Prometheus: A Recursively Self-Improving NAS System

Alex Zhang[1,*], **Hui Lu**[2,**]

Received April 17, 2025
Accepted August 22, 2025
Electronic access September 30, 2025

Neural Architecture Search (NAS) automates the process of neural network design, enabling the discovery of architectures that can surpass those manually constructed by human experts. For NAS systems involving a controller (often trained with reinforcement learning (RL)), one limitation is the intelligence of the controller. We introduce Prometheus to address this barrier. Project Prometheus is a series of NAS systems that utilize network morphism techniques, which allow edits to a neural network to be applied during training with minimal drop in performance, to edit both a convolutional neural network trained on image recognition tasks and also itself. The self-editing allows it to increase its own processing capacity to achieve better rewards from the RL system. Prometheus is currently in a proof- of-concept stage and has several limitations. One notable constraint is that the timing of its self-edits is governed by a human-defined heuristic, which restricts the system's autonomy. Nevertheless, despite these early-stage limitations, the final system still achieves very competitive performance on the Canadian Institute For Advanced Research 10-class image dataset (CIFAR-10), achieving a mean accuracy of 95.47%±0.60%. It also performs competitively on the Street View House Numbers dataset (SVHN), the Fashion Modified National Institute of Standards and Technology dataset (Fashion-MNIST), and the CIFAR-100 dataset, with mean accuracies of 97.09%±0.15%, 95.57%±0.20%, and 73.26%±0.81% respectively.

Keywords: Neural Architecture Search, Reinforcement Learning, Self-Improving Systems, Artificial Intelligence, Deep Learning.

1 Introduction

Artificial Intelligence and machine learning have evolved exponentially in recent years. A particularly powerful branch of machine learning, called deep learning, uses models known as neural networks, which are computational systems loosely inspired by the structure of the human brain. The design of a neural network (the number of layers, the number of neurons per layer, etc.) is called its architecture. Using neural networks, deep learning has grown into an indispensable tool for humanity, allowing us to utilize patterns in data. However, designing an effective neural network architecture is often very intuition-based and time-consuming. Optimization often takes significant amounts of time and human effort. The subfield of neural architecture search (NAS) addresses this issue directly, as it seeks to automate the process of neural network architecture optimization.

There are many types of NAS systems, with the earliest forms being evolutionary algorithms, which use genetic algorithms to evolve network topologies. One notable work is that of Zoph and Le,[1] which famously utilized over 500,000 GPU hours to find an optimal architecture, achieving a state-of-the-art accuracy of 96.35% on the Canadian Institute For Advanced Research 10-class image dataset (CIFAR-10). Subsequent works have striven to minimize the computational effort required to train NAS systems, like DARTS (Differentiable Architecture Search)[2] and ENAS (Efficient Neural Architecture Search)[3], which rely on clever strategies like gradient-based search and weight sharing to lessen computation. More recent advances include extensible zero-cost proxies like Eproxy[4], robust training-free NAS techniques[5], and hardware-aware multi-objective differentiable NAS[6]. Beyond efficiency, several works tackle controller adaptability. MetaD2A[7] demonstrates dataset-to-architecture transfer via meta-learning. Graph HyperNetworks[8] generate weights directly from an architectures graph, allowing instant evaluation. Population Based Training[9] shows that learners can improve themselves during training by mutating their hyperparameters. Learned optimizers[10] replace hand-designed optimization rules with trainable neural controllers. Finally, NetAdapt[11] adapts networks mid-training to resource constraints using iterative pruning and morphism-based edits.

One notable strategy for adapting a network is network morphism[12]. Network morphism allows editing the overall architecture (macro-architecture) of a neural network during training without massive performance drops by initializing each new edit using its identity matrix, meaning that directly after performing the edit, the neural network is approximately identical to the original network, preventing performance drops[12]. This clever strategy was applied with reinforcement learning by Cai et al[13].

[1] *1960 Cate Mesa Road, Cate School, Carpinteria, CA, USA.*
[*] *Alex_Zhang@cate.org* ✉
[2] *Department of Computer Science, Missouri State University, 901 S. National Ave., Springfield, MO, USA.*
[**] *HuiLiu@MissouriState.edu* ✉

Despite these clever strategies addressing the computational bottleneck of NAS, the reinforcement learning (RL) controllers within RL NAS studies are still human-designed and can require immense human trial and error to perfect. There does not yet exist a system that optimizes an RL controller along with the target network. To address this research gap, we propose Prometheus, a proof-of-concept NAS pipeline that uses network morphism and RL to optimize the architectures of both the network trained on the central task (the target neural network) and the RL controller. Even in its proof-of-concept stage, Prometheus still achieves competitive accuracies across all the datasets it was evaluated on, demonstrating the adaptability of the recursively self-editing system. This paper demonstrates the novel self-editing method first by going through 2 preliminary systems, then on to the full system of Prometheus. Details of Prometheus are illustrated in section 2. We presented experiments and results in section 3. Finally, we discuss the systems limitations and further implications in section 4.

2 Methods

Prometheus is a system designed to explore the following question: Can a model not only learn, but also learn how to improve itself? This idea of recursive self-improvement is a powerful conceptual precursor to artificial general intelligence (AGI). Prometheus attempts to embody that principle by developing a controller that evolves itself while evolving the neural networks it designs. The system supports self-pruning and growth in response to training signals such as stagnation and instability. All edits are function-preserving, achieved through Net2Net-style initialization.

Network Morphism

It is crucial to first define network morphism. Network morphism techniques allow the agent to modify the target networks architecture without completely resetting its learned weights, accelerating the search process. The specific transformations implemented are:

- **Net2Wider (widen):** Following the Net2Wider operation from Chen et al.,[14] this action increases the width (number of output channels) of a convolutional layer. The weights for the new channels are initialized by replicating filters from the original layer, ensuring the new, wider layer can initially mimic the function of the old one.

- **Net2Deeper (deepen):** This operation, also from the original Net2Net,[14] increases network depth by inserting a new convolutional layer. The new layer is initialized to perform an identity mapping using a Dirac delta function for its weights. This allows the network to add the new layer without an initial drop in performance, giving it the capacity to learn a more complex function over time.

- **Custom Thinning (thin):** As the inverse of widening, we implemented a custom structured pruning operation. This action reduces a layers width by keeping the first k convolutional filters (where k is the new, smaller width) and discarding the rest. While this operation directly reduces model complexity and parameter count, it is a lossy transformation that alters the networks function and relies on subsequent fine-tuning to recover performance.

- **Custom Shallowing (shallow):** To reverse the deepening process, we implemented a direct layer removal operation. This edit identifies a target convolutional layer and removes it, along with its subsequent normalization layer, from the networks computational graph. This is our most aggressive complexity-reduction edit, as it fundamentally alters the networks representational capacity.

2.1 Preliminary Systems

System 1: LSTM Controller + Block-based Search + Basic Self-Edits

The first variant pairs a reinforcement learning agent (a long-short-term-memory (LSTM) controller) with pre-designed convolutional blocks (e.g., ResNet,[15] Inception[16], SE-blocks[17], Bottleneck Blocks[18]). Using network morphism, the controller could dynamically widen, deepen, or simplify the architecture during training. Crucially, this version introduces basic self-editing. The controller could widen or deepen its own LSTM architecture at fixed intervals, laying the groundwork for recursive improvement. Though the pre-designed blocks constrained its architectural expression, it was advantageous in its simplification of the task, especially when paired with a simple LSTM architecture.

System 2: GNN Controller + Fine-Grained Editing + Attention-Based Sampling

To address the inefficiency of block-level edits, System 2 replaces them with a granular, operation-level search. The controller, now a Graph Neural Network (GNN), represents the target convolutional neural network as a directed acyclic graph (DAG) and chooses from fine-grained edits like inserting individual 2D convolutional layers or Rectified Linear Unit (ReLU) layers, resizing channels, or creating skip connections. The controller self-edits using a new suite of GNN transformations, guided by attention mechanisms to focus search in promising regions. This expanded search space gives this system the expressive advantage, but it is harder to master this search space,

which is why the RL controller was upgraded to a more complex GNN.

2.2 System 3

Search Space

System 3, our last iteration, returned to block-level editing (though more basic) while retaining the flexible, graph-based representation pioneered in System 2. The design objective was maximum stability with near-autonomous operation, so the GNN-driven reinforcement-learning controller remains at the core, yet both its action space and the environments feedback signals have been substantially enriched. At the architectural level, every permissible edit is a function-preserving block transformation. The following are the possible edits:

- **Add Convolutional Block:** Appends a Conv2d $\rightarrow$ 2d batch normalization $\rightarrow$ ReLU trio at a stages end. The Conv2d is identity-initialized, and 2d batch normalization starts with $\gamma = 1.0$, $\beta = 0.0$, guaranteeing no immediate change in output (Net2Deeper style). After the block is appended using Net2DeeperNet, the agent chooses a channel multiplier to set the blocks width, which is applied using Net2WiderNet.
- **Add Linear Block:** Deepens the classifier by inserting a Linear $\rightarrow$ 1d batch normalization $\rightarrow$ ReLU right before the final layer. The new Linear layer is initialized as an identity matrix, again ensuring a function-preserving transformation. Once again, the agent is able to choose a width that is applied using Net2WiderNet.
- **Resize Layer:** Selects any Conv2d or Linear node and scales its output dimension by a chosen factor. A learned attention head first pinpoints the most promising layer, then picks the resize factor.
- **Add Skip Connection:** Creates a shortcut between two nodes within the same stage, chosen via attention over all valid sourcedestination pairs. If channel counts differ, an identity 1×1 convolution is inserted automatically to keep the DAG valid.

Evidently, the block-based search space of System 1 is adapted to be much simpler.

Initial Network

The search process is initialized with a simple VGG-style CNN backbone [19] configured for the CIFAR-10 dataset. This starting architecture consists of three sequential stages. Each stage features a core block of Conv2d $\rightarrow$ 2d batch normalization $\rightarrow$ ReLU, with channel dimensions increasing from 64 to 128 and finally to 256. The first two stages are each followed by a 2×2 max-pooling layer for spatial downsampling. The feature extractor is connected to a classifier head composed of an adaptive average pooling layer and a single linear layer that maps the final 256-dimensional feature vector to the 10 output classes. This well-defined initial state serves as the starting point for all subsequent architectural modifications by the RL agent. In addition, the initial network was trained for 50 epochs at the start. If it hadnt been initially trained, the first reward for the RL agent would always be massive and unrepresentative of the edit, as initial improvement to a neural network is usually fast.

Adaptive Training

To ensure stability and efficiency, the training process for both the target network and the RL controller is highly adaptive. For the target CNN, the number of post-edit training epochs scales in proportion to the parameter-change ratio, granting larger, more disruptive edits extra time to converge. The number of epochs, E_{post}, is calculated as:

$$E_{\text{post}} = \min\left(100, \text{round}\left(E_{\text{base}} \cdot \max\left(1.0, \frac{P_{t+1}}{P_t}\right)\right)\right) \quad (1)$$

where $E_{\text{base}} = 25$, and P_t, P_{t+1} are the parameter counts before and after the edit. Post-edit training utilizes a CosineAnnealingLR scheduler over this adaptive number of epochs. To further improve model performance, we apply strong augmentations, including `RandomCrop`, `RandAugment`, `RandomErasing`, and dataset-specific augmentations like `AutoAugment` when applicable. To accelerate training, the target CNN is trained using mixed precision with `autocast` and a `GradScaler`, and its gradients are clipped after each backward pass to prevent exploding gradients.

The RL component was optimized with Advantage Actor-Critic (A2C). The controllers parameters θ and the value functions parameters ϕ are updated by minimizing a composite loss function $L(\theta, \phi)$, which is composed of a policy loss for the actor, a value loss for the critic, and an entropy bonus to encourage exploration:

$$L(\theta, \phi) = -\log \pi_\theta(a_t|s_t)A_t + \beta_v(R_t - V_\phi(s_t))^2 - \beta_e H(\pi_\theta(\cdot|s_t)) \quad (2)$$

where $A_t = R_t - V_\phi(s_t)$ is the advantage, which is treated as a constant when updating the policy. The policy loss, $-\log \pi_\theta(a_t|s_t)A_t$, updates the actor ($\theta$) to make actions that produced a positive surprise more likely. The value loss, $(R_t - V_\phi(s_t))^2$, is a mean-squared error term that trains the critic (ϕ) to produce more accurate value estimates. Finally, H is the policy entropy, and β_v, β_e are loss coefficients. For our system, the entropy bonus was set to $-0.0005 \cdot \text{entropy}$, and the controllers gradients are clipped to a maximum norm of 5.0 before each update.

Safety Nets

Stability is protected by safety nets. After every edit, the system performs a single-batch dummy training cycle to detect any immediate gradient problems that could cause "Not a Number" (NaN) values or structural errors. We set a maximum of 10 edit attempts per iteration, where if 10 edits fail, the iteration is considered failed and skipped (with -2 penalty per failed attempt, -5 for failed iteration). AMP auto-disables after 3 consecutive NaN batches to maintain stability. A rollback is triggered if $\text{acc}_{t+1} < \text{acc}_{\text{best}} - \delta_{\text{rollback}}$, where $\delta_{\text{rollback}} = 0.04$.

Whenever a major architectural modification inflates the parameter count, the target CNN automatically triggers a warm-up schedule. During this phase, newly added batch normalization layers are initially frozen and then unfrozen over five "warm-up" epochs via a special learning rate ramp, preventing their untrained statistics from destabilizing gradients.

Before committing any edit, we clone the current Target CNN via a deep copy and apply the proposed edit on this clone to project the post-edit parameter count. If the projected count exceeds 30 million parameters, we reject the action without touching the live model. In the case that the target neural network bypasses the check and reaches past 30 million parameters, the model is automatically reverted, and the controller is given a -50 reward.

Whenever an edit alters the output channel dimension of a layer, the system immediately inspects the subsequent layer. If a dimensional mismatch is detected, the input channels of the subsequent layer are automatically rebuilt to match the new output dimension of the edited layer.

RL Controller

The RL controller is a GNN, specifically a Graph Convolutional Network (GCN)[20], that ingests the networks DAG. In this formulation, the target CNN is represented as a graph $G_t = (V, E)$, where each node $v_i \in V$ represents an operation with a feature vector $x_i \in \mathbb{R}^7$. The feature vector encodes operation type, normalized output channels, stage and operator indices, stride, a binary convolution flag, and spatial dimensions.

The target network begins as a simple three-stage backbone, which is then modified by the search process. Each operation (Conv2D[21], ReLU[22], etc.) is treated as a node, with edges representing data flow.

On every forward pass, the GCN performs message passing, letting each node aggregate information from its neighbors through multiple layers; the resulting graph-level context feeds into policy heads that issue structurally informed actions, surpassing the sequential view of an LSTM.

The GCN encoder processes the graph of the target network, G_t, to produce a matrix of numerical representations (node embeddings), denoted by Z. The formal relationship is:

$$Z = \text{GCN}(G_t) \tag{3}$$

The resulting matrix Z is defined as belonging to the space $Z \in \mathbb{R}^{|V| \times d_h}$. This notation specifies the matrixs structure and content:

- $\mathbb{R}$ indicates that the entries of the matrix are real numbers.
- $|V|$ is the total number of nodes (layers or operations) in the target networks graph. This determines the number of rows in the matrix.
- d_h is the dimensionality of the embedding for each node (e.g., a 128-dimensional vector). This determines the number of columns.

To make a decision, the controller needs a single, high-level summary of the entire network. This is achieved by creating a global graph embedding, z_g, via mean-pooling. This process simply averages all the individual node embeddings:

$$z_g = \frac{1}{|V|} \sum_{i=1}^{|V|} z_i \tag{4}$$

Here, z_i represents the embedding vector for a single node i (i.e., the i-th row of matrix Z), and the formula calculates their mean to produce a single, dense vector z_g representing the entire graph.

RL Implementation

The RL agent was given the ability to prune itself in addition to growing itself, but only after certain triggers. We define an iteration as one complete propose-train-evaluate cycle. If the validation accuracy does not beat the best validation accuracy after 5 iterations, a self-edit chosen by the RL agent along with a reversion to the best model occurs. A five-step window creates a balance between efficiency and guaranteed performance. When a model fails 3 dummy forward passes in a row, a self-edit occurs. While these heuristics serve as good measures to facilitate self-editing in this project, it is a limitation to the freedom of the RL agent. Upon triggering, we revert the target CNN weights to the best checkpoint so far, apply the sampled grow/prune action from the controller, and then resume the outer search loop. Growth choices comprise deepening the GNN, widening its hidden layers, and deepening a head, while pruning counterparts are pruning the GNN, shrinking the hidden layers, and pruning a head. Pruning options were only enabled after the meta-agents parameter count exceeded 15,000. The agent can thus expand its GNN depth, widen hidden sizes, or deepen policy heads when facing a harder search problem, yet later shed surplus capacity for efficiency once the task simplifies.

Replay Buffer and Reward Function

A replay buffer is implemented to increase the richness of the meta-agents learning of self-edits. The replay buffer (capacity 50, batch size 8) stores $(\text{state}, \text{action}, \text{reward})$ triples for self-edits of the controller. The reward function R_i at iteration i is formulated as:

$$R_i = 100 \cdot (\text{acc}_{i+1} - \text{acc}_i) \tag{5}$$

where acc_{i+1} is the post-edit validation accuracy, and acc_i is the pre-edit validation accuracy.

Adaptive Learning Rate

The meta-agents own learning rate is adaptive. It decays as the target models accuracy rises, enabling broad exploration early on and fine-grained exploitation near convergence. The meta-agents learning rate, lr_{meta}, is annealed based on the target accuracy acc:

$$lr_{\text{meta}} = lr_{\text{base}} - \min\left(1, \frac{\max(0, \text{acc} - \text{acc}_{\text{base}})}{\text{acc}_{\text{target}} - \text{acc}_{\text{base}}}\right)(lr_{\text{base}} - lr_{\min}) \tag{6}$$

where $\text{acc}_{\text{base}} = 0.80$ and $\text{acc}_{\text{target}} = 0.93$ define the accuracy range for annealing.

Attention Mechanism

As the architecture grows, the count of possible edits explodes. The agent employs an attention mechanism built atop the GNNs node embeddings:

- **Candidate identification:** Enumerate all valid targets for the current action (every Conv2D for resizing, every legal pair for a skip).
- **Scoring:** Feed the embeddings of these candidates into specialized heads that output a raw score (logit) per candidate. For a set of candidate nodes C, a scoring head f_{score} computes a logit q_i for each candidate node $v_i \in C$ based on its embedding z_i:

$$q_i = f_{\text{score}}(z_i) \tag{7}$$

- **Probability distribution:** Convert logits to a categorical distribution (softmax). The probability of selecting node v_i is given by:

$$P(\text{node} = v_i) = \frac{\exp(q_i)}{\sum_{v_j \in C} \exp(q_j)} \tag{8}$$

- **Informed sampling:** Sample from this distribution rather than greedily picking the top score, maintaining a balance between exploitation and exploration.

This combination of DAG-aware representation and attention-guided sampling enables Prometheus to navigate the immense edit space efficiently.

Reproducibility and Ethical Considerations

The three systems were all run on Google Colab Notebooks, and the link to them is here: `https://github.com/PlushyWushy/Prometheus/tree/main`. This study exclusively utilized publicly available and anonymized image datasets, avoiding concerns related to human subject privacy or data confidentiality. The research adheres to standard scientific practices for reproducibility and responsible innovation.

All experiments were conducted using PyTorch version 2.8.0, Torchvision 0.23.0, and PyTorch Geometric 2.6.1, running on Python 3.12 with CUDA support enabled via the +cu126 build. The discrete action space for resizing layers consisted of the factors [0.25, 0.5, 0.75, 1.25, 1.5, 1.75].

Other hyperparameters included:

- Batch size = 128
- AdamW learning rate = 0.001
- Dropout rate = 0.2
- Gradient clipping max norm = 5.0
- BN recalibration over 200 batches
- Warm-up LR ramp over 5 epochs when parameter ratio $>$ 1.2
- Meta-agent entropy coefficient = 0.001
- Edit-type embedding dimension = 16
- Initial GNN hidden dimension = 32 with 2 layers (minimums: 16 hidden units, 1 layer)
- Maximum MLP head depth = 8
- Meta-agent replay buffer capacity = 50 with batch size 8

3 Experiments and Results

To evaluate the performance and design principles of Prometheus, we conducted a series of experiments. Our evaluation is structured to analyze the evolution of our design by comparing the performance of our system variants and position our final system within the landscape of existing NAS methods.

3.1 Experimental Setup

Datasets. All three systems were evaluated on the CIFAR-10 dataset[23], which is a standard benchmark for image recognition and serves as the primary dataset for our development and comparative analysis. It consists of 60,000 32×32 color images across 10 balanced object classes, with 50,000 images for training and 10,000 for testing.

System 3 was additionally evaluated on the Street View House Numbers (SVHN), Fashion Modified National Institute of Standards and Technology (Fashion-MNIST), and CIFAR-100 datasets. The SVHN corpus[24] contains cropped 32×32 color images of house-number digits (09) collected from Google Street View, with 73,257 training samples and 26,032 test samples in the trainval split.

Fashion-MNIST[25], a tougher version of the original MNIST dataset, features 70,000 28×28 grayscale images across ten clothing categories, divided into 60,000 training examples and 10,000 test examples.

The CIFAR-100 dataset is another standard benchmark for image recognition and serves as a more challenging dataset for our system. It is a 100-class version of the CIFAR-10 dataset.

Implementation Details. We ran all searches on a single NVIDIA L4 Tensor Core GPU. The specific hyperparameters for each system variant were kept distinct to reflect their unique designs. For System 1, the target CNN was trained using AdamW with a weight decay of 10^{-4}. The LSTM controller used an Adam optimizer with a learning rate of 5×10^{-4} and a discount factor (γ) of 0.99. The A2C loss function used an entropy coefficient (β_e) of 0.01. The search was run for 150 iterations, and the reported accuracy is the average over 10 runs.

For System 2, the target CNN was trained with AdamW (weight decay 5×10^{-4}) and a Cosine Annealing learning rate schedule. The GNN controller used an Adam optimizer with a learning rate of 5×10^{-4}. The search was regularized with MixUp ($\alpha = 0.4$). The search was run for 200 iterations, and the reported accuracy is the average over 10 runs.

For System 3, each search was run for 150 iterations. The meta-agents learning rate was annealed from a base of 5×10^{-4} down to a minimum of 10^{-5}. All tests involving System 3 were conducted 10 times on random seeds.

3.2 Comparison of System Variants

We present a comparison of the three system variants on CIFAR-10. System 3 significantly outperformed the first two systems, achieving a final accuracy of $95.47\% \pm 0.60\%$ ($n = 10$) on CIFAR-10, while Systems 1 and 2 achieved only $82.91\% \pm 5.16\%$ ($n = 10$) and $90.57\% \pm 0.68\%$ ($n = 10$), respectively.

While the granular search of System 2 produced compact models (2.6M parameters average), its performance was limited. Though flexible, it had to navigate a massive search space, leading to a much harder policy to learn. In contrast, System 1, with its complex pre-designed blocks, discovered massive parameter networks that were numerically unstable and ultimately underperformed. It was given a toolkit of powerful blocks, but the simple LSTM controller was unable to combine them effectively.

Table 1 Comparative analysis of the Prometheus variants. All results are averaged over 10 runs.

Attribute	System 1	System 2	System 3
Controller	LSTM	GNN	GNN
Search Space	Block-Based	Granular	Block-Based
Final Params (Avg.)	28.0 M	2.6 M	12.5 M
Search Time (GPU Hours)	12	9	23
CIFAR-10 Acc. (%)	82.91 ± 5.16	90.57 ± 0.68	95.47 ± 0.60

However, by pairing an intelligent, adaptive GNN controller with a powerful, structured block-based action space, System 3 discovers significantly more efficient architectures than System 1 that achieve higher accuracy. This demonstrates that the combination of a smart agent and an effective action space is extremely important for navigating the trade-offs between model complexity and performance.

3.3 Analysis of Self-Editing Mechanism

To isolate the direct impact of the recursive self-modification component, we conducted an ablation study comparing our full Prometheus System 3 with a version where the controllers self-editing capabilities were disabled. In this ablated version, the GNN-based controller maintained a fixed architecture throughout the entire search.

Table 2 Ablation study on the self-editing mechanism. Results are reported as mean $\pm$ standard deviation over 10 runs.

System Variant	Self-Editing	CIFAR-10 Acc. (%)
System 3 (Ablated)	Disabled	95.10 ± 0.55
System 3 (Full)	Enabled	95.47 ± 0.60

The ablation shows a 0.37% average gain when the controllers self-editing capability is enabled. To determine if this improvement was statistically significant, we performed a paired two-sample t-test on the results from each seed. The analysis confirms that the performance gain from self-editing is statistically significant ($p = 0.012$) at an $\alpha = 0.05$ level. This provides strong evidence that allowing the controller to adapt its own architecture during the search leads to the discovery of superior final network architectures.

3.4 Analysis of Search Space

To analyze the effectiveness of the block-based search space, we conducted an ablation study, comparing the full System 3 with

a variant using the granular search space from System 2.

Table 3 Ablation study comparing a block-based and a granular search space on CIFAR-10. Results are averaged over 10 runs.

System Variant	Search Space	CIFAR-10 Acc. (%)
System 3 (Ablated)	Granular	85.45 ± 0.65
System 3 (Full)	Block-based	95.47 ± 0.60

The 10.02% average difference between the granular and block-based search spaces demonstrates the power of a structured search space. When held to the same standards and procedures as the full System 3, the granular search space fails to perform as well.

3.5 Comparison with Other NAS Methods

CIFAR-10. On the CIFAR-10 benchmark, Prometheus achieves a top-1 accuracy of $95.47\% \pm 0.60\%$ (Table 4). Our system is competitive with its closest architectural antecedent, EAS [13] (95.77%), and outperforms several modern differentiable methods, including DrNAS [26]. Compared to our own implementation of differentiable architecture search, Prometheus outperforms it by an average of 2.59%. Despite operating on a more controlled, block-like search space, Prometheus achieved higher accuracy than the more flexible differentiable NAS. This suggests that a structured search space can be more effective than unrestricted flexibility. Search times were comparable: differentiable NAS took 16 hours on average, while Prometheus required 23 hours.

Table 4 CIFAR-10 accuracy of sequential/controller-based and other modern NAS methods.

Method	Top-1 Acc. (%)
MetaQNN [27]	93.08
NAS-RL [1]	96.35
PNAS [28]	96.60
ENAS [3]	97.11
EAS [13]	95.77
DeepSwarm [29]	88.69
RSPS [30]	84.07 ± 3.61
SETN [31]	87.64 ± 0.00
DrNAS [26]	94.36 ± 0.00
PC-DARTS [32]	93.66 ± 0.17
GDAS [33]	93.61 ± 0.09
First Order Differentiable (Our Impl.)	92.88 ± 0.46
Prometheus (System 3) (Average)	95.47 ± 0.60
Prometheus (System 3) (Best of 10 runs)	96.58

While top-performing methods like ENAS [3] achieve higher raw accuracy, Prometheus demonstrates that recursive self-editing can yield strong results. The novelty of its self-editing system positions it as a successful proof-of-concept, even against established NAS systems.

Street View House Numbers (SVHN). Prometheus was benchmarked on SVHN, with training segmented into three 50-iteration sessions due to Colabs 24-hour limit. Total search time was 49 GPU hours. Table 5 compares Prometheus to recent NAS methods.

Table 5 Test accuracy comparison on SVHN. Prometheuss accuracy is averaged over $n = 10$ runs.

Method	Accuracy (%)
DrNAS [26]	96.30
RSPS [30]	96.17
SETN [31]	96.02
GDAS [33]	95.57
PC-DARTS [32]	95.40
EAS [13]	98.27
ResNet-110 [7]	98.27
Prometheus (System 3)	97.09 ± 0.15

Prometheus surpasses every NAS baseline except EAS, despite using a single self-editing controller and no manual dataset-specific tuning.

Fashion-MNIST. Table 6 compares Prometheus to evolutionary and swarm-based baselines.

Table 6 Fashion-MNIST accuracy comparison. Prometheuss accuracy is averaged over 10 runs.

Method	Accuracy (%)
MO-ResNet [34] (best)	95.91
DeepSwarm [29] (best)	93.56
Christoforidis et al. [35]	93.25
ECToNAS [36]	87.20
Prometheus (System 3)	95.57 ± 0.20

Prometheus outperforms all except MO-ResNet. The latters residual-centric bias and grayscale-specific tuning likely give it an edge on Fashion-MNIST.

CIFAR-100. Table 7 compares Prometheus to established NAS methods on this more challenging dataset.

Prometheus remains competitive on CIFAR-100, validating its robustness and generalization across datasets.

3.6 Meta-Agent Growth

We analyzed the GNN controllers architectural evolution over a representative 150-iteration search run on CIFAR-10. The primary question was whether the controller would meaningfully alter its own structure in response to the searchs difficulty.

Figure 1 illustrates the change in the controllers total parameter count as the search progresses. The controller did not

Table 7 CIFAR-100 accuracy comparison. Prometheuss accuracy is averaged over $n = 10$ runs.

Method	Accuracy (%)
DrNAS [26]	73.51 ± 0.00
GDAS [33]	70.70 ± 0.30
PC-DARTS [32]	66.64 ± 2.34
SETN [31]	59.09 ± 0.24
RSPS [30]	52.31 ± 5.77
Prometheus (System 3)	73.26 ± 0.81

maintain a fixed architecture; instead, it grew significantlyfrom approximately 3,000 parameters to over 24,000, an 8-fold increase in complexity. No pruning operations were conducted during this run. Complexity was valued, as increased capacity often correlated with improved target network performance.

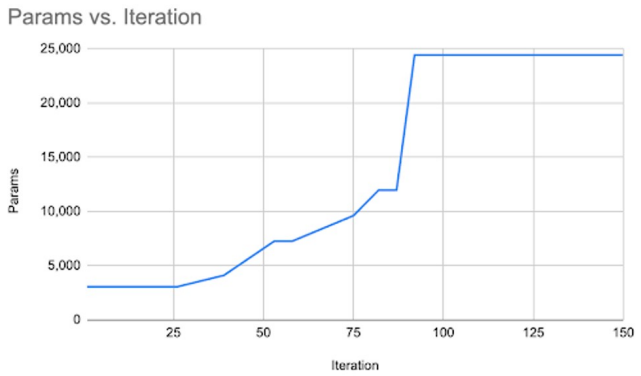

Fig. 1 Parameters of the meta-agent (y-axis) vs. iteration count (x-axis) from a run on CIFAR-10.

4 Discussion

In this paper, we presented Prometheus, a NAS system grounded in the idea of recursive self-improvement. Through three successive variants, we showed that an RL-based controller can modify both a target networks architecture and its own. Our final system, System 3, combines a self-editing GNN controller, a block-based action space, and heuristic-driven adaptation to achieve a competitive 95.47%±0.60% on CIFAR-10, performing well compared to its closest conceptual predecessor, EAS [13], and supporting our approach as an effective search strategy.

Using triggers tied to stagnation and instability, the controllers architecture is edited, intentionally trading short- term policy stability for longer-term capability. This behavior is enabled by function-preserving Net2Net edits that avoid catastrophic forgetting during self-modification. In theory, as the RL agent grows more capable, its edits to itself get better, causing it to grow even more capable, creating an ever-improving loop. We provide evidence that heuristic- triggered, function-preserving self-edits can improve search outcomes on small- to medium-scale vision benchmarks, offering a proof-of-concept for adaptive and autonomous NAS agents.

The progression through the three systems indicates that an extensively pre-designed block-level search (System 1) tends to inflate into inefficient, underperforming models. On the other hand, an overly granular, operation-level search (System 2) produces compact architectures but still struggles to find an optimal architecture. System 3 suggests a balanced path: a GNN controller manipulating basic blocks, guided by explicit complexity penalties and heuristic self-regulation. This is the system that ultimately outperformed the rest.

On the optimum triggers for self-editing, such as the stagnation window or rollback threshold, it is plausible that they would need to be adapted when scaling to more complex domains. A task like ImageNet, with its longer convergence times, might benefit from a different set of triggers to balance exploration and stability. An investigation into how these parameters scale with dataset complexity remains a crucial step toward creating a robust self-improving system.

4.1 Limitations

A key limitation of this study lies in its reliance on heuristic timing for self-editing. Specifically, the controller does not learn when to initiate edits; instead, it follows a set of predefined rules. This was an intentional design decision. As a proof-of-concept, our primary objective was to demonstrate that a controller could successfully modify its own architecture to better search performance, rather than to address the meta-learning challenge of optimizing the timing of such modifications. Employing a fixed trigger simplified the experimental setup and enabled us to more directly attribute performance improvements to the self-editing mechanism itself. Although a learned meta-policy would likely offer superior adaptability, our findings show that even a simple rule-based approach can produce measurable benefits, supporting the viability of heuristic self-regulation as a strategy for improving search quality.

Another limitation is type of evaluation datasets. The current evaluation focuses exclusively on image-based datasets. This leaves open the question of how well Prometheus would perform in other domains such as natural language processing, time series forecasting, or multimodal problems.

4.2 Future Work

Along with increasing the difficulty of the task at hand, future work could involve:

- **Hierarchical RL:** Training a high-level policy to decide

when and how to self-edit, rewarding actions by the long-term gains of the lower-level (target-editing) policy.

- **Utility-Based Pruning:** Inspired by synaptic pruning, replace instability heuristics with a mechanism that learns to remove the least useful components (e.g., neurons with minimal contribution), improving efficiency without blunt penalties.
- **Cross-domain evaluation:** Extending Prometheus to non-vision domains such as text, time series, and audio, in order to validate its generality and adaptability.
- **Learned self-modification strategies:** Replacing fixed heuristics with a learned meta-policy that enables the controller to autonomously decide when and how to alter its own architecture.

Together, these directions point toward the development of a truly autonomous NAS system, and one capable of learning not only architectures but also how to evolve itself over time.

References

1 B. Zoph and Q. V. Le, *Neural architecture search with reinforcement learning*, arXiv preprint arXiv:1611.01578,.

2 H. Liu, K. Simonyan and Y. Yang, *Darts: Differentiable architecture search*, arXiv preprint arXiv:1806.09055,.

3 H. Pham, M. Y. Guan, B. Zoph, Q. V. Le and J. Dean, *Efficient neural architecture search via parameter sharing*, arXiv preprint arXiv:1802.03268,.

4 Y. Li, J. Li, C. Hao, P. Li, J. Xiong and D. Chen, *Extensible and efficient proxy for neural architecture search*, arXiv preprint arXiv:2210.09459,.

5 Z. He, Y. Shu, Z. Dai and B. K. H. Low, *Robustifying and boosting training-free neural architecture search*, arXiv preprint arXiv:2403.07591,.

6 R. S. Sukthanker, A. Zela, B. Staffler, S. Dooley, J. Grabocka and F. Hutter, *Multi-objective differentiable neural architecture search*, arXiv preprint arXiv:2402.18213,.

7 H. Lee, E. Hyung and S. J. Hwang, *Rapid neural architecture search by learning to generate graphs from datasets*, arXiv preprint arXiv:2107.00860, 2021.

8 C. Zhang, M. Ren, R. Urtasun and G. E. Hinton, *Graph hypernetworks for neural architecture search*, arXiv preprint arXiv:1810.05749,.

9 M. Jaderberg, V. Dalibard, S. Osindero, W. M. Czarnecki, J. Donahue, A. Razavi, O. Vinyals, T. Green, I. Dunning, K. Simonyan, C. Fernando and K. Kavukcuoglu, *Population based training of neural networks*, arXiv preprint arXiv:1711.09846,.

10 M. Wichrowska, N. Maheswaranathan, M. W. Hoffman and J. Sohl-Dickstein, *Learned opti- mizers that scale and generalize*, arXiv preprint arXiv:1703.04813,.

11 T.-W. Yang, Y.-H. Chen and V. Sze, *Netadapt: Platform-aware neural network adaptation for mobile applications*, arXiv preprint arXiv:1804.03230, 2018.

12 T.-W. Wei, C.-P. Wang and Y.-Y. Chen, *Network morphism*, arXiv preprint arXiv:1603.01670,.

13 H. Cai, T. Chen, W. Zhang, Y. Yu and J. Wang, *Efficient architecture search by network transformation*, arXiv preprint arXiv:1707.04873,.

14 T. Chen, I. Goodfellow and J. Shlens, *Net2net: Accelerating learning via knowledge transfer*, arXiv preprint arXiv:1511.05641,.

15 K. He, X. Zhang, S. Ren and J. Sun, *Deep residual learning for image recognition*, arXiv preprint arXiv:1512.03385,.

16 C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke and A. Rabinovich, *Going deeper with convolutions*, arXiv preprint arXiv:1409.4842,.

17 J. Hu, L. Shen and G. Sun, *Squeeze-and-excitation networks*, arXiv preprint arXiv:1709.01507,.

18 M. Sandler, A. Howard, M. Zhu, A. Zhmoginov and L.-C. Chen, *Mobilenetv2: Inverted residuals and linear bottlenecks*, arXiv preprint arXiv:1801.04381,.

19 K. Simonyan and A. Zisserman, *Very deep convolutional networks for large-scale image recognition*, arXiv preprint arXiv:1409.1556,.

20 T. N. Kipf and M. Welling, *Semi-supervised classification with graph convolutional networks*, arXiv preprint arXiv:1609.02907,.

21 Y. LeCun, L. Bottou, Y. Bengio and P. Haffner, *Gradient-based learning applied to document recognition*.

22 V. Nair and G. E. Hinton, *Rectified linear units improve restricted boltzmann machines*.

23 A. Krizhevsky and G. Hinton, *Learning multiple layers of features from tiny images*.

24 Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu and A. Y. Ng, *Reading digits in natural images with unsupervised feature learning*.

25 H. Xiao, K. Rasul and R. Vollgraf, *Fashion-mnist: A novel image dataset for benchmarking machine learning algorithms*, arXiv preprint arXiv:1708.07747,.

26 X. Chen, L.-J. Xie, J. Wu and Q. Tian, *Progressive differentiable architecture search: Bridging the depth gap between search and evaluation*, arXiv preprint arXiv:2006.10355,.

27 B. Baker, O. Gupta, N. Naik and R. Raskar, *Designing neural network architectures using reinforcement learning*, arXiv preprint arXiv:1611.02167,.

28 C. Liu, B. Zoph, M. Neumann, J. Shlens, W. Hua, L.-J. Li, L. Fei-Fei, A. Yuille, J. Huang and K. Murphy, *Progressive neural architecture search*, arXiv preprint arXiv:1712.00559,.

29 E. Byla and W. Pang, *Deepswarm: Optimising convolutional neural networks using swarm intelligence*, arXiv preprint arXiv:1905.07350, 2019.

30 L. Li and A. Talwalkar, *Random search and reproducibility for neural architecture search*, arXiv preprint arXiv:1902.07638,.

31 X. Dong and Y. Yang, *One-shot neural architecture search via self-evaluated template network*, arXiv preprint arXiv:1910.05733,.

32 Y. Xu, L. Xie, X. Zhang, X. Chen, G.-J. Qi, Q. Tian and H. Zhang, *Pc-darts: Partial channel connections for memory-efficient differentiable architecture search*, arXiv preprint arXiv:1907.05737,.

33 X. Dong and Y. Yang, *Searching for a robust neural architecture in four gpu hours*, arXiv preprint arXiv:1910.04465,.

34 S. Wang, H. Tang and J. Ouyang, *A neural architecture search method using auxiliary evaluation metrics based on resnet architecture*, arXiv preprint arXiv:2505.01313,.

35 A. Christoforidis, G. Kyriakides and K. Margaritis, *A novel evolutionary algorithm for hierarchical neural architecture search*, arXiv preprint arXiv:2107.08484,.

36 E. J. Schiessler, R. C. Aydin and C. J. Cyron, *Ectonas: Evolutionary cross-topology neural architecture search*, arXiv preprint arXiv:2403.05123,.