Efficient Non-Parametric Optimizer Search for Diverse Tasks

Ruochen Wang[1], **Yuanhao Xiong**[1], **Minhao Cheng**[2], **Cho-Jui Hsieh**[1]
[1]Department of Computer Science, UCLA, [2]HKUST
ruocwang@ucla.edu minhaocheng@ust.hk {chohsieh, yhxiong}@cs.ucla.edu

Abstract

Efficient and automated design of optimizers plays a crucial role in full-stack AutoML systems. However, prior methods in optimizer search are often limited by their scalability, generalization, or sample efficiency. With the goal of democratizing research and application of optimizer search, we present the first efficient, scalable and generalizable framework that can directly search on the tasks of interest. We first observe that optimizer updates are fundamentally mathematical expressions applied to the gradient. Inspired by the innate tree structure of the underlying math expressions, we re-arrange the space of optimizers into a super-tree, where each path encodes an optimizer. This way, optimizer search can be naturally formulated as a path-finding problem, allowing a variety of well-established tree traversal methods to be used as the search algorithm. We adopt an adaptation of the Monte Carlo method to tree search, equipped with rejection sampling and equivalent-form detection that leverage the characteristics of optimizer update rules to further boost the sample efficiency. We provide a diverse set of tasks to benchmark our algorithm and demonstrate that, with only 128 evaluations, the proposed framework can discover optimizers that surpass both human-designed counterparts and prior optimizer search methods. Our code is publicly available at https://github.com/ruocwang/enos.

1 Introductions

Motivated by a vision of democratizing machine learning, the central objective for automated machine learning (AutoML), such as automated architecture [22, 26, 29, 40, 55, 60, 62, 65] / optimizer [10, 12, 15, 17, 59, 68] / loss [36] / augmentation search [37, 39], lies in reducing the need for expert design on a diverse set of tasks. To achieve this goal, it is critical for AutoML systems to exhibit a high level of efficiency, so that they can be directly applied to a variety of tasks without consuming a humongous amount of computing resources. A widely successful example of such an effort is DARTS [40] in Neural Architecture Search (NAS), which reduces the search cost from thousands of GPU days of early RL-based algorithms to a single digit, enabling direct application of NAS systems to a wide range of tasks [32, 33, 38, 44, 50].

Inspired by the success of efficient NAS methods, we turn our attention to another important but much less studied area of AutoML - **Automated optimizer search, where an efficient, scalable and generalizable framework is still absent.** Optimizer search aims to automatically design a suitable update function that takes gradients as inputs and produces update directions for the optimizee's parameters. Pioneering work in this area, coined *Learning to Optimize (L2O)*, adopts a data-driven approach by replacing human-designed update rules with a learnable parametric function [10, 12, 17, 59]. However, parametric optimizers are fundamentally not scalable to large models or datasets, as inferring its parameters typically requires expensive meta-learning steps such as backpropagating through gradient descent [10, 15, 68]. Moreover, the learned optimizer often generalizes poorly to even minor variants of its training task (Figure 3) [10, 68]. Poor scalability and generalization prevent L2O from being served as a general-purpose optimizer search framework that can be **directly applied to tasks of interest**.

The aforementioned limitations of parametric optimizers bring our attention to another line of method that searches over the discrete space of non-parametric update functions [1], which generally exhibit the same level of scalability and generality as human-designed optimizers [15, 51]. NOS-RL [15] extends early RL-based NAS framework [22] to optimizer search, proposing to learn a sequential controller to produce optimizer update rules according to a predefined pattern. However, NOS-RL is sample inefficient, requiring over 10k evaluations to find good candidates. More recently, AutoML-Zero [51, 69] proposes to search over the vast space of computer codes for the entire ML pipeline (including the optimizer). The excessive generality of its search space makes it even more costly to run than RL-based method. The search cost of existing non-parametric optimizer search frameworks makes them computationally prohibitive not only for practitioners to apply but also for researchers to analyze.

With the goal of democratizing research and practical applications of automated optimizer design, we introduce the first efficient, scalable, and generalizable optimizer search framework that can be directly applied to a wide range of tasks. We observe that non-parametric update rules are essentially mathematical expressions, with an innate tree structure where nodes are elementary math operators and edges represent their I/Os. Consequently, generating an update rule can be viewed as progressively appending nodes to the expression tree until it is complete. Inspired by this observation, we re-imagine the optimizer search space as a super-tree of mathematical expressions. Each leaf node on the super-tree contains an optimizer, and the path towards it represents the generation process of that optimizer's underlying expression. With the tree-structured search space, optimizer search can be naturally formulated as a path-finding problem, allowing a wide range of well-established tree-traversal methods to be used as the search algorithm. We show that a simple adaptation of Monte Carlo Sampling [30, 53], equipped with our proposed rejection sampling and equivalent-form detection, can already produce remarkable results on our search space within a fraction of budgets compared with NOS-RL ($\sim 1\%$).

We extensively evaluate the proposed framework on a diverse set of learning tasks: digit classification with MNISTNET [10], image classification with ConvNet [15], graph learning with (Cluster-)GAT [21, 28], norm-bounded adversarial attack on robustly trained models [20, 45, 46], and BERT fine-tuning on NLP datasets [34, 56]. These tasks cover both constraint and unconstrained optimizations and span over a large variety of models and datasets. Despite the simplicity, the proposed framework is able to discover update rules that surpass human-designed optimizers and prior optimizer search methods, with a budget of only 128 evaluations. We hope the proposed framework could lower the barrier of entry to practical non-parametric optimizer search, thereby providing an entry point for researchers and practitioners from ML community and beyond to study and utilize automated optimizer search systems.

2 Efficient, scalable and generalizable framework for optimizer search

2.1 Optimizer design space

Notations and problem formulation Deep learning tasks are frequently expressed as optimizing a loss function $L(\cdot)$ defined over parameter domain $\theta \in \Theta$. The minimizer of L can thus be obtained by $\theta^* = \arg\min_{\theta \in \Theta} L(\theta)$. For differentiable functions, a standard optimizer typically takes the form of iterative gradient descent: $\theta_{t+1} = \theta_t - \gamma * \phi(\nabla_\theta L(\theta_t))$, where t is the current iteration, γ is the learning rate and ϕ denote the update function. Existing optimizers primarily differ in their design of update function ϕ; For example, vanilla gradient descent uses identity mapping $\phi(x) = x$ as the update function, whereas Adam adopts a momentum-based dynamic learning rate schema: $\phi(\nabla_\theta L(\theta_t)) = m(\nabla_\theta L(\theta_t))/\sqrt{m((\nabla_\theta L(\theta_t))^2)}$, where $m(\cdot)$ denotes the momentum function with an internal state.

The goal of optimizer search is to automatically find a suitable update function ϕ over some hypothesis space Φ. The hypothesis spaces used in prior work can be divided into two categories: non-parametric and parametric spaces. Most human-designed optimizers belong to the first category, where the update

[1]Sometime it is referred to as **symbolic optimizers**, which is a somewhat inaccurate categorization as symbolic functions could also contain learnable parameters.

function ϕ is not trainable. Learnable optimizers, such as L2LGD2 [10] and SymbolicL2O [68], fall into the second category. Our work mainly focuses on non-parametric optimizer search, with the goal of providing an efficient, scalable and generalizable optimizer search framework that can be directly apply to various tasks.

Optimizer update rules as expression trees The first step toward such a framework is to understand the structure of non-parametric optimizers. We realize that, fundamentally, optimizers are mathematical expressions consisting of elementary operators ($+$, $-$, $sign()$, $inputs$, e.t.c.). Math expressions have an inherent tree structure that preserves its order of execution, where nodes are operators and edges represent their I/Os. For instance, Diagram 1 shows the expression tree of Adam [9]:

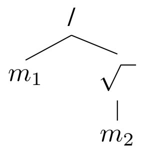

Diagram 1: Adam optimizer

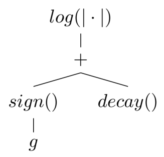

Diagram 2: Our discovered Optimizer for adversarial attack

where m_1 and m_2 denote the first and second order momentum (which can also be broken down into their own expression trees).

Therefore, the generation of an update rule, as a mathematical expression, can also be conducted via top-down node selection: Take Adam as an example, we first select division ($/$) as the root node. For its left child, we pick m_1, which is a leaf node and thus ends the branch. For the right child, we select $\sqrt{\ }$, and subsequently pick m_2 to follow it. At this point, there is no empty branches left, and we obtain the complete update rule for Adam.

A tree-structured search space Inspired by the completion process of update rules, we rearrange all expressions into a **super-tree**, where each leaf node contains an update rule and each path represents its completion process. The super-tree can be generated in a top-down manner: Starting from the root node with an empty update rule, we generate each of its child nodes by inserting a different operator into the update rule, and repeat this process for the generated nodes. Consequently, an optimizer can be sampled by traversing the super-tree until a leaf node is reached. Since the super-tree can grow infinitely deep, it is often desirable to restrict the tree to a predefined depth N, where only the paths that can be completed within depth N are included. Figure 1 provides an instantiation of our super-tree, where the paths leading to Adam and SGD optimizers are displayed as an example.

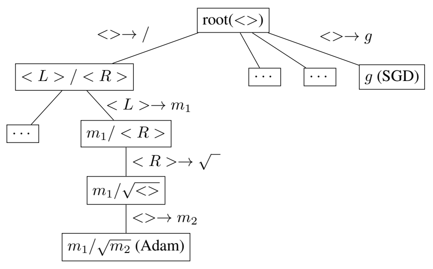

Figure 1: An illustration of traversing the super-tree to discover Adam and SGD.

The benefit of arranging the optimizer space into a tree is two folds. Firstly, the tree-based search space is tight:

Proposition 1 *Define the length of an update rule as the number of operators it includes, then the above tree-based search space is* ***tight****: a super-tree with a maximum depth of N covers all update rules of length no greater than N.*

In a tight search space, all optimizers can be represented at the right level of complexity, allowing them to be visited by the search algorithm without exploring unnecessarily deep into the super-tree. Although tightness is a fairly obvious result for our space, it is not the case for the previous search space defined in NOS-RL, as we will explain later. Secondly, with our super-tree, optimizer search can be naturally formulated as a path-finding problem, allowing a variety of well-established tree traversal methods to be deployed as search algorithms.

Contents To concretize the content of the search space, we allow three types of operators in the optimizer update rule:

- a set of p_1 unary operators (e.g. $log(|\cdot|)$, $exp(\cdot)$, $\sqrt{|\cdot|}$, $sign(\cdot)$, $drop(\cdot)$)
- a set of p_2 binary operators (e.g. $+$, $-$, $\times$, $/$, $pow(\cdot,\cdot)$)
- a set of L leaf values (input operators) containing gradient-based terms (e.g. g, m_1), decays (e.g. $cosine\_decay$), and constants (e.g. 1, 2)

This categorization of mathematical operators is not new, as it is also adopted in symbolic math solver [35] and NOS-RL [15].

Comparison with NOS-RL's search space Although both NOS-RL [15] and our framework use elementary math operators as building blocks for optimizers, they have little in common in terms of the arrangement of the search spaces. Optimizers in NOS-RL's search space are formed by a chain of predefined motifs: $b(u(I), u(I))$, where b, u, I denote binary, unary and input operators. Due to the fixed structure of such motifs, NOS-RL's search space is not tight: there exist many optimizers that take extra longer sequences to express, potentially lowering their chance of being discovered by the search algorithm. For instance, Diagram 2 shows an optimizer of length 5, but it takes $(10-1)$ nodes (two chained motifs) to represent it under NOS-RL's arrangement; Moreover, NOS-RL's search space also requires extra bypass operators (e.g. $u(x) = x$ and $b(x, y) = x$) to cover even human-design optimizers such as Adam and PGD, further increasing the complexity. In contrast, our representation of optimizers is directly inspired by the innate structure of its underlying mathematical expressions, resulting in a tight tree-based search space. In our search space, optimizer search can be naturally formulated as a form of top-down path-finding problem. In the next sections, we will detail our choice of algorithms for traversing the super-tree, as well as several techniques that leverage the characteristics of optimizer update rules to boost the sample efficiency.

2.2 Monte Carlo Sampling for tree traversal

We adopt a simple adaptation of Monte Carlo Sampling to tree traversal [30, 53, 70] (MCT) as the search algorithm. The idea is to assign scores to the nodes in the super-tree (Figure 1), and use these scores to guide the tree traversal. We define the score of a node v as a Monte Carlo estimation over unrolling steps from v: If v is an internal node, we randomly generate a set of unrolled paths from v to the corresponding leaf nodes, and take the average score of the resulting optimizers as the score for v; If v is a leaf node, we set its score to 0 as it cannot be expanded. The search can thus be conducted as follows: 1). Starting from the root node $v^{(0)}$ at level 0, we generate all child nodes $\{v^{(1)}\}$ of $v^{(0)}$ by inserting each operator from the candidate pool to the update rule in $v^{(0)}$; 2). From there, we select the child node $v^{*(1)}$ with the highest MC score to expand, and move on to the next level; 3). The process is repeated until a predefined maximum search level is reached. Algorithm 1 in the Appendix provides a detailed summary of the complete search process.

Directly applying the MCT algorithm to optimizer search would not perform well under limited search budgets, due to two unique characteristics of optimizer update rules that challenge the sample efficiency of the Monte Carlo estimates. Firstly, the majority of mathematical expressions, when deployed as optimizer update rules, perform poorly or even would not converge. This is usually not the case for other AutoML tasks such as neural architecture search, as most networks in the search space perform reasonably well. The large body of poor-performing optimizers not only consumes precious search budget, but also causes the MC estimation to be unstable. Secondly, there exists many mathematical redundancies in the expression space, for example: $sign(sign(sign(x)))$ can be reduced to $sign(x)$, and $\frac{m_1+\sqrt{m_2}}{\sqrt{m_2}}$ is equivalent to $\frac{m1}{\sqrt{m2}} + 1$. Identifying and eliminating these redundancies would not only save budget, but also prevent the sampling distribution from biasing toward mathematically simple and shallow update rules. To address these issues and further boost the sample efficiency, we propose two sets of techniques - rejection sampling and equivalent-form detection. When combined with these techniques, the simple MCT algorithm becomes particularly effective for the optimizer search task. We will discuss them in detail in the following sections.

2.3 Rejection sampling

Eliminating poor optimizers with a train-free task-agnostic test Inspired by the characteristics of optimizer update rules, we develop a train-free task-agnostic test to eliminate poor optimizers without evaluating them. We propose a necessary condition for a valid optimizer: it must produce an acute angle with steepest descent direction (i.e. gradients). We check the validity of optimizers against this condition and only evaluate those that pass the test. For the test to be task-agnostic, we feed the optimizer with a batch of random Gaussian vectors in place of actual gradients. Formally, the descent test can be written as:

$$E_{u \sim \mathcal{N}(0,1)}\left[\cos(\phi(\nabla_\theta \mathcal{L}), \nabla_\theta \mathcal{L})\right] > \lambda_d$$

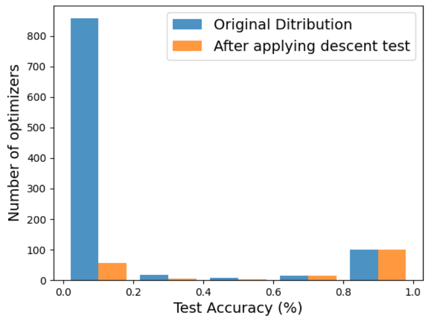

Figure 2: Performance distribution of optimizers after applying descent test, under $\lambda_d = 0.15$ and a batch size of 25.

where λ_d is a predefined threshold. Although our descent test is by-no-mean comprehensive, it can effectively rule out a large chunk of poor optimizers with negligible false-negative rates, as demonstrated in Figure 2.

Reducing the variance of MC estimates via score thresholding After applying the descent test, there still remains a non-negligible portion of poor optimizers. When sampled during the unrolling step, these optimizers would drastically lower the Monte Carlo score of the stem node, causing the MC estimation to exhibit high variance and thus become unreliable. This adverse effect is especially severe under the efficient setting when the sample size is small. Therefore, we propose to simply reject candidates with scores lower than a predefined threshold, thereby removing them from the MC scores of the corresponding stem nodes.

2.4 Detecting and handling redundancies in mathematically equivalent forms

On-the-fly constraint tree-traversal for redundant path pruning One benefit of formulating the search problem as top-down path-finding is that we can easily apply constraints on-the-fly to eliminate undesirable branches - those that lead to mathematically redundant expressions in our case. We identify three main categories of such constraints:

- child operator that nullifies its parent's operator. e.g. $-(-x) = x$, $ln(e^x) = x$
- child operator that is redundant under its parent. e.g. $clip(clip(x)) = clip(x)$
- sequence of operators that reduces to a constant in the search space. e.g. $\sqrt{|sign(x)|} = 1$

The complete sets of constraints we used can be found in the Appendix. Enforcing these constraints during the traversal can effectively trim down the search tree, allowing the algorithm to explore branches that lead to more diverse and complex expressions.

Hashing mathematically equivalent expressions Besides enforcing constraints during the traversal, it is also important to detect mathematically equivalent optimizers to avoid duplicated evaluations. One can always apply off-the-shelf symbolic solvers to identify the equivalence of two expressions, ϕ and ϕ', by checking if $(\phi - \phi')$ can be reduced to 0. However, it could become extremely slow as the pool of evaluated optimizers $\{\phi_i\}_1^N$ gets larger and larger, since we need to solve N pair of equations every time a new update rule is sampled. Instead, we apply hashing to efficiently query the evaluated candidate pool for mathematically equivalent optimizers. Concretely, we assign each optimizer a hash code, obtained by feeding a fixed probing vector as input to the optimizer and recording its output. The probing vector is pre-sampled from Gaussian distribution. When a newly sampled optimizer arrives, we only need to compare its code with the hash table to check the existence of its equivalent form. Empirically, it is much faster to run the proposed hashing-based checker than symbolic solvers.

3 Discussions and relationship to prior work

Automated optimizer design Optimization plays a crucial role in training deep learning models. Generally, there does not exist one optimizer that aces all scenarios, as different tasks (dataset, architecture, loss, parameterization, e.t.c.) might favor different optimization methods [2, 10]. The demand for task-specific optimizers stimulates research interest in developing automated systems for optimizer design [10, 12, 15, 17, 41, 51, 59, 66, 68]. Early work adopts a data-driven method by modeling the optimizer update with a parametric function [10, 12, 59]. L2LGD2 [10] deploys an LSTM model as the update function that takes historical gradients as input and produces the update direction. However, parametric optimizer search methods are fundamentally limited by its scalability, as inferring its parameters requires expensive meta-learning steps such as back-propagation through optimization [10, 59, 68]. Although SymbolicL2O [68] improves the scalability of the learned LSTM optimizer by distilling it into a lightly-parameterized symbolic optimizer, it still requires a pretrained LSTM model to begin with. Instead of learning a parametric optimizer, NOS-RL [15] directly searches over a discrete space of non-parametric update functions comprised of mathematical operators. It extends early RL-based NAS method [22] to optimizer search, by training a sequential controller to produce the optimizer update rule according to a predefined pattern. However, similar to its NAS counterpart, NOS-RL is also computationally expensive, requiring over 10k evaluations to find good candidates.

Symbolic optimization and differential program synthesis Symbolic optimization (SO) [1, 13, 53, 61, 63, 64] aims at optimizing an objective over a symbolic hypothesis space of functions (or more broadly, programs). One line of work attempts to recover the unknown equation from its generated data, with great potential in automating scientific discoveries [54, 64]. Another line of methods aims at finding a more interpretable and generalizable symbolic model to replace the black-box neural networks [53, 61, 63]; Applications that witnessed some success include learning symbolic policy networks for RL [63] and sequential classification models [53, 61]. The latter is often studied under the concept of Program Synthesis [24], where a model is extended to include programmatic rules such as if-else clause, indexing, e.t.c. SO is closely connected to AutoML at a high level, as both fields frame their problems as discrete optimization. Indeed, many existing optimizer search methods can find their counterparts in symbolic optimization. Our method is also inspired by the rich body of literature in deep symbolic mathematics and program synthesis, which also explores tree-based expression spaces for differential equations and programs [30, 35, 53, 61, 64]. However, due to significant differences in taskonomy, SO and AutoML methods are often developed separately, converging into different branches of techniques. Symbolic optimization often studies tasks where candidates are cheap to evaluate but finding the global optimal is desired [1, 54, 64]; As a result, sample efficiency is often not the primary concern. Much to the opposite, in AutoML tasks, candidate evaluations are extremely expensive; Therefore, it is more beneficial to identify a good-enough candidate within a limited amount of budget.

4 Empirical evaluations on a diverse set of tasks

We extensively evaluate the proposed framework on a suite of tasks, covering a variety of models and datasets. On standard benchmark tasks for optimizer search, our method is able to discover optimizers that outperform its human-designed and automatically searched counterparts. In addition, we also show that the proposed framework enables automated optimizer design for many other popular learning tasks, such as adversarial attack, GNN training, and BERT finetuning. Due to the space limits, we will include detailed descriptions, search settings, and discovered optimizers for each task in the Appendix.

4.1 General setting

MCT algorithm Across all experiments, we limit the maximum level of MCT traversal to 4, and set the number of Monte Carlo samples to 32 (a multiple of 8 for parallelism on 8-GPU servers) for each level. This amounts to a fixed total budget of 128 evaluations. The maximum depth for the super-tree is set to 10, which already covers many top-performing optimizers for various tasks. We use a similar set of elementary operations as NOS-RL to build the optimizers, with only minor adjustments for some tasks (see Appendix for more details).

Optimizer evaluation We follow the default settings and hyperparameters for each task, and only swap out the optimizer; This potentially puts our algorithm at a disadvantage, as the hyperparameters are usually tuned around the default optimizers. Before optimizer evaluation, we perform grid search on a small proxy task (fewer steps) to find a proper learning rate. During the grid search, we also aggressively terminate optimizers if their performance falls under a certain threshold. Since early stopped optimizers consume fewer resources than a full evaluation, we do not count them into the budget (number of evaluations).

4.2 Hand-written digit classification

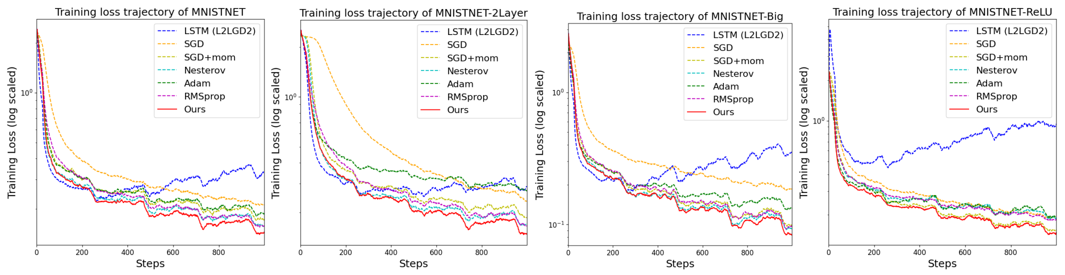

Figure 3: Training loss trajectory on hand-written digit classification task (log scaled). Each optimizer is evaluated for 4 random seeds. Our method is marked in red.

We first compare our method with the LSTM-based optimizer (L2LGD2) on hand-written digit classification. Following L2LGD2 [10], the goal is to minimize the cumulative training loss of a single-hidden-layer MLP with Sigmoid activation (MNISTNET) on the MNIST dataset; The search is conducted on MNISTNET for 100 steps with a batch size of 128, and the discovered optimizers are subsequently transferred to three variants of MNISTNET with different activations (MNISTNET-ReLU), number of hidden layers (MNISTNET-2Layer), and dimensions (MNISTNET-Big). Under this setting, our method finishes in 0.92h on RTX 2080ti, much faster than L2LGD2 (2.62h).

As shown in Figure 3, our discovered optimizer achieves the lowest training loss under both direct search and transfer settings. Notable, the LSTM-based parametric update function indeed converges faster when the number of steps is close to the search phase (black-dotted vertical line on Figure 3). However, it extrapolates poorly to longer trajectories. As the training proceeds, all other non-parametric optimizers eventually catch-up, achieving much lower training loss. Moreover, LSTM-based optimizer also generalizes poorly to other model variants (most noticeably MNISTNET-ReLU), revealing its tendency to overfit the search task.

4.3 Image classification with ConvNet

We proceed to evaluate our method on the CIFAR-10 [8] classification task proposed in NOS-RL [15]. The model of choice is a 2-layer ConvNet. Each layer of this network contains a 32-filter 3x3 convolution with ReLU activation and batch normalization. Following NOS-RL's setting, for every optimizer, the best learning rate is searched over a grid of $\{1e^{-5}, 1e^{-4}, 1e^{-3}, 1e^{-2}, 1e^{-1}, 1\}$ with 1 epoch of training, and the discovered learning rate is subsequently used to train the model for a longer period of time (5 epochs). Since NOS-RL's implementation is not open-sourced, we reproduce and compare with the two families of discovered optimizers described in NOS-RL paper: AddSign and PowSign.

The results are summarized in Table 1. For NOS-RL, we display the performance of the top 4 variants of PowSign and AddSign, which are obtained after training the controller for over 10k evaluations (Figure 4 in the NOS-RL paper [15]). With only a fraction ($\sim$1%) of the search budget, our framework is able to discover optimizers that reach a test accuracy of 77.02%, topping both PowSign and AddSign optimizers and also human-designed ones by a sizable margin. The sheer reduction in search cost and the improvement in search performance evince the efficiency and effectiveness of the proposed framework for discovering better optimizers.

4.4 Adversarial attack

Next, we apply our framework to discover optimizers for constraint optimization. We select adversarial attack, which aims at finding norm-bounded perturbations in the input space that alter the model's predictions. The de facto optimizer used in adversarial attack is Projected Gradient Descent

Table 1: Performance of automated search algorithms on CIFAR-10.

Optimizer	**Test Accuracy (%)**	**Search Method**	**Search Budget (#evaluations)**
SGD	70.99% ± 2.12	manual	-
SGD + Momentum	74.12% ± 0.44	manual	-
Nesterov	74.15% ± 0.52	manual	-
Adam	73.42% ± 0.56	manual	-
RMSprop	71.42% ± 1.42	manual	-
PowSign-ld	75.48% ± 0.45	RL on hand-crafted patterns	>10,000
PowSign-cd	76.21% ± 0.16	RL on hand-crafted patterns	>10,000
AddSign-ld	75.54% ± 0.39	RL on hand-crafted pattern space	>10,000
AddSign-cd	76.07% ± 0.59	RL on hand-crafted pattern space	>10,000
Ours	**77.02% ± 0.19**	MCT on super-tree space	**128**

(PGD) [20]. We consider the most popular l_∞-norm setting. For l_∞-norm bounded attack, PGD takes the form of: $x = Proj_{B_\epsilon^\infty(x_o)}(x + \gamma sign(\nabla_x L(x)))$, where $B_\epsilon^\infty(x_o)$ represents a ϵ ball around the original image x_o w.r.t. l_∞-norm. The models of choice come from the AutoAttack library [46], which holds a leaderboard of top defense methods. Following their settings, we set $\epsilon = 8/255$, and run each optimizer once for 100 steps on every image from the test split [46].

On this task, we mainly search for the update rule inside the projection operator (e.g. $sign()$ for PGD). The search is conducted on the pre-trained Carmon2019 model [27], and the proposed optimizer is subsequently evaluated on other top defense methods for WideResNet [14] (WRN-<depth>-<width>) and ResNet [11] (RN-<depth>). As shown in Table 2, our discovered optimizer consistently outperforms PGD by a sizable margin. Surprisingly, we found that the algorithm tends to pick $log(|\cdot|)$ rather than $sign(\cdot)$ as the first operator, resulting in many log-based optimizers that surpass sign-based PGD.

Table 2: Attack success rate of different optimizers on top defense methods on CIFAR-10.

Defense Models	**PGD**	**APGD**	**Ours**
Carmon2019 (WRN-28-10) [27]	37.83%	38.22%	**38.35%**
Gowal2020‡ (WRN-70-16) [48]	31.10%	**32.00%**	**32.00%**
Gowal2020‡ (WRN-34-20) [48]	40.05%	40.46%	**40.50%**
Gowal2020‡ (WRN-28-10) [48]	33.65%	34.33%	**34.34%**
Sehwag2020‡ (WRN-28-10) [52]	40.00%	40.43%	**40.46%**
Wu2020‡ (WRN-28-10) [58]	36.41%	36.70%	**36.78%**
Wang2020‡ (WRN-28-10) [42]	37.78%	38.16%	**38.27%**
Engstrom2019 (RN-50) [31]	47.76%	48.25%	**48.32%**
Wong2020Fast (RN-18) [57]	53.69%	54.11%	**54.19%**

‡ Methods that explore extra data during robust training.

In addition to PGD, we also compare our log-based optimizer with the best handcrafted and tuned optimizer for adversarial attack: Adaptive PGD (APGD) [46]; The design of APGD is packed with domain expertise: it combines a well-tuned momentum update rule with a conditional learning rate decay based on a handcrafted schedule and sophisticated decay conditions (see Appendix for details). However, the performance of our automatically discovered optimizer rivals APGD across various defense methods, despite of having a much simpler form (see Appendix for details). This result demonstrates the potential of applying our framework to reduce the need of human expertise in designing optimizers for diverse tasks.

4.5 Node classification on graphs

We next test our framework for optimizing graph neural networks to classify nodes on graphs. The model of interest is Graph Attention Network (GAT) [21], one of the most widely used architectures in graph learning tasks. We compare our method against Adam [9] - the standard optimizer for optimizing GATs - on five commonly used graph datasets: OGBN-Product [49], Cora [4], Citeseer [3], PubMed [6], and PPI [18]. Among them, OGBN-Product is the largest in scale, consisting of 2,449,029 nodes. Since standard GATs cannot scale to this dataset, we instead adopt an adaptation of cluster-GCN [28] to GAT as the testbed, termed Cluster-GAT. Cluster-GAT trains standard GAT on smaller partitions of the original graph, thereby allow-

Table 3: Performance of our discovered optimizers against Adam on GATs on five commonly used Graph datasets of diverse size. Results that use the same GAT implementations are grouped together.

Dataset	**Adam**	**Ours**
Products	77.49% ± 0.56†	**80.15% ± 0.16**
Cora	84.72% ± 0.32	**85.20% ± 0.19**
Citeseer	71.70% ± 1.03	**73.10% ± 0.43**
PubMed	78.20% ± 0.22	**79.25% ± 0.70**
PPI	97.53% ± 0.45‡	**98.13% ± 0.10**‡

† Our reproduced accuracy using ogbn-leaderboard's implementation is lower than the displayed number (79.23% ± 0.78).
‡ F1 Score

ing the model to be applied to large-scale graphs. We refer the reader to the Appendix for detailed descriptions of all GAT implementations and experimental setups.

The results are summarized in Table 3. On all datasets, our search algorithm is able to discover optimizers that outperform Adam. An interesting observation is that the top-performing optimizers discovered for this task almost always contain $sign(\cdot)$ operators, revealing the potential of adopting sign-based optimizers to improve the training of graph neural networks.

4.6 BERT fine-tuning on NLP datasets

We also evaluate the proposed framework on BERT finetuning task on GLUE benchmark [25]. For this task, we follow all configurations of the Hugging-Face [56] implementations: we finetune a pretrained BERT (base cased) model for 3 epochs on Cola [43], STS-B [16] and RTE [7] dataset, and 5 epochs on MRPC [5] and WNLI [25] dataset. The batch size is set to 32. We compare our discovered optimizers with the default AdamW [19]. As shown in Table 4, our automatically discovered optimizers outperform AdamW on all datasets.

Table 4: Performance of our discovered optimizers for BERT finetuning on GLUE tasks.

Dataset	AdamW	Ours
Cola	$59.56 \pm 2.04^{\star}$	$\mathbf{60.89 \pm 1.33}^{\star}$
MRPC	$82.84 \pm 0.57^{\ddagger}$	$\mathbf{86.64 \pm 0.94}^{\ddagger}$
STS-B	$87.80 \pm 1.14^{\dagger}$	$\mathbf{88.91 \pm 0.30}^{\dagger}$
RTE	$65.97 \pm 1.56^{\ddagger}$	$\mathbf{68.50 \pm 1.93}^{\ddagger}$
WNLI	$53.17 \pm 5.49^{\ddagger}$	$\mathbf{56.34 \pm 0.00}^{\ddagger}$

* Mathews Correlation.
† Spearman Correlation.
‡ Accuracy (%).

5 Ablation study

In this section, we ablate the proposed framework using the MNISTNET task. All experiments are repeated over 4 random seeds to account for randomness in the search phase.

Random search baseline We study the effectiveness of our MCT algorithm alone by comparing it with random sampling. Concretely, instead of traversing the tree based on MC scores, we randomly generate all optimizers from the root. Everything else in our framework remains unchanged, including our rejection sampling and equivalent-form detection techniques. This is equivalent to Random Search on our search space. As shown in Table 5, MCT algorithm outperforms Random Search baseline by a sizable margin, showing that the Monte Carlo node scoring schema can indeed guide the traversal towards promising branches of the tree.

Table 5: Performance comparison of Random Search and MCT.

Method	Training Loss (sum)	Test Accuracy
Random	60.25 ± 2.99	$88.02\% \pm 1.53$
MCT	$\mathbf{59.25 \pm 2.50}$	$\mathbf{89.43\% \pm 0.85}$

Score thresholding As discussed in prior sections, score thresholding is important to the performance of the MCT algorithm. To verify this, we ablate this technique by disabling it in our framework while keeping everything else the same. Without score thresholding, the cumulative training loss of the proposed optimizers raises from 59.25 ± 2.50 to 60.25 ± 2.87, similar to that of random search.

6 Conclusion

Despite the recent advancement of practical AutoML systems in automatizing the design of architectures, data augmentation policies, and hyperparameters, progress in automated discovery of optimizers is still inadequate due to the limitations of prior methods in terms of 1). efficiency, 2). generalization, and 3). scalability. In this paper, we introduce the first optimizer search framework that meets all these criteria, allowing it to be directly applied to the tasks of interest. The proposed framework demonstrates promising results across a variety of tasks, from image classification, adversarial attack, to graph learning and BERT finetuning. Our method by-no-mean solves the optimizer search problem, as there is plenty of room for improvement on the algorithm and search space; Rather, our goal is to open up a new possibility for future development in non-parametric optimizer search methods. We hope the proposed framework could democratize research and applications of automated optimizer search, and stimulate interest among researchers and practitioners.

7 Limitations

Our view of this work is as a starting point of an efficient, scalable, and generalizable framework for optimizer search. And we expect plenty of room for improvement for future works. For instance, we identify the following concrete limitations of the method:

1. We use pre-computed momentum terms as input to our search space. This is a practice borrowed from NOS-RL. Adding commonly used terms ease the job of the search algorithm because it does not have to rediscover them from scratch every time. However, searching for novel momentum update rules could potentially help to find even stronger optimizers. In principle, our framework allows it: one can do this by inserting an operator with its own internal state. This would serve as a direction for future explorations.
2. Identifying proper hyperparameters for an optimizer is essentially for evaluation. In the current work, we use a simple grid search to discover the best learning rate for an optimizer. While it works fine for our tasks, this could potentially be suboptimal as it might underestimate some optimizers. Leveraging advanced fast HPO during the search phase could be another direction to explore.
3. Although our framework is 100x faster than the comparable method (NOS-RL), it still requires 128 evaluations in the search phase. These evaluations can be largely parallelized. But potentially, the efficiency can be improved further with better search algorithms, more train-free tests, knowledge transfer, e.t.c.

Reproducibility & ethics statements

Reproducibility We have specified the setup for ours experiments in the main paper and Appendix, including settings for each task and hyperparameters for our method. The code and the optimizers found by our methods will also be published on Github upon acceptance to encourage future development. Before then, a copy of our code is included in the supplementary material for reference.

Ethics We are not aware of any potential ethical concerns regarding our work.

Acknowledgments and Disclosure of Funding

This work is supported in part by NSF under IIS-2008173, IIS-2048280, an Okawa research grant and a Google research scholar award.

References

[1] Miles Cranmer. Pysr: Fast & parallelized symbolic regression in python/julia, september 2020. *URL https://doi. org/10.5281/zenodo*, 4052869.

[2] David H Wolpert and William G Macready. No free lunch theorems for optimization. *IEEE transactions on evolutionary computation*, 1(1):67–82, 1997.

[3] C Lee Giles, Kurt D Bollacker, and Steve Lawrence. Citeseer: An automatic citation indexing system. In *Proceedings of the third ACM conference on Digital libraries*, pages 89–98, 1998.

[4] Andrew Kachites McCallum, Kamal Nigam, Jason Rennie, and Kristie Seymore. Automating the construction of internet portals with machine learning. *Information Retrieval*, 3(2):127–163, 2000.

[5] Bill Dolan and Chris Brockett. Automatically constructing a corpus of sentential paraphrases. In *Third International Workshop on Paraphrasing (IWP2005)*, 2005.

[6] Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Galligher, and Tina Eliassi-Rad. Collective classification in network data. *AI magazine*, 29(3):93–93, 2008.

[7] Luisa Bentivogli, Peter Clark, Ido Dagan, and Danilo Giampiccolo. The fifth pascal recognizing textual entailment challenge. In *TAC*, 2009.

[8] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[9] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[10] Marcin Andrychowicz, Misha Denil, Sergio Gomez, Matthew W Hoffman, David Pfau, Tom Schaul, Brendan Shillingford, and Nando De Freitas. Learning to learn by gradient descent by gradient descent. *Advances in neural information processing systems*, 29, 2016.

[11] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[12] Ke Li and Jitendra Malik. Learning to optimize. *arXiv preprint arXiv:1606.01885*, 2016.

[13] Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. Neuro-symbolic program synthesis. *arXiv preprint arXiv:1611.01855*, 2016.

[14] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. *arXiv preprint arXiv:1605.07146*, 2016.

[15] Irwan Bello, Barret Zoph, Vijay Vasudevan, and Quoc V Le. Neural optimizer search with reinforcement learning. In *International Conference on Machine Learning*, pages 459–468. PMLR, 2017.

[16] Daniel Cer, Mona Diab, Eneko Agirre, Inigo Lopez-Gazpio, and Lucia Specia. Semeval-2017 task 1: Semantic textual similarity-multilingual and cross-lingual focused evaluation. *arXiv preprint arXiv:1708.00055*, 2017.

[17] Yutian Chen, Matthew W Hoffman, Sergio Gómez Colmenarejo, Misha Denil, Timothy P Lillicrap, Matt Botvinick, and Nando Freitas. Learning to learn without gradient descent by gradient descent. In *International Conference on Machine Learning*, pages 748–756. PMLR, 2017.

[18] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. *Advances in neural information processing systems*, 30, 2017.

[19] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

[20] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. *arXiv preprint arXiv:1706.06083*, 2017.

[21] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.

[22] Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning. In *ICLR*, 2017.

[23] Stefan Falkner, Aaron Klein, and Frank Hutter. Bohb: Robust and efficient hyperparameter optimization at scale. In *International Conference on Machine Learning*, pages 1437–1446. PMLR, 2018.

[24] Lazar Valkov, Dipak Chaudhari, Akash Srivastava, Charles Sutton, and Swarat Chaudhuri. Houdini: Lifelong learning as program synthesis. *Advances in Neural Information Processing Systems*, 31, 2018.

[25] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R Bowman. Glue: A multi-task benchmark and analysis platform for natural language understanding. In *International Conference on Learning Representations*, 2018.

[26] Sirui Xie, Hehui Zheng, Chunxiao Liu, and Liang Lin. Snas: stochastic neural architecture search. *arXiv preprint arXiv:1812.09926*, 2018.

[27] Yair Carmon, Aditi Raghunathan, Ludwig Schmidt, John C Duchi, and Percy S Liang. Unlabeled data improves adversarial robustness. *Advances in Neural Information Processing Systems*, 32, 2019.

[28] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. Cluster-gcn: An efficient algorithm for training deep and large graph convolutional networks. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 257–266, 2019.

[29] Xuanyi Dong and Yi Yang. Searching for a robust neural architecture in four gpu hours. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 1761–1770, 2019.

[30] Kevin Ellis, Maxwell Nye, Yewen Pu, Felix Sosa, Josh Tenenbaum, and Armando Solar-Lezama. Write, execute, assess: Program synthesis with a repl. *Advances in Neural Information Processing Systems*, 32, 2019.

[31] Logan Engstrom, Andrew Ilyas, Hadi Salman, Shibani Santurkar, and Dimitris Tsipras. Robustness (python library), 2019.

[32] Golnaz Ghiasi, Tsung-Yi Lin, and Quoc V Le. Nas-fpn: Learning scalable feature pyramid architecture for object detection. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 7036–7045, 2019.

[33] Yufan Jiang, Chi Hu, Tong Xiao, Chunliang Zhang, and Jingbo Zhu. Improved differentiable architecture search for language modeling and named entity recognition. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 3585–3590, 2019.

[34] Jacob Devlin Ming-Wei Chang Kenton and Lee Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of NAACL-HLT*, pages 4171–4186, 2019.

[35] Guillaume Lample and François Charton. Deep learning for symbolic mathematics. In *International Conference on Learning Representations*, 2019.

[36] Chuming Li, Xin Yuan, Chen Lin, Minghao Guo, Wei Wu, Junjie Yan, and Wanli Ouyang. Am-lfs: Automl for loss function search. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 8410–8419, 2019.

[37] Sungbin Lim, Ildoo Kim, Taesup Kim, Chiheon Kim, and Sungwoong Kim. Fast autoaugment. *Advances in Neural Information Processing Systems*, 32, 2019.

[38] Chenxi Liu, Liang-Chieh Chen, Florian Schroff, Hartwig Adam, Wei Hua, Alan L Yuille, and Li Fei-Fei. Auto-deeplab: Hierarchical neural architecture search for semantic image segmentation. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 82–92, 2019.

[39] Hanxiao Liu, Karen Simonyan, and Yiming Yang. Autoaugment: Learning augmentation policies from data. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2019.

[40] Hanxiao Liu, Karen Simonyan, and Yiming Yang. DARTS: Differentiable architecture search. In *International Conference on Learning Representations*, 2019.

[41] Luke Metz, Niru Maheswaranathan, Jeremy Nixon, Daniel Freeman, and Jascha Sohl-Dickstein. Understanding and correcting pathologies in the training of learned optimizers. In *International Conference on Machine Learning*, pages 4556–4565. PMLR, 2019.

[42] Yisen Wang, Difan Zou, Jinfeng Yi, James Bailey, Xingjun Ma, and Quanquan Gu. Improving adversarial robustness requires revisiting misclassified examples. In *International Conference on Learning Representations*, 2019.

[43] Alex Warstadt, Amanpreet Singh, and Samuel R Bowman. Neural network acceptability judgments. *Transactions of the Association for Computational Linguistics*, 7:625–641, 2019.

[44] Daoyuan Chen, Yaliang Li, Minghui Qiu, Zhen Wang, Bofang Li, Bolin Ding, Hongbo Deng, Jun Huang, Wei Lin, and Jingren Zhou. Adabert: Task-adaptive bert compression with differentiable neural architecture search. *arXiv preprint arXiv:2001.04246*, 2020.

[45] Francesco Croce, Maksym Andriushchenko, Vikash Sehwag, Edoardo Debenedetti, Nicolas Flammarion, Mung Chiang, Prateek Mittal, and Matthias Hein. Robustbench: a standardized adversarial robustness benchmark. *arXiv preprint arXiv:2010.09670*, 2020.

[46] Francesco Croce and Matthias Hein. Reliable evaluation of adversarial robustness with an ensemble of diverse parameter-free attacks. In *ICML*, 2020.

[47] Lukasz Dudziak, Thomas Chau, Mohamed S. Abdelfattah, Royson Lee, Hyeji Kim, and Nicholas D. Lane. Brp-nas: Prediction-based nas using gcns. In *NeurIPS*, 2020.

[48] Sven Gowal, Chongli Qin, Jonathan Uesato, Timothy Mann, and Pushmeet Kohli. Uncovering the limits of adversarial training against norm-bounded adversarial examples. *arXiv preprint arXiv:2010.03593*, 2020.

[49] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. Open graph benchmark: Datasets for machine learning on graphs. *Advances in neural information processing systems*, 33:22118–22133, 2020.

[50] Abhinav Mehrotra, Alberto Gil CP Ramos, Sourav Bhattacharya, Łukasz Dudziak, Ravichander Vipperla, Thomas Chau, Mohamed S Abdelfattah, Samin Ishtiaq, and Nicholas Donald Lane. Nas-bench-asr: Reproducible neural architecture search for speech recognition. In *International Conference on Learning Representations*, 2020.

[51] Esteban Real, Chen Liang, David So, and Quoc Le. Automl-zero: Evolving machine learning algorithms from scratch. In *International Conference on Machine Learning*, pages 8007–8019. PMLR, 2020.

[52] Vikash Sehwag, Shiqi Wang, Prateek Mittal, and Suman Jana. Hydra: Pruning adversarially robust neural networks. *Advances in Neural Information Processing Systems*, 33:19655–19666, 2020.

[53] Ameesh Shah, Eric Zhan, Jennifer Sun, Abhinav Verma, Yisong Yue, and Swarat Chaudhuri. Learning differentiable programs with admissible neural heuristics. *Advances in neural information processing systems*, 33:4940–4952, 2020.

[54] Silviu-Marian Udrescu and Max Tegmark. Ai feynman: A physics-inspired method for symbolic regression. *Science Advances*, 6(16):eaay2631, 2020.

[55] Ruochen Wang, Minhao Cheng, Xiangning Chen, Xiaocheng Tang, and Cho-Jui Hsieh. Rethinking architecture selection in differentiable nas. In *International Conference on Learning Representations*, 2020.

[56] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online, October 2020. Association for Computational Linguistics.

[57] Eric Wong, Leslie Rice, and J Zico Kolter. Fast is better than free: Revisiting adversarial training. *arXiv preprint arXiv:2001.03994*, 2020.

[58] Dongxian Wu, Shu-Tao Xia, and Yisen Wang. Adversarial weight perturbation helps robust generalization. In *NeurIPS*, 2020.

[59] Tianlong Chen, Xiaohan Chen, Wuyang Chen, Howard Heaton, Jialin Liu, Zhangyang Wang, and Wotao Yin. Learning to optimize: A primer and a benchmark. *arXiv preprint arXiv:2103.12828*, 2021.

[60] Xiangning Chen, Ruochen Wang, Minhao Cheng, Xiaocheng Tang, and Cho-Jui Hsieh. DrNAS: Dirichlet neural architecture search. In *ICLR*, 2021.

[61] Guofeng Cui and He Zhu. Differentiable synthesis of program architectures. *Advances in Neural Information Processing Systems*, 34, 2021.

[62] Shoukang Hu, Ruochen Wang, HONG Lanqing, Zhenguo Li, Cho-Jui Hsieh, and Jiashi Feng. Generalizing few-shot nas with gradient matching. In *International Conference on Learning Representations*, 2021.

[63] Mikel Landajuela, Brenden K Petersen, Sookyung Kim, Claudio P Santiago, Ruben Glatt, Nathan Mundhenk, Jacob F Pettit, and Daniel Faissol. Discovering symbolic policies with deep reinforcement learning. In *International Conference on Machine Learning*, pages 5979–5989. PMLR, 2021.

[64] Brenden K Petersen, Mikel Landajuela, T Nathan Mundhenk, Claudio P Santiago, Soo K Kim, and Joanne T Kim. Deep symbolic regression: Recovering mathematical expressions from data via risk-seeking policy gradients. In *Proc. of the International Conference on Learning Representations*, 2021.

[65] Nicholas Roberts, Mikhail Khodak, Tri Dao, Liam Li, Christopher Ré, and Ameet Talwalkar. Rethinking neural operations for diverse tasks. *Advances in Neural Information Processing Systems*, 34, 2021.

[66] Paul Vicol, Luke Metz, and Jascha Sohl-Dickstein. Unbiased gradient estimation in unrolled computation graphs with persistent evolution strategies. In *International Conference on Machine Learning*, pages 10553–10563. PMLR, 2021.

[67] Ruochen Wang, Xiangning Chen, Minhao Cheng, Xiaocheng Tang, and Cho-Jui Hsieh. RANK-NOSH: Efficient predictor-based architecture search via non-uniform successive halving. In *ICCV*, 2021.

[68] Wenqing Zheng, Tianlong Chen, Ting-Kuei Hu, and Zhangyang Wang. Symbolic learning to optimize: Towards interpretability and scalability. In *International Conference on Learning Representations*, 2021.

[69] Xiangning Chen, Chen Liang, Da Huang, Esteban Real, Yao Liu, Kaiyuan Wang, Cho-Jui Hsieh, Yifeng Lu, and Quoc V Le. Evolved optimizer for visio. In *AutoML 2022 Workshop*, 2022.

[70] Yu A Shreider. *The Monte Carlo method: the method of statistical trials*, volume 87. Elsevier, 2014.

Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes]
   (b) Did you describe the limitations of your work? [Yes] See Section 6
   (c) Did you discuss any potential negative societal impacts of your work? [Yes] We are not aware of any negative societal impacts of this work.
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [N/A]
   (b) Did you include complete proofs of all theoretical results? [N/A]
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes]
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes] Due to space limits, some of these training details are included in the Appendix
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [Yes] The error bars for all optimizers are shown in the main text. The only exception is MNISTNET task, where we removed error bar for a cleaner plot. We will display the error bar for this task in the Appendix.
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes]
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [Yes]
   (b) Did you mention the license of the assets? [Yes]
   (c) Did you include any new assets either in the supplemental material or as a URL? [Yes]
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A]
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A]
5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]

A Appendix

A.1 Pseudocode for our search algorithm

Our framework follows a standard search pipeline:

1. Candidate proposal: the search algorithm samples an optimizer from the search space.
2. Candidate evaluation: The optimizer is evaluated from scratch, by using it to optimize a model and obtaining the performance of the model.
3. Search: The optimizer score is used to guide the search algorithm to propose new optimizers.
4. Loop: Repeat step 1 - 3 until a predefined search budget is reached.
5. Finally, the Top optimizers visited by the search algorithm will be returned.

This procedure is commonly used in other AutoML domains, such as Neural Architecture Search [47, 67] and Hyperparameter Optimization [23]. Algorithm 1 and 2 summarize the complete search process.

Algorithm 1: MCT algorithm

```
Input: Candidate set $\mathcal{A}$, constraints $\mathcal{C}$, operator set $\mathcal{O}$, maximum super-tree depth $D$, maximum
  traversal level $L$, MC sample size $M$ for each level, score threshold $\rho$, proposal size $K$.
Main search:
for level in 1 to $L$ do
    current node $v_c$ = root node ;            // root node hosts an empty update rule
    score dict $S = \emptyset$ ;          // scores of optimizers generated from stem nodes
    number of samples $m = 1$
    while $m <= M$ do
        $u$ = randomly pick a child of $v_c$, by inserting an operator $o \in \mathcal{O}$ to $v_c$ under constraint $C$;
        $\phi = unroll(u, \mathcal{C}, \mathcal{O}, D)$;
        if not $descent\_test(\phi)$ then
            continue;
        $s_\phi = evaluate(\phi)$ ;        // the score for early stopped $\phi$ are set to $\rho$
        if $s_\phi > \rho$ then
            register $(u, s_\phi)$ in $\mathcal{S}$;
            register $(\phi, s_\phi)$ in $\mathcal{A}$;
            $m$ += 1;
    for each node $u$ in $\mathcal{S}$ do
        if $u$ is a non-leaf node then
            compute the average score of $u$;
        else
            set the score of $u$ to 0;
    $v_c$ = node with the best score as computed above ;      // move on to the next level
return: $Top_K(\mathcal{A})$;
```

B Set of operators used for constructing the search space

Inspired by NOS-RL, we adopt the following set of mathematical operators in our experiments:

- Unary operators: $-(\cdot)$, $exp(\cdot)$, $log(|\cdot|)$, $\sqrt{|\cdot|}$, $clip_{0.003}(\cdot)$, $drop_{0.1}(\cdot)$, $sign(\cdot)$
- Binary operators: $+$, $-$, $\times$, $/$, $pow(\cdot, \cdot)$

Algorithm 2: Pseudocode for the unrolling step

```
1  Input: Stem node $v$, constraint $\mathcal{C}$, operator set $\mathcal{O}$, maximum super-tree depth $D$
2  Unroll:
3  set current node $v_c = v$
4  while True do
5      $v_c$ = randomly pick a child of $v_c$, by inserting an operator $o \in \mathcal{O}$ to $v_c$ under constraint $C$;
6      if $\phi_{v_c}$ is a complete update rule then
7          break;
8      else if length($\phi_{v_c}$) == $D$ then
9          $v_c = v$ ;                                   // restart unrolling
10         continue;
11 return: $\phi_{v_c}$
```

- Input Operators: g, g^2, g^3, m_1, m_2, m_3, $sign(g)$, $sign(m_1)$, $Adam$, $RMSprop$, 1, 2, ld, cd, rd

Here, m_1, m_2, m_3 denote the first, second and third order momentum respectively, and ld, cd, rd denote linear decay, cosine decay, and restart decay [15]:

$$\textbf{linear decay}: \quad 1 - \frac{t}{T}$$
$$\textbf{cosine decay}: \quad 0.5 * (1 + \cos(2\pi n \frac{t}{T}))$$
$$\textbf{restart decay}: \quad 0.5 * (1 + \cos(\pi \frac{(tn)\%T}{T}))$$

where t and T are the current and maximum step. Following NOS-RL, we use $n = 0.5$ for cosine decay and $n = 20$ for restart decay. We set the bound for $clip$ operator to 0.003, and the dropout ratio to 0.1 for $drop$ operator. Note that one can always include more options of these values by adding new operator variants to the space (e.g. $drop_{0.3}()$ with dropout ratio set to 0.3). For all input operators, we use their default PyTorch implementations and hyper-parameters. The only exception is the learning rates for Adam and RMSprop. We found that under the default learning rate, the norm of Adam and RMSprop is sometimes quite small compared with other operands such as $sign(g)$, making them potentially less effective as a submodule of some optimizers. Therefore, we raise their default learning rate by $3\times$ in our experiment. Note that we make one minor adjustment to the set for the ConvNet task: g^3, m_3, $Adam$, and $RMSprop$ are removed as they rarely show up on the tops optimizers.

Our set of operators is a subset of the full operator set presented in Section 4.1 of the NOS-RL paper. However, note that NOS-RL also uses much smaller subsets rather than the full set to conduct the search. We refer the readers to "Further discussions on NOS-RL baseline" in Appendix D.3 for more details.

Suggestions on constructing operator sets for future tasks Our provided set of operators and heuristics are quite generic. Empirically, we found that the operator set is good enough for constructing high-performant optimizers for all tasks used in this paper, including MNIST, CNN training, BERT finetuning, GNN training, and adversarial attacks. And we use the same on-the-fly constraints, train-free test, and score threshold for all experiments.

For future tasks, we recommend starting with the provided set and incorporating new operators when necessary. For inspiration on what to add, the user might look into 1). what extra mathematical operations are used in existing optimizers for this task 2). the nature of the task (e.g. some might require max or sinusoid functions).

With an augmented operator set, other components in our algorithm can largely remain the same. 1). our train-free test, score thresholding, and math equivalent detection are generic and independent of the operator set 2). It could be beneficial to augment the on-the-fly constraints slightly when the new operators are added. It is because the on-the-fly constraints are mainly used for eliminating mathematical redundancies, and new redundancies might surface when new operators are included. But this should be an extra gain. We encourage the users to follow our general guidelines in Section 2.4 for spotting these redundancies.

C Choice of hyperparamters for the search algorithm

We will elaborate more on our choice of hyperparameters for the search phase. Note that our search hyperparameters are held fixed throughout all our experiments, across all the tasks.

Score threshold The purpose of score thresholding is mainly to reject apparently poor optimizers - those that lead to near-random-guess accuracy or exploding loss. So we simply set it (20%) to be slightly higher than random guessing accuracy and used it for all our experiments.

Number of samples in Monte Carlo estimation We set and fix the number of total samples in Monte Carlo estimation to 32 - a multiple of 8 for parallelization. If this number is too small, then the MC scores for each branch would be of high variance; Setting this number too big does not hurt performance, but also might not be necessary as it consumes extra budget.

Super Tree Depth We choose ten because we found that most top optimizers are within this range. We also experiment with increased depth (15, 20) but only find marginal gains sometimes. The discovered longer optimizers are also less interpretable.

The number of traversal levels We use a fixed number to fix the same budget for all experiments (recall that num_mc_samples * num_levels = total budget) (line 267). But in practice, this hyperparameter needs not to be preset or tuned; One can just progressively increase it and stop when there is no further gain in terms of the proposed optimizer's performance.

Learning rates for optimizer evaluation We perform simple grid search to identify a suitable learning rate for each optimizer. We use one main grid for our experiments, **without** tuning it just to get more favorable results. However, we adjust the grid slightly for different tasks for the following reasons:

1. We use $\{1e^{-5}, 1e^{-4}, 1e^{-3}, 1e^{-2}, 1e^{-1}, 1\}$ for ConvNet task proposed in NOS-RL. This grid is too sparse, but it is adopted in NOS-RL (Sec 4.3). So we have to use the same grid for a fair comparison with NOS-RL.
2. For all other tasks, we use this grid: $\{0.0001, 0.0003, 0.0006, 0.001, 0.003, 0.006, 0.01, 0.03, 0.06, 0.1, 0.3, 1.0\}$. One can always use the full grid. However, this grid is sometimes unnecessarily long for some tasks - the first or the last few rates are never selected. Due to the sheer size of experiments in this paper and limited resources, we often shrink this grid mildly to speed up the runs a little. The exact truncated grids for each task are reported in the following section.
3. We always want to ensure that the grid covers the provided learning rate of our baseline optimizers for a fair comparison. For the BERT task, since the baseline (AdamW)'s learning rate is set to $2e^{-5}$ (outside this grid), we extend our grid to cover it, avoiding underestimating AdamW.

D More details on experimental settings

D.1 General settings for our search algorithm

Search configurations For all experiment, we allow 4 levels of traversal and set the number of Monte Carlo samples for each level to 32. This amounts to a total budget of 128 evaluations. The maximum depth for the super-tree is set to 10. The evaluation of Monte Carlo samples for each level of traversal are completely independent, and therefore can be easily parallelized onto multiple GPUs. As mentioned in the main text, we also apply score thresholding during the Monte Carlo estimation. We use a universal threshold of 10 for losses, 20% for accuracy and correlations. After the search phase, top 5 optimizers are usually proposed for further evaluations.

Early stopping We also early stop poor optimizers to speedup the search process. We use the following standard procedure for deciding whether to terminate the training of an optimizer: If the search signal is training loss, we track if the moving average of the training loss keeps increasing for certain amount of consecutive steps. If the search signal is accuracy or correlations, we check if the accuracy fails to reach the score threshold after 10% of training.

Constraints We use the following constraints during tree traversal: 1). $log(exp(\cdot))$ and $-(-(\cdot))$ are prohibited. 2). $sign(\cdot)$ must not be followed by $sign(\cdot)$, $sign(m_1)$, $sign(g)$, $clip_{0.003}(\cdot)$, 1, 2, ld, cd, and rd. 3). $\sqrt{|\cdot|}$ must not be followed by $sign$ and 1. 4). $clip_{0.003}(\cdot)$ must not be followed by $clip_{0.003}(\cdot)$, 1, 2, ld, cd, and rd.

D.2 Hand-digit classification with MNISTNET

Table 6: Performance of different optimizers on MNISTNET models.

Method	MNISTNET		MNISTNET-2Layer		MNISTNET-Big		MNISTNET-ReLU	
	Train Loss (Sum)	Test Acc	Train Loss (Sum)	Test Acc	Train Loss (Sum)	Test Acc	Train Loss (Sum)	Test Acc
SGD	364.96 ± 3.32	$93.09\% \pm 0.17$	638.23 ± 12.91	$92.27\% \pm 0.44$	334.72 ± 1.90	$93.88\% \pm 0.08$	317.33 ± 7.47	$93.56\% \pm 0.37$
SGD + Mom	276.26 ± 10.78	$93.07\% \pm 0.46$	358.61 ± 8.96	$93.05\% \pm 0.32$	207.54 ± 5.12	$95.29\% \pm 0.25$	265.15 ± 9.58	$94.03\% \pm 0.53$
Nesterov	248.96 ± 6.51	$93.53\% \pm 0.32$	317.86 ± 6.38	$93.32\% \pm 0.32$	192.03 ± 13.13	$95.35\% \pm 0.31$	283.50 ± 41.82	$92.95\% \pm 0.83$
Adam	327.15 ± 11.55	$91.54\% \pm 0.53$	403.07 ± 31.20	$90.69\% \pm 0.54$	219.25 ± 4.43	$94.29\% \pm 0.33$	273.02 ± 15.47	$92.56\% \pm 0.72$
RMSprop	269.48 ± 5.74	$93.72\% \pm 0.17$	336.99 ± 13.33	$93.44\% \pm 0.37$	230.69 ± 4.30	$95.01\% \pm 0.20$	280.28 ± 11.59	$93.61\% \pm 0.33$
L2LGD2	300.94 ± 12.49	$90.63\% \pm 0.14$	338.18 ± 11.69	$90.11\% \pm 0.30$	286.63 ± 8.33	$90.94\% \pm 0.32$	791.35 ± 55.13	$84.24\% \pm 1.49$
Ours	$\mathbf{237.76 \pm 5.34}$	$\mathbf{93.86\% \pm 0.23}$	$\mathbf{291.90 \pm 7.89}$	$\mathbf{93.75\% \pm 0.38}$	$\mathbf{186.17 \pm 6.68}$	$\mathbf{95.42\% \pm 0.16}$	$\mathbf{238.19 \pm 8.37}$	$\mathbf{94.29\% \pm 0.30}$

Task setting In this task, the goal is to minimize the cumulative training loss of a simple MLP (MNISTNET). All experimental setups (including model variants) and the LSTM-based optimizer baseline are borrowed from the open-sourced PyTorch implementation of L2LGD2[2] The default MNISTNET has one 20-dimensional hidden layers with Sigmoid activation. In addition, we also consider three other variants of MNISTNET: 1). MNISTNET-2Layer, which doubles the number of layers in MNISTNET; 2). MNISTNET-Big, which doubles the hidden layer dimension of MNISTNET; 3). MNISTNET-ReLU, which replaces the Sigmoid activation in MNISTNET with ReLU. All models are trained for 1000 steps with a batch size of 128 on the MNIST dataset. We use a fixed 50/50 split of training and testing set for MNIST.

Optimizer evaluation For each optimizer, the best learning rate is obtained from $\{0.0006, 0.001, 0.003, 0.006, 0.01, 0.03, 0.06, 0.1, 0.3, 1.0\}$. During the grid search, we train the network for 100 steps. After that, the network is retrained for 1000 steps with the best learning rate. We record the cumulative training loss and test accuracy of each optimizer for comparison. Note that following the L2LGD2 implementation, the grid search for the LSTM-based optimizer is applied at training time rather than test time.

Search setting Again, we follow the search settings implemented in the L2LGD2 codebase for our experiment. The search is conducted on the default MNISTNET by training it for 100 steps on the training split. Standard early stopping procedure is enabled for this task. Empirically, we found that roughly 7.2% optimizers are terminated.

Discovered optimizers We represent some of the discovered optimizer down below. Note that the forms of these optimizers are already simplified using the Sympy library.

mnist1: $m_1 + RMSprop * exp(Adam)$

mnist2: $m_1 * (exp(Adam) + exp(exp(Adam))) + RMSprop$

mnist3: $m_1 + RMSprop * rd^{g^3}$

Interestingly, the pattern $m_1 + RMSprop$ shows up quite frequently in the discovered optimizers for this task.

[2] `https://github.com/chenwydj/learning-to-learn-by-gradient-descent-by-gradient-descent`

More experimental results In addition to the convergence figures in the main text, we also present the results in tabular form. Table 6 summarizes both the cumulative training loss and test accuracy of the optimizers on four MNISTNET models. All models are trained for 1000 steps. Our discovered optimizer achieves the best cumulative training loss and test accuracy for all cases.

D.3 Image classification with ConvNet

Task setting The goal of this task is to train a ConvNet on CIFAR-10 dataset. Following NOS-RL, the ConvNet has two 32-filter 3x3 convolution layers, each followed by ReLU activation and batch normalization [15]. We use a fixed held-out validation set of 5000 images for grid search. Note that the held-out validation set is used throughout the search phase, and it will be added back to the training set during final evaluation of the proposed optimizers.

Optimizer evaluation The grid search is performed over $\{1e^{-5}, 1e^{-4}, 1e^{-3}, 1e^{-2}, 1e^{-1}, 1\}$ for 1 epoch of training, and the best learning rate is selected based on the accuracy on held-out validation set. After that, the optimizers are trained for a longer period of time (5 epochs). The batch size is set to 100.

Search setting For this task, we disable early stopping (i.e. all optimizers will be counted into the budget.), to establish fair comparisons with NOS-RL's search budget; The reason is as follows: Although NOS-RL also aggressively early stops poor optimizers, the authors added them back when plotting Figure 4 in their paper; And since our estimation of NOS-RL's search budget comes from Figure 4, it would be rational to also disable it in our experiment. Each evaluation takes about 3 minutes to finish. And the duration for the entire search phase is around 7 hours on a single RTX 2080ti.

Discovered optimizers Some of the discovered optimizers are shown below. Note that the forms of these optimizers are already simplified using Sympy library.

conv1: $cd * drop_{0.1}(g)/ld$

conv2: $cd * sign(m_1) * |m_2|^{\sqrt{|ld|}/2}$

conv3: $drop_1(cd * m_1)$

conv4: $m_1 * (rd + |g|) * exp(cd)$

Further discussions on NOS-RL baseline NOS-RL applies Reinforcement Learning to train a LSTM controller to generate optimizer update rules according to a predefined pattern. Due to the training difficulty, NOS-RL adopts a multi-config-multi-run search strategy: It conducts the search multiple times with different subset of operators (unknown) and different optimizer length (5,10,15,20). Out of all search runs with different configurations, two families of best optimizers are reported in the paper. This leads to several challenges that prevent us from obtaining exact comparisons with NOS-RL: 1). It is difficult to know or measure the exact search cost of NOS-RL due to its multi-config-multi-run strategy. The paper only mentions that a single search run can finish in one-day with heavily parallelism on Google's infrastructures. Therefore, the best we can do is to make an estimation based on Figure 4 in their paper, where it shows that the controller begins to converge at least after 10k evaluations. 2). The particular subsets of operators used during each search run is also unknown; The paper only mentions that the search spaces generated by these subsets typically contains 10^6 to 10^{11} update rules. As a result, we have to pick our own operator set to run the search on.

We conjecture that the main purpose of NOS-RL paper is to offer the discovered optimizer for practitioners to use, rather than providing a baseline to stimulate further developments of non-parametric optimizer search methods. This can be evidenced by its prohibitive search cost, and also by the fact that the source code is not released. The nature of NOS-RL, combined with aforementioned challenges, necessitate an open-sourced resource-friendly non-parametric optimizer search framework for the community, which we hope to provide in this work.

D.4 Adversarial Attack

Task setting Adversarial attack aims at finding a norm-bounded perturbation to the input space that misleads the model predictions. In this case, the parameter to be optimized is the data itself. We use the AutoAttack library[3] to implement our experiments for this task. The library contains a set of defense methods, as well as an implementation of the APGD optimizer that we used as the baseline. The attack is conducted on the default test split of CIFAR-10 dataset, which contains 10000 images. Our metric of choice is attack success rate. Concretely, if the perturbed image successfully mislead the model's prediction into a wrong class, then the attack is successful for that image. The success rate is thus the percentage of images that the optimizer successfully attacked.

Optimizer evaluation The search is conducted on the Carmon2019 method. We use $\{0.001, 0.003, 0.006, 0.01, 0.03, 0.06, 0.1, 0.3, 1\}$ to search for the best learning rate. The grid search is conducted on only 1000 test images. After that, the optimizers will be evaluated by training for 100 steps.

Search setting During the search phase, all optimizers are evaluated with only 20 steps, as it is usually enough to identify top optimizers. To further reduce the search cost, we use only 400 images for grid search and 4800 (400 * 12) images for evaluation. The search takes around one GPU day to finish on a single RTX 2080ti.

Discovered optimizers We present some of the discovered optimizers for adversarial attack down below.

attack1: $log(|cd + sign(g)|)$

attack2: $log(|cd + exp(g^3) * sign(g)|)$

attack3: $log(|ld + sign(RMSprop)|)$

As mentioned in the main text, we found that many top optimizers are log-based. More specifically, these optimizers often have the form of $log(|decay + sign(\cdot)|)$. The discovered log-based optimizers are also highly effective when transferred to other defense models, as showing in Table 2.

Discussion on Adaptive Projected Gradient Descent (APGD) optimizer APGD is currently the strongest manually design and tuned optimizer for adversarial attack. It consists of two parts: 1). a momentum update rule and 2). a dynamic learning rate decay schema. The momentum update rule takes the following form:

$$z^{(k+1)} = Proj_{B_\epsilon^\infty(x_o)}(x^{(k)} + \gamma^{(k)} sign(\nabla_x L(x^{(k)})))) \tag{1}$$

$$x^{(k+1)} = Proj_{B_\epsilon^\infty(x_o)}(x^{(k)} + (1-\mu)(z^{(k+1)} - x^{(k)}) + \mu(x^{(k)} - x^{(k-1)})) \tag{2}$$

where k is the current step and μ is the momentum ratio. The μ is tuned to be 0.25, much lower than the default momentum ratio for standard optimizers such as SGD and PGD. The dynamic learning rate decay schema halves the learning rate if a set of two conditions are satisfied at some predefined steps $\{w_j\}_{j=1}^{100}$:

$$\sum_{i=w_{j-1}}^{w_j-1} \mathbb{1}_{\mathcal{L}(x^{(i+1)}) < \mathcal{L}(x^{(i)})} < \rho * (w_j - w_{j-1}) \tag{3}$$

$$\gamma^{(w_j)} \equiv \gamma^{(w_{j-1})} \text{ and } \mathcal{L}_{min}^{(w_j)} \equiv \mathcal{L}_{min}^{(w_{j-1})} \tag{4}$$

where ρ is a threshold term and $\mathcal{L}$ denotes the loss function. The steps (w_j) to check for these conditions are set to $\{23, 42, 58, 71, 81, 88, 94, 100\}$. As we can see, APGD has a quite complicated form, and its design also packs a lot of human expertise. On the other hand, our automatically discovered optimizers are much simpler, while also rivaling the performance of APGD.

[3] https://github.com/fra31/auto-attack

D.5 Node classification on graphs

Task Setting We consider node classification task on graphs. The model of choice is Graph Attention Network (GAT). There exists many PyTorch implementations of GAT and its variants, each covers only some datasets. As a result, we have to use more than one codebase for this experiment. For training cluster-GAT on OGBN-Products dataset, we use the official implementation from OGBN library[4]. For training vanilla GAT on Cora dataset, we use pyGAT [5]. For training vanilla GAT on Citeseer, PubMed, and PPI dataset, we use the implementations from DGL library[6]. We follow the instructions provided in their README.md files to run all of our experiments. The only except is for Cora dataset, where we disable early stopping in the original implementation. The reason is that the default criteria often terminate training prematurely for our optimizers.

Optimizer evaluation The grid search is conducted over $\{0.0003, 0.0006, 0.001, 0.003, 0.006, 0.01, 0.03, 0.06\}$, as we found that most of the optimizers' best learning rates (including Adam) fall into this range. After the grid search, all optimizers are evaluated under 4 random seeds. As mentioned above, all other hyperparameters, including the total number of epochs, batch size, weight decay, e.t.c., are set to their default values as in the original codebases.

Search setting We deploy standard early stopping schema for this task. The percentage of early terminated optimizers is around 10% to 17% for vanilla GATs. We found that Cluster-GAT is much harder to optimize: roughly 25% of the optimizers are early terminated. The search is conducted on each dataset separately, and we will discuss the transferability of the discovered optimizers in the next paragraph. For Products dataset, we parallelize the search over 8 RTX 2080ti GPUs. All other datasets are ran on a single device. The search takes about 2 GPU days to finish for Products dataset, 1 GPU day for PPI and Cora, and 3 hours for PubMed and Citeseer.

Discovered optimizers We present some of the discovered optimizers on each dataset down below. Interestingly, we found that sign-based optimizers dominate the graph learning task.

products1: $ld * sign(m_1) - Adam$

products2: $ld * (sign(m_1) - RMSprop)$

cora: $sign(m_1) + m_3$

citeseer: $drop_{0.1}((sign(g) - m_1)/cd)$

pubmed: $sign(m_1) + sign(m_1 - drop_{0.1}(g^3))$

ppi: $drop_{0.1}(rd^2) * sign(m_1)$

More experimental results on the transfer setting Among the optimizers listed above, we found that "product2" optimizer exhibits the highest level of transferability. As shown in Table 7, it outperforms Adam on all but the citeseer dataset. Note that for graph learning task, there exists a non-negligible performance gap between transfer and direct search settings. We conjecture that it is because different graph dataset might indeed require different optimizers. The reason is as follows: Most graph neural networks, including GATs, adopt the message passing framework, where the features of neighboring nodes are passed to the target node through their edges in the forward pass. Since the connectivity of nodes are defined by the adjacency matrix in the dataset, the computation graph (and thus the learning process) is inherently

Table 7: Performance of our discovered optimizers against Adam on GATs on five commonly used Graph datasets of diverse size. Results that use the same GAT implementations are grouped together.

Dataset	Adam	products2
Products	$77.49\% \pm 0.56^{\dagger}$	$\mathbf{79.98\% \pm 0.17}$
Cora	$84.72\% \pm 0.32$	$\mathbf{84.87\% \pm 0.29}$
Citeseer	$71.70\% \pm 1.03$	$\mathbf{71.78\% \pm 0.51}$
PubMed	$\mathbf{78.20\% \pm 0.22}$	$77.12\% \pm 0.53$
PPI	$97.53\% \pm 0.45^{\ddagger}$	$\mathbf{98.38\% \pm 0.07}^{\ddagger}$

† Our reproduced accuracy using ogbn-leaderboard's implementation is lower than the displayed number (79.23% $\pm$ 0.78).
‡ F1 Score

[4] https://github.com/dmlc/dgl/tree/master/examples/pytorch/ogb/cluster-gat
[5] https://github.com/Diego999/pyGAT
[6] https://github.com/dmlc/dgl/tree/master/examples/pytorch/gat

encoded in the dataset itself. Moreover, some of the datasets and models we considered are inherently heterogeneous. For example, PPI is designed for inductive learning, whereas all other dataset are for transductive setting; and also the Cluster-GAT model we used for OGBN-Products dataset is inherently different from vanilla GATs.

D.6 BERT fine-tunning on NLP datasets

Task setting We use Hugging Face's official implementation of BERT finetuning task for our experiment[7]. The goal of this task is to finetune a pretrained BERT (base cased) model on a set of NLP datasets. Following the instructions on the official repo, we set the number of epochs to 5 for MRPC and WNLI, and 3 for CoLA, STS-B, and RTE dataset. We also observe that finetuning for more epochs generally harms the performance of all optimizers. The model is trained with a batch size of 32 on a single GPU. We refer the reader to Hugging Face's offical repo (link in the footnote) for more details on this task.

Optimizer evaluation We use $\{2e^{-5}, 0.0001, 0.0003, 0.0006, 0.001, 0.003, 0.006, 0.01\}$ for learning rate grid search. Note that this grid is intentionally shifted to cover Hugging Face's default learning rate for AdamW ($2e^{-5}$). After the grid search, the optimizers are trained for 4 random seeds with the best learning rate.

Search setting Similar to GAT task, our search is conducted on each dataset separately, and we will discuss the transferability of discovered optimizers later. We early stop optimizers if their performance (accuracy, Matthew's correlation, or Spearman's correlation) fall below the default threshold (0.2) during the grid search. Empirically, we found that roughly 13.5% optimizers are terminated. This rate is slightly higher than that of MNISTNET task (7.2%), because we are using the same threshold as MNISTNET task even though the accuracies (or correlations) on BERT fine-tuning tasks are much lower. For this task, we parallelize the search over 8 RTX A6000 GPUs. The search can be finished in less than 10 hours on all dataset.

Discovered optimizers We present some of the discovered optimizers on each dataset down below. Similar to those found on the GAT tasks, many optimizers are sign-based. Note that the power operator in cola2 optimizer might return NaN, which will be mapped to 0 by the sign function.

cola1: $drop_{0.1}(Adam + g^3)$

cola2: $sign(m_1^{clip_{0.003}(rd)+clip_{0.003}(sign(g))-sign(m_1)})$

mrpc: $drop_{0.1}(clip_{0.003}(sign(g) + sign(RMSprop) * rd))$

stsb: $sign(RMSprop + 2/(g + m_3))$

rte: $drop_{0.1}(clip_{0.003}(m_1 - \sqrt{|drop_{0.1}(g^3)|} * sign(m_1)))$

wnli: $sign(m_1) - m_3$

More experimental results on the transfer setting We found that both "cola2" and "rte" optimizer exhibits high level of transferability. Although they perform slightly worse than the optimizers directly searched on the target dataset, it still consistently outperforms AdamW by a sizable margin. The results are summarized in Table 8. Note that some of our reproduced results for AdamW is a bit different than the reported numbers on the Hugging Face repository. The reason is that we run each optimizer for 4 seeds and report the average results, whereas the official repository only records the number after a single run.

Table 8: Performance of our discovered "cola2" optimizer for BERT finetuning task. Results above baseline are bolded.

Dataset	AdamW	cola2 optimizer	rte optimizer
Cola	$59.56 \pm 2.04^\star$	$\mathbf{60.05 \pm 2.38}^\star$	$\mathbf{59.91 \pm 1.54}^\star$
MRPC	$82.84 \pm 0.57^\ddagger$	$\mathbf{85.48 \pm 0.74}^\ddagger$	$\mathbf{85.60 \pm 0.68}^\ddagger$
STS-B	$87.80 \pm 1.14^\dagger$	$\mathbf{87.90 \pm 0.28}^\dagger$	$\mathbf{87.91 \pm 0.98}^\dagger$
RTE	$65.97 \pm 1.56^\ddagger$	$\mathbf{66.52 \pm 1.83}^\ddagger$	$\mathbf{68.50 \pm 1.93}^\ddagger$
WNLI	$53.17 \pm 5.49^\ddagger$	$\mathbf{56.34 \pm 0.00}^\ddagger$	$\mathbf{55.28 \pm 1.83}^\ddagger$

* Mathews Correlation.
† Spearman Correlation.
‡ Accuracy (%).

[7] `https://github.com/huggingface/transformers/tree/main/examples/pytorch/text-classification`

D.7 License

The Hugging Face libary we used is licensed under Apache License 2.0. All other public repositories are licensed under MIT License.