Investigating Multi-Objective Optimization in Neural Architecture Search: Insights, Limitations, and Opportunities

Nirosh Sivanesan
Seminar in Applied Optimization
AI in Medicine Master's Program

April 22, 2025

AI-Disclosure Statement. *Selected generative-AI tools—OpenAI ChatGPT and DeepSeek—were used solely to polish wording and to generate LaTeX code for mathematical expressions, thereby streamlining the editorial process. All ideas, analyses, and conclusions originate with the author and were rigorously reviewed for accuracy and coherence. Every AI-suggested passage was cross-checked against primary sources and empirical evidence, then revised where necessary. The manuscript has undergone final human verification to ensure full compliance with academic standards; the author bears complete responsibility for its content and scholarly integrity.*

1 Context and Motivation

Modern neural networks are continuously growing in complexity, driven by the need to achieve higher predictive performance in diverse, real-world applications. However, increased complexity often comes with significant trade-offs in terms of computational cost, latency, energy consumption, and model size. Managing these conflicting requirements manually has become increasingly challenging, motivating research into automated approaches.
Neural Architecture Search (NAS), a prominent sub-field of Automated Machine Learning (AutoML), addresses this challenge by automatically discovering high-performing neural network architectures. Due to the inherently multi-objective nature of this task—where objectives like accuracy, inference latency, and model efficiency often conflict—Multi-Objective Optimization (MOO) methods are crucial. These methods systematically explore trade-offs between conflicting objectives, enabling practitioners to make informed decisions based on specific deployment scenarios.
This seminar project investigates the application and adaptation of MOO techniques within NAS frameworks, focusing primarily on two optimization strategies currently at the forefront of NAS research: evolutionary algorithms and gradient-based approaches. Although Reinforcement Learning (RL) was historically foundational in NAS, recent literature highlights evolutionary and gradient-based methods as more effective and computationally efficient alternatives, as shown in Fig. 1 [11]. Thus, our investigation emphasizes these two approaches, providing deeper insights into their MOO foundations, adaptations, and innovative applications within NAS.

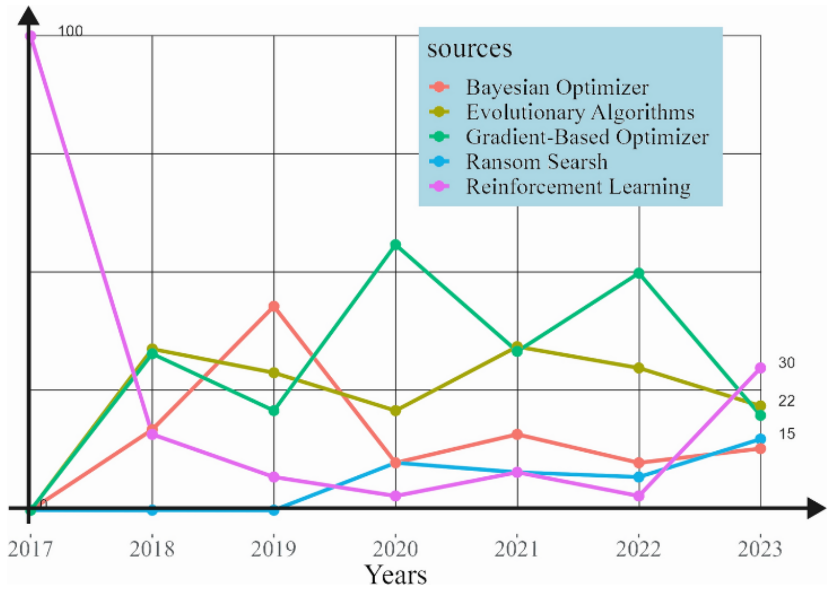

Fig. 8 Percentage of different optimizers used in SSt of NAS algorithms

Figure 1: Popularity trends of NAS search strategies (Credit: [11]).

2 Central Research Questions

Our study aims to provide a clear, structured overview of how Multi-Objective Optimization (MOO) techniques are adapted to address the specific demands of Neural Architecture Search (NAS). Rather than developing novel methods, our goal is to consolidate a comprehensive understanding of current practices, challenges, and advancements within the field.

2.1 Key Challenges in Applying MOO to NAS

Applying MOO techniques to NAS involves overcoming several fundamental challenges:

- **Combinatorial Discreteness:** NAS search spaces are vast and discrete, involving combinatorial choices of operations, hyperparameters, and connectivity. Traditional continuous optimization methods must therefore be specially adapted using discrete encodings, relaxation techniques, or evolutionary mechanisms.
- **Rugged, Multimodal Landscapes:** The performance landscape of NAS is highly multimodal, featuring numerous local optima of comparable quality. Effective search strategies must incorporate diversity-preserving and noise-resistant mechanisms to navigate these complex landscapes without premature convergence.
- **Noisy Evaluations:** NAS evaluations are inherently noisy due to factors like random weight initialization, data augmentation procedures, and hardware performance fluctuations. Robust optimization requires methods such as surrogate modeling, uncertainty-aware rankings, or aggregation strategies to stabilize candidate comparisons.
- **High-Dimensional Objectives:** Real-world NAS applications typically involve multiple conflicting objectives (e.g., accuracy, model size, computational cost, latency on different hardware). Objectives often exhibit correlations, resulting in degenerate Pareto fronts that require sophisticated Pareto-sorting and diversity-maintenance approaches.
- **Prohibitive Computational Costs:** Training a single architecture candidate fully can take substantial GPU resources and computational time (days or even weeks). To manage these computational constraints, MOO techniques often employ multi-fidelity evaluation approaches (e.g., early stopping, successive halving), weight-sharing mechanisms (supernets), and computationally inexpensive performance proxies.

2.2 Scope of This Study

[ON PROGRESS...]
This report specifically reviews two leading NAS optimization strategies—evolutionary algorithms and gradient-based methods—focusing on how they leverage MOO:

- **Evolutionary MOO Approaches:** Evolutionary algorithms (e.g., NSGA-II adapted for discrete spaces) - NSGA-Net [10], NSGA-Net V2 and exploring NSCA if possible.
- **Gradient-Based MOO Approaches:** Pareto Front profiling with MODNAS [12]

1. How they leverage MOO ? 2. How they balance between exploration and exploitation ? 3. Additionally, we provide a comprehensive review of the field including:

- Problem Formulation
- NAS three pillars: Search Space, Search Strategy and Validation Strategy

Through this structured exploration, we intend to equip readers with a solid understanding of how MOO is integrated into NAS, the inherent challenges researchers currently face, and the promising opportunities this intersection presents.

3 Background

3.1 Automated Machine Learning, Hyperparameter Optimization, and Neural Architecture Search — Past, Present, and Future

Automated Machine Learning (AutoML) has emerged as a transformative concept within the broader machine learning community, aspiring to **systematically automate** the construction of high-performance learning pipelines. Among the various branches of AutoML, **Hyperparameter Optimization (HPO)** and **Neural Architecture Search (NAS)** have gained particular prominence. In the following, we briefly review the foundations and distinctions between HPO and NAS, primarily drawing upon the comprehensive survey of Baratchi et al. (2024) [1].

3.1.1 Hyperparameter Optimization (HPO)

Hyperparameter Optimization (HPO) seeks to discover optimal hyperparameter configurations for a given, fixed machine learning algorithm. Examples of hyperparameters include learning rates, regularization coefficients, ensemble sizes, and activation functions, all of which can heavily influence both the training dynamics and the generalization capacity of the resulting model.
Typical HPO methods operate over a search space of hyperparameters that is often low- to moderate-dimensional.

3.1.2 Neural Architecture Search (NAS)

Neural Architecture Search (NAS) extends beyond hyperparameter tuning by **automatically designing** the underlying structure of neural networks. This encompasses decisions regarding layer types (e.g., convolutional layers, transformers), connectivity patterns (e.g., skip connections), and other architectural details.
Traditionally, designing deep neural architectures required significant expertise, intuition, and extensive experimentation. NAS frames network architecture design as an **optimization** problem, typically involving a combinatorial design space.

3.1.3 HPO vs. NAS: A Comparative Summary

Table 1 summarizes key differences between HPO and NAS.

3.2 Multi-Objective Optimization (MOO) Concepts

MOO aims to simultaneously optimize multiple conflicting objectives. Key definitions are:

Definition 1 (Pareto Dominance). Given two candidate solutions $x_1, x_2 \in \Omega$, solution x_2 dominates x_1 if $f_m(x_2) \leq f_m(x_1) \, \forall m$, with at least one strict inequality.

Table 1: Comparison Between HPO and NAS

Aspect	**Hyperparameter Optimization (HPO)**	**Neural Architecture Search (NAS)**
Objective	Optimize hyperparameters of fixed model	Discover optimal neural architectures
Search Space	Learning rate, batch size, regularization terms	Layer types, connections, operations, etc.
Complexity	Moderate (continuous & categorical)	High (large, combinatorial, discrete)
Typical Techniques	Bayesian optimization, random search	RL, evolutionary methods, differentiable methods
Evaluation Cost	Lower; fixed model	High; each architecture requires training

Definition 2 (**Pareto Optimality and Pareto Front**). A solution x^* is Pareto optimal if no other solution dominates it. The set of all Pareto-optimal solutions forms the Pareto set, and their image under objective functions forms the Pareto front.

3.3 Prominent Multi-Objective Algorithms

[Working on...]

- **Non-dominated Sorting Genetic Algorithm (NSGA-II)**: evolutionary method using Pareto-based sorting.
- **Non-Dominated Sorting Crisscross Algorithm (NSCA)**: efficient handling of many-objective problems.
- **Multiple Gradient Descent (MGD)**: gradient-based multi-objective optimization method for deep networks.

3.4 Non-dominated sort genetic algorithm (NSGA)

[Working on...]

3.5 Non-Dominated Sorting Crisscross Algorithm(NSCA)

[Working on...]

3.6 Multiple Gradient Descent

[Working on...]

4 Comprehensive Review and Problem Formulation of NAS

Neural Architecture Search (NAS) is often recognized as an inherently multi-objective optimization (MOO) problem, owing to the fact that neural networks must balance competing requirements in modern deep-learning applications [9]. While *predictive performance* (e.g., accuracy or BLEU score) remains the primary objective, other criteria such as latency, energy consumption, and model size (number of parameters, FLOPs) also substantially affect the practical feasibility of the discovered architectures. Hence, as a multi-objective problem, NAS is concerned with simultaneously optimizing these conflicting objectives and identifying efficient trade-offs, often represented by the Pareto front.

4.1 Multi-Objectives NAS

Most existing NAS approaches incorporate (at least) two major objectives:

- **Predictive Performance:** This typically refers to accuracy or an equivalent metric (error rate, F1-score, perplexity) that quantifies how effectively the architecture performs on a given task [5, 15].
- **Resource Efficiency:** These criteria focus on efficiency and feasibility under real-world deployment constraints [13]. Examples include:
  - *Latency*: forward pass inference time measured on specific target hardware .
  - *Energy Consumption*: total inference energy usage.
  - *Model Complexity*: FLOPs, parameter counts, multiply-add operations.
  - *Memory Footprint*: peak memory usage of the architecture [2].

Because maximizing predictive accuracy can often imply higher computational demands, typical trade-offs include "accuracy *vs.* latency" or "accuracy *vs.* model complexity." In more advanced or real-time edge scenarios (e.g., mobile deployment), energy and memory constraints can form part of a larger objective set, making NAS a many-objective search [13].

4.2 Problem Definition and Relevant Concepts

Neural Architecture Search (NAS) can be viewed as a specialized hyperparameter optimization problem in which the *architectural* hyperparameters (decision variables) of a neural network are optimized. When *multiple* performance metrics need to be optimized simultaneously (e.g., predictive accuracy, model size, and inference latency), NAS becomes a multi-objective optimization (MOO) problem. Formally, and following [9] (see also references therein), a unified MOO formulation for NAS can be written as:

$$\begin{aligned} \min_{x} \quad & \mathbf{F}(x) = \big[\, f_1(x),\ f_2(x),\ \ldots,\ f_M(x) \big]^T, \\ \text{subject to} \quad & x \in \Omega, \\ & w^*(x) = \arg\min_{w} \mathcal{L}_{\text{train}}\big(x, w\big), \end{aligned} \tag{1}$$

where:

- x denotes the *architecture hyperparameters* (decision variables), chosen from a search space Ω.
- $w^*(x)$ is the *optimal weight set* of the chosen architecture x, obtained by minimizing the training loss $\mathcal{L}_{\text{train}}$ over the training set $\mathcal{D}_{\text{train}}$.
- $\mathbf{F}(x) = [f_1(x), f_2(x), \ldots, f_M(x)]^T$ is the vector of M objective functions (*e.g.*, error-related metrics, complexity measures, and hardware-related measures).

In many NAS scenarios, these objectives decompose as follows [9]:

1. **Predictive Error Related:** $f_e(x; w^*)$, measuring, for instance, validation error on a hold-out dataset $\mathcal{D}_{\text{valid}}$.
2. **Complexity Related:** $(f_{c1}(x), \ldots, f_{cK}(x))$, such as FLOPs or total parameter count.
3. **Hardware Related:** $(f_{h1}(x), \ldots, f_{hT}(x))$, such as inference latency or memory usage on a specific device (mobile phone, GPU, *etc.*).

Since training a given architecture x to obtain $w^*(x)$ is itself computationally expensive, large-scale NAS tasks typically become expensive black-box multi-objective optimization problems.
NAS under multiple competing performance criteria naturally induces a multi-objective optimization paradigm. Decision variables x span the neural architecture design space Ω, while each point in that space induces a model $w^*(x)$ after training. The goal is to identify Pareto-optimal trade-offs across the chosen objectives, whether they be predictive performance, complexity, or hardware metrics.

NAS Three Pillars

Neural Architecture Search (NAS) is often conceptualized as involving three foundational components or "pillars" (cf. Elsken et al. 2019b[5]; White et al. 2023[14]). Specifically, NAS solutions must specify: **Search Space (SSp)**, **Search Strategy (SSt)**, **Validation Strategy (VSt)**.

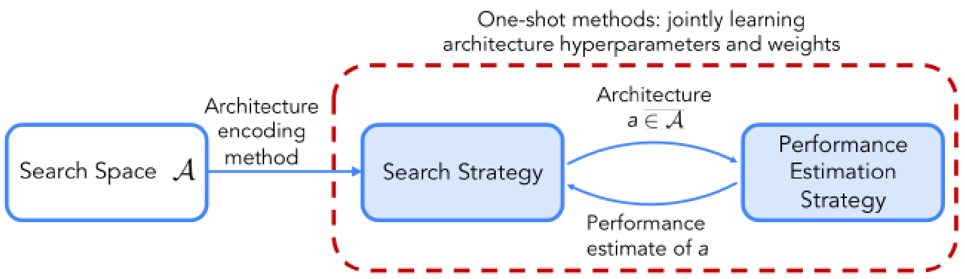
One-shot methods: jointly learning architecture hyperparameters and weights
Search Space $\mathcal{A}$
Architecture encoding method
Search Strategy
Architecture $a \in \mathcal{A}$
Performance Estimation Strategy
Performance estimate of a

Figure 2: Credit: [14]

These three pillars cover (1) *what* is being searched, (2) *how* the search is conducted, and (3) *how* candidate architectures are evaluated.
NAS thus comprises:

1. **Search Space (SSp):** defining the pool of candidate architectures;
2. **Search Strategy (SSt):** selecting an exploration method (e.g. black-box, one-shot, hybrid);

3. **Validation Strategy (VSt):** evaluating and ranking candidates to guide further search.

The efficiency, scalability, and ultimate performance of NAS arise from the interplay between the search space (SSp), search strategy (SSt), and validation strategy (VSt). For a detailed treatment and examples, see Sections 2–5 of (White et al. 2021c) [14]; the following sections are closely modeled on that work.

4.3 Search Space (SSp)

The Search Space is the set of possible neural architectures that NAS can explore. It provides the "building blocks" or "design patterns" for constructing candidate models.

Definition and Scope. A single search space Ω might contain anywhere from a few thousand to over 10^{20} possible architectures, depending on how it is defined. The nature of the search space often determines how difficult the downstream optimization becomes and how easily discovered architectures generalize to real-world tasks.

Terminology To navigate different NAS studies, it's necessary to agree on a few core terms. Here we will use the terminology define in [14] :

- **Operation / Primitive:** The smallest choiceable unit, usually "activation–operation–normalization" (e.g. ReLU–Conv1×1–BatchNorm).
- **Layer:** In simple chains, this is the same as an operation. Sometimes it refers to a known composite, like an inverted-bottleneck block.
- **Block / Module:** A short sequence of layers grouped together in chain-based designs.
- **Cell:** A small directed acyclic graph (DAG) of operations, reused throughout the network.
- **Motif:** A recurring sub-pattern of operations; cells can be viewed as high-level motifs, while single operations form base-level motifs.

Types of Search Spaces. Modern NAS employs four primary search space designs, each balancing flexibility and computational cost:

- **Macro Search Spaces:** These search over whole-network designs or their high-level settings:
  - *Full-architecture search:* Treats the entire model as a single DAG, choosing both operations and their connections. For example, NASBOT (Kandasamy et al., 2018) allows selecting convolutions, pooling layers, or fully connected layers in a DAG of depth ≤ 25.
  - *Macro-hyperparameter search:* Keeps a fixed backbone but optimize high-level parameters like network depth, width, or resolution.

  While macro search spaces offer high expressivity for discovering novel architectures, their flexibility results in computationally expensive searches. The following sections discuss more constrained alternatives.

- **Chain-Structured Search Spaces:** Build a straight line of layers, often based on existing designs (ResNet, MobileNet). Simple and fast, but limited by the fixed sequential order.

- **Cell-Based Search Spaces:** Inspired by repetitive motifs in human-designed networks (e.g., ResNet blocks), cell-based spaces decompose the search into micro-level cells (reusable DAGs) and a fixed macro-level skeleton. This approach reduces complexity while maintaining performance, as cells optimized for small datasets (e.g., CIFAR-10) often transfer to larger ones (e.g., ImageNet) via scaling.
- **Hierarchical Search Spaces:** Hierarchical spaces generalize cell-based designs by nesting motifs at multiple levels:
  - *Two-level hierarchies:* MnasNet [13] combines MobileNetV2 with macro-level hyper-parameters; Chen et al. (2021a) design transformer blocks mixing convolutions and self-attention.
  - *Multi-level hierarchies:* Liu et al. (2018b) propose a 3-level hierarchy (primitives $\rightarrow$ local motifs $\rightarrow$ global motifs), later extended to 4+ levels (Chrostoforidis et al., 2021; Ru et al., 2020b).

Impact on Difficulty and Novelty. A smaller, more constrained space typically shortens the search time and makes discovered solutions more reproducible. However, very narrow or hand-crafted spaces can bias the search toward known design motifs, reducing the chance of discovering truly novel architectures.

4.4 Search Strategy (SSt)

The Search Strategy is the algorithm that iteratively selects or refines candidate architectures within the given space Ω. In other words, it describes *how* NAS explores and exploits the search space to locate high-performing models.

Black-Box Optimization Methods. These strategies treat the objective (e.g., validation error) as a black box:

- **Evolutionary / Genetic Algorithms** keep a population of architectures and repeatedly select, mutate, and replace them.

  Key Idea: Maintain a *population* of architectures, evolving them through selection, crossover, and mutation [4]. *Advantages:* EA can discover diverse solutions and mitigate local minima by exploring multiple "lineages." Weight inheritance (e.g., *Lamarckian Evolution* [4]) further enhances efficiency. *Popularity:* Often used in multi-objective contexts (accuracy *vs.* latency) and can surpass random or naive methods [10].
- **Random Search (RS).** *Key Idea:* Randomly sample architectures from the search space. *Pros & Cons:* While RS is conceptually simple, it can be very slow on large spaces, since it ignores feedback from previous evaluations. It typically serves as a baseline for comparing validation strategies rather than a strong competitor for large-scale NAS tasks.
- **Reinforcement Learning** methods model architecture design choices as actions of a controller that seeks to maximize predicted accuracy [15]

  Key Idea: Model architecture choices as sequential actions of a controller network, with a reward based on validation accuracy or other metrics [15].

Historical Role: Early NAS breakthroughs employed RL but required massive compute resources (e.g., 800 GPU-days in [15]). Later works, such as ENAS, drastically reduced this overhead via shared weights.

Modern Outlook: Although RL's popularity initially declined, improved proxies and partial training have reintroduced it as a feasible approach [**?**].

- **Bayesian Optimization** builds a probabilistic surrogate to guide the next candidate selection [7].

  Key Idea: Constructs a surrogate model (e.g., Gaussian Processes or neural networks) to map architectural hyperparameters to performance, then picks new architectures using an acquisition function [7] . *Reduced Evaluations:* BO focuses on promising subregions of the space, requiring fewer high-fidelity architecture evaluations [**?**]. *Challenges:* Designing surrogates that handle large, graph-like spaces remains non-trivial. Hybrid BO approaches can combine partial training or zero-cost proxies for efficiency [**?**].

- **Monte Carlo Tree Search** expands nodes in a tree where each move corresponds to adding a layer or operation.

One-Shot (Supernet-Based) Approaches. Rather than training each candidate from scratch, one-shot methods train a *supernetwork* (or "hypernetwork") that contains all possible subnetworks in Ω. Subnetworks then "inherit" weights from this single supernet to yield quick performance estimates. Notable subtypes include:

- **Differentiable Search** (e.g., DARTS (Liu et al. 2019c[8])) that relaxes discrete architecture decisions and jointly optimizes architecture and network weights via gradient-based methods.

- **Sampling-Based One-Shot** methods, such as RandomNAS or ProxylessNAS, which sample or mask out candidate subnetworks during supernet training.

  Key Idea: Relax discrete architecture choices into continuous variables to allow gradient-based training of "architecture hyperparameters" alongside network weights [8].

  GPU-Friendliness: GOs take advantage of GPU parallelism for large matrix operations, hence their popularity. However, discretizing or pruning the final architecture can lead to instability if not well-regularized.

  Examples: DARTS [8], ProxylessNAS [2], ...

Multi-Fidelity & Speedup Techniques.

- *Performance prediction* or *"zero-cost" proxies* to estimate candidate performance rapidly without full training (Mellor et al. 2021) [1]

- *Successive Halving/Hyperband* to allocate more training budget only to more promising architectures (Li et al. 2018; Falkner et al. 2018) [1].

- *Network morphisms* to warm-start new architectures from trained "parents," saving training time (Elsken et al. 2017) [1].

4.5 Validation Strategy (VSt)

The Validation Strategy, sometimes called the "performance estimation strategy" or "evaluation strategy," defines how we assess the performance of candidate architectures during (and often after) the search.

Standard Training-Evaluation Pipeline. A typical approach is to split the dataset $\mathcal{D}$ into training and validation subsets, train each candidate architecture on $\mathcal{D}_{\text{train}}$, and use $\mathcal{D}_{\text{val}}$ to measure predictive metrics such as accuracy or loss.

Accelerated Evaluation. As full training can be extremely costly for thousands of architecture candidates, many speedup techniques exist, for instance:

- **Early Stopping** : If an architecture's partial learning curve suggests poor final performance, we stop training.
- **Learning Curve Extrapolation** to predict final performance from only a few epochs of training.
- **One-Shot / Shared Weights** (Pham et al. 2018; Bender et al. 2018)) [1] to avoid training each candidate from scratch.

Feeding Results Back into SSp & SSt. The outcome of each validation pass (the measured performance of a candidate) can be used:

1. To guide the search strategy (e.g., sampling or mutating new candidates in promising directions).
2. To prune unpromising portions of the search space or freeze certain design decisions.

4.6 Challenges in NAS-MOO

A range of difficulties arises from formulating NAS as in Eq. (1):

- **Discrete Search Space:** NAS design variables (e.g., operator type, connection topology, kernel sizes) typically yield a large, combinatorial decision space [8].
- **Non-convex, Multi-modal Fitness Landscape:** Multiple architectures can exhibit very similar (or even the same) performance, introducing multi-modality and making the global search challenging [**?**, **?**].
- **Noisy Objectives:** Evaluating a single architecture x can produce stochastic outcomes due to random weight initialization, data augmentation, hardware load conditions, etc. [9].
- **Many Objectives and Possibly Degenerate Pareto Fronts:** Real scenarios can involve 4-8 objectives, e.g., error, model size, FLOPs, latency for multiple hardware devices [9]. These objectives may be correlated, leading to degenerate (lower-dimensional) Pareto fronts.
- **High Computational Cost:** Full training of each candidate architecture can take days on GPU resources. This overhead is amplified in MOO contexts that require exploring wide ranges of solutions.

Recent works highlight a lack of consistent, standardized benchmarks and general formulations for NAS as MOO, limiting direct comparison of algorithms and reproducibility [9]. While partial solutions such as tabular or surrogate-based NAS benchmarks exist, a general multiobjective suite with hardware metrics remains relatively scarce, necessitating more comprehensive frameworks (e.g., EvoXBench [9]).

Overall, while the synergy between NAS and MOO is highly promising, critical challenges remain: discrete and multi-modal search spaces, noise, many objectives, and high computational overhead. Approaches in evolutionary multi-objective optimization offer a flexible and robust foundation for tackling these concerns, but standardization of formulations, metrics, and benchmarks is essential [9]. Recent endeavors such as EvoXBench [9] provide a critical step toward more rigorous multi-objective NAS development and fair algorithmic comparisons.

5 One-shot Paradigm and Bi-Level Optimization

Recent differentiable NAS methods address the massive, discrete nature of neural architecture search ($\sim 10^{36}$ possible networks) by (i) *weight sharing* in a "supernetwork," and (ii) *continuous relaxation* of originally discrete architectural choices.

Cell-Based DAG Representation Typically, networks are composed of repeated "cells," each treated as a small directed acyclic graph (DAG) with N nodes and edges $\{(i \to j)\}$. Each edge $(i \to j)$ corresponds to a choice from a candidate operation set $\mathcal{O}$, such as $\{3 \times 3\,\mathrm{conv}, \mathrm{skip}, \mathrm{maxpool}\}$.

5.1 Hypernetwork

Let's briefly introduce the concept of *hypernetwork* (often abbreviated *hypernet*) ;

Conceptual Framework A *hypernetwork* is a neural network H designed to dynamically generate parameters (Θ) for a separate target network F. Originally introduced by Ha et al. [6], hypernets build upon concepts from evolutionary computing, such as the HyperNEAT framework, where a small, compact network or *genome* encodes the structure and parameters of a much larger, complex network or *phenotype*.
The key motivation behind hypernets is to address the challenge of directly optimizing neural networks with millions of parameters (Θ). Instead of directly learning all these parameters through standard gradient-based optimization, hypernets shift this complexity into learning a smaller set of parameters (Φ).
By decoupling direct optimization from parameter generation, hypernets transform the training process. They learn a meta-mapping from contextual inputs (such as task identities, embeddings, or noise vectors) to suitable target-network weights. This strategy significantly reduces the effective dimensionality of the search space, making optimization more tractable, efficient, and potentially better suited to scenarios with limited data or high adaptability requirements.

Problem Setup Let $\mathcal{T}$ denote a machine learning task with dataset (X, Y), where $X \in \mathbb{R}^{N \times d}$ is the feature matrix and $Y \in \mathbb{R}^N$ the target vector. Individual samples are denoted (x, y) with $x \in X$, $y \in Y$. A deep neural network (DNN) $F(X; \Theta)$ with parameters Θ solves $\mathcal{T}$ by minimizing the loss $\mathcal{L}(Y, \hat{Y})$, where $\hat{Y} = F(X; \Theta)$ denotes predictions.

Standard DNN Training The conventional learning paradigm solves:

$$\min_{\Theta} \mathcal{L}\big(Y, F(X;\Theta)\big) \tag{2}$$

Through backpropagation, gradients $\nabla_{\Theta}\mathcal{L}$ are computed and used to update Θ via optimization algorithms (e.g., Adam [?]). The final optimized Θ is frozen for inference: $\hat{Y}_{\text{test}} = F(X_{\text{test}};\Theta)$.

Hypernetwork Framework We reparameterize the weight generation process by introducing a *hypernet*. Let $H(C;\Phi)$ be a hypernetwork that maps context vectors C to DNN parameters: Training now solves a *nested* problem:

$$\Theta \;=\; H(C;\Phi), \qquad \hat{Y} \;=\; F\big(X;\Theta\big), \qquad \min_{\Phi}\; \mathcal{L}\big(Y,\hat{Y}\big).$$

The optimization problem becomes:

$$\min_{\Phi} \mathcal{L}\Big(Y, F\big(X; H(C;\Phi)\big)\Big) \tag{3}$$

Only Φ is updated; Θ is *generated on-the-fly*. The figure 3 shows a schematic illustration of that process.

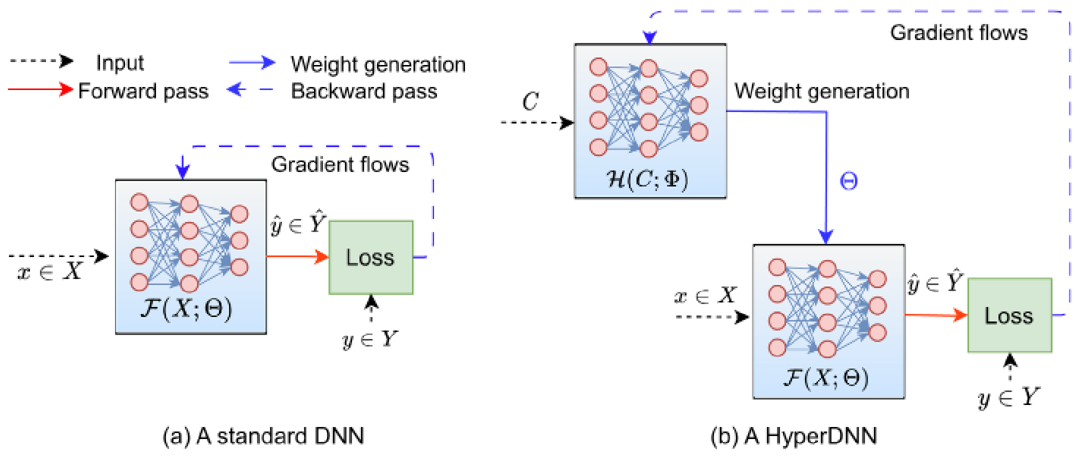

Fig. 1 An overview of the architectures and gradient flows for a standard DNN $\mathcal{F}(X;\Theta)$ and the same DNN implemented with hypernets, referred to as HyperDNN $\mathcal{F}(X;\Theta) = \mathcal{F}(X;\mathcal{H}(C;\Phi))$. For the DNN, gradients flow through the DNN, and DNN weights Θ are learned during training. For the HyperDNN, gradients flow through the hypernet, and hypernet weights Φ are learned during training to produce DNN weights Θ as outputs

Figure 3: Credit: [3]

Relation with NAS. In hypernetwork-based NAS the context vector C *is the architecture*: its graph adjacency and operator labels. Graph HyperNetworks encode this graph and, in a single forward pass, output weights $\Theta = H(C;\Phi)$, eliminating per-candidate training and yielding 10–20$\times$ faster evaluations than conventional NAS loops [3].
Because the same generator parameters Φ serve *all* candidates, knowledge is softly shared across thousands of graphs, accelerating convergence and expanding the searchable design space.

5.2 Weight Sharing Paradigm (Supernetwork One-shot)

REVIEW MATERIAL FROM Brock et al. SMASH and Pham et al. ENAS !!!

5.3 One-Shot Architecture Search and the Role of Weight Sharing

A major limitation in traditional NAS methods is the immense computational burden of individually training numerous candidate architectures from scratch. To mitigate this, recent research introduced *one-shot architecture search*, a methodology where all potential operations and connections are combined into a single, large, over-parameterized network known as the *supernetwork* [**?**]. Specific child architectures are subsequently obtained by selecting subsets of operations from this supernetwork, a paradigm referred to as *weight sharing*. This approach dramatically reduces computational overhead by enabling different architectures to reuse the same learned weights, eliminating the need for redundant training runs.
An intuitive example, illustrated by Bender et al. [**?**], involves a simple scenario where a network position has three candidate operations—a 3×3 convolution, a 5×5 convolution, and a max-pooling layer. Instead of separately training three distinct models, one constructs and trains a single unified model that contains all three operations simultaneously (the *one-shot* model). To evaluate specific architectures, operations are selectively activated (while others are zeroed out), enabling quick assessment of which choices yield the best predictive performance.
In more realistic scenarios, the number of architectural decisions grows rapidly, causing an exponential increase in the search space. However, due to weight sharing, the complexity of the supernetwork grows linearly rather than exponentially, significantly reducing computational demands. Consequently, NAS methods leveraging weight sharing, such as SMASH [**?**] and ENAS [**?**], have proven substantially more efficient compared to conventional NAS approaches.
Yet, the effectiveness of weight sharing is not without skepticism. It is inherently counterintuitive that a single set of learned parameters can adequately serve a diverse set of candidate architectures. Indeed, one-shot models typically function only to efficiently rank candidate architectures; the best architectures identified through this process are usually retrained independently from scratch to achieve optimal performance. Addressing these concerns, approaches like SMASH introduce a hypernetwork to dynamically generate architecture-specific weights [**?**], whereas ENAS iteratively refines both the supernetwork's weights and an auxiliary controller to identify promising subsets of architectures [**?**].
Importantly, Bender et al. [**?**] demonstrated that the effectiveness of one-shot NAS does not strictly depend on sophisticated reinforcement learning (RL) controllers or hypernetworks. Instead, their findings suggest that simple gradient-based optimization of the supernetwork is sufficient to achieve strong results, underscoring the intrinsic capability of weight sharing as the primary driver of efficiency.
The conceptual foundations of one-shot NAS also intersect with ideas from meta-learning, in that training a single model to rapidly adapt to or evaluate many tasks (in this context, architectures) aligns closely with meta-learning's principle of leveraging shared knowledge across related scenarios.

5.4 Continuous Relaxation

Neural Architecture Search (NAS) is inherently combinatorial, requiring the selection of exactly one operator (such as a 3×3 convolution, max pooling, identity, or zero) for each edge in a directed

acyclic graph (DAG). Consequently, traditional search methods, including exhaustive search, reinforcement learning (RL), and evolutionary algorithms, face significant scalability issues due to the combinatorial explosion of possibilities.
First introduced by DARTS [8], Liu & al. addressed this critical bottleneck by introducing a *continuous relaxation* of discrete choices into differentiable parameters, making standard gradient-based optimization techniques applicable and significantly improving scalability and efficiency.
To enable gradient-based optimization of the supernetwork, each discrete architectural decision is replaced by a continuous, differentiable representation. Instead of selecting a single operation, we introduce a real-valued vector of parameters, α, which serves as weights in a continuous mixture of candidate operations. During the training phase, these parameters become learnable via gradient descent, allowing the network to smoothly explore the architectural search space. Ultimately, the final discrete architecture is recovered by selecting the operation corresponding to the highest learned weight for each choice.

Softmax Mixture of Operations Let $\mathcal{O}$ denote the set of candidate operations and $x^{(i)}$ represent the input tensor at node i. For every edge (i, j), we introduce a parameter vector $\boldsymbol{\alpha}^{(i,j)} \in \mathbb{R}^{|\mathcal{O}|}$, converting the discrete operator selection into a differentiable mixture defined by a softmax distribution:

$$\tilde{o}^{(i,j)}(x) \;=\; \sum_{o\in\mathcal{O}} \frac{\exp\big(\alpha_o^{(i,j)}\big)}{\sum_{o'\in\mathcal{O}} \exp\big(\alpha_{o'}^{(i,j)}\big)}\, o(x), \tag{4}$$

As all candidate operations $o(\cdot)$ are executed and weighted smoothly, the entire supernetwork remains fully differentiable with respect to the *continuous architecture parameters*:

$$\boldsymbol{\alpha} \;=\; \big\{\boldsymbol{\alpha}^{(i,j)}\big\}_{(i,j)}.$$

Recovering a Discrete Graph Once the architecture search converges, each edge in the supernetwork is discretized by selecting the operator with the highest learned weight:

$$o^{(i,j)} \;=\; \arg\max_{o\in\mathcal{O}} \alpha_o^{(i,j)},$$

resulting in a discrete architecture ready for final training and deployment.

5.5 Bi-Level Optimization

Besides the architecture parameters $\boldsymbol{\alpha}$, the supernetwork contains standard neural network weights w (such as convolutional kernels) within each operation. DARTS employs a *bi-level optimization* strategy, aiming to discover the architecture parameters that minimize validation loss, while simultaneously optimizing the weights w to minimize training loss. This relationship is formalized by the following nested optimization problem [8]:

$$\begin{aligned} &\min_{\alpha}\; L_{\text{val}}\big(w^*(\alpha), \alpha\big) \\ &\quad\text{subject to}\quad w^*(\alpha) \;=\; \arg\min_{w}\; L_{\text{train}}\big(w, \alpha\big), \end{aligned} \tag{5}$$

where L_{train} and L_{val} are calculated on separate training and validation splits, paralleling the conventional train/selection strategy used in reinforcement learning and evolutionary NAS methods. The optimization problem (5) is solved approximately through alternating gradient steps:

1. **Inner update (network weights)**: Using batches from the training dataset, weights w are updated through standard optimization steps (e.g., SGD or Adam) based on the training loss L_{train}.
2. **Outer update (architecture parameters)**: Using batches from the validation dataset, the architecture parameters $\boldsymbol{\alpha}$ are updated by backpropagating the gradient of the validation loss L_{val}, again using SGD or Adam.

Two variants of this bi-level optimization approach exist:

- The *second-order variant* computes exact bi-level gradients by explicitly differentiating through the inner optimization loop.
- The computationally efficient *first-order variant* approximates the gradient by treating the network weights w as fixed during the outer optimization step.

Summary By combining weight-sharing through a unified *supernetwork* with differentiable architecture parameters $\boldsymbol{\alpha}$, we transform the neural architecture search problem from a computationally intractable combinatorial challenge into a feasible, gradient-based optimization process. This approach reduces computational costs dramatically—by several orders of magnitude—while achieving architectures with competitive performance.

5.6 Architecture Gradient Approximation

Computing the exact gradient of the validation loss with respect to architecture parameters α involves solving a nested optimization problem, which is computationally expensive. To bypass this complexity, we introduce a practical approximation based on a single-step inner optimization, inspired by methods common in meta-learning and hyperparameter optimization.

5.6.1 Proposed Single-Step Approximation

Formally, the true architecture gradient is given by:

$$\nabla_\alpha \mathcal{L}_{\text{val}}(w^*(\alpha), \alpha),$$

where the weights $w^*(\alpha)$ minimize the training loss $\mathcal{L}_{\text{train}}$ exactly:

$$w^*(\alpha) = \arg\min_w \mathcal{L}_{\text{train}}(w, \alpha).$$

However, full convergence is impractical. Thus, we approximate $w^*(\alpha)$ by performing just one gradient descent step starting from the current weights w:

$$w' = w - \xi \nabla_w \mathcal{L}_{\text{train}}(w, \alpha),$$

with $\xi > 0$ as the inner-step learning rate. The architecture gradient then becomes:

$$\nabla_\alpha \mathcal{L}_{\text{val}}(w^*(\alpha), \alpha) \approx \nabla_\alpha \mathcal{L}_{\text{val}}(w', \alpha). \tag{6}$$

Interpretation: This ”one-step look-ahead” method implicitly encodes how slight changes in architecture parameters influence the training dynamics without incurring the cost of full optimization.

5.6.2 Second-Order Gradient Expansion

Applying the chain rule to the approximation yields two distinct terms:

$$\nabla_\alpha \mathcal{L}_{\text{val}}(w', \alpha) = \underbrace{\nabla_\alpha \mathcal{L}_{\text{val}}(w', \alpha)}_{\text{Direct effect}} - \xi \underbrace{\nabla^2_{\alpha,w} \mathcal{L}_{\text{train}}(w, \alpha) \, \nabla_{w'} \mathcal{L}_{\text{val}}(w', \alpha)}_{\text{Indirect effect (second-order)}}. \tag{7}$$

The **first term** captures the direct influence of architecture parameters α on the validation loss. The **second-order term** captures the indirect influence, reflecting how changes in architecture parameters alter the trajectory of the inner optimization (weights w). This Hessian-vector product is computationally intensive if computed naively, with complexity $\mathcal{O}(|\alpha||w|)$.

5.6.3 Reducing Complexity via Finite Difference Approximation

To significantly reduce computational complexity, we approximate the second-order term via finite differences. Define a small scalar perturbation ε and consider two perturbed weight vectors:

$$w_\pm = w \pm \varepsilon \nabla_{w'} \mathcal{L}_{\text{val}}(w', \alpha).$$

Then the Hessian-vector product simplifies to:

$$\nabla^2_{\alpha,w} \mathcal{L}_{\text{train}}(w, \alpha) \, \nabla_{w'} \mathcal{L}_{\text{val}}(w', \alpha) \approx \frac{\nabla_\alpha \mathcal{L}_{\text{train}}(w_+, \alpha) - \nabla_\alpha \mathcal{L}_{\text{train}}(w_-, \alpha)}{2\varepsilon}. \tag{8}$$

This approach reduces complexity dramatically—from $\mathcal{O}(|\alpha||w|)$ to just $\mathcal{O}(|\alpha| + |w|)$—requiring only two forward (for w) and two backward passes (for α).

5.6.4 First-Order vs. Second-Order Approximation

- **First-order ($\xi = 0$):** Ignores indirect effects; computationally efficient but empirically less accurate, as it neglects how weight optimization reacts to architecture changes.
- **Second-order ($\xi > 0$):** Includes indirect curvature information, capturing critical interactions between architecture and training dynamics, thus producing superior empirical results at modest computational overhead.

5.6.5 Practical Algorithm and Empirical Insights

The iterative optimization procedure (Algorithm 1) alternates between inner (weight) and outer (architecture) updates. Although theoretical convergence guarantees remain open, experiments show robust practical convergence [8]. Momentum-based optimization seamlessly integrates into this framework without altering the conceptual derivation.

6 Linear Scalarization

7 Multiple Gradient Descent (MGD)

7.1 Problem Definition

8 Method

[On Process...]
1. Design of Search Space 2. Define the objectives 3. Fitness Evaluator ? 3. Build a one-shot supernet ? 4. Implement algorithm loop

8.1 Design of Search Space

How does the design of a search space differs from the different optimization method ? If we use an evolutionary method versus a gradient based optimizer which use a one-sot architecture space with a hypernetwork ?
1. Define the search space - Design of Search Space.
”A search space defines all attainable solutions to an optimization algorithm. In general, every search space follows the schema outline in algorithm 1 from supplmentary materials.
Designing a good search space for one-shot architecture search is a challenging problem, as it requires us to balance a number of competing requirements.
Search Space Size ?
What are the limitations ?
github pages of the main papers studied :
Gradient-based optimizer : https://github.com/automl/modnas
Evolutionary algorithm : https://github.com/EMI-Group/EvoXBench

References

[1] Mitra Baratchi, Can Wang, Steffen Limmer, Jan N. van Rijn, Holger Hoos, Thomas Bäck, and Markus Olhofer. Automated machine learning: Past, present and future. *Artificial Intelligence Review*, 57(122), 2024.

[2] Han Cai, Ligeng Zhu, and Song Han. Proxylessnas: Direct neural architecture search on target task and hardware. *arXiv preprint arXiv:1812.00332*, 2018.

[3] Vinod Kumar Chauhan, Jiandong Zhou, Ping Lu, Soheila Molaei, and David A Clifton. A brief review of hypernetworks in deep learning. *Artificial Intelligence Review*, 57(9):250, 2024.

[4] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Efficient multi-objective neural architecture search via lamarckian evolution. *arXiv preprint arXiv:1804.09081*, 2018.

[5] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. *Journal of Machine Learning Research*, 20(55):1–21, 2019.

[6] David Ha, Andrew Dai, and Quoc V Le. Hypernetworks. *arXiv preprint arXiv:1609.09106*, 2016.

[7] Kirthevasan Kandasamy, Willie Neiswanger, Jeff Schneider, Barnabas Poczos, and Eric P Xing. Neural architecture search with bayesian optimisation and optimal transport. *Advances in neural information processing systems*, 31, 2018.

[8] Hanxiao Liu, Karen Simonyan, and Yiming Yang. Darts: Differentiable architecture search. *International Conference on Learning Representations (ICLR)*, 2019.

[9] Zhichao Lu, Ran Cheng, Yaochu Jin, Kay Chen Tan, and Kalyanmoy Deb. Neural architecture search as multiobjective optimization benchmarks: Problem formulation and performance assessment. *IEEE transactions on evolutionary computation*, 28(2):323–337, 2023.

[10] Zhichao Lu, Ian Whalen, Vishnu Boddeti, Yashesh Dhebar, Kalyanmoy Deb, Erik Goodman, and Wolfgang Banzhaf. Nsga-net: A multi-objective genetic algorithm for neural architecture search. 2018.

[11] Sasan Salmani Pour Avval, Nathan D Eskue, Roger M Groves, and Vahid Yaghoubi. Systematic review on neural architecture search. *Artificial Intelligence Review*, 58(3):73, 2025.

[12] Rhea Sanjay Sukthanker, Arber Zela, Benedikt Staffler, Samuel Dooley, Josif Grabocka, and Frank Hutter. Multi-objective differentiable neural architecture search. In *Proceedings of the International Conference on Learning Representations (ICLR)*. ICLR, 2025.

[13] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V Le. Mnasnet: Platform-aware neural architecture search for mobile. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 2820–2828, 2019.

[14] Colin White, Mahmoud Safari, Rhea Sukthanker, Binxin Ru, Thomas Elsken, Arber Zela, Debadeepta Dey, and Frank Hutter. Neural architecture search: Insights from 1000 papers. *arXiv preprint arXiv:2301.08727*, 2023.

[15] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.