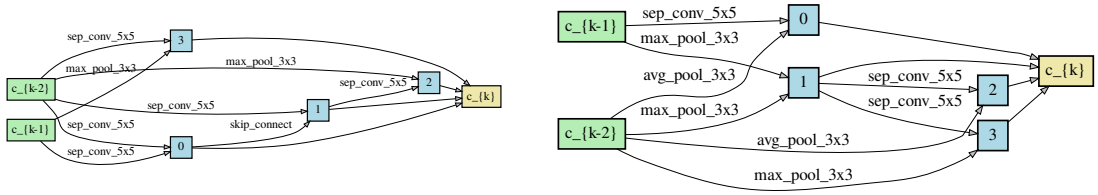
Figure 4.6: Normal cell and reduction cell (child graph) found by SF on CIFAR-10. (a) Normal cell after restrain. (b) Reduction cell after restrain.



(a) Normal cell found by RELAX before restrain

(b) Reduction cell found by RELAX before restrain

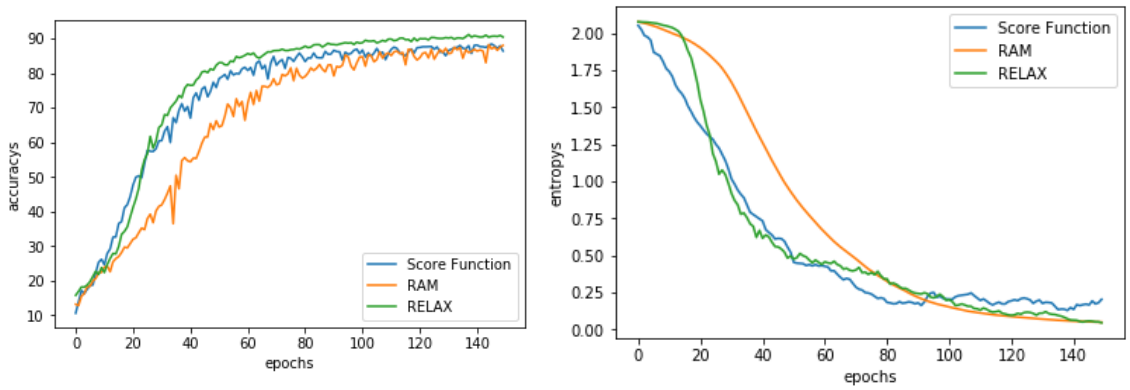
Figure 4.7: Normal cell and reduction cell (child graph) found by RELAX on CIFAR-10. (a) Normal cell before restrain. (b) Reduction cell after restrain.



(a) Normal cell found by RELAX after restrain

(b) Reduction cell found by RELAX after restrain

Figure 4.8: Normal cell and reduction cell (child graph) found by RELAX on CIFAR-10. (a) Normal cell after restrain. (b) Reduction cell after restrain.



(a) Validation accuracy during architecture search with RAM, RELAX and Score Function estimator.

(b) Entropy of architecture distribution during architecture search with RAM, RELAX and Score Function estimator.

Figure 4.9: Validation accuracy and entropy of architecture distribution during architecture search

4.2 Architecture Evaluation on CIFAR-10

To evaluate the performance of the neural network obtained in architecture search stage, we follow the same process as in ENAS, DARTS and SNAS. Once the convolutional cells are found in architecture search stage, a larger network consists of 20 cells where reduction cells are located at 1/3 and 2/3 of the total depth of the network is trained from scratch for 900 epochs with batch size to be determined in order to fit into a single GPU. Other parameters are the same as in DARTS. The training process for architecture evaluation stage takes around 2 days on a single GPU. We evaluate convolutional cells found by three different architecture search strategies: RAM estimator (figure 4.4), SF estimator with constant variates (figure 4.6), RELAX estimator (figure 4.8), and compare their results with other image classifiers on CIFAR-10.

Architecture	Test Error(%)	Params(M)	GPU days	Search Method
NASNet-A[Zoph and Le, 2016]	2.65	3.3	1800	RL
AmoebaNet-A[Real et al., 2019]	3.34	3.2	3150	evolution
AmoebaNet-B[Real et al., 2019]	2.55	2.8	3150	evolution
Hierarchical Evo[Liu et al., 2017]	3.75	15.7	300	evolution
PNAS[Liu et al., 2018]	3.41	3.2	225	SMOB
ENAS[Pham et al., 2018]	2.89	4.6	0.5	RL
DARTS[Liu et al., 2019]	2.76	3.3	1	gradient-based
SNAS[Xie et al., 2019]	2.85	2.8	1.5	gradient-based
RAM	2.62	3.6	1.25	gradient-based
SF	2.64	3.4	0.4	gradient-based
RELAX	2.70	3.6	0.6	gradient-based

Table 4.1: Classification errors of different estimators with other state-of-the-art image classifiers on CIFAR-10. All of our experiments are done using a single V100 GPU.

Table 4.1 represents the evaluation results on CIFAR-10. As shown in the table, all of our methods (SF, RAM, RELAX) achieved comparable results with the state-of-the-art RL-based and evolution-based Neural Architecture Search strategies with three orders of magnitude less computation resources. Besides, our methods outperform other gradient-based Neural Architecture Search strategies like DARTS and SNAS on the same search space.

5 Conclusion

In this work, we present several gradient-based neural architecture search frameworks by using stochastic architecture search which is proved to be efficient in SNAS. Comparing to other gradient-based neural architecture search strategies, the key contribution of our work is that we develop different neural architecture search strategies by introducing several unbiased and low variance gradient estimators. Our experiments on CIFAR-10 show that all of our new strategies match or outperform other frameworks on the same search space.

References

E. Andriyash, A. Vahdat, and B. Macready. Improved gradient-based optimization over discrete distributions. *arXiv:1810.00116v2 [stat.ML]*, 2018.

W. Grathwohl, D. Choi, Y. Wu, G. Roeder, and D. Duvenaud. Backpropagation through the void: Optimizing control variates for black-box gradient estimation. *arXiv:1711.00123v3 [cs.LG]*, 2018.

K. Gregor, I. Danihelka, A. Mnih, C. Blundell, and D. Wierstra. Deep autoregressive networks. *arXiv:1310.8499v2 [cs.LG]*, 2014.

S. Gu, S. Levine, I. Sutskever, and A. Mnih. Muprop: Unbiased backpropagation for stochastic neural networks. *arXiv:1511.05176v3 [cs.LG]*, 2016.

E. Jang, S. Gu, and B. Poole. Categorical reparameterization with gumbel-softmax. *arXiv:1611.01144v5 [stat.ML]*, 2017.

H. Jin and Q. S. X. Hu. Auto-keras: An efficient neural architecture search system. *arXiv:1806.10282v3 [cs.LG]*, 2019.

R. J.Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Reinforcement Learning, pp. 5-32.*, 1992.

K. Kandasamy, W. Neiswanger, J. Schneider, B. Poczos, and E. Xing. Neural architecture search with bayesian optimisation and optimal transport. *arXiv:1802.07191v3 [cs.LG]*, 2019.

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. *arXiv:1712.00559v3 [cs.CV]*, 2018.

H. Liu, K. Simonyan, O. Vinyals, C. Fernando, and K. Kavukcuoglu. Hierarchical representations for efficient architecture search. *arXiv:1711.00436v2 [cs.LG]*, 2017.

H. Liu, K. Simonyan, and Y. Yang. Darts: Differentiable architecture search. *arXiv:1806.09055v2 [cs.LG]*, 2019.

C. J. Maddison, A. Mnih, and Y. W. Teh. The concrete distribution: A continuous relaxation of discrete random variables. *arXiv:1611.00712v3 [cs.LG]*, 2017.

A. Mnih and K. Gregor. Neural variational inference and learning in belief networks. *arXiv:1402.0030v2 [cs.LG]*, 2014.

A. Mnih and D. J. Rezende. Variational inference for monte carlo objectives. *arXiv:1602.06725v2 [cs.LG]*, 2016.

H. Pham, M. Y. Guan, B. Zoph, Q. V. Le, and J. Dean. Efficient neural architecture search via parameter sharing. *arXiv:1802.03268v2 [cs.LG]*, 2018.

E. Real, A. Aggarwal, Y. Huang, and Q. V. Le. Regularized evolution for image classifier architecture search. *arXiv:1802.01548v7 [cs.NE]*, 2019.

S. Tokui and I. sato. Categorical reparameterization with gumbel-softmax. *arXiv:1611.01239v1 [stat.ML]*, 2016.

G. Tucker, A. Mnih, C. J. Maddison, D. Lawson, and J. Sohl-Dickstein. Rebar: Low-variance, unbiased gradient estimates for discrete latent variable models. *arXiv:1703.07370v4 [cs.LG]*, 2017.

S. Xie, H. Zheng, C. Liu, and L. Lin. Snas: Stochastic neural architecture search. *arXiv:1812.09926v2 [cs.LG]*, 2019.

B. Zoph and Q. V. Le. Neural architecture search with reinforcement learning. *arXiv:1611.01578v2 [cs.LG]*, 2016.