CONSTRAINT-GUIDED HARDWARE-AWARE NAS THROUGH GRADIENT MODIFICATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Neural Architecture Search (NAS), particularly gradient-based techniques, has proven highly effective in automating the design of neural networks. Recent work has extended NAS to hardware-aware settings, aiming to discover architectures that are both accurate and computationally efficient. Many existing methods integrate hardware metrics into the optimization objective as regularization terms, which introduces differentiability requirements and hyperparameter tuning challenges. This can either result in overly penalizing resource-intensive architectures or architectures failing to meet the hardware constraints of the target device. To address these challenges, we propose CONNAS, a novel gradient-based NAS framework that enforces hardware constraints directly through gradient modification. This approach eliminates the need for differentiable hardware metrics and regularization weights. The novelty in CONNAS lies in modifying gradients with respect to architectural choices, steering the search away from infeasible architectures while ensuring constraint satisfaction. Evaluations on the NATS-Bench benchmark demonstrate that CONNAS consistently discovers architectures that meet the imposed hardware constraints while achieving performance within just 0.14% of the optimal feasible architecture. Additionally, in a practical deployment scenario, CONNAS outperforms handcrafted architectures by up to 1.55% in accuracy under tight hardware budgets.

1 INTRODUCTION

Deep neural networks have proven to be very successful in numerous applications ranging from image recognition (Krizhevsky et al., 2017) and speech recognition (Hinton et al., 2012) to time series segmentation (Lea et al., 2017). More recently, the use of deep neural networks on constrained hardware has gained significant interest in the context of machine learning on the edge (EdgeML), where resource-constrained devices such as mobile devices, embedded computers, and microcontrollers are used to perform inference tasks. These devices often have limited computational power, memory, and energy resources. As a result, it is crucial to design efficient neural network architectures that can operate effectively within these hardware constraints.

A key challenge in EdgeML is designing neural network architectures that meet hardware constraints while maintaining high performance. In recent years, Neural Architecture Search (NAS) was introduced to automate the design of neural networks, and to potentially find novel architectures that outperform manually designed models by experts (Elsken et al., 2019b). In NAS, a search algorithm is used to explore a search space of possible neural network architectures, aiming to find the best-performing architecture for a given task. Models originating from NAS techniques have already outperformed human-designed models on tasks such as image classification (Real et al., 2017; Zoph et al., 2018) and semantic segmentation (Chen et al., 2018). Based on the type of search algorithm used, NAS methods can be categorized into four groups: *(i)* Reinforcement Learning (RL) (Zoph & Le, 2017; Zoph et al., 2018; Baker et al., 2017), *(ii)* Evolutionary Algorithms (EA) (Real et al., 2017; Salimans et al., 2017), *(iii)* Bayesian Optimization (White et al., 2021), *(iv)* and Gradient-based methods (Liu et al., 2019). The latter has gained significant popularity in recent years due to its efficiency and ability to scale to large search spaces compared to the other methods.

Recently, the use of NAS for designing more efficient neural network architectures has been increasingly explored. Prior work, specifically on gradient-based NAS, often integrates hardware

metrics as regularization terms into the optimization function (Cai et al., 2019; Wu et al., 2019; Wan et al., 2020). While this can help guide the search toward hardware-friendly architectures, these approaches often face several challenges. First, to be compatible with gradient-based optimization, the hardware metrics must be differentiable. Second, each regularization term requires appropriate weighting, introducing additional hyperparameters that demand careful tuning. Improper weighting can either overly penalize resource-intensive architectures (resulting to simpler models with suboptimal performance) or fail to enforce the hardware constraints (yielding architectures unsuitable for deployment on the target device). Consequently, these methods often require multiple search runs with varying weights assigned to the hardware-related terms until a feasible architecture is found, a process that can be both tedious and time-consuming.

To overcome these limitations, we introduce CONNAS, a novel hardware-aware, gradient-based NAS algorithm. Our work makes the following key contributions:

- **Direct constraint enforcement:** Unlike prior approaches that incorporate hardware metrics as regularization terms in the loss function, CONNAS modifies gradients with respect to architectural choices to directly enforce hardware constraints, effectively steering the search away from infeasible architectures.
- **No need for differentiable hardware metrics:** Our method eliminates the requirement for differentiable hardware metrics and the associated techniques needed to enforce differentiability.
- **Explicit constraint specification:** CONNAS allows for the explicit definition of hardware constraints without relying on hyperparameter tuning to balance the importance of hardware metrics.

We evaluate CONNAS on the NATS-Bench benchmark (Dong et al., 2021), demonstrating its ability to systematically find architectures that satisfy various hardware constraints while achieving performance close to the optimal architecture available within the search space, reaching as little as a 0.14% difference in accuracy. Our experiments show that CONNAS significantly outperforms existing loss-based methods, which often struggle to find feasible architectures under strict hardware constraints. Additionally, we validate CONNAS in a practical use case, where it successfully discovers high-performing architectures under strict hardware constraints (no more than a model size of 64kB and 18kB memory usage), consistently outperforming the best handcrafted architectures by up to 1.55% accuracy.

2 RELATED WORK

Hardware-aware NAS. Over the years, the focus of NAS has shifted from discovering top-performing novel architectures to emphasizing computational efficiency. The design of neural networks has increasingly been guided by hardware-aware considerations, particularly in the context of deployment on resource-constrained devices. Early works primarily focused on optimizing hardware metrics which could be estimated based on the type of operations found in the model, such as the number of Floating Point Operations (FLOPs) (Xie et al., 2019; Zhou et al., 2018). However, other works have proposed to rely on real hardware metrics for more representative evaluation, which are generally obtained through look-up tables or regressors trained on real on-device benchmarks of various neural operations (Cai et al., 2019; Wu et al., 2019; Wan et al., 2020; Tan et al., 2019). Additionally, other contributions have a more specific focus on NAS techniques and search spaces tailored for ultra-constrained devices, such as microcontrollers (Lin et al., 2020; Liberis et al., 2021).

The most straightforward way to regularize hardware-related objectives is by incorporating hardware metrics as additional terms in the optimization function (Xie et al., 2019; Cai et al., 2019; Wu et al., 2019; Dong & Yang, 2019a; Wan et al., 2020; Bender et al., 2020). A weighting factor is then introduced to adjust the relative importance of these hardware metrics in comparison to the task-specific loss, effectively determining whether more constrained architectures should be favored. However, this fixed approach does not offer a direct mechanism for enforcing hardware constraints. Instead, it merely guides the search toward architectures with more favorable hardware metric values (typically simpler architectures with limited representational capacity) regardless of whether the current architecture actually satisfies the imposed constraints. In response, Tan et al. (2019); Zhou et al. (2018); Dong & Yang (2019a); Bender et al. (2020) propose dynamic optimization functions where hardware-related terms are adjusted depending on whether the current architecture meets the

hardware constraints. The rationale behind this approach is to find architectures that are close to the constraint boundary, which makes it possible to explore more complex architectures that still satisfy the constraints. This is in contrast to methods with a fixed loss term, which may limit the search to simpler architectures. Meanwhile, other techniques (Elsken et al., 2019a; Liberis et al., 2021), typically non-gradient-based methods, use a selection procedure based on hardware metric values, selecting only architectures that satisfy hardware constraints for further optimization.

Gradient-based NAS. Gradient-based NAS techniques, originally proposed by DARTS (Liu et al., 2019), have been used by recent works to efficiently search for neural network architectures. In gradient-based NAS, all possible architectures are represented within an over-parameterized network, where each candidate path is associated with a continuous architecture weight α. This formulation leads to a bi-level optimization problem, in which both the architecture α and its weights w are jointly optimized using gradient descent. Traditionally, the loss function would be defined as the task-specific loss, such as cross-entropy for classification tasks. However, in hardware-aware NAS, the loss function is augmented with regularization terms to account for hardware metrics.

While hardware metrics for individual architectures are not differentiable, state-of-the-art techniques ensure that these metrics can be fully factorized over the architecture parameters. This enables expressing hardware metrics as a weighted sum of contributions from each candidate operation in the search space, thereby making them differentiable with respect to the architecture parameters. However, this approach requires knowledge of each operation's contribution to the overall hardware metric, which becomes computationally intractable in large search spaces. To address this, Xie et al. (2019) propose approximating the hardware-related terms using Monte Carlo estimation. Nevertheless, if the sampling distribution is poorly calibrated, the resulting estimates may be inaccurate.

Constraint-aware Training. The authors of Constraint Guided Gradient Descent (CGGD) (Van Baelen & Karsmakers, 2023) investigated how background knowledge can be enforced during neural network training through inequality constraints. Rather than incorporating these constraints as penalty terms in the loss function, they proposed a novel approach that enforces them by directly modifying the gradient updates of the weights, guiding the optimization process toward a set of weights that satisfy the specified constraints. In this work, we explore how this approach can be leveraged to directly enforce hardware constraints during the search process.

3 METHOD

3.1 PROBLEM FORMULATION

The search space S is modeled as an over-parameterized neural network that encompasses all possible architectures. This is achieved by defining, for each layer in the network, a set of candidate operations (e.g., convolutions with varying kernel sizes) that can be used to construct a neural network. Each candidate operation is associated with a continuous architecture weight α, which enables the use of gradient descent to jointly optimize both operation weights w and the architecture weights α. A final architecture A can then be derived by selecting a subset of candidate operations based on the learned architecture weights α. More formally, let the over-parameterized network S be defined as a directed acyclic graph (DAG) with L edges where each edge e_l is associated with a set of candidate operations $\mathcal{O}_l = \{o_{l,1}, \dots o_{l,N}\}$ with corresponding architecture weights $\boldsymbol{\alpha_l} = [\alpha_{l,1}, \dots, \alpha_{l,N}]$ and operation weights $\boldsymbol{w_l} = [w_{l,1}, \dots, w_{l,N}]$. An architecture $A \in S$ is derived by selecting, for each edge e_l, the operation with the highest architecture weight, i.e., $A = (o_{1,k_1}, \dots, o_{L,k_L})$ where $k_l = \arg\max_k \alpha_{l,k}$. Let c_k denote the k-th hardware metric function (e.g., number of parameters), and b_k its upper bound. The set of hardware constraints is defined as $\mathcal{C} = \{ (c_1, b_1), \dots, (c_M, b_M)\}$, where each constraint is expressed as $c_k(A) \leq b_k$. The search process can then be formulated as follows:

$$\arg\min_{w,\alpha} \mathcal{L}_{\text{task}}(w, \alpha) \quad \text{s.t.} \quad c_k(A) \leq b_k, \quad k = 1, \dots, M \tag{1}$$

In other words, the goal is to find architecture weights α that yield an architecture $A(\alpha)$ minimizing the task-specific loss $\mathcal{L}_{\text{task}}$, while ensuring that the resulting architecture lies within the feasibility region FR, defined as the subset of architectures that satisfy all hardware constraints in $\mathcal{C}$. Formally, we aim to find $A(\alpha) \in FR$, or at least ensure that the sampled architectures A^k generated during optimization converge to FR, i.e., $\lim_{k\to\infty} d(A^k, FR) = 0$.

The value for a hardware metric is obtained using $c_k(A)$, where c_k may be implemented as a look-up table or a learned regressor. Unlike prior work on hardware-aware, gradient-based NAS (Cai et al., 2019; Wu et al., 2019; Wan et al., 2020; Dong & Yang, 2019a), we do not impose that the hardware metrics are made differentiable with respect to the architecture weights α.

3.2 Gradient Update Scheme

We adopt the gradient update scheme as proposed in CGGD (Van Baelen & Karsmakers, 2023) to enforce the hardware constraints $c_k(A) \leq b_k,\ k = 1, \ldots, M$. However, unlike CGGD, which aims to enforce constraints on the model's output predictions, we use this approach to manipulate the architecture weights α to steer the search process towards feasible architectures A. Specifically, each architecture weight $\alpha_{l,j}$ is updated as follows:

$$\alpha_{l,j} \leftarrow \alpha_{l,j} - \eta_\alpha \cdot (\nabla_{\alpha_{l,j}} \mathcal{L}_{\text{task}}(w, \alpha) + R \cdot dir_{\mathcal{C}}(\alpha_{l,j}) \cdot \max\left\{\left\|\nabla_{\alpha_{l,j}} \mathcal{L}_{\text{task}}(w, \alpha)\right\|, \epsilon\right\}) \tag{2}$$

Here, R is a rescale factor that determines the strength of the constraints enforcement, and $\boldsymbol{dir_{\mathcal{C}}(\alpha_l)} = [dir_{\mathcal{C}}(\alpha_{l,1}), \ldots, dir_{\mathcal{C}}(\alpha_{l,N})]$ is a unit vector indicating the direction in which the architecture weights should be modified to satisfy the constraints. An arbitrarily small constant ϵ is added to allow adjustments even when the loss gradient is zero.

3.3 Gradient Direction

The choice of R and $\boldsymbol{dir_{\mathcal{C}}(\alpha_l)}$ is crucial, as it determines whether the architecture weights are modified in a way that effectively reduces the hardware constraint violations, and ultimately leads to convergence to a feasible architecture. Van Baelen & Karsmakers (2023) proved that convergence towards the FR is guaranteed if $R > 1$ and $\boldsymbol{dir_{\mathcal{C}}(\alpha_l)}$ is chosen to be the shortest path with respect to the Euclidean distance to the feasibility region.

The authors have shown that the performance is not highly sensitive to the exact value of R, as long as it is strictly larger than 1. In other words, there is no need for additional hyperparameter tuning, unlike prior work that relies on weighted regularization terms. To compute the direction of the shortest path to FR, we first evaluate the hardware metric values of all candidate operations $\mathcal{O}_l$ under the current architecture configuration. For each edge e_l, we construct a set of candidate architectures $\mathcal{A}_l = \{A_{l,1}, \ldots, A_{l,N}\}$ by replacing the l-th operation with $o_{l,j}$ while keeping the rest fixed:

$$A_{l,j} = (o_{1,k_1}, \ldots, o_{l,j}, \ldots, o_{L,k_L}) \tag{3}$$

We then evaluate $c_k(A_{l,j})$ for each candidate and constraint $(c_k, b_k) \in \mathcal{C}$ in order to determine $\boldsymbol{dir_c^k(\alpha_l)}$, which is the direction to the FR for constraint $c_k(A_{l,j}) \leq b_k$. Based on whether the candidate architectures $\mathcal{A}_l$ satisfy the hardware constraint, we distinguish three cases:

1. All candidates on e_l satisfy the hardware constraint ($\forall j \in \{1, \ldots, N\},\ c_k(A_{l,j}) \leq b_k$). In this case, we set $\boldsymbol{dir_c^k(\alpha_l)} = 0$ for all candidates, as no modification is needed to satisfy the constraint for this edge.

2. Some candidates on e_l violate the hardware constraint ($\exists j,\ c_k(A_{l,j}) > b_k$). In this situation, the direction is computed by aggregating the unit vectors that point away from the candidates that violate the constraint, while pointing towards those that satisfy it. More specifically, we first identify the set of candidates that satisfy the constraint:

$$\mathcal{F} = \{j \mid c_k(A_{l,j}) \leq b_k\} \tag{4}$$

For each pair (j, m) where $j \in \mathcal{F}$ and $m \notin \mathcal{F}$, we compute the unit vector that points from candidate m to candidate j, as shown in Figure 1a:

$$\boldsymbol{u_{j,m}} = [\underbrace{0, \ldots, 0}_{j-1}, 1, \underbrace{0, \ldots, 0}_{m-j-1}, -1, \underbrace{0, \ldots, 0}_{N-m}]/\sqrt{2} \tag{5}$$

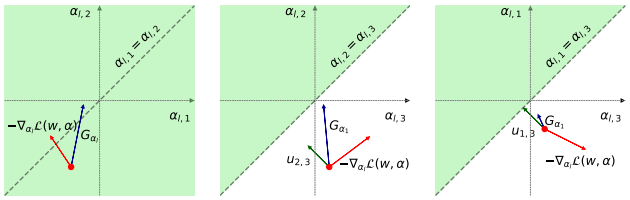
Then, the final direction $\boldsymbol{dir_c^k(\alpha_l)}$ is determined by summing these unit vectors and normalizing it back to a unit vector:

$$\boldsymbol{dir_c^k(\alpha_l)} = \sum_{j \in \mathcal{F}} \sum_{m \notin \mathcal{F}} \boldsymbol{u_{j,m}} \Big/ \left\| \sum_{j \in \mathcal{F}} \sum_{m \notin \mathcal{F}} \boldsymbol{u_{j,m}} \right\| \tag{6}$$

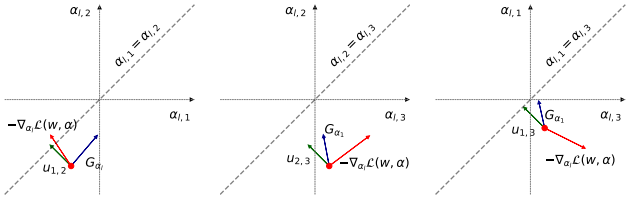
3. No candidate on e_l satisfies the hardware constraint ($\forall j \in \{1, \ldots, N\},\ c_k(A_{l,j}) > b_k$). In this case, the direction is computed in such that candidates with higher hardware metric values are

penalized while those with lower values are rewarded. First, the candidates are ranked based on their hardware metric values. Then, for each ranked candidate, we compute unit vectors $\boldsymbol{u_{j,m}}$ (as defined in Equation 5) that point from candidates with higher hardware metric values to those with lower ones. Specifically:

- First, we compute the unit vectors between the candidate with the highest hardware metric value and all other candidates.
- Then, we compute the unit vectors between the two candidates with the highest hardware metric values and all others.
- This process is repeated until all candidates, except the one with the lowest hardware metric value, have been considered, as shown in Figure 1b. Finally, the unit vectors are aggregated in the same manner as in Equation 6.



(a) Both $A_{l,1}$ and $A_{l,2}$ satisfy the constraint, $A_{l,3}$ violates it, so the resulting gradient G_{α_1} is decreased for $\alpha_{l,3}$ and increased for $\alpha_{l,1}$ and $\alpha_{l,2}$.



(b) No candidate for e_j satisfies the constraint, so the resulting gradient G_{α_1} is increased / decreased based on the ranked hardware metric values of each candidate.

Figure 1: The architecture space for a single edge e_l with three candidate operations parameterized by $\alpha_{l,1}, \alpha_{l,2}$, and $\alpha_{l,3}$ where $c_k(A_{l,1}) < c_k(A_{l,2}) < c_k(A_{l,3})$. The shaded area represents the feasibility region FR for c_k. $\boldsymbol{G_{\alpha_l}}$ is the gradient after modification.

3.4 Constraint-guided Hardware-aware NAS

We refer to Algorithm 1 for the modified training procedure. The optimization of operation w and architecture α weights is performed in an interleaved manner, as proposed in prior work (Cai et al., 2019; Wu et al., 2019; Wan et al., 2020; Dong & Yang, 2019a;b). After optimization of w, the gradient $\nabla_\alpha \mathcal{L}_{\text{task}}(w, \alpha)$ is computed, but instead of immediately updating α, it is stored for later modification. Then, for each hardware metric c_k and its associated constraint $c_k(A) \leq b_k$, the current architecture A is checked whether it violates the constraint. If so, the direction for modification $\boldsymbol{dir_c^k(\alpha)}$ is computed as described in Section 3.3. All directions are then summed and normalized to a unit vector to obtain $\boldsymbol{dir_{\mathcal{C}}(\alpha)}$. Finally, the stored gradient is modified, as explained in Section 3.2, and used to update the architecture weights α.

4 Experiments and Results

We evaluate the effectiveness of ConNAS on two benchmark tasks: the NATS-Bench Topology and Size Search Spaces (Section 4.1). In addition, we apply our method to a practical use case involving edge-based condition monitoring of induction motors, where the goal is to discover architectures suitable for deployment across diverse hardware configurations (Section 4.2).

Algorithm 1 Training procedure under hardware constraints

Input: Search space S, current architecture A derived from α, hardware constraints $\mathcal{C}$, rescale factor R, epsilon ϵ, learning rates η_w, η_α, loss function $\mathcal{L}_{\text{task}}$, batch of data (X, y)
$w \leftarrow w - \eta_w \cdot \nabla_w \mathcal{L}_{\text{task}}(w, \alpha)$ {Update operation weights}
$\boldsymbol{G_\alpha} = \nabla_\alpha \mathcal{L}_{\text{task}}(w, \alpha)$ {Store gradients w.r.t. architecture weights}
$dirs_c = [\,]$ {For each unsatisfied hardware constraint, compute direction to feasibility region}
for $(c_k, b_k) \in \mathcal{C}$ **do**
  if $c_k(A) > b_k$ **then**
    Append $calculate\_gradient\_direction(A, c_k)$ to $dirs_c$
  end if
end for
$\boldsymbol{dir_C(\alpha)} = \sum_{\boldsymbol{dir_c^k} \in dirs_c} \boldsymbol{dir_c^k} / \left\| \sum_{\boldsymbol{dir_c^k} \in dirs_c} \boldsymbol{dir_c^k} \right\|$ {Aggregate directions into single unit vector}
$\boldsymbol{G_\alpha} \leftarrow \boldsymbol{G_\alpha} + R \cdot \boldsymbol{dir_C(\alpha)} \cdot \max\{\|\boldsymbol{G_\alpha}\|, \epsilon\}$ {Modify gradient toward the feasibility region}
$\alpha \leftarrow \alpha - \eta_\alpha \boldsymbol{G_\alpha}$ {Update architecture weights}

4.1 Experiments on NATS-Bench

NATS-Bench, proposed by Dong et al. (2021), is a unified benchmark for NAS designed for image classification tasks, including two search spaces: the Topology Search Space (NATS-Bench-TS) and the Size Search Space (NATS-Bench-SS). NATS-Bench-TS is a cell-based search space, which contains 15,625 unique architectures, while NATS-Bench-SS is a layer-wise search space, consisting of 32,768 unique architectures. We refer to Dong et al. (2021) for a detailed description of the search spaces. A key advantage of NATS-Bench over other NAS benchmarks is that it provides test accuracies for all architectures across both search spaces on CIFAR-10, CIFAR-100, and ImageNet16-120. This enables rapid evaluation without the need for retraining. Additionally, we evaluate all architectures using three commonly used hardware metrics relevant to edge devices: number of parameters, number of FLOPs, and peak memory usage. A detailed explanation of how the hardware metrics are computed is provided in Appendix A. Importantly, our approach is not limited to these metrics and can be integrated with any other hardware metric, regardless of how it is retrieved or computed.

We use a Gumbel-Softmax (Jang et al., 2017), as proposed in Dong & Yang (2019a); Wu et al. (2019); Wan et al. (2020), to relax the categorical distribution of the candidate operations $\mathcal{O}_l$. At each edge e_l, the output y_l is computed as a weighted sum of the outputs produced by all candidate operations:

$$\boldsymbol{y_l} = \sum_j^N \frac{\exp((\log(\alpha_{l,j}) + g_{l,j})/\tau)}{\sum_j^N \exp((\log(\alpha_{l,j}) + g_{l,j})/\tau)} \cdot o_{l,j}(\boldsymbol{x_l}) \tag{7}$$

where $g_{l,j} \sim \text{Gumbel}(0, 1)$ and temperature τ controls the smoothness of the distribution. A higher temperature is used at the beginning of the search to encourage exploration, making the distribution closer to uniform. As the search progresses, the temperature is gradually lowered, making the distribution sharper and closer to an $argmax$, which promotes exploitation. It is worth noting that ConNAS is agnostic to the specific relaxation method used. Other approaches, such as proposed by DARTS (Liu et al., 2019) or ProxylessNAS (Cai et al., 2019), although not tested here, could also be applied.

We run ConNAS on each search space and corresponding set of hardware constraints for a total of 150 epochs. During the first 100 epochs, the temperature τ is linearly annealed from 10 to 0.1. The rescale factor R is set to 1.2, though additional experiments show that the performance is not very sensitive to this choice (see Appendix B). Among the final 50 epochs, we select the architecture with the lowest validation loss that satisfies the hardware constraints. Finally, the performance of the selected architecture is obtained from the test accuracies provided by NATS-Bench. Each experiment is repeated 5 times, as in Dong et al. (2021). The mean and standard deviation of the results are reported in Table 4.

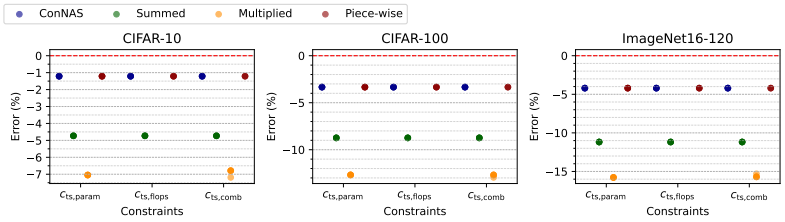
We further compare ConNAS against several regularization-based baselines that incorporate hardware constraints into the loss function, following approaches proposed in prior work on hardware-aware, gradient-based NAS (see Table 1). These strategies are based on the approaches used by ProxylessNAS (Cai et al., 2019), FBNet (Wu et al., 2019), FBNetV2 (Wan et al., 2020), and TAS (Dong & Yang, 2019a). We set the weighting factor $\lambda_{\text{hardware}} = 1$ for all hardware metrics, however, we refer to Appendix C for additional experiments with different weighting factors. We also adopt the

Table 1: **Overview of loss-based baselines used for comparison.** Here, $c'_k(\alpha)$ is the min-max normalization of $c_k(\alpha)$. The term b_k represents the upper bound associated with c_k. $\lambda_{\text{hardware}}$ is a weighting factor which is set to 1 in our experiments.

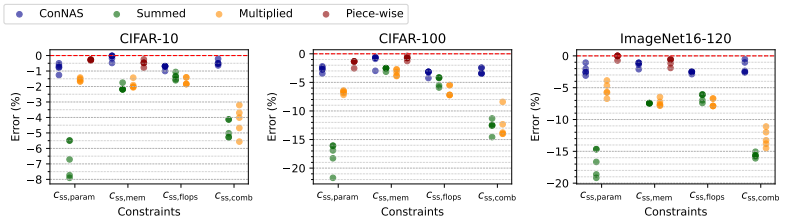
Name	Loss Function	Related Gradient-based NAS Algorithm
Summed	$\mathcal{L}_{\text{task}}(w,\alpha) + (\lambda_{\text{hardware}}/M) \cdot \sum_{k=1}^{M} c'_k(\alpha)$	ProxylessNAS (Cai et al., 2019)
Multiplied	$\mathcal{L}_{\text{task}}(w,\alpha) \cdot \prod_{k=1}^{M} c'_k(\alpha)^{\lambda_{\text{hardware}}}$	FBNet (Wu et al., 2019), FBNetV2 (Wan et al., 2020)
Piece-wise	$\mathcal{L}_{\text{task}}(w,\alpha) + (\lambda_{\text{hardware}}/M) \cdot \sum_{k=1}^{M} h_j(\alpha)$ where $h_j(\alpha) = \begin{cases} c'_k(\alpha), & \text{if } c_k(\alpha) > b_k \\ -c'_k(\alpha), & \text{if } c_k(\alpha) < b_k \\ 0, & \text{otherwise} \end{cases}$	TAS (Dong & Yang, 2019a)

same relaxation as described in Equation 7, along with the same architecture selection procedure. More training details are provided in Appendix D.

CONNAS consistently finds architectures that satisfy the specified hardware constraints while maintaining strong predictive performance. The resulting architectures have relative errors of at most -1.24%, -3.38%, and -4.20% compared to the optimal feasible solution on CIFAR-10, CIFAR-100, and ImageNet16-120, respectively. Compared to the loss-based baselines, CONNAS yields considerably better results than both the SUMMED and MULTIPLIED variants. Yet, experiments with lower weighting factors (Appendix C) show improved performance for these baselines, but do not guarantee that valid architectures will be found within the given search budget. This highlights that the weighting in loss-based baselines exhibits higher sensitivity, requiring careful tuning for each hardware constraint setting, whereas CONNAS is more robust to the choice of rescale factor R. The PIECE-WISE baseline, on the other hand, achieves comparable predictive performance to CONNAS, but fails to always identify valid architectures under most constraint configurations, regardless of the weighting factor used.



(a) NATS-Bench-TS



(b) NATS-Bench-SS

Figure 2: **Comparison of the relative error of CONNAS with loss-based baselines.** Runs that did not satisfy the hardware constraints are excluded.

4.2 PRACTICAL USE CASE: CONDITION MONITORING OF INDUCTION MOTORS

Condition monitoring of industrial assets has received an increase interest over the years (Surucu et al., 2023), preventing unexpected failures and costly downtimes. By leveraging edge machine learning, data from these assets can be captured and processed locally to detect faults in real-time. To demonstrate the applicability of our work, we validate our approach through a use case focused

Table 2: **NATS-Bench classification results.** Test accuracy (mean ± std over 5 runs) is reported under hardware constraints, including the number of parameters, number of FLOPs, and peak memory usage. Memory constraints are only applied to NATS-Bench-SS, since all architectures in NATS-Bench-TS have the same memory usage. The constraints are chosen such that about 50% of the architectures in each search space meet them. $c_{\text{ts,comb}}$ and $c_{\text{ss,comb}}$ are a combination of the former constraints. Since NATS-Bench provides performance data for each architecture in the search space, we report the relative error (between parentheses) compared to the optimal architecture satisfying the hardware constraints. Best results for each constraint and dataset are highlighted in bold. Performance of runs that do not always satisfy the constraints are reported in gray.

Method	Top-1 accuracy (%)			Satisfied
	CIFAR-10	CIFAR-100	ImageNet16-120	
NATS-Bench-TS				
$c_{\text{ts,param}}$: #parameters ≤ 127514 *(7984 architectures satisfied)*				
ConNAS	**92.29** ± 0.00 *(-1.21)*	**67.48** ± 0.00 *(-3.34)*	**39.40** ± 0.00 *(-4.20)*	✓
Summed	88.77 ± 0.00 *(-4.73)*	62.08 ± 0.00 *(-8.74)*	32.42 ± 0.00 *(-11.18)*	✓
Multiplied	86.45 ± 0.00 *(-7.05)*	58.14 ± 0.00 *(-12.68)*	27.82 ± 0.00 *(-15.78)*	✓
Piece-wise	**92.29** ± 0.00 *(-1.21)*	**67.48** ± 0.00 *(-3.34)*	**39.40** ± 0.00 *(-4.20)*	✓
$c_{\text{ts,flops}}$: #FLOPs $\leq 59100M$ *(7954 architectures satisfied)*				
ConNAS	**92.29** ± 0.00 *(-1.21)*	**67.48** ± 0.00 *(-3.34)*	**39.40** ± 0.00 *(-4.20)*	✓
Summed	88.77 ± 0.00 *(-4.73)*	62.08 ± 0.00 *(-8.74)*	32.42 ± 0.00 *(-11.18)*	✓
Multiplied	86.45 ± 0.00 *(-7.05)*	58.14 ± 0.00 *(-12.68)*	27.82 ± 0.00 *(-15.78)*	✓
Piece-wise	92.25 ± 0.08	67.36 ± 0.27	39.30 ± 0.22	4 runs
$c_{\text{ts,comb}}$ *(7954 architectures satisfied)*				
ConNAS	**92.29** ± 0.00 *(-1.21)*	**67.48** ± 0.00 *(-3.34)*	**39.40** ± 0.00 *(-4.20)*	✓
Summed	88.77 ± 0.00 *(-4.73)*	62.08 ± 0.00 *(-8.74)*	32.42 ± 0.00 *(-11.18)*	✓
Multiplied	86.63 ± 0.18 *(-6.87)*	58.09 ± 0.12 *(-12.73)*	27.98 ± 0.18 *(-15.62)*	✓
Piece-wise	92.22 ± 0.10	67.24 ± 0.33	39.20 ± 0.27	3 runs
NATS-Bench-SS				
$c_{\text{ss,param}}$: #parameters ≤ 261650 *(16385 architectures satisfied)*				
ConNAS	**91.74** ± 0.29 *(-0.66)*	**66.22** ± 0.47 *(-2.70)*	**39.77** ± 0.77 *(-2.26)*	✓
Summed	85.87 ± 1.16 *(-6.66)*	51.11 ± 2.35 *(-17.81)*	25.27 ± 2.13 *(-16.76)*	✓
Multiplied	90.97 ± 0.11 *(-1.56)*	62.19 ± 0.30 *(-6.73)*	36.69 ± 1.10 *(-5.34)*	✓
Piece-wise	92.25 ± 0.02	67.34 ± 0.54	41.89 ± 0.33	0 runs
$c_{\text{ss,mem}}$: peak memory usage $\leq 655kB$ *(20480 architectures satisfied)*				
ConNAS	**93.28** ± 0.15 *(-0.14)*	69.70 ± 1.02 *(-1.16)*	44.58 ± 0.43 *(-1.35)*	✓
Summed	91.33 ± 0.19 *(-2.09)*	68.22 ± 0.28 *(-2.64)*	38.49 ± 0.03 *(-7.44)*	✓
Multiplied	91.53 ± 0.26 *(-1.89)*	67.57 ± 0.56 *(-3.29)*	38.55 ± 0.56 *(-7.38)*	✓
Piece-wise	92.93 ± 0.19 *(-0.49)*	**70.10** ± 0.32 *(-0.76)*	**44.98** ± 0.59 *(-0.95)*	✓
$c_{\text{ss,flops}}$: #FLOPs $\leq 344194M$ *(16385 architectures satisfied)*				
ConNAS	**92.22** ± 0.14 *(-0.75)*	**67.10** ± 0.48 *(-3.38)*	**41.79** ± 0.16 *(-2.58)*	✓
Summed	91.59 ± 0.21 *(-1.38)*	65.68 ± 0.85 *(-4.80)*	37.84 ± 0.62 *(-6.53)*	✓
Multiplied	91.30 ± 0.24 *(-1.67)*	63.97 ± 0.94 *(-6.51)*	36.97 ± 0.64 *(-7.40)*	✓
Piece-wise	92.57 ± 0.23	68.36 ± 0.68	42.71 ± 0.46	0 runs
$c_{\text{ss,comb}}$ *(13342 architectures satisfied)*				
ConNAS	**92.05** ± 0.16 *(-0.48)*	**65.87** ± 0.55 *(-3.04)*	**40.22** ± 0.96 *(-1.81)*	✓
Summed	87.76 ± 0.59 *(-4.77)*	56.22 ± 1,18 *(-12.69)*	26.39 ± 0.37 *(-15.64)*	✓
Multiplied	88.30 ± 0.92 *(-4.23)*	56.41 ± 2.39 *(-12.50)*	29.10 ± 1.37 *(-12.93)*	✓
Piece-wise	92.26 ± 0.01	67.47 ± 0.42	41.97 ± 0.35	0 runs

on condition monitoring of induction motors, aiming to detect eccentricity faults using a neural network deployed on an edge device. An eccentricity fault occurs when the rotor is not perfectly centered within the stator, resulting in an uneven air gap, causing an unbalanced magnetic pull (Desenfans et al., 2024). The prediction task involves classifying the type of eccentricity, specifically distinguishing between no eccentricity fault, static eccentricity (where the rotor remains consistently off-center relative to the stator), dynamic eccentricity (where the rotor's off-center position rotates over time relative to the stator), and mixed eccentricity (a combination of static and dynamic eccentricity).

The goal in this use case is to discover neural network architectures that are deployable across a wide range of microcontrollers with diverse hardware capabilities. To define realistic deployment

constraints, we base our hardware constraints on the STM32 family of microcontrollers[1], which are widely adopted in industrial applications. We construct a search space based on 1D convolutions consisting of 8 layers. Each layer has varying configurations in terms of the number of filters, convolution types, and the inclusion of skip connections, resulting in a total of 1.94 billion unique architectures. The detailed description of the search space is provided in Appendix E. The dataset contains 31,920 instances, split into 25,530 for training and 6,390 for testing. Each instance consists of 256 time steps sampled at 5 kHz, with eight features: *stator currents* (3), *phase voltages* (3), *rotor speed* (1), and *rotor angle* (1). We use 80% of the training set to train the supernet, while the remaining 20% serves as a validation set for architecture selection. The training and selection procedure follows the same approach outlined in Section 4.1. The selected architectures are retrained from scratch using five-fold cross-validation over 200 epochs. Their performance is subsequently evaluated on the test set. We compare the architectures found by CONNAS with hand-crafted models[2] sampled from the search space (listed in Appendix F). Further implementation details, including hyperparameter settings, are provided in Appendix D.

Table 3 compares the performance and resource consumption of architectures discovered by CONNAS with manually designed baselines. CONNAS consistently identifies architectures that outperform handcrafted ones in terms of accuracy under specific hardware constraints. For example, under constraint c_1, CONNAS discovers architectures that achieve an accuracy of 97.07%, which is 1.55% higher than CONV1D-REG, the best manually designed model under the same constraint. At c_4, the discovered architecture reaches 94.37% accuracy, compared to 93.93% for CONV1D-REG-MIN.

Table 3: **Classification performance and resource usage of models discovered by ConNAS under various hardware constraints, compared to handcrafted baseline models.** The constraints c_1, c_2, c_3, and c_4 are defined as follows: c_1: model size $\leq$ 2MB, peak memory usage $\leq$ 640kB; c_2: model size $\leq$ 256kB, peak memory usage $\leq$ 112kB; c_3: model size $\leq$ 128kB, peak memory usage $\leq$ 36kB; c_4: model size $\leq$ 64kB, peak memory usage $\leq$ 18kB.

Name	Top-1 accuracy (%)	Satisfied constraints				model size (bytes)	peak memory usage (bytes)
		c_1	c_2	c_3	c_4		
ConNAS (unconstrained)	97.36 ±0.27					6.06M ± 1.61M	524k ± 0.00
ConNAS + c_1	97.09 ±0.47	✓				1.93M ± 13.2k	524k ± 0.00
ConNAS + c_2	96.17 ±0.84	✓	✓			122k ± 30.7k	69.6k ± 0.00k
ConNAS + c_3	94.96 ±0.69	✓	✓	✓		26.0k ± 13.5k	16.8k ± 10.1k
ConNAS + c_4	94.37 ±1.94	✓	✓	✓	✓	8.69k ± 2.12k	12.3k ± 0.00
Conv1D-Reg	95.54 ±0.29	✓				135k	16.4k
Conv1D-DS	93.07 ±0.35	✓	✓	✓	✓	51.3k	16.4k
Conv1D-Reg-Max	93.31 ±1.86					22.1M	524k
Conv1D-DS-Max	96.72 ±0.24					7.50M	524k
Conv1D-Reg-Min	90.15 ±4.33	✓	✓	✓	✓	3.47k	12.3k
Conv1D-DS-Min	93.93 ±1.64	✓	✓	✓	✓	2.32k	12.3k

5 CONCLUSION

We introduced CONNAS, a novel hardware-aware, gradient-based NAS technique that explicitly enforces hardware constraints during the search process. Unlike prior approaches that rely on weighted loss terms for hardware-aware regularization, CONNAS directly modifies the gradients of architecture parameters when hardware constraints are violated, effectively steering the search toward hardware-feasible solutions. Experiments on NATS-Bench demonstrate that CONNAS consistently discovers architectures that satisfy various hardware constraints, achieving performance close to the optimal and significantly outperforming existing loss-based methods in both performance and compliance with hardware constraints. Furthermore, we validated CONNAS in a practical use case, where it successfully identified high-performing architectures under tight resource limitations. These results highlight CONNAS as a promising approach for the automatic design of deep learning models for deployment in resource-constrained environments.

[1] Specifically, the STM32H56, STM32G49, STM32C09, and STM32G05 series.

[2] We could have also compared to the loss-based baselines as in Section 4.1. However, since loss-based baselines require differentiable hardware metrics, this comparison is not feasible as factorizing the metrics with respect to α is intractable due to the large search space.

REFERENCES

Bowen Baker, Otkrist Gupta, Nikhil Naik, and Ramesh Raskar. Designing neural network architectures using reinforcement learning. In *International Conference on Learning Representations*, 2017.

Gabriel Bender, Hanxiao Liu, Bo Chen, Grace Chu, Shuyang Cheng, Pieter-Jan Kindermans, and Quoc V. Le. Can weight sharing outperform random architecture search? an investigation with TuNAS. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 14323–14332, 2020.

Han Cai, Ligeng Zhu, and Song Han. ProxylessNAS: Direct neural architecture search on target task and hardware. In *International Conference on Learning Representations*, 2019.

Liang-Chieh Chen, Maxwell Collins, Yukun Zhu, George Papandreou, Barret Zoph, Florian Schroff, Hartwig Adam, and Jon Shlens. Searching for efficient multi-scale architectures for dense image prediction. In *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

Francois Chollet. Xception: Deep learning with depthwise separable convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1251–1258, 2017.

Philip Desenfans, Zifeng Gong, Dries Vanoost, Konstantinos Gryllias, Jeroen Boydens, and Davy Pissoort. The influence of the unbalanced magnetic pull on fault-induced rotor eccentricity in induction motors. *Journal of Vibration and Control*, 30(5):943–959, 2024.

Philip Desenfans, Zifeng Gong, Dries Vanoost, and Davy Pissoort. A python magnetic equivalent circuit model for rapid data generation of induction motors in faulty conditions., 2025. URL `https://gitlab.kuleuven.be/m-group-campus-brugge/wavecore_public/phd-philip-desenfans/immec`.

Xuanyi Dong and Yi Yang. Network pruning via transformable architecture search. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019a.

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

Xuanyi Dong, Lu Liu, Katarzyna Musial, and Bogdan Gabrys. NATS-bench: Benchmarking NAS algorithms for architecture topology and size. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pp. 1–1, 2021. ISSN 0162-8828, 2160-9292, 1939-3539.

Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Efficient multi-objective neural architecture search via lamarckian evolution. In *International Conference on Learning Representations*, 2019a.

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

Geoffrey Hinton, Li Deng, Dong Yu, George E. Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N. Sainath, and Brian Kingsbury. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 29(6):82–97, 2012. ISSN 1558-0792.

Eric Jang, Shixiang Gu, and Ben Poole. Categorical reparameterization with gumbel-softmax. In *International Conference on Learning Representations*, 2017.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84–90, 2017. ISSN 0001-0782, 1557-7317.

Colin Lea, Michael D. Flynn, Rene Vidal, Austin Reiter, and Gregory D. Hager. Temporal convolutional networks for action segmentation and detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 156–165, 2017.

Edgar Liberis, Łukasz Dudziak, and Nicholas D. Lane. μnas: Constrained neural architecture search for microcontrollers. In *Proceedings of the 1st Workshop on Machine Learning and Systems*, EuroMLSys '21, pp. 70–79, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450382984.

Ji Lin, Wei-Ming Chen, Yujun Lin, John Cohn, Chuang Gan, and Song Han. MCUNet: Tiny deep learning on IoT devices. In *Advances in Neural Information Processing Systems*, volume 33, pp. 11711–11722. Curran Associates, Inc., 2020.

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

Esteban Real, Sherry Moore, Andrew Selle, Saurabh Saxena, Yutaka Leon Suematsu, Jie Tan, Quoc V. Le, and Alexey Kurakin. Large-scale evolution of image classifiers. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, pp. 2902–2911. JMLR.org, 2017.

Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, and Ilya Sutskever. Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864*, 2017.

Onur Surucu, Stephen Andrew Gadsden, and John Yawney. Condition monitoring using machine learning: A review of theory, applications, and recent advances. *Expert Systems with Applications*, 221:119738, 2023. ISSN 0957-4174.

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*, pp. 2820–2828, 2019.

Quinten Van Baelen and Peter Karsmakers. Constraint guided gradient descent: Training with inequality constraints with applications in regression and semantic segmentation. *Neurocomputing*, 556:126636, 2023. ISSN 0925-2312.

Alvin Wan, Xiaoliang Dai, Peizhao Zhang, Zijian He, Yuandong Tian, Saining Xie, Bichen Wu, Matthew Yu, Tao Xu, Kan Chen, Peter Vajda, and Joseph E. Gonzalez. FBNetV2: Differentiable neural architecture search for spatial and channel dimensions. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 12962–12971, 2020.

Colin White, Willie Neiswanger, and Yash Savani. Bananas: Bayesian optimization with neural architectures for neural architecture search. In *AAAI*, pp. 10293–10301, 2021.

Bichen Wu, Xiaoliang Dai, Peizhao Zhang, Yanghan Wang, Fei Sun, Yiming Wu, Yuandong Tian, Peter Vajda, Yangqing Jia, and Kurt Keutzer. FBNet: Hardware-aware efficient ConvNet design via differentiable neural architecture search. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 10726–10734, 2019.

Sirui Xie, Hehui Zheng, Chunxiao Liu, and Liang Lin. SNAS: stochastic neural architecture search. In *International Conference on Learning Representations*, 2019.

Yanqi Zhou, Siavash Ebrahimi, Sercan Ö Arık, Haonan Yu, Hairong Liu, and Greg Diamos. Resource-efficient neural architect. *arXiv preprint arXiv:1806.07912*, 2018.

Barret Zoph and Quoc Le. Neural architecture search with reinforcement learning. In *International Conference on Learning Representations*, 2017.

Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 8697–8710, 2018.

A Computation Hardware Metrics

In this work, we focus on three hardware metrics: the number of parameters, the number of FLOPs, and the peak memory usage during inference. The following subsections describe how each of these metrics is computed. It is important to note that our proposed method is not limited to these metrics and can be applied to any hardware metric, whether derived analytically, measured through profiling, or predicted by learned models.

A.1 Number of Parameters

The total number of parameters in a candidate architecture is computed by summing the number of parameters of each individual layer. To estimate the model size in bytes, we multiply the total number of parameters by 4, assuming 32-bit floating-point representation.

A.1.1 Convolutional Block

A convolutional block consists of a convolutional layer, followed by a batch normalization and a ReLU activation. We assume that batch normalization is fused into the convolutional layer during inference.

Regular Convolution:

$$\text{params} = (C_{in} \times k_h \times k_w + 1) \times C_{out} \tag{8}$$

where C_{in} and C_{out} are the input and output channels, k_h and k_w are kernel dimensions, and the $+1$ accounts for the bias term.

Depthwise Separable Convolution:

$$\text{params} = (C_{in} \times k_h \times k_w + 1) + (C_{in} \times C_{out} + 1) \tag{9}$$

where the first term corresponds to the depthwise convolution and the second to the pointwise convolution.

A.2 Fully Connected Layer

$$\text{params} = (N_{in} + 1) \times N_{out} \tag{10}$$

where N_{in} and N_{out} are the input and output features, respectively, and the $+1$ accounts for the bias term.

A.3 Number of FLOPs

Similar to the parameter count, the total number of FLOPs is computed by summing the FLOPs of each individual layer.

A.3.1 Convolutional layer

$$\text{FLOPs} = 2 \times (C_{in} \times k_h \times k_w + 1) \times H_{out} \times W_{out} \times C_{out} \tag{11}$$

Here, H_{out} and W_{out} are the output feature map dimensions.

A.3.2 ReLu

$$\text{FLOPs} = H_{out} \times W_{out} \times C_{out} \tag{12}$$

A.3.3 Linear Layer

$$\text{FLOPs} = 2 \times (N_{in} + 1) \times N_{out} + N_{out} \tag{13}$$

A.3.4 Global Pooling Layer

$$\text{FLOPs} = (k_h \times k_w + 1) \times H_{out} \times W_{out} \times C_{out} \tag{14}$$

A.4 PEAK MEMORY USAGE

Peak memory usage during inference is estimated by computing the combined size of input and output feature maps for each layer. The maximum of these values across all layers is taken as the peak memory usage.

B Ablation Rescale Factor

Table 4: **NATS-Bench classification results on ConNAS with different rescale factors.** Test accuracy (mean ± std over 5 runs) is reported under hardware constraints, including the number of parameters, number of FLOPs, and peak memory usage. Memory constraints are only applied to NATS-Bench-SS, since all architectures in NATS-Bench-TS require the same amount of memory. The constraints are chosen so that about 50% of the architectures in each search space meet them. $c_{\text{ts,comb}}$ and $c_{\text{ss,comb}}$ are a combination of the former constraints. Since NATS-Bench provides performance data for each architecture in the search space, we report the relative error (between parentheses) compared to the optimal architecture satisfying the hardware constraints.

Rescaling factor	Top-1 accuracy (%) CIFAR-10	CIFAR-100	ImageNet16-120	Satisfied
NATS-Bench-TS				
$c_{\text{ts,param}}$: #parameters ≤ 127514 *(7984 architectures satisfied)*				
1.2	92.29 ± 0.00 *(-1.21)*	67.48 ± 0.00 *(-3.34)*	39.40 ± 0.00 *(-4.20)*	✓
1.5	92.29 ± 0.00 *(-1.21)*	67.48 ± 0.00 *(-3.34)*	39.40 ± 0.00 *(-4.20)*	✓
2.0	92.29 ± 0.00 *(-1.21)*	67.48 ± 0.00 *(-3.34)*	39.40 ± 0.00 *(-4.20)*	✓
5.0	90.24 ± 1.87 *(-3.26)*	64.38 ± 2.83 *(-6.44)*	35.36 ± 3.69 *(-8.24)*	✓
10.0	89.58 ± 1.51 *(-3.92)*	63.31 ± 2.33 *(-7.51)*	33.96 ± 3.04 *(-9.64)*	✓
$c_{\text{ts,flops}}$: #FLOPs $\leq 59100M$ *(7954 architectures satisfied)*				
1.2	92.29 ± 0.00 *(-1.21)*	67.48 ± 0.00 *(-3.34)*	39.40 ± 0.00 *(-4.20)*	✓
1.5	91.61 ± 1.52 *(-1.89)*	66.45 ± 2.31 *(-4.37)*	38.05 ± 3.01 *(-5.55)*	✓
2.0	92.29 ± 0.00 *(-1.21)*	67.48 ± 0.00 *(-3.34)*	39.40 ± 0.00 *(-4.20)*	✓
5.0	90.24 ± 1.87 *(-3.26)*	64.38 ± 2.83 *(-6.44)*	35.36 ± 3.69 *(-8.24)*	✓
10.0	89.56 ± 1.52 *(-3.94)*	63.35 ± 2.31 *(-7.47)*	34.01 ± 3.01 *(-9.59)*	✓
$c_{\text{ts,comb}}$ *(7954 architectures satisfied)*				
1.2	92.29 ± 0.00 *(-1.21)*	67.48 ± 0.00 *(-3.34)*	39.40 ± 0.00 *(-4.20)*	✓
1.5	91.61 ± 1.52 *(-1.89)*	66.45 ± 2.31 *(-4.37)*	38.05 ± 3.01 *(-5.55)*	✓
2.0	92.29 ± 0.00 *(-1.21)*	67.48 ± 0.00 *(-3.34)*	39.40 ± 0.00 *(-4.20)*	✓
5.0	89.56 ± 1.52 *(-3.94)*	63.35 ± 2.31 *(-7.47)*	34.01 ± 3.01 *(-9.59)*	✓
10.0	89.56 ± 1.52 *(-3.94)*	63.35 ± 2.31 *(-7.47)*	34.01 ± 3.01 *(-9.59)*	✓
NATS-Bench-SS				
$c_{\text{ss,param}}$: #parameters ≤ 261650 *(16385 architectures satisfied)*				
1.2	91.74 ± 0.29 *(-0.79)*	66.22 ± 0.47 *(-2.70)*	39.77 ± 0.77 *(-2.27)*	✓
1.5	92.09 ± 0.24 *(-0.44)*	65.92 ± 0.65 *(-3.00)*	40.29 ± 1.10 *(-1.74)*	✓
2.0	91.87 ± 0.07 *(-0.66)*	66.22 ± 0.60 *(-2.70)*	39.48 ± 1.29 *(-2.55)*	✓
5.0	91.85 ± 0.19 *(-0.68)*	65.97 ± 0.38 *(-2.95)*	39.55 ± 0.53 *(-2.48)*	✓
10.0	91.67 ± 0.08 *(-0.86)*	65.99 ± 0.36 *(-2.93)*	39.31 ± 0.21 *(-2.73)*	✓
$c_{\text{ss,mem}}$: peak memory usage $\leq 655kB$ *(20480 architectures satisfied)*				
1.2	93.28 ± 0.20 *(-0.14)*	69.70 ± 1.02 *(-1.16)*	44.58 ± 0.43 *(-1.35)*	✓
1.5	93.26 ± 0.15 *(-0.16)*	70.14 ± 0.32 *(-0.72)*	44.67 ± 0.68 *(-1.26)*	✓
2.0	93.19 ± 0.17 *(-0.23)*	69.54 ± 0.95 *(-1.32)*	44.42 ± 0.38 *(-1.51)*	✓
5.0	93.26 ± 0.16 *(-0.16)*	70.06 ± 0.15 *(-0.80)*	44.65 ± 0.18 *(-1.29)*	✓
10.0	93.29 ± 0.11 *(-0.13)*	70.01 ± 0.24 *(-0.85)*	44.62 ± 0.22 *(-1.31)*	✓
$c_{\text{ss,flops}}$: #FLOPs $\leq 344194M$ *(16385 architectures satisfied)*				
1.2	92.22 ± 0.14 *(-0.75)*	67.10 ± 0.48 *(-3.38)*	41.79 ± 0.16 *(-2.57)*	✓
1.5	92.23 ± 0.19 *(-0.74)*	67.00 ± 0.80 *(-3.48)*	41.45 ± 0.92 *(-2.92)*	✓
2.0	92.10 ± 0.16 *(-0.87)*	67.08 ± 1.24 *(-3.40)*	41.63 ± 1.19 *(-2.74)*	✓
5.0	92.22 ± 0.08 *(-0.75)*	67.54 ± 1.09 *(-2.94)*	42.09 ± 0.53 *(-2.28)*	✓
10.0	92.31 ± 0.14 *(-0.66)*	66.76 ± 0.73 *(-3.72)*	41.70 ± 0.51 *(-2.67)*	✓
$c_{\text{ss,comb}}$ *(13342 architectures satisfied)*				
1.2	92.01 ± 0.19 *(-0.52)*	66.65 ± 0.39 *(-2.27)*	41.01 ± 0.33 *(-1.02)*	✓
1.5	91.72 ± 0.16 *(-0.81)*	66.02 ± 0.70 *(-2.90)*	40.23 ± 0.51 *(-1.81)*	✓
2.0	91.99 ± 0.38 *(-0.54)*	66.15 ± 0.45 *(-2.77)*	40.63 ± 0.98 *(-1.40)*	✓
5.0	91.94 ± 0.27 *(-0.59)*	66.02 ± 0.50 *(-2.90)*	40.57 ± 0.84 *(-1.46)*	✓
10.0	91.67 ± 0.36 *(-0.86)*	65.36 ± 0.89 *(-3.56)*	40.13 ± 0.65 *(-1.91)*	✓

C Ablation Weighting Factor Loss-based Baseline Methods

Table 5: **Comparison of classification results on NATS-Bench-TS across baseline methods and weighting factors.** Test accuracy (mean ± std over 5 runs) is reported under hardware constraints, including the number of parameters, and number of FLOPs. $c_{\text{ts,comb}}$ are a combination of the former constraints. The relative error compared to the optimal architecture satisfying the hardware constraints is reported between parentheses. Best results for each method, constraint, and dataset are highlighted in bold. Performance of experiments that do not satisfy the hardware constraints are reported in gray.

Method	$\lambda_{\text{hardware}}$	Top-1 accuracy (%)			Satisfied
		CIFAR-10	CIFAR-100	ImageNet16-120	
$c_{\text{ts,param}}$: #parameters ≤ 127514 *(7984 architectures satisfied)*					
Summed	**0.5**	**92.29 ± 0.00 *(-1.21)***	**67.48 ± 0.00 *(-3.34)***	**39.40 ± 0.00 *(-4.20)***	✓
Summed	1.0	88.77 ± 0.00 *(-4.73)*	62.08 ± 0.00 *(-8.74)*	32.42 ± 0.00 *(-11.18)*	✓
Summed	1.5	88.77 ± 0.00 *(-4.73)*	62.08 ± 0.00 *(-8.74)*	32.42 ± 0.00 *(-11.18)*	✓
Multiplied	**0.5**	**86.45 ± 0.00 *(-7.05)***	**58.14 ± 0.00 *(-12.68)***	**27.82 ± 0.00 *(-15.78)***	✓
Multiplied	**1.0**	**86.45 ± 0.00 *(-7.05)***	**58.14 ± 0.00 *(-12.68)***	**27.82 ± 0.00 *(-15.78)***	✓
Multiplied	1.5	86.42 ± 0.06 *(-7.08)*	58.09 ± 0.12 *(-12.73)*	27.92 ± 0.21 *(-15.68)*	✓
Piece-wise	0.5	92.06 ± 0.06	66.98 ± 0.14	39.00 ± 0.14	0 runs
Piece-wise	**1.5**	**92.29 ± 0.00 *(-1.21)***	**67.48 ± 0.00 *(-3.34)***	**39.40 ± 0.00 *(-4.20)***	✓
Piece-wise	**1.5**	**92.29 ± 0.00 *(-1.21)***	**67.48 ± 0.00 *(-3.34)***	**39.40 ± 0.00 *(-4.20)***	✓
$c_{\text{ts,flops}}$: #FLOPs $\leq 59100M$ *(7954 architectures satisfied)*					
Summed	**0.5**	**92.29 ± 0.00 *(-1.21)***	**67.48 ± 0.00 *(-3.34)***	**39.40 ± 0.00 *(-4.20)***	✓
Summed	1.0	88.77 ± 0.00 *(-4.73)*	62.08 ± 0.00 *(-8.74)*	32.42 ± 0.00 *(-11.18)*	✓
Summed	1.5	88.77 ± 0.00 *(-4.73)*	62.08 ± 0.00 *(-8.74)*	32.42 ± 0.00 *(-11.18)*	✓
Multiplied	**0.5**	**86.45 ± 0.00 *(-7.05)***	**58.14 ± 0.00 *(-12.68)***	**27.82 ± 0.00 *(-15.78)***	✓
Multiplied	**1.0**	**86.45 ± 0.00 *(-7.05)***	**58.14 ± 0.00 *(-12.68)***	**27.82 ± 0.00 *(-15.78)***	✓
Multiplied	1.5	86.40 ± 0.08 *(-7.10)*	58.03 ± 0.15 *(-12.79)*	28.01 ± 0.26 *(-15.59)*	✓
Piece-wise	0.5	92.25 ± 0.57	67.73 ± 1.30	39.24 ± 0.29	0 runs
Piece-wise	1.0	92.25 ± 0.08	67.36 ± 0.27	39.30 ± 0.22	4 runs
Piece-wise	**1.5**	**92.29 ± 0.00 *(-1.21)***	**67.48 ± 0.00 *(-3.34)***	**39.40 ± 0.00 *(-4.20)***	✓
$c_{\text{ts,comb}}$ *(7954 architectures satisfied)*					
Summed	**0.5**	**92.29 ± 0.00 *(-1.21)***	**67.48 ± 0.00 *(-3.34)***	**39.40 ± 0.00 *(-4.20)***	✓
Summed	1.0	88.77 ± 0.00 *(-4.73)*	62.08 ± 0.00 *(-8.74)*	32.42 ± 0.00 *(-11.18)*	✓
Summed	1.5	88.77 ± 0.00 *(-4.73)*	62.08 ± 0.00 *(-8.74)*	32.42 ± 0.00 *(-11.18)*	✓
Multiplied	0.5	86.45 ± 0.00 *(-7.05)*	58.14 ± 0.00 *(-12.68)*	27.82 ± 0.00 *(-15.78)*	✓
Multiplied	**1.0**	**86.63 ± 0.18 *(-6.87)***	58.09 **± 0.12 *(-12.73)***	**27.98 ± 0.18 *(-15.62)***	✓
Multiplied	1.5	81.15 ± 6.24 *(-12.35)*	51.40 ± 7.44 *(-19.42)*	23.89 ± 4.02 *(-19.71)*	✓
Piece-wise	0.5	92.04 ± 0.06	67.06 ± 0.16	39.01 ± 0.10	0 runs
Piece-wise	1.0	92.22 ± 0.10	67.24 ± 0.33	39.20 ± 0.27	3 runs
Piece-wise	**1.5**	**92.29 ± 0.00 *(-1.21)***	**67.48 ± 0.00 *(-3.34)***	**39.40 ± 0.00 *(-4.20)***	✓

Table 6: **Comparison of classification results on NATS-Bench-SS across baseline methods and weighting factors.** Test accuracy (mean ± std over 5 runs) is reported under hardware constraints, including the number of parameters, number of FLOPs, and peak memory usage. $c_{\text{ss,comb}}$ is a combination of the former constraints. The relative error compared to the optimal architecture satisfying the hardware constraints is reported between parentheses. Best results for each method, constraint, and dataset are highlighted in bold. Performance of experiments that do not satisfy the hardware constraints are reported in gray.

Method	$\lambda_{\text{hardware}}$	Top-1 accuracy (%)			Satisfied
		CIFAR-10	CIFAR-100	ImageNet16-120	
$c_{\text{ss,param}}$: #parameters ≤ 261650 *(16385 architectures satisfied)*					
Summed	**0.5**	**90.87 ± 0.02 *(-1.66)***	**61.31 ± 0.61 *(-7.61)***	**35.34 ± 0.31 *(-6.69)***	✓
Summed	1.0	85.87 ± 1.16 *(-6.66)*	51.11 ± 2.35 *(-17.81)*	25.27 ± 2.13 *(-16.76)*	✓
Summed	1.5	83.47 ± 0.74 *(-9.06)*	45.36 ± 2.95 *(-23.56)*	21.59 ± 1.01 *(-20.44)*	✓
Multiplied	0.5	92.11 ± 0.20	66.24 ± 0.94	41.21 ± 0.84	1 run
Multiplied	**1.0**	**90.97 ± 0.11 *(-1.56)***	**62.19 ± 0.30 *(-6.73)***	**36.69 ± 1.10 *(-5.34)***	✓
Multiplied	1.5	89.77 ± 0.29 *(-2.76)*	59.22 ± 1.93 *(-9.70)*	33.97 ± 1.05 *(-8.06)*	✓
Piece-wise	0.5	92.26 ± 0.01	67.41 ± 0.38	42.10 ± 0.15	0 runs
Piece-wise	1.0	92.25 ± 0.02	67.34 ± 0.54	41.89 ± 0.33	0 runs
Piece-wise	1.5	92.25 ± 0.02	67.34 ± 0.54	41.89 ± 0.33	0 runs
$c_{\text{ss,mem}}$: peak memory usage $\leq 655kB$ *(20480 architectures satisfied)*					
Summed	0.5	**91.61 ± 0.09 *(-1.81)***	67.90 ± 0.24 *(-2.96)*	38.30 ± 0.18 *(-7.63)*	✓
Summed	**1.0**	91.33 ± 0.19 *(-2.09)*	**68.22 ± 0.28 *(-2.64)***	**38.49 ± 0.03 *(-7.44)***	✓
Summed	**1.5**	91.33 ± 0.19 *(-2.09)*	**68.22 ± 0.28 *(-2.64)***	**38.49 ± 0.03 *(-7.44)***	✓
Multiplied	**0.5**	**92.01 ± 0.45 *(-1.41)***	**68.24 ± 0.34 *(-2.62)***	**40.20 ± 0.98 *(-5.73)***	✓
Multiplied	1.0	91.53 ± 0.26 *(-1.89)*	67.57 ± 0.56 *(-3.29)*	38.55 ± 0.56 *(-7.38)*	✓
Multiplied	1.5	91.67 ± 0.33 *(-1.75)*	67.04 ± 0.97 *(-3.82)*	38.07 ± 1.37 *(-7.86)*	✓
Piece-wise	0.5	93.08 ± 0.20 *(-0.34)*	70.23 ± 0.16 *(-0.63)*	**45.14 ± 0.31 *(-0.79)***	✓
Piece-wise	1.0	92.93 ± 0.19 *(-0.49)*	70.10 ± 0.32 *(-0.76)*	44.98 ± 0.59 *(-0.95)*	✓
Piece-wise	**1.5**	**93.13 ± 0.19 *(-0.29)***	**70.30 ± 0.19 *(-0.56)***	45.02 ± 0.32 *(-0.91)*	✓
$c_{\text{ss,flops}}$: #FLOPs $\leq 344194M$ *(16385 architectures satisfied)*					
Summed	**0.5**	**91.98 ± 0.00 *(-0.99)***	**66.66 ± 0.00 *(-3.82)***	**40.00 ± 0.00 *(-4.37)***	✓
Summed	1.0	91.59 ± 0.21 *(-1.38)*	65.68 ± 0.85 *(-4.80)*	37.84 ± 0.62 *(-6.53)*	✓
Summed	1.5	90.82 ± 0.12 *(-2.15)*	63.58 ± 0.91 *(-6.90)*	36.03 ± 0.59 *(-8.34)*	✓
Multiplied	**0.5**	**91.96 ± 0.04 *(-1.01)***	**66.76 ± 0.23 *(-3.72)***	**39.94 ± 0.13 *(-4.43)***	✓
Multiplied	1.0	91.30 ± 0.24 *(-1.67)*	63.97 ± 0.94 *(-6.51)*	36.97 ± 0.64 *(-7.40)*	✓
Multiplied	1.5	91.14 ± 0.46 *(-1.83)*	64.88 ± 0.91 *(-5.60)*	37.27 ± 0.86 *(-7.10)*	✓
Piece-wise	0.5	92.70 ± 0.18	69.10 ± 0.24	44.08 ± 0.18	0 runs
Piece-wise	1.0	92.57 ± 0.23	68.36 ± 0.68	42.71 ± 0.46	0 runs
Piece-wise	1.5	92.33 ± 0.00	67.66 ± 0.00	43.27 ± 0.00	0 runs
$c_{\text{ss,comb}}$ *(13342 architectures satisfied)*					
Summed	**0.5**	**90.66 ± 0.11 *(-1.87)***	**63.04 ± 0.34 *(-5.87)***	**35.59 ± 0.19 *(-6.44)***	✓
Summed	1.0	87.76 ± 0.59 *(-4.77)*	56.22 ± 1.18 *(-12.69)*	26.39 ± 0.37 *(-15.64)*	✓
Summed	1.5	83.99 ± 0.00 *(-8.54)*	48.64 ± 0.00 *(-20.27)*	22.50 ± 0.00 *(-19.53)*	✓
Multiplied	**0.5**	**90.08 ± 0.26 *(-2.45)***	**61.02 ± 1.21 *(-7.89)***	**34.06 ± 0.88 *(-7.97)***	✓
Multiplied	1.0	88.30 ± 0.92 *(-4.23)*	56.41 ± 2.39 *(-12.50)*	29.10 ± 1.37 *(-12.93)*	✓
Multiplied	1.5	88.45 ± 0.96 *(-4.08)*	56.25 ± 3.19 *(-12.66)*	30.13 ± 2.11 *(-11.90)*	✓
Piece-wise	0.5	92.53 ± 0.14	67.69 ± 0.63	42.78 ± 0.57	0 runs
Piece-wise	1.0	92.26 ± 0.01	67.47 ± 0.42	41.97 ± 0.35	0 runs
Piece-wise	1.5	92.26 ± 0.00	67.58 ± 0.00	42.03 ± 0.00	0 runs

D TRAINING DETAILS

To facilitate reproducibility, we provide additional experimental details in this appendix.[3]

D.1 NATS-BENCH

We adopt the same data augmentation techniques and training procedure as described in Dong et al. (2021). The search process is performed in two stages.

In the first stage, a supernet is trained on 50% of the CIFAR-10 training set (25,000 images) for 150 epochs, using a batch size of 64. The operation weights are optimized via Nesterov momentum Stochastic Gradient Descent (SGD) with a momentum of 0.9 and a weight decay of 5×10^{-4}. The initial learning rate is set to 0.0025 and annealed to 0.001 over 100 epochs using a cosine schedule. Architecture parameters are optimized using Adam with a learning rate of 0.001, weight decay of 1×10^{-3}, and $\beta_1 = 0.5$, $\beta_2 = 0.999$. The sampling temperature starts at 10 and is linearly annealed to 0.1 over 100 epochs. The rescale factor R is kept constant throughout training (we use $R = 1.2$ for most experiments; an ablation study on the rescale factor is presented in Appendix B).

In the second stage, an architecture is selected based on the lowest cross-entropy loss, evaluated on the remaining 50% of the training set, while satisfying the imposed hardware constraints. The NATS-Bench performance lookup table is then used to retrieve the test accuracy of the sampled architecture on CIFAR-10, CIFAR-100, and ImageNet16-120.

All experiments are repeated five times using fixed random seeds. The search is performed on a single NVIDIA V100 GPU and took approximately 5 hours to complete.

D.2 CONDITION MONITORING USE CASE

For the condition monitoring use case, a dataset is created from Desenfans et al. (2025), representing readings from a voltage meter, a current sensor and an encoder.[4] The complete dataset contains 31,920 time series instances, split into 25,530 train for training and 6,390 for testing. Each instance consists of 256 time steps with 8 features: 3-phase current, 3-phase voltage, motor speed, and rotor angle. The instances are labeled as either *no fault*, *static eccentricity fault*, *dynamic eccentricity fault*, or *mixed eccentricity fault*. A Fast Fourier Transform (FFT) is applied to the 3-phase current and voltage signals, and all 8 features are normalized to the range $[0, 1]$. The search process is again performed in two stages.

In the first stage, a supernet is trained on 80% of the training set (20424 instances) for 150 epochs, using a batch size of 128. The operation weights are optimized via Nesterov momentum Stochastic Gradient Descent (SGD) with a momentum of 0.9 and a weight decay of 5×10^{-4}. The initial learning rate is set to 0.0025 and annealed to 0.001 over 100 epochs using a cosine schedule. Architecture parameters are optimized using Adam with a learning rate of 0.001, weight decay of 1×10^{-3}, and $\beta_1 = 0.5$, $\beta_2 = 0.999$. The sampling temperature starts at 10 and is linearly annealed to 0.1 over 100 epochs. The rescale factor R is kept constant at 1.2.

In the second stage, an architecture is selected based on the lowest cross-entropy loss, evaluated on the remaining 20% of the training set, while satisfying the imposed hardware constraints. The selected architectures are trained using 5-fold cross-validation, where each fold uses 80% of the training data for training over 250 epochs with a batch size of 16. Optimization is performed using SGD with a momentum of 0.9 and a weight decay of 1×10^{-4}. The initial learning rate is set to 0.0025 and annealed to 0.0001 using a cosine scheduler.

All experiments are repeated five times using fixed random seeds. The search is performed on a single NVIDIA P100 GPU and took approximately 7 hours to complete.

[3] The code required to run all experiments will be made publicly available upon publication of the paper.

[4] The dataset will be made publicly available upon publication of the paper.

E CONDITION MONITORING SEARCH SPACE

The search space for the condition monitoring use case is defined as a supernet composed of eight consecutive edges. Each edge can select from the following candidate operations:

- A 1D convolution followed by a batch normalization and a ReLU activation
- A 1D depthwise separable convolution, where each convolution is followed by a batch normalization and a ReLU activation
- An identity operation (available only on even-numbered edges)

The classification head consists of a global average pooling layer followed by a fully connected layer.

For each convolution, the number of output channels can be chosen from $\{16, 32, 64, 256, 512\}$. Additionally, a stride of 2 is applied to convolutions on odd-numbered edges. Convolutional filters are shared across operations, as proposed in Wan et al. (2020), to reduce the number of trainable parameters and improve memory and computational efficiency.

F CONDITION MONITORING HAND-CRAFTED BASELINES

Table 7: **Overview of hand-crafted architectures used for comparison.** Each convolutional block consists of a convolutional operation (regular or depthwise seperable (Chollet, 2017)), a batch normalization, and a ReLU activation.

Name	Number of blocks	Kernel	Channels	Strides	Convolution type
Conv1D-Reg	4	3	[16, 32, 64, 128]	[2, 2, 2, 2]	Regular
Conv1D-DS	4	3	[16, 32, 64, 128]	[2, 2, 2, 2]	Depthwise Separable
Conv1D-Reg-Max	8	3	[512, 512, 512, 512, 512, 512, 512, 512]	[2, 1, 2, 1, 2, 1, 2, 1]	Regular
Conv1D-DS-Max	8	3	[512, 512, 512, 512, 512, 512, 512, 512]	[2, 1, 2, 1, 2, 1, 2, 1]	Depthwise Separable
Conv1D-Reg-Min	4	3	[8, 8, 8, 8]	[2, 2, 2, 2]	Regular
Conv1D-DS-Min	4	3	[8, 8, 8, 8]	[2, 2, 2, 2]	Depthwise Separable

G DISCLOSURE ON THE USE OF GENERATIVE ARTIFICIAL INTELLIGENCE

We used a large language model (LLM) as a writing aid. Ideation, scientific content, experimental design, and analysis were conducted entirely by the authors without any contribution of an LLM.