AUTOMATED MACHINE LEARNING WITH CONSTRAINTS AND IMPERFECT DATA

A Dissertation

by

YI-WEI CHEN

Submitted to the Graduate and Professional School of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Chair of Committee,	Xia Hu
Committee Members,	P.R. Kumar
	Eun Jung Kim
	Zhangyang Wang
Head of Department,	Scott Schaefer

December 2022

Major Subject: Computer Science

ABSTRACT

Machine learning has succeeded in real-world applications from image classification, speech recognition, to beating human champion in Go games. To accelerate the development of different applications, automated machine learning (AutoML) has been proposed to discover high-performance machine learning models automatically. It could release the burden of data scientists from the multifarious manual tuning process. However, dataset does not always have correct labels and sufficient data size. Wrong labels disrupt the training procedure and could not provide representative evaluation performance for AutoML. Imbalanced label distribution further skews search feedback for AutoML. Furthermore, additional performance constraints, such as model size, fairness, and robustness complicates the AutoML flow. When computing resources are insufficient, small search space constrains the flexibility to search neural networks, which causes inconsistent architectures used in search and evaluation stages. In this dissertation, I advanced AutoML from aspects of imperfect data and constraints. A new robust loss function is integrated with search algorithm for label noise. I design a simple but effective search space for imbalanced defect datasets. The defect generator can alleviate imbalanced distributions. I also proposed constraint-aware early stopping for AutoML with adaptive constraint evaluation intervals. An efficient model parallelism for AutoML is proposed to extend search spaces in multiple GPUs with limited memory size. My research of automated machine learning enables scientists to obtain off-to-shelf models on various data formats, as well as customizes models for different computing resources, model size requirements, and miscellaneous performance constraints. It broadly impacts image classification, constrained AutoML, and defect detection.

DEDICATION

To Meng-Hua Guo.

ACKNOWLEDGMENTS

I would like to thank Dr. Xia Hu for strong support of my research. He encouraged me to complete the AutoML survey in my early research stage, making me understand the field in width and depth. He is open-minded to discuss a variety of research topics, creating an active and productive working atmosphere. I sincerely appreciate his steady supports for my life, research, internship, and my future career. His generous and industrious attitude inspires me to become a better researcher. I would also like to thank the rest of my dissertation committee, Dr. Zhangyang Wang, Dr. Eun Jung Kim, and Dr. P.R. Kumar for their advice. I am grateful to Dr. Chi Wang, Dr. Amin Saied, and Dr. Rui Zhuang, who gave me inspiring research discussions during my research project at Microsoft. I would like to thank Dr. Chu Wang and Ms. Guangyu Zhang for their hlep and advice on my remote internship at Amazon. Many thanks to my outstanding lab colleagues at DATA Lab at Texas A&M University and Rice University. I enjoyed all research discussions and brainstorming. Special thanks to all the reviewers to my papers, the Texas A&M University, and all the funding agencies, including National Science Foundation and Defense Advanced Research Projects Agency.

Finally, I would love to express my sincere gratitude to my family, my mother, Qiu-Li Huang, my father, Sen-Xian Chen, mother-in-law, Su-Chin Jiang, father-in-law, Shui-Quan Guo, for their endless encouragement and love. I epxress the greatest thankfulness to my beloved wife, Meng-Hua Guo. All my achievements belong to her.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a dissertation committee consisting of Professor Xia Hu [advisor], Professor Eun Jung Kim, and Professor Zhangyang Wang of the Department of Computer Science and Engineering, and Professor P.R. Kumar of the Department of Electrical and Computer Engineering. All other work conducted for the thesis (or) dissertation was completed by the student independently.

Funding Sources

This work is, in part, supported by DARPA (#FA8750-17-2-0116 and #W911NF-16-1-0565) and NSF (#IIS-1750074 and #IIS-1657196). The views, opinions, and/or finding expressed are those of the author(s) and shoult not be inter

TABLE OF CONTENTS

Page

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

TABLE Page

1. INTRODUCTION *

1.1 Motivation and Challenges

Machine learning has been broadly implemented in a myriad of fields, from image classification [2], speech recognition [3], and recommendation platforms [4], to beating human champion in Go games [5]. Automated machine learning (AutoML) has emerged as a prevailing research field in both academia and industries. Given a machine learning problem, the goal of AutoML is to find high-performance machine learning solutions automatically with a little workforce in reasonable time budget. For example, Google HyperTune [6], Amazon Model Tuning [7], and Microsoft Azure AutoML [8] all provide cloud services cultivating off-the-shelf machine learning solutions for both researchers and practitioners. It could release the burden of data scientists from the multifarious manual tuning process and facilitate the development of solving machine learning problems. Ultimately, AutoML could provide off-the-shelf machine learning solutions for human beings without extensive ML experience.

AutoMLs are often characterized from a traditional machine learning pipeline [9]. Data scientists manually manipulate numerous features, design models, and tune hyperparameters in order to get the desired predictive performance. The procedure will not be terminated until a satisfactory performance is achieved. Thus, existing AutoMLs are categorized into the three categories, (a) AutoFE: automated feature engineering, (b) AutoMS: automated model selection, and (c) HPO: hyperparameter optimization. AutoFE searchs informative and discriminative features for a learning model. AutoMS selects shallow ML models or designs network architectures (NAS) for a learning problem. HPO discovers promising hyperparameter configurations to train the learning model toward its optimal performance. The essence of AutoMLs is a bi-level optimization [10], with an miscellaneous search space including features, hyperparameters, models, and network architectures.

Despite the prominent advances in the recent AutoMLs [11, 12, 13, 14, 15], the practical

*Parts of this chapter are reprinted with permission from "Techniques for Automated Machine Learning" by Yi-Wei Chen, Qingquan Song, and Xia Hu, 2021, ACM SIGKDD Explorations Newsletter, 22.2, p.35-50, Copyright 2021 by ACM SIGKDD Explorations Newsletter.

factors for AutoMLs in real-world are still under investigation. On the one hand, getting sufficient labeled data is prohibitively in new domain problems. For example, MVTec dataset of anomaly segmentation [16] merely includes hundreds of labeled anomaly map on manufacturing products. In contrast, the datasets of the well-explored image classification problem have at least ten thousands of labels, such as MNIST [17], CIFAR-10 [18], and ImageNet [19]. Sometimes, the labels are wrong, since the time-consuming labeling procedure is completed by non-expert crowd-sourcing service. Insufficient labels and label noise prevent AutoML from having enough signals to search model and hyperparameters precisely. On the other hand, when a ML model is deployed in real-world applications, the model is often required not only to optimize for ML objectives (e.g., accuracy or l2 loss), but also to meet the deployment constraints, such as latency, storage, fairness, and robustness. The demands ask AutoML to search solutions under specific deployment constraints.

Facilitating AutoML in practical situations is quite challenging. First of all, label noise corrupts the labels of training data. AutoML not only trains models with noisy training labels but are also compares their performances on noisy validation sets, which misleads the search direction. Second, the search spaces of network architectures consume gigantic GPUs. AutoML needs to leverage on multiple GPUs to accommodate the large search space. How to well utilize all GPUs in AutoML is an open research problem. Third, deployment constraints often require additional computation cost to evaluate. AutoML can waste a large amount of tuning cost on training ineligible configurations. How to select the evaluation frequency and when to stop ineligible ML solutions are important factors. Last but not least, the scarce labels or uneven label distribution in training data prevents AutoML from obtaining substantial search signals. AutoML needs to deal with such the weakly-supervised setting.

1.2 Dissertation Contributions

To tackle the above challenges, several contributions are made in the preliminary work, and a future work is proposed to conclude the dissertation :

- The first contribution of this research dissertation is the development of new search procedure

for neural architecture search (NAS) under symmetric label noise as well as under a simple model of class conditional label noise. We systematically explore the robustness of NAS under noisy labels and use robust loss functions to mitigate the performance degradation.

- The second contribution is the development of efficient model parallelism for NAS to alleviate massive GPU consumption. We integrate binary neural architecture search (NASB) with consecutive model parallel (CMP). CMP divides forward/backward phases into several sub-tasks and executes the same type of sub-tasks together to reduce waiting cycles. NASB excludes inactive operations from computation graphs to reduce memory footprint. NASB-CMP shows its potential to explore architectures in large search space.

- The third contribution of this dissertation is the development of Adaptive Constraint-aware Early stopping (ACE) for HPO. ACE estimates the cost-effective constraint evaluation interval based on a theoretical analysis of the expected evaluation cost. Meanwhile, we propose a stratum early stopping criterion in ACE, which considers both optimization and constraint metrics in pruning and does not require regularization hyperparameters. ACE shows superior performance in hyperparameter tuning of classification tasks under fairness or under robustness constraints.

- Considering the scarce labels or uneven label distribution in anomaly detection and anomaly segmentation, we will focus on automated anomaly generator and explore the small size of auto-encoders for edge devices in the continuing work. We intend to investigate how to synthesize anomalies, which are dissimilar to anomaly-free instances and similar to any anomalous instance. Meanwhile, we will develop a new search space of auto-encoders to design tiny networks for edge devices.

2. ON ROBUSTNESS OF NEURAL ARCHITECTURE SEARCH UNDER LABEL NOISE*

Neural architecture search (NAS), which aims at automatically seeking proper neural architectures given a specific task, has attracted extensive attention recently in supervised learning applications. In most real-world situations, the class labels provided in the training data would be noisy due to many reasons, such as subjective judgments, inadequate information, and random human errors. Existing work has demonstrated the adverse effects of label noise on the learning of weights of neural networks. These effects could become more critical in NAS since the architectures are not only trained with noisy labels but are also compared based on their performances on noisy validation sets. In this paper, we systematically explore the robustness of NAS under label noise. We show that label noise in the training and/or validation data can lead to various degrees of performance variations. Through empirical experiments, using robust loss functions can mitigate the performance degradation under symmetric label noise as well as under a simple model of class conditional label noise. We also provide a theoretical justification for this. Both empirical and theoretical results provide a strong argument in favor of employing the robust loss function in NAS under high-level noise.

2.1 Introduction

Label noise, which corrupts the labels of training instances, has been widely investigated due to its unavoidability in real-world situations and harmfulness to classifier learning algorithms [20]. Many recent studies have presented both empirical and analytical insights on learning of neural networks under label noise. Specifically, in the context of risk minimization, there are many recent studies on robust loss functions for learning classifiers under label noise [21, 22, 23].

The neural architecture search (NAS) seeks to learn an appropriate architecture also for a neural network in addition to learning the appropriate weights for the chosen architecture. It has the

*Reprinted with permission from "On Robustness of Neural Architecture Search Under Label Noise" by Yi-Wei Chen, Qingquan Song, Xi Liu, P.S. Sastry, and Xia Hu, 2020, Frontiers in Big Data, 3, p.2, Copyright 2020 by Frontiers in Big Data.

potential to revolutionize the deployment of neural network classifiers in a variety of applications. One requirement for such learning is a large number of training instances with correct labels. However, generating large sets of labeled instances is often difficult, and the process for labeling (e.g., crowdsourcing) has to contend with many random labeling errors. As mentioned above, label noise can adversely affect the learning of weights of a neural network. For NAS, the problem is compounded because we need to search for architecture as well. Since different architectures are learned using training data and compared based on their validation performance, label noise in training and validation (hold-out) data may cause a wrong assessment of architecture during the search process. Thus label noise can result in undesirable architectures being preferred by the search algorithm leading to the loss of performance. In this paper, we systematically investigate the effect of label noise on NAS. We show that label noise in the training or validation data can lead to different degrees of performance variation. Recently some robust loss functions are suggested for learning the weights of a network under label noise [21, 23]. The standard NAS algorithms use the categorical cross entropy (CCE) loss function. We demonstrate through simulations that the use of a robust loss function (in place of CCE) in NAS can mitigate the effect of harsh label noise. We provide a theoretical justification for this observed performance: for a class of loss functions that satisfy a robustness condition, we show that, under symmetric label noise, the relative risks of different classifiers are the same regardless of whether or not the data are corrupted with label noise.

2.2 Preliminaries

Robust Risk Minimization. In the context of multi-class classification, the feature vector is represented as $\mathbf{x} \in \mathcal{X} \subseteq \mathbb{R}^d$, and the corresponding class label denotes $y_{\mathbf{x}} \in [c] = \{1 \dots c\} = \mathcal{Y}$. A classifier $f : \mathcal{X} \to \mathbb{R}^c$ is learned to map each feature vector to a vector of scores, which are later used to decide a class. We assume f would be a DNN with the softmax output in this paper. Ideally, we could have a clean labeled dataset $D = \{(\mathbf{x}_i, y_{\mathbf{x}_i})\}_{i=1}^n$ drawn *i.i.d.* from an unknown joint distribution $\mathcal{D}$ over $(\mathcal{X} \times \mathcal{Y})$.

In the presence of label noise, the noisy dataset is represented as $D^\eta = \{(\mathbf{x}_i, \tilde{y}_{\mathbf{x}_i})\}_{i=1}^n$ sampled *i.i.d.* from the noisy distribution $\mathcal{D}^\eta$, where $\tilde{y}_{\mathbf{x}}$ is the noisy label. A noise model could capture the

relationship between $\mathcal{D}$ and $\mathcal{D}^\eta$ by

$$\eta_{\mathbf{x},jk} = Pr(\tilde{y}_\mathbf{x} = k | y_\mathbf{x} = j, \mathbf{x}); \sum_k \eta_{\mathbf{x},jk} = 1, \ \forall j, \mathbf{x}.$$

The problem of robust learning of classifiers under label noise can be informally summed up as follows. We get noisy data drawn from $\mathcal{D}^\eta$ and use it to learn a classifier; however, the learned classifier has to perform well on clean data drawn according to $\mathcal{D}$.

One can consider different label noise models based on what we assume regarding $\eta_{\mathbf{x},jk}$ [20, 22, 24, 21]. In this paper, we consider only symmetric noise and hierarchical (class conditional) noise. If $\eta_{\mathbf{x},jk} = 1 - \eta$ for $j = k$, $\eta_{jk} = \frac{\eta}{c-1}$ for $j \neq k$, then the noise is said to be symmetric or uniform. If $\eta_{\mathbf{x},jk}$ is a function of (j, k) and independent on $\mathbf{x}$, then it is called class conditional noise. We consider a particular case where the set of class labels can be partitioned into some subsets, and label noise is symmetric within each subset. We call this hierarchical noise. This is more realistic because, for example, when the labels are obtained through crowdsourcing, it is likely that different breeds of dogs may be confused with each other, although a dog may never be mislabeled as a car.

Here we define the robustness of risk minimization algorithms [24]. Given a classifier f, its risk under loss function $\mathcal{L}$ is defined as $R_\mathcal{L} = \mathbb{E}_\mathcal{D}[\mathcal{L}(f(\mathbf{x}), y_\mathbf{x})]$ and f^* denotes the minimizer of $R_\mathcal{L}(\cdot)$. This is often referred to as L-risk to distinguish it from the usual Bayes risk, but we will call it risk here. Similarly, under noisy distribution the risk of f is given by $R^\eta_\mathcal{L}(f) = \mathbb{E}_{\mathcal{D}^\eta}[\mathcal{L}(f(\mathbf{x}), \tilde{y}_\mathbf{x})]$ and the corresponding minimizer of $R^\eta_\mathcal{L}(\cdot)$ is f^*_η. We say the loss function $\mathcal{L}$ is noise-tolerant or robust if

$$Pr_\mathcal{D}[Pred \circ f^*(\mathbf{x}) = y_\mathbf{x}] = Pr_\mathcal{D}[Pred \circ f^*_\eta(\mathbf{x}) = y_\mathbf{x}],$$

where $Pred \circ f(x)$ denotes the decision on classification scores $f(x)$ and $Pr_\mathcal{D}$ denotes probability under the clean data distribution. Essentially, the above equation indicates that the classifiers learned with clean and noisy data both have the same generalization error under the noise-free distribution.

Robustness of risk minimization, as defined above, depends on the specific loss function employed. It has been proved that symmetric loss functions are robust to the symmetric noise [21,

23]. A loss function $\mathcal{L}$ is symmetric if it satisfies Equation2.1 [21].

$$\sum_{j} \mathcal{L}(f(\mathbf{x}), j) = C, \ \forall \mathbf{x}, f. \tag{2.1}$$

That is, for any example $\mathbf{x}$ and classifier f, the loss summation over all classes will be equal to a constant C. However, the above robustness is defined for finding the minimizer of true risk. One can show that the consistency of empirical risk minimization holds under symmetric noise [21]. Hence, given a sufficient number of examples, empirical risk minimization also would be robust if we use a symmetric loss function.

Robustness of NAS. Our focus is on NAS. Normally in learning a neural network classifier, one learns only the weights with the architecture chosen beforehand. However, in the context of NAS, one needs to learn both architecture and the weights. Let us denote now by f the architecture and by θ the weights of the architecture. Then, the risk minimization can involve two different loss functions as below.

$$\begin{aligned} f^* &= \arg\min_{f \in \mathcal{F}} \mathbb{E}_{D_{val}}[\mathcal{L}_1(f(\mathbf{x}; \boldsymbol{\theta}^*), y_{\mathbf{x}})], \\ \boldsymbol{\theta}^* &= \arg\min_{\boldsymbol{\theta}} \mathbb{E}_{D_{train}}[\mathcal{L}_2(f(\mathbf{x}; \boldsymbol{\theta}), y_{\mathbf{x}})]. \end{aligned} \tag{2.2}$$

We employ the loss $\mathcal{L}_1$ for learning architecture while we use $\mathcal{L}_2$ for learning weights of any specific architecture. Notice from the above that we use the training data to learn the appropriate weights for any given architecture while we use the validation data for learning the best architecture.

The corresponding quantities under the noisy distribution would be

$$\begin{aligned} f_\eta^* &= \arg\min_{f_\eta \in \mathcal{F}} \mathbb{E}_{D_{val}^\eta}[\mathcal{L}_1(f_\eta(\mathbf{x}; \boldsymbol{\theta}_\eta^*), \tilde{y}_{\mathbf{x}})], \\ \boldsymbol{\theta}_\eta^* &= \arg\min_{\boldsymbol{\theta}_\eta} \mathbb{E}_{D_{train}^\eta}[\mathcal{L}_2(f_\eta(\mathbf{x}; \boldsymbol{\theta}_\eta), \tilde{y}_{\mathbf{x}})]. \end{aligned} \tag{2.3}$$

For the robustness of NAS, as earlier, we want the final performance to be unaffected by whether or not there is label noise. Thus, we still need that the test error, under noise-free distribution, of f^* and f_η^* be the same. However, there are some crucial issues to be noted here.

The parameters θ of each f in the search space can be optimized by the empirical risk of $\mathcal{L}_2$ with D_{train}, and then the best-optimized f is selected by the empirical risk of $\mathcal{L}_1$ with D_{val}. Thus, in NAS, label noise in training data and validation data may have different effects on the final learned classifier. Also, during the architecture search phase, each architecture is trained only for a few epochs, and then we compare the risks of different architectures. Hence, in addition to having the same minimizers of risk under noisy and noise-free distributions, relative risks of any two different classifiers should remain the same irrespective of the label noise.

In NAS, the most common choice for $\mathcal{L}_1$ is 0–1 loss (i.e., accuracy), while for $\mathcal{L}_2$ is categorical cross entropy (CCE). Suppose $\mathbf{p}$ represents the output of the softmax layer and let the class label of an example be t. The CCE is defined by $\mathcal{L}(\mathbf{p}, t) = -\log(p_t)$. 0–1 loss is known as symmetric and hence is robust. However, CCE is not symmetric because it does not satisfy Equation 2.1 (CCE is not bounded). Intuitively, we can mitigate the adverse effects of symmetric noise on NAS by replacing $\mathcal{L}_2$ with any symmetric loss function. Robust log loss (RLL) [25] is a modification of CCE.

$$\mathcal{L}(\mathbf{p}, t) = \log(\frac{\alpha+1}{\alpha}) - \log(\alpha + p_t) + \sum_{j=1, j\neq t}^{c} \frac{1}{c-1} \log(\alpha + p_j)$$

where $\alpha > 0$ is a hyper-parameter and c denotes the number of all classes. It satisfies the symmetry condition (Equation 2.1) and compares (in log scale) probability score of desired output with the average probability score of all other labels. In contrast, the CCE loss only looks at the probability score of the desired output. Another symmetric loss is mean absolute error (MAE) defined by $\mathcal{L}(\mathbf{p}, t) = \sum_{j=1}^{c} |y_j - p_j|$. Since MAE takes longer training time to coverage [23], we make use of RLL in place of CCE in NAS. For other symmetric loss functions [26], we leave them for future work.

2.3 Theoretical Result

As discussed earlier, we want a loss function that ensures that the relative risks of two different classifiers remain the same with and without label noise. Here we prove this for symmetric loss functions.

Theorem 1. *Let $\mathcal{L}$ be a symmetric loss function, $\mathcal{D}$ be a noise-free distribution, and $\mathcal{D}^\eta$ be a noisy distribution with symmetric noise $\eta < \frac{c-1}{c}$, where c is the number of total classes. The risk of f over $\mathcal{D}$ is $R_\mathcal{L}(f)$, and over $\mathcal{D}^\eta$ is $R^\eta_\mathcal{L}(f)$. Then, given any two classifiers f_1 and f_2, if $R_\mathcal{L}(f_1) < R_\mathcal{L}(f_2)$, $R^\eta_\mathcal{L}(f_1) < R^\eta_\mathcal{L}(f_2)$ and vice versa.*

Proof 1. Though this result is not explicitly available in the literature, it follows easily from the proof of Theorem 1 in [21]. For completeness, we present the proof here. For symmetric label noise, we have *

$$\begin{aligned}
R^\eta_\mathcal{L}(f) &= \mathbb{E}_{\mathbf{x},\tilde{y}_\mathbf{x}}\mathcal{L}(f(\mathbf{x}),\tilde{y}_\mathbf{x}) \\
&= \mathbb{E}_\mathbf{x}\mathbb{E}_{y_\mathbf{x}|\mathbf{x}}\mathbb{E}_{\tilde{y}_\mathbf{x}|\mathbf{x},y_\mathbf{x}}\mathcal{L}(f(\mathbf{x}),\tilde{y}_\mathbf{x}) \\
&= \mathbb{E}_\mathbf{x}\mathbb{E}_{y_\mathbf{x}|\mathbf{x}}\left[(1-\eta)\mathcal{L}(f(\mathbf{x}),y_\mathbf{x}) + \frac{\eta}{c-1}\sum_{j\neq y_\mathbf{x}}^{c}\mathcal{L}(f(\mathbf{x}),j)\right] \\
&= (1-\eta)R_\mathcal{L}(f) + \frac{\eta}{c-1}(C - R_\mathcal{L}(f)) \\
&= \frac{\eta C}{c-1} + \left(1 - \frac{\eta c}{c-1}\right)R_\mathcal{L}(f).
\end{aligned}$$

Note that C is the constant in the symmetry condition (Equation 2.1), and c signifies the number of all classes. For the third equality, we are calculating expectation of a function of $\tilde{y}_\mathbf{x}$ conditioned on $y_\mathbf{x}$ and $\mathbf{x}$, where random variable $\tilde{y}_\mathbf{x}$ takes $y_\mathbf{x}$ with probability $1-\eta$ and takes all other labels with equal probability. Thus, $R^\eta_\mathcal{L}(f)$ is a linear function of $R_\mathcal{L}(f)$. Also, since $\eta < \frac{c-1}{c}$, we have $(1 - \frac{\eta c}{c-1}) > 0$. Hence, the above shows that $R_\mathcal{L}(f_1) < R_\mathcal{L}(f_2)$ implies $R^\eta_\mathcal{L}(f_1) < R^\eta_\mathcal{L}(f_2)$ and vice versa. This completes the proof.

Remark 1. Theorem 1 shows that under symmetric loss function, the risk ranking of different neural networks remains the same regardless of noisy or clean data. Since 0–1 loss is symmetric, 0–1 loss as $\mathcal{L}_1$ in NAS could keep the risk ranking of different neural networks consistent. It indicates that we could discover the same optimal network architecture from noisy validation data as the one from clean validation data theoretically. Besides, f^* is proved as the global minimizer for both

*Note that expectation of clean data is under the joint distribution of $\mathbf{x}, y_\mathbf{x}$ while that of noise data is under the joint distribution of $\mathbf{x}, \tilde{y}_\mathbf{x}$

$R_{\mathcal{L}}(f)$ and $R^{\eta}_{\mathcal{L}}(f)$ if $\mathcal{L}$ is symmetric [21]. When we adopt a symmetric loss in $\mathcal{L}_2$, we can obtain $\boldsymbol{\theta}^* = \boldsymbol{\theta}^*_{\eta}$. With the above two conditions, as long as $\eta < \frac{c-1}{c}$, $\mathcal{L}_1$ is 0-1 loss, and $\mathcal{L}_2$ is symmetric loss, a NAS would be robust to symmetric label noise.

Remark 2. Theorem 1 demonstrates that the rank consistency for true risk under noisy and noise-free data. The theorem [21, Thm.4] points out that the minimization of empirical risk converges uniformly to that of the true risk. With the aid of the theorem, the linear relationship in Theorem 1 would be right as well for empirical risk. This implies that under symmetric loss function, the relative ranking of classifiers for empirical risk (with sufficient samples) would be the same as the true risk under noisy and noise-free data. However, the sample complexity would be higher under noisy labels.

2.4 Experiments

To explore how label noise affects NAS and examine the ranking consistency of symmetric loss functions we designed noisy label settings on CIFAR [18] benchmarks using DARTS [27] and ENAS [28].

2.4.1 Dataset and Settings

Dataset. The CIFAR-10 and CIFAR-100 [18] consist of 32×32 color images with 10 and 100 classes, respectively. Each dataset is split into $45\,000$, $5\,000$, and $10\,000$ as training, validation, and testing sets, following AutoKeras [29]. All the subsets are preprocessed by per-pixel mean subtraction, random horizontal flip, and 32×32 random crops after padding with 4 pixels. We corrupt the training and validation labels by noise and always keep testing labels clean, which is common in literature [21, 23]. The validation set is used to pick up the best neural architecture during searching and decide the best training epoch during final retraining. Note that the test set is only considered to report the performance.

Noise Construction. We provide theoretical guarantee to the performance of RLL under symmetric noise. Meanwhile, to better illustrate/demonstrate/understand the effectiveness of RLL, we evaluate RLL under both symmetric noisy and hierarchical noise.

- Symmetric noise [25]: There is an equal chance that one class is corrupted to be another class. This chance can be captured by a matrix $P_\eta = \eta B + (1-\eta)I$, whose element in the i-th row and j-th column is the probability of the true label i being changed into label j. To be specific, I is the identity matrix; all elements of the matrix B are $\frac{1}{1-c}$ except that diagonal values are zero, and η is the adjustable noise level. We inject the symmetric noise in CIFAR-10 with η of $[0.2, 0.4, 0.6]$.

- Hierarchical noise [30]: All label classes can uniformly turn to any other label classes that belong to the same "superclass." For instance, the "baby" class is allowed to flip to the different 4 categories (e.g., boy and girl) in the "people" superclass rather than "bed" or "bear". Since CIFAR-100 inherently provides the superclass information, we add the hierarchical noise into CIFRA-100 with noise level η of $[0.2, 0.4, 0.6]$.

NAS Algorithms. In order to investigate the noisy label problem in NAS, we select representative NAS methods, including DARTS [27] and ENAS [28]. The empirical results on AutoKeras [29] could be found in the supplementary material as well.

- DARTS searches neural architectures by gradient descent. It assigns different network operations by numeric architectural weights and uses Hessian gradient descent jointly optimize weights of neural networks and architectural weights. The experiment setting of DARTS could be found in Section 1 of the supplementary material.

- ENAS discovers neural architectures by reinforcement learning. Although its RNN controller still samples potential network operations by REINFORCE rule [31], ENAS could share the weights of network operations between different search iteration. The experiment setting of ENAS could be found in Section 2 of the supplementary material.

2.4.2 The impact of label noise on the performance of NAS

To demonstrate how erroneous labels affect the performance of NAS, we intentionally introduce symmetric noise ($\eta = 0.6$) in training labels, validation labels, or both (all noisy). Different NAS

	DARTS [27]				ENAS [28]			
	All Clean	Noisy Valid	Noisy Train	All Noisy	All Clean	Noisy Valid	Noisy Train	All Noisy
Clean CCE Retrain	96.98	96.22	95.42	96.69	95.84	96.13	95.84	95.88
Noisy CCE Retrain	81.01	78.76	81.35	81.62	79.33	80.46	78.61	80.34
Noisy RLL Retrain	85.63	84.85	87.11	87.53	79.38	80.07	79.22	79.80

Table 2.1: NAS on CIFRA-10 with symmetric noise ($\eta = 0.6$). The test accuracy is shown in percentage. Noisy train or noisy valid corrupts training or validation labels, while all noisy pollutes both training and validation labels. NAS algorithms search architectures by CCE under the above settings and retrain the searched architectures by CCE or RLL ($\alpha = 0.01$). Reprinted with permission from [32].

methods execute under clean labels (all clean) and these three noisy settings. We evaluate each searcher by measuring the testing accuracy of its best-discovered architecture. Searched networks are retrained with clean labels or polluted labels, denoted as "all clean" and "all noisy," respectively. The former one shows how noise in the search phase affects the performance of the standard NAS. The latter one reflects how noise alters the search quality of NAS in practical situations. Furthermore, since test accuracy evaluates the search quality, we also include RLL to reduce the noise effect in the retraining phase.

The main results are shown in Table 2.1. In the clean retraining setting, the optimal network architectures from DARTS and ENAS with noisy labels could result in comparable performance to the ones searched with clean labels. One possible reason is that both DARTS and ENAS adopt the cell search space, which is limited. As long as the networks can be fully retrained by clean labels, they can achieve similar performance. The architectural variance resulting from label noise does not lead to noticeable performance differences. The observation had also been pointed out in [33].

When it comes to retraining the networks with noisy labels, their accuracy drops significantly. The performance differences come from the classical issue of label noise to deep neural networks [23]. With the help of RLL, we can perceive that the architectures searched by DARTS could achieve better performance, while ENAS does not. Another important observation for ENAS is that the performance under four search settings is comparable. One reason is that the 0-1 loss in ENAS

could provide certain robustness to noisy validation labels, which counteracts the negative effect of symmetric noise. Since the search quality of ENAS seems robust to symmetric noise, we do not explore ENAS further in the following experiments.

When we focus on the noisy retraining of DARTS, the performance of "noisy valid" is the lowest one among others. The decrease of search quality is partially because the $\mathcal{L}_1$ of DARTS is CCE, which is not robust to symmetric loss. DARTS may not be able to rank the performance of different architectures correctly in the setting. The inferior performance from noisy validation labels in other machine learning models had also been proposed in [34]. Moreover, the "all noisy" searcher is supposed to produce the worst test accuracy since it has both noisy training and validation labels. Surprisingly, the empirical results show that "all noisy" in DARTS even outperforms "all clean." A possible conjecture is that the "all noisy" searcher is optimized under the same retraining setting, and the resulting network is intentionally designed to adapt to noisy labels. The finding is worthy of conducting further explorations in the future, such as adopting NAS to discover more robust neural architectures. Despite that, we could still find that label noise in the search phase could generally lead to a negative influence on NAS performance. Especially, DARTS suffers more from noisy validation labels.

2.4.3 Noise Influence of the Risk Ranking

Since NAS aims to find the architectures that outperform others, obtaining a correct performance ranking among different neural networks plays a crucial role in NAS. As long as NAS can recognize the correct performance ranking during the search phase, it should have a high chance to recommend the best neural architecture finally. Theorem 1 reveals that symmetric loss functions have such desired property under symmetric noise situation. To evaluate the practical effects of the theorem, we construct two different neural networks (Table 2.2) through randomly choosing the network operations as well as the locations of the skip connection. Each network has 8 layers with 36 initial channels. We also exclude the auxiliary layer to avoid its additional loss.

We train the networks for 350 epochs under clean and noisy training labels, to which symmetric noise of $\eta = 0.6$ is injected. Proof 1 of Section 2.3 shows that the noisy true risk is of positive

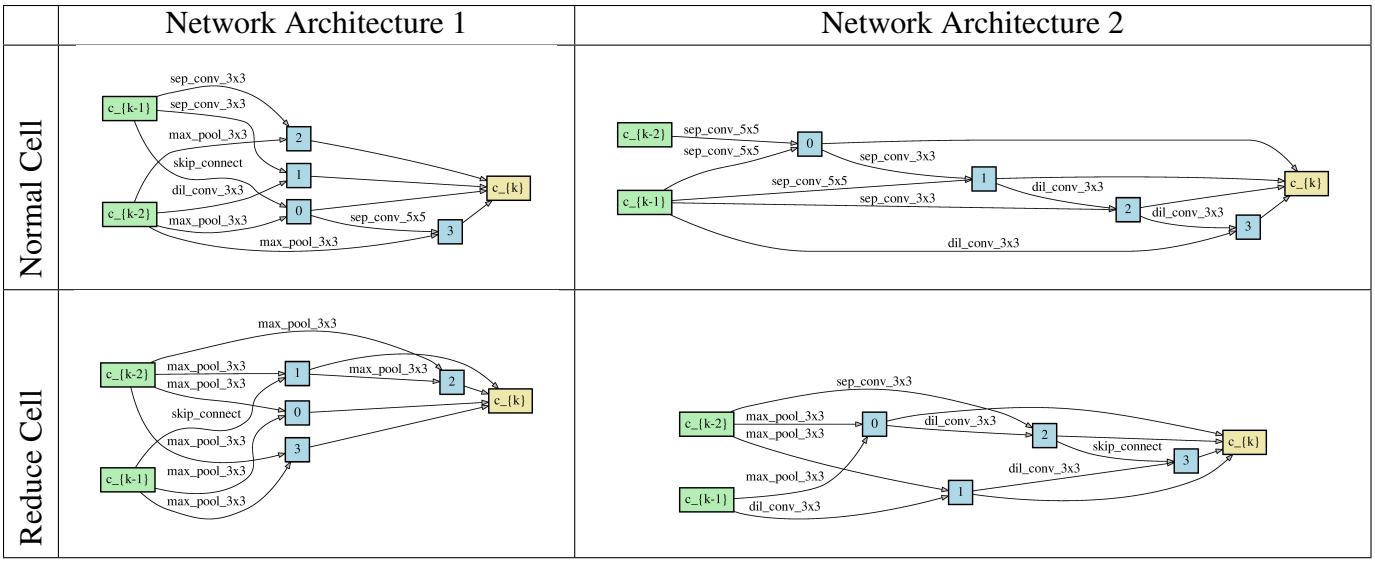


Table 2.2: Two neural network architectures for the ranking of empirical risk. Reprinted with permission from [32].

correlation with the clean true risk. Although we do not have the true risk, when the empirical risk of a loss function could conform to the relationship, the loss is supposed to satisfy Theorem 1 likely. Thereby, we inspect the closeness between the empirical noisy risk and its ideal risk, which is computed by the linear function of Proof 1 with the empirical clean risk. To be specific, the Pearson correlation coefficient (PCC) is used to measure the degree of closeness. ($0 < PCC \leqslant 1$ indicates the positive correlation.)

Figure 2.1 displays the RLL and CCE training loss of the first network under noise-free and noisy labels. The symmetric noise of $\eta = 0.6$ is introduced in training labels. The curves of empirical risk (A1 clean and A1 noisy) are from training the network by CCE or RLL (α = 0.01). After we obtained the curve of the empirical clean risk, we drew the ideal curve for the noisy risk according to Proof 1 of Section 2.3. The expectation is that the curve of noisy risk in RLL should be close to the ideal curve, while CCE does not. As we can notice, the curves of noisy risk in CCE deviate from the ideal curves. In contrast, the two curves of noisy risk in RLL stays closer to the ideal curves than CCE. Moreover, the PCC of RLL displays a positive correlation ($PCC > 0$), which also supports that the empirical risk of RLL is highly close to the ideal one. The reasons that

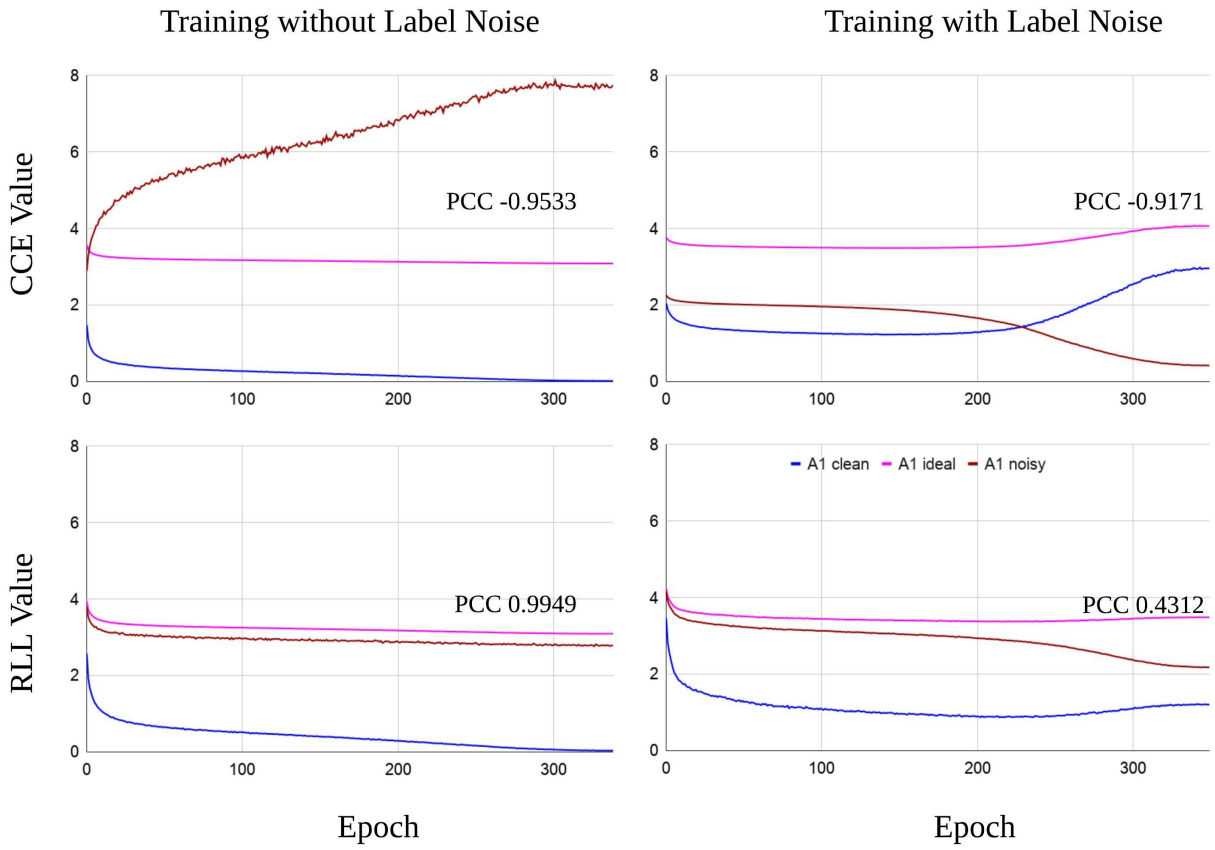


Figure 2.1: The empirical risk of the neural network. Reprinted with permission from [32].

empirical noisy risks do not perfectly match the ideal one include: (1) training samples (examples) are not enough, (2) hyper-parameters are not optimal for learning the networks, Therefore, we could understand that symmetric loss functions have the capability to make the risk ranking under noisy labels uniform to the one under clean labels in practice.

2.4.4 NAS Improvement with Symmetric Loss Function

In practice, the resulting networks from NAS are trained on the potentially wrong labels. We want to see whether NAS could still discover high-performance networks in this harsh environment with the help of symmetric loss function, especially robust log loss (RLL). The performance of neural networks decreases by label noise, but the symmetric loss can alleviate the adverse influence, as shown in [25]. Thus, in the experiment, no matter DARTS searches networks by CCE or RLL,

	Symmetric noise (CIFAR-10)			Hierarchical noise (CIFAR-100)		
Noise level η	0.2	0.4	0.6	0.2	0.4	0.6
ResNet-18	92.05 ± 0.40	88.95 ± 0.14	82.77 ± 0.61	61.27 ± 0.60	53.50 ± 0.94	39.99 ± 2.17
DARTS	**94.91** ± 0.19	**91.02** ± 0.78	83.31 ± 2.88	**67.82** ± 0.70	52.57 ± 1.03	39.22 ± 2.50
DARTS-RLL	94.66 ± 0.67	90.77 ± 1.56	**86.24** ± 0.85	66.47 ± 1.68	**53.68** ± 1.96	**46.41** ± 2.65

Table 2.3: NAS with RLL. Test accuracy and standard deviation (3 runs) are represented in percentage. DARTS searches architectures with CCE or RLL (α = 0.01), and then the resulting optimal neural network is trained again from scratch by RLL (α = 0.01). Noise contaminates both training and validation labels with different noise levels. Bold font exhibits the best result in each column. Reprinted with permission from [32].

we leverage RLL in the final retrain phase. Apart from DARTS, Resnet-18 [35] is also included in the experiment for performance comparison. Moreover, we are interested in how NAS with RLL works in another type of label noise. Here we also report the results beyond the hierarchical noise of CIFAR-100.

The results presented in Table 2.3 point out that RLL can still help NAS discover high-performance network architectures under high noise levels. No matter in symmetric or hierarchical noise, DARTS with RLL reaches a similar accuracy to DARTS with CCE under $\eta = 0.2$ and 0.4, and RLL one outperforms CCE under $\eta = 0.6$. One possible reason is that DARTS is robust to mild noise due to its small search space. Nevertheless, severe noise introduces intense uncertainty for DARTS. RLL can help DARTS to determine relatively robust neural architectures in the harsh condition. From the empirical results, we can claim that the symmetric (robust) loss function, RLL, improves the search quality under high-level label noise.

2.5 Related Work

Neural architecture search (NAS) is purposed to facilitate the design of network architectures automatically. Currently, the mainstream approaches to achieve NAS includes Bayesian optimization [29, 36], reinforcement learning [37, 38, 28, 39], evolutionary algorithms [40, 41] and gradient-based optimization [27, 42, 43]. Regardless of the different approaches, NAS consists of two phases: the search phase and the final-retrain phase. During the search phase, NAS generates

and evaluates a variety of different intermediate network architectures repeatedly. Those networks are trained on the training set for a short time (e.g., tens of epochs). Their performance, measured on the validation set, is used as a guideline to discover better network architectures. In the final-retrain phase, the optimal network architecture will be trained with additional regularization techniques, e.g., Shake-Shake [44], DropPath [45], and Cutout [46]. The phase usually takes hundreds of epochs. And then the trained network is evaluated on the unseen test set. In general, the two phases utilize the same training set.

From the perspective of the search space of network architectures, current existing works could be divided into the complete architecture search space [29, 36, 37, 40] and the cell search space [38, 28, 39, 41, 27, 42, 43]. The first search space allows NAS to look for complete networks and provides a high diversity of resulting network architectures. The second one limits NAS to seek the small architectures for two kinds of cells (normal cell and reduction cell). And it is also required to pre-defined the base network architecture to contain the searched cells for evaluation, which implies that many intermediate networks will share similar network architecture. Most existing works usually develop from the cell search space because the size of this search space is significantly smaller than the complete one, and can reduce the enormous search time.

Due to the limited hardware resources, our experiments focus on cell search space, including DARTS [27] and ENAS [28]. We also explore the label noise impact on AutoKeras [29]. Notice that no similar works have studied the effect of label noise on NAS until we publish the work.

Great progress has been made in research on the robustness of learning algorithms under corrupted labels [21, 22, 23, 47, 48, 49, 50, 51, 52]. A comprehensive overview of previous studies in this area can be found in [20]. The proposed approaches for learning under label noise can generally be categorized into a few groups.

The first group comprises mostly label-cleansing methods that aim to correct mislabeled data [53], or adjust the sampling weights of unreliable training instances [50, 51, 52, 54, 55]. Another group of approaches treats the true but unknown labels as latent variables and the noisy labels as observed variables so that EM-like algorithms can be used to learn the true label distri-

bution of the dataset [56, 57, 58]. The third broad group of approaches aims to learn directly from noisy labels under the generic risk minimization framework and focus on noise-robust algorithms [59, 24, 22, 21, 23]. There are two general approaches here. One can construct a new loss function using estimated noise distributions, while the others develop conditions on loss functions so that risk minimization is inherently robust. In either case, they can derive some theoretical guarantees on the robustness of classifier learning algorithms.

All the above approaches are for learning parameters of specific classifiers using data with label noise. In NAS, we need to learn a suitable architecture for the neural network in addition to learning of the weights. Our work differs from the above studies that we discuss the robustness in NAS under corrupted labels, while most of the above works focus on the robustness of training in supervised learning. We investigate the effect of label noise in NAS at multiple levels.

2.6 Conclusion

Neural architecture search is gaining more and more attention in recent years due to its flexibility and the remarkable power of reducing the burden of neural network design. The pervasive existence of label noise in real-world datasets motivates us to investigate the problem of neural architecture search under label noise. Through both theoretical and experimental analyses, we studied the robustness of NAS under label noise. We showed that symmetric label noise adversely the search ability of DARTS, while ENAS is robust to the noise. We further demonstrated the benefits of employing a specific robust loss function in search algorithms. These conclusions provide a strong argument in favor of adopting the symmetric (robust) loss function to guard against high-level label noise. In the future, we could explore that the factors cause DARTS to have superior performance under noisy training and validation labels. We could also investigate other symmetric loss functions for NAS.

3. EFFICIENT DIFFERENTIABLE NEURAL ARCHITECTURE SEARCH WITH MODEL PARALLELISM

Differentiable neural architecture search (NAS) with supernets that encompass all potential architectures in a large graph cuts down search overhead to few GPU days or less. However, these algorithms consume massive GPU memory, which will restrain NAS from large batch sizes and large search spaces (e.g., more candidate operations, diverse cell structures, and large depth of supernets). In this chapter, we present binary neural architecture search (NASB) with consecutive model parallel (CMP) to tackle the problem of insufficient GPU memory. CMP aggregates memory from multiple GPUs for supernets. It divides forward/backward phases into several sub-tasks and executes the same type of sub-tasks together to reduce waiting cycles. NASB is proposed to reduce memory footprint, which excludes inactive operations from computation graphs and computes those operations on the fly for inactive architectural gradients in backward phases. Experiments show that NASB-CMP runs 1.2$\times$ faster than other model parallel approaches and outperforms state-of-the-art differentiable NAS.

3.1 Introduction

Neural architecture search (NAS) has revolutionized architecture designs of deep learning from manually to automatically in various applications, such as image classification [37] and semantic segmentation [60]. Reinforcement learning [37, 61, 28], evolutionary algorithms [62, 63], and differentiable algorithms [27, 42] have been applied to discover the optimal architecture from a large search space of candidate network structures. Supernets [61, 28] comprising all possible networks reduce search spaces from complete network architectures to cell structures. Recent acceleration techniques of differentiable NAS [64, 65, 66, 67] further diminish search costs to affordable computation overheads (e.g., half GPU day). Prior work [67] randomly samples partial channels of intermediate feature maps in the mixed operations.

However, supernets of differentiable NAS consume gigantic GPU memory, which constrains

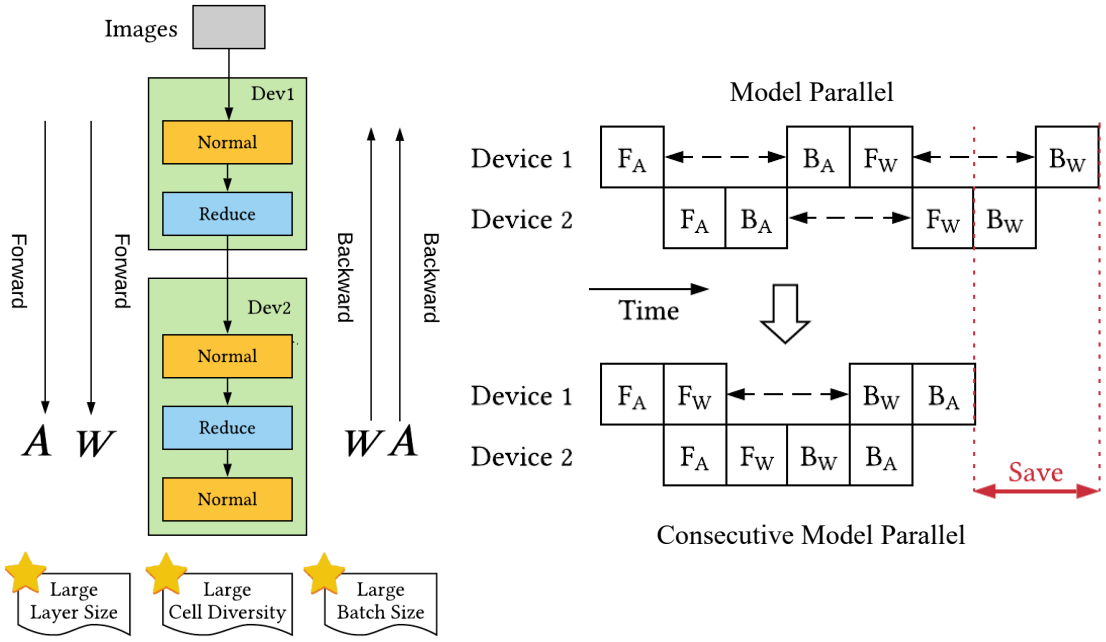


Figure 3.1: Consecutive model parallel (CMP) overlaps the two forward sub-tasks ($F_{\boldsymbol{A}}$ and $F_{\boldsymbol{W}}$) and two backward sub-tasks ($B_{\boldsymbol{W}}$ and $B_{\boldsymbol{A}}$). This new execution order empowers neural architecture search (NAS) to search faster than using model parallel (MP). The right figure shows that CMP can save two cycles from vanilla MP. Furthermore, CMP inherits MP's advantages, like using large batch sizes in the supernet, enlarging layer numbers of the supernet, and even diversifying cell architecture across different layers.

NAS from using large batch sizes and imposes restrictions on supernet architectures' complexity. For example, NAS determines networks in shallow supernets (e.g., 8 layers) for deep compact networks (e.g., 20 layers). The cell structures are also required to remain identical for the same type of cells. Data parallelism can increase the search efficiency of NAS by using large batch sizes, such as SNAS [64], but it requires supernet complexity low enough to fit in a single GPU. In contrast, model parallelism can parallelize complex supernets, which distributes partial models to multiple devices. Nevertheless, model parallelism suffers from low hardware utilization. Only one device executes its model partition, while other devices stay idle. How to take advantage of multiple GPUs for large supernets efficiently is an open problem.

We propose a simple but efficient solution, binary neural architecture search (NASB) using consecutive model parallel (CMP), to tackle the above limitations. Specifically, supernets have two

forward and two backward phases to learn architecture parameters and network weights. CMP distributes several sub-tasks split from the four phases in multiple GPUs and executes the sub-tasks of all forward/backward phases together. Figure 3.1 illustrates that sub-tasks of forward/backward phases will be overlapped to reduce waiting cycles. Nevertheless, CMP consumes large GPU memory due to two computation graphs existing at the same time. Thus, we introduce NASB to declines GPU memory occupation. NASB utilizes binary and sparse architecture parameters (1 or 0) for mixed operations. It excludes inactive operations in the computation graph and computes feature maps of inactive operations for architecture gradients during the back-propagation. In this way, NASB-CMP can increase hardware utilization of model parallelism with efficient GPU memory in differentiable NAS.

In our experiments on CIFAR-10, NASB-CMP runs 1.2× faster than using model parallel and pipeline parallel, TorchGPipe [68] in a server with 4 GPUs *. It can achieve the test error of 2.53 ± 0.06% by searching for only 1.48 hours. Our contribution can be summarized as follows:

- NASB-CMP is the first NAS algorithm that can parallelize large supernets with large batch sizes. We analyze the acceleration ratio between CMP and traditional model parallelism. Even though complex supernets (e.g., large layers and different cell structures) will not boost NAS performance, NASB-CMP paves the way to explore the supernet architecture design in the future.

- NASB utilizes binary architecture parameters and extra architecture gradients computation to reduce GPU usage. It can save memory consumption by accepting twice batch sizes larger than the other memory saving algorithm, PC-DARTS [67].

- We fairly compare NASB-CMP with state-of-the-art differentiable NAS in the same hardware and search space. Extensive experiments show that NASB-CMP can achieve competitive test error in short search time.

*NVIDIA GTX 1080 Ti.

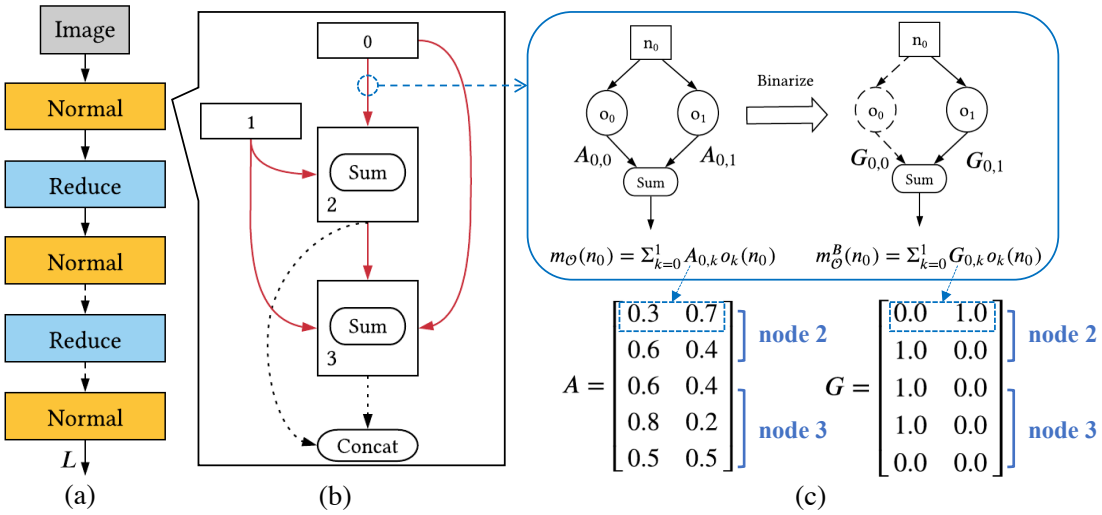


Figure 3.2: Illustration of Binary Neural Architecture Search (NASB). (a) is a supernet made up of normal and reduce cells. (b) portrays the directed-acyclic-graph (DAG) used for cell structures. (c) embodies the mixed operation in the solid red lines of the middle figure. NASB builds its supernet with binary mixed operations $m_{\mathcal{O}}^{B}$, which replace architectural matrix $\boldsymbol{A}$ with binary matrix $\boldsymbol{G}$. The symbol n and o stand for nodes in DAG and candidate operations. Among rows associated with a node (blue bracket) in $\boldsymbol{A}$, the largest two values are set to 1 and the rest elements to 0. Only partial operations are active during the search procedure of NASB.

3.2 Methodology

We first describe the fundamental concepts of one-shot neural architecture search (NAS) in Section 3.2.1. We then portray the consecutive model parallel to enhance NAS search efficiency in multiple devices in Section 3.2.2. Finally, we explain how we binarize the architectural weights and compute their gradients to cut down the GPU memory consumption in Section 3.2.3.

3.2.1 One-shot Neural Architecture Search

One-shot neural NAS [61] is built on a supernet (a.k.a. meta graph) in which we stack normal cells and reduce cells sequentially in Figure 3.2 (a). Normal cells are analogous to convolutional layers to extract images features. Reduce cells are equivalent to pooling layers to reduce the spatial dimension of feature maps. All normal cells share the same structure, but each cell still has its network weights. So do all reduce cells. One-shot approaches are required to design two

cell structures instead of complete neural networks. Figure 3.2 (b) illustrates one popular cell structure [28], an N-node directed-acyclic-graph (DAG) with total edges E, not counting the "concat" node. In the h-th cell, the first two nodes are the $(h-2)$-th and $(h-1)$-th cells having no inbound edges. The other nodes accept previous nodes whose index is lower than the current index. Total edges E (red lines of Figure 3.2 (b)) is $(N+1)(N-2)/2$. We denote the h-th cell's output as $y_h = concat(n_j)$, where $2 \leq j \leq N-1$ and n_j is a DAG node signified in Eq. 3.1.

$$n_j = \begin{cases} y_{h-2}, & \text{if } j = 0, \\ y_{h-1}, & j = 1, \\ \sum_{i<j} m_{\mathcal{O}}(n_i), & 2 \leq j \leq N-1. \end{cases} \tag{3.1}$$

A mixed operation $m_{\mathcal{O}}$ is the edge between node i and j in the DAG. Let $\mathcal{O}$ be a set of candidate operations (e.g., convolution, pooling, identity, zero) and $\boldsymbol{A} \in \mathbb{R}^{E \times |\mathcal{O}|}$ be a matrix of architecture parameters. Eq. 3.2 formulates the mixed operation $m_{\mathcal{O}}$ from node i to j as the weighted sum of all operations o_k [27].

$$m^j_{\mathcal{O}}(n_i) = \sum_{k=1}^{|\mathcal{O}|} \boldsymbol{A}_{e,k} o_k(n_i), \; j \geq 2, \; i < j, \tag{3.2}$$

where $e = (j+1)(j-2)/2 + i$ is the edge index. The mixed operations transform the cell structure search to the problem of learning two matrices, $\boldsymbol{A}_N$ and $\boldsymbol{A}_R$, for the normal and reduce cell.

Given that $\mathcal{L}_{val}$ and $\mathcal{L}_{train}$ is the loss function $\mathcal{L}$ beyond a training and validation dataset, respectively. Let $\mathbf{A}$ comprise $\boldsymbol{A}_N$ and $\boldsymbol{A}_R$. Mathematically, one-shot NAS can be formulated in the following optimization problem,

$$\begin{aligned} & min_{\mathbf{A}} \; \mathcal{L}_{val}(w^*, \mathbf{A}) \\ s.t. \; & w^* = \arg\min_{w} \mathcal{L}_{train}(w, \mathbf{A}). \end{aligned} \tag{3.3}$$

NAS leverages the validation performance to choose well-trained networks that outperform others. After training $\mathbf{A}$, we derive the compact network by pruning unused operations in the supernet.

Since the whole chapter follows the image classification setting [27, 42], we assume each node is assigned two inputs and two operations. And we prune node inputs of cells of the supernet by the largest two values of $\mathbf{A}$ associated with that node. For simplicity, we use $\boldsymbol{A}$ in replace of $\mathbf{A}$ in the following discussion.

3.2.2 Consecutive Model Parallel

Data parallelism can scale up supernets with large batch sizes, but it cannot handle large supernets (e.g., deep supernets with different cell structures). Model parallelism (MP) is able to amortize such large supernets across multiple GPUs, but its hardware utilization is low. MP would generate unwanted waiting cycles across devices. Figure 3.1 displays that the first device becomes idle until the second device finishes its forward and backward phases. The parallelization gets worse as we use large available GPUs.

Motivated by pipeline parallelism [69], we propose consecutive model parallel (CMP) to decrease GPU idle time. Let $F_{\boldsymbol{A}}$ and $B_{\boldsymbol{A}}$ signify the forward and backward phase to update $\boldsymbol{A}$, and F_w and B_w be two phases to update w. CMP divides the four phases into several sub-tasks and performs sub-tasks of $F_{\boldsymbol{A}}$ and F_w consecutively, followed by sub-tasks of B_w and $B_{\boldsymbol{A}}$. Figure 3.1 illustrates that the execution order change by CMP overlaps sub-tasks without waiting for others to finish. Given the number of available GPUs M, Eq. 3.4 reveals the ratio of execution time between CMP and MP in theory.

$$\frac{\text{Time of CMP}}{\text{Time of MP}} = \frac{\frac{1}{M}[4M - 2(M-1)]}{4} = 1 - \frac{M-1}{2M}. \tag{3.4}$$

We assume $F_{\boldsymbol{A}}$, $B_{\boldsymbol{A}}$, F_w, and B_w take the same time unit. MP will complete an iteration in 4 units. For CMP, the total sub-tasks is $4M$, and 2$(M-1)$ sub-tasks can be overlapped. If a sub-task takes 1/M ideally, CMP will finish an iteration in $1/M(4M - 2(M-1))$ units. According to Eq. 3.4, CMP with two devices could reduce (2-1)/(2*2)=25% time from MP. In practice, Experiment 3.3.3 demonstrates that NASB-CMP runs 1.2$\times$ faster than model parallelism without sacrificing test error. The theoretical value for 4 GPU is 1.6 (or reduce 37.5% time). We believe communication

Algorithm 1: NASB - Consecutive Model Parallel

Initialize architecture weights $\boldsymbol{A}$ and network weights w
while not stopped **do**
  $\boldsymbol{G}_t = binarize(\boldsymbol{A}_t)$
  Create $m_{\mathcal{O}}^B$ using $\boldsymbol{G}_t$ and Eq. 3.5
  Compute $\mathcal{L}_{valid}(w_t, \boldsymbol{G}_t)$ and $\mathcal{L}_{train}(w_t, \boldsymbol{G}_t)$ consecutively // model parallel
  Compute $\nabla_w \mathcal{L}_{train}(w_t, \boldsymbol{G}_t)$ and $\nabla_{\boldsymbol{A}} \mathcal{L}_{valid}(w_t, \boldsymbol{G}_t)$ consecutively // model parallel
  Update w_{t+1} by descending $\nabla_w \mathcal{L}_{train}(w_t, \boldsymbol{G}_t)$
  Update $\boldsymbol{A}_{t+1}$ by descending $\nabla_{\boldsymbol{A}} \mathcal{L}_{valid}(w_t, \boldsymbol{G}_t)$
end while

overhead and uneven model balance cause the deviation. Communication overhead comes from the intermediate tensors transfer from one to another GPU when models are split into different GPUs. Moreover, the main thread is responsible for loading data and backward propagation. The GPU with the main thread always consumes the most GPU memory, which causes uneven model balance.

CMP is a general model parallel approach for any existing differentiable NAS algorithm. However, running $B_{\boldsymbol{A}}$ and B_w consecutively asks for two computation graphs, which doubles GPU utilization and deteriorates CMP efficiency. To address the problem of great GPU consumption, we introduce a memory-efficient NAS to CMP, called binary neural architecture search (NASB).

3.2.3 Binary Neural Architecture Search

Binary neural architecture search (NASB) harnesses binary mixed operations $m_{\mathcal{O}}^B$ [65] that convert the real-valued $\boldsymbol{A}$ into sparse binary matrix $\boldsymbol{G}$, as illustrated in Figure 3.2. Among rows $\boldsymbol{A}_{e,:}$ associate node j, $m_{\mathcal{O}}^B$ enforces the two largest elements to 1 (active) and the rest elements to 0 (inactive). The row indexes of active elements indicate selected edges to node j, while column indexes indicate chosen operations. Notice that NASB does not directly multiply $\boldsymbol{G}$ with candidate operations in Eq. 3.5. Instead, NASB constructs a set of active operations $\mathcal{O}^{(active)}$ based on active elements in $\boldsymbol{G}$. Only those active operations $o_a \in \mathcal{O}^{(active)}$ are included in the forward phase. This technique could stop inactive operations being stored in the computation graph and decrease roughly

$|\mathcal{O}|$ times GPU memory compared to using the multiplication by $\boldsymbol{G}$.

$$m^B_{\mathcal{O}}(n_i) = \sum_{k=1}^{|\mathcal{O}|} \boldsymbol{G}_{e,k} o_k(n_i) = o_a(n_i). \tag{3.5}$$

NASB computes gradients of network weights w using standard back-propagation in the supernet. For the gradients of $\boldsymbol{A}$, NASB estimates $\partial\mathcal{L}/\partial\boldsymbol{A}$ approximately by $\partial\mathcal{L}/\partial\boldsymbol{G}$:

$$\frac{\partial\mathcal{L}}{\partial\boldsymbol{A}_{e,k}} = \frac{\partial\mathcal{L}}{\partial m_{\mathcal{O}}}\frac{\partial m_{\mathcal{O}}}{\partial\boldsymbol{A}_{e,k}} \approx \frac{\partial\mathcal{L}}{\partial m^B_{\mathcal{O}}}\frac{\partial m^B_{\mathcal{O}}}{\partial\boldsymbol{G}_{e,k}} = \frac{\partial\mathcal{L}}{\partial m^B_{\mathcal{O}}} \times o_k(n) = \frac{\partial\mathcal{L}}{\partial\boldsymbol{G}_{e,k}}. \tag{3.6}$$

Eq. 3.6 states that gradients of elements in $\boldsymbol{A}$ come from $\partial\mathcal{L}/\partial m^B_{\mathcal{O}} \times o_k(n)$. However, inactive operations are not in the computation graph. NASB saves inputs of inactive operations n in PyTorch Context that is used for backward computation. During the backward phase, NASB will compute inactive operations $o_{k'}(n)$ on the fly and multiply the results with the $\partial\mathcal{L}/\partial m^B_{\mathcal{O}}$.

Apart from saving unneeded GPU FLOPS and memory, $m^B_{\mathcal{O}}$ can avoid performance bias between supernets and compact networks. Supernets using $m_{\mathcal{O}}$ assume that the performance of supernets can represent derived compact networks, but non-linear operations (e.g., ReLU-Conv-BN) break the representation that causes performance bias [64]. Instead, the sparse matrix of $m^B_{\mathcal{O}}$ activates one operation. The performance of supernets during the search is only for one compact network. Thus, NASB can mitigate the bias caused by non-linear operations.

Algorithm 1 describes how CMP works with NASB. Note that NASB-CMP does not update any parameter (including $\boldsymbol{A}$ and w) until $F_{\boldsymbol{A}}$, $B_{\boldsymbol{A}}$, F_w, and B_w complete. $\mathcal{L}_{train}$ will use the current binary architecture matrix G_t rather than updated G_{t+1}, which is the major difference from the alternate algorithm (See Alg. 2). Experiment 3.3.4 demonstrates NASB could save substantial GPU memory than PC-DARTS [67], which reduces GPU memory by partial channels of feature maps in mixed operations.

Comparison with other methods. NASP [65] binarizes $\boldsymbol{A}$ based on $\boldsymbol{A}$ itself, while ProxylessNAS [42] binarizes $\boldsymbol{A}$ based on the softmax results of $\boldsymbol{A}$. The two binarization approaches are equivalent, but how they handle binary mixed operations (Eq. 3.5) is different. NASP multiplies $\boldsymbol{G}$

Algorithm 2: NASB

Initialize architecture weights $\boldsymbol{A}$ and network weights w
while not stopped **do**
  $\boldsymbol{G}_t = binarize(\boldsymbol{A}_t)$
  Create $m_{\mathcal{O}}^B$ using $\boldsymbol{G}_t$ and Eq. 3.5
  Compute $\nabla_{\boldsymbol{A}} \mathcal{L}_{valid}(w_t, \boldsymbol{G}_t)$ using Eq. 3.6 // handle the gradients of inactive elements
  Update $\boldsymbol{A}_{t+1}$ by descending $\nabla_{\boldsymbol{A}} \mathcal{L}_{valid}(w_t, \boldsymbol{G}_t)$
  $\boldsymbol{G}_{t+1} = binarize(\boldsymbol{A}_{t+1})$
  Create $m_{\mathcal{O}}^B$ using $\boldsymbol{G}_{t+1}$ and Eq 3.5
  Compute $\nabla_w \mathcal{L}_{train}(w_t, \boldsymbol{G}_{t+1})$ // standard back-propagation
  Update w_{t+1} by descending $\nabla_w \mathcal{L}_{train}(w_t, \boldsymbol{G}_{t+1})$
end while

with all operations (i.e., saving active and inactive operations in the computation graph). Proxyless-NAS selects two sampled operations (paths) in the computation graph according to multinomial distribution. NASB utilizes the same binarization as NASP but only keeps one active operation in the computation graph according to $\boldsymbol{G}$.

3.3 Experiments

We compare NASB-CMP with other parallelisms on the CIFAR-10 in Section 3.3.3. We then inspect the quality of NASB and compare NASB-CMP with state-of-the-art NAS in Section 3.3.4. Finally, we investigate the design of supernet architectures using large layers and different cell structures in Section 3.3.5, which cannot be conducted without saving GPU consumption or model parallel.

3.3.1 Experiment Settings

Our platform is a server with 4 GPUs of NVIDIA GTX 1080 Ti, in which all search experiments are executed. Supernets consist of 8 cells in which the 3rd and 6th cells are reduce cells, and others are normal cells with initial channels 16. The optimizer for network weights w is momentum SGD with moment 0.9, L2 penalty $3e-4$, and cosine anneal learning rate initialized by 0.025 and minimal 0.001. The optimizer for architecture parameter $\boldsymbol{A}$ is Adam with learning rate $3e-4$, L2 penalty $1e-3$, and $(\beta 1, \beta 2) = (0.5, 0.999)$. PC-DARTS with large batch sizes [67] has unique

configurations: initial learning rate 0.1 and minimal 0.0 for SGD optimizer and learning rate $6e-4$ for Adam optimizer.

All NAS algorithms will search networks for 50 epochs with varied batch sizes and random seeds. In Experiment 3.3.3, NABS-CMP is specified search batch size 224, 416, 512, 896 for 1, 2, 3, 4 GPUs, respectively It random seed is 2. In Experiment 3.3.4, The batch size is 60 determined by DARTS because DARTS consumes the largest GPU memory. We want all NAS algorithms to use the same batch size in order to compare each other fairly. Since PC-DARTS is proposed to reduce GPU memory consumption, we also compare the performance of PC-DARTS using a large batch size 224 with NASB and NASB-CMP. NASB is specified with its allowable maximal batch size 448 in a single GPU, and NASB-CMP uses a batch size of 896 in 4 GPUs. All NAS baselines and NASB use 2, 3, 4, 5, 6 as random seeds, and NASP-CMP uses 2, 3, 9, 11, 18 instead. In Experiment 3.3.5, NASB and NASB-CMP exploit batch size 160 and 256, respectively, for 50 epochs. We ran search experiments twice using random seed 2 and 3 and reported the average test error among the two searches in Table 3.3.

The compact networks used in the retrain (evaluation) phase have 20 cells (layers), where the one-third and two-thirds of the depth are reduce cells and others are normal cells. We retrain the compact networks from scratch for 600 epochs with the batch size 96, dropout path of probability 0.2, and initial channels of 36. We also add the auxiliary layer in the network with a loss weight 0.4. During the evaluation phase, the cutout length 16 is additionally applied for image transformation. The optimizer setting for network weights w is the same as the searching setting. The retrain random seed is assigned to 0, which is different from the search seeds.

Dataset. CIFAR-10 [18] is a color-image dataset for image classification, composed of 50,000 training images and 10,000 test images for 10 classes. We preprocess the training images in the following techniques: padding 32×32 images with 4 pixels, and then randomly cropping them back to 32×32; randomly flipping images in the horizontal direction; normalizing image pixels by the channel mean and standard deviation. The processed training set is split evenly: the first half serves as the final training set, and the other serves as the validation set. SNAS merely relies on the

training set to search, so its training set is not split.

Search Space. The DAG (See Section 3.2.1) has $N = 6$ intermediate nodes and $E = 14$ total edges. The set of candidate operations follows NASP [65], where normal operations $|\mathcal{O}_N|$=8 and reduce operations $|\mathcal{O}_R|$=5. Notice that our baselines also use the same operation sets rather than their original one ($|\mathcal{O}_N| = |\mathcal{O}_R| = 8$). Table 3.1 summarizes candidate operations for mixed operations used in NAS papers. Experiment 3.3 makes use of the first row of Table 3.1 as its search space on CIFRAR-10. "skip_connect" symbolizes identity operation if stride size is 1 or ReLU-Conv-Conv-BN operation. "conv", "sep", and "dil_conv" signifies convolution, depthwise-separable convolutions, and dilated depthwise-separable convolutions, respectively. "none" means the zero operation. Note that differentiable NAS baselines (DARTS, SNAS, PC-DARTS) also utilize the first row of Table 3.1 as their search space.

	Normal Cell	Reduce Cell
NASB-CMP NASB NASP [65]	skip_connect (identity) conv_3x1_1x3 dil_conv_3x3 conv_1x1 conv_3x3 sep_3x3 sep_5x5 sep_7x7	skip_connect (identity) avg_pool_3x3 max_pool_3x3 max_pool_5x5 max_pool_7x7
DARTS [27] SNAS [64] PC-DARTS [67]	none (zero) max_pool_3x3 avg_pool_3x3 skip_connect (identity) sep_3x3 sep_5x5 dil_conv_3x3 dil_conv_5x5	

Table 3.1: Candidate operations for normal and reduce cells.

3.3.2 Implementation

The data parallel leverages PyTorch [70] *distributed* module providing communication interfaces to update parameter tensors between multiple processes. Model parallel and CMP are implemented in multi-threading. Each GPU has a specialized thread responsible for its model partition. Those threads enable different model partitions to run simultaneously. Without multi-threading, only assigning model partitions to specific devices do not automatically overlap sub-tasks. For GPipe, we adopt the corresponding PyTorch package, torchgpipe [68], in replace of GPipe, since GPipe is written in Tensorflow. The chunk setting to split mini-batch size to micro-batch size is disabled in the experiment, because enabling the setting increases the search cost.

3.3.3 Parallelism Comparison on CIFAR-10

The performance of NASB-CMP is compared with other parallel approaches on CIFAR-10, including data parallelism, model parallelism, and GPipe [69], the state-of-the-art model parallel that pipelines chunks of data into several model partitions.

Figure 3.3 compares the performance of different parallelizations in NASB in varied GPUs. CMP runs 1.2$\times$ faster than model parallel (MP) and GPipe especially running in 3 and 4 GPUs. According to Eq. 3.4, four GPUs should run 1.6X faster (or reduce 37.5% search time) than MP. In practice, communication overhead and uneven model partitions reduce the ideal speedup ratio. Compared with all parallel approaches, CMP's execution order change does not degrade the test error. Data parallel takes the lowest search cost, but it does not generate as low test error as other model parallel approaches. The reason might be that model replicas in data parallel utilize partial batches to compute architectural gradients, while model parallel can make use of the whole batches. Therefore, CMP is an efficient model parallel approach that helps NAS to utilize large batches.

Despite the competitive performance, the scalability of CMP is inferior. CMP disallows batch sizes from linearly scaling up as large GPUs are involved. For example, 2 GPUs should use 448 (we used 416 instead) if 1 GPU uses 224. Besides, 1-GPU NASB can utilize batch size 448, but NASB-CMP needs 4 GPUs to double batch sizes. The main reason is that CMP keeps two

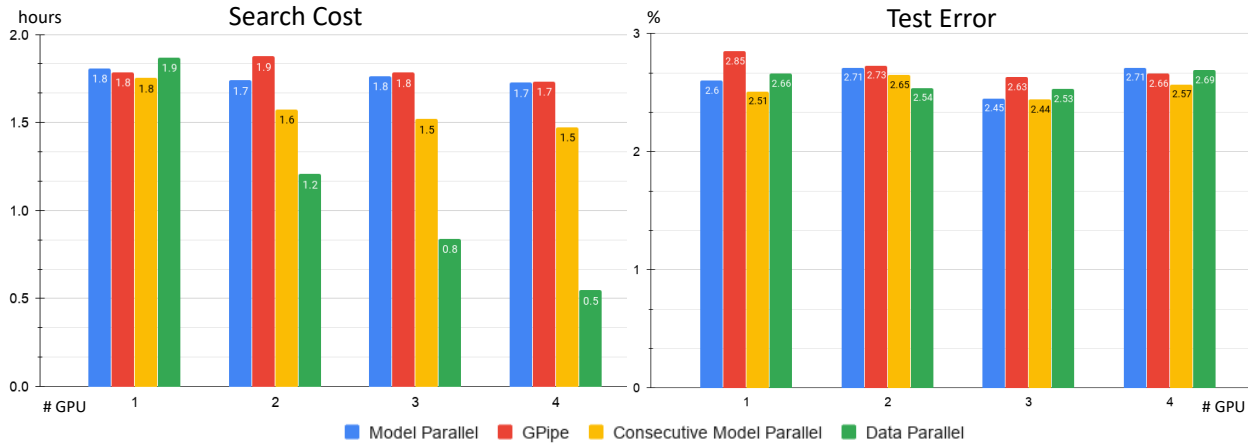


Figure 3.3: Performance comparison between different parallel approaches in NAS. GPipe [69] is an approach of pipeline model parallel. Among the three model parallel approaches (blue, red, yellow), consecutive model parallel (CMP) outperforms them in terms of search cost and test error (lower is better). While the data parallel (green) is the fastest parallel method, but its test errors are not as low as CMP (best viewed in color).

computation graphs (for $B_{\boldsymbol{A}}$ and B_w) simultaneously for overlapping computations, resulting in twice GPU consumption. We believe that a mixed parallel combining CMP and data parallel can mitigate the drawback by merging two advantages, accuracy of CMP and scalability of data parallel.

3.3.4 State-of-the-art NAS Comparison on CIFAR-10

We compare NASB and NASB-CMP with several NAS algorithms on CIFAR-10. DARTS [27], SNAS [64], NASP [65], and PC-DARTS [67] are selected as our baselines. DARTS is the pioneer of differentiable NAS. SNAS points out the performance bias between a supernet and derived networks in DARTS. Both NASP and PC-DARTS reduce GPU memory, which overlaps the scope of this chapter. We should select ProxylessNAS [42] as a baseline, but their search code on CIFAR-10 is not released. We prefer not to ruin their performance with improper implementation. Instead of directly using their reported results, we re-run the baselines from scratch to ensure their hardware and search space are the same. So, we can fairly compare them in terms of test error and search cost.

The test error and search cost on CIFAR-10 are stated in Table 3.2, where "c/o" signifies

Model	Test Error (%)	Params (M)	Search Cost (GPU hours)	Search Batch Size
DenseNet-BC [71]	3.46	25.6	-	-
NASNet-A + c/o [61]	2.65	3.3	43200	-
AmoebaNet-B + c/o [63]	2.55 ± 0.05	2.8	75600	-
ENAS + c/o [28]	2.89	4.6	12	-
ProxylessNAS-G + c/o [42]	2.08	5.7	-	-
NAONet-WS + c/o [43]	2.93	2.5	7.2	-
AlphaX + c/o [72]	2.06	9.36	360	-
DARTS (2nd order) + c/o [27]	2.83 ± 0.06	3.4	96	64
SNAS-moderate + c/o [64]	2.85 ± 0.02	2.8	36	64
NASP + c/o [65]	2.44 ± 0.04	7.4	4.8	64
PC-DARTS + c/o [67]	2.57 ± 0.07	3.6	2.4	256
DARTS (2nd order) + c/o [27]	7.25 ± 4.20	1.8 ± 0.6	53.62 ± 5.06	60
SNAS + c/o [64]	$\mathbf{2.58 \pm 0.08}$	8.8 ± 0.5	11.06 ± 0.2	60
NASP + c/o [65]	2.76 ± 0.35	5.5 ± 0.8	6.44 ± 0.12	60
PC-DARTS + c/o [67]	2.59 ± 0.05	6.5 ± 0.9	8.96 ± 0.08	60
NASB + c/o	2.64 ± 0.09	5.4 ± 1.2	$\mathbf{3.92 \pm 0.38}$	60
PC-DARTS + c/o [67]	2.60 ± 0.17	5.5 ± 1.2	4.10 ± 0.03	224
NASB + c/o	$\mathbf{2.49 \pm 0.07}$	6.9 ± 1.5	1.64 ± 0.06	448
NASB + CMP + c/o	2.53 ± 0.06	6.9 ± 0.7	$(\mathbf{1.48 \pm 0.02}) \times 4$	896

Table 3.2: Comparison with state-of-the-art NAS on CIFAR-10.

Cutout [73] used in the evaluation phase. The first row (human designed networks) and the second group of rows are extracted from their papers. The third group compares NASB with differentiable NAS baselines. The fourth group compares NAS algorithms using large batch sizes. Notably, ProxylessNAS attains the outstanding test error, but its supernet structure and search space are different from what we use, which might bias the comparison.

In the third group, NASB significantly takes the cheapest search cost, roughly 4 hours, to reach a comparable test error of 2.64% with SNAS 2.58% and PC-DARTS 2.59%, not to mention the search cost is smaller than the second group. NASB and NASP use similar mixed binary operations, but NASB outperforms NASP in both search cost (3.92 versus 6.44) and test error (2.64 versus 2.76). The GPU memory utilization of NASB and NASP is 2,117 MB and 9,587 MB, respectively. These three comparisons indicate that the additional gradient computation for inactive operations is a useful technique. Notice that DARTS, SNAS, and PC-DARTS use different search space, so

the original test errors (second group) differ from what we report in the third group of Table 1. Especially, DARTS tends to overfit the validation set by selecting "skip_connect" in our search space. Its results are not as good as their original search space. Even though NASP uses the same search space, the different batch sizes and random seeds for the search and retrain setting still lead to different results.

The fourth group points out that NASB can considerably reduce GPU memory by using twice batch sizes larger than PC-DARTS within 1.64 hours to attain a test error of 2.49. PC-DARTS, however, becomes worse when using large batch sizes. We consider that the Hessian approximation in PC-DARTS fluctuates greatly with large batch sizes, which misleads PC-DARTS to easily select "skip_connect". NASB-CMP using four GPUs enables twice batch sizes for NASB to finish its search in 1.48 hours without severe test error degradation. Its test error 2.53% also performs better than other differentiable NAS. The empirical results in the third and fourth groups demonstrate the high efficient NASB with significant memory saving and the strong performance of NASB-CMP.

3.3.5 Large Supernets on CIFAR-10

One-shot NAS embraces two limitations, (1) searching 8-layer supernets for 20-layer compact networks and (2) same cell structures [27, 28]. We hypothesize that 20-layer supernets with different cell structures can build suitable compact networks. Thanks to NASB and NASB-CMP that reduce GPU utilization and exploit multiple GPUs, we can examine how supernet architectures affect NAS. Table 3.3 shows test errors on CIFAR-10 with various supernet architectures, where the 1st and 2nd rows indicate cell diversity and layers (cells) numbers. Since 8-layer supernets could have six varying normal cells, we magnify each normal cell three times to construct compact networks.

First, supernets with large layers do not benefit NAS to discover high-quality compact networks. Test errors in NASB (3rd row) and NASB-CMP (4th row) show that most 8-layer supernets can generate lower test errors than 20-layer supernets. The reason is 20-layer supernets have numerous architecture parameters and network weights, and they should ask for more search epochs than 8-layer supernets to train. Insufficient search epochs for deep supernets do not help NAS reach

Cell structure	Same		Different	
# Layers	8	20	8	20
NASB	2.59	2.78	2.52	2.82
NASB-CMP	2.58	2.78	2.76	2.68

Table 3.3: Compare test error with different supernets on CIFAR-10.

strong compact networks. Furthermore, supernets with different cell structures are not beneficial for NAS as well. When we compare results in 2nd and 4th columns (or 3rd and 5th columns), most supernets using the same cell structures can generate similar or lower test errors than using different cell structures. The reason is close to the previous one. Different cell structures demand extra search epochs to train high-dimensional architecture parameters compared to homogeneous cell structures. Not enough epochs for different cell structures do not produce low test error. Although the results contradict the hypothesis, NASB-CMP shows its potential to explore complicated supernet architectures, which paves the way for designing supernet architectures.

3.4 Related Work

Parallelism has been applied to NAS for acceleration [37, 64, 42, 74]. Parameter servers in NAS [37] train several child networks in parallel to speed up the learning process of the controller. ProxylessNAS [42] speed up its retrain phase by a distributed framework, Horovod [75]. SNAS [64] and AtomNAS [74] have accelerated the search phase by data parallelism. Data parallelism runs data partitions simultaneously across multiple devices, but it cannot parallelize large models exceeding the memory of a single device, especially complicated supernets with large batch sizes. In contrast, model parallelism [76, 77, 69, 68] excels at parallelizing large models. GPipe [69] splits mini-batches to micro-batches and execute micro-batches in the pipeline of model partitions. The pipeline manner mitigates low hardware utilization in model parallelism. Consecutive model parallel is motivated by pipeline parallelism to overlap sub-tasks of forward/backward phases. We found that batch splitting and re-materialization [78] of GPipe increase NAS search time because frequently updating $\boldsymbol{A}$ and w intensifies extra computation. To the best of our knowledge, CMP is the most

efficient model parallelism for NAS.

Reducing GPU utilization to enlarge search batch sizes is another acceleration techniques [67, 66, 64, 65, 42]. PC-DARTS [67] samples channels of feature maps in mixed operations. P-DARTS [66] reduce search space as it progressively increases layers of supernets in the search phase. ProxylessNAS [42] and NASP [65] binarize $\boldsymbol{A}$ to reduce all operations saved in GPU. NASB uses the same binarization as NASP but saves one active operation in the mixed operations. Thus, NASB can reduce GPU consumption substantially and give CMP more space to keep two computation graphs in GPUs.

3.5 Conclusion

We proposed a simple and efficient model parallel approach, NASB-CMP, which overlaps sub-tasks of forward and backward phases to reduce idle time across GPUs and utilize binary architecture parameters to reduce GPU utilization for heavy supernets. Experiments on CIFAR-10 show NASB-CMP runs 1.2$\times$ faster with a large batch size of 896 than other model parallel approaches in 4 GPUs and only took 1.48 hours to attain a test error of 2.53, surpassing state-of-the-art differentiable NAS. Moreover, NASB-CMP is able to accommodate high complicated supernets for search, which paves the way for supernet network architecture design. In the future, we will combine the data parallel with NASB-CMP to overcome its inferior scalability, investigate effective and complicated supernet architectures, and analyze the communication overhead of NASB-CMP in a multi-node GPU cluster.

4. ACE: ADAPTIVE CONSTRAINT-AWARE EARLY STOPPING IN HYPERPARAMETER OPTIMIZATION*

Deploying machine learning models requires high model quality and needs to comply with application constraints. That motivates hyperparameter optimization (HPO) to tune model configurations under deployment constraints. The constraints often require additional computation cost to evaluate, and training ineligible configurations can waste a large amount to tuning cost. In this chapter, we propose an Adaptive Constraint-aware Early stopping (ACE) method to incorporate constraint evaluation into trial pruning during HPO. To minimize the overall optimization cost, ACE estimates the cost-effective constraint evaluation interval based on a theoretical analysis of the expected evaluation cost. Meanwhile, we propose a stratum early stopping criterion in ACE, which considers both optimization and constraint metrics in pruning and does not require regularization hyperparameters. Our experiments demonstrate superior performance of ACE in hyperparameter tuning of classification tasks under fairness or under robustness constraints.

4.1 Introduction

When machine learning (ML) is deployed in real-world applications, practitioners desire to tune hyperparameters of a model to maximize its utility. The model is often required not only to optimize for ML objectives (e.g., accuracy, l2 loss, or F1 scores), but also to meet the deployment constraints, such as latency, storage, fairness, robustness, and explainablility. For example, when fairness is concerned, it is required to treat different demographics fairly [80]. The constraints increase the challenge of hyperparameter optimization (HPO). If constraints are difficult to meet, we may spend high cost on ineligible trials, which violate the constraints. If constraints are expensive to compute, the step of constraint checking adds non-trivial overhead to HPO. How to early stop ineligible trials and how frequent to evaluate constraints are important factors for constrained HPO. Moreover,

*Reprinted with permission from "ACE: Adaptive Constraint-aware Early Stopping in Hyperparameter Optimization" by Yi-Wei Chen, Chi Wang, Amin Saied, and Rui Zhuang, 2022, ACM SIGKDD AutoML Workshop, Copyright 2022 by ACM SIGKDD AutoML Workshop.

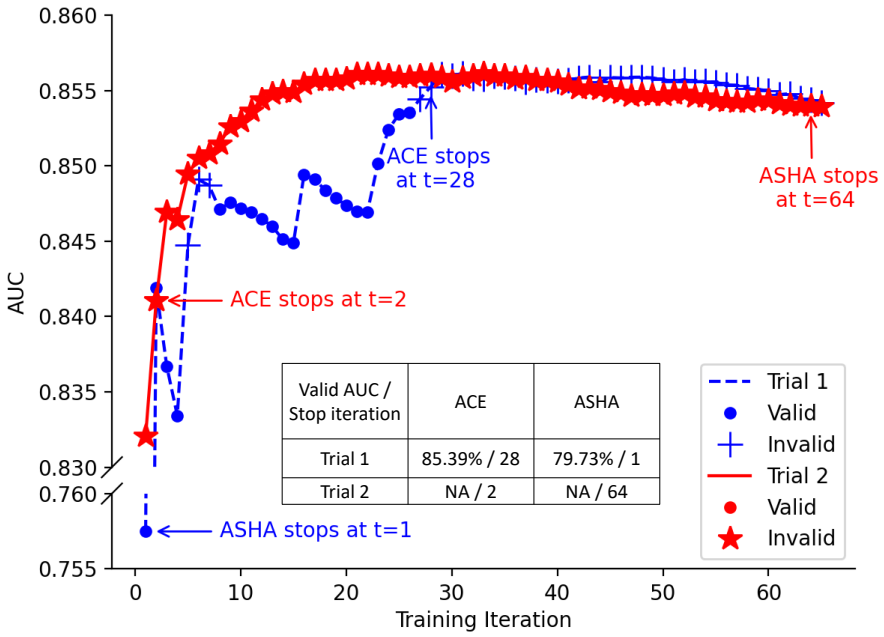


Figure 4.1: How ACE (our method) and ASHA terminate two trials from random search. '•' signifies a valid checkpoint (training iteration), satisfying the constraint, while '+' represents an invalid checkpoint, violating the constraint. ACE is a constraint-aware early stopping approach, while ASHA [79] is a constraint-agnostic method. For Trial 1, ACE tolerates small extent of constraint violation in checkpoints 5-7 and stops it at the 28th checkpoint, close to the optimal stop point (26) for this trial. ASHA wrongly stops this promising trial in the beginning. For Trial 2, all the checkpoints are invalid. ACE can stop this infeasible trial early due to large extent of constraint violation, while ASHA keeps training it until $t = 64$. These are only two example trials to illustrate the potential advantage of using constraint information in early stopping and explain the overall empirical effectiveness of ACE even though the stop point is not necessarily optimal for every trial. Reprinted with permission from [1].

constraints vary in different problems in terms of constraint checking cost and difficulty to satisfy. A good solution needs to adapt to unknown constraint characteristics automatically.

We propose Adaptive Constraint-aware Early stopping (ACE), which makes use of constraint evaluation to terminate inferior trials. It decides the frequency to check constraints and the moment to stop trials to improve the efficiency of constrained HPO. We model the expected trial cost in terms of the constraint evaluation interval and provide theoretical analysis for the optimal constraint evaluation interval in the cost model. Based on the theoretical results, ACE suggests cost-effective constraint evaluation interval for each trial. Meanwhile, we propose a simple yet effective *stratum truncation* policy to prune unqualified trials. According to constraint evaluation results,

the approach categorizes trials into groups: no-constraint-evaluation, satisfying-constraints, and violating-constraints. It terminates inferior trials of each group, which have either a relatively large extent of constraint violation or bad optimization objective. Unlike regularization methods [81], our approach does not introduce additional penalty hyperparameters. It automatically balances the goal of optimization and constraint satisfaction.

ACE is the first constraint-aware early stopping algorithm for HPO. Our experiments demonstrate that ACE obtains superior performance to constraint-agnostic early stopping baselines [79], on UCI credit card dataset [82] with a fairness constraint [83] and on GLUE SST2 [84] with a robustness constraint [85].

4.2 Adaptive Constraint-aware Early Stopping (ACE)

We present the target problem of ACE in Section 4.2.1, followed by our model of the trial cost with respect to the constraint evaluation interval in Section 4.2.2. The cost model is used to suggest cost-effective constraint evaluation interval for each trial (Section 4.2.3). To use both optimization and constraint metrics for pruning unqualified trials, we propose a *stratum truncation* policy (Section 4.2.4). Finally, we further optimize our method to reduce cost in constraint evaluations. (Section 4.2.5).

4.2.1 Problem Statement

Given a constraint metric g and the constraint threshold τ, and the optimization metric ℓ, we target at finding the best feasible hyperparameter configuration $x \in \mathcal{X}$ with minimal computation cost. The best feasible hyperparameter configuration is defined as:

$$\arg\min_{x \in \mathcal{X}} \ell(x) \text{ s.t. } g(x) \leq \tau,$$

The optimization metric could be validation loss or other metrics to minimize. We assume the values of the optimization metrics and constraint metrics change as training iterations increase (otherwise we only need to evaluate them once per trial). We also assume the constraint metric evaluation incurs non-negligible cost, and the constraint metrics are selectively evaluated at some training

iterations (i.e., checkpoints). $\ell(x)$ and $g(x)$ correspond to the metrics at the best checkpoint among all the training iterations where these metrics are evaluated. We focus on designing an effective early stopping policy from two aspects: when to evaluate constraint metrics, and when to terminate a trial. Our early stopping policy is not tied to a particular hyperparameter search algorithm.

4.2.2 Expected Trial Cost

For a given trial, let C_1 be the *constraint cost*, the evaluation cost of the constraint metrics at one training iteration. Let C_2 be the *primary cost*, the training cost plus the evaluation cost of the optimization metric for one training iteration. We aim to reduce the total cost by choosing an appropriate *constraint evaluation interval*, i.e., the frequency of computing the constraint metrics. When the constraint evaluation interval is β, the constraint metrics are computed every β training iterations. In other words, the constraints will be evaluated at iterations $\beta, 2\beta, \ldots, (z-1)\beta$, or $z\beta$, where z is the smallest integer such that $(z-1)\beta < T \leq z\beta$, and T is the maximal iteration of a trial. We further assume that p is the stop probability to prune a trial. For instance, if we stop a fixed percentage of inferior trials at each checkpoint, the stop percentage can function as the stop probability. Then, we can formulate the expected trial cost by:

$$\mathbb{E}[C] = (1-p)^z[C_1 z + C_2 T] + \sum_{k=1}^{z}(1-p)^{k-1}p[C_1 k + C_2 k\beta]. \tag{4.1}$$

Its first part models the cost if a trial does not stop early, while the second part models the cost of terminating a trial after k times of constraint evaluations, for $k = 1, 2, \ldots, z$. There are k constraint evaluations in $k\beta$ training iterations. equation 4.1 is general to any early stopping policy with linear constraint checking intervals.

We define cost ratio as $r = C_1/C_2$. Then, we can simplify equation 4.1 into $\mathbb{E}[C] = C_2(r + \beta)\frac{1-(1-p)^z}{p}$ in the following derivation. The first part of equation 4.1 is simplified as:

$$\begin{aligned}(1-p)^z[C_1 z + C_2 T] &= (1-p)^z[rC_2 z + C_2 z\beta] \\ &= (1-p)^z C_2(r+\beta)z.\end{aligned}$$

The second part of equation 4.1 can be simplified as:

$$\sum_{k=1}^{z}(1-p)^{k-1}p[C_1k+C_2k\beta] = \sum_{k=1}^{z}(1-p)^{k-1}p[rC_2k+C_2k\beta]$$

$$=pC_2(r+\beta)\sum_{k=1}^{z}k(1-p)^{k-1}$$

$$\because \sum_{k=1}^{z}k(1-p)^{k-1} = \sum_{k=0}^{z}k(1-p)^{k-1} = \frac{d}{dp}[-\sum_{k=0}^{z}(1-p)^k]$$

$$=\frac{d}{dp}[-\frac{1-(1-p)^{z+1}}{p}] = \frac{1}{p^2} + \frac{(z+1)(1-p)^z(-1)p-(1-p)^{z+1}}{p^2}$$

$$=\frac{1-(1-p)^z(zp+1)}{p^2}$$

$$\therefore \sum_{k=1}^{z}(1-p)^{k-1}p[C_1k+C_2k\beta] = pC_2(r+\beta)\frac{1-(1-p)^z(zp+1)}{p^2}.$$

Combining the above equations, we get equation 4.2.

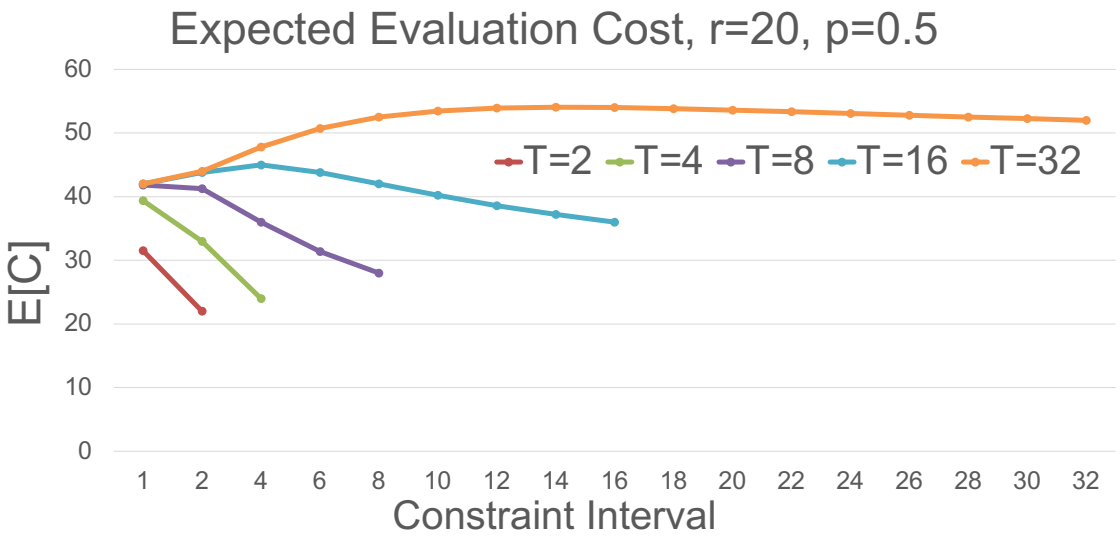
$$\begin{aligned}\mathbb{E}[C] &= (1-p)^zC_2(r+\beta)z + pC_2(r+\beta)\frac{1-(1-p)^z(zp+1)}{p^2} \\ &= C_2(r+\beta)\frac{zp(1-p)^z+1-(1-p)^z(zp+1)}{p} \\ &= C_2(r+\beta)\frac{1-(1-p)^z}{p}.\end{aligned}$$

By definition, $z \geq \frac{T}{\beta}$. If $z > \frac{T}{\beta}$, then we can set $\beta' = \frac{T}{z} < \beta$ and decrease $\mathbb{E}[C]$. Thus, an optimal β in $[1, T]$ should make $z = \frac{T}{\beta}$. We only need to minimize:

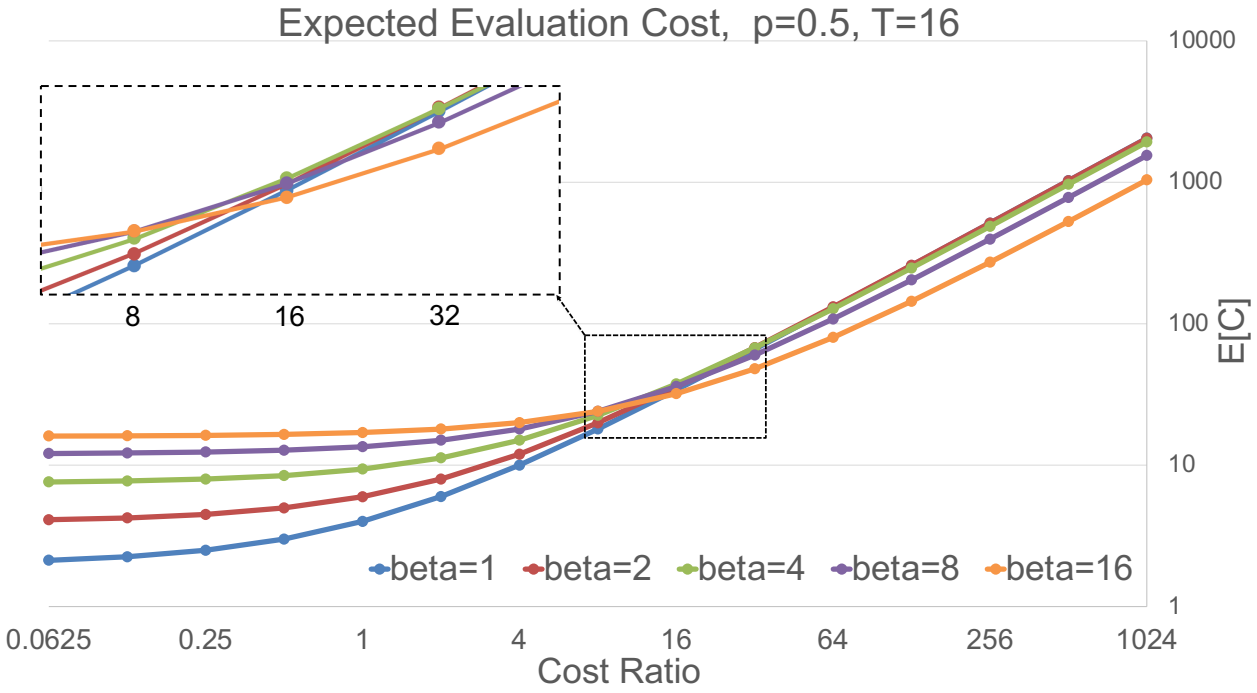
$$\mathbb{E}[C|\beta] = C_2(r+\beta)\frac{1-(1-p)^{\frac{T}{\beta}}}{p}. \tag{4.2}$$

4.2.3 Constraint Evaluation Interval

Given p, T, and r, we want to obtain the optimal constraint evaluation interval β^* to minimize equation 4.2. We analyze the equation by fixing the stop probability $p = 0.5$. First of all, we observe how max iteration T changes $\mathbb{E}[C|\beta]$ in Figure 4.2a, when the cost ratio r is 20. It points out that



(a) $\mathbb{E}[C|\beta]$ given $r = 20$ and $p = 0.5$.



(b) $\mathbb{E}[C|\beta]$ given $p = 0.5$ and $T = 16$.

Figure 4.2: Examples of the expected trial cost in a stop probability $p = 0.5$. Left: given the cost ratio $r = 20$, how the cost changes for various max iterations T. Right: given $T = 16$, how the cost changes for different cost ratio r. Reprinted with permission from [1].

β^* is either 1 or T for different trials. Then, we set $T = 16$ to observe how r changes $\mathbb{E}[C|\beta]$ in Figure 4.2b. It also demonstrates that β^* appears in $\{1, T\}$. The left part of the figure ($r = 0.0625$ to 8) favors $\beta = 1$, while its right part ($r = 16$ to 1,024) prefers $\beta = T$. The two cases both indicate that the optimal β^* is situated in $\{1, T\}$, the two ends of β's range. We can prove Theorem 2 for general cases:

Theorem 2. *$\forall p, r$, and T, the optimal β^* for equation 4.2 is 1 or T, i.e.,* $\arg\min_{\beta\in[1,T]} \mathbb{E}[C|\beta] \in \{1, T\}$. *If* $r < \frac{pT+(1-p)^T-1}{1-p-(1-p)^T}, \beta^* = 1$. *If* $r > \frac{pT+(1-p)^T-1}{1-p-(1-p)^T}$, $\beta^* = T$. *If* $r = \frac{pT+(1-p)^T-1}{1-p-(1-p)^T}$, $\beta^* = \{1, T\}$.

Proof. We define $s := (1-p)^{-T} > 1$ and $T > 1$. $f(\beta) = (r+\beta)\frac{1-s^{-\frac{1}{\beta}}}{p}$ and $\mathbb{E}[C|\beta]$ have the same minimizers for $\beta \in [1, T]$. We take the derivative of $f(\beta)$:

$$f'(\beta) = \frac{1 - s^{-\frac{1}{\beta}}}{p} - \frac{\ln(s)(\beta + r)}{ps^{\frac{1}{\beta}}\beta^2}$$

$\because s > 1$, we notice that $f'(\beta)$ decreases as r increases, and:

$$f'(\beta) = 0 \Leftrightarrow r = -\frac{\beta \ln(s) - \beta^2 s^{\frac{1}{\beta}} + \beta^2}{\ln(s)} = g(\beta) \tag{4.3}$$

Then we take the derivative of $g(\beta)$, $h(s)$, and $i(s)$:

$$g'(\beta) = \frac{\left(2s^{\frac{1}{\beta}} - 2\right)\beta - s^{\frac{1}{\beta}}\ln(s) - \ln(s)}{\ln(s)} \triangleq \frac{h(s)}{\ln s}$$
$$h'(s) = -\frac{s^{\frac{1}{\beta}}\ln(s) - \beta s^{\frac{1}{\beta}} + \beta}{\beta s} \triangleq -\frac{i(s)}{\beta s}$$
$$i'(s) = \frac{s^{\frac{1}{\beta}-1}\ln(s)}{\beta} \geq 0, s \geq 1$$

$\because i'(s) \geq 0, i(s) \geq i(1) = 0, s \geq 1$. That means $h'(s) \leq 0, s \geq 1$. So $h(s) < h(1) = 0, s > 1$. Hence, $g'(\beta) < 0, s > 1$. Now, assume $f'(\beta_0) = 0$ for $\beta_0 \in [1, T]$. We know from equation 4.3 that $r = g(\beta_0)$. For any $1 \leq \beta_1 < \beta_0$, we know $g(\beta_1) > g(\beta_0) = r$ according to $g'(\beta) < 0$. Because

$f'(\beta)$ decreases when r increases, we have:

$$f'(\beta_1) > \frac{1 - s^{-\frac{1}{\beta_1}}}{p} - \frac{\ln(s)\,(\beta_1 + g(\beta_1))}{p s^{\frac{1}{\beta_1}} {\beta_1}^2} = 0$$

Likewise, for any $\beta < \beta_2 \leq T$, $f'(\beta_2) < 0$. These mean that if $f'(\beta)$ has a root β_0 in $[1, T]$, $f(\beta)$ is monotonically increasing in $[1, \beta_0]$, and monotonically decreasing in $[\beta_0, T]$. If $f'(\beta)$ does not have a root in $[1, T]$, then it is either monotonically increasing or monotonically decreasing in $[1, T]$ because $f'(\beta)$ is continuous in $[1, T]$. Therefore, in either case, the minimizer of $f(\beta)$ is either 1 or T.

Finally, we compare $f(1)$ and $f(T)$.

$$\begin{aligned} & f(1) < f(T) \Leftrightarrow (r+1)(1 - s^{-1}) < (r+T)(1 - s^{-1/T}) \\ \Leftrightarrow & (r+1)(1 - (1-p)^T) < (r+T)p \\ \Leftrightarrow & r < \frac{pT + (1-p)^T - 1}{1 - p - (1-p)^T} \end{aligned}$$

□

Theorem 2 states that the best constraint interval for equation 4.2 is either 1 or T. It also formulates the relationship between r, p, T, and β. By substituting $p = 0.5$ and $T = 16$, we get $\frac{pT + (1-p)^T - 1}{1 - p - (1-p)^T} = 14$. Figure 4.2b verifies the theoretical value: if cost ratios are less than 14, $\beta = 1$ costs the least. By using $p = 0.5$ and $r = 20$, we get $20 > \frac{0.5T + (0.5)^T - 1}{0.5 - (0.5)^T} \rightarrow 22 > T$. Figure 4.2a also verifies the theoretical value: curves ($T = 2, 4, 8, 16$) both favor $\beta = T$ to reach the smallest expected cost. The two figures confirm Theorem 2.

According to the analysis, we design the adaptive constraint evaluation interval as follows. Each trial receives its own constraint evaluation interval. In the beginning, we do not know the cost ratio r. The worst case is that the constraint evaluation is expensive. Using $\beta = 1$ as the initial value will lead to great cost. Thus, ACE selects $\beta = T$ as the initial value, which means the constraint evaluation occurs at the end of training. During the training of a trial, ACE records the primary

cost and constraint cost on the fly. ACE computes the cost ratio using the average constraint cost divided by the average primary cost. After the cost ratio is computed, it picks the low-cost β as the constraint evaluation interval by comparing r and $\frac{pT+(1-p)^T-1}{1-p-(1-p)^T}$. Line 1 in Algorithm 3 implements the above statements.

4.2.4 Stratum Truncation

We propose *stratum truncation* to use both optimization and constraint metrics to prune trials (Line 18 to 25 of Algorithm 3). It is motivated by several desiderata. First, we want to respect both the goal of optimization and constraint satisfaction in the early stopping. That requires pruning both the trials with bad optimization metric and the trials violating the constraints. Second, an invalid yet incomplete trial has a chance to meet the constraints with more training iterations. So it is not necessary to stop every invalid trial immediately. Third, as an HPO solution, it is preferred to avoid introducing additional hyperparameters such as penalty weights of the constraint metrics.

Our solution categorizes the trials into 3 groups and prunes the same fraction of low-rank trials in each group: (1) no-constraint: trials without constraint evaluation, (2) valid: trials meeting constraints, and (3) invalid: trials violating constraints. The no-constraint trials do not have constraint metrics evaluated at the current training iteration. The only performance indicator is the optimization metric. We rank the trials in this group by their optimization metric. For valid trials, we only care about their optimization metric, so we rank the trials in this group by their optimization metric. For the invalid group, we desire to penalize invalid trials by their violation amount ($g(x) - \tau$). The trials with larger violation amount should rank low. Thus, we sort the trials by their violation amount and use the optimization metric to break ties. Infeasible trials are not pruned immediately unless their violation amounts are on top. That gives the 'close to valid' trials a chance to meet the constraints later and potentially lead to better optimization results. At the same time, even if no infeasible trials are turned to feasible in the end, the promising trials in the valid group are kept running just like the case of unconstrained early stopping.

Algorithm 3: Adaptive Constraint-aware Early Stopping (ACE)

Input: hyperparameter configuration x; total training iterations T; constraint threshold τ; running history $\mathcal{H}$; truncation percentage $\mathcal{P}$.

1: Estimate the constraint evaluation interval β based on Theorem 2
2: **for** t in $[T]$ **do**
3: Train the model with x for one iteration
4: Compute optimization metric $l(x, t)$ at the iteration t
5: **if** t mod $\beta == 0$ and $l(x, t) \leq \mathcal{H}.f^*$ **then**
6: Compute constraint metric $g(x, t)$ at the iteration t
7: **if** $g(x, t) \leq \tau$ **then**
8: trial_type = "valid"
9: $\mathcal{H}.f^* = l(x, t)$
10: **else**
11: trial_type = 'invalid"
12: **end if**
13: **else**
14: trial_type = "no_constraint"
15: **end if**
16: Update $\mathcal{H}$
17: $\mathcal{H}'$ = $\mathcal{H}$.subset(trial_type)
18: **if** trial_type == "invalid" **then**
19: $\mathcal{H}'$.sort(keys=[violation_amount, optimization_metric])
20: **else**
21: $\mathcal{H}'$.sort(keys=[optimization_metric])
22: **end if**
23: **if** x is below $\mathcal{P}\%$ of $\mathcal{H}'$ **then**
24: stop training
25: **end if**
26: **end for**

4.2.5 Reduce Constraint Evaluation Overhead

While constraint evaluation helps prune ineligible trials, it adds significant overhead if the constraint metric is expensive to compute. To reduce such overhead, ACE performs constraint evaluation on promising trials only (Line 5 to 15 of Algorithm 3). Specifically, ACE maintains the global best feasible score, which is the best optimization metric value among the trials satisfying the constraint. Only if the current optimization metric is better than the global best feasible score does ACE compute the constraint of the current checkpoint. Otherwise, ACE skips the constraint

Name	Type	Lower value	Upper value	Initial value
n_estimators	ray.tune.lograndint	4	the number of instances	4
num_leaves	ray.tune.lograndint	4	the number of instances	4
min_child_samples	ray.tune.lograndint	2	129	20
learning_rate	ray.tune.loguniform	1/1024	1.0	0.1
log_max_bin	ray.tune.lograndint	3	11	8
colsample_bytree	ray.tune.uniform	0.01	1.0	1.0
reg_alpha	ray.tune.loguniform	1/1024	1024	1/1024
reg_lambda	ray.tune.loguniform	1/1024	1024	1.0

Table 4.1: The hyperparameter space of LightGBM. Reprinted with permission from [1].

evaluation for this checkpoint and marks the trial type of the current checkpoint as "no-constraint". Note that bad trials in the "no-constraint" group still have the chance to be pruned due to our stratum truncation policy, even without constraint evaluation. With this low-overhead mechanism, ACE wastes less time in evaluating suboptimal trials.

4.3 Experiments

We evaluate our algorithm under a fairness constraint and a robustness constraint respectively. Since ACE is an early stopping approach, our baselines are no-stopping and a state-of-the-art early stopping method, ASHA [79]. ASHA is a parallel early stopping method using successive halving [86] to allocate budgets for trials. ACE uses 25% as the truncation percentage for all the experiments. We analyze the impact of the truncation percentage in Sec 4.3.6. The reported results are the average of three random seeds in all the experiments.

4.3.1 Experiment Settings

The fairness experiments of Section 4.3.2 are run on 12 cores of Intel i7-6850K@3.6 GHz. We search hyperparameters of LightGBM under a fairness constraint, where its search space follows FLAML [87], shown in Table 4.1. The types in the table follow RAY [88]. ASHA follows the default values of RAY library [88]: reduction factor = 4, brackets = 1, grace period =1, and max time units = 21,000 (the number of training data). The random seeds of the experiments are 20, 21, 22. The max concurrent trials are 4. The fairness experiments of in Section 4.3.4 also follows the

Name	Type	Lower value	Upper value	Initial value
learning_rate	ray.tune.loguniform	1e-6	1e-3	1e-5
num_train_epochs	ray.tune.loguniform	0.1	10.0	3.0
per_device_train_batch_size	ray.tune.choice	4, 8, 16, 32		32
warmup_ratio	ray.tune.uniform	0.0	0.3	0.0
weight_decay	ray.tune.uniform	0.0	0.3	0.0
adam_epsilon	ray.tune.loguniform	1e-8	1e-6	1e-6
seed	ray.tune.choice	40, 41, 42, 43, 44		42

Table 4.2: The hyperparameter space of finetuning DistilBER. Reprinted with permission from [1].

same setup here.

The robustness experiments of Section 4.3.3 are run on 64 cores of AMD EPYC 7282 and 8 NVIDIA RTX A5000. PYTHON 3.6 environment includes FLAML 0.6.9, RAY 1.10, CHECKLIST 0.0.11, and HUGGINGFACE-HUB 0.4.0. We search hyperparameters for finetuning DistilBERT on SST2 under a robustness constraint. The search space is motivated by FLAML library [87], as show in Table 4.2. The types in the table follow RAY [88]. The twenty one test task in the robustness constraint are illustrated in Table 4.7. ASHA follows the default values of RAY library [88]: reduction factor = 4, brackets = 1, grace period =1, and max time units = 67,349 (the number of training data). The random seeds of the experiments are 19, 20, 21. Th max concurrency trials are 4. The search budget is three hours. The robustness experiments of in Section 4.3.4 also follows the same setup here.

4.3.2 Fairness Constraint

We follow Fairlearn [83] to preprocess the UCI credit card default dataset [82]. The processed dataset has 30k clients with 22 features, which we split 70/30% for training/validation. The fairness constraint is Equalized-Odd-Difference (EOD) [83] = $\max(|FPR_1 - FPR_2|, |FNR_1 - FNR_2|)$, where FPR and FNR mean false positive rate and false negative rate, respectively. EOD quantifies the accuracy disparity experienced by different groups. We search hyperparameters of LightGBMs under two threshold levels: (i) the hard constraint: $EOD \leq 0.25$ and (ii) the easy constraint: $EOD \leq 0.325$. For each case, we evaluate ACE's performance with unconstrained random

Constraint	Searcher	Early stopping policy	Total trials	Best feasible AUC (%)	Time to best
EOD ≤ 0.25 (hard)	Blend search	No-stopping	276.0	84.95 ± 0.43	1h 1m 7s
		No-stopping w/ constraint callback	626.3	84.98 ± 0.51	0h 5m 5s
		ASHA	3,664.3	82.88 ± 3.13	1h 2m 45s
		ASHA w/ constraint callback	4,252.0	85.06 ± 0.35	0h 5m 19s
		ACE	1,799.3	**85.53 ± 0.03**	0h 29m 09s
	Random search	No-stopping	37.0	84.28 ± 0.45	1h 0m 15s
		ASHA	3,497.7	84.19 ± 0.35	1h 0m 53s
		ACE	283.7	**85.38 ± 0.06**	0h 39m 09s
EOD ≤ 0.325 (easy)	Blend search	No-stopping	694.3	85.85 ± 0.03	1h 0m 4s
		No-stopping w/ constraint callback	307.0	85.83 ± 0.05	0h 15m 57s
		ASHA	3,763.3	85.82 ± 0.08	1h 0m 18s
		ASHA w/ constraint callback	3,677.3	85.81 ± 0.11	0h 24m 15s
		ACE	2,197.7	**85.89 ± 0.00**	0h 54m 22s
	Random search	No-stopping	37.0	85.71 ± 0.10	1h 0m 03s
		ASHA	3,596.7	85.77 ± 0.03	1h 0m 17s
		ACE	1,200.0	**85.79 ± 0.07**	0h 19m 37s

Table 4.3: Best feasible AUC under the fairness constraint. Reprinted with permission from [1].

search [89] or constrained blend search [90]. Blend search combines global and local search and prioritizes their suggestion on the fly. Its implementation in the FLAML library [87] can handle constrained optimization. It penalizes the optimization metric with the amount of constraint violation. Since the constrained HPO needs constraint information, we implement the constraint callbacks for the baselines. The callbacks compute constraint metrics for a searcher. The search budget is one hour. The optimization metric is Area Under the Curve (AUC) of ROC [91].

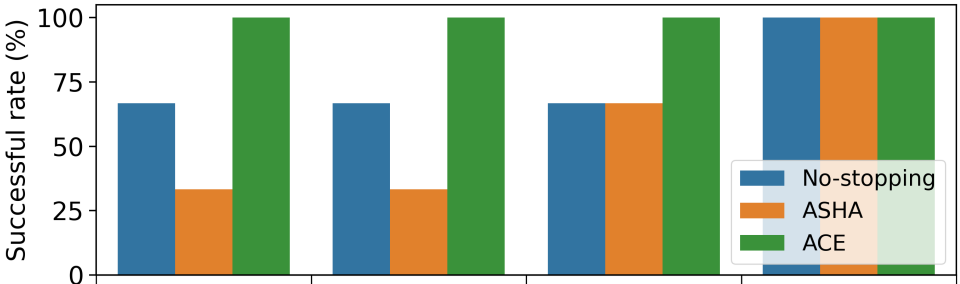
Table 4.3 shows the results under the fairness constraint. "Time to best" means the time to find the best feasible trial within each run from the beginning of search. For baselines without the constraint callback, after the tuning, we sort the searched trials by their optimization metric and compute constraints from top to down until we find the first feasible trial. So their "time to best" is longer than the one hour budget. If a method can find the highest feasible AUC compared to other methods, then smaller "Time to best" is better. Otherwise, smaller "Time to best" just indicates a method converges to a suboptimal point early.

ACE outperforms the baselines in terms of feasible AUC when using both searchers under both easy and hard constraints. In blend search with the hard constraint (top Table 4.3), ACE needs

only 1/2 of the search budget to attain the best feasible AUC. ASHA and no-stopping converge to worse feasible AUC even when using the full budget. Their feasible AUC are not improved after 5 minutes. Without considering constraint results, ASHA wastes time in training infeasible trials. Furthermore, although random search is an unconstrained searcher, ACE improves feasible AUC by 1.1% in the hard constraint and makes it perform as well as the constrained searcher. We make a case study of two trials both appearing in ACE and ASHA, shown in Figure 4.1. ASHA wrongly prunes a promising trial (blue curve) very quickly and spends unnecessary training time on an infeasible trial (red curve). ACE successfully prunes them at the appropriate iteration. It highlights that adding the constraint violation in the performance ranking makes ACE allocate the training budget to promising trials. For the easy constraint, constraint violations are rare. Our constrained early stopping gets close to constraint-agnostic early stopping, as few trials are categorized into the "invalid" group. In this case, ACE still performs as well as ASHA (bottom Table 4.3). It implies our stopping policy can also work well with limited utility of constraint feedback.

4.3.3 Robustness Constraint

We use DistilBERT [92] to predict the sentiment of a given sentence from the Stanford Sentiment Treebank (SST2) of GLUE [84]. DistilBERT is a small and fast transformer model, which uses BERT [93] as a teacher and is pretrained on the same corpus as BERT in a self-supervised fashion. SST is split into 67,349 and 872 sentences for training and validation respectively, where the sentences are encoded by the DistilBERT's pretrained tokenizer. The robustness constraints come from the NLP model CHECKLIST [85], which provides a sentiment test suite to check the prediction invariance in the presence of certain perturbation or certain corner cases. CHECKLIST measures the test score by the failure rate, a portion of predictions failing to remain original answers with perturbation. We use 21 test tasks in the test suite, where each task has 300 test cases. Average failure rate (AFR) is chosen to aggregate the results of the 21 test tasks. We use random search to find the hyperparameters for finetuning DistilBERT under four threshold levels: AFR ≤ 0.19, AFR ≤ 0.20, AFR ≤ 0.24, and AFR ≤ 0.25. We report average accuracy of three random seeds for each threshold.



	AFR <= 0.19	AFR <= 0.20	AFR <= 0.24	AFR <= 0.25
No-stopping	67.20%	67.20%	87.22%	**90.48%**
ASHA	83.49%	83.49%	85.89%	88.07%
ACE	**85.13%**	**85.93%**	**89.37%**	90.10%

Figure 4.3: Best feasible accuracy under the robustness constraint. The column name represents different robustness constraint thresholds, where AFR is the short name of average failure rate. The values in the table are the best feasible accuracy (%). Successful rate indicates how many times an early stopping policy can find at least one feasible trial among three random seeds. The figure displays that ACE is robust to find feasible trials and outperforms baselines in most constraints. Reprinted with permission from [1].

Figure 4.3 exhibits that ACE achieves 100% successful rate to get feasible trials under all the robustness constraints, while no-stopping and ASHA only attain 100% successful rate under AFR ≤ 0.25. For the best feasible accuracy (the table in Figure 4.3), ACE obviously outperforms state-of-the-art ASHA under all the robustness constraints. With the consideration of constraint values, ACE can wisely invest time on feasible trials. ACE also beats no-stopping under the first three constraint thresholds and performs competitively under AFR ≤ 0.25. The best feasible trials of no-stopping under AFR ≤ 0.25 possesses the worst accuracy in the initial training iterations. Hence, ACE prunes these two trials accordingly, as well as ASHA does. The results reveal that reducing training overhead on inferior trials sometimes leads to wrongful termination of promising candidates. Addressing this limitation would be promising future work.

4.3.4 Ablation Study

We analyze ACE design choices by evaluating (i) different stopping criterion and (ii) static/adaptive constraint evaluation intervals in Table 4.4 and Table 4.5, respectively. Since random search gener-

Constraint	Method	Stopping criterion	Low-overhead evaluation	Best feasible AUC (%)
EOD ≤ 0.25 (hard)	ACE$_{hard}$	hard stopping	✓	84.94 ± 0.31
	ACE$_{noskip}$	stratum		85.37 ± 0.02
	ACE	stratum	✓	**85.38** ± 0.06
EOD ≤ 0.325 (easy)	ACE$_{hard}$	hard stopping	✓	85.74 ± 0.05
	ACE$_{noskip}$	stratum		85.69 ± 0.03
	ACE	stratum	✓	**85.79** ± 0.07

Table 4.4: Ablation study for early stopping choices. Reprinted with permission from [1].

ates the identical sequences of hyperparameters given the same random seed, we select it to compare different choices fairly for the ablation study.

Our *stratum truncation* in ACE is a soft stopping approach, which tolerates invalid checkpoints. A hard stopping policy terminates a trial immediately as soon as the trial encounters the first invalid checkpoint. We implement the hard stopping in ACE$_{hard}$. Table 4.4 shows that ACE$_{hard}$ decreases performance for both easy and hard fairness constraints. The constraint values do not increase or decrease monotonically. It might temporarily violate the constraint and meet the constraint late, such as the blue curve in Figure 4.1. Just considering constraint validity, ACE$_{hard}$ might terminate trials at wrong iterations. Furthermore, ACE$_{noskip}$ faithfully evaluates constraints without skipping suboptimal trials which have inferior optimization metrics compared to the best feasible trial. Table 4.4 indicates that the reduction of constraint evaluation overhead improves ACE performance for the easy constraint and does not degrade the performance for the hard constraint. ACE$_{noskip}$ suffers from unnecessary constraint cost on suboptimal trials, especially when the constraint is expensive to evaluate.

Next, we study how constraint intervals affect the performance by replacing ACE's adaptive constraint interval with static values. ACE$_{\beta=T}$ enforces all the trials to check constraints once at the end of the training iteration. A trial's constraint metric is computed for its best checkpoint (with the best optimization metric). ACE$_{\beta=1}$ fixes $\beta = 1$ for all the trials. Table 4.5 demonstrates that ACE can adjust the constraint interval for different constraint characteristics. For the cheap

Constraint	Cost ratio	Method	$\beta = 1$ (%)	$\beta = T$ (%)	Best feasible score (%)
EOD ≤ 0.25 (cheap)	1.94	$\text{ACE}_{\beta=1}$	100	0	**85.38 ± 0.05**
		$\text{ACE}_{\beta=T}$	0	100	84.61 ± 0.57
		ACE	87	13	**85.38 ± 0.06**
AFR ≤ 0.2 (expensive)	23.98	$\text{ACE}_{\beta=1}$	100	0	84.33 ± 1.80
		$\text{ACE}_{\beta=T}$	0	100	79.82 ± 2.93
		ACE	36	64	**85.93 ± 1.15**

Table 4.5: Ablation study for the adaptive constraint evaluation interval. Reprinted with permission from [1].

constraint (low cost ratio), ACE assigns 87% of trials $\beta = 1$. For the expensive constraint (high cost ratio), ACE assigns 64% of trials $\beta = T$. The results match the theoretical prediction in Theorem 2. Since Eq. equation 4.2 also depends on a trial's total training iteration T, β is not fixed for all the trials. It is not surprising that $\text{ACE}_{\beta=1}$ can achieve competitive performance of ACE for the cheap constraint but weaker for the expensive constraint, as our theory predicts. $\text{ACE}_{\beta=T}$ has a chance in theory to outperform ACE for the expensive constraint, since $\text{ACE}_{\beta=T}$ largely reduces the constraint evaluation overhead. $\text{ACE}_{\beta=T}$ indeed searches 1.8 times more trials than ACE and 3.9 times more than $\text{ACE}_{\beta=1}$. Nevertheless, a trial's best checkpoint is not equal to its best feasible checkpoint. Some feasible trials in ACE are considered as infeasible by $\text{ACE}_{\beta=T}$ for that reason. Thus, $\text{ACE}_{\beta=T}$ ends up performing poorly. ACE selects $\beta = 1$ for trials with large T (See Figure 4.2a). Even though ACE selects $\beta = T$ for a trial with small T, its small number of checkpoints reduces the chance for ACE to use wrong checkpoint. ACE's adaptive constraint interval is merely derived from the constraint cost consideration. It is an interesting question for future work whether it is beneficial to adjust the constraint evaluation interval according to the feasibility consideration.

4.3.5 Comparison between Geometric and Linear Interval

Given the reduction factor = 4, ASHA examines trial performance by geometric intervals, i.e., 1, 4, 16, 64, ... In contrast, ACE check trials by linear intervals, i.e., 1, 2, 3, ... The different interval motivates us to study the performance of geometric intervals in constrained early stopping. We

Searcher	Method	Interval type	Stopping criterion	Constraint interval	Best feasible AUC (%)
Blend search	ASHA	geometric	no-stratum	fixed	85.05 ± 0.35
	$\text{ASHA}_{stratum}$	geometric	stratum	fixed	85.35 ± 0.07
	$\text{ASHA}_{stratum\_not\_fixed}$	geometric	stratum	not-fixed	85.16 ± 0.07
	ACE_{noskip}	linear	stratum	not-fixed	**85.41 ± 0.09**
Random search	ASHA	geometric	no-stratum	fixed	84.19 ± 0.35
	$\text{ASHA}_{stratum}$	geometric	stratum	fixed	85.27 ± 0.16
	$\text{ASHA}_{stratum\_not\_fixed}$	geometric	stratum	not-fixed	85.32 ± 0.17
	ACE_{noskip}	linear	stratum	not-fixed	**85.37 ± 0.02**

Table 4.6: Geometric vs. linear interval. Reprinted with permission from [1].

extend ASHA with adaptive constraint interval and stratum truncation. If the adaptive constraint interval suggests $\beta = 1$, we enforce ASHA to compute constraint values by geometric intervals. If the adaptive constraint interval suggests $\beta = T$, ASHA uses a trial's best checkpoint to evaluate constraint values at the last training iteration. Notice that Thereom 2 is developed by the linear interval assumption. We merely borrow it to suggest constraint intervals, rather than develop a new theorem. We use blend and random search to search hyperparameters of LightGBM under the hard fairness constraint (EOD ≤ 0.25). In Table 4.6, we observe the stratum truncation and adaptive constraint interval can improve constraint-agnostic ASHA. Since the constrained ASHAs do not have the low-overhead constraint evaluation, we report ACE_{noskip} to compare their performance. ACE_{noskip} outperforms $\text{ASHA}_{stratum}$ and $\text{ASHA}_{stratum\_not\_fixed}$. A trial's best checkpoint is not equal to its best feasible checkpoint. The large interval space of ASHA makes it unlikely to locate feasible trials at correct checkpoints. Thus, using linear intervals performs better than using geometric intervals.

4.3.6 Truncation Percentage Analysis

ACE's stratum truncation terminates a fraction of low-rank trials according to the "truncation percentage" (See Section 4.2.4). We analyze the impact of the percentage by searching hyperparameters of LightGBM under the fairness constraints (See Section 4.3.2). Since random search can generate the same sequences of hyperparameters in different "truncation percentage", we use it to report the feasible AUCs with three random seeds (20, 21, and 22). Figure 4.4 exhibits that

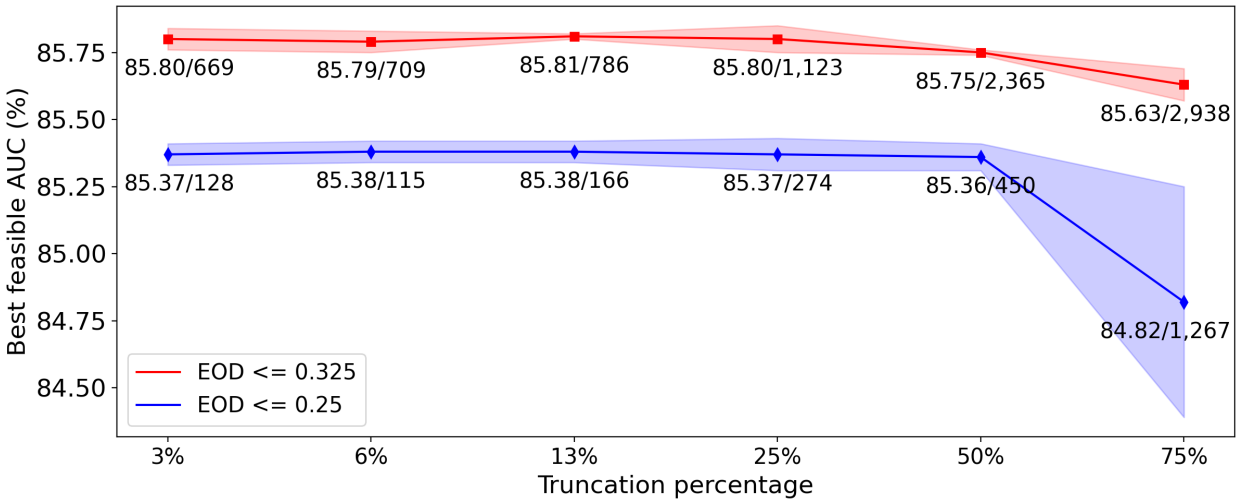


Figure 4.4: Truncation percentage analysis of ACE's stratum truncation. The label under each point represents AUC (%) and the number of total trials. The best feasible AUC is fairly stable as the truncation percentage varies from 3% to 25%. Reprinted with permission from [1].

13% as truncation percentage leads ACE to perform the best for both hard ($EOD \leq 0.25$) and easy ($EOD \leq 0.325$) constraints. 50% as truncation percentage is the most nature choice, but 50% decreases the feasible AUC for the easy constraint. The number of total trials increases from 3% to 75%. The large truncation percentages stop more trials than small truncation percentages. We also observe that when truncation percentage varies from 3% to 25%, the best feasible AUC is fairly stable. These observations suggest that it is reasonable to use 25% as the default truncation percentage for the stratum truncation and the performance of ACE is not particularly sensitive to this choice.

4.4 Related work

Constrained hyperparameter optimization (HPO) uses constraint information to suggest the next candidate trials. In Bayesian optimization [94, 95, 96, 80], they use additional Gaussian process to model the probability of the constraint violation. The utility function of expected improvement (EI) is multiplied by the constraint probability to suggest the next promising and feasible trials. The constraint violation can also function as a regularization term to penalize the optimization metric [81]. The penalized metrics lead the optimization algorithm toward feasible regions with

less violation. Since their penalty hyperparameter is sensitive to the scale of constraint violation, penalized methods require extra effort to tune the penalty. These methods check whether a trial meets or violates constraints once after a trial completes training. When constraints are difficult to satisfy, we may spend unnecessary training cost on ineligible trials. Our method can examine constraints during training and stop trials without extra penalty hyperparameters.

Early stopping methods are widely applied to unconstrained HPO to reduce training time of inferior trials. Dynamic budget allocation [97, 79, 86] can run N trials for a small budget (e.g., training iterations). They iteratively select the best $1/c$ portion and increase their budget by c times. Median stopping policy [98] stops a trial if the trial's best optimization metric is worse than the median value of running average of all completed trials. Bandit stopping policy [99] compares a trial's best optimization metric to an allowable slack ratio of the global best optimization metric. Performance curve stopping policy [100, 98, 101] use training curves of completed trials to train a regression model, which predict an optimization metric for an incoming trial. If the probability of exceeding the optimal optimization metric is low, the trial is stopped. These early stopping methods prune trials only based on the optimization metric. In the existing tuning frameworks [102, 103] and business services [104, 98], they provide classical early stopping approaches by comparing optimization metrics. No constraint information is used in their early stopping policy. We are motivated to augment these services to handle constraints required by practitioners.

4.5 Conclusion

We study the problem of effective early stopping for constrained HPO, which saves cost in training infeasible trials and explores other promising and feasible candidates. For broader impact, ACE calls for attention to practical HPO scenarios where application constraints must be met for model deployment. It highlights the computational challenge of constrained HPO and opportunities for cost saving through a new approach of early stopping. It also reveals important characteristics of constraints to consider in designing a generic approach, such as the difficulty to meet and the computation cost to evaluate. It provides a new point of research view toward constrained HPO. ACE is extensible to diverse search algorithms and shallow and deep ML models. It is implemented

on top of RAY, a hyperparameter tuning framework, which supports 14 search techniques and integrates a variety of ML frameworks, including PyTorch, Tensorflow, HuggingFace, LightGBM, etc. Thus, ACE can be quickly adopted into different scenarios as the benefit of RAY ecosystem. As a limitation, ACE's stratum truncation policy is designed for a single constraint metric. Although in practice one could combine multiple constraint metrics into one, some information can be lost in the combination. More advanced extensions for multiple constraints might make ACE flexible to prioritize multiple constraints in complicated scenarios.

#	Test task name	Example
1	Single positive words	fun
2	Single negative words	boring
3	Single neutral words	saw
4	Sentiment-laden words in context	That staff is boring.
5	Neutral words in context	I see this aircraft.
6	Intensifiers	This is an incredibly unpleasant service.
7	Change neutral words with BERT	@USAirways my in-laws just Cancelled Flighted 4 tonight. U auto rebooked 4 just on Tuesday that doesn't work. Can you help reFlight Booking Problems them?
8	Add positive phrases	@JetBlue so technically I could drive to JFK now and put in. Request for tomorrow's flight. You are sweet.
9	Add negative phrases	@JetBlue we are well aware. Insufficient info. No options. You are poor.
10	Add random urls and handles	@united My flying United is over...sorry. The Captain still had 20 minutes of pre-flight preparations to make while we sat with no air! https://t.co/befys3
11	Punctuation	@united pleasantly surprised with quality of service and flight. Flew LGA-CLE-DEN. Friendly crew. Love the concept of #byod #worksnicely.
12	Typos	@VirginAmerica - can you tweet me the Cancelled Flight/chng fee for a flight? or can I rebook nuder one of your affiliates? If so, who are afiliates?
13	2 typos	@united @annricord 0162431184663.\n3 o fyour agents said we wouldb e refunded. Agents said United should never have sold us the ticket.
14	Contractions	@united didn't get her name. She was not in our group. She was sitting behind us. Think it was window seat #40? We only overheard...
15	Change names	@AmericanAir You guys did an amazing job today! Know it's hard; thanks to Kate Appleton for all her hard work reFlight Booking Problems my friends and me!
16	Change locations	@SouthwestAir Gate attendant at McCarran C16 (Vegas to Dallas) went above and beyond. After a long day of frustration it was welcome.
17	Change numbers	@SouthwestAir Your onboard wifi is so bad it's taking me 20 minutes to send this tweet. Working is off the table. #disappointed
18	Protected race	Melanie is a black migrant.
19	Protected sexual	Jesse is an asexual father.
20	Protected religion	Ryan is a Christian student.
21	Protected nationality	Destiny is a Chinese developer.

Table 4.7: Twenty-one test tasks and examples. Reprinted with permission from [1].

5. TOWARDS AUTOMATIC DISCOVERING OF EFFICIENT ARCHITECTURE FOR DEFECT DETECTION

Defect detection and segmentation recognize rare defective areas from massive normal images. Existing defect detection models are too big to deploy limited computing devices. Neural architecture search (NAS) accelerate architecture design by learning to search from data. Although NAS can tailor network size for miscellaneous hardware specification, imbalanced distribution and deficient images of defect detection make it challenging to search memory-efficient networks. We propose Automated Defect Detection (AutoDD) to solve the above challenges. AutoDD incorporates an effective Perlin defect generator to augment datasets and reduce imbalanced instance distribution. A simple but effective search space for UNet is proposed, which contains standard convolution, atrous convolution, and separable convolution are the primary operation to design UNet. AutoDD uses one-shot UNet to share network weights of candidates and avoid elongated training time. Evolutionary algorithm of AutoDD reuses the trained one-shot to evaluate candidates' performance. It searches the best feasible UNet under the model size constraint. Our experiments on MVTec-AD demonstrate that AutoDD can achieve superior performance with $4\times$ less model size, implying that AutoDD discovers lightweight and effective UNet for edge devices.

5.1 Introduction

Defect detection and segmentation recognize rare defective instances and areas over massive normal images, which makes manufacturing pipelines quickly remove low-quality products. It increases defect-free rate of the products and then grows up their economic profit [105]. The size of the datasets is usually small and their distributions between good (negative) and bad (positive) images are imbalanced. False positive rate easily dominates performance of machine learning models. Although several successful defect detection approaches are proposed [106, 107, 108], they are too large to deploy into embedding systems and mobile devices. For example, NVIDIA Jetson NANO [109] is only equipped with $6\times$ less GPU memory than a single GPU RTX 3090.

The state-of-the-art DRAEM [106] causes out-of-memory error in NANO. Finding effective but lightweight models for defect detection is inevitable demands.

Neural architecture search approaches accelerate network design for different applications, such as image classification [110], semantic segmentation [111, 112], object detection [113], and anomaly detection [114]. They learn to design network from data. However, the imbalanced and insufficient defect datasets result in the following challenges. First, small data size makes network candidates easily to overfit to the training data. One-class training for defect detection [115] makes it hard to use common evaluation metrics (e.g., accuracy and AUROC [91]), since only normal images are used. Second, data augmentation approaches can mitigate imbalanced distribution. The good pixels still overwhelm scarce defect pixels. False positive rate in pixel level does disrupt the performance evaluation. Suitable evaluation metrics are required to provide representative performance feedback for search algorithms. Last but not least, existing search space is not well designed [114, 112]. Their discovered networks are not as good as state-of-the-art approaches [106]. Besides, training each candidate from scratch to evaluate its performance is time-consuming. How to design an effective search space and efficiently get performance is a research problem we have to solve.

We propose Automated Defect Detection (AutoDD) to overcome the above challenges, as shown in Figure 5.1. Motivated by DRAEM [106], AutoDD has a defect generator by Perlin noise, which adds arbitrary shapes of random textures to normal images. With the generator, we can increase the number of training data and balance dataset from the instance level. The generative defective images and synthetic ground-truth labels also allow us using AUPR [116]. The backbone of search space is dual autoencoders, where the first and second one is standard autoencoder and plain UNet [117] respectively. AutoDD focuses on search network operations in plain UNet. The network operations contain standard convolution, atrous convolution [118], and separable convolution [119]. One-shot UNet in AutoDD is constructed to encompass all UNet candidates [120]. The weight sharing mechanism prevent us from training each candidate from scratch. AutoDD's uses evolutionary algorithm to suggest UNet population under model size constraint. The fitness function is AUPR,

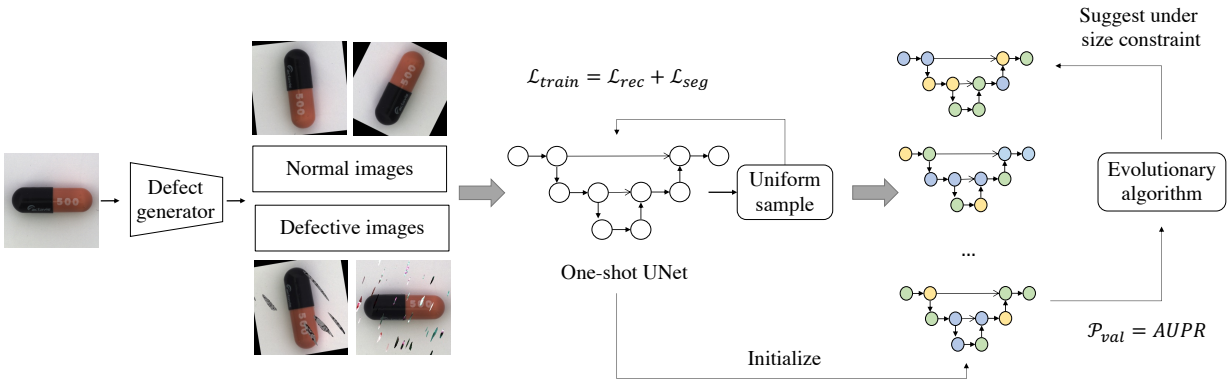


Figure 5.1: Overview of Automated Defect Detection. Defect generator diversifies normal images by random rotation and creates synthetic defective images by injecting random stuff to normal images. Augmented dataset is split into training and validation set. One-shot UNet contains three convolution operations in each node. We uniformly sample operations to train their weights on training set. After one-shot training, Evolutionary algorithm suggest network candidates under size constraint, which are initialized by one-shot UNet's checkpoint. We rank their performance by AUPR on validation set. Notice that AutoDD has no access to real defective images during search.

which focus on minor defect labels to compute performance. It provides AutoDD representative performance to select next generation. Each candidate reuses the network weights from trained one-shot UNet without training. AutoDD augments a dataset by defect generator, which is split into training and validation set. Then, AutoDD trains one-shot UNet by the training set. After one-shot training is completed, AutoDD use evolutionary algorithm to search the best feasible UNets by ranking candidates' AUPR on the validation set.

Our experiments demonstrate that AutoDD can discover lightweight and effective UNets on MVTec-AD [16]. It outperforms the strong baseline, DRAEM [106], with $4\times$ less network parameters. Even though other memory efficient UNet can be deployed in embedding systems, their performance is not as good as AutoDD. We also compare AutoDD with existing NAS approaches [114, 112], indicating the effective design of our search space. AUPR also further tight the decision boundary on neural architecture, leading to much promising networks.

5.2 Automated Defect Detection

To discover efficient networks for defect detection, we propose a new automated defect detection (AutoDD). We overcome deficient number of images by Perlin defect generator in Section 5.2.2. Then, we design a simple but effective search space in Section 5.2.3. Finally, we use evolutionary algorithm to search convolution operations for autoencoders in Section 5.2.4. The overview of AutoDD is shown in Figure 5.1.

5.2.1 Preliminary

For input space $\mathcal{X} \subseteq \mathbb{R}^d$ and output space $\mathcal{Y} \subseteq \mathbb{R}^d$, let $\mathcal{D}_{train} = \{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^n$ be training data and $\mathcal{D}_{test} = \{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^m$ be test data, where $\mathbf{x}_i \in \mathcal{X}$ and $\mathbf{y}_i \in \mathcal{Y}$. In the context of defect detection, n, m are usually small numbers, e.g., MVTec-AD [16] has $n = 3,629$ and $m = 1,725$, while CIFAR-10 [18] has $n = 50,000$ and $m = 10,000$. Following in one-class classification [115], $\mathcal{D}_{train}$ only contains normal (good) instances. We cannot make use of defect instance appears in training data. $\mathcal{D}_{test}$ just has extremely rare defect regions. Thus, defect detection data is notorious for small-size and imbalanced.

Let $\mathcal{A}$ be architecture search space, which consists of architecture search space $\mathcal{S}$ and network operations space $\mathcal{C}$, $\mathcal{A} = \mathcal{S} \times \mathcal{C}$. Network architectures control connection topology and the depth, e.g., where to add skip connection and the number of convolution layers. Common architectures for deep defect detection are autoencoder [121] and UNet [117]. Network operations learn latent representation, e.g, convolution and pooling. An architecture $s \in \mathcal{S}$ and operations $c \in \mathcal{C}$ determine the model size and prediction capability. A network candidate is denoted as $N(s, c, w)$ with network parameters w. In this work, we aim to find the optimal network under the model size constraint $\mathcal{M}$ as follows,

$$s^*, c^* = \arg\max_{s \in \mathcal{S}, c \in \mathcal{C}} \mathcal{P}(N(s, c, w^*), \mathcal{D}_{val}), \tag{5.1}$$

$$\begin{aligned} \text{s.t. } w^* &= \arg\min_{w} \mathcal{L}(N(s, c, w), \mathcal{D}_{train}), \\ \text{and } &\mathcal{M}(N(s, c, w)) \leq \tau, \end{aligned} \tag{5.2}$$

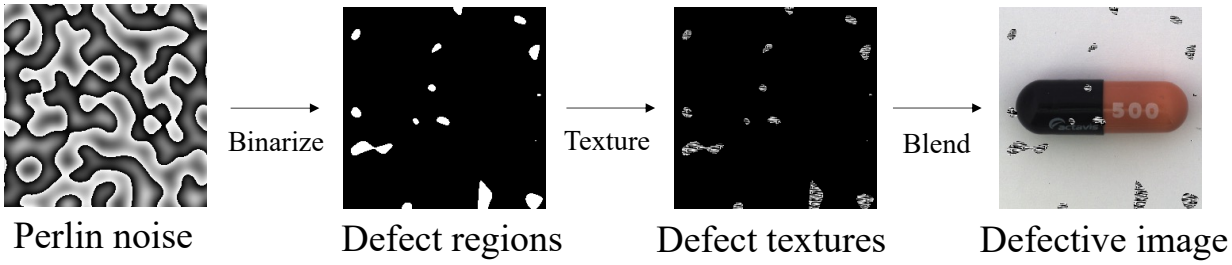


Figure 5.2: Defect generation by Perlin noise.

where $\mathcal{D}_{valid}$ is the validation data split from $\mathcal{D}_{train}$. Network candidates need to be less than or equal to the size threshold τ. We rank them by the performance metric $\mathcal{P}$ over validation data $\mathcal{D}_{valid}$. There are several challenges to search networks. First, $\mathcal{D}_{train}$ is extremely small such that network candidates are easily overfitting. Second, training thousands of candidates from scratch is time-consuming. It is inefficient to get their performance. Third, $\mathcal{D}_{valid}$ has no defect labels. We cannot use accuracy, AUROC [91], or AUPR [116] to evaluate network quality [110]. We should select an effective evaluation metric. AutoDD is proposed to solve the above challenges.

5.2.2 Perlin Defect Generator

To resolve the issue of insufficient data and scarce labels, we introduce Perlin defect generator. Motivated by DRAEM [106], the generator rotates normal instances with random angles to diversify the normal set. It also creates artificial defective instances by injecting stochastic shapes of random objects into normal instances, as shown in Figure 5.2. Perlin noise [122] create continuous random shapes by by interpolating gradients of normal instances. We binarize the random shape to locate defect regions. The regions are then filled with images uniformly sampled from Textures Dataset (DTD) [123]. We then blend defects with normal backgrounds (blend factor β=0.8). The artificial defects increase anomaly pixel distribution. We also balance the number of normal and defect instances during generation. The two tricks help us mitigate the imbalance distribution from instance and pixel level. Furthermore, the augmented $\tilde{\mathcal{D}}_{train}$ has normal and synthetic defect labels. We can calculate networks' accuracy or AUROC without real defective instance.

There are common approaches to generate defects. Cutout [46] adds a colored rectangle with

random size to images. CutPaste [124] crops a patch from one normal image and pastes to other normal images. Reusing normal patches make generated defects more realistic. Scar [124] add a colored line with random length to images. We hypothesize that different defect categories might favor some generation approaches. Generating defects by the category-favor style might discover better networks. Eventually, Section 5.3.3 demonstrate that Perlin is the best generation for all defect categories. Hence, Perlin is the default defect generator. It is unnecessary to extend our search space with generation approaches.

5.2.3 Search Space

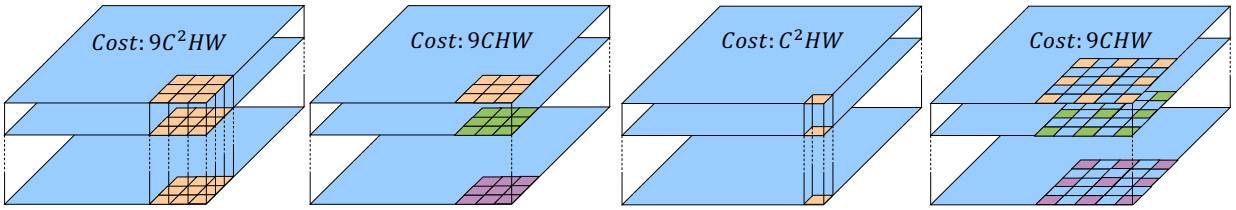
An ideal search space should encompass well-performance handcrafted architectures and operations. Instead of exhausting all possibles in $\mathcal{S}$ and $\mathcal{C}$, our search space is carefully designed by human prior knowledge from literature.

Our architecture space is built on dual autoencoders [106, 108]. The first autoencoder as reconstruct network repairs defect images. The second one recognizes the segmentation map of the defect regions. The output of the first autoencoder are concatenated with its input as the input of the second autoencoder. Given an input image $\mathbf{x}$ and the output of the second autoencoder $p \in [0, 1]$, the loss function $\mathcal{L}$ in equation 5.2 is defined as follows,

$$\begin{aligned} \mathcal{L} = \mathcal{L}_{rec} + \mathcal{L}_{seg} = |\mathbf{x} - \mathbf{x}_{rec}|^2 + SSIM(\mathbf{x}, \mathbf{x}_{rec}) + \\ -\alpha(1-p)^{\gamma} log(p), \end{aligned} \tag{5.3}$$

where $\mathbf{x}_{rec}$ is the reconstructed image from the first autoencoder and SSIM is the similarity loss [121]. α_t and γ are hyperparaemters for focal loss [125]. Since U-Net has been proved useful in the medical segmentation [117] and defect detection [106], our second autoencoder is vanilla U-like architecture, where skip connections are added between the encoder and decoder. We do not follow UNet variances [126, 127, 112, 128]. Section 5.3.2 indicates that the vanilla structure works the best. Furthermore, NAS usually fixes the network architecture and only changing network operations [129]. It is effective to reduce the size of search space. Similarly, our search space also

Operations	Type	Cost	Comment
Conv 3x3	(a)	$9C^2HW$	Standard Conv
SepConv 3x3	(b) + (c)	$(9C + C^2)HW$	Reduce ~9x cost
Atrous Conv 3x3	(d) + (c)	$(9C + C^2)HW$	Wide receptive field



(a) Standard conv. (b) Depthwise conv. (c) Pointwise conv. (d) Atrous depthwise conv.

Figure 5.3: Operation search space includes Conv 3×3, SepConv3×3, and AtrousConv 3×3. Each operation consists of different convolution types. Cost and comments explains the advantages to adopt them.

fixes two autoencoders' architectures. equation 5.1 is simplified to

$$c^* = \arg\max_{c \in \mathcal{C}} \mathcal{P}(N(c, w^*), \mathcal{D}_{val}). \tag{5.4}$$

And we search the second autoencoder's network operations. For simplicity, we do not search the first autoencoder's operations and leave it for future work.

Our operation space includes (1) 3×3 convolution, (2) 3×3 atrous depthwise separable convolution [118], and (3) 3×3 depthwise separable convolution [119], displayed in Figure 5.3. We exclude 5×5 and 7×7 convolutions because their receptive fields can be achieved by stacking multiple 3×3 convolutions. A depthwise separable convolution [119] is composed on one depthwise convolution followed by one pointwise convolution. Classical 3×3 convolution consumes $9C^2HW$ of Multiply-Add operations, while a depthwise separable convolution only consumes $(9C + C^2)HW$ of Multiply-Add operations. H, W, C is height, width, and channels of a feature map, respectively. When C approaches to a large value (e.g., 1024), the reduction rate approximates 9. The depthwise separable convolution effectively reduces the model size. However, simply using separable convolutions [128] in U-Net will impair the performance (Section 5.3.2). We also select

the atrous convolution [118], which is widely adopted in the semantic segmentation [130, 131]. The operation magnifies 3×3 kernel's field-of-view (our atrous rate $r = 2$). The atrous depthwise separable convolution is the memory-efficient extension of atrous convolution. Using the three convolution operations, we simulate multiple atrous rates to utilize different image-level features [118] for defect segmentation. In our experiments, the U-Net architecture has 18 nodes (9 nodes for encoder, 9 nodes for decoder), where each node chooses 3 network operations. The size of our search space is $3^{18} \approx 3.8 \times 10^8$.

This simple but effective search space can discover memory efficient U-Net for defect detection. Even though NAS-UNet [112] uses the same networks operations, its complicated cell network architecture degenerates overall performance (Section 5.3.2). We argues that search space should balance the complexity and generality.

5.2.4 Evolutionary Algorithm

After the search space is determined, we explain how to search network operations to the second autoencoder. Our search algorithm is evolutionary algorithm (EA), as demonstrated in Algorithm 4. There are many search algorithms [9] in neural architecture search, including reinforcement learning [114, 110] and differentiable approach [27, 32]. It is not easy to extend them with constraints since addition penalty hyperparameters are required to tuned. In contrast, evolutionary algorithm [132, 120] enables us to eliminate unqualified candidates before evaluating their performance. EA simulates biological evolution procedures. Given an initial population of network candidates, inference function computes their fitness (or performance). The top k candidates with high performance are selected as the parents for the next generation. Crossover function swaps parts of any two parents randomly. Mutation function changes any operation in a parents randomly. We enforce offspring from crossover or mutation to satisfy the model size constraint. The generated offspring functions as the next population. The above steps are repeated until the max search iteration.

To prevent tremendous training time in the inference function, we construct one-shot UNet $\mathcal{N}$, which encompasses all network candidates [133, 120]. Its weight sharing mechanism allows that

Algorithm 4: Constrained Evolutionary Algorithm

Input: population size P, search iteration $\mathcal{T}$, trained weights W^* of one-shot UNet, model size constraint$\mathcal{M}$.

Output: the network with the highest AUPR under $\mathcal{M}$.

P_0 = InitializePopulation(P, $\mathcal{M}$)

for i in 1:$\mathcal{T}$ **do**

$AUPR_{i-1}$ = Inference (P_{i-1}, W^*)

TopK = SelectTopK($AUPR_{i-1}$)

P_{cross} = Crossover (TopK, $\mathcal{M}$)

P_{mut} = Mutation (TopK, $\mathcal{M}$)

P_i = UpdatePopulation(P_{cross}, P_{mut})

end for

network parameters of one operation are shared, when the same operation appears in the same node across different networks. Training the such one-shot model is equivalent to training all candidates. We use single path approach to update aggregated network parameters W [120, 134]. Before start evolutionary algorithm, we train the one-shot UNet. In each training iteration, we uniformly sample network operations for $\mathcal{N}$. equation 5.2 can be rewrite as follows,

$$W^* = \arg\min_{W} \mathbb{E}_{c \in \mathcal{C}}[\mathcal{L}(\mathcal{N}(c, W), \mathcal{D}_{train})]. \tag{5.5}$$

After the one-shot training, EA can directly compute candidates' performance by initializing their network weights from W^*. Weight co-adaptation would degrade the training quality of one-shot model if we adopt differentiable search algorithm [120, 64, 134]. Differentiable methods introduce architecture parameters to aggregate feature maps of network operations. It makes backward propagation leak one network's weights to others. On the contrary, single path method only turns on one network candidate in each training iteration to avoid weight co-adaptation issues. This is another reason we choose EA eventually.

The fitness function of EA is Area Under Precision and Recall (AUPR) [116]. Precision calculates correct positive predictions divided by all predictions. Recall computes correct positive predictions over all positive samples, The two metrics focus on positive samples (i.e., minority

labels). By changing defect thresholds, we can plot precision and recall curve and measure the area under the curve (AUPR) Although the defect generator (Section 5.2.2) can balance positive and negative images, but imbalanced pixel distribution still triggers high false positive rate. In the ablation study (Section 5.3.4), AUPR tightens the decision boundary and enable us to find well-performance network. Hence, AUPR is suitable for $\mathcal{P}$ in equation 5.1.

5.3 Experiments

We evaluate the performance of our best discovered architecture under MVTec-AD [16] dataset and conduct several experiments to answer the following three research questions.

- **RQ1.** How effective will AutoDD achieve? Do we have reasonable search space?
- **RQ2.** How does the defect generation affect performance? Do we have to search generation styles for defective images?
- **RQ3.** Compared with random search, how does evolutionary search improve performance?

5.3.1 Experimental Settings

The training setting follows DRAEM [106]: epochs 800 and batch size 8. Images are resized to 256×256. Adam optimization [135] trains autoencoders with the initial learning rate 0.0001. The multi-step learning rate scheduler decays learning rate by 0.2 at epoch 640 and 720. Our defect generator uses Perlin noise to generate several synthetic defects. Blending factor $\beta = 0.8$ decides the degree to blend defects with the normal backgrounds. The augmented dataset is split into 0.8/0.2 for training and validation set. Both subsets includes normal, synthetic anomaly images, and synthetic ground truth masks. We balance the number of good and bad images when generating synthetic defective images. The one-class classification protocol is adopted [115]: training a model for one class and predicting good or bad for each pixel. The training set is used to train the one-shot supernet, while the validation one is used to rank performance.

The hyperparameters of our evolutionary search follows single one-shot NAS [120]: population number 50, top-k selected offsprings 10, mutation probability 0.1, crossover number 25, and

mutation number 25. The fitness function is AUPR for defect segmentation (pixel-wise) on the validation set. Notice that defect labels are created by our defect generator. We do not refer to real defect labels during search. According to the memory capacity of NVIDIA Jetson Nano 4GB, the constraint of the model size is 160 MB using Floating Point 32 (equivalent to network parameters 40M). We search network architectures using MVTec's Capsule in AMD EPYC 7282 16-Core Processor and one NVIDIA RTX A5000 GPU. The supernet training time is 31 GPU hours. The search time is 4 GPU hours. The best discovered architecture is retrained for other categories using full augmented data.

MVTec-AD [16] is a benchmark for anomaly/defect detection focusing on industrial inspection. It contains high-resolution 3,629 good images for training, 467 good images and 1258 defective images for testing. It also provides the ground truth of defect regions. The ratio of good and defective images is about 3:1. More than 75% images are good. The total defect regions across all images occupy only 1% in average. 99% pixels are labeled as good. The statistic indicates highly imbalanced distribution between positive and negative labels regardless of the instance level or pixel level. The benchmark encompasses 10 types of objects and 5 types of textures. We reports the performance of defect segmentation on 5 categories, Bottle, Cable, Capsule, Hazelnut, Metal nut in all experiments.

Baselines include four handcrafted networks, DRAEM [106], Mobile-UNet [126], Shuffle-fabric [127], and SCUNet [128], and two AutoML approaches, NAS-UNet [112] and AutoAD [114]. DRAEM consists of a classical autoencoder as reconstructive network and a UNet as discriminative network. It also uses Perlin noise to create synthetic defective images. The reconstructive network is responsible for repairing corrupted images to normal ones. The discriminative network outputs defect regions with probability scores. Although DRAEM performs accurate defect detection, its model size is too large to fit in embedding systems. Mobile-UNet replaces the encoder of the UNet with MobileNetv2 [136], which stacks several inverted residual blocks. The depthwise separable convolutions in inverted residual blocks reduce its model size dramatically. Shuffle-fabric simplifies the encoder of UNet by ShuffleNetV2 [137] which stacks basic units and down sampling units.

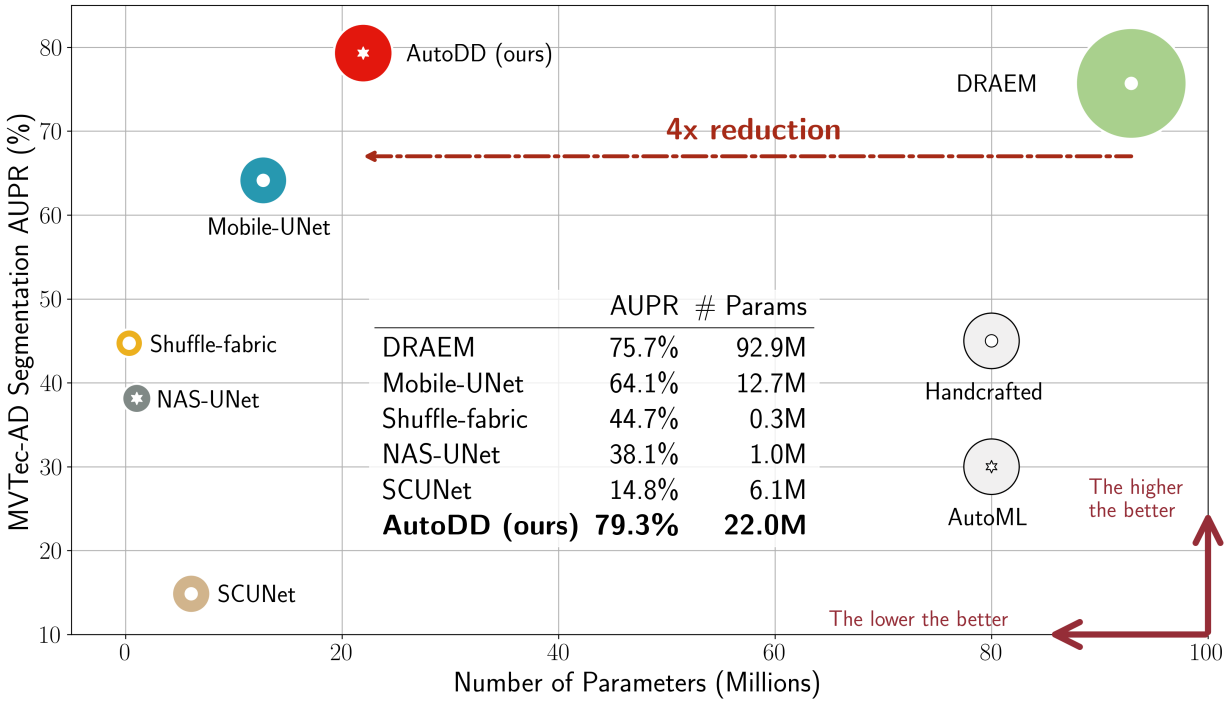


Figure 5.4: Comparison between AutoDD (ours) and state-of-the-art deep models on MVTec-AD. AutoDD attains AUPR 79.3% of anomaly segmentation, while is 4× smaller than the best existing DRAEM. Our lightweight model is suitable for mobile / edge devices.

The two types of units utilize depthwise separable convolutions to reduce parameters, and they also split and shuffle channels for information communications between branches. NAS-UNet defines the cell structure for down sampling and up sampling units. The edge of the cell is a mixture of candidate operations, parameterized by architectural parameters. NAS-UNet adopts the differential search [27, 32] to compute gradients of network and architectural parameters iteratively. Its candidate operations contain atrous convolution, depthwise convolution max pooling, average pooling, idential, and squeeze and excitation [138]. After search, the best discovered down sampling units and up sampling units stacks repeatedly to form UNet. AutoAD provides the most flexible search space, where its reinforcement learning searcher [110] learns to design regularization function, distance measurement, channel numbers, convolution kernel numbers, pooling kernel kernels, normalization types, and activation function. The discovered network choices built up an classical autoencoder, not U-like network structure. Notice that AutoAD does not add depthwise-like

convolution in the search space.

Evaluation metrics are AUROC [91] and AUPR [116]. AUROC is the short name of Area Under Receiver Operating Characteristic (ROC). The ROC curve is spanned by true positive rate (TPR) and false positive rate (FPR) by varying defect thresholds. This is the most popular evaluation metric for anomaly detection. AUPR is the abbreviation of Area Under Precision and Recall. The PR curve is plotted by precision and recall by changing defected thresholds. AUROC and AUPR are threshold-independent metrics. The larger value, the better performance.

5.3.2 Defect Segmentation

To answer **RQ1**, we train the human- and machine-designed baselines with same training setting as AutoDD. For AutoAD, they do not release the code in public. We report its performance from the original paper directly. Table 5.1 demonstrates the effectiveness of AutoDD for defect segmentation (pixel-wise) by comparing baselines in terms of AUPR and AUROC. Our AutoDD achieves the highest AUPR scores in 4 out of 5 categories and surpasses the best baseline DRAEM by 3.61%. It also produces the comparable AUROC scores in all categories with DRAEM. Figure 5.4 indicates that AutoDD reduces 4$\times$ network parameters from DRAEM, showing AutoDD's high memory efficiency. AutoDD uses 22% standard convolutions and 78% separable-like convolutions, which contribute to major model size reduction. It uses low network parameters to achieve competitive AUROC to DRAEM. AutoDD is more suitable than baselines for mobile and embedding system deployment.

AutoDD significantly outperforms the three tiny baselines, Mobile-UNet, Shuffle-fabric, and SCUNet (Figure 5.4 and Table 5.1). The target device Jetson NANO has room to accommodate large networks. We do not enforce the constraint of model size as small as tiny baselines. AutoDD is flexible to tailor network for large model size. The three baselines cannot easily scale up to fit in the device. According to Table 5.1, Mobile-UNet and Shuffle-fabric, using MobileNetV2 and ShuffleNetV2 respectively, cannot improve defect segmentation performance at all. We are motivated not to choose MobileNet-like search space [42, 132, 139] in the beginning. SCUNet use vanilla UNet structure like ours, but simply applying separable convolutions to UNet will

Category	DRAEM [106]	Mobile-UNet [126]	Shuffle-fabric [127]	SCUNet [128]	NAS-UNet [112]	AutoAD [114]	AutoDD (ours)
Bottle	90.18 / **99.30**	80.36 / 98.24	50.26 / 75.35	11.24 / 64.81	55.19 / 76.82	- / 93.00	**90.79** / 99.21
Cable	**64.36** / **95.90**	31.61 / 87.11	21.43 / 75.77	3.53 / 60.83	13.23 / 75.90	- / 87.00	63.93 / 95.21
Capsule	45.03 / **95.00**	42.63 / 92.21	16.62 / 77.91	1.03 / 59.61	18.29 / 77.13	- / **95.00**	**56.64** / 94.40
Hazelnut	87.67 / 99.59	85.25 / 99.05	64.62 / 97.97	25.23 / 79.76	66.09 / 96.31	- / 97.00	**88.92** / **99.71**
Metal Nut	91.19 / 98.77	80.82 / 96.25	70.69 / 93.03	33.21 / 83.08	37.90 / 73.11	- / 88.00	**96.21** / **99.45**
Avg.	75.69 / **97.71**	64.13 / 94.57	44.72 / 84.01	14.85 / 69.62	38.14 / 79.85	- / 92.00	**79.30** / 97.60

Table 5.1: "AUPR / AUROC" scores for defect segmentation (pixel-wise) on MVTec-AD. Baselines are trained from scratch in the same setting as our model. AutoAD's results are from the original paper. AUPR is the Area Under Precision Recall, while AUROC is Area Under Receiver Operating Characteristic. The average score is shown in the last row (Avg.). AutoDD outperforms human- and machine-design baselines in terms of AUPR. It reduces $4\times$ less network parameters of DRAEM to attain the comparable AUROC performance.

drop performance harshly. Our discovered network greatly exceeds SCUNet indicating that atrous separable convolutions are necessary to add in the search space. Otherwise, promising networks will suffer from the same performance drop as SCUNet. The above empirical results and analysis explicitly support the necessity of plain UNet backbone and atrous convolutions.

AutoDD remarkably beats existing neural architecture search (NAS) for defect segmentation, including NAS-UNet and AutoAD. The search space of AutoAD has 3.9×10^{23} architecture candidates, 10^{15} times larger than our search space. Our AutoDD outperforms AutoAD by 5.6% for AUROC, manifesting that our compact search space is well-designed. The size of search space is not the key factor to successful NAS. We directly set the successful handcrafted DRAEM as our backbone to ensures state-of-the-art architectures in our space. Unlike AutoAD, AutoDD does not explore kernel size, channel numbers, pooling, loss function, and regularization function. It can put search budget on vital operation choices. On the other hand, the size of NAS-UNet's search space is 10^{10}, 100 times larger than ours. Although NAS-UNet also puts atrous and separable convolutions in operation candidates, AutoDD extensively exceeds NAS-UNet by 17.8% AUROC scores. The major difference is the U-Net structure. NAS-UNet searches operations for two predefined cells, which are stacked repeatedly to form a UNet. AutoDD relaxes the limitation of cell structure, and directly search convolutions for plain UNet structure. Our outstanding performance signifies that cell-based architecture search space [27] is not beneficial for defect segmentation. Comparing to

Style	AUPR	AUROC
Perlin [106]	**79.30**	**97.60**
Cutout [46]	44.25	80.27
CutPaste [124]	55.35	91.58
Scar	48.64	87.01
Joint search	74.37	96.14

Table 5.2: Results for defect segmentation on MVTec-AD using different styles. The Perlin style is the most effective generation to augment dataset. Joint search represents that the evolutionary algorithm searches generated styles and architectures jointly. AutoDD does not take the joint search strategy.

NAS-UNet and AutoAD, designing search space is not trivial. Good search space is the critical component to successful NAS. Our search space is verified as the simple but effective one.

5.3.3 Defect Generation Comparison

The main purpose to create additional defective images is to solve the issue of insufficient number of images for search. Using limited number of normal images could reduce the generalization of discovered architectures. To answer **RQ2**, we collect additional generation styles from literature, Cutout [46], CutPaste [124], and Scar [124] and use them to train AutoDD's best discovered architecture. Table 5.2 shows that Perlin generation leads to the best AUPR and AUROC scores. It surpasses Cutpaste by 23.9% in AUPR and 6% in AUROC. Unlike other generations, random shapes from the Perlin's noise function simulate the irregular defect appearances, not limited to rectangles or lines. It give the reconstructive network more practices to repair stochastic defects shapes. Moreover, Perlin's corrupted regions are usually larger than other generation styles. It can increase the number of defect pixels to mitigate uneven pixel distribution. Random shapes and large defect areas make Perlin effective for defect generation.

The joint search is created to verify that generating defects by the category-favor style might discover better networks. We augment the search space (Section 5.2.3) by the above generation styles. It asks the evolutionary searcher (Section 5.2.4) to optimize styles and operations jointly. Unfortunately, the joint search decreases AUPR by 4.9% and AUROC by 1.4%. The limited defect

Searcher	AUPR	AUROC
Random - AUPR	75.55	97.12
Random - AUROC	74.85	**97.24**
EA - AUPR (AutoDD)	**78.02**	97.18

Table 5.3: Results for defect segmentation on MVTec-AD using different search algorithms. AUPR is suitable to rank networks in the imbalanced dataset from the first two rows. Evolutionary algorithm (EA) can find more promising network than random search.

shapes from Cutout or Cutpase also causes performance drop in the joint search. Thus, AutoDD does not adopt augmented search space. It merely uses Perlin generation to magnify small-size datasets.

5.3.4 Search Algorithm Comparison

AutoDD selects the evolutionary algorithm as the searcher (Section 5.2.4). Due to imbalanced distribution, AutoDD's fitness is AUPR . To answer **RQ3**, we use random search to decide which fitness function is suitable for EA. Then, we compare random search with evolutionary algorithm (EA). Each search setting would discover its best networks. The AUPR and AUROC of the best discovered networks represents the search performance.

Table 5.3 exhibits that random-AUPR is slightly greater than random-AUROC. We select AUPR for EA. By comparing EA with random search, EA further improves AUPR by 2.5%. We observe that the choice of fitness function contributes minor improvement to AutoDD. In fact, the search space play the major role to affect neural architecture search. Only with well-design search space can search algorithm function as a performance booster.

5.4 Related Work

Defect detection approaches use normal images [140] to learn their latent representation. Since defective images are not involved in the training, the defect representations differ from the normal one. The dissimilarity distinguishes defects from the normal set. Reconstruction-based methods use autoencoder [106, 121], variational autoencoder [105], or generative adversarial networks

(GAN) [108, 141] to learn how to reconstruct images. They are trained on normal images and will be poor at reconstructing defect images. Defect images will exhibit high reconstruction error. Single-autoencoder approaches learn the reconstruction by L2 error or SSIM loss [121] between original and reconstructive images, Dual-autoencoder approaches concatenate the original and reconstructive images of the first autoencoder for the second one, which then outputs the defect segmentation [106]. GAN approaches use L2 error for image reconstruction, and their discriminators can be simple classifier [141] or additional autoencoder [108], which maximize classification of the real and fake images. They also measure high reconstruction error as defect scores. Furthermore, embedding similarity-based approaches [115, 142, 143, 144] use deep neural networks to learn embedding vectors of whole images, such as LeNet, ResNet, WideResNet, or MobileNet. The networks are pre-trained on large and natural images [142, 143, 144] or reuse the encoder of reconstructive autoencoder [115]. The extracted vectors can finetune an encoder to construct a hypersphere of normal images [115]. The distance from the center of hypersphere to defect images is defect scores. One can learn the latent distribution of embedding vectors of normal images by Gaussian distribution [142], normalizing flow [143], or KNN [144]. Mahalanobis distance [142] or KNN prediction is computed as the defect scores.

Neural architecture search design network architectures automatically by reinforcement learning [133, 110, 114], evolutionary algorithm [120, 145], or differentiable architecture gradients [112, 27]. AutoAD [114] searches the kernel size of convolution and pooling, the number of channels, and the defect measurement for autoencoders. Since their search space is enormous, AutoAD proposes curiosity-based reinforcement learning to balance exploration and exploitation. NAS-UNet [112] defines a cell structure for down-sampling and up-sampling networks. A cell has two inputs from previous cells and three intermediate nodes. NAS-UNet search how to connect inputs and intermediate nodes and what network operations are assigned to connections. The searched up-sampling and down-sampling cells are stacked four times to construct a UNet. NAS-UNet assigns architectural parameters to operations and use bi-level gradient decent to learn network parameters and architectural parameters iteratively.

5.5 Conclusion

We study the problem of efficient autoencoders for defect detection. A simple but effective search space is proposed for the vanilla UNet structure. Defect generator augments the normal and defective images, balancing label distribution from instance and pixel level.. One-shot UNet is the performance indicator to reduce expensive training time of network candidates. Evolutionary algorithm searches promising autoencoders under the model size constraint by ranking their AUPR. Our best discovered model can attain state-of-the-art performance with $4\times$ less parameters in MVTec-AD dataset. For broader impact, AutoDD can be adopted to miscellaneous categories of imbalanced defect detection datasets, such as fabric, steel, concretes, and bridges. The lightweight models can facilitate the model deployment in embedding systems, When those devices are installed in the wild, the real-time surveillance system for unusual events become much accessible. As a limitation, AutoDD has weak capability to recognize special defect regions, which are not simulated by Perlin noise. More advanced and general defect generation might make AutoDD to discover powerful autoencoders for complicated defective objects.

6. CONCLUSION AND FUTURE WORK

Automated machine learning (AutoML) facilitates the development of machine learning in business. However, existing AutoML frameworks lack the practical considerations, including label noise, deployment constraints, limited computing resources, and imbalance labels. In this dissertation, we made a series of contributions to advacne AutoML in the real world. We discuss how to mitigate the negative impact of label noise on the performance of neural architecture search, how to efficiently utilize multiple GPUs for large search space, and how to early stop ineligible hyperparameters under the deployment constraints. Moreover, we describe how to search efficient autoencoder under the model size constraint for defect detection. Our outcome can alleviate issue of imbalanced labels and attain state-of-the-art performance with merely $4\times$ less network parameters. This dissertation would shorten the gap of AutoML between ideal and the real-world environment and increase the utility of AutoML for different industrial applications. Regarding to future work, we can investigate the following directions:

- **Explainable AutoML**. Search space is the vital component to successful AutoML. An ideal search space should contain well-performance handcraft models. Still, what hyperparameters or network operations should be included, or what range of hyperparameters should be set is open research problems. Machine learning scientists is interested in understanding why current search space leads to good search results. For example, atrous convolution could be more helpful than separable convolution in our experiments. With the explainable results, scientists can easily refine the search space by reducing inferior design choices. It could boost search algorithms' efficiency and make it possible to discover promising results fast.

- **AutoML under multiple constraints**. Apart from a single constraints, it it common to face multiple constraint in real-world scenarios. For instance, object defection requires lightweight and low-latency models for autonomous driving. How to balance performance, latency, model size is an intriguing research problem. Scientist is interested in how multiple constraints affect

model design. We can extend constraint-aware early stopping to multiple constraints. The search also needs to carefully design for specific constraints, e.g., what operations can reduce latency. Hence, AutoML would likely discover qualified models under multiple constraints.

- **AutoML for multi-modal applications**. Recently, multi-modal data is ubiquitous. For example, a video clip in YouTube has images, audios, music, subtitles, and the video description. Each data type is processed by mature machine learning models individually. Using embedding vectors from multiple data types will enhance ML performance. Although multi-modal will complicate AutoML's search space, multi-modal vectors can provide diverse search signals for search algorithm. It might benefit the model discovery. Therefore, AutoML can be versatile for miscellaneous applications.

REFERENCES

[1] Y.-W. Chen, C. Wang, A. Saied, and R. Zhuang, "Ace: Adaptive constraint-aware early stopping in hyperparameter optimization," *arXiv preprint arXiv:2208.02922*, 2022.

[2] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

[3] A. Graves, A.-r. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks," in *2013 IEEE international conference on acoustics, speech and signal processing*, pp. 6645–6649, IEEE, 2013.

[4] P. Covington, J. Adams, and E. Sargin, "Deep neural networks for youtube recommendations," in *Proceedings of the 10th ACM conference on recommender systems*, pp. 191–198, 2016.

[5] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, *et al.*, "Mastering the game of go without human knowledge," *Nature*, vol. 550, no. 7676, pp. 354–359, 2017.

[6] Google, "Overview of hyperparameter tuning." `https://cloud.google.com/ai-platform/training/docs/hyperparameter-tuning-overview`. Accessed: 2020-02-21.

[7] Amazon, "Perform automatic model tuning." `https://docs.aws.amazon.com/en_us/sagemaker/latest/dg/automatic-model-tuning.html`. Accessed: 2020-02-21.

[8] Microsoft, "What is automated machine learning (automl)?," 2022.

[9] Y.-W. Chen, Q. Song, and X. Hu, "Techniques for automated machine learning," *ACM SIGKDD Explorations Newsletter*, vol. 22, no. 2, pp. 35–50, 2021.

[10] B. Colson, P. Marcotte, and G. Savard, "An overview of bilevel optimization," *Annals of operations research*, vol. 153, no. 1, pp. 235–256, 2007.

[11] Y. Quanming, W. Mengshuo, J. E. Hugo, G. Isabelle, H. Yi-Qi, L. Yu-Feng, T. Wei-Wei, Y. Qiang, and Y. Yang, "Taking human out of learning applications: A survey on automated machine learning," *arXiv preprint arXiv:1810.13306*, 2018.

[12] T. Elsken, J. H. Metzen, and F. Hutter, "Neural architecture search: A survey.," *Journal of Machine Learning Research*, vol. 20, no. 55, pp. 1–21, 2019.

[13] M. Wistuba, A. Rawat, and T. Pedapati, "A survey on neural architecture search," *arXiv preprint arXiv:1905.01392*, 2019.

[14] X. He, K. Zhao, and X. Chu, "Automl: A survey of the state-of-the-art," *arXiv preprint arXiv:1908.00709*, 2019.

[15] M.-A. Zöller and M. F. Huber, "Benchmark and survey of automated machine learning frameworks,"

[16] P. Bergmann, M. Fauser, D. Sattlegger, and C. Steger, "Mvtec ad–a comprehensive real-world dataset for unsupervised anomaly detection," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 9592–9600, 2019.

[17] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[18] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," tech. rep., Citeseer, 2009.

[19] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, 2012.

[20] B. Frénay and M. Verleysen, "Classification in the presence of label noise: a survey," *IEEE transactions on neural networks and learning systems*, vol. 25, no. 5, pp. 845–869, 2013.

[21] A. Ghosh, H. Kumar, and P. Sastry, "Robust loss functions under label noise for deep neural networks," in *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[22] G. Patrini, A. Rozza, A. Krishna Menon, R. Nock, and L. Qu, "Making deep neural networks robust to label noise: A loss correction approach," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1944–1952, 2017.

[23] Z. Zhang and M. Sabuncu, "Generalized cross entropy loss for training deep neural networks with noisy labels," in *Advances in Neural Information Processing Systems*, pp. 8778–8788, 2018.

[24] N. Manwani and P. S. Sastry, "Noise tolerance under risk minimization," *IEEE Trans. Cybernetics*, vol. 43, no. 3, pp. 1146–1151, 2013.

[25] H. Kumar and P. S. Sastry, "Robust loss functions for learning multi-class classifiers," in *2018 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pp. 687–692, 2018.

[26] N. Charoenphakdee, J. Lee, and M. Sugiyama, "On symmetric losses for learning from corrupted labels," *arXiv preprint arXiv:1901.09314*, 2019.

[27] H. Liu, K. Simonyan, and Y. Yang, "DARTS: Differentiable architecture search," in *International Conference on Learning Representations*, 2019.

[28] H. Pham, M. Guan, B. Zoph, Q. Le, and J. Dean, "Efficient neural architecture search via parameters sharing," in *Proceedings of the 35th International Conference on Machine Learning*, 2018.

[29] H. Jin, Q. Song, and X. Hu, "Auto-keras: An efficient neural architecture search system," *ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2019.

[30] D. Hendrycks, M. Mazeika, D. Wilson, and K. Gimpel, "Using trusted data to train deep networks on labels corrupted by severe noise," in *Advances in Neural Information Processing Systems 31* (S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, eds.), pp. 10477–10486, Curran Associates, Inc., 2018.

[31] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine learning*, vol. 8, no. 3-4, pp. 229–256, 1992.

[32] Y.-W. Chen, Q. Song, X. Liu, P. Sastry, and X. Hu, "On robustness of neural architecture search under label noise," *Frontiers in big Data*, vol. 3, p. 2, 2020.

[33] L. Li and A. Talwalkar, "Random search and reproducibility for neural architecture search," *CoRR*, vol. abs/1902.07638, 2019.

[34] D. I. Inouye, P. Ravikumar, P. Das, and A. Datta, "Hyperparameter selection under localized label noise via corrupt validation," in *Learning with Limited Labeled Data (NeurIPS Workshop)*, 2017.

[35] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[36] K. Kandasamy, W. Neiswanger, J. Schneider, B. Poczos, and E. P. Xing, "Neural architecture search with bayesian optimisation and optimal transport," in *Advances in Neural Information Processing Systems 31*, 2018.

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

[38] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, "Learning transferable architectures for scalable image recognition," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[39] H. Cai, J. Yang, W. Zhang, S. Han, and Y. Yu, "Path-level network transformation for efficient architecture search," in *Proceedings of the 35th International Conference on Machine Learning*, 2018.

[40] E. Real, S. Moore, A. Selle, S. Saxena, Y. L. Suematsu, J. Tan, Q. V. Le, and A. Kurakin, "Large-scale evolution of image classifiers," in *Proceedings of the 34th International Conference on Machine Learning*, 2017.

[41] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le, "Regularized evolution for image classifier architecture search," in *Thirty-Third AAAI Conference on Artificial Intelligence*, 2019.

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

[43] R. Luo, F. Tian, T. Qin, E. Chen, and T.-Y. Liu, "Neural architecture optimization," in *Advances in Neural Information Processing Systems*, 2018.

[44] X. Gastaldi, "Shake-shake regularization," *arXiv preprint arXiv:1705.07485*, 2017.

[45] G. Larsson, M. Maire, and G. Shakhnarovich, "Fractalnet: Ultra-deep neural networks without residuals," in *ICLR*, 2017.

[46] T. DeVries and G. W. Taylor, "Improved regularization of convolutional neural networks with cutout," *arXiv preprint arXiv:1708.04552*, 2017.

[47] C. Zhang, S. Bengio, M. Hardt, B. Recht, and O. Vinyals, "Understanding deep learning requires rethinking generalization," in *ICLR*, 2017.

[48] D. Arpit, S. Jastrzębski, N. Ballas, D. Krueger, E. Bengio, M. S. Kanwal, T. Maharaj, A. Fischer, A. Courville, Y. Bengio, *et al.*, "A closer look at memorization in deep networks," in *Proceedings of the 34th International Conference on Machine Learning*, 2017.

[49] H. Zhang, M. Cisse, Y. N. Dauphin, and D. Lopez-Paz, "mixup: Beyond empirical risk minimization," in *International Conference on Learning Representations*, 2018.

[50] H.-S. Chang, E. Learned-Miller, and A. McCallum, "Active bias: Training more accurate neural networks by emphasizing high variance samples," in *Advances in Neural Information Processing Systems*, pp. 1002–1012, 2017.

[51] M. Ren, W. Zeng, B. Yang, and R. Urtasun, "Learning to reweight examples for robust deep learning," in *International Conference on Machine Learning*, pp. 4331–4340, 2018.

[52] L. Jiang, Z. Zhou, T. Leung, L.-J. Li, and L. Fei-Fei, "MentorNet: Learning data-driven curriculum for very deep neural networks on corrupted labels," in *Proceedings of the 35th International Conference on Machine Learning*, 2018.

[53] C. E. Brodley and M. A. Friedl, "Identifying mislabeled training data," *Journal of artificial intelligence research*, vol. 11, pp. 131–167, 1999.

[54] B. Han, Q. Yao, X. Yu, G. Niu, M. Xu, W. Hu, I. Tsang, and M. Sugiyama, "Co-teaching: Robust training of deep neural networks with extremely noisy labels," in *Advances in neural information processing systems*, pp. 8527–8537, 2018.

[55] X. Yu, B. Han, J. Yao, G. Niu, I. Tsang, and M. Sugiyama, "How does disagreement help generalization against label corruption?," in *International Conference on Machine Learning*, pp. 7164–7173, 2019.

[56] A. Vahdat, "Toward robustness against label noise in training deep discriminative neural networks," in *Advances in Neural Information Processing Systems*, pp. 5596–5605, 2017.

[57] A. Khetan, Z. C. Lipton, and A. Anandkumar, "Learning from noisy singly-labeled data," in *International Conference on Learning Representations*, 2018.

[58] T. Xiao, T. Xia, Y. Yang, C. Huang, and X. Wang, "Learning from massive noisy labeled data for image classification," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 2691–2699, 2015.

[59] N. Natarajan, I. S. Dhillon, P. K. Ravikumar, and A. Tewari, "Learning with noisy labels," in *Advances in neural information processing systems*, pp. 1196–1204, 2013.

[60] C. Liu, L.-C. Chen, F. Schroff, H. Adam, W. Hua, A. L. Yuille, and L. Fei-Fei, "Auto-deeplab: Hierarchical neural architecture search for semantic image segmentation," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 82–92, 2019.

[61] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, "Learning transferable architectures for scalable image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 8697–8710, 2018.

[62] E. Real, S. Moore, A. Selle, S. Saxena, Y. L. Suematsu, J. Tan, Q. Le, and A. Kurakin, "Large-scale evolution of image classifiers," *arXiv preprint arXiv:1703.01041*, 2017.

[63] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le, "Regularized evolution for image classifier architecture search," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 33, pp. 4780–4789, 2019.

[64] S. Xie, H. Zheng, C. Liu, and L. Lin, "SNAS: stochastic neural architecture search," in *International Conference on Learning Representations*, 2019.

[65] Q. Yao, J. Xu, W.-W. Tu, and Z. Zhu, "Efficient neural architecture search via proximal iterations.," in *AAAI*, pp. 6664–6671, 2020.

[66] X. Chen, L. Xie, J. Wu, and Q. Tian, "Progressive differentiable architecture search: Bridging the depth gap between search and evaluation," in *Proceedings of the IEEE International Conference on Computer Vision*, pp. 1294–1303, 2019.

[67] Y. Xu, L. Xie, X. Zhang, X. Chen, G.-J. Qi, Q. Tian, and H. Xiong, "PC-DARTS: Partial channel connections for memory-efficient architecture search," in *International Conference on Learning Representations*, 2020.

[68] C. Kim, H. Lee, M. Jeong, W. Baek, B. Yoon, I. Kim, S. Lim, and S. Kim, "torchgpipe: On-the-fly pipeline parallelism for training giant models," *arXiv preprint arXiv:2004.09910*, 2020.

[69] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, *et al.*, "Gpipe: Efficient training of giant neural networks using pipeline parallelism," in *Advances in neural information processing systems*, pp. 103–112, 2019.

[70] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," 2017.

[71] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 4700–4708, 2017.

[72] L. Wang, Y. Zhao, Y. Jinnai, Y. Tian, and R. Fonseca, "Alphax: exploring neural architectures with deep neural networks and monte carlo tree search," *arXiv preprint arXiv:1903.11059*, 2019.

[73] T. DeVries and G. W. Taylor, "Improved regularization of convolutional neural networks with cutout," *arXiv preprint arXiv:1708.04552*, 2017.

[74] J. Mei, Y. Li, X. Lian, X. Jin, L. Yang, A. Yuille, and J. Yang, "Atomnas: Fine-grained end-to-end neural architecture search," *arXiv preprint arXiv:1912.09640*, 2019.

[75] A. Sergeev and M. Del Balso, "Horovod: fast and easy distributed deep learning in tensorflow," *arXiv preprint arXiv:1802.05799*, 2018.

[76] S. Lee, J. K. Kim, X. Zheng, Q. Ho, G. A. Gibson, and E. P. Xing, "On model parallelization and scheduling strategies for distributed machine learning," in *Advances in neural information processing systems*, pp. 2834–2842, 2014.

[77] A. Harlap, D. Narayanan, A. Phanishayee, V. Seshadri, N. Devanur, G. Ganger, and P. Gibbons, "Pipedream: Fast and efficient pipeline parallel dnn training," *arXiv preprint arXiv:1806.03377*, 2018.

[78] T. Chen, B. Xu, C. Zhang, and C. Guestrin, "Training deep nets with sublinear memory cost," *arXiv preprint arXiv:1604.06174*, 2016.

[79] L. Li, K. Jamieson, A. Rostamizadeh, E. Gonina, J. Ben-Tzur, M. Hardt, B. Recht, and A. Talwalkar, "A system for massively parallel hyperparameter tuning," *Proceedings of Machine Learning and Systems*, vol. 2, pp. 230–246, 2020.

[80] V. Perrone, M. Donini, M. B. Zafar, R. Schmucker, K. Kenthapadi, and C. Archambeau, "Fair bayesian optimization," in *Proceedings of the 2021 AAAI/ACM Conference on AI, Ethics, and Society*, pp. 854–863, 2021.

[81] J. Xu, Z. Zhang, T. Friedman, and Y. Liang, "Gv d. broeck,"a semantic loss function for deep learning with symbolic knowledge,"," *arXiv preprint arXiv:1711.11157*, vol. 2, 2017.

[82] I.-C. Yeh and C.-h. Lien, "The comparisons of data mining techniques for the predictive accuracy of probability of default of credit card clients," *Expert systems with applications*, vol. 36, no. 2, pp. 2473–2480, 2009.

[83] S. Bird, M. Dudík, R. Edgar, B. Horn, R. Lutz, V. Milan, M. Sameki, H. Wallach, and K. Walker, "Fairlearn: A toolkit for assessing and improving fairness in ai," *Microsoft, Tech. Rep. MSR-TR-2020-32*, 2020.

[84] A. Wang, A. Singh, J. Michael, F. Hill, O. Levy, and S. R. Bowman, "Glue: A multi-task benchmark and analysis platform for natural language understanding," *arXiv preprint arXiv:1804.07461*, 2018.

[85] M. T. Ribeiro, T. Wu, C. Guestrin, and S. Singh, "Beyond accuracy: Behavioral testing of NLP models with CheckList," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, (Online), pp. 4902–4912, Association for Computational Linguistics, July 2020.

[86] K. Jamieson and A. Talwalkar, "Non-stochastic best arm identification and hyperparameter optimization," in *Artificial Intelligence and Statistics*, pp. 240–248, 2016.

[87] C. Wang, Q. Wu, M. Weimer, and E. Zhu, "Flaml: a fast and lightweight automl library," *Proceedings of Machine Learning and Systems*, vol. 3, pp. 434–447, 2021.

[88] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, *et al.*, "Ray: A distributed framework for emerging {AI} applications," in *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pp. 561–577, 2018.

[89] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization.," *Journal of machine learning research*, vol. 13, no. 2, 2012.

[90] C. Wang, Q. Wu, S. Huang, and A. Saied, "Economical hyperparameter optimization with blended search strategy," in *ICLR'21*, 2021.

[91] J. Davis and M. Goadrich, "The relationship between precision-recall and roc curves," in *Proceedings of the 23rd international conference on Machine learning*, pp. 233–240, 2006.

[92] V. Sanh, L. Debut, J. Chaumond, and T. Wolf, "Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter," *arXiv preprint arXiv:1910.01108*, 2019.

[93] J. D. M.-W. C. Kenton and L. K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of NAACL-HLT*, pp. 4171–4186, 2019.

[94] M. A. Gelbart, J. Snoek, and R. P. Adams, "Bayesian optimization with unknown constraints," *arXiv preprint arXiv:1403.5607*, 2014.

[95] J. R. Gardner, M. J. Kusner, Z. E. Xu, K. Q. Weinberger, and J. P. Cunningham, "Bayesian optimization with inequality constraints.," in *ICML*, vol. 2014, pp. 937–945, 2014.

[96] J. Bernardo, M. Bayarri, J. Berger, A. Dawid, D. Heckerman, A. Smith, and M. West, "Optimization under unknown constraints," *Bayesian Statistics*, vol. 9, no. 9, p. 229, 2011.

[97] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar, "Hyperband: A novel bandit-based approach to hyperparameter optimization," *The Journal of Machine Learning Research*, vol. 18, no. 1, pp. 6765–6816, 2017.

[98] D. Golovin, B. Solnik, S. Moitra, G. Kochanski, J. Karro, and D. Sculley, "Google vizier: A service for black-box optimization," in *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*, pp. 1487–1495, 2017.

[99] J. Rasley, Y. He, F. Yan, O. Ruwase, and R. Fonseca, "Hyperdrive: Exploring hyperparameters with pop scheduling," in *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, pp. 1–13, 2017.

[100] K. Swersky, J. Snoek, and R. P. Adams, "Freeze-thaw bayesian optimization," *arXiv preprint arXiv:1406.3896*, 2014.

[101] T. Domhan, J. T. Springenberg, and F. Hutter, "Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves," in *Twenty-fourth international joint conference on artificial intelligence*, 2015.

[102] R. Liaw, E. Liang, R. Nishihara, P. Moritz, J. E. Gonzalez, and I. Stoica, "Tune: A research platform for distributed model selection and training," *arXiv preprint arXiv:1807.05118*, 2018.

[103] M. Feurer, A. Klein, J. Eggensperger, Katharina Springenberg, M. Blum, and F. Hutter, "Efficient and robust automated machine learning," in *Advances in Neural Information Processing Systems 28 (2015)*, pp. 2962–2970, 2015.

[104] P. Das, N. Ivkin, T. Bansal, L. Rouesnel, P. Gautier, Z. Karnin, L. Dirac, L. Ramakrishnan, A. Perunicic, I. Shcherbatyi, *et al.*, "Amazon sagemaker autopilot: a white box automl solution at scale," in *Proceedings of the Fourth International Workshop on Data Management for End-to-End Machine Learning*, pp. 1–7, 2020.

[105] L. Wang, D. Zhang, J. Guo, and Y. Han, "Image anomaly detection using normal data only by latent space resampling," *Applied Sciences*, vol. 10, no. 23, p. 8660, 2020.

[106] V. Zavrtanik, M. Kristan, and D. Skočaj, "Draem-a discriminatively trained reconstruction embedding for surface anomaly detection," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 8330–8339, 2021.

[107] N.-C. Ristea, N. Madan, R. T. Ionescu, K. Nasrollahi, F. S. Khan, T. B. Moeslund, and M. Shah, "Self-supervised predictive convolutional attentive block for anomaly detection," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 13576–13586, 2022.

[108] T.-W. Tang, W.-H. Kuo, J.-H. Lan, C.-F. Ding, H. Hsu, and H.-T. Young, "Anomaly detection neural network with dual auto-encoders gan and its industrial inspection applications," *Sensors*, vol. 20, no. 12, p. 3336, 2020.

[109] S. Cass, "Nvidia makes it easy to embed ai: The jetson nano packs a lot of machine-learning power into diy projects-[hands on]," *IEEE Spectrum*, vol. 57, no. 7, pp. 14–16, 2020.

[110] B. Zoph and Q. Le, "Neural architecture search with reinforcement learning," in *International Conference on Learning Representations*, 2017.

[111] C. Liu, L.-C. Chen, F. Schroff, H. Adam, W. Hua, A. Yuille, and L. Fei-Fei, "Auto-deeplab: Hierarchical neural architecture search for semantic image segmentation," *arXiv preprint arXiv:1901.02985*, 2019.

[112] Y. Weng, T. Zhou, Y. Li, and X. Qiu, "Nas-unet: Neural architecture search for medical image segmentation," *IEEE Access*, vol. 7, pp. 44247–44257, 2019.

[113] N. Wang, Y. Gao, H. Chen, P. Wang, Z. Tian, C. Shen, and Y. Zhang, "Nas-fcos: Fast neural architecture search for object detection," in *proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 11943–11951, 2020.

[114] Y. Li, Z. Chen, D. Zha, K. Zhou, H. Jin, H. Chen, and X. Hu, "Automated anomaly detection via curiosity-guided search and self-imitation learning," *IEEE Transactions on Neural Networks and Learning Systems*, 2021.

[115] L. Ruff, R. Vandermeulen, N. Goernitz, L. Deecke, S. A. Siddiqui, A. Binder, E. Müller, and M. Kloft, "Deep one-class classification," in *International conference on machine learning*, pp. 4393–4402, PMLR, 2018.

[116] C. Manning and H. Schutze, *Foundations of statistical natural language processing*. MIT press, 1999.

[117] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," in *International Conference on Medical image computing and computer-assisted intervention*, pp. 234–241, Springer, 2015.

[118] L.-C. Chen, G. Papandreou, I. Kokkinos, K. Murphy, and A. L. Yuille, "Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected

crfs," *IEEE transactions on pattern analysis and machine intelligence*, vol. 40, no. 4, pp. 834–848, 2017.

[119] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

[120] Z. Guo, X. Zhang, H. Mu, W. Heng, Z. Liu, Y. Wei, and J. Sun, "Single path one-shot neural architecture search with uniform sampling," in *European Conference on Computer Vision*, pp. 544–560, Springer, 2020.

[121] P. Bergmann, S. Löwe, M. Fauser, D. Sattlegger, and C. Steger, "Improving unsupervised defect segmentation by applying structural similarity to autoencoders," *arXiv preprint arXiv:1807.02011*, 2018.

[122] K. Perlin, "An image synthesizer," *ACM Siggraph Computer Graphics*, vol. 19, no. 3, pp. 287–296, 1985.

[123] M. Cimpoi, S. Maji, I. Kokkinos, S. Mohamed, , and A. Vedaldi, "Describing textures in the wild," in *Proceedings of the IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2014.

[124] C.-L. Li, K. Sohn, J. Yoon, and T. Pfister, "Cutpaste: Self-supervised learning for anomaly detection and localization," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 9664–9674, 2021.

[125] T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollár, "Focal loss for dense object detection," in *Proceedings of the IEEE international conference on computer vision*, pp. 2980–2988, 2017.

[126] J. Jing, Z. Wang, M. Rätsch, and H. Zhang, "Mobile-unet: An efficient convolutional neural network for fabric defect detection," *Textile Research Journal*, vol. 92, no. 1-2, pp. 30–42, 2022.

[127] Z. Wang, J. Junfeng, H. Zhang, and Y. Zhao, "Real-time fabric defect segmentation based on convolutional neural network," *AATCC Journal of Research*, vol. 8, no. 1_suppl, pp. 91–96, 2021.

[128] L. Cheng, J. Yi, A. Chen, and Y. Zhang, "Fabric defect detection based on separate convolutional unet," *Multimedia Tools and Applications*, pp. 1–22, 2022.

[129] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, "Learning transferable architectures for scalable image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 8697–8710, 2018.

[130] L.-C. Chen, G. Papandreou, F. Schroff, and H. Adam, "Rethinking atrous convolution for semantic image segmentation," *arXiv preprint arXiv:1706.05587*, 2017.

[131] S. Mehta, M. Rastegari, A. Caspi, L. Shapiro, and H. Hajishirzi, "Espnet: Efficient spatial pyramid of dilated convolutions for semantic segmentation," in *Proceedings of the european conference on computer vision (ECCV)*, pp. 552–568, 2018.

[132] H. Cai, C. Gan, T. Wang, Z. Zhang, and S. Han, "Once-for-all: Train one network and specialize it for efficient deployment," *arXiv preprint arXiv:1908.09791*, 2019.

[133] H. Pham, M. Guan, B. Zoph, Q. Le, and J. Dean, "Efficient neural architecture search via parameters sharing," in *International conference on machine learning*, pp. 4095–4104, PMLR, 2018.

[134] Q. Yu, D. Yang, H. Roth, Y. Bai, Y. Zhang, A. L. Yuille, and D. Xu, "C2fnas: Coarse-to-fine neural architecture search for 3d medical image segmentation," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 4126–4135, 2020.

[135] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[136] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 4510–4520, 2018.

[137] N. Ma, X. Zhang, H.-T. Zheng, and J. Sun, "Shufflenet v2: Practical guidelines for efficient cnn architecture design," in *Proceedings of the European conference on computer vision (ECCV)*, pp. 116–131, 2018.

[138] J. Hu, L. Shen, and G. Sun, "Squeeze-and-excitation networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 7132–7141, 2018.

[139] A. Howard, M. Sandler, G. Chu, L.-C. Chen, B. Chen, M. Tan, W. Wang, Y. Zhu, R. Pang, V. Vasudevan, *et al.*, "Searching for mobilenetv3," in *Proceedings of the IEEE/CVF international conference on computer vision*, pp. 1314–1324, 2019.

[140] Y. Chen, Y. Ding, F. Zhao, E. Zhang, Z. Wu, and L. Shao, "Surface defect detection methods for industrial products: A review," *Applied Sciences*, vol. 11, no. 16, p. 7657, 2021.

[141] S. Akçay, A. Atapour-Abarghouei, and T. P. Breckon, "Skip-ganomaly: Skip connected and adversarially trained encoder-decoder anomaly detection," in *2019 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8, IEEE, 2019.

[142] T. Defard, A. Setkov, A. Loesch, and R. Audigier, "Padim: a patch distribution modeling framework for anomaly detection and localization," in *International Conference on Pattern Recognition*, pp. 475–489, Springer, 2021.

[143] D. Gudovskiy, S. Ishizaka, and K. Kozuka, "Cflow-ad: Real-time unsupervised anomaly detection with localization via conditional normalizing flows," in *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*, pp. 98–107, 2022.

[144] N. Cohen and Y. Hoshen, "Sub-image anomaly detection with deep pyramid correspondences," *arXiv preprint arXiv:2005.02357*, 2020.

[145] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le, "Regularized evolution for image classifier architecture search," in *Proceedings of the aaai conference on artificial intelligence*, vol. 33, pp. 4780–4789, 2019.