Guided Evolution with Binary Predictors for ML Program Search

John D. Co-Reyes[*], **Yingjie Miao**[*], **George Tucker, Aleksandra Faust, Esteban Real**
Google DeepMind

Abstract

Primitive-based evolutionary AutoML discovers novel state-of-the-art ML components by searching over programs built from low-level building blocks. While very expressive, these spaces have sparsely distributed good performing candidates. This poses great challenges in efficient search. Performance predictors have proven successful in speeding up search in smaller and denser Neural Architecture Search (NAS) spaces, but they have not yet been tried on these larger primitive-based search spaces. Through a unified graph representation to encode a wide variety of ML components, we train a binary classifier online to predict which of two given candidates is better. We then present an adaptive mutation method that leverages the learned binary predictor and show how it improves local search. We empirically demonstrate our method speeds up end-to-end evolution across a set of diverse problems including a 3.7x speedup on the symbolic search for ML optimizers and a 4x speedup for RL loss functions.

1 Introduction

While neural architecture search (NAS [1]) is a major area in AutoML [2], there is a growing body of work that searches for general ML program components beyond architectures, such as the entire learning program, RL loss functions, and ML optimizers [3, 4, 5]. The underlying search spaces commonly use fine-grained primitive operators such as tensor arithmetic and have few human imposed priors, resulting in long programs or computation graphs with many nodes. While expressive enough to allow discoveries of novel ML components that achieve state-of-the-art results compared to human designed ones, such search spaces have near infinite size. The extremely sparse distribution of good candidates in the space (since small changes usually lead to dramatic performance degradation) poses great challenges for efficient search of performant programs. Unlike NAS where several successful search paradigms exist (Bayesian optimization [6, 7], reinforcement learning [8, 9], differentiable search [10]), regularized evolution [11] remains a dominant search method on these primitive search spaces due to its simplicity and effectiveness. The main technique for speeding up search on these spaces so far is functional equivalent caching (FEC [12]), which skips repeated evaluation of duplicated candidates. While effective, FEC does not exploit any learned structure in the evaluated candidates. Can we devise learning-based methods that capture global knowledge of all programs seen so far to improve search efficiency?

Performance predictors [13] have shown successes in speeding up search in many NAS search spaces but have not yet been tried on primitive-based search spaces. Prior work uses regression models, trained to predict architecture performance, to rank top candidates before computationally expensive evaluation. An alternative is to train binary relation predictors [14, 15], which predict which candidate

[*]Equal contribution. Order decided randomly.
Correspondence to: `{jcoreyes, yingjiemiao}@google.com`

from a pair is better. It is not obvious if these methods can be successfully applied to these spaces out-of-the-box. Previous work usually trains performance predictors on NAS search spaces using only a few hundred (or fewer) randomly sampled candidates [14]. Prediction on unseen candidates relies on strong generalization performance and we argue that this is relatively easy for many NAS search spaces, but much harder for these primitive search spaces. [16] showed the NAS-Bench-101 fitness landscape is very smooth with most candidates having similar fitness and the global optimum is at most 6 mutation steps away from more than 30% of the search space. [17] further showed that random search is a strong baseline on NAS-Bench-101 which suggests that a predictor trained on a fixed offline dataset collected from random sampling will generalize well. In contrast, in [3] and [5], random search fails completely on these larger search spaces which have sparse reward distribution. So random search alone will struggle to capture enough representative data for generalization. We argue online learning will be necessary to quickly adapt predictors to newly discovered candidates and make new predictions on the current distribution of data. Many predictor-based methods have not been tested in this setting, and are usually trained on fixed offline datasets. The online learning regime also creates the challenge of balancing between exploration and exploitation. Exploration is important for escaping local optima and providing diverse training data for the predictors while exploiting an overfitted predictor can degrade long term evolution performance.

In this paper, we propose training a binary relation predictor online to guide mutations and speed up evolution. This predictor is trained continuously with pairs of candidates discovered by evolution to predict which candidate is better. We introduce a novel mutation algorithm for combining these predictors with evolution to continually score mutations until we find a candidate with a higher predicted fitness than its parent, bypassing wasteful computation on (likely) lower fitness candidates. Our method provides large benefits to evolution in terms of converging more quickly and to a higher fitness over a range of problems including a 3.7x speedup on ML optimizers and 4x speedup on RL loss functions. We show that obtaining generalization with this binary predictor is much easier than with a fitness regression model and that using state-of-the-art graph neural networks (GNNs) gives better results. Our unified representation and training architecture can be applied to generic ML components search.

2 Related Work

NAS with performance predictors. Many performance predictor methods [13] have been proposed to speed up NAS. Once trained, these models are used for selecting the most promising architecture candidates for full training, reducing the resources (compute and walltime) wasted on unpromising ones. One popular category of these methods is to train a regression model [18, 19, 20, 21, 22] to predict performance of an architecture solely based on its encoding. Some regression models, such as ReNAS [23], are trained directly with a ranking loss instead of MSE. An alternative is to train pairwise binary relation models [14, 15, 24]. Such model takes a pair of candidates as input and predicts which candidate is better. This is motivated by the observation that predicting relative performance of a pair of candidates is sufficient for ranking. BRP-NAS [14] shows that binary predictor models are more effective than regression models for candidate selection. BRP-NAS alternates between two phases: i) use the predictor to rank candidates and ii) select the top candidates for full training and the results are used to improve the predictor. In [14], these two phases alternate very few times to collect no more than a few hundred fully trained candidates. Fewer works explore the integration of binary predictor with evolution. To our best knowledge, [15] is the closest to our work, which uses the predictor to rank a list of offspring candidates by doing pairwise comparisons between candidates. In contrast, our work uses the binary predictor to compare the child candidate with the parent (the tournament selection winner) which explicitly encourages hill climbing [25]. In addition, problems studied in this work are of much larger scale compared to those in [15]. We note that hill climbing strategy has been tried for NAS in [26], however this does not make use of performance predictors and evolutionary algorithms.

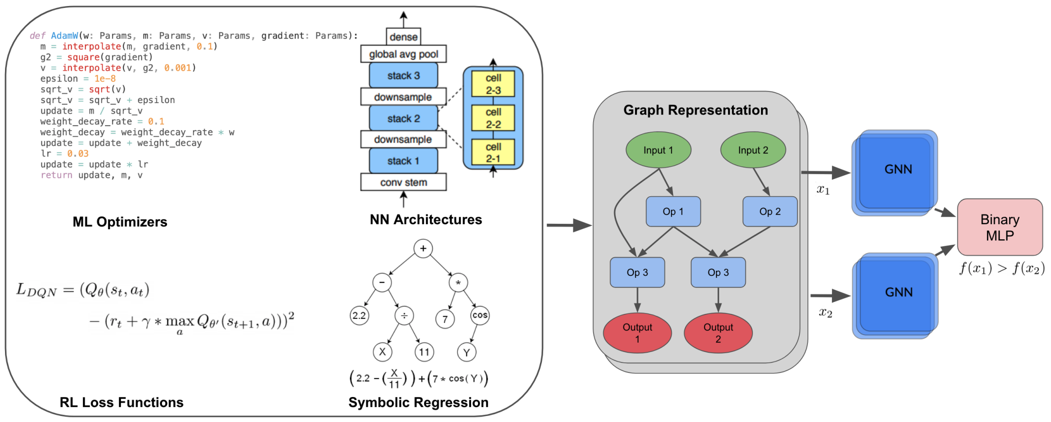
General ML program component search. Recent work [3, 4, 5] searches for ML program components beyond neural network architectures. These search spaces use primitive operators as building blocks and few human priors are imposed on how they should be combined. This is in stark contrast with many NAS search spaces where non-trivial human priors are included in the design. For example, it is a common choice to bundle ReLU, Convolution, and Batch Normalizaton as one atomic operator (in a fixed order), as in NAS-Bench-101 and DARTS [10]. Due to the use of primitive operators and few constraints, these general ML program search spaces are much larger and have sparse rewards. This makes them more challenging than typical NAS search spaces. Previously, hashing based techniques [12] are the main method for speeding up search on these large primitive search spaces, while predictive models have not been tried. We show that predictive models can provide extra speedup on top of FEC techniques in the ML optimizer search space.

Learning and program synthesis. There are many works that learn a generative model or policy over discrete objects. Work such as [8, 27, 28, 29, 30] use reinforcement learning to optimize this model. Other works combine a generative model with evolution such as using an LLM as a mutator [31] or training a generator to seed the population [32]. Training a generative model in this large combinatorial space is generally more difficult than training a binary discriminator and requires more complicated algorithms such as RL whereas our method is a simple modification to evolution.

3 Method

In this section we describe the search representation, how the binary predictor can work over a variety of ML components, and how to combine the model with evolution.

3.1 Search Representation



```
def AdamW(w: Params, m: Params, v: Params, gradient: Params):
  m = interpolate(m, gradient, 0.1)
  g2 = square(gradient)
  v = interpolate(v, g2, 0.001)
  epsilon = 1e-8
  sqrt_v = sqrt(v)
  sqrt_v = sqrt_v + epsilon
  update = m / sqrt_v
  weight_decay_rate = 0.1
  weight_decay = weight_decay_rate * w
  update = update + weight_decay
  lr = 0.03
  update = update * lr
  return update, m, v
```

$$L_{DQN} = (Q_\theta(s_t, a_t) - (r_t + \gamma * \max_a Q_{\theta'}(s_{t+1}, a)))^2$$

Figure 1: We encode a variety of ML components (learning optimizers input as Python code, NN architectures, RL loss functions, and symbolic equations) into the same computation graph representation and learn a GNN-based binary predictor over pairs of individuals to predict which graph has better performance.

We build a general framework to encode a wide variety of ML components with the same GNN architecture. We first convert each task (as described below) into a directed acyclic graph (DAG) where a DAG consists of input nodes, intermediate operation nodes, and output nodes. A directed edge from node a to node b would indicate that a is an input for operation b. The number of possible identities for each node will depend on the possible operations for that task.

Neural Network Architectures: A neural network architecture cell is readily represented as a DAG where nodes are operations such as 3x3 convolution and edges define data flow between ops [16].

Symbolic Regression: The objective is to recover a range of target equations such as $log(1+x)+x^2$. While equations are normally represented as trees, the DAG formulation allows variables to be reused since children can have multiple parents. Node ops are basic math operators [33].

ML Optimizer Algorithms: The optimizer for updating a neural network such as AdamW [34] is parsed as Python code [5]. Each line is an assignment that becomes a node in the DAG similar to [35]. Node ops can be math operators or high level function calls.

RL Loss Functions: The loss function to train an RL agent such as DQN [36] is converted into a DAG as in [4]. Nodes are operations such as apply a Q network on an environment observation and the output node is the final loss to minimize.

3.2 Architecture + Training

We describe the general architecture and how to train the binary predictor. Once a problem can be converted into a DAG, we compute embeddings for each node and edge based on its identity. Between domains, node operations will differ so we use different embedding layers between tasks where the number of possible embeddings depends on the maximum number of operations. Edge embeddings will similarly depend on the max number of possible connections for that task. Given these embeddings, we can then apply a GNN or graph Transformer to compute the DAG encoding $g(x)$ of any graph x.

Given a dataset of individuals, we want to train a predictor f, that can take in any pair of individuals (x_1, x_2) and predict whether the fitness of x_1 is greater than the fitness of x_2. Our predictor is a 2 layer binary MLP which takes in the concatenated graph encodings $concat(g(x_1), g(x_2))$ and outputs a logistic score. The predictor and GNN are trained end-to-end to minimize the binary cross entropy loss over randomly sampled pairs from the current dataset. During evolution, this predictor is trained online and so will improve as the dataset size increases with more individuals. The dataset is a fixed size queue and training happens at set intervals during evolution. More details on the architecture and training algorithm are in Appendix 6.1.

3.3 Combining Binary Models with Evolution

A binary predictor $f(\cdot, \cdot)$ can be combined with a vanilla mutator (e.g. modifying a few nodes and edges in a DAG) to form what we call a *mutation strategy*. There are many possible mutation strategies and we propose a particular design which achieves good performance.

We briefly recap how regularized evolution works and introduce a few terms. Regularized Evolution (RegEvo) has two phases: In the first phase, we initialize a *population* queue of size P with randomly generated candidates, each candidates is assigned a *fitness* score. In the second phase, we repeat the following loop until we have swept over a desired number of candidates: i) Randomly sample T candidates from P and select the one with the highest fitness. This step is known as *tournament selection* [37] of size T; ii) A *mutator* randomly mutates the selected candidate p to a child c, and c's fitness score is computed; iii) Add c to the end of the queue and remove the oldest item from the head of the queue.

Algorithm 1 Binary Predictor-based Adaptive Mutation with Re-Tournament (PAM-RT)

```
Input: Population buffer $P$, trained predictor $f$, max attempts K.
1: $accept \leftarrow False$
2: $attempts \leftarrow 0$
3: while $accept == False$ and $attempts < K$ do
4:     $parent \leftarrow tournament\_selection(P)$
5:     $child \leftarrow mutate(parent)$
6:     $accept \leftarrow f(child, parent) > 0.5$
7:     $attempts \leftarrow attempts + 1$
8: end while
Output: $child$
```

In prior work combining predictors with evolution, it is common to generate a list of candidates and use the predictor to rank these candidates. For example, in [15] a parent is mutated to a list of child candidates, and the model is used to score each candidate (against other candidates) and take the argmax as the final candidate. More precisely, if we have a list of candidates c_i, then score$(c_i) = \sum_{j \neq i} f(c_i, c_j)$ and f takes discrete values in $\{-1, 1\}$. We call this approach max pairwise (**Max-Pairwise**).

In this work, we explore a different family of strategies based on the heuristic of hill climbing [25] which performs iterative local search to find incrementally better solutions. We use the binary model to compare the child against the parent (instead of against each other), and this explicitly encourages selecting a child that is likely better than the parent. In its most basic form, we run tournament selection once to obtain a parent p, mutate it to obtain a child candidate c, and check if the child is likely better than the parent by evaluating $f(c, p)$; if not, we retry the mutation from p and otherwise accept c. We refer this method as predictor-based adaptive mutation (**PAM**). A variant is to retry the tournament if the mutated child is rejected by the model which we refer to as predictor-based adaptive mutation with re-tournament (**PAM-RT**). We summarize PAM-RT in Algorithm 1. PAM can be obtained by a one line change by moving the tournament-selection before the while-loop.

Hill climbing properties of PAM and PAM-RT. We provide some analysis on the local search properties of our method to motivate their design. Given a standard evolution mutator m, the *natural* *hill climbing rate* q at a point p is the probability that the random child $m(p)$ is better than p. We define the *modified hill climbing rate* as the same probability but for a child produced from our proposed method. In Section 6.2 we show that as long as the model is better than random, the modified rate is always higher than the natural rate, and increasing model accuracy leads to larger gains, as expected.

PAM-RT has an extra re-tournament mechanism compared to PAM which gives PAM-RT a bias towards selecting parents that are more likely leading to a hill climbing child – *if the model is better than random*. At each iteration in PAM-RT, the probability of accepting a child is $p_{accept} = (2a-1)q + (1-a)$ (using Eq. 1 in Section 6.2), which increases with q if $a > 0.5$. Therefore PAM-RT will accept children of parents with larger q more quickly on average. We hypothesis that PAM-RT explores more due to such bias.

4 Experiments

In our experiments we show that PAM-RT can provide a meaningful speedup to evolution. We perform ablations showing the importance of several design choices and provide analysis on how to combine our predictor with evolution.

4.1 Experimental Setup

We describe the search spaces, their fitness definitions, and the random generators and mutators we use in regularized evolution.

NAS-Bench-101 [16] is a NAS benchmark for image classification on CIFAR-10 [38]. The search space consists of directed acyclic graphs (DAGs) with up to 7 vertexes and up to 9 edges. Two special vertexes are `IN` and `OUT`, representing the input and output tensors of the architecture. Remaining vertexes can choose from one of three operations: 3x3 convolution, 1x1 convolution, and 3x3 max-pool. This search space contains 510M distinct architectures (of which 423K are unique after deduping by graph isomorphisms). Every candidate in the search space has been pre-evaluated with metrics including validation accuracy and test accuracy. Evaluation of a single candidate is a fast in-memory table lookup which enables noisy oracle experiments in Section 4.2.1. In this work, we use the validation accuracy (after 108 epochs of training) as the fitness. We use the same random generator and mutator as published in [16]. Although NAS-Bench-101 does not use a primitive search space, we include it here because it is a well-accepted benchmark supporting extensive ablation studies. It is also a test to see if our proposed method works on more traditional NAS spaces.

Nguyen [33] is a benchmark for symbolic regression tasks. The goal is to recover ground truth single-variable or two-variable functions. Candidate functions are constructed from $\{+, -, *, /, \sin, \cos, \exp, \log, x\}$ (and additionally with $\{y\}$ for two-variable functions). We choose the Ngyuen benchmark because it shares features with primitive-based AutoML search spaces, such as having sparsely distributed high performing candidates. We choose a few Nguyen tasks (Nguyen-2, Nguyen-3, Nguyen-5, Nguyen-7, Nguyen-12) to represent varied search space sizes and difficulties. Given a candidate, we first compute the root mean square error (RMSE) between the candidate and target function over a set of uniformly sampled points from the specified domain of each task (e.g. 20 points in $(0, 1)$ for Nguyen-12). The fitness is obtained by applying a "flip-and-squash" function $(2/\pi)\arctan(x\pi/2)$ so that fitness is within $[0, 1]$ and lower RMSE maps to higher fitness. If any candidate function generates NAN, its fitness is defined as 0. The DAG has a maximum of 15 vertices with 8 possible ops.

AutoRL Loss Functions introduced in [4] are represented as DAGs and the goal is to find RL loss functions that enable efficient RL training. The DAG has a maximum of 20 vertexes excluding the input and parameters vertexes. Remaining vertexes can choose from a list of 26 possible ops as detailed in [4]. We use three environments (CartPole-v0, MountainCar-v0, Acrobot-v1) from the OpenAI Gym suite [39] with the CartPole environment used as a "hurdle" environment. Fitness is defined as the average normalized return from all three environments if the performance is greater than 0.6 on Cartpole, otherwise only the normalized return from Cartpole is used (and the rest two environments are skipped to save compute). Random generators and mutators are the same as in [4].

ML optimizers (Hero) [5] defines a search space for ML optimizers and uses evolution to discover a novel optimizer that achieves state-of-the-art performance on a wide range of machine learning tasks.

The search space consists of a sequence of python assignment statements which can use common math functions or high level functions such as linear interpolation (example in Figure 1 top left). We convert the sequence of statements to an equivalent DAG. Evolution is warm started from AdamW and fitness is the validation log likelihood loss of the trained model. [5] uses up to 100 TPUs v4 per experiment so to reduce compute, we focus on a smaller training configuration that can train an optimizer on a language modelling task in less than 10 minutes on a GPU (Nvidia Tesla V100).

Evolution and trainer setup

For population and tournament sizes, (P, T), we use (100, 20) for NAS-Bench-101, (100, 25) for Nguyen, (300, 25) for AutoRL, and (100, 25) for Hero. We swept over tournament size for the baseline. For all tasks we use the GPS graph Transformer [40] except for ML optimizers where we use SAT [41] for the graph encoder. Samples from evolution are added to a fixed size replay buffer that is used to train the model online (Algorithm 2). We use Adam with a learning rate of $1\mathrm{e}{-4}$ and train the model over the replay buffer at a fixed frequency as new samples are added. More details on architecture and training are in Appendix 6.1. For evolution curves, we plot 95% confidence intervals (± 2 standard errors) over at least 5 seeds.

4.2 Results

4.2.1 Can good predictors possibly speed up evolution?

Although a good predictor can improve local search, it is not guaranteed that it improves long term performance even if the model is perfect. In addition, training an accurate predictor online confounds with the predictor's effect on evolution. For example, an inaccurate model may lead to the collection of more sub-optimal data for training, creating a negative feedback loop. To reduce confounding factors and as a sanity check, we first run experiments assuming access to an oracle model and measure the effect of the model's simulated accuracy on evolution.

We conduct (noisy) oracle experiments using NAS-Bench-101 and Nguyen tasks because oracle models are available for them (pre-computed for NAS-Bench-101 and fast to compute for Nguyen tasks). Given an oracle $g(\cdot)$ that assigns a fitness for any candidate, we can simulate a binary prediction model $f(\cdot,\cdot)$ of any accuracy a by randomly flipping the ground truth ordering with probability $1-a$. More precisely, given two candidates x and y, $f(x,y) = \mathbb{1}_{\{g(x)>g(y)\}}$ with probability a and $f(x,y) = 1 - \mathbb{1}_{\{g(x)>g(y)\}}$ with probability $1-a$. We use the noisy oracle models in the PAM-RT setting. Figure 2 (left) shows that on NAS-Bench-101 evolution's performance

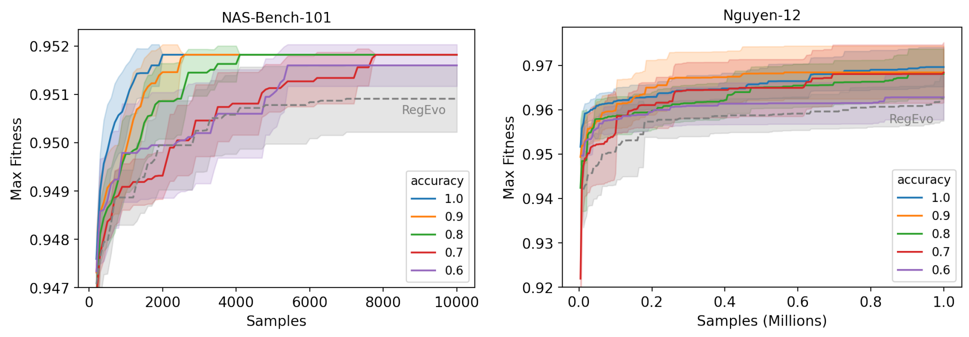


Figure 2: Noisy oracle experiments on NAS-Bench-101 and Nguyen-12 show the benefit of a using a predictor with evolution. Dashed curves show regularized evolution baseline.

correlates well with model's accuracy where a perfect model (100% accuracy) converges the fastest and the least accurate model (60%) converges the slowest. This suggests that improving model's accuracy in general help with evolution for NasBench101. For Nguyen-12, the correlation is weaker, as depicted in Figure 2 (right), however models with $> 70\%$ accuracy still provide a benefit.

These experiments support the use of predictors to speed up evolution. They also highlight the importance of the predictor's accuracy for end-to-end performance, motivating us doing design ablations in Section 4.2.4 to identify models that work best.

4.2.2 Trained predictors speed up evolution

We perform experiments on larger primitive-based search spaces to see if our method can speed up evolution. Nguyen, AutoRL, and Hero have much larger and sparser search spaces than NAS-Bench-101. Search space size can approximately be measured by number of vertices and possible ops for each vertex. These 3 task for (# nodes, # ops) are (15, 8), (20, 26), and (30, 43) whereas NAS-Bench-101 is (7, 3). In Figure 3 and 4, we show that our method increases the convergence speed of evolution and reaches a higher maximum fitness with fewer samples. Across all 5 Nguyen tasks, our method significantly speeds up evolution and for example achieves the maximum fitness on Nguyen-3 in 20k samples while RegEvo fails to reach the maximum fitness in 100k samples. On Hero, our method achieves the same average maximum fitness in 10k samples as RegEvo does in 37k samples (both the baseline and PAM-RT use FEC). On AutoRL, our method achieves the same fitness as RegEvo in 4x less compute.

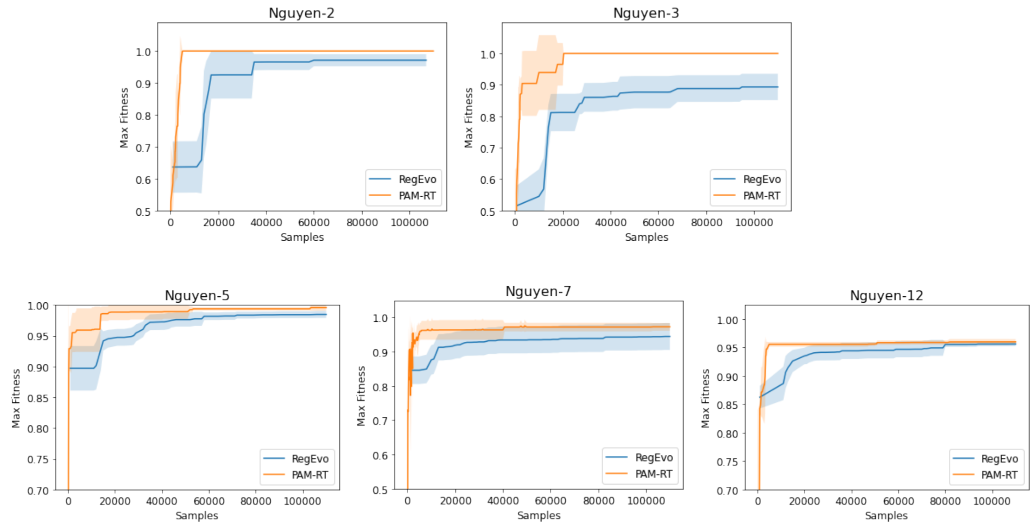


Figure 3: On all symbolic regression tasks, our method PAM-RT can provide faster convergence compared to regularized evolution.

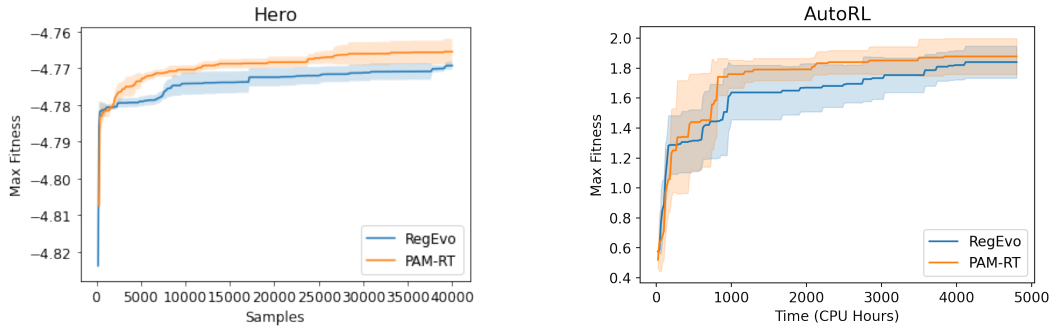
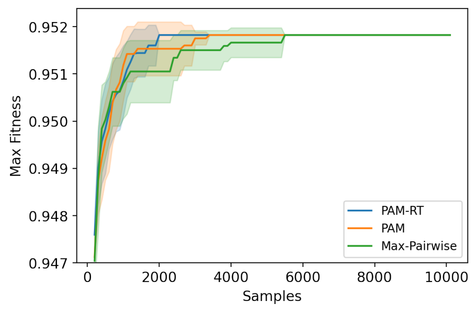


Figure 4: PAM-RT has better sample efficiency and higher maximum performance compared to regularized evolution on harder search spaces, Hero and AutoRL.

4.2.3 PAM-RT is an effective mutation strategy

We show that PAM-RT is an effective way to combine binary predictor with evolution compared to other mutation strategies. In Figure 5 (left), we compare PAM, PAM-RT, and Max-Pairwise on NAS-Bench-101 using an oracle model. We observe that while all three methods converge to the same max reward value, PAM-RT reaches that value faster. In Figure 5 (right), we show the cumulative hill climbing rate for these methods along with the regularized evolution baseline. The cumulative hill climbing rate at step N is defined as the average of the observed one-sample hill climbing rate at all steps up to N. PAM-RT and PAM both have higher rate than regularized evolution, and PAM-RT has slightly higher rate than PAM. This agrees with the analysis in Section 3.3. In Figure 9 (Appendix), we compare the cumulative hill climbing rate of PAM-RT for different noisy oracles and show that evolution's performance strongly correlates with this rate on NAS-Bench-101. Note that Max-Pairwise has an overall lower cumulative hill climb rate even compared to the regularized evolution baseline. One hypothesis is that selecting the max from a list of candidates exploits the model more in the early phase of evolution, making subsequent local improvements harder.



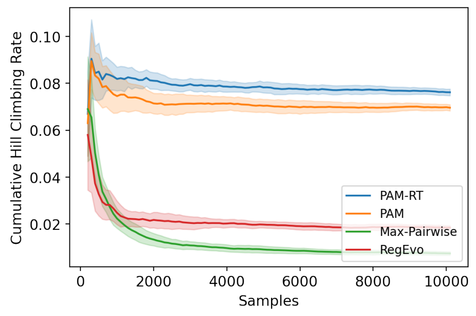
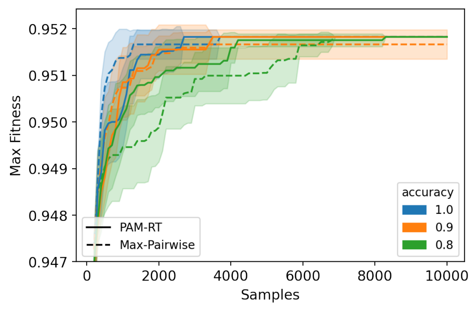
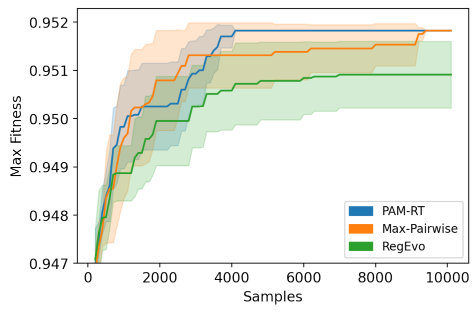


Figure 5: Comparison of three mutation strategies on NAS-Bench-101. **Left**: Fitness curves using the perfect oracle showing PAM-RT with the quickest convergence. **Right**: PAM-RT has the highest cumulative hill climbing rate compared to other strategies.

We also show PAM-RT is robust when the model is not perfect. In Figure 6a, we show that PAM-RT is similar to or better than Max-Pairwise when using noisy oracles of different accuracies. In Figure 6b, we show results with learned models and compare them with regularized evolution. Both PAM-RT and Max-Pairwise are better than the baseline, but PAM-RT reaches the best performance faster than Max-Pairwise. We emphasize that the same model architecture and training logic are used in both cases.



(a) Noisy Oracles



(b) Learned Models

Figure 6: Comparison of PAM-RT and Max-Pairwise on NAS-Bench-101. **Left**: Fitness curves showing PAM-RT (solid line) is more robust to predictor accuracy using a noisy oracle compared to Max-Pairwise (dashed line). **Right**: Using an online learned predictor and regularized evolution, PAM-RT is still better than Max-Pairwise.

4.2.4 Design Ablations

Here we show what design choices matter for PAM-RT. For ablations, we use a set of Nguyen tasks as they have varying difficulty and search space sizes while still having cheap evaluation for easier analysis.

Binary vs Regression: Most prior work uses a regression predictor so we study the effect of using a regression vs a binary predictor. We collect $10,000$ samples from regularized evolution, train for 1000 epochs on a training set, and measure accuracy on a held out test set. The accuracy for the regression predictor is measured by comparing which of the two predicted scores is higher for a pair of graphs. In Figure 7, we show that the binary predictor achieves significantly higher test accuracy across all symbolic regression tasks. This further motivates the use of binary predictors since accuracy can have a large impact on evolution convergence as shown in Figure 2 (left). As [14] pointed out, a binary predictor can leverage $O(N^2)$ training samples with N evaluations, but a regression can only use $O(N)$ training samples. A regression predictor must also generalize to unseen higher fitnesses while a binary predictor just has to predict comparisons. This partly explains why they are more effective and generalize better.

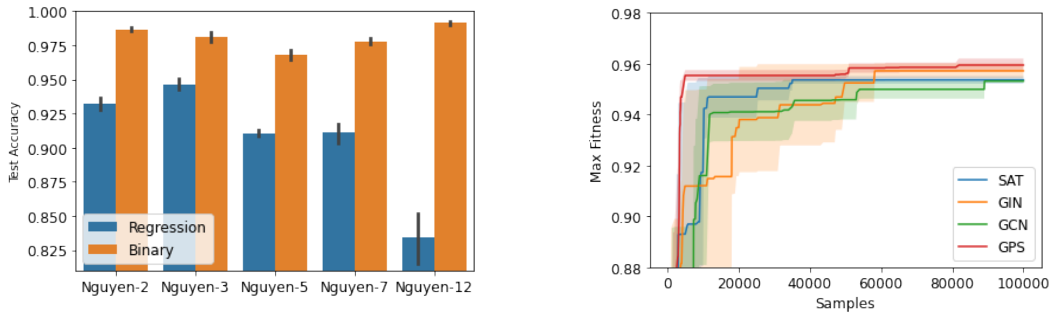


Figure 7: **Left**: Test accuracy for regression vs binary predictor on random pairs from a fixed 10k dataset collected with regularized evolution for a range of symbolic regression tasks. Binary predictors have consistently higher test accuracy. **Right**: Task performance on Nguyen-12 task using different GNN architecture with GPS being the most performant architecture.

Does the choice of GNN matter? We perform an ablation experiment comparing different popular GNNs and graph Transformers including GPS [40], SAT [41], GCN [42], and GIN [43]. We observe that the choice of GNN architecture has an effect on the convergence of evolution. For the hardest symbolic regression task, Nguyen-12, GPS performs the best in terms of converging quickly to the highest max fitness. Older networks such as GCN and GIN take longer to converge and reach a lower max fitness.

5 Discussion

We have presented a method for speeding up evolution with learned binary predictors to more efficiently search for a wide range of ML components. The same graph representation and GNN-based predictor can be used over diverse domains and provide faster evolutionary search for symbolic reqression equations, ML optimizers, and RL loss functions, as well as traditional NAS spaces. Through ablations, we showed the importance of the mutation strategy, the use of a binary predictor instead of a regression model, and state-of-the-art GNN architectures.

Predictor's accuracy and generalization capability are important for this method to provide large benefits. Immediate future work could focus on representation learning for better generalization over graphs such as better graph architecture priors, unsupervised objectives, or even sharing data across tasks. Longer term, one could consider alternative optimization methods for searching over ML programs, but that still use learned representations over computation graphs such as an RL policy. One potential avenue could be using generative models such as LLMs to propose promising candidates. Speeding up the search for ML components with learning is a promising direction because it could eventually create a virtuous cycle of continuous self improvement.

Author Contributions

JC, YM, and ER conceived the project. JC and YM proposed and implemented the algorithm, ran the experiments, performed the analysis, and wrote the paper. ER was the principal advisor. GT and AF provided feedback.

Acknowledgments

We thank Hanjun Dai, Yujin Tang, Yingtao Tian, Xingyou Song, Daiyi Peng, Yutian Chen, and David So for helpful discussions. We thank Bryan Tantisujjatham for helping with code infrastructure. We are grateful to Chen Liang for reviewing the manuscript.

References

[1] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey, 2019.

[2] Xin He, Kaiyong Zhao, and Xiaowen Chu. Automl: A survey of the state-of-the-art. *Knowl. Based Syst.*, 212:106622, 2021.

[3] Esteban Real, Chen Liang, David R. So, and Quoc V. Le. Automl-zero: Evolving machine learning algorithms from scratch. In *International Conference on Machine Learning*, 2020.

[4] John D. Co-Reyes, Yingjie Miao, Daiyi Peng, Esteban Real, Quoc V. Le, Sergey Levine, Honglak Lee, and Aleksandra Faust. Evolving reinforcement learning algorithms. In *ICLR*, 2021.

[5] Xiangning Chen, Chen Liang, Da Huang, Esteban Real, Kaiyuan Wang, Yao Liu, Hieu Pham, Xuanyi Dong, Thang Luong, Cho-Jui Hsieh, Yifeng Lu, and Quoc V. Le. Symbolic discovery of optimization algorithms, 2023.

[6] Kirthevasan Kandasamy, Willie Neiswanger, Jeff Schneider, Barnabás Póczos, and Eric P. Xing. Neural architecture search with bayesian optimisation and optimal transport. In *NeurIPS*, 2018.

[7] Colin White, Willie Neiswanger, and Yash Savani. BANANAS: bayesian optimization with neural architectures for neural architecture search. In *AAAI*, 2021.

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

[9] Hieu Pham, Melody Y. Guan, Barret Zoph, Quoc V. Le, and Jeff Dean. Efficient neural architecture search via parameter sharing. In *ICML*, 2018.

[10] Hanxiao Liu, Karen Simonyan, and Yiming Yang. DARTS: differentiable architecture search. *CoRR*, abs/1806.09055, 2018.

[11] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V. Le. Regularized evolution for image classifier architecture search. In *AAAI*, 2019.

[12] Ryan Gillard, Stephen Jonany, Yingjie Miao, Michael Munn, Connal de Souza, Jonathan Dungay, Chen Liang, David R. So, Quoc V. Le, and Esteban Real. Unified functional hashing in automatic machine learning, 2023.

[13] Colin White, Arber Zela, Robin Ru, Yang Liu, and Frank Hutter. How powerful are performance predictors in neural architecture search? In *NeurIPS*, 2021.

[14] 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.

[15] Hao Hao, Jinyuan Zhang, Xiaofen Lu, and Aimin Zhou. Binary relation learning and classifying for preselection in evolutionary algorithms. *IEEE Trans. Evol. Comput.*, 24(6):1125–1139, 2020.

[16] Chris Ying, Aaron Klein, Eric Christiansen, Esteban Real, Kevin Murphy, and Frank Hutter. Nas-bench-101: Towards reproducible neural architecture search. In *ICML*, 2019.

[17] Liam Li and Ameet Talwalkar. Random search and reproducibility for neural architecture search. In *Proceedings of the Thirty-Fifth Conference on Uncertainty in Artificial Intelligence, UAI*, 2019.

[18] Chenxi Liu, Barret Zoph, Maxim Neumann, Jonathon Shlens, Wei Hua, Li-Jia Li, Li Fei-Fei, Alan L. Yuille, Jonathan Huang, and Kevin Murphy. Progressive neural architecture search. In *ECCV*, 2018.

[19] Wei Wen, Hanxiao Liu, Yiran Chen, Hai Helen Li, Gabriel Bender, and Pieter-Jan Kindermans. Neural predictor for neural architecture search. In *ECCV*, 2020.

[20] Shun Lu, Jixiang Li, Jianchao Tan, Sen Yang, and Ji Liu. TNASP: A transformer-based NAS predictor with a self-evolution framework. In *NeurIPS*, 2021.

[21] Yanan Sun, Handing Wang, Bing Xue, Yaochu Jin, Gary G. Yen, and Mengjie Zhang. Surrogate-assisted evolutionary deep learning using an end-to-end random forest-based performance predictor. *IEEE Transactions on Evolutionary Computation*, 24(2):350–364, 2020.

[22] Yameng Peng, Andy Song, Vic Ciesielski, Haytham M. Fayek, and Xiaojun Chang. PRE-NAS: predictor-assisted evolutionary neural architecture search. In *GECCO*, 2022.

[23] Yixing Xu, Yunhe Wang, Kai Han, Yehui Tang, Shangling Jui, Chunjing Xu, and Chang Xu. Renas: Relativistic evaluation of neural architecture search. In *CVPR*, 2021.

[24] Yaofo Chen, Yong Guo, Qi Chen, Minli Li, Wei Zeng, Yaowei Wang, and Mingkui Tan. Contrastive neural architecture search with neural architecture comparators. In *CVPR*, 2021.

[25] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 3 edition, 2010.

[26] Thomas Elsken, Jan-Hendrik Metzen, and Frank Hutter. Simple and efficient architecture search for convolutional neural networks, 2017.

[27] Prajit Ramachandran, Barret Zoph, and Quoc V. Le. Searching for activation functions. *ArXiv*, abs/1710.05941, 2018.

[28] Irwan Bello, Barret Zoph, Vijay Vasudevan, and Quoc V. Le. Neural optimizer search with reinforcement learning. In *ICML*, volume 70 of *Proceedings of Machine Learning Research*, pages 459–468. PMLR, 2017.

[29] Daniel A. Abolafia, Mohammad Norouzi, and Quoc V. Le. Neural program synthesis with priority queue training. *ArXiv*, abs/1801.03526, 2018.

[30] Brenden K. Petersen, Mikel Landajuela, T. Nathan Mundhenk, Cláudio Prata Santiago, Sookyung Kim, and Joanne Taery Kim. Deep symbolic regression: Recovering mathematical expressions from data via risk-seeking policy gradients. In *ICLR*, 2021.

[31] Angelica Chen, David Dohan, and David R. So. Evoprompting: Language models for code-level neural architecture search. *ArXiv*, abs/2302.14838, 2023.

[32] Terrell Nathan Mundhenk, Mikel Landajuela, Ruben Glatt, Cláudio P. Santiago, Daniel M. Faissol, and Brenden K. Petersen. Symbolic regression via neural-guided genetic programming population seeding. *ArXiv*, abs/2111.00053, 2021.

[33] Nguyen Quang Uy, Nguyen Xuan Hoai, Michael O'Neill, Robert I. McKay, and Edgar Galván López. Semantically-based crossover in genetic programming: application to real-valued symbolic regression. *Genetic Programming and Evolvable Machines*, 12:91–119, 2011.

[34] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization, 2019.

[35] John R. Koza. Genetic programming - on the programming of computers by means of natural selection. In *Complex Adaptive Systems*, 1993.

[36] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *ArXiv*, abs/1312.5602, 2013.

[37] David E. Goldberg and Kalyanmoy Deb. A comparative analysis of selection schemes used in genetic algorithms. In *Foundations of Genetic Algorithms*, 1990.

[38] Alex Krizhevsky. Learning multiple layers of features from tiny images. 2009.

[39] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[40] Ladislav Rampásek, Michael Galkin, Vijay Prakash Dwivedi, Anh Tuan Luu, Guy Wolf, and Dominique Beaini. Recipe for a general, powerful, scalable graph transformer. In *NeurIPS*, 2022.

[41] Dexiong Chen, Leslie O'Bray, and Karsten M. Borgwardt. Structure-aware transformer for graph representation learning. In *International Conference on Machine Learning*, 2022.

[42] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *ICLR (Poster)*, 2017.

[43] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? In *ICLR*, 2019.

[44] Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[45] Weihua Hu, Bowen Liu, Joseph Gomes, Marinka Zitnik, Percy Liang, Vijay S. Pande, and Jure Leskovec. Strategies for pre-training graph neural networks. *arXiv: Learning*, 2019.

6 Supplementary Material

6.1 Training Details

Single-process training. For smaller scale tasks (NAS-Bench-101 and Ngyuen), we use a single-process training procedure (Algorithm 2) which alternates between two phases: collecting new samples and fitting the binary predictor. An optional exploration parameter ϵ may be used to allow a small probability of using the vanilla mutator instead of a mutation strategy with the predictor.

Distributed training. For larger scale tasks (Hero and AutoRL), we use asynchronous distributed training to speed up candidate evaluation and model training. We use a central *population server* that tracks all the candidates discovered so far while maintaining a population buffer. A single *learner* periodically syncs new candidates data from the population server and maintains the bounded replay buffer. The learner starts learning as soon as enough data has been collected, and it learns continuously—not blocked by workers' progress. Parallel *worker*s sync population buffer and model parameters from the population server and the learner respectively. Workers are responsible for tournament selection, mutation, and candidate evaluation. Candidates are sent to the population server once evaluation completes.

Architecture details. For the graph encoder, we use the standard GPS architecture from Pytorch Geometric [44] with 10 layers of GPS convolutions. Nodes and edges from the DAG are input into an embedding layer of size 64 and 16 respectively to obtain node and edge embeddings which are then processed by the GNN. The GPS convolution has 64 input channels, 4 heads with an attention dropout of 0.5, and uses a GINE [45] message passing layer which uses a 2 layer (64, 64) MLP. A final global addition pooling layer across node features is applied to obtain the graph embedding of size 64. The binary predictor is a 2 layer MLP of size (64, 64) with dropout of 0.2. For all MLPs, we use ReLU activations.

Other training details. The predictor is trained with Adam with a learning rate of $1\mathrm{e}{-4}$ and weight decay of $1\mathrm{e}{-5}$. Samples from evolution are added to the replay buffer. For NAS-Bench-101 we train the predictor for 100 epochs every 100 samples. For Nguyen, we train the predictor for 10 epochs every 100 samples. One epoch is iterating over the shuffled replay buffer twice to obtain pairs of graphs and then minimizing the binary classification loss on these pairs. For Hero and AutoRL, we use the asynchronous distributed training logic described above. We detail configurations and hyperparameters in Table 1.

Task	NAS-Bench-101	Nguyen	Hero	AutoRL
Population Size	100	100	100	300
Tournament Size	20	25	25	25
Num Workers	1 CPU	1 GPU	20 GPUs	100 CPUs
Replay Buffer Size	1000	10000	50000	100000
Min Data Before Model Use	100	100	1000	3000

Table 1: Hyperparameters for tasks

Compute budget. Evaluating time for a candidate varies greatly for AutoRL (from under one minute to over an hour due to the hurdle mechanism). Therefore, we set a timing budget of total 4800 CPU hours (summed from 100 parallel workers), instead of a budget on number of samples, as they vary greatly for a given time budget.

Training tricks. We discuss important tricks for training and using the model. We found that delaying using the model until enough samples have filled the replay buffer to be important for Hero (1000 samples) and AutoRL ($3,000$ samples). This could be because these search spaces are larger, so the model can easily overfit to a small number of samples and struggle to generalize. Another important hyperparameter related to this one, was using a large enough replay buffer size. Smaller buffer sizes (< 1000) had a detrimental effect.

Algorithm 2 Online Training of Binary Predictors with Evolution

```
Input: Total samples $S$, random mutation probability $\epsilon$, training frequency $F$, max attempts $K$.
 1: Initialize population buffer $P$, predictor $f$, replay buffer $D$
 2: $samples \leftarrow 0$
 3: while $samples < S$ do
 4:     if $Uniform(0,1) < \epsilon$ or $samples <$ min data then
 5:         $child \leftarrow RandomMutation(P)$
 6:     else
 7:         $child \leftarrow$ PAM-RT($P, f, K$)          ▷ Select child with Algorithm 1
 8:     end if
 9:     if $samples \bmod F == 0$ then
10:         $f \leftarrow TrainBinary(f, D)$
11:     end if
12:     $D \leftarrow D \cup child$                    ▷ Remove oldest if hit max $D$ size
13:     $P \leftarrow P \cup child$                    ▷ Remove oldest if hit max $P$ size
14:     $samples \leftarrow samples + 1$
15: end while
```

6.2 How PAM improves local search

We show how the modified hill climbing rate (defined in Section 3.3) relates to the natural hill climbing rate for a given model accuracy using PAM. As a corollary, we show the modified rate is always higher than the natural rate—*if the model is better than random.*

For simplicity, assume we can retry as many times as needed (i.e., $K \to \infty$ in Algorithm 1). Let p denote the parent, c denote the child, q be the probability $c > p$ and a be the binary predictor's accuracy. At each step, the probability of accepting c is:

$$p_{accept} = qa + (1-q)(1-a). \tag{1}$$

Here we assume the event $c > p$ (or $c \leq p$) and the event that model makes a correct (or incorrect) prediction are independent. Let $d = 1 - p_{accept}$ denote the probability we reject c.

For a sequence of trials, let random variable $C_i \in \{0, 1\}$ denote if the child sampled at i-th round is better than the parent (*according to the ground truth*) and let the random variable $A_i \in \{0, 1\}$ denote we accept the child *according to the model*. The probability that we eventually accept a child that's better than the parent is the sum of probability of the following mutually exclusive events:

1. The first mutation is good and we accept it: $P(C_1 = 1)P(A_1 = 1|C_1 = 1) = q \cdot a$.
2. We reject the first mutation, and the second mutation is good and we accept it: $P(A_1 = 0)P(C_2 = 1)P(A_2 = 1|C_2 = 1) = d \cdot q \cdot a$.
3. In general, $d^{n-1} \cdot q \cdot a$ if we accept at n-th trial (n starts with 1).

Summing over the geometric series gives

$$\begin{aligned} q \cdot a \cdot \frac{1}{1-d} &= \frac{qa}{p_{accept}} \\ &= \frac{qa}{qa + (1-q)(1-a)} \\ &= \frac{1}{1 + \frac{1-a}{a} \cdot (\frac{1}{q} - 1)}, \end{aligned}$$

recovering the result in Section 3.3. It is easy to see that if the model accuracy is better than random ($a > 0.5$), the modified hill climb rate is always higher than the natural hill climb rate. Increasing model accuracy leads to larger gains (Figure 8).

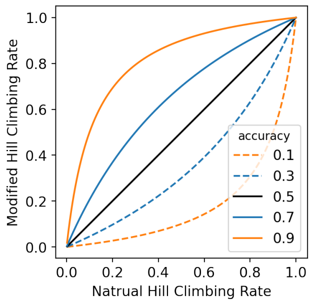


Figure 8: Modified hill climbing rate as a function of natural hill climbing rate for different model accuracy levels (using PAM).

6.3 Exploitation and Exploration of PAM-RT on NAS-Bench-101

We show PAM-RT strikes a balance between exploitation and exploration on NAS-Bench-101. In Figure 9, we show that on NAS-Bench-101, evolution's performance strongly correlates with the modified hill climbing rate (and the predictor's accuracy). This suggests that PAM-RT can exploit the model for improved local search. In Figure 10, we study the diversity of samples both in the population buffer and during the whole evolution process for the experiments used in Figure 5. These measurements may serve as an indicator for the degree of exploration. We observe that PAM and PAM-RT explore more than the baseline RegEvo whereas Max-Pairwise explores much less.

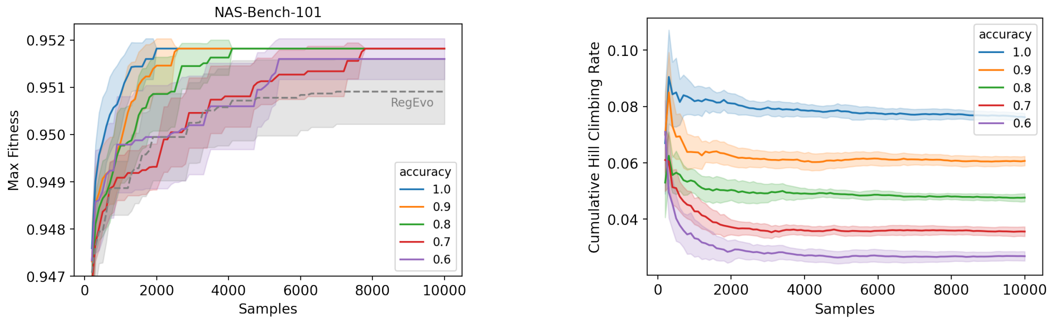


Figure 9: Strong correlation among evolution's performance, cumulative hill climbing rate, and the predictor's accuracy on NAS-Bench-101. **Left**: Evolution curves for noisy oracles of different accuracies. **Right**: Cumulative hill climbing rates in the corresponding experiments.

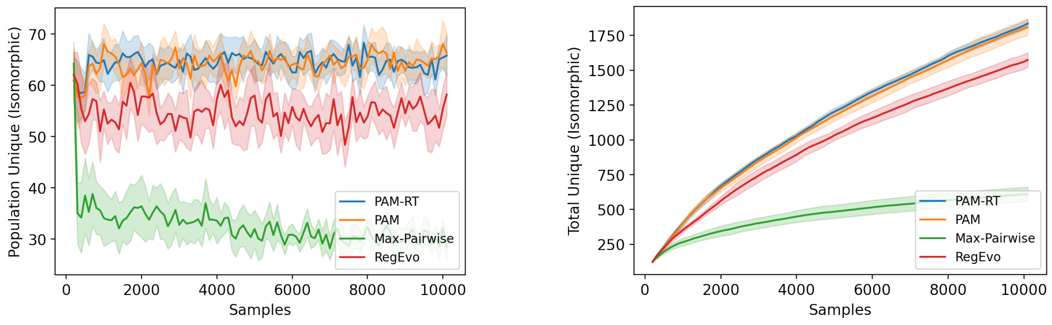
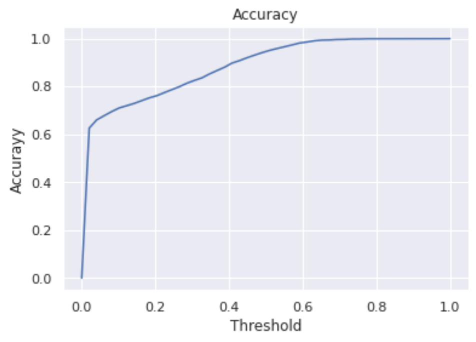


Figure 10: A followup study for Figure 5 on NAS-Bench-101 using the perfect oracle. **Left**: Number of unique candidates (by isomorphism [16]) in the population buffer. **Right**: Number of total unique candidates (by isomorphism).

6.4 Empirical Analysis

To better understand how the model helps and where it fails, we perform a counterfactual experiment where we run RegEvo as usual and train the PAM-RT model online but do not use it for mutations.



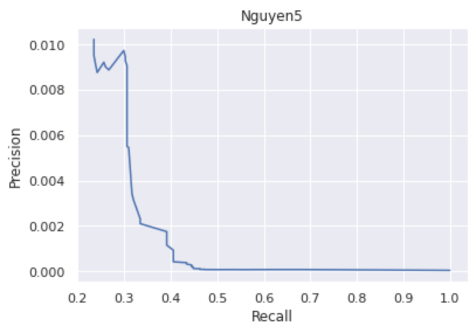
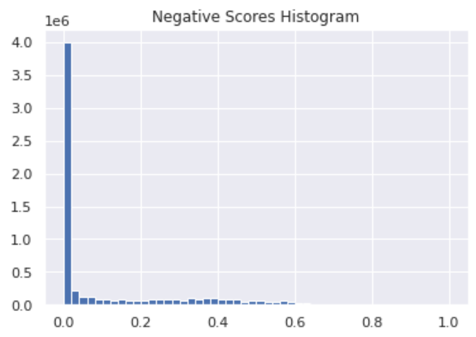


Figure 11: **Left**: Accuracy of our model for various classifier thresholds. Evaluated on data collected during RegEvo with a model trained online on Nguyen5. **Right**: Precision recall curve of our model.



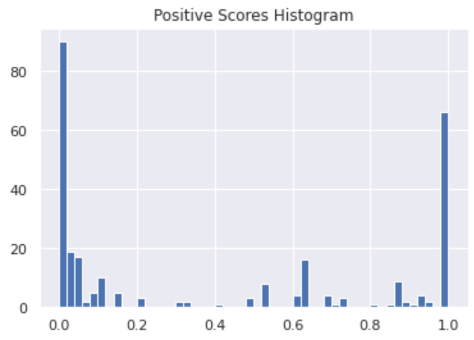


Figure 12: **Left**: Histogram of scores for negative individuals. **Right**: Histogram of scores for positive individuals.

This gives insight into the performance of the model given the ground truth. We run this experiment on Nguyen-5 for easier analysis. At each mutation step, we sample 64 candidates, use the model to score these candidates, and save the score and actual fitness of each candidate. We run this for 100k steps to get $6.4e6$ scores. We define a positive individual to be the case where its fitness is higher than the parent, and a negative individual is defined as the complement. In Figure 11, we plot the accuracy curve and the precision-recall curve of these candidates over a range of classifier thresholds. As expected, accuracy is relatively high (> 0.95 for a classifier threshold of 0.5). During evolution most individuals are negatives and the model correctly scores most of these individuals with low scores. We observe this in Figure 12 where most of the negative scores are less than 0.5. However, in Figure 11, we see that precision is quite low (around 0.01) and drops quickly to close to 0 for higher levels of recall. Positive individuals are quite rare during evolution and while the model can correctly score some of them (0.43 true positive rate for 0.5 threshold) as seen in Figure 12, there are a good portion of positives with less than 0.5 score. From this analysis, we see that the model is good at ruling out many negative individuals and this would logically make evolution more efficient. However the model could be better at letting more positives individuals through and reducing the number of false negatives. Further analysis of why there is a high number of false negatives with almost 0 score could provide insight for how to improve the model.