Generalizable Symbolic Optimizer Learning

Xiaotian Song[1], Peng Zeng[1], Yanan Sun[1]*, and Andy Song[2]

[1] College of Computer Science, Sichuan University
[2] School of Computing Technologies, RMIT University
{songxt,zengpeng7}@stu.scu.edu.cn; ysun@scu.edu.cn; andy.song@rmit.edu.au

Abstract. Existing automated symbolic optimizer design methods necessitate the use of proxies, often resulting in significant performance degradation when transferring to a target domain. In this paper, we propose a learning based model called Symbolic Optimizer Learner (SOL) that can discover high-performance symbolic optimizers directly on the target. SOL is integrated with symbols and can be directly transformed into a symbolic optimizer. In addition, an unrolled optimization approach is introduced for SOL training. SOL can be embedded into the training process of neural networks, optimizing the target directly without any proxies. Our extensive experiments demonstrate the good performance and high generalizability of SOL through diverse tasks, ranging from classifications to adversarial attacks, from GNN to NLP tasks. On image classification, SOL achieved ∼5× speedup and ∼3% accuracy gain. On adversarial attacks, SOL achieved the best attack success rate across seven SOTA defense models. On GNN training, SOL discovered optimizers can outperform Adam on three different datasets. On BERT fine-tuning, SOL also outperformed AdamW on five benchmarks. The source code is available at https://github.com/songxt3/SOL.

Keywords: Optimizer · Neural Network · Learning to Optimize

1 Introduction

Optimizers play an important role in neural networks (NNs) training. As shown in Fig. 1, the convergence speed and accuracy of ResNet [17] using different optimizers are significantly different in CIFAR-10 classification [21]. Many efficient optimizers have been developed based on optimization theory and practical know-how, for example, first-order momentum-based methods like Stochastic Gradient Descent (SGD) [31], second-order momentum-based approaches like RMSprop [36], and hybrids like Adam [20]. With a plethora of available optimizers, picking the most suitable one for a specific task and configuring its parameters can be a time-consuming and laborious trial-and-error process. Therefore, there is a growing interest in the automatic design of optimizers tailored to specific tasks, which has gained prominence in recent years [1, 2, 5, 27, 28].

* Corresponding author.

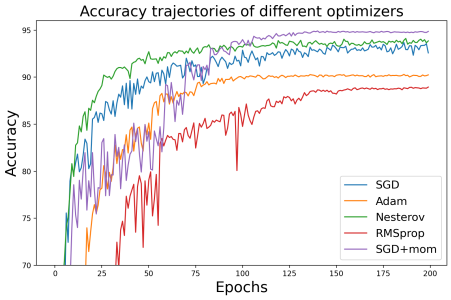

Fig. 1: Convergence speed and accuracy of ResNet-18 using different optimizers in the CIFAR-10 image classification task.

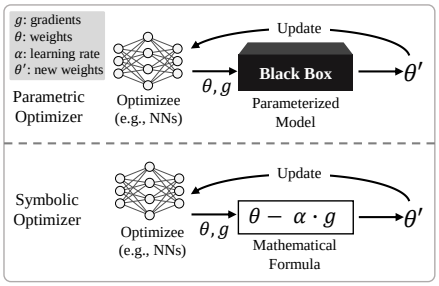

Fig. 2: Illustration of the difference between the symbolic optimizer and the parametric optimizer.

The rationale behind automated optimizer design is to create a suitable optimizer for a given task without iterative handcrafting [6]. Compared to manually-designed optimizers, such as Momentum [37] and Adam, auto-designed optimizers are expected to complete a set of optimizees from the same task domain faster, yet generating a higher-quality model with a similar amount of computing budget. Auto-designed optimizers are mainly in two categories: parametric optimizers and symbolic optimizers. The difference is shown in Fig. 2. The parametric optimizers commonly refer to parameterized models (e.g., NNs) trained to optimize a given task. They tend to overfit during training, resulting in poor generalizability to other tasks [40]. Sometimes they are even worse than handcrafted optimizers. In addition, the inherent complex and black-box nature of parameterized models make these optimizers hard to scale and interpret [48].

Compared to parametric optimizers, symbolic optimizers are in the form of mathematical formulas like manually designed optimizers. They are generally considered to have better generalizability [7], scalability [30], and interpretability [48]. However, existing auto-design symbolic optimizer methods generally search over the discrete space with Reinforcement Learning (RL) [2] or Evolutionary Computation (EC) [7, 29, 30], which require sampling many optimizers during the search process and evaluating every one of them on the given task. To reduce the search cost, these methods usually perform the search on a small proxy task and then transfer the generated optimizer onto the target tasks. The transfer process itself still needs manual tuning of hyperparameters [2]. That defeats the original purpose of automated optimizer design. Moreover, there is no guarantee of performance for these optimizers, if the manual tuning is absent [40]. It is desirable to have an automated design method to generate high-performance symbolic optimizers directly based on the target tasks of interest.

Hence we propose Symbolic Optimizer Learner (SOL), of which the inputs are the current states (e.g., gradient, momentum, etc.) in an NN training process, and the outputs are the updated weights of NN. Additionally, we offer a comprehensive set of symbols, including input variables and operators, as ingredients for

composing high-performance optimizers for diverse tasks. These symbols can be integrated as computational units in SOL. Consequently, we can directly extract symbols and their corresponding weights after training, obtaining a ready-to-use symbolic optimizer without extra processing.

Our main contributions are summarized below:

- We propose SOL that can directly learn and formulate a symbolic optimizer without post-processing. Compared to existing methods [1, 48], SOL can produce a scalable, generalizable, and interpretable symbolic optimizer.
- We introduce an unrolled optimization approach, which embeds SOL's learning into NN training. Unlike existing methods [2,29], no proxy is needed here. Optimzers can be directly built on tasks through backward propagation.
- We demonstrate that symbolic optimizers generated by SOL can outperform both handcrafted and auto-designed optimizers on a wide range of tasks, e.g., achieving significant improvement against Adam on various tasks.

2 Related Works

2.1 Hand-designed Optimizers

Optimizer is crucial for the training of deep learning models. Over the past, various hand-designed optimizers have been introduced for training NNs, achieving good performance, such as SGD [31], Momentum [37], Adagrad [12], RMSprop [36], Adam [20], etc. As shown in Table 1, these hand-designed optimizers are basically symbolic optimizers. Selecting a suitable one from them needs experience. Setting hyperparameters for these optimizers also requires experience, plus multiple trials to optimize performance. For example, in the PyTorch framework, the hyperparameters that need to be set when using the Adam optimizer include α, β_1, β_2, eps, weight_decay, amsgrad, and so on.

Table 1: Handcrafted optimizers: θ are neural network parameters; g represents the gradient; α, β_1, β_2, γ are hyperparameters of the optimization algorithm.

Name	Updated Rule
Adam	$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$, $v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$, $\hat{m}_t = m_t / (1-\beta_1^t)$, $\hat{v}_t = v_t / (1-\beta_2^t)$, $\Delta\theta_t = -\alpha \hat{m}_t \hat{v}_t^{-1/2}$
SGD	$\Delta\theta_t = -\alpha g_t$
SGD + Momentum	$m_t = \gamma m_{t-1} + (1-\gamma) g_t$, $\Delta\theta_t = -\alpha m_t$
Adagrad	$G_t = G_{t-1} + g_t^2$ $\Delta\theta_t = -\alpha g_t G_t^{-1/2}$
RMSprop	$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$, $\Delta\theta_t = -\alpha g_t v_t^{-1/2}$

2.2 Automated Optimizer Design

There are two types of auto-designed optimizers: *parametric optimizers* and *symbolic optimizers*. Pioneering work of the first category is also coined as Learning to Optimize (L2O) [6]. L2O in general aims to learn a parameterized model that takes the optimizee's local states (such as gradients and decays) as inputs. The output are the updates for the weights. For example, L2LGD2 [1] uses LSTM (Long Short-Term Memory) to design an optimizer that determines the direction and step size of the optimizee's gradients. Models used in L2O variants

are mainly recurrent neural networks [22], most of which [1, 24, 45] are based on LSTM architecture [19]. In L2O, an LSTM is unrolled to perform iterative updates and trained to find short optimization routes. One set of parameters are shared across all the unrolled steps. Another than LSTM, different types of NN models, such as MLP [28] and hierarchical recurrent neural networks [42], also appear in existing L2O studies. Despite of multiple practical benefits, L2O also has some limitations, such as the poor scalability on optimizees with many parameters [48], and the poor generalizability cross different tasks [6]. Moreover, L2O variants are often based on NNs, which are non interpretable black-box [14].

The second category, symbolic optimizers (aka non-parametric optimizers), search over a discrete space to find a good optimizer. Optimizers in this approach are usually represented by discrete structures (e.g., tree) or computer codes. Multiple search strategies based on RL and EC, have been proposed to perform the search. For instance, NOS-RL [2] leverages RL to learn a controller to produce update rules, represented by symbolic trees. AutoML-Zero [30] uses EC to search over a vast space of computer codes for the whole ML pipeline, including the optimizers. Existing methods based on RL and EC require numerous evaluation of optimizers that are sampled during the process. The evaluation process is costly, computationally prohibitive for practitioners to apply or to analyze [40]. Moreover, to reduce the overall cost, these methods tend to conduct search on a smaller proxy task, and then transfer the discovered optimizer to the target. Unfortunately, the transfer process often leads to a performance drop.

Hence it is desirable to have a method that can perform directly on the target to train the symbolic optimizer. *Symbolic L2O* [48] equips prior L2O methods with symbolic regression to obtain symbolic optimizers. This method first trains an NN for the optimization task, and then fits the model using a symbolic regression algorithm. Such approach does direct trains on the target, but the symbolic regression results in additional computational cost. In order to address the aforementioned issues, SOL is proposed as it can directly generate an optimizer after training, eliminating the need for symbolic regression.

3 Approach

3.1 Problem Formulation

An optimization problem can be mathematically formulated as $min_{\theta \in \Theta} l(\theta)$, where $l(\theta)$ represents the *optimizee*, θ is termed as parameters to be optimized in domain Θ. The algorithm to solve the problem is called *optimizer*. For a differentiable optimization problem, a standard optimizer takes an update at iteration t based on the gradient at the current point θ_t: $\theta_{t+1} = \theta_t - \alpha \times \phi(\bigtriangledown_\theta l(\theta_t))$, where α denotes the step size and ϕ is the update function. The design of update function ϕ differs significantly between existing optimizers. Traditional manually designed optimizers represent the update functions with mathematical equations. For example, the commonly used Adam optimizer is in the form of $\phi(\bigtriangledown_\theta l(\theta_t)) = m(\bigtriangledown_\theta l(\theta_t))/\sqrt{m((\bigtriangledown_\theta l(\theta_t))^2)}$, where $m(\cdot)$ is the momentum function with an internal state. In our method, ϕ is a special model, SOL.

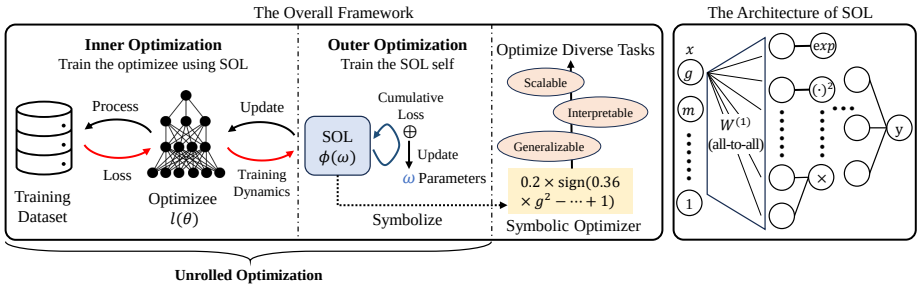

Fig. 3: Left: The overall framework. In the training stage, SOL is trained on the optimizee $l(\theta)$ via a bi-level process. After training, the optimized SOL is transformed into a symbolic form, which can be used to optimize more tasks. **Right:** The architecture of SOL. The inputs x of SOL are input features built from an optimizee's state including g, m, etc. The output y is the updates for the parameters. The hidden layer in SOL is a linear mapping followed by a set of computational units including exp, $(\cdot)^2$, etc.

3.2 Overall Framework

The overall framework of our method is depicted on the left of Fig. 3. SOL is trained on the optimize at first. Then, the learned optimizer is transformed into a symbolic form after training. The detailed steps are presented in Algorithm 1. To efficiently learn an optimizer, we formulate the learning process as a bi-level optimization problem with inner and outer, which can be formalized as:

$$\theta_{t+1} = \theta_t - \alpha \times \phi(\bigtriangledown_\theta l(\theta_t), \omega) \tag{1}$$

$$\omega_{\tau+1} = \omega_\tau - \beta \times \psi(\bigtriangledown_\omega L(\omega_\tau)) \tag{2}$$

Equation (1) represents the inner optimization. The weights θ of a target optimizee $l(\theta)$ is optimized by repeatedly applying an update function $\phi(\cdot)$. In our method, the update function is modeled by the learnable optimizer, SOL. It takes the optimizee's states at iteration t as inputs and outputs the updated states for the next iteration $t+1$. Notably, ω denotes the parameters of the learnable optimizer, which will be optimized in the outer optimization. Equation (2) represents the outer optimization. In the stage, the optimizer parameters ω are updated to minimize outer-objective $L(\omega_\tau)$, where τ is termed as the length of the inner loop. L aims to properly measure the optimizer performance. For convenience, it can be the sum value or the final value of the loss in the inner loop.

3.3 Symbolic Optimizer Learner (SOL)

As mentioned above, the objective of our proposed method is to learn a symbolic optimizer based on optimization tasks. We first define the symbolic search space upon manually designed optimizers and previous works [2, 20, 48]. More specifically, there are three types of operators usable in the symbolic optimizers:

Algorithm 1: Proposed Algorithm

```
Input: Training data, an optimizee, max epoch, length, unrolling length
Output: A symbolic optimizer
Initialize a SOL based on the optimize, and an optimizer for the SOL;
while epoch < max epoch do
    Reinitialize an optimizee;
    while i < length//unrolling length do
        while j < unrolling length do
            Update the optimizee by SOL according the loss on training data;
            Cumulate loss;
            j += 1;
        end
        Update the SOL according to the cumulative loss;
        i += 1;
    end
    epoch += 1;
end
Get the symbolic optimizer from the SOL;
Return the symbolic optimizer;
```

- a set of input operators built from optimizee's current states including gradients (e.g., g, m), decays (e.g., *linear decay*), and constants (e.g., 1, 2)
- a set of unary operators (e.g., $(\cdot)^2$, $exp(\cdot)$, $sign(\cdot)$)
- a set of binary operators (e.g., $+, \times$)

With the defined symbolic search space, we need to incorporate the symbols into a parametric model which can be trained and then transformed into symbolic equations. SOL, the special NN model, is proposed for this purpose. It is inspired by equation learning [32]. SOL is a variant of multi-layered feed-forward network with special computational units. Its architecture is illustrated on the right hand side of Fig. 3. Its inputs x are input features built from the optimizee's state including g, m, etc. The output is the updates of parameters in the optimizee, which is denoted as y. In SOL, an L-layer architecture contains L-1 hidden layers, each being a linear mapping followed by a non-linear transformations $f(\cdot)$. The linear mapping is similar to other NNs, but not the non-linear transformations. Activation functions in other NNs are often ReLU or Tanh, while $f(\cdot)$ is a set of computational units, including various unary and binary operators available in the search space. The computational units receive the components in $c^{(i)}$ as inputs and then produce the output $h^{(i)}$:

$$h^{(i)} = f(c^{(i)}) = \begin{bmatrix} f_1(c_1^{(i)}) \\ f_2(c_2^{(i)}) \\ \cdots \\ f_{n_h}(c_{n_c-1}^{(i)}, c_{n_c}^{(i)}) \end{bmatrix} \tag{3}$$

The final layer, i.e., L^{th}, does not have the activation function, so it computes the output by a linear read-out $y = W^{(L)}h^{(L-1)} + b^{(L)}$. During training, the weights W of the SOL are iteratively updated to reduce the loss. After training, the computational units associated with high weights are subsequently extracted to create the symbolic optimizers. The input features of SOL preserve the current training state of the optimize. Our observation from the experiments suggests that a reasonable increase of input features can significantly improve the optimizer's performance. Moreover, the computational units can be added and duplicated within each layer. This design reduces SOL's sensitivity to random initialization and fosters a smoother optimization landscape, thereby reducing the network's vulnerability to be trapped in local minima.

By adjusting the number of hidden layers, SOL can be adapted to different optimization tasks. It is worth noting that, for most tasks, shallow SOL (e.g., 1-2 layers) is already sufficient to optimize them efficiently. More importantly, SOL can be trained by backward propagation. This feature enables convenient integration with other NN-based models for an end-to-end training, allowing efficient learning of optimizer for diverse range of tasks.

3.4 Unrolled Optimization of SOL

With the SOL model, we need an efficient way to train it on different optimization tasks. As introduced in Sec. 3.2, SOL can be trained by minimizing the outer-objective $L(\omega)$. During the backward propagation process, the derivatives of L with respect to the SOL parameters ω need to be computed. Due to the oscillations in NN training, direct optimization of the SOL using loss from each iteration would be unstable. On the other hand, computing the loss for the entire NN training process is expensive. Hence unrolled optimization is introduced to balance the training performance and computational cost.

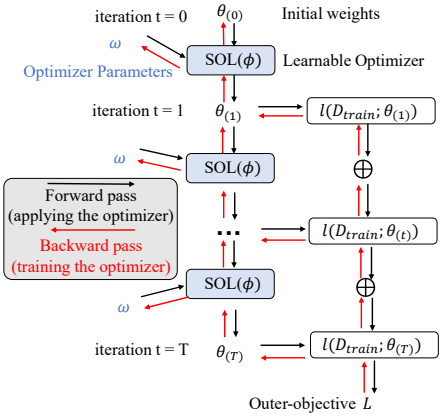

Fig. 4: SOL's unrolled optimization process.

Unrolled optimization draws inspiration from LSTM training, where the network is unrolled into a feedforward structure along the time dimension. Similarly, during SOL training, we iteratively unroll the process into a comprehensive computational graph. This graph can then be optimized by backpropagating the accumulated loss across multiple iterations.

The process is illustrated in Fig. 4. In the forward propagation, the SOL optimizer ϕ is iteratively applied to optimize the weights θ of a target optimizee,

which can form an unrolled computational graph. In the backward propagation, the outer-objective L backward propagates through this unrolled computational graph to compute gradients for the optimizer parameters ω. As demonstrated in existing works, it is computationally costly to unroll the whole inner optimization process to just obtain a single outer gradient. Compared to entire backward propagation, truncated backward propagation is a more desirable solution. It partitions the unrolled optimization process into separate segments, and multiple outer gradients can be computed over shorter segments. Instead of computing the full gradient from iteration $t = 0$ to T, we compute gradients in separate windows from $t = a$ to $a + \tau$. The gradients from these segments can be used to update ω without unrolling all T iterations, dramatically decreasing the computation needed for each update to ω. The number of inner-steps is called the unrolling length. It is challenging to set the length properly. A large number of steps per truncation can result in exploding gradients making outer-training difficult, while a small number of steps may produce biased gradients, consequently poor performance. The optimal unrolling length needs to be determined according to the specific optimization task in hand.

3.5 Discussions and Relationship to Prior Works

SOL aims to automatically design symbolic optimizers for diverse tasks using gradient descent directly. Compared to other automated symbolic optimizer design methods, our approach does not need to evaluate a large group of candidate optimizers that are generated through RL or EC search, hence significantly reduces the evaluation overhead occurred in the training process.

The training methods of SOL, such as bi-level optimization, are derived from L2O. Like the parametric model in L2O, SOL is also suitable to be directly integrated into the training process of NNs for learning. However, our method diverges from existing L2O methods both in motivation and the final product. Most of the L2O variants employed LSTM-based NN models. They use NN models to fit update rules and, therefore, do not include a symbolic search space. These NN models, known as parametric optimizers, often contain numerous parameters, which compromises their generalizability across diverse tasks, particularly large scale tasks. In addition, SOL is proposed with an aim of designing interpretable optimizers that can in the expression of mathematical equations, commonly known as symbolic optimizers. Hence, similar to previous automated symbolic optimizer search methods [2, 40, 48], we also incorporate a symbolic search space. The features constructed based on the current state of the optimizee are used as the inputs of SOL. The activation layer of SOL is replaced with unary and binary operators, which are necessary components for the construction of symbolic optimizers. These designs enable and facilitate SOL to discover suitable symbolic optimizers more successfully for different tasks, without relying on numerous parameters. That can be demonstrated through the following experiments over five groups of tasks, ranging from image classification to adversarial attack, to node classification, and to BERT fine-tuning.

4 Experiment Settings

Following conventions [2, 40], we first perform experiments on the standard automated optimizer design benchmarks, and demonstrate that SOL can discover optimizers that outperform both manually designed and automatically designed counterparts. Secondly, SOL is also evaluated on other popular ML tasks, including adversarial attack, GNN training and NLP model tuning. These are to further demonstrate that SOL can automatically design suitable optimizers for these tasks as well. Our experiments adopt the following operators, as suggested in [2, 48]:

- Input operators: g, g^2, g^3, ag, m, v, $sign(g)$, $sign(m)$, ad, rs, ld, cd, 1, 2
- Unary operators: $identical$, $(\cdot)^2$, exp, $sign$, $-(\cdot)$
- Binary operators: $+$, $-$, $\times$

These symbolic operators cover the gradients (g, g^2, and g^3), first-order momentum (m), second-order momentum (v), linear decay (ld), and cosine decay (cd). They are commonly used in current optimizers. The numbers, 1 and 2, are the constants that can be used in the optimizer. Division could cause numerical problems in a densely connected NN (e.g., inf or NaN), hence it is excluded from the operator set. Adam, denoted as ad, is included as an input operator, as the absence of $\div$ would prevent this type of operator from being explicitly represented. The two decay operators, ld and cd, are in below equations:

- **linear decay**: $1 - t/T$,
- **cosine decay**: $0.5 \times (1 + cos(2\pi nt/T))$,

where t and T are the current step and maximum step respectively, and n is set to 0.5 for cosine decay. The operator ag denotes preprocessed g. It is to make the computation of g more stable, using the following formula:

$$ag = log(|g| + \epsilon)/p \times in - (1 - in),$$

where $in = |g| > e^{-p}$, and p, ϵ are set to 10 and 10^{-6} respectively. We use the default PyTorch setting and hyper-parameters for all the input operators. The investigation on how the choice of operators impacts the optimizers are presented in **Appendix A1**.

The hyperparameter setting in the optimizer learning process (e.g., epochs and unrolled length) follows existing L2O studies. Other task-related hyperparameters, e.g., learning rate and batch size, can be adjusted but are not impactful, as presented in **Appendix A2**. We conduct the experiments with different hyperparameters setting to analysis their impact, as presented in **Appendix A3**.

5 Experiments and Results

To validate the generalizability of SOL, five different tasks are involved, including hand-written digit classification, image classification, adversarial attack, node classification, and BERT fine-tuning. They are presented from Sec. 5.1 to Sec. 5.5. The symbolic forms of the generated optimizers are analyzed and visualized in **Appendix A4**.

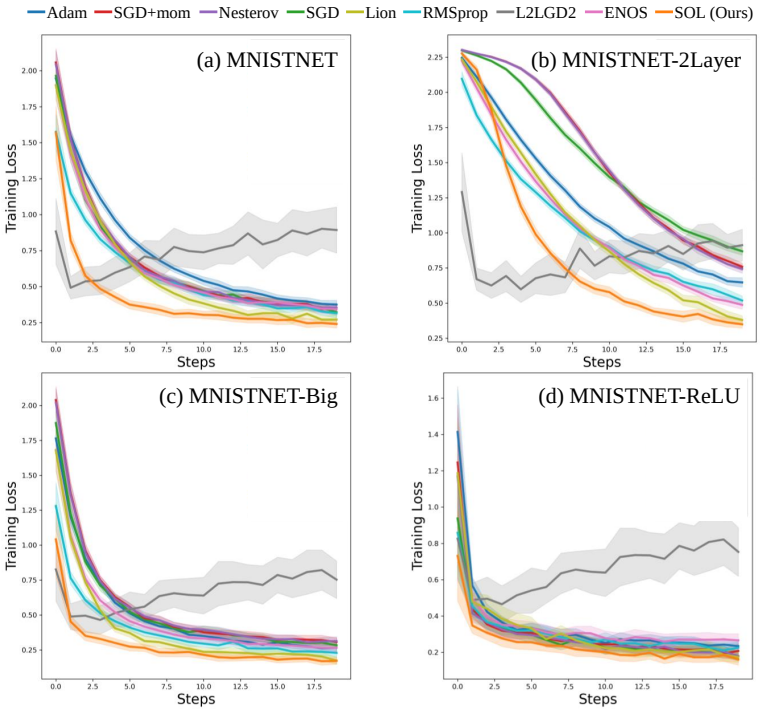

Fig. 5: Training loss trajectory on hand-written digit classification with MNISTNET.

5.1 Task 1: Hand-written Digit Classification with MNISTNET

We first evaluate SOL on a standard automated optimizer design benchmark MNISTNET [1, 40]. The optimizer learning process is conducted on the standard MNISTNET. After that, the learned symbolic optimizer is utilized to train the network from scratch and compared with other optimizers. The competitors are in two categories: five manually designed optimizers (Adam, SGD, Momentum, Nesterov, and RMSprop) and three automatically designed optimizers (L2LGD2 [1], ENOS [40], Lion [7]). In order to verify the generalizability of SOL, we transfer the learned optimizer to optimize three variants of MNISTNET with a number of hidden layers (MNISTNET-2Layer), different dimensions (MNISTNET-Big), and activations (MNISTNET-ReLU).

As shown in Fig. 5, the optimizer discovered by SOL achieves the lowest training loss on the MNISTNET and its variants. This result demonstrates the effectiveness and generalizability of the proposed method. More specifically, in terms of the convergence speed, SOL achieved the first place on MNISTNET-ReLU and marginally behind L2LGD2 in the other three experiments. However, the fast convergence in the early training iterations of L2LGD2 is essentially a result of overfitting. As demonstrated in the figure, L2LGD2 gradually shows an oscillating upward trend after convergence. The training loss of SOL at first

Table 2: Performance of optimizers designed by different methods on CIFAR-10.

Optimizer	Acc.(%)	Search Method
Adam	67.15%±0.47	manual
SGD	43.22%±0.59	manual
SGD+Momentum	66.34%±0.05	manual
Nesterov	66.79%±0.47	manual
RMSprop	66.47%±0.47	manual
PowSign-ld	35.18%±0.19	NOS-RL [2]
PowSign-cd	35.56%±0.41	NOS-RL [2]
AddSign-ld	34.04%±0.42	NOS-RL [2]
AddSign-cd	34.33%±0.50	NOS-RL [2]
Lion	67.02%±0.11	EC-based [7]
conv2	67.65%±0.10	ENOS [40]
Ours	**68.75%±0.35**	Meta Training

Table 3: Test accuracy on CIFAR-10 with ResNet-18.

Optimizer	Acc.(%)	Search Time
SGD	89.08%	-
SGD+mom	91.19%	-
Adam	91.50%	-
SymbolicL2O	93.45%	5.65h
Ours	93.16%	1.60h

Table 4: GPU memory usage during training.

Model	#Para.	Ours	L2LGD2
ResNet18	11.2M	2,125MB	6,307MB
ResNet34	21.8M	2,531MB	11,761MB
ResNet50	25.6M	3,169MB	24,541MB

converges rapidly and then gradually stabilizes. This comparison illustrates the superiority of the discovered optimizer by SOL against parametric optimizers. In terms of final training loss, SOL achieved the best performance among all eight competitors in all four experiments. Its lead is significant on MNISTNET and MNISTNET-2Layer.

5.2 Task 2: Image Classification with ConvNet and ResNet

We further validate SOL on another automated optimizer design benchmark, proposed in NOS-RL [2]. This benchmark is to discover a suitable optimizer that can efficiently train a CNN, namely ConvNet, to classify the CIFAR-10 dataset. More specifically, the ConvNet is with 2 convolution layers, each of which contains a 32-filter 3×3 convolution with ReLU activation and batch normalization. Apart from the optimizers generated by NOS-RL [2], we also include classic hand-designed optimizers, as well as more recent automatically generated optimizers [7, 40] in the comparison.

The results are summarized in Table 2. Compared to manually designed optimizers (the first block with 5 methods), our method achieved better performance than the most commonly used Adam optimizer and others. Compared to the automatically designed competitors (the second block with 6 methods), the test accuracy achieved by our method is also the highest. In addition, the efficiency of our method is also better than other optimizer search methods. For example, NOS-RL [2] sampled 15,000 optimizers. Each of them needs 10 minutes for performance evaluation, which requires about a day of search time, even with massively parallel evaluation. In contrast, our method can learn an optimizer for the task in a mere 3 hours. The above results illustrate SOL can significantly reduce the computation cost and improve the performance.

Although simple MNISTNET and ConvNet are common benchmarks in the field of automated optimizer search, their accuracy on image classification tasks is significantly lower than complex CNN (e.g., ResNet18). SymbolicL2O [48]

has taken the lead in exploring automatic search optimizers for commonly used CNNs, such as ResNet. Following the work of SymbolicL2O, we also evaluate the proposed method on ResNet18 and compare it with manually designed optimizers and SymbolicL2O. The results are shown in Table 3. The optimizer learned by SOL again outperformed all the competitors, except for an accuracy that is 0.29% behind SymbolicL2O. As for the search time, SOL is 3.5× faster than SymbolicL2O. It took only 1.60h, instead of 5.65h, in the search process. Search time indicates the algorithm efficiency, arguably more important for automatically designed methods in practice. More analysis about training time is presented in **Appendix A5**.

More importantly, SOL shows advantage in generalizability and scalability. As for generalizability, we transfer the optimizer searched on ResNet18 to a completely different task, the fine-tuning LLMs task. More specifically, we fine-tuned the LLaMA-7B model on the Alpaca dataset. The validation error was 0.93 using the AdamW optimizer and 0.91 using ours. Our optimizer actually has a faster convergence speed than Adamw as shown in Fig. 6. As for scalability, we studied the GPU memory usage. As indicated in Table 4, the GPU memory usage of SOL increases with the number of parameters, at a rate below linear growth. In comparison, L2LGD2, the well-known automatically designed parametric optimizers, shows an exponential increase. This means that the optimizer searched by SOL can easily be applied to large-scale tasks.

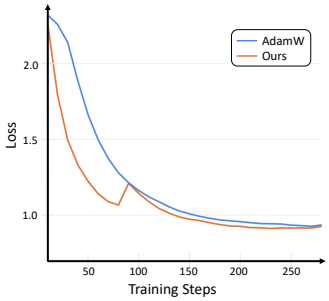

Fig. 6: Training loss trajectory on tuning LLaMA-7B.

5.3 Task 3: Adversarial Attack

We further evaluate SOL on adversarial attack as suggested in [40], to see how would SOL perform under constraints. Adversarial attacks are a typical constraint optimization problem that aims at finding norm-bounded perturbations in the input space that alter a model's predictions. Projected Gradient Descent (PGD) [25] is the optimizer commonly used in adversarial attacks. In commonly used l_∞-norm setting, PGD takes the form of : $x = Proj_{B_\epsilon^\infty(x_o)}(x + \gamma sign(\bigtriangledown_x L(x)))$, where $B_\epsilon \infty(x_o)$ represents a ϵ ball around the original image x_o w.r.t. l_∞-norm. The aim is to automatically design the update rule inside the projection operator. We choose seven top defense models from RobustBench [8]. The learning process is conducted in Carmon2019 model [4]. After that, the learned optimizer is subsequently evaluated on the other six top defense methods for WideResNet [47] and ResNet [18]. Following the settings [8], ϵ is set to 8/255. Each optimizer runs once for 100 steps on every image from the test split.

The results are shown in Table 5. The optimizer discovered by SOL outperforms PGD on all the defense models. Moreover, we also compared the discovered optimzier with Adaptive PGD (APGD) [9] which is the best handcrafted

Table 5: Attack success rate of optimizers against defense models on CIFAR-10.

Defense Models	PGD	APGD	Ours
Carmon2019 [4](WRN-28-10)	38.67%	39.06%	**39.45%**
Gowal2020 [16](WRN-34-20)	39.98%	40.02%	**40.61%**
Gowal2020 [16](WRN-28-10)	35.93%	35.93%	**36.33%**
Sehwag2020 [33](WRN-28-10)	40.63%	**41.02%**	**41.02%**
Wu2020 [44](WRN-28-10)	41.41%	41.41%	**41.80%**
Engstrom201 [13](RN-50)	49.38%	49.60%	**49.90%**
Wong2020 [43](RN-18)	54.69%	**55.08%**	**55.08%**

Table 6: Classification accuracy of our optimizer against Adam.

Model	Dataset	Adam	Ours
GAT	Cora	82.48%±0.52	**83.10%±0.57**
	Citeseer	65.33%±0.30	**66.07%±0.24**
	PubMed	75.40%±0.81	**76.58%±0.26**
JK-Net	Cora	75.80%±0.10	**76.70%±0.07**
	Citeseer	59.93%±0.23	**61.02%±0.25**
	PubMed	73.87%±0.30	**74.20%±0.71**

and tuned optimizer for adversarial attacks. APGD combines a well-tuned momentum update rule with a conditional learning rate decay on a handcrafted schedule, which can be adapted for different tasks. Our results show that the optimizer from SOL can also rival APGD, clearly demonstrating the effectiveness of SOL for constraint optimization tasks. In addition, more experiments about fewer attack iterations and different attack principles are in **Appendix A6**.

5.4 Task 4: Node Classification on Graphs

To see whether SOL can generate optimizers to train GNNs to classify nodes on graphs, the Graph Attention Network (GAT) [38] and Jumping Knowledge Network (JK-Net) [46] are used. They are the most widely used models in graph learning. The comparison is performed on three graph benchmarks including Cora [26], Citeseer [15], PubMed [34]. The number of iterations is set to 50. For each iteration, GAT is trained for 250 epochs, and the unrolling length is 50.

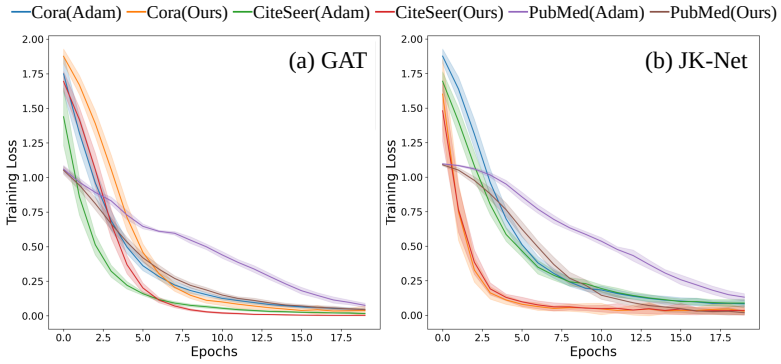

Fig. 7: Training loss trajectory with GAT and JK-Net.

The experiment results are shown in Table 6. On all the graph benchmarks, the proposed method can discover optimizers outperforming Adam, which is the standard optimizer for optimizing GNNs. In addition, it is worth noting that

GAT and JK-Net are two different GNN architectures, based on graph attention and graph convolution respectively. Nevertheless, the proposed method can still discover suitable optimizers for them (as shown in Fig. 7). This result further demonstrates SOL's generalizability and its potential to reduce the need for expert knowledge in designing different optimizers.

5.5 Task 5: BERT Fine-tuning on NLP tasks

To evaluate SOL's capability for large language models, we evaluated it on BERT [10]. In the experiment, a pre-trained BERT (base-uncased) is finetuned on the GLUE benchmark [39]. We follow the default configurations: the number of epochs is set to 3 for Cola [41], SST-2 [35], and RTE [3] dataset, and 5 epochs on MPRC [11] and WNLI [39] dataset. The batch size is set to 32.

The experiment results are listed in Table 7. We compare the optimizer learned by our method with AdamW [23], which is the default optimizer for BERT finetuning. It can be seen from Table 7 that the learned optimizers by our method outperformed AdamW on all five datasets. Such a result demonstrates the effectiveness of the proposed method of learning optimizers even for large language models.

Table 7: Performance of BERT finetuning on GLUE.

Dataset	AdamW	Ours
Cola	61.01±0.84	**63.14±0.56**
SST-2	90.87±0.96	**91.83±0.38**
RTE	63.87±1.63	**66.83±2.01**
MRPC	82.38±1.06	**84.62±1.02**
WNLI	54.07±3.29	**55.63±0.28**

6 Conclusion

The goal of this study is to develop a method that enables the automated design of high-performance symbolic optimizers directly on a diverse range of tasks. To achieve this, we propose a symbolic optimizer learning model, namely SOL, that takes an optimizee's states as input and produces the updates for the optimizee's parameters. As a learnable model, SOL can be integrated into the optimizee's training process and directly optimized by gradient descent. More importantly, the trained SOL can then be transformed into a mathematical expression, i.e., a symbolic optimizer, which is scalable and generalizable, capable of optimizing new tasks. The proposed method is evaluated on a diverse set of five different tasks, including image classification, adversarial attack, graph node classification, and large language model finetuning. The experiment results clearly demonstrate the effectiveness of the proposed method, as it can automatically design symbolic optimizers, outperforming both manually-designed and other auto-designed counterparts, on these diverse tasks. Hereby we conclude that our proposed symbolic optimizer learner is effective, yet generalizable and scalable, offering a highly competitive option for network optimization. Our future work will focus on simplifying the learned symbolic optimizers to enhance their interpretability and facilitate future improvement.

Acknowledgments

This work was supported by National Natural Science Foundation of China under Grant 62276175 and Innovative Research Group Program of Natural Science Foundation of Sichuan Province under Grant 2024NSFTD0035.

References

1. Andrychowicz, M., Denil, M., Gomez, S., Hoffman, M.W., Pfau, D., Schaul, T., Shillingford, B., de Freitas, N.: Learning to learn by gradient descent by gradient descent. Advances in Neural Information Processing Systems **29** (2016)
2. Bello, I., Zoph, B., Vasudevan, V., Le, Q.V.: Neural optimizer search with reinforcement learning. In: International Conference on Machine Learning. pp. 459–468 (2017)
3. Bentivogli, L., Clark, P., Dagan, I., Giampiccolo, D.: The fifth pascal recognizing textual entailment challenge. TAC **7**, 8 (2009)
4. Carmon, Y., Raghunathan, A., Schmidt, L., Duchi, J.C., Liang, P.S.: Unlabeled data improves adversarial robustness. Advances in Neural Information Processing Systems **32** (2019)
5. Cauligi, A., Culbertson, P., Stellato, B., Bertsimas, D., Schwager, M., Pavone, M.: Learning mixed-integer convex optimization strategies for robot planning and control. In: 2020 59th IEEE Conference on Decision and Control (CDC). pp. 1698–1705 (2020)
6. Chen, T., Chen, X., Chen, W., Wang, Z., Heaton, H., Liu, J., Yin, W.: Learning to optimize: A primer and a benchmark. The Journal of Machine Learning Research **23**(1), 8562–8620 (2022)
7. Chen, X., Liang, C., Huang, D., Real, E., Wang, K., Pham, H., Dong, X., Luong, T., Hsieh, C.J., Lu, Y., et al.: Symbolic discovery of optimization algorithms. Advances in Neural Information Processing Systems **36** (2024)
8. Croce, F., Andriushchenko, M., Sehwag, V., Debenedetti, E., Flammarion, N., Chiang, M., Mittal, P., Hein, M.: Robustbench: a standardized adversarial robustness benchmark. In: Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (2021), https://openreview.net/forum?id=SSKZPJCt7B
9. Croce, F., Hein, M.: Reliable evaluation of adversarial robustness with an ensemble of diverse parameter-free attacks. In: International Conference on Machine Learning. pp. 2206–2216 (2020)
10. Devlin, J., Chang, M.W., Lee, K., Toutanova, K.: Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805 (2018)
11. Dolan, B., Brockett, C.: Automatically constructing a corpus of sentential paraphrases. In: Third International Workshop on Paraphrasing (IWP2005) (2005)
12. Duchi, J., Hazan, E., Singer, Y.: Adaptive subgradient methods for online learning and stochastic optimization. Journal of Machine Learning Research **12**(7) (2011)
13. Engstrom, L., Ilyas, A., Salman, H., Santurkar, S., Tsipras, D.: Robustness (python library), 2019. URL https://github. com/MadryLab/robustness **4**(4), 4–3 (2019)
14. Fan, F.L., Xiong, J., Li, M., Wang, G.: On interpretability of artificial neural networks: A survey. IEEE Transactions on Radiation and Plasma Medical Sciences **5**(6), 741–760 (2021). https://doi.org/10.1109/TRPMS.2021.3066428

15. Giles, C.L., Bollacker, K.D., Lawrence, S.: Citeseer: An automatic citation indexing system. In: Proceedings of the third ACM conference on Digital libraries. pp. 89–98 (1998)
16. Gowal, S., Qin, C., Uesato, J., Mann, T., Kohli, P.: Uncovering the limits of adversarial training against norm-bounded adversarial examples. arXiv preprint arXiv:2010.03593 (2020)
17. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: Proceedings of the IEEE conference on computer vision and pattern recognition. pp. 770–778 (2016)
18. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. pp. 770–778 (2016)
19. Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural Computation **9**(8), 1735–1780 (1997)
20. Kingma, D.P., Ba, J.: Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980 (2014)
21. Krizhevsky, A., Hinton, G.: Learning multiple layers of features from tiny images. Handbook of Systemic Autoimmune Diseases **1**(4) (2009)
22. Lipton, Z.C., Berkowitz, J., Elkan, C.: A critical review of recurrent neural networks for sequence learning. arXiv preprint arXiv:1506.00019 (2015)
23. Loshchilov, I., Hutter, F.: Decoupled weight decay regularization. arXiv preprint arXiv:1711.05101 (2017)
24. Lv, K., Jiang, S., Li, J.: Learning gradient descent: Better generalization and longer horizons. In: International Conference on Machine Learning. pp. 2247–2255 (2017)
25. Madry, A., Makelov, A., Schmidt, L., Tsipras, D., Vladu, A.: Towards deep learning models resistant to adversarial attacks. arXiv preprint arXiv:1706.06083 (2017)
26. McCallum, A.K., Nigam, K., Rennie, J., Seymore, K.: Automating the construction of internet portals with machine learning. Information Retrieval **3**, 127–163 (2000)
27. Metz, L., Harrison, J., Freeman, C.D., Merchant, A., Beyer, L., Bradbury, J., Agrawal, N., Poole, B., Mordatch, I., Roberts, A., et al.: Velo: Training versatile learned optimizers by scaling up. arXiv preprint arXiv:2211.09760 (2022)
28. Metz, L., Maheswaranathan, N., Nixon, J., Freeman, D., Sohl-Dickstein, J.: Understanding and correcting pathologies in the training of learned optimizers. In: International Conference on Machine Learning. pp. 4556–4565 (2019)
29. Orchard, J., Wang, L.: The evolution of a generalized neural learning rule. In: 2016 International Joint Conference on Neural Networks (IJCNN). pp. 4688–4694 (2016)
30. Real, E., Liang, C., So, D., Le, Q.: Automl-zero: Evolving machine learning algorithms from scratch. In: International Conference on Machine Learning. pp. 8007–8019 (2020)
31. Robbins, H., Monro, S.: A stochastic approximation method. The Annals of Mathematical Statistics pp. 400–407 (1951)
32. Sahoo, S., Lampert, C., Martius, G.: Learning equations for extrapolation and control. In: International Conference on Machine Learning. pp. 4442–4450 (2018)
33. Sehwag, V., Wang, S., Mittal, P., Jana, S.: Hydra: Pruning adversarially robust neural networks. Advances in Neural Information Processing Systems **33**, 19655–19666 (2020)
34. Sen, P., Namata, G., Bilgic, M., Getoor, L., Galligher, B., Eliassi-Rad, T.: Collective classification in network data. AI Magazine **29**(3), 93 (2008)

35. Socher, R., Perelygin, A., Wu, J., Chuang, J., Manning, C.D., Ng, A.Y., Potts, C.: Recursive deep models for semantic compositionality over a sentiment treebank. In: Proceedings of the 2013 conference on empirical methods in natural language processing. pp. 1631–1642 (2013)
36. Tieleman, T., Hinton, G.: Lecture 6.5-rmsprop, coursera: Neural networks for machine learning. University of Toronto, Technical Report **6** (2012)
37. Tseng, P.: An incremental gradient (-projection) method with momentum term and adaptive stepsize rule. SIAM Journal on Optimization **8**(2), 506–531 (1998)
38. Veličković, P., Cucurull, G., Casanova, A., Romero, A., Lio, P., Bengio, Y.: Graph attention networks. arXiv preprint arXiv:1710.10903 (2017)
39. Wang, A., Singh, A., Michael, J., Hill, F., Levy, O., Bowman, S.R.: Glue: A multi-task benchmark and analysis platform for natural language understanding. arXiv preprint arXiv:1804.07461 (2018)
40. Wang, R., Xiong, Y., Cheng, M., Hsieh, C.J.: Efficient non-parametric optimizer search for diverse tasks. Advances in Neural Information Processing Systems (2022)
41. Warstadt, A., Singh, A., Bowman, S.R.: Neural network acceptability judgments. Transactions of the Association for Computational Linguistics **7**, 625–641 (2019)
42. Wichrowska, O., Maheswaranathan, N., Hoffman, M.W., Colmenarejo, S.G., Denil, M., Freitas, N., Sohl-Dickstein, J.: Learned optimizers that scale and generalize. In: International Conference on Machine Learning. pp. 3751–3760 (2017)
43. Wong, E., Rice, L., Kolter, J.Z.: Fast is better than free: Revisiting adversarial training. arXiv preprint arXiv:2001.03994 (2020)
44. Wu, D., Xia, S.T., Wang, Y.: Adversarial weight perturbation helps robust generalization. Advances in Neural Information Processing Systems **33**, 2958–2969 (2020)
45. Xiong, Y., Hsieh, C.J.: Improved adversarial training via learned optimizer. In: Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part VIII 16. pp. 85–100 (2020)
46. Xu, K., Li, C., Tian, Y., Sonobe, T., Kawarabayashi, K.i., Jegelka, S.: Representation learning on graphs with jumping knowledge networks. In: Proceedings of the 35th International Conference on Machine Learning. vol. 80, pp. 5453–5462 (2018)
47. Zagoruyko, S., Komodakis, N.: Wide residual networks. arXiv preprint arXiv:1605.07146 (2016)
48. Zheng, W., Chen, T., Hu, T.K., Wang, Z.: Symbolic learning to optimize: Towards interpretability and scalability. In: International Conference on Learning Representations (2022)