Neural Networks Performance Prediction using Weights and Gradients Analysis

Anonymous authors
Paper under double-blind review

Abstract

Neural network performance predictors are widely used to accelerate neural architecture search, but existing methods face a persistent trade-off: learning-based predictors require costly per-dataset initialization, while lightweight proxies are fast yet struggle to exploit prior experience and often degrade under dataset shift. We introduce NAP2, a hybrid performance predictor that models early training dynamics. NAP2 tracks the temporal evolution of layer-wise weight and gradient statistics over a small number of mini-batches, producing accurate rankings from as little as 100 mini-batches per candidate. Crucially, NAP2 supports cross-dataset reuse: a predictor trained on one dataset can be applied to another without fine-tuning, avoiding the re-initialization overhead incurred by many model-based approaches. Experiments on NAS-Bench-201 across CIFAR-10, CIFAR-100, and ImageNet16-120 show that NAP2 is competitive with strong hybrid baselines under limited budgets and delivers cost-effective cross-dataset transfer, outperforming established learning-curve and zero-cost baselines at short query times. We further demonstrate robustness to significant distribution shift, with a predictor trained on CIFAR-10 transferring effectively to SVHN. Our code and trained models are available at https://anonymous.4open.science/r/NAP2-6027/README.md.

1 Introduction

Neural network performance prediction aims to estimate the final performance of candidate architectures without fully training each one, and it is a key ingredient in scaling neural architecture search (NAS). Performance predictors are commonly compared under constraints on (i) predictor initialization cost (the compute required before deployment), (ii) per-architecture query cost, and (iii) ranking accuracy, typically measured via rank correlation.

Following the taxonomy of (White et al., 2021), existing predictors can be broadly grouped into: *(a)* model-based predictors that learn from previously evaluated architectures (Ji et al., 2024); *(b)* learning-curve predictors that extrapolate performance from partial training traces (Ding et al., 2025); *(c)* zero-cost predictors that compute training-free (or nearly training-free) proxies at/near initialization (Lee & Ham, 2024); and *(d)* weight-sharing approaches that evaluate many sub-networks using shared parameters (Zhang et al., 2023). Model-based predictors can yield low query cost after an expensive initialization phase, but they often require re-training or adaptation when moving to a new dataset, limiting their practical reuse under dataset shift. Zero-cost proxies avoid initialization and minimize query time, but their static nature typically imposes a ceiling on ranking fidelity, since the predictor does not improve as more training signal becomes available. Learning-curve predictors can be accurate, but they generally require partial training per candidate, increasing query time.

This study focuses on cross-dataset reuse of performance predictors: given a predictor trained on a source dataset, the goal is to rank architectures on a target dataset with minimal per-architecture training and without target-specific predictor re-initialization. The main technical difficulty is that dataset shift changes both attainable accuracies and early optimization behavior, so predictors that rely on dataset-specific calibrations or purely static signals often degrade when transferred.

To address the limitations described above, we propose Neural Architectures Performance Prediction (NAP2), a hybrid predictor (in the sense of combining a learned predictor with learning-curve signals) that represents early training dynamics via the temporal evolution of layer-wise summary statistics of weights and gradients. Unlike static, architecture-only descriptors, NAP2 explicitly leverages short learning-curve information (e.g., the first few hundred mini-batches) while still amortizing experience through a learned predictor that can be reused across datasets. Empirically, we show that NAP2 can produce strong rankings from very short partial training runs (e.g., 100 mini-batches per candidate) and that predictors trained on one dataset can be applied to other datasets without fine-tuning while retaining competitive rank correlation.

We evaluate NAP2 on NAS-Bench-201 across CIFAR-10, CIFAR-100, and ImageNet16-120, demonstrating competitive results against strong hybrid baselines under limited budgets, and cross-dataset transfer compared to established learning-curve and zero-cost baselines in cross-dataset settings. Our contributions are as follows:

- We introduce NAP2, a hybrid performance predictor that models early learning dynamics using time-indexed statistics of weights and gradients.
- We demonstrate cross-dataset reuse of a learned predictor without target-dataset fine-tuning, while using only short partial training per queried architecture.
- We demonstrate robustness to significant distribution shift, showing that a predictor trained on CIFAR-10 transfers effectively to SVHN without fine-tuning, outperforming baselines by a wide margin.

2 Related Work

Neural network performance prediction methods differ in computational cost, prediction accuracy, and data types used, but can all be evaluated by three metrics: *a)* **Initialization** (pre-use compute/time to train the predictor); *b)* **Query** (compute/time per prediction); and *c)* **Performance** (accuracy after training).

In White et al. (2021), the authors propose the following categorization for performance predictors:

Model-based methods fully train many architectures, extract meta-features, and train a model to predict performance, resulting in high initialization but low query times. Prediction methods include Bayesian optimization (Shi et al., 2020; Kandasamy et al., 2018), evolutionary algorithms (Xue et al., 2023; Xiao et al., 2024), graph neural networks (Gao et al., 2023; Li et al., 2022), and Gaussian processes (Li et al., 2020; Lee et al., 2017). Recent advances, like CAP (Ji et al., 2024), leverage context to require fewer annotations.

Learning curve-based methods predict final performance by extrapolating from partial learning curves or losses. Techniques fit parametric models to validation accuracies (Domhan et al., 2015), or use iterative pruning as in Hyperband (Li et al., 2017) and SMAC3 (Lindauer et al., 2022). Bayesian neural networks can extrapolate partially trained curves (Klein et al., 2022). Recent works enhance these with dataset-agnostic learning (Ang et al., 2024), and structured neural representations for better modeling (Ding et al., 2024). However, query time is high, as each candidate generally needs partial training.

Zero-cost methods eliminate initialization and minimize query time by using single or few mini-batches, or even no training. Predictions rely on activation similarities (Mellor et al., 2021), or data-dependent/independent features; for example, synflow (Tanaka et al., 2020) computes a L1 path-norm from weights only. Approaches like Zen-NAS (Lin et al., 2021) require only a few forward passes for ranking, and Nas-Bench-Suite-Zero (Krishnakumar et al., 2022) provides a unified suite of such proxies. Some estimate performance using gradients or Jacobian measures (Abdelfattah et al., 2021).

Weight-sharing methods train a single over-parameterized network representing many sub-networks, improving efficiency. Notable approaches include DARTS (Liu et al., 2018) and its variants (Ye et al., 2022; Huang et al., 2023), ENAS (Pham et al., 2018) parameter sharing, ProxylessNAS' direct search (Cai et al., 2018), and TuNAS (Bender et al., 2020) for transferability. An approach using a single-path in the over-parameterized network further reduced search cost (Guo et al., 2020).

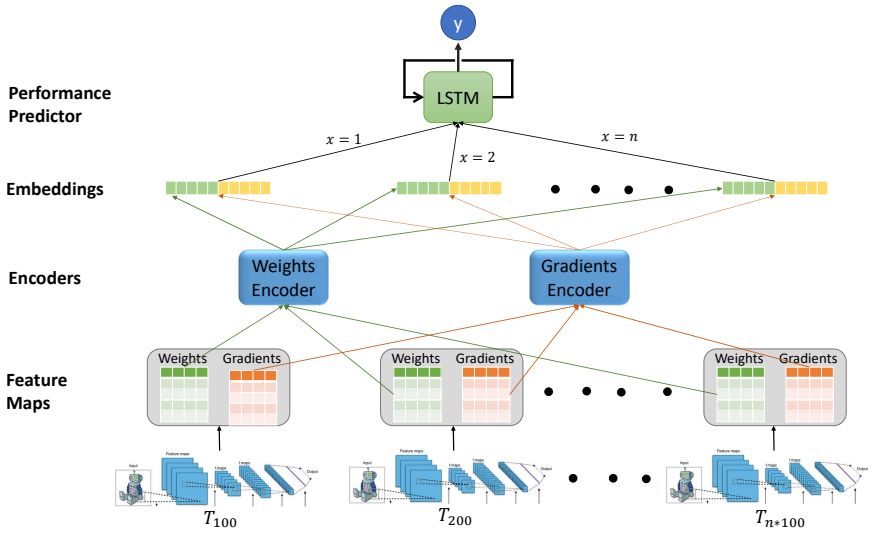


Figure 1: The proposed NAP2 architecture. At fixed batch intervals, we extract feature maps that represent the weights and gradients of the analyzed network. These feature maps are compressed using an encoder, and then provided as input to an LSTM model. This model predicts the analyzed network's final performance. Note that because LSTMs don't require a fixed-length input, NAP2 can produce predictions after any sequence length.

Our proposed approach falls into a category defined by White et al. (2021) as *hybrid*, combining model-based and learning curve-based techniques. While NAP2's query time is longer than zero-cost approaches, it delivers accurate predictions from as little as 100 mini-batches (about 9 seconds on a GTX 1080 Ti; see Section 4.5), and models trained on one dataset generalize to others without fine-tuning, positioning our method as a highly efficient learning curve-based predictor.

3 The Proposed Method

Overview. NAP2 is presented in Figure 1. Starting with a set of architectures $|A|$, we analyze each architecture by repeatedly *snapshotting* its internal state early in training: every fixed interval of mini-batches, we log the network's weights and the corresponding gradients, and then transform these raw tensors into a fixed-shape representation. Concretely, at every snapshot we (i) extract layer-wise summary statistics from both weights and gradients, (ii) assemble them into two *feature maps* (one for weights, one for gradients) with a uniform tensor shape *independent of the architecture topology*, (iii) compress each feature map into a compact embedding using a convolutional autoencoder, and finally (iv) feed the resulting embedding sequence into an LSTM performance predictor. The LSTM can be queried after any number of snapshots, enabling prediction from very short partial training runs. A visual summary of the full four-stage pipeline, including tensor dimensions at each stage, is provided in Appendix A.

Our approach is grounded in the hypothesis that the early training dynamics of neural networks provides a tractable proxy for the geometry of the loss landscape (Li et al., 2018). Features such as the evolution of weight magnitudes and gradient statistics reflect the sharpness or flatness of the loss surface – properties linked to generalization (Hochreiter & Schmidhuber, 1997). Recent work has shown that early-phase trajectories of weights and sharpness reveal universal, architecture-agnostic patterns predictive of later performance (Kalra et al., 2025). We hypothesize that capturing these *distinctive, yet broadly shared, aspects of learning dynamics* can support robust and generalizable performance prediction.

3.1 Theoretical Motivation

This section provides the theoretical grounding for NAP2's design choices. We organize the discussion around three questions: (i) why weight and gradient statistics should relate to generalization, (ii) why temporal dynamics carry more signal than static snapshots, and (iii) why these features should transfer across datasets. A detailed formal treatment is provided in Appendix G.

Weight and gradient statistics as generalization proxies. PAC-Bayes generalization theory establishes that the expected generalization gap of a stochastic neural network is controlled by a complexity term that depends on the parameter-space distance between the learned weights and a reference prior, typically measured through weight norms McAllester (1999); Neyshabur et al. (2017). More concretely, Bartlett et al. (2017) showed that for feedforward networks, generalization bounds scale with the product of per-layer spectral norms, while Neyshabur et al. (2015) derived bounds based on per-layer Frobenius norms. Our L1 and L2 norm meta-features track precisely these quantities at the layer level, providing the predictor with direct access to the terms that govern generalization in these frameworks. The variance and standard deviation features further characterize the distribution of norms across units within a layer, capturing intra-layer heterogeneity that aggregate norms alone miss.

Beyond norms, the distributional shape of weight matrices carries generalization-relevant information. Martin & Mahoney (2021) demonstrated that the heavy-tail structure of weight matrix spectra is predictive of test-set generalization, with networks exhibiting heavier-tailed weight distributions tending to generalize better. Our kurtosis and skewness features serve as tractable proxies for this tail behavior without requiring expensive spectral decompositions.

Gradient statistics complement the weight-based features by characterizing the local loss landscape geometry. The gradient covariance across training examples is closely related to the Fisher information matrix Amari (1998), whose eigenspectrum reflects the curvature of the loss surface. By tracking both weights – where the network is in parameter space – and gradients – what the landscape looks like at that location – NAP2 obtains a more complete picture of the optimization state than either signal alone.

Temporal dynamics versus static snapshots. A single snapshot of weight or gradient statistics reveals nothing about the trajectory the network is following. Training dynamics are highly structured: for example, the edge-of-stability phenomenon Cohen et al. (2021) shows that under gradient descent, sharpness evolves in a characteristic pattern constrained by the step size, rather than varying randomly. Kalra et al. (2025) further demonstrated that trajectories of weight norms and sharpness exhibit universal patterns across architectures and datasets. By feeding a temporal sequence of feature maps into an LSTM, NAP2 captures the velocity and curvature of the optimization trajectory – not just its current position. This is strictly more informative than any single-time-point measurement, and explains why NAP2 improves rapidly over the first several snapshots before plateauing (Section 6.1). Li et al. (2018) showed that network architecture significantly shapes loss landscape geometry, with skip connections and depth producing smoother surfaces – a structural property linked to generalization Hochreiter & Schmidhuber (1997); Keskar et al. (2016).

Why cross-dataset transfer should work. The key hypothesis underlying NAP2's transferability is that the optimization trajectory features we extract are more strongly governed by the network architecture than by the specific dataset. Choromanska et al. (2015) showed that under simplifying assumptions, the structure of the loss surface – including the distribution of critical points – is primarily determined by the network topology. Similarly, the implicit bias of gradient descent depends on architectural properties such as depth and layer type Gunasekar et al. (2018). While different datasets shift the loss surface, they do so largely in ways that affect absolute performance levels rather than the qualitative dynamics of how different architectures traverse the landscape. Our min-max normalization of target accuracies (Section 4.4) removes dataset-specific scale, leaving the predictor to learn relative orderings driven by architecture-dependent dynamics. The empirical success of NAP2's cross-dataset transfer—including the challenging CIFAR-10 to SVHN shift (Section 5.3)—provides evidence that this hypothesis holds in practice. A more formal analysis of the conditions under which transfer succeeds is given in Appendix G.5.

3.2 Meta-Features Extraction and Representation

Meta-Features Extraction. For each analyzed architecture A, and at every snapshot time t, we extract *two parallel sets of layer-wise meta-features*: one computed from the current weights, and one computed from the current gradients. Meta-features are extracted from every *trainable* layer (dense or convolutional) at fixed training intervals; in our experiments we use an interval of 100 mini-batches, which we found to balance temporal resolution with storage overhead.

Step 1: Produce a vector of unit-level values per layer. For each layer, we first define the *unit* over which we will compute statistics, and produce a 1D list of values:

- **Dense layers:** the unit is a neuron. We compute one scalar per neuron by applying the chosen statistic to that neuron's incoming weights (or incoming gradients). Thus, a dense layer with 100 neurons yields 100 unit-level values.
- **Convolutional layers:** the unit is a filter (kernel). We compute one scalar (i.e., meta-feature value) per filter by applying the chosen statistic to that filter's kernel weights (or kernel gradients), flattened across input channels and spatial dimensions. Thus, a convolution with 128 filters yields 128 unit-level values (one per filter), regardless of the number of input channels.

Step 2: Add a global layer value. In addition to unit-level values, we also compute a *single global value* for the layer by applying the same statistic to all weights (or all gradients) in that layer. This global value summarizes the overall layer behavior and complements the unit-level values, which capture intra-layer heterogeneity.

Step 3: Compute the meta-feature families. We compute the following feature "families" for each layer, and do so identically for weights and gradients:

1) General statistics meta-features. This group consists of basic statistical operations: max, min, mean, variance, standard deviation, and median. Additionally, we compute five percentiles: the 0th (minimum), 25th, 50th (median), 75th, and 100th (maximum). Together, these five percentiles form the standard five-number summary, with the 25th and 75th percentiles capturing the interquartile range of the distribution. These features reflect the overall scale and variability of the weights and gradients, helping to characterize general learning behavior across layers.

2) Distribution-based meta-features. We calculate the co-variance, kurtosis and skewness of each analyzed layer.

$$\text{Skewness - } \frac{\mu_3}{\sigma^3} \tag{1}$$

$$\text{Kurtosis - } \frac{\mu_4}{\sigma^4} \tag{2}$$

where $\mu_i = \frac{1}{N}\sum_k (w_{ij\,k} - \overline{w_{ij}})^i$, w_{ij} is the weight, i is the index of the layer and j is the index of the neuron, σ is the standard deviation and N is the number of weights of the current neuron. Skewness and kurtosis capture asymmetry and sharpness in the distribution, which relate to curvature and geometry of the local loss surface.

3) Norm-based meta-features. We calculate the L_1 and L_2 norms over the weights/gradients of each layer. We apply the following approach: for a layer with weights W the norms are: $L_1 : \ ||W||_1 = \sum_i abs(w_i)$; $L_2 : \ ||W||_2 = \sqrt{\sum_i w_i^2}$. These features quantify update magnitudes, serving as a proxy for learning intensity and layer-wise signal strength.

Overall, we extract 24 meta-features per layer (12 for each of our two types). All statistical computations use NaN-aware implementations to ensure robust feature extraction even when layers contain inactive units. Formal definitions of all twelve meta-features, the sampling strategy, and the complete feature map construction procedure are given in Appendix B.

Rationale for weights and gradients. Weights-based features describe the evolving representation stored in the parameters, while gradients-based features describe the current optimization signal driving parameter

updates. Combining both is intended to capture learning dynamics rather than relying on static architectural descriptors, supporting transfer across datasets by conditioning on *how* the model learns rather than only on *what* it is.

Meta-Features Representation. A key challenge is that architectures differ in depth and layer widths, while the predictor requires a fixed-size input. We therefore convert the per-layer meta-features at a snapshot into two fixed-shape *feature maps* (FM): one for weights and one for gradients. The feature maps are built with three explicit axes: (i) *layer index*, (ii) *values-per-feature*, and (iii) *feature type*.

Step 1: Per-layer matrices of shape $[100, 12]$**.** For each layer, and separately for weights and gradients, we create a matrix with 12 columns (one column per meta-feature) and 100 rows (one row per stored value of that feature). The 100 stored values are constructed as follows:

- For each meta-feature (each of the 12 columns), we place the **global layer value** in the first row.
- We then fill the remaining 99 rows with **unit-level values** (one per neuron/filter), using random sampling when the layer has more than 99 units.
- If a layer contains fewer than 99 units, we pad with zeros to reach exactly 100 rows.

This step ensures that every layer, regardless of width, is represented by a fixed $[100, 12]$ matrix, while still preserving a coarse notion of within-layer variability via the unit-level rows.

Step 2: Stack layers into architecture-level feature maps of shape $[65, 100, 12]$**.** We stack the per-layer $[100, 12]$ matrices along the depth axis to form an architecture-level feature map. For each architecture at each snapshot we thus obtain *two* feature maps (weights and gradients), each of shape $[65, 100, 12]$:

- We keep the *first 65 layers* of the architecture; deeper networks are truncated by discarding the remaining layers.
- Networks with fewer than 65 layers are padded with all-zero layers at the end.

After this construction, each architecture snapshot is represented by two tensors with identical shape, enabling a single downstream encoder and predictor to be trained across heterogeneous architectures.

3.3 Generating Feature Maps-based Embeddings

While informative, our feature maps have high dimensionality, which makes training the prediction model computationally expensive. A more condensed FM representation will enable us to deploy a smaller performance prediction model and use less training data.

We use convolutional autoencoders (CAE) to create embeddings for each feature map. The architecture of the CAE is symmetric, consisting of five layers with dimensions: 512, 256, 128, 256, and 512. We use the ReLU activation function and apply batch normalization after each layer. We use the Mean Squared Error (MSE) loss function and measure the distance between the original and reconstructed feature map using MSE as our loss function. The embedding layer is a vector v, where $|v| = 128$. At inference time, only the two CAE encoders and the LSTM are needed, totaling approximately 28M parameters. Full architectural specifications for both the CAE and the LSTM predictor, including layer configurations, training hyperparameters, and parameter counts, are provided in Appendix C.

3.4 Training the Prediction Model

Our training process has two stages. First, we train two autoencoders – one for each feature map type (weights-based and gradients-based) – using training samples generated by randomly selecting architectures and time steps and retrieving the corresponding FM. In the second stage, we train a performance prediction model: an LSTM with one hidden layer and a Sigmoid output.

We augment each architecture in our training set with feature map embedding sequences of lengths 1 to 10, creating diverse training examples to help the model learn accurate predictions even with limited data. Training on varying sequence lengths enables efficient predictions from incomplete information. We also use the maximal sequence (23 snapshots) to ensure the model can utilize all available data, producing 11 training sequences per architecture (10 of varying lengths and 1 maximal).

Our training process yields a performance prediction model that is both generic – by training on diverse neural architectures, encouraging generalization – and robust, as it accurately predicts using anywhere from a single snapshot to the full sequence.

4 Experimental Setup

4.1 Evaluated Datasets

We replicate the setup of in White et al. (2021), and use three well-known datasets: CIFAR-10, CIFAR-100, and ImageNet16-120, each with 1,000 randomly sampled, disjoint architectures from NAS-Bench-201 (Dong & Yang, 2020), for a total of 3,000 distinct architectures. Following White et al. (2021), all sampled architectures were trained to convergence, and their meta-features (see previous section) were extracted to train our prediction model. Random subsets of these architectures were used for both training and testing, with varying train set sizes to control initialization times. Results were averaged over multiple runs to assess performance and stability. Detailed setup and methodology are provided in the following sections.

4.2 Baselines

Our evaluation consists of two parts. In the first, we compare NAP2 with current top performing hybrid performance prediction methods. In the second part, we perform a cross-dataset evaluation to assess NAP2's transferability to new datasets and architectures. Since existing hybrid and model-based methods require retraining on the target dataset, in the cross-dataset setting we compare NAP2 to learning curve-based and zero-cost methods, which similarly operate without target-specific initialization. In our first experimental setup, we *compare NAP2 to leading hybrid performance prediction methods*, which require fully trained architectures for model building and partial training on test architectures for feature extraction. As baselines, we use LcSVR (Baker et al., 2017) and Omni (White et al., 2021), two strong hybrid predictors that combine model-based learning with training-stage signals. LcSVR uses SVR on handcrafted and early stopping features, while Omni extends model-based predictors by integrating learning curve extrapolation (SoTL-E) into SemiNAS to balance predictive strength and efficiency. Omni is a strong, generalizable baseline, as shown in (White et al., 2021).

In our second experimental setup, we *evaluate NAP2 in a cross-dataset scenario*, training the prediction model on one dataset and applying it to new, unseen datasets and architectures. This setup allows the predictor to be reused without retraining. We compare NAP2 against baselines that similarly require no initialization on the target dataset. These include learning curve extrapolation methods, such as SoTL-E (Ru et al., 2021), which use partial training, and zero-cost proxies like SynFlow (Tanaka et al., 2020), GradNorm (Abdelfattah et al., 2021), and SNIP (Lee et al., 2018). The latter are training-free methods that estimate performance at initialization, providing a static baseline for our dynamic approach.

4.3 Evaluation Metrics

The authors of (White et al., 2021) use three metrics to report the performance of their evaluated predictors:

Initialization Time. This metric is applicable to predictors that pre-train a learning model (e.g., model-based methods). It measures the *total time* required to train the set of architectures based on which the learning model is created.

Query Time. Measures the average time the predictor needs to predict the performance of a *single architecture*.

Kendall rank correlation. Evaluates prediction accuracy by measuring the rank correlation (i.e., similarity in ordering) of two ranked lists. We use Kendall rank correlation to compare the rankings produced by each evaluated predictor to those reported in the NAS-Bench-201 dataset.

4.4 Training the NAP2 Prediction Model

The NAP2 prediction model requires extraction of meta-data from neural architectures throughout their training. Our training strictly followed that of the NAS-Bench-201 benchmark to ensure consistency and comparability with established results. We used the Nesterov momentum SGD optimizer, Cross-entropy loss function, an initial learning rate of 0.1 that decayed to 0 with a cosine annealing schedule, weight decay of 0.0005, and a batch size of 256.

For data augmentation, we applied random horizontal flips with a probability of 0.5 and random cropping. Specifically, we used a 32×32 patch with 4 pixels padding for CIFAR-10 and CIFAR-100 and a 16×16 patch with 2 pixels padding for ImageNet-16-120. Normalization was performed over the RGB channels. We extracted our meta-features at 100 mini-batch intervals. For cross-dataset experiments, target accuracies are min-max normalized to $[0, 1]$ using training-set statistics, so the predictor learns relative rankings rather than dataset-specific absolute values. Complete dataset specifications, normalization parameters, augmentation pipelines, and data split details are provided in Appendix D.

4.5 Experimental Setup

Our evaluation followed the methodology outlined by White et al. (2021). Each run samples 200 architectures and averages results over ten runs. The same training and test sets were used by all evaluated algorithms.

In our first set of experiments, comparison to hybrid methods, we obtained different initialization times by training a varying number of architectures to train the learning models of all evaluated algorithms. The number of trained architectures ranged from 50 to 700. In our second set of experiments, where the learning model is trained on architectures from one dataset and applied to another, we use 700 architectures for training. All experiments ran on a single GTX 1080 Ti, matching (White et al., 2021) for timing comparability. Full details on randomness control, hardware specifications, software versions, and computational cost are provided in Appendix E.

5 Evaluation Results

5.1 Evaluation Setup #1: Comparison to Hybrid Performance Prediction Methods

The results of our evaluation are presented in Tables 1-3. We evaluated the baselines on various initialization and query times to provide a comprehensive analysis of their performance. In the smaller datasets – CIFAR-10 and CIFAR-100 – NAP2 consistently outperforms the baselines in a large majority of initialization and query time combinations. In the larger ImageNet dataset, NAP2 performs better at longer query times, while the use of different initialization times does not discern any clear pattern.

In general, on CIFAR-10 and CIFAR-100, NAP2 outperforms both baselines in the large majority of initialization–query time combinations; on ImageNet16-120, it is competitive at shorter query times and surpasses the baselines at the longest query time. Our results are supported by the paired-t statistical test, which showed differences between NAP2 and the baselines are significant ($p < 0.01$) across all reviewed experiments. It is important to note that NAP2 uses a novel method to predict neural network performance, which opens the way for a possible future integration of NAP2 with other methods. We leave this research direction to future work.

5.2 Evaluation Setup #2: Cross-Dataset Generalization

A key challenge for performance predictors is generalizing to new, unseen datasets, which saves the large computational cost of retraining for each task (White et al., 2021). As explained in our description of the baselines, *NAP2 is a hybrid approach capable of cross-dataset performance prediction without requiring*

Table 1: Hybrid baselines performance on CIFAR-10.

Init Time	Query Time	LcSVR	Omni_seminas	NAP2
1.8e5	54	0.4415 ± 0.1036	0.5246 ± 0.0911	**0.5619 ± 0.0848**
1.8e5	72	0.5302 ± 0.0920	0.5388 ± 0.1162	**0.5795 ± 0.0844**
1.8e5	90	0.5312 ± 0.0562	0.5321 ± 0.1481	**0.5957 ± 0.0864**
1.8e5	108	0.5895 ± 0.0722	0.5598 ± 0.1131	**0.6025 ± 0.0914**
1.8e5	198	**0.7000 ± 0.0518**	0.5119 ± 0.1113	0.6200 ± 0.0999
3.6e5	54	0.4572 ± 0.0651	0.5667 ± 0.1230	**0.6140 ± 0.0923**
3.6e5	72	0.4992 ± 0.0671	0.5673 ± 0.1295	**0.6295 ± 0.0827**
3.6e5	90	0.5669 ± 0.0729	0.5958 ± 0.1483	**0.6398 ± 0.0790**
3.6e5	108	0.6403 ± 0.0759	0.5840 ± 0.1125	**0.6456 ± 0.0771**
3.6e5	198	**0.7309 ± 0.0601**	0.6240 ± 0.1115	0.6566 ± 0.0722
1.1e6	54	0.5009 ± 0.0821	0.5798 ± 0.0995	**0.6703 ± 0.0396**
1.1e6	72	0.5779 ± 0.0559	0.5864 ± 0.1098	**0.6960 ± 0.0443**
1.1e6	90	0.6126 ± 0.0713	0.6243 ± 0.1124	**0.7099 ± 0.0472**
1.1e6	108	0.6422 ± 0.0542	0.6209 ± 0.1268	**0.7149 ± 0.0485**
1.1e6	198	**0.7570 ± 0.0562**	0.6862 ± 0.1130	0.7348 ± 0.0559
2.2e6	54	0.4837 ± 0.0619	0.6278 ± 0.0886	**0.7304 ± 0.0332**
2.2e6	72	0.5868 ± 0.0596	0.6305 ± 0.0810	**0.7565 ± 0.0322**
2.2e6	90	0.6395 ± 0.0565	0.6574 ± 0.1212	**0.7749 ± 0.0291**
2.2e6	108	0.6744 ± 0.0732	0.6705 ± 0.0927	**0.7900 ± 0.0250**
2.2e6	198	0.7900 ± 0.0536	0.7456 ± 0.1049	**0.8269 ± 0.0174**

Table 2: Hybrid baselines performance on CIFAR-100.

Init Time	Query Time	LcSVR	Omni_seminas	NAP2
1.8e5	54	0.2943 ± 0.0733	**0.5624 ± 0.1960**	0.5285 ± 0.1234
1.8e5	72	0.4133 ± 0.1081	**0.5673 ± 0.1860**	0.5660 ± 0.0885
1.8e5	90	0.4032 ± 0.0790	0.5540 ± 0.1680	**0.5755 ± 0.0885**
1.8e5	108	0.4434 ± 0.0655	**0.6110 ± 0.1481**	0.5801 ± 0.0865
1.8e5	198	0.4992 ± 0.0847	**0.6165 ± 0.1041**	0.5770 ± 0.0829
3.6e5	54	0.3148 ± 0.0540	0.5778 ± 0.1648	**0.6623 ± 0.0237**
3.6e5	72	0.3986 ± 0.0618	0.5783 ± 0.1881	**0.6718 ± 0.0222**
3.6e5	90	0.4452 ± 0.0392	0.5982 ± 0.1874	**0.6796 ± 0.0237**
3.6e5	108	0.4754 ± 0.0386	0.6090 ± 0.1684	**0.6821 ± 0.0198**
3.6e5	198	0.5882 ± 0.0294	0.6587 ± 0.1238	**0.6862 ± 0.0235**
1.1e6	54	0.4113 ± 0.0507	0.6187 ± 0.1571	**0.6807 ± 0.0529**
1.1e6	72	0.4643 ± 0.0407	0.6324 ± 0.0824	**0.6984 ± 0.0443**
1.1e6	90	0.5003 ± 0.0363	0.6387 ± 0.1309	**0.7086 ± 0.0390**
1.1e6	108	0.5360 ± 0.0433	0.6575 ± 0.0861	**0.7147 ± 0.0343**
1.1e6	198	0.6253 ± 0.0361	0.7225 ± 0.0553	**0.7265 ± 0.0326**
2.2e6	54	0.4180 ± 0.0399	0.6456 ± 0.1031	**0.7013 ± 0.0492**
2.2e6	72	0.4581 ± 0.0622	0.6562 ± 0.1058	**0.7239 ± 0.0411**
2.2e6	90	0.5105 ± 0.0368	0.6855 ± 0.0708	**0.7368 ± 0.0347**
2.2e6	108	0.5513 ± 0.0441	0.6934 ± 0.0794	**0.7500 ± 0.0326**
2.2e6	198	0.6362 ± 0.0280	0.7567 ± 0.0868	**0.7906 ± 0.0261**

fine-tuning on the target dataset. Since NAP2 does not require initialization in cross-dataset setups, we compare NAP2 to top-performing learning curve extrapolation methods, which also have no initialization time. We structure our evaluation in two parts: first, we assess generalization across common image classification benchmarks. Second, to test robustness, we evaluate NAP2's performance under a significant dataset distribution shift.

The results of our evaluation are shown in Tables 4-6. From *all* the algorithms evaluated in (White et al., 2021), we report only those achieving top-3 performance for at least one query time. For NAP2, we present results with performance prediction models trained on one of the other two datasets. We also compare against LC-PFN (Adriaensen et al., 2023), a pre-trained learning curve predictor trained on over 1 million learning curves from diverse tasks. While LC-PFN benefits from extensive pre-training on external data, it provides a strong baseline for cross-dataset performance prediction, as it operates without fine-tuning on the target dataset. Note that LC-PFN requires at least 5 observations to produce reliable predictions, as its Bayesian prior-based approach needs sufficient data points to stabilize; consequently, LC-PFN shows negative or unreliable correlations in the earliest steps (1–4), which is reflected in our tables. In contrast,

Table 3: Hybrid baselines performance on ImageNet16-120.

Init Time	Query Time	LcSVR	Omni_seminas	NAP2
1.8e5	54	0.2708 ± 0.0769	**0.5370 ± 0.1755**	0.5028 ± 0.0772
1.8e5	72	0.3689 ± 0.0615	**0.5579 ± 0.1886**	0.5278 ± 0.0703
1.8e5	90	0.4038 ± 0.0538	**0.5793 ± 0.1476**	0.5499 ± 0.0672
1.8e5	108	0.4147 ± 0.0693	0.5458 ± 0.1251	**0.5654 ± 0.0667**
1.8e5	198	0.5289 ± 0.0604	0.5919 ± 0.1139	**0.6174 ± 0.0629**
3.6e5	54	0.3204 ± 0.0510	**0.5732 ± 0.1570**	0.5466 ± 0.0683
3.6e5	72	0.4114 ± 0.0646	**0.5797 ± 0.1934**	0.5721 ± 0.0482
3.6e5	90	0.4326 ± 0.0428	**0.5905 ± 0.2362**	0.5876 ± 0.0405
3.6e5	108	0.4472 ± 0.0500	**0.6072 ± 0.1868**	0.6018 ± 0.0400
3.6e5	198	0.5887 ± 0.0420	0.6169 ± 0.1898	**0.6315 ± 0.0422**
1.1e6	54	0.3555 ± 0.0576	**0.6595 ± 0.2000**	0.6415 ± 0.0341
1.1e6	72	0.4120 ± 0.0538	0.6607 ± 0.1572	**0.6647 ± 0.0360**
1.1e6	90	0.4706 ± 0.0455	**0.7018 ± 0.1703**	0.6740 ± 0.0359
1.1e6	108	0.5060 ± 0.0539	**0.7033 ± 0.1730**	0.6838 ± 0.0318
1.1e6	198	0.6165 ± 0.0411	**0.7273 ± 0.1167**	0.7241 ± 0.0321
2.2e6	54	0.3424 ± 0.0458	**0.6973 ± 0.1592**	0.6516 ± 0.0478
2.2e6	72	0.4136 ± 0.0452	**0.7124 ± 0.1285**	0.6752 ± 0.0403
2.2e6	90	0.4293 ± 0.0572	**0.7190 ± 0.1317**	0.6933 ± 0.0314
2.2e6	108	0.4926 ± 0.0469	**0.7217 ± 0.1758**	0.7095 ± 0.0226
2.2e6	198	0.6105 ± 0.0353	0.7517 ± 0.1722	**0.7725 ± 0.0180**

Table 4: The performance of learning curve-based predictors on CIFAR-10. NAP2 is trained on the two other datasets. SynFlow, GradNorm, and SNIP are zero-cost proxies (static performance).

Alg./ Q. time	SynFlow	GradNorm	SNIP	SoTL-E	SoTL	Early Stop (ACC.)	LCE-m	LC-PFN	NAP2 (IN)	NAP2 (C-100)
9	0.52 ± 0.04	0.46 ± 0.04	0.46 ± 0.04	0.48 ± 0.04	0.41 ± 0.04	0.43 ± 0.03	0.23 ± 0.04	0.31 ± 0.04	0.54 ± 0.09	**0.58 ± 0.08**
18	0.52 ± 0.04	0.46 ± 0.04	0.46 ± 0.04	0.63 ± 0.02	0.51 ± 0.03	0.52 ± 0.03	0.36 ± 0.04	0.37 ± 0.03	0.61 ± 0.08	**0.64 ± 0.05**
27	0.52 ± 0.04	0.46 ± 0.04	0.46 ± 0.04	**0.7 ± 0.01**	0.58 ± 0.02	0.56 ± 0.02	0.4 ± 0.03	0.48 ± 0.03	0.66 ± 0.05	0.67 ± 0.05
63	0.52 ± 0.04	0.46 ± 0.04	0.46 ± 0.04	**0.79 ± 0.01**	0.69 ± 0.02	0.59 ± 0.02	0.54 ± 0.03	0.61 ± 0.03	0.70 ± 0.03	0.72 ± 0.04
99	0.52 ± 0.04	0.46 ± 0.04	0.46 ± 0.04	**0.81 ± 0.01**	0.72 ± 0.02	0.63 ± 0.02	0.54 ± 0.04	0.66 ± 0.02	0.72 ± 0.03	0.73 ± 0.05
153	0.52 ± 0.04	0.46 ± 0.04	0.46 ± 0.04	**0.83 ± 0.01**	0.76 ± 0.01	0.64 ± 0.02	0.55 ± 0.04	0.69 ± 0.02	0.72 ± 0.03	0.73 ± 0.05
207	0.52 ± 0.04	0.46 ± 0.04	0.46 ± 0.04	**0.84 ± 0.01**	0.78 ± 0.01	0.69 ± 0.02	0.59 ± 0.03	0.71 ± 0.02	0.72 ± 0.03	0.72 ± 0.05

NAP2 demonstrates consistent performance from the very first observation, highlighting an advantage for early-stage prediction scenarios.

Our findings show that NAP2 consistently outperforms all baselines for shorter query times, though performance varies by dataset. *As dataset size and complexity grow, NAP2's relative performance improves*: on CIFAR-10, it leads for the two shortest query times; on CIFAR-100, for three; and on ImageNet, for all but the longest, where SoTL-E narrowly surpasses it (0.7 vs. 0.68).

Notably, even with the shortest query time (nine seconds, or 100 mini-batches), NAP2 achieves a Kendall Tau score of about 0.5 across all datasets - more than double the baselines on the larger datasets. This strong early performance highlights the transferability and effectiveness of weight and gradient modeling in predicting future network performance.

We further compare NAP2 against zero-cost proxies (SynFlow, GradNorm, SNIP), which represent the limit of "training-free" prediction. As shown in Tables 4-6, static proxies achieve a fixed correlation ceiling (e.g., ∼0.52 on CIFAR-10, ∼0.47 on ImageNet) that cannot improve with more compute. NAP2 matches or exceeds this ceiling at its very first step (100 mini-batches), achieving correlations of 0.54 (CIFAR-10), 0.51 (CIFAR-100), and 0.51 (ImageNet). Crucially, unlike static proxies, NAP2 leverages learning dynamics to rapidly improve, reaching correlations of >0.60 within just 600 mini-batches. This demonstrates that while zero-cost priors provide a good starting point, capturing the dynamic trajectory of training offers significantly higher predictive fidelity for a small computational cost.

5.3 Robustness to Distribution Shift: A Case Study on SVHN

While CIFAR-10/100 and ImageNet differ in many ways, they also share notable similarities, potentially enhancing NAP2's effectiveness in cross-dataset scenarios. To further assess NAP2's generalizability, we

Table 5: The performance of learning curve-based predictors on CIFAR-100. NAP2 is trained on the two other datasets. SynFlow, GradNorm, and SNIP are zero-cost proxies (static performance).

Alg./ Q. time	SynFlow	GradNorm	SNIP	SoTL-E	SoTL	Early Stop (ACC.)	LCE-m	LC-PFN	NAP2 (IN)	NAP2 (C-10)
9	0.51 ± 0.04	0.46 ± 0.03	0.46 ± 0.03	0.27 ± 0.05	0.21 ± 0.04	0.22 ± 0.03	0.15 ± 0.04	-0.13 ± 0.02	0.46 ± 0.10	**0.51 ± 0.07**
18	0.51 ± 0.04	0.46 ± 0.03	0.46 ± 0.03	0.43 ± 0.04	0.33 ± 0.04	0.39 ± 0.03	0.24 ± 0.04	-0.10 ± 0.03	0.54 ± 0.09	**0.56 ± 0.06**
27	0.51 ± 0.04	0.46 ± 0.03	0.46 ± 0.03	0.56 ± 0.02	0.44 ± 0.03	0.47 ± 0.03	0.27 ± 0.04	0.08 ± 0.03	**0.61 ± 0.08**	0.59 ± 0.06
63	0.51 ± 0.04	0.46 ± 0.03	0.46 ± 0.03	**0.74 ± 0.01**	0.62 ± 0.02	0.57 ± 0.02	0.41 ± 0.03	0.33 ± 0.03	0.68 ± 0.03	0.66 ± 0.04
99	0.51 ± 0.04	0.46 ± 0.03	0.46 ± 0.03	**0.79 ± 0.01**	0.68 ± 0.02	0.58 ± 0.02	0.53 ± 0.04	0.43 ± 0.03	0.70 ± 0.03	0.68 ± 0.02
153	0.51 ± 0.04	0.46 ± 0.03	0.46 ± 0.03	**0.82 ± 0.01**	0.72 ± 0.02	0.59 ± 0.02	0.52 ± 0.04	0.54 ± 0.02	0.70 ± 0.03	0.68 ± 0.03
207	0.51 ± 0.04	0.46 ± 0.03	0.46 ± 0.03	**0.84 ± 0.01**	0.76 ± 0.01	0.61 ± 0.02	0.48 ± 0.04	0.61 ± 0.02	0.70 ± 0.04	0.67 ± 0.03

Table 6: The performance of learning curve-based predictors on ImageNet. NAP2 is trained on the two other datasets. SynFlow, GradNorm, and SNIP are zero-cost proxies (static performance).

Alg./ Q. time	SynFlow	GradNorm	SNIP	SoTL-E	SoTL	Early Stop (ACC.)	LCE	LC-PFN	NAP2 (C-10)	NAP2 (C-100)
9	0.47 ± 0.03	0.41 ± 0.04	0.42 ± 0.04	0.2 ± 0.06	0.2 ± 0.05	0.22 ± 0.04	0.1 ± 0.05	-0.02 ± 0.05	**0.51 ± 0.04**	0.49 ± 0.07
18	0.47 ± 0.03	0.41 ± 0.04	0.42 ± 0.04	0.27 ± 0.05	0.24 ± 0.04	0.27 ± 0.04	0.16 ± 0.05	-0.05 ± 0.04	0.55 ± 0.04	**0.56 ± 0.05**
27	0.47 ± 0.03	0.41 ± 0.04	0.42 ± 0.04	0.35 ± 0.04	0.3 ± 0.04	0.31 ± 0.03	0.17 ± 0.05	-0.03 ± 0.05	0.57 ± 0.03	**0.59 ± 0.04**
63	0.47 ± 0.03	0.41 ± 0.04	0.42 ± 0.04	0.53 ± 0.03	0.42 ± 0.03	0.4 ± 0.03	0.26 ± 0.05	0.24 ± 0.04	0.60 ± 0.03	**0.63 ± 0.03**
99	0.47 ± 0.03	0.41 ± 0.04	0.42 ± 0.04	0.58 ± 0.02	0.47 ± 0.03	0.42 ± 0.03	0.35 ± 0.05	0.30 ± 0.04	0.60 ± 0.04	**0.65 ± 0.03**
153	0.47 ± 0.03	0.41 ± 0.04	0.42 ± 0.04	0.65 ± 0.02	0.53 ± 0.02	0.46 ± 0.02	0.43 ± 0.04	0.34 ± 0.04	0.62 ± 0.03	**0.66 ± 0.03**
207	0.47 ± 0.03	0.41 ± 0.04	0.42 ± 0.04	**0.7 ± 0.02**	0.58 ± 0.02	0.52 ± 0.02	0.48 ± 0.04	0.40 ± 0.04	0.64 ± 0.03	0.68 ± 0.03

Table 7: Accuracy score distribution comparison between CIFAR-10 and SVHN, highlighting the distribution shift.

Performance Range	CIFAR-10 (%)	SVHN (%)
[0–0.85]	40.0	5.3
[0.85–0.92]	41.5	6.7
[0.92–0.95]	18.5	13.7
[0.95–1]	0.0	74.3

Table 8: The probability of returning a top-performing architecture at the top of NAP2's ranked list of architectures. NAP2 was trained on CIFAR-10 and tested on CIFAR-100.

mini-batch step ($\times 10^2$)	1	2	3	4	5	10	15	20
top-3	0.2	0.4	0.3	0.2	0.2	0.1	0.1	0.2
top-5	0.2	0.4	0.3	0.3	0.3	0.4	0.3	0.4
top-10	0.2	0.6	0.4	0.6	0.5	0.6	0.5	0.5
top-20	0.5	0.7	0.9	0.8	0.8	0.7	0.6	0.6

Table 9: The probability of returning a top-performing architecture at the top of NAP2's ranked list of architectures. NAP2 was trained on CIFAR-10 and tested on ImageNet.

mini-batch step ($\times 10^2$)	1	2	3	4	5	10	15	20
top-3	0.0	0.1	0.0	0.2	0.2	0.1	0.4	0.3
top-5	0.1	0.2	0.1	0.3	0.3	0.4	0.5	0.5
top-10	0.1	0.2	0.3	0.4	0.5	0.6	0.6	0.7
top-20	0.3	0.5	0.5	0.6	0.6	0.7	0.7	0.7

Table 10: The probability of returning a top-performing architecture at the top of NAP2's ranked list of architectures. NAP2 was trained on CIFAR-100 and tested on ImageNet.

mini-batch step ($\times 10^2$)	1	2	3	4	5	10	15	20
top-3	0.1	0.3	0.2	0.2	0.3	0.2	0.2	0.2
top-5	0.3	0.3	0.6	0.3	0.4	0.3	0.3	0.4
top-10	0.3	0.5	0.8	0.5	0.8	0.8	1.0	0.9
top-20	0.3	0.8	0.9	0.9	0.9	1.0	1.0	1.0

evaluated it on the Street View House Numbers (SVHN) dataset, which is markedly different from the other datasets.

We randomly sampled 100 NAS-Bench-201 architectures and trained them on SVHN until convergence, ensuring none overlapped with previous experiments. We repeated this process three times. As shown in Table 7, the performance distribution on SVHN (compared to CIFAR-10) is distinct and challenging: when most architectures perform well, ranking becomes more difficult.

Figure 2 presents our cross-dataset results. We applied the model trained on CIFAR-10 (with no fine-tuning) and compared its performance to two baselines: SoTL-E (Ru et al., 2021), the top baseline from our experiments, and Early stop, a straightforward but effective benchmark. NAP2 clearly outperforms both:

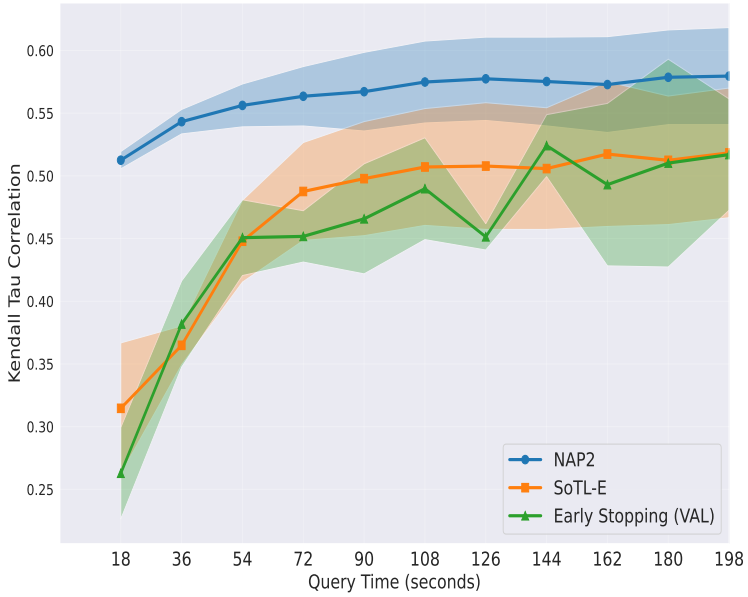


Figure 2: Kendall Tau on SVHN under cross-dataset transfer (NAP2 trained on CIFAR-10, no fine-tuning). Baselines are learning curve-based algorithms.

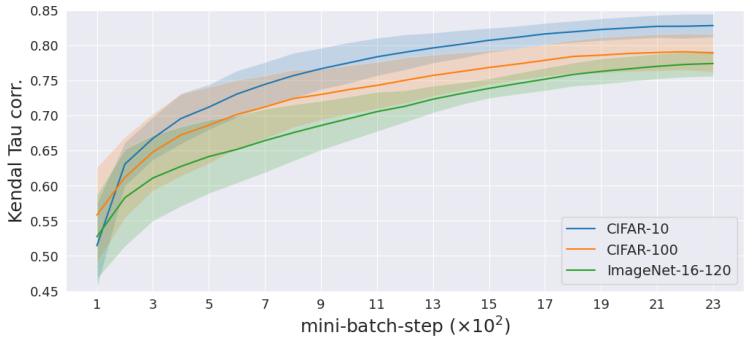


Figure 3: The Kendall Tau Rank Correlation achieved by NAP2 on the evaluated datasets, as a function of the number of mini-batches used to train each neural architecture.

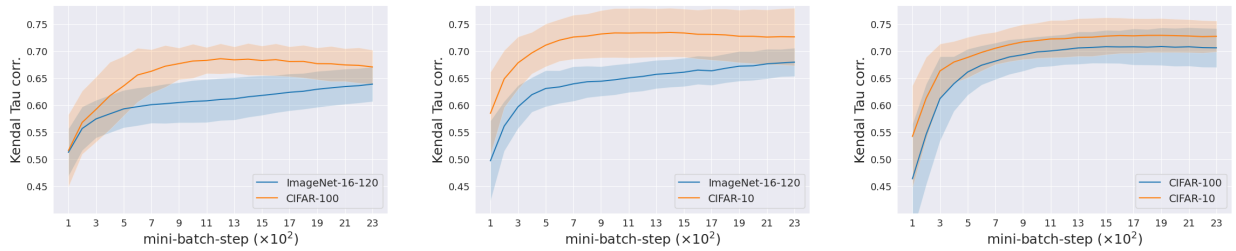
after a single epoch, it reaches a Kendall Tau of 0.5125, 80% higher than its closest competitor. Even after 11 epochs, neither baseline matches NAP2's single-epoch result. These results strengthen our claims regarding NAP2's transferability.

6 Analysis & Discussion

6.1 Analyzing performance as a function of query time

Our evaluation shows that NAP2 achieves relatively high Kendall Tau correlation values (about 0.5) after only a single input (100 mini-batches). However, as query times lengthen, NAP2 becomes less effective compared to some of the baselines. This analysis aims to identify the query-time range where NAP2 is most effective.

Figures 3 and 4 show performance in our two evaluation setups: single- and cross-dataset setup. Despite differences in absolute performance, two key conclusions emerge: (1) the largest improvement in NAP2's



(a) Trained on CIFAR-10 (b) Trained on CIFAR-100 (c) Trained on ImageNet-16-120

Figure 4: Kendall Tau achieved by NAP2 in cross-dataset transfer. Each panel shows a model trained on one dataset, evaluated on the other two, as a function of query mini-batches ($\times 100$).

performance occurs within the first ten inputs, after which gains slow significantly; (2) predictive abilities plateau after about 20 inputs.

We hypothesize that as training progresses, network weight and activation changes diminish, limiting NAP2's ability to provide further insights at later stages. We plan to address this in future work. We also explored an alternative architecture using separate LSTMs for weights and gradients (fusing only at the final hidden state), but found no improvement over early fusion via concatenation (see Appendix C).

6.2 Identifying Top-Performing Architectures

While NAP2 performs well in cross-dataset setups, a key practical question is whether it can reliably identify top-performing architectures. The Kendall Tau metric from previous evaluations does not address this, as it weights all architectures equally. Additionally, prediction models trained on datasets with different average performance may struggle to select the best architectures in new datasets. To investigate this, we assess NAP2's ability to identify top performers *in a cross-dataset setup*. After every 100 mini-batches, we checked if the true best architecture appeared in the top 3/5/10/20 of NAP2's rankings. Across ten runs per dataset (with 200-architecture test sets), we calculated how often the best architecture was highly ranked.

Tables 8–10 report results for cases where the model is trained on a smaller dataset and tested on a larger one. While performance varies, NAP2 typically needs only 600-700 batches (3 epochs on CIFAR-100, 1.2 on ImageNet) to reliably surface top candidates. Although comparable baselines are not provided in White et al. (2021), these results show that NAP2 not only ranks but also identifies top-performing architectures early—a crucial trait for NAS scenarios requiring quick identification of promising models. Across all configurations, Top-K recall improves monotonically with query time and generally plateaus after 1,000–1,500 mini-batches, providing a natural early-stopping signal for NAS practitioners. For a comprehensive analysis including in-domain results, Top-N Coverage metrics, and additional cross-dataset configurations, see Appendix F.

7 Conclusions and Future Work

We presented NAP2, a neural network performance prediction method that analyzes the temporal evolution of weight and gradient statistics during the early stages of training to predict final performance. On NAS-Bench-201, NAP2 is competitive with strong hybrid baselines under limited budgets and consistently outperforms learning-curve and zero-cost baselines at short query times in cross-dataset settings. A distinguishing property of NAP2 is its support for cross-dataset transfer: prediction models trained on one dataset can be applied to unseen datasets without re-initialization or fine-tuning, a capability that distinguishes it from many existing hybrid or model-based approaches that require target-domain training. This transferability extends even to significant distribution shift, as demonstrated by the SVHN case study, where NAP2 trained on CIFAR-10 achieved 80% higher Kendall Tau than its closest competitor after a single epoch.

Several directions for future work emerge from this study. The most immediate is the integration of NAP2 into a full NAS framework. While the present work focuses on performance prediction in isolation, deploying

NAP2 as the evaluation engine within a NAS loop would require balancing exploration of the architecture search space with exploitation of NAP2's predictions. We plan to investigate the use of deep reinforcement learning to guide this process, using NAP2's early predictions as reward signals to adaptively allocate training budget across candidate architectures.

Beyond NAS, NAP2's layer-wise weight and gradient representations may find applications in related domains. In neural network robustness and verification, methods such as ReluPlex rely on analyzing individual neurons to verify network properties; NAP2's per-layer statistics could serve as a guide for identifying which layers or neurons are most informative to target, potentially reducing the computational cost of verification. More broadly, the ability to characterize a network's learning trajectory from short training runs opens possibilities for tasks such as predicting training stability, detecting pathological optimization behavior early, or guiding hyperparameter selection across datasets.

References

Mohamed S Abdelfattah, Abhinav Mehrotra, Łukasz Dudziak, and Nicholas D Lane. Zero-cost proxies for lightweight nas. *arXiv preprint arXiv:2101.08134*, 2021.

Steven Adriaensen, Herilalaina Rakotoarison, Samuel Müller, and Frank Hutter. Efficient bayesian learning curve extrapolation using prior-data fitted networks. *Advances in Neural Information Processing Systems*, 36:19858–19886, 2023.

Shun-Ichi Amari. Natural gradient works efficiently in learning. *Neural computation*, 10(2):251–276, 1998.

Sui Paul Ang, Soan TM Duong, Son Lam Phung, and Abdesselam Bouzerdoum. Dap: A dataset-agnostic predictor of neural network performance. *Neurocomputing*, 583:127544, 2024.

Bowen Baker, Otkrist Gupta, Ramesh Raskar, and Nikhil Naik. Accelerating neural architecture search using performance prediction. *arXiv preprint arXiv:1705.10823*, 2017.

Peter L Bartlett, Dylan J Foster, and Matus J Telgarsky. Spectrally-normalized margin bounds for neural networks. *Advances in neural information processing systems*, 30, 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. *arXiv preprint arXiv:1812.00332*, 2018.

Anna Choromanska, Mikael Henaff, Michael Mathieu, Gérard Ben Arous, and Yann LeCun. The loss surfaces of multilayer networks. In *Artificial intelligence and statistics*, pp. 192–204. PMLR, 2015.

Jeremy M Cohen, Simran Kaur, Yuanzhi Li, J Zico Kolter, and Ameet Talwalkar. Gradient descent on neural networks typically occurs at the edge of stability. *arXiv preprint arXiv:2103.00065*, 2021.

Yanna Ding, Zijie Huang, Xiao Shou, Yihang Guo, Yizhou Sun, and Jianxi Gao. Architecture-aware learning curve extrapolation via graph ordinary differential equation. *arXiv preprint arXiv:2412.15554*, 2024.

Yanna Ding, Zijie Huang, Xiao Shou, Yihang Guo, Yizhou Sun, and Jianxi Gao. Architecture-aware learning curve extrapolation via graph ordinary differential equation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pp. 16289–16297, 2025.

Tobias Domhan, Jost Tobias Springenberg, and Frank Hutter. Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves. In *Twenty-fourth international joint conference on artificial intelligence*, 2015.

Xuanyi Dong and Yi Yang. Nas-bench-201: Extending the scope of reproducible neural architecture search. *arXiv preprint arXiv:2001.00326*, 2020.

Yanjie Gao, Xianyu Gu, Hongyu Zhang, Haoxiang Lin, and Mao Yang. Runtime performance prediction for deep learning models with graph neural network. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pp. 368–380. IEEE, 2023.

Suriya Gunasekar, Jason D Lee, Daniel Soudry, and Nati Srebro. Implicit bias of gradient descent on linear convolutional networks. *Advances in neural information processing systems*, 31, 2018.

Zichao Guo, Xiangyu Zhang, Haoyuan Mu, Wen Heng, Zechun Liu, Yichen Wei, and Jian Sun. Single path one-shot neural architecture search with uniform sampling. In *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part XVI 16*, pp. 544–560. Springer, 2020.

Sepp Hochreiter and Jürgen Schmidhuber. Flat minima. *Neural computation*, 9(1):1–42, 1997.

Lan Huang, Shiqi Sun, Jia Zeng, Wencong Wang, Wei Pang, and Kangping Wang. U-darts: Uniform-space differentiable architecture search. *Information Sciences*, 628:339–349, 2023.

Han Ji, Yuqi Feng, and Yanan Sun. Cap: A context-aware neural predictor for nas. In *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence (IJCAI-24)*, pp. 4219–4227, 2024. doi: 10.24963/ijcai.2024/466. URL `https://www.ijcai.org/proceedings/2024/466`.

Dayal Singh Kalra, Tianyu He, and Maissam Barkeshli. Universal sharpness dynamics in neural network training: Fixed point analysis, edge of stability, and route to chaos. In *Proceedings of the 13th International Conference on Learning Representations (ICLR)*, 2025. Available at OpenReview: `https://openreview.net/forum?id=VZN0irKnl0`.

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.

Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On large-batch training for deep learning: Generalization gap and sharp minima. *arXiv preprint arXiv:1609.04836*, 2016.

Aaron Klein, Stefan Falkner, Jost Tobias Springenberg, and Frank Hutter. Learning curve prediction with bayesian neural networks. In *International conference on learning representations*, 2022.

Arjun Krishnakumar, Colin White, Arber Zela, Renbo Tu, Mahmoud Safari, and Frank Hutter. Nas-bench-suite-zero: Accelerating research on zero cost proxies. *Advances in Neural Information Processing Systems*, 35:28037–28051, 2022.

Jaehoon Lee, Yasaman Bahri, Roman Novak, Samuel S Schoenholz, Jeffrey Pennington, and Jascha Sohl-Dickstein. Deep neural networks as gaussian processes. *arXiv preprint arXiv:1711.00165*, 2017.

Junghyup Lee and Bumsub Ham. Az-nas: Assembling zero-cost proxies for network architecture search. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 5893–5903, 2024.

Namhoon Lee, Thalaiyasingam Ajanthan, and Philip HS Torr. Snip: Single-shot network pruning based on connection sensitivity. *arXiv preprint arXiv:1810.02340*, 2018.

Hao Li, Zheng Xu, Gavin Taylor, Christoph Studer, and Tom Goldstein. Visualizing the loss landscape of neural nets. *Advances in neural information processing systems*, 31, 2018.

Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *The Journal of Machine Learning Research*, 18 (1):6765–6816, 2017.

Ming Li, Xiangru Wang, Yi Wang, Yuting Chen, and Yixuan Chen. Study-gnn: a novel pipeline for student performance prediction based on multi-topology graph neural networks. *Sustainability*, 14(13):7965, 2022.

Zhihang Li, Teng Xi, Jiankang Deng, Gang Zhang, Shengzhao Wen, and Ran He. Gp-nas: Gaussian process based neural architecture search. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 11933–11942, 2020.

Ming Lin, Pichao Wang, Zhenhong Sun, Hesen Chen, Xiuyu Sun, Qi Qian, Hao Li, and Rong Jin. Zen-nas: A zero-shot nas for high-performance image recognition. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 347–356, 2021.

Marius Lindauer, Katharina Eggensperger, Matthias Feurer, André Biedenkapp, Difan Deng, Carolin Benjamins, Tim Ruhkopf, René Sass, and Frank Hutter. Smac3: A versatile bayesian optimization package for hyperparameter optimization. *J. Mach. Learn. Res.*, 23(54):1–9, 2022.

Hanxiao Liu, Karen Simonyan, and Yiming Yang. Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055*, 2018.

Charles H Martin and Michael W Mahoney. Implicit self-regularization in deep neural networks: Evidence from random matrix theory and implications for learning. *Journal of Machine Learning Research*, 22(165): 1–73, 2021.

David A McAllester. Pac-bayesian model averaging. In *Proceedings of the twelfth annual conference on Computational learning theory*, pp. 164–170, 1999.

Joe Mellor, Jack Turner, Amos Storkey, and Elliot J Crowley. Neural architecture search without training. In *International conference on machine learning*, pp. 7588–7598. PMLR, 2021.

Behnam Neyshabur, Ryota Tomioka, and Nathan Srebro. Norm-based capacity control in neural networks. In *Conference on learning theory*, pp. 1376–1401. PMLR, 2015.

Behnam Neyshabur, Srinadh Bhojanapalli, and Nathan Srebro. A pac-bayesian approach to spectrally-normalized margin bounds for neural networks. *arXiv preprint arXiv:1707.09564*, 2017.

Hieu Pham, Melody Guan, Barret Zoph, Quoc Le, and Jeff Dean. Efficient neural architecture search via parameters sharing. In *International conference on machine learning*, pp. 4095–4104. PMLR, 2018.

Robin Ru, Clare Lyle, Lisa Schut, Miroslav Fil, Mark van der Wilk, and Yarin Gal. Speedy performance estimation for neural architecture search. *Advances in Neural Information Processing Systems*, 34:4079–4092, 2021.

Han Shi, Renjie Pi, Hang Xu, Zhenguo Li, James Kwok, and Tong Zhang. Bridging the gap between sample-based and one-shot neural architecture search with bonas. *Advances in Neural Information Processing Systems*, 33:1808–1819, 2020.

Hidenori Tanaka, Daniel Kunin, Daniel L Yamins, and Surya Ganguli. Pruning neural networks without any data by iteratively conserving synaptic flow. *Advances in neural information processing systems*, 33: 6377–6389, 2020.

Colin White, Arber Zela, Robin Ru, Yang Liu, and Frank Hutter. How powerful are performance predictors in neural architecture search? *Advances in Neural Information Processing Systems*, 34:28454–28469, 2021.

Jiamin Xiao, Kuoyong Yu, Bo Zhao, and Derong Liu. Evolutionary neural architecture search with performance predictor based on hybrid encodings. In *2024 14th International Conference on Information Science and Technology (ICIST)*, pp. 854–859. IEEE, 2024.

Yu Xue, Chen Chen, and Adam Słowik. Neural architecture search based on a multi-objective evolutionary algorithm with probability stack. *IEEE Transactions on Evolutionary Computation*, 27(4):778–786, 2023.

Peng Ye, Baopu Li, Yikang Li, Tao Chen, Jiayuan Fan, and Wanli Ouyang. b-darts: Beta-decay regularization for differentiable architecture search. In *proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 10874–10883, 2022.

Tunhou Zhang, Dehua Cheng, Yuchen He, Zhengxing Chen, Xiaoliang Dai, Liang Xiong, Feng Yan, Hai Li, Yiran Chen, and Wei Wen. Nasrec: weight sharing neural architecture search for recommender systems. In *Proceedings of the ACM Web Conference 2023*, pp. 1199–1207, 2023.

Appendix

A Pipeline Overview

This appendix details the end-to-end pipeline of NAP2, visualizing the data transformations and tensor dimensions from raw architecture to performance prediction. The process, illustrated in Figure 5, consists of four stages.

- **Stage 1: Architecture Training & Snapshotting**. The neural architecture is trained for a limited number of steps (T' snapshots). At each snapshot t, we extract the model's weights ($\mathbf{W}_t$) and gradients ($\mathbf{G}_t$).

- **Stage 2: Feature Map Construction**. For each snapshot, we compute 12 statistical meta-features (e.g., mean, variance, quantiles) for sampled units across the network's layers. As shown in Figure 5, we sample $S = 99$ units per layer (plus one for the layer index, totaling 100 values). This results in a structured **Feature Map (FM)** for weights and gradients, with dimensions:

$$\mathrm{FM}_t \in \mathbb{R}^{12\times 65\times 100} \tag{3}$$

  representing (12 features, 65 layers, 100 values).

- **Stage 3: Feature Compression (CAE Encoders)**. To reduce dimensionality, the feature maps are passed through pre-trained Convolutional Autoencoder (CAE) encoders. We use separate encoders for weights (CAE_W) and gradients (CAE_G). Each encoder maps the high-dimensional feature map to a compact embedding:

$$\mathbf{e}_{W_t}, \mathbf{e}_{G_t} \in \mathbb{R}^{128} \tag{4}$$

  These embeddings are concatenated to form a fused representation $\mathbf{z}_t \in \mathbb{R}^{256}$ for step t.

- **Stage 4: Temporal Modeling & Prediction**. The sequence of fused embeddings $\mathbf{Z} = (\mathbf{z}_1, \ldots, \mathbf{z}_{T'})$ is fed into an LSTM predictor with a hidden size of 2048. The final hidden state $\mathbf{h}'_{T'}$ is processed by a dense head (Linear $\rightarrow$ ReLU $\rightarrow$ Linear $\rightarrow$ Sigmoid) to output the predicted performance score $\hat{y} \in [0, 1]$.

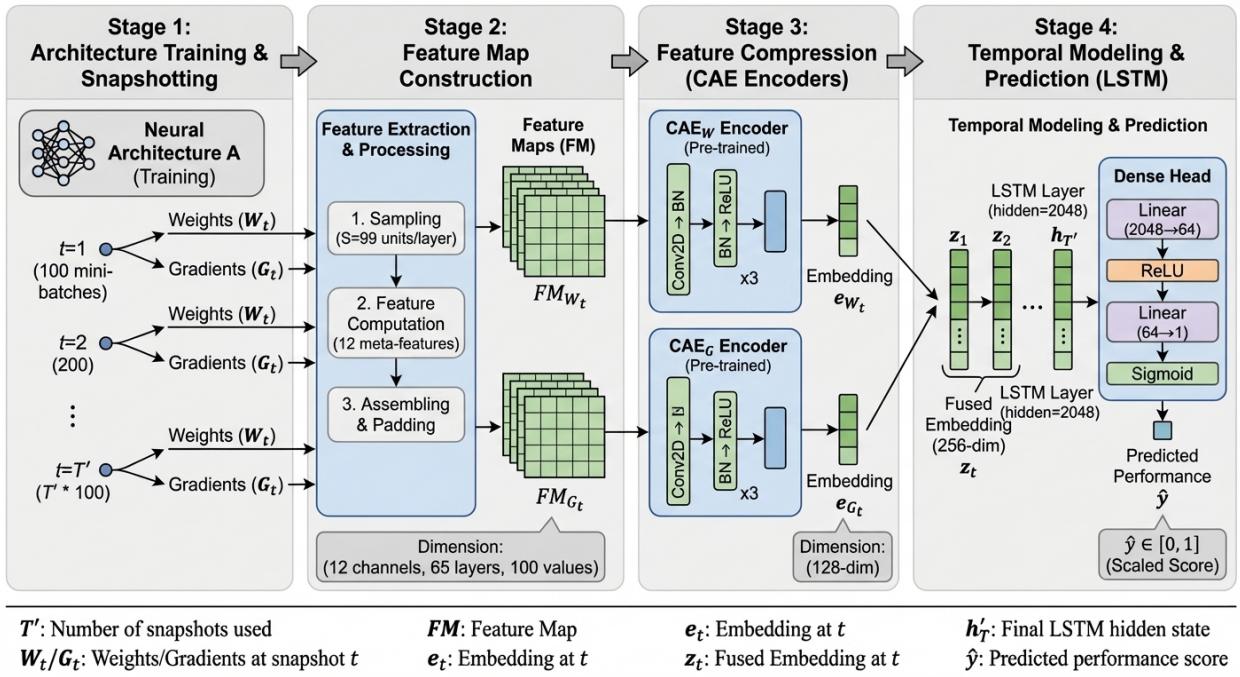


Figure 5: High-level overview of the NAP2 pipeline, detailing the four stages: (1) Partial training and snapshotting, (2) Construction of feature maps from weights and gradients, (3) Compression of feature maps using pre-trained CAE encoders into 128-dimensional embeddings, and (4) Temporal modeling using an LSTM to predict the final performance score.

B Feature Definitions & Notation

At each step of our proposed approach, we extract features from each layer in the analyzed network during its early training stages. These features capture the evolution of weights and gradients, which serve as proxies for the network's learning dynamics and ultimately its final performance. We now expand on the various features and the way they are generated.

Our features consist of two groups, each containing 12 meta-features. One group is computed from the network's weights at each training snapshot, and the other from the gradients. These features are computed independently for weights and gradients, allowing us to capture both the evolving parameter state and the optimization dynamics simultaneously. We describe each feature in detail below, explaining what it measures, how it is computed, and why it is informative for performance prediction.

B.1 Overview of the Feature Extraction Process

During training, we log weights and gradients every 100 mini-batches, capturing $T = 23$ snapshots of the network's state. For each snapshot t and each layer ℓ, we extract 12 meta-features from the weights $\mathbf{W}_{\ell,t}$ and 12 from the gradients $\mathbf{G}_{\ell,t}$.

To handle the varying sizes of layers across different architectures, we employ a sampling strategy: for layers with more than $S = 99$ units (neurons or filters), we deterministically sample exactly S units using a fixed seed. This ensures reproducibility while maintaining computational tractability. We then compute the 12 features over these sampled values, combining both a global statistic computed over all sampled units and per-unit statistics that capture layer heterogeneity.

The resulting feature vectors are assembled into fixed-shape feature maps. Each snapshot produces a feature map with shape $(L, V, F) = (65, 100, 12)$, where L is the maximum number of layers, V is the values dimension storing multiple values per feature, and F is the number of feature types. We capture $T = 23$

such snapshots over time. This uniform representation enables us to train a single predictor model across diverse architectures while preserving the temporal evolution of layer-wise statistics.

B.2 Motivation and Rationale

Why these features? The selected meta-features jointly capture *scale* (min/max/mean), *dispersion* (variance/std), *shape* (skewness/kurtosis), *energy* (L1/L2 norms), and *robust order statistics* (quantiles). This balances sensitivity to training dynamics with robustness to outliers and initialization noise. Gradients reflect optimization signals; weights reflect the evolving representation. Their combination improves early predictive power and transferability across datasets.

Why fixed-shape maps? NAS architectures vary in depth and layer width. To train one predictor, we require a uniform input shape. We therefore map per-layer statistics into fixed-size feature maps with consistent axes and padding/truncation policies. This enables a single encoder to learn invariant patterns of training dynamics across heterogeneous architectures.

B.3 Notation

- $\mathbf{W}_{\ell,t} \in \mathbb{R}^{d_\ell}$: Vectorized weight parameters of layer ℓ at snapshot t (after sampling)
- $\mathbf{G}_{\ell,t} \in \mathbb{R}^{d_\ell}$: Vectorized gradient parameters of layer ℓ at snapshot t (after sampling)
- d_ℓ: Effective dimensionality of layer ℓ after sampling S units/filters
- $L = 65$: Maximum number of layers (architectures with < 65 layers are zero-padded)
- $T = 23$: Total number of training snapshots (one per $\Delta t = 100$ mini-batches)
- $S = 99$: Number of units/filters sampled per layer for computational efficiency
- $F = 12$: Number of meta-features extracted per layer per time step

B.4 Sampling Strategy

Neural architectures vary significantly in layer widths. Some convolutional layers contain hundreds of filters, while dense layers may have thousands of neurons. Computing features over all units would be computationally expensive and create variable-sized feature maps that cannot be processed uniformly.

We therefore sample a fixed number of units per layer. If a layer contains more than $S = 99$ units, we uniformly sample exactly S units using deterministic indexing based on a fixed seed. This ensures that the same layer produces identical samples across different runs, enabling reproducibility. For convolutional layers, we sample S filters and compute features over their flattened kernel weights. For dense layers, we sample S neurons and compute features over their weight vectors. Layers with $\leq S$ units use all available values.

$$\text{If } |\mathbf{W}_\ell| > S: \quad \mathbf{W}_{\ell,t} \leftarrow \text{sample}(\mathbf{W}_{\ell,t}, S, \text{seed} = 42) \tag{5}$$

We choose $S = 99$ because it provides a good balance between computational efficiency and representativeness. Combining one global statistic (computed over all sampled units) with 99 per-unit values gives us 100 values per feature per layer, which matches our feature map design where the values dimension is set to approximately 100. This sampling strategy allows us to capture both the overall layer behavior and its internal heterogeneity, providing enough detail to characterize the layer's state while remaining computationally manageable.

B.5 The Twelve Meta-Features

For notational clarity, we use $\mathbf{x} \in \mathbb{R}^d$ to denote either $\mathbf{W}_{\ell,t}$ or $\mathbf{G}_{\ell,t}$ in the following definitions. Each feature is computed independently for weights and gradients, yielding 12 weight-based features and 12 gradient-based features per layer per snapshot.

We organize our 12 features into several categories: central tendency features that capture the typical value, dispersion features that measure spread and range, distribution shape features that characterize asymmetry and tail behavior, norm features that quantify magnitude, and quantile features that provide robust order statistics.

B.5.1 Central Tendency Features

Mean (f_1): The arithmetic mean captures the average value of the weights or gradients in the layer. This feature reflects the typical scale of parameters and their updates.

$$f_1(\mathbf{x}) = \frac{1}{d}\sum_{j=1}^{d} x_j \tag{6}$$

Median (f_2): The median provides a robust measure of central tendency that is less sensitive to outliers than the mean. This feature helps characterize the typical parameter value even when the distribution is skewed or contains extreme values.

$$f_2(\mathbf{x}) = \text{median}(\{x_1, \ldots, x_d\}) \tag{7}$$

B.5.2 Dispersion Features

Variance (f_3): Variance measures the spread of values around the mean. Higher variance indicates more diverse parameter values or gradient magnitudes, which may signal different learning dynamics.

$$f_3(\mathbf{x}) = \frac{1}{d}\sum_{j=1}^{d}(x_j - \bar{x})^2, \quad \bar{x} = f_1(\mathbf{x}) \tag{8}$$

Standard Deviation (f_4): The standard deviation is the square root of variance, providing a measure of spread in the same units as the original values. It complements variance and is used in the normalization of higher-order moments.

$$f_4(\mathbf{x}) = \sqrt{f_3(\mathbf{x})} \tag{9}$$

Minimum (f_5): The minimum value indicates the lower bound of the distribution. For weights, this captures the smallest parameter magnitude; for gradients, it reflects the smallest update magnitude in the layer.

$$f_5(\mathbf{x}) = \min_{j\in\{1,\ldots,d\}} x_j \tag{10}$$

Maximum (f_6): The maximum value indicates the upper bound of the distribution. This feature, together with the minimum, provides the range of values and helps identify extreme parameter values or gradient spikes.

$$f_6(\mathbf{x}) = \max_{j\in\{1,\ldots,d\}} x_j \tag{11}$$

B.5.3 Distribution Shape Features

Skewness (f_7): Skewness measures the asymmetry of the distribution. Positive skewness indicates a long tail on the right, while negative skewness indicates a long tail on the left. The distribution of weights and gradients often exhibits asymmetry, which relates to the curvature and geometry of the loss landscape.

$$f_7(\mathbf{x}) = \frac{1}{d}\sum_{j=1}^{d}\left(\frac{x_j - \bar{x}}{\sigma}\right)^3, \quad \sigma = f_4(\mathbf{x}) \tag{12}$$

Kurtosis (f_8): Kurtosis measures the heaviness of the distribution's tails compared to a normal distribution. Higher kurtosis indicates heavier tails and more extreme values. We compute excess kurtosis (subtracting 3) following the SciPy convention, so a normal distribution has kurtosis of 0.

$$f_8(\mathbf{x}) = \frac{1}{d}\sum_{j=1}^{d}\left(\frac{x_j - \bar{x}}{\sigma}\right)^4 - 3 \tag{13}$$

B.5.4 Covariance Feature

Covariance (f_9): We compute covariance as the unbiased sample variance using Bessel's correction. This is effectively the covariance of the parameter values with themselves, providing a measure of variability that accounts for the sample size. Mathematically, this is $\mathrm{Cov}(\mathbf{x}, \mathbf{x})$ using $(d-1)$ in the denominator rather than d.

$$f_9(\mathbf{x}) = \frac{1}{d-1}\sum_{j=1}^{d}(x_j - \bar{x})^2 = \frac{d}{d-1} \cdot f_3(\mathbf{x}) \tag{14}$$

Note that this differs from temporal covariance (covariance across time steps) or cross-layer covariance (covariance between different layers), which are not computed here.

B.5.5 Norm Features

L1 Norm (f_{10}): The L1 norm (Manhattan distance) sums the absolute values of all parameters. This measures the total magnitude of weights or gradients without squaring, making it less sensitive to large outliers than the L2 norm.

$$f_{10}(\mathbf{x}) = \sum_{j=1}^{d}|x_j| \tag{15}$$

L2 Norm (f_{11}): The L2 norm (Euclidean distance) measures the magnitude of the weight or gradient vector. This is a fundamental quantity in optimization, as it appears in weight decay regularization and gradient clipping strategies. Larger L2 norms may indicate sharper loss landscapes or more aggressive updates.

$$f_{11}(\mathbf{x}) = \sqrt{\sum_{j=1}^{d} x_j^2} = \|\mathbf{x}\|_2 \tag{16}$$

B.5.6 Quantile Feature (Multi-valued)

Quantiles (f_{12}): We compute five quantiles to capture the distribution shape: the minimum (0th percentile), first quartile (25th percentile), median (50th percentile), third quartile (75th percentile), and maximum (100th percentile). Together, these form the standard five-number summary, with the 25th and 75th percentiles capturing the interquartile range of the distribution.

$$f_{12}(\mathbf{x}) = \begin{bmatrix} q_0(\mathbf{x}) \\ q_{25}(\mathbf{x}) \\ q_{50}(\mathbf{x}) \\ q_{75}(\mathbf{x}) \\ q_{100}(\mathbf{x}) \end{bmatrix} = \begin{bmatrix} f_5(\mathbf{x}) \\ \text{Q1} \\ f_2(\mathbf{x}) \\ \text{Q3} \\ f_6(\mathbf{x}) \end{bmatrix} \tag{17}$$

Although this feature returns 5 values, we conceptually count it as a single feature in our $F = 12$ count, as the quantiles together characterize a single aspect of the distribution (its order statistics).

B.6 Feature Map Construction

After extracting the 12 features for each layer at each snapshot, we organize them into feature maps with a fixed shape that can be processed uniformly by our encoder models. This section explains how we construct these feature maps and the rationale behind our design choices.

At each snapshot t, for each layer ℓ, we compute feature vectors where each feature f_i may contain multiple values (stored in dimension V). Conceptually, we extract 12 feature types per layer per snapshot. These features are then assembled into feature maps:

Each snapshot produces a feature map with shape (L, V, F), where for each of the L layers and $F = 12$ features, we store $V = 100$ values. Over $T = 23$ snapshots, we obtain sequences of feature maps that are processed temporally by the LSTM predictor.

Why fixed-shape maps? NAS architectures vary significantly in depth (number of layers) and width (number of units per layer). To train a single predictor model that works across all architectures, we need a uniform input representation. By mapping all architectures to fixed-size feature maps with shape $(L, V, F) = (65, 100, 12)$ per snapshot (captured over $T = 23$ snapshots), we enable our encoder to learn patterns that generalize across diverse architectures while preserving the temporal evolution captured in the sequence of snapshots.

Depth dimension ($L = 65$): We set the maximum number of layers to 65, chosen as a balance between capturing sufficient depth and maintaining computational efficiency. This value proved effective in our experiments. Architectures with fewer layers are zero-padded at the end. Architectures with more than 65 layers are truncated, keeping only the first 65 layers.

Temporal dimension ($T = 23$): We capture $T = 23$ snapshots during training, one every 100 mini-batches. This spans approximately the first 2,300 mini-batches of training, which corresponds to roughly the first 16 epochs for CIFAR-10/100 (with batch size 256). We found empirically that predictive power plateaus after about 20 snapshots, so $T = 23$ provides sufficient temporal coverage while remaining computationally efficient.

Feature dimension ($F = 12$): We extract exactly 12 meta-features per layer per snapshot, as described above. Each feature represents one aspect of the distribution (e.g., mean, variance, skewness). Multiple values per feature (e.g., the five percentiles from the quantile feature, or values from both global and per-unit computations) are stored in the values dimension V, maintaining $F = 12$ as the number of distinct feature types.

Values dimension ($V \approx 100$): For each feature, we store approximately 100 values: one global statistic computed over all sampled units, plus $S = 99$ per-unit values. This provides a compact yet expressive representation that captures both layer-level summary statistics and unit-level heterogeneity. The values dimension is set to exactly 100 in our implementation, with padding applied when layers have fewer than 99 units.

Padding policy: Missing layers (for architectures with < 65 layers) are zero-padded at the end. Missing unit values (for layers with < 99 units) are also zero-padded to reach the fixed values dimension $V = 100$. Padding is applied after feature extraction but before feature map assembly.

Normalization: We apply normalization through batch normalization layers in the CAE encoder (see Appendix C.1). Each convolutional layer in the encoder is followed by a BatchNorm2d layer, which normalizes activations using batch statistics during training. This approach allows the normalization to adapt to the training distribution of feature maps, automatically handling the varying scales across different feature types (e.g., norms may have vastly different magnitudes than mean/variance).

B.7 Implementation Details

NaN Handling: During training, some layers may contain dead neurons or experience gradient vanishing, leading to NaN values in weights or gradients. All our statistical computations use NaN-aware implementations (e.g., `np.nanmean`, `np.nanstd`) that skip NaN values, ensuring robust feature extraction even when some units are inactive.

Dense vs. Convolutional Layers:

- **Dense layers**: We sample $S = 99$ neurons uniformly at random (with deterministic indexing). For each sampled neuron, we extract its incoming weight vector. Features are computed over all sampled weight values concatenated together.
- **Convolutional layers**: We sample $S = 99$ filters uniformly at random (with deterministic indexing). For each sampled filter, we flatten its kernel weights into a vector. For example, a 3×3 filter with C input channels is flattened to $9C$ values. Features are computed over all sampled filter weights concatenated together.

Global and per-unit computation: As described in the paper, we compute features in two ways: first, a single value over the entire layer's data, and second, values for each individual unit (neuron or filter). Both types of values are collected and stored in the values dimension V for each feature, providing both layer-level summary statistics and unit-level details while maintaining $F = 12$ feature types.

Reproducibility: The sampling process uses fixed seeds and deterministic indexing, ensuring that the same architecture produces identical feature maps across runs. Architecture training also uses fixed seeds and determinism flags (see Appendix E), ensuring that weights and gradients are logged identically. With identical inputs, feature extraction is bitwise-stable across runs on the same platform.

C Model Specifications

This appendix provides complete architectural specifications for the convolutional auto-encoders (CAE) and the LSTM predictor, including layer configurations, hyperparameters, fusion mechanisms, and loss functions.

C.1 Convolutional Auto-Encoder (CAE)

We train two separate CAEs: one for weights feature maps (CAE_W) and one for gradients feature maps (CAE_G). Both share identical architecture but are trained independently on their respective data.

C.1.1 Input Specification

- **Input shape**: $(F, L, V) = (12, 65, 100)$
- $F = 12$: Number of feature channels (meta-features)
- $L = 65$: Number of layers (padded/truncated)
- $V = 100$: Feature map values dimension
- **Data type**: Double precision (`float64`)

C.1.2 Encoder Architecture

The encoder consists of three convolutional blocks, each comprising Conv2D, BatchNorm2D, and ReLU activation.

Table 11: CAE Encoder Layer Specifications

Layer	**Type**	**In Ch.**	**Out Ch.**	**Kernel**	**Stride**	**Padding**
Input	–	–	12	–	–	–
Conv2D-1	Conv2D	12	512	$(1, 100)$	$(1, 1)$	$(0, 0)$
BatchNorm-1	BatchNorm2D	512	512	–	–	–
Activation-1	ReLU	512	512	–	–	–
Conv2D-2	Conv2D	512	256	$(65, 1)$	$(1, 1)$	$(0, 0)$
BatchNorm-2	BatchNorm2D	256	256	–	–	–
Activation-2	ReLU	256	256	–	–	–
Conv2D-3	Conv2D	256	128	$(1, 1)$	$(1, 1)$	$(0, 0)$
BatchNorm-3	BatchNorm2D	128	128	–	–	–
Activation-3	ReLU	128	128	–	–	–
Output	–	–	128	$(1, 1, 128)$	–	–

Output shape after encoder: $(128, 1, 1) \rightarrow$ flattened to $\mathbb{R}^{128}$ (embedding dimension).

C.1.3 Decoder Architecture

The decoder mirrors the encoder structure using transposed convolutions to reconstruct the input.

C.1.4 CAE Training Configuration

- **Loss function**: Mean Squared Error (MSE) reconstruction loss with sum reduction

$$\mathcal{L}_{\text{CAE}} = \sum_{f,\ell,v} \left(\mathbf{X}_{f,\ell,v} - \hat{\mathbf{X}}_{f,\ell,v} \right)^2 \tag{18}$$

- **Optimizer**: Adam with learning rate 0.001

Table 12: CAE Decoder Layer Specifications

Layer	Type	In Ch.	Out Ch.	Kernel	Stride	Padding
Input	–	–	128	$(1, 1, 128)$	–	–
ConvT2D-1	ConvTranspose2D	128	256	$(1, 1)$	$(1, 1)$	$(0, 0)$
BatchNorm-1	BatchNorm2D	256	256	–	–	–
Activation-1	ReLU	256	256	–	–	–
ConvT2D-2	ConvTranspose2D	256	512	$(65, 1)$	$(1, 1)$	$(0, 0)$
BatchNorm-2	BatchNorm2D	512	512	–	–	–
Activation-2	ReLU	512	512	–	–	–
ConvT2D-3	ConvTranspose2D	512	12	$(1, 100)$	$(1, 1)$	$(0, 0)$
Output	–	–	12	$(12, 65, 100)$	–	–

- **Learning rate schedule**: StepLR with step size 20 and gamma 0.15, preceded by a 5-epoch warmup from 10^{-6} to 0.001
- **Batch size**: 32
- **Epochs**: 150
- **Early stopping**: Patience of 25 epochs on validation loss (or training loss if no validation set)
- **Weight initialization**: PyTorch default (Kaiming for Conv2D)

C.2 LSTM Predictor

The LSTM predictor takes a sequence of fused embeddings and outputs a scalar performance score.

C.2.1 Input Specification

- **Input shape**: $(T', D) = (T', 256)$
- $T' \in \{1, 2, \ldots, 10\}$: Sequence length during training (variable for augmentation). During inference, the full sequence up to $T = 23$ snapshots can be used.
- $D = 256$: Fused embedding dimension (128 from CAE_W + 128 from CAE_G)
- **Batch-first format**: Tensor shape is (B, T', D) where B is batch size

C.2.2 Architecture Specification

C.2.3 LSTM Forward Pass

The forward computation is defined as:

$$\mathbf{h}_t, \mathbf{c}_t = \text{LSTM}(\mathbf{z}_t, \mathbf{h}_{t-1}, \mathbf{c}_{t-1}), \quad t = 1, \ldots, T' \tag{19}$$

$$\mathbf{h}_{T'} \in \mathbb{R}^{2048} \quad \text{(final hidden state)} \tag{20}$$

$$\mathbf{o}_1 = \text{ReLU}(\mathbf{W}_1 \mathbf{h}_{T'} + \mathbf{b}_1), \quad \mathbf{W}_1 \in \mathbb{R}^{64 \times 2048} \tag{21}$$

$$\hat{y} = \sigma(\mathbf{W}_2 \mathbf{o}_1 + \mathbf{b}_2), \quad \mathbf{W}_2 \in \mathbb{R}^{1 \times 64} \tag{22}$$

where $\sigma(\cdot)$ is the sigmoid function ensuring $\hat{y} \in [0, 1]$. For in-domain experiments where targets are in $[0, 100]$, the model output is scaled accordingly. For cross-dataset experiments with normalized targets in $[0, 1]$, the output matches the target range directly.

Table 13: LSTM Predictor Layer Specifications

Layer	**Type**	**Input Dim**	**Output Dim**
Input	–	$(B, T', 256)$	$(B, T', 256)$
LSTM	LSTM	256	2048
	hidden_size=2048, *num_layers*=1		
	bidirectional=False, *batch_first*=True		
Select Last	Indexing	$(B, T', 2048)$	$(B, 2048)$
Dense-1	Linear	2048	64
Activation-1	ReLU	64	64
Dense-2	Linear	64	1
Activation-2	Sigmoid	1	1
Output	–	$(B, 1)$	$\hat{y} \in [0, 1]$

C.2.4 LSTM Training Configuration

- **Loss function**: Mean Absolute Error (MAE)

$$\mathcal{L}_{\text{LSTM}} = \frac{1}{B} \sum_{i=1}^{B} |\hat{y}_i - y_i| \tag{23}$$

  where y_i is the true accuracy of architecture i. For in-domain experiments, $y_i \in [0, 100]$ (raw percentage). For cross-dataset experiments, $y_i \in [0, 1]$ (min-max normalized). See §C.6 for details on target normalization.

- **Alternative loss** (explored): Rank Consistency Loss (see §C.5)
- **Optimizer**: Adam with learning rate 0.001
- **Learning rate schedule**: StepLR with step size 20 and gamma 0.15, preceded by a 5-epoch warmup from 10^{-6} to 0.001
- **Batch size**: 32
- **Epochs**: 150
- **Sequence augmentation**: Train on lengths $T' \in \{1, \ldots, 10\}$ with same target y

C.3 Fusion Mechanism

At each time step t, embeddings from both CAEs are fused via concatenation:

$$\mathbf{z}_t = \begin{bmatrix} \mathbf{e}_t^W \\ \mathbf{e}_t^G \end{bmatrix} \in \mathbb{R}^{256} \tag{24}$$

where:

- $\mathbf{e}_t^W = \text{CAE}_W(\mathbf{X}_t^W) \in \mathbb{R}^{128}$: Weights embedding at time t
- $\mathbf{e}_t^G = \text{CAE}_G(\mathbf{X}_t^G) \in \mathbb{R}^{128}$: Gradients embedding at time t
- No additional learned projection is applied (simple concatenation)

The full sequence is then constructed:

$$\mathbf{Z} = [\mathbf{z}_1, \mathbf{z}_2, \ldots, \mathbf{z}_{T'}] \in \mathbb{R}^{T' \times 256} \tag{25}$$

C.4 Alternative Architectures Explored

C.4.1 Combined LSTM (Separate Encoders)

An alternative architecture uses two separate LSTMs for weights and gradients, fusing only at the final hidden states:

$$\mathbf{h}_{T'}^{W} = \text{LSTM}_{W}(\mathbf{e}_{1}^{W}, \ldots, \mathbf{e}_{T'}^{W}) \tag{26}$$

$$\mathbf{h}_{T'}^{G} = \text{LSTM}_{G}(\mathbf{e}_{1}^{G}, \ldots, \mathbf{e}_{T'}^{G}) \tag{27}$$

$$\mathbf{h}_{\text{fused}} = [\mathbf{h}_{T'}^{W}; \mathbf{h}_{T'}^{G}] \in \mathbb{R}^{4096} \tag{28}$$

This variant (class `CombinedLSTMRegressor` in code) was explored but not used in final experiments.

C.5 Loss Functions

C.5.1 Primary Loss: Mean Absolute Error (MAE)

The main training objective for the LSTM predictor:

$$\mathcal{L}_{\text{MAE}} = \frac{1}{B} \sum_{i=1}^{B} |\hat{y}_i - y_i| \tag{29}$$

Rationale: MAE is less sensitive to outliers than MSE and directly optimizes for ranking quality. For cross-dataset experiments, MAE is applied to normalized targets, ensuring the model focuses on relative rankings rather than absolute values.

C.5.2 Experimental Loss: Rank Consistency

A custom ranking loss based on soft Spearman correlation was explored:

$$\mathcal{L}_{\text{rank}} = 1 - \frac{6 \sum_{i=1}^{B} (\text{rank}(\hat{y}_i) - \text{rank}(y_i))^2}{B(B^2 - 1)} \tag{30}$$

where ranks are computed via soft-ranking approximation to maintain differentiability. This loss did not provide significant improvements over MAE in preliminary experiments.

C.6 Target Normalization

Target normalization depends on the experimental setup:

In-domain experiments: Targets are used in their raw form (accuracy percentage $y_{\text{raw}} \in [0, 100]$). No normalization is applied, as the model learns to predict absolute accuracy values within the same dataset's distribution. This approach is suitable when training and testing on the same dataset, where absolute accuracy values are meaningful and comparable.

Cross-dataset experiments: Targets are normalized using min-max normalization to $[0, 1]$ to account for different accuracy scales across datasets. For cross-dataset transfer, we compute dataset-specific statistics:

$$y = \frac{y_{\text{raw}} - y_{\min}}{y_{\max} - y_{\min}} \tag{31}$$

where $y_{\min}$ and $y_{\max}$ are the minimum and maximum accuracy values in the training set. This normalization ensures that the model focuses on relative ranking (which is what matters for cross-dataset transfer) rather than absolute accuracy values, which can vary significantly across datasets with different complexity (e.g., CIFAR-10 vs. ImageNet-16-120).

C.7 Model Size & Complexity

Table 14: Model Parameter Counts

Component	Parameters	Type
CAE Encoder (W)	9.17M	Trainable
CAE Decoder (W)	9.17M	Trainable
CAE Encoder (G)	9.17M	Trainable
CAE Decoder (G)	9.17M	Trainable
LSTM	19.02M	Trainable
Dense Head	131K	Trainable
Total	55.70M	–

Parameter count details:

- **CAE Encoder**: Three Conv2D layers (12→512, 512→256, 256→128 channels) with BatchNorm2d after each, totaling 9,169,536 parameters per encoder.
- **CAE Decoder**: Three ConvTranspose2D layers (128→256, 256→512, 512→12 channels) with BatchNorm2d after the first two, totaling 9,169,164 parameters per decoder.
- **LSTM**: Single-layer LSTM with input size 256, hidden size 2048, plus two linear layers (2048→64→1), totaling 19,021,953 parameters (18,890,752 for LSTM core + 131,136 for first dense layer + 65 for second dense layer).
- CAEs are trained independently; only encoders are used during LSTM training (decoders are discarded after CAE training).

C.8 Visual Diagrams

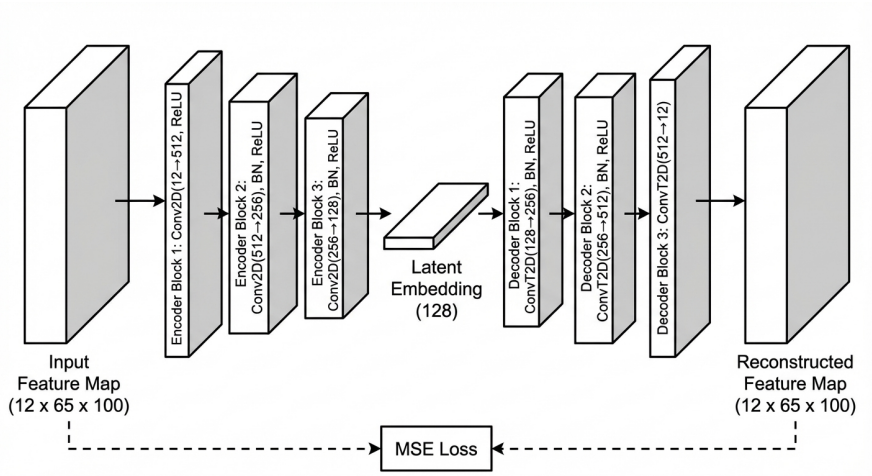


Figure 6: Convolutional auto-encoder (CAE) architecture with three-stage encoder (12→512→256→128) compressing feature maps to 128-dimensional embeddings, trained with MSE reconstruction loss.

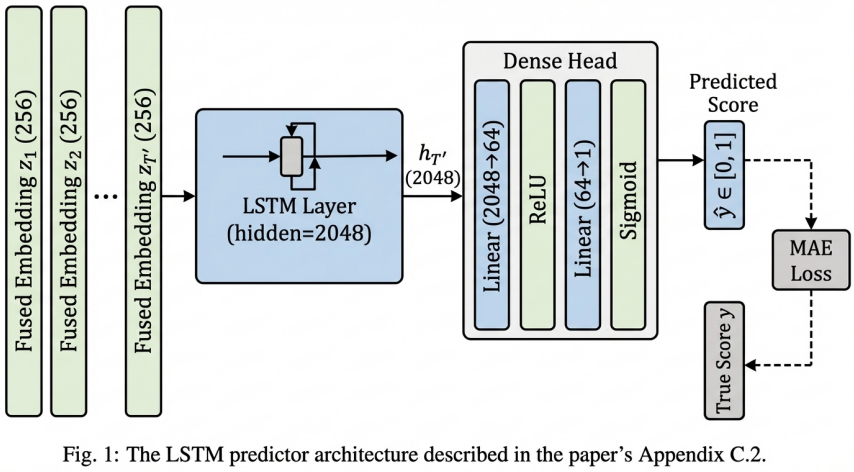


Figure 7: LSTM predictor architecture processing fused embedding sequences through a single-layer LSTM (hidden=2048) followed by a dense head to output performance scores $\hat{y} \in [0, 1]$, trained with MAE loss.

D Data & Preprocessing

This appendix specifies datasets, normalization parameters, and data augmentation used for training NAS-Bench-201 architectures to generate performance logs.

D.1 Datasets

Table 15: Dataset Specifications

Dataset	Size	Classes	Image Res.	Role
CIFAR-10	50k / 10k	10	32×32	Primary
CIFAR-100	50k / 10k	100	32×32	Primary
ImageNet-16-120	151k / 6k	120	16×16	Primary
SVHN	73k / 26k	10	32×32	Dist. shift

D.2 Normalization Parameters

Table 16: Per-Channel Normalization (applied as $(x/255 - \mu)/\sigma$)

Dataset	Mean μ (R, G, B)	Std σ (R, G, B)
CIFAR-10	$(0.4914, 0.4822, 0.4465)$	$(0.2470, 0.2430, 0.2610)$
CIFAR-100	$(0.5071, 0.4867, 0.4408)$	$(0.2675, 0.2565, 0.2761)$
ImageNet-16-120	Mean image subtraction†	
SVHN	$(0.5000, 0.5000, 0.5000)$	$(0.5000, 0.5000, 0.5000)$

†ImageNet-16-120 uses mean image computed from training batches, subtracted before ToTensor conversion. Unlike other datasets, no per-channel normalization is applied after mean subtraction; the images are normalized to $[0, 1]$ via division by 255, then mean-subtracted, and converted to tensors directly.

D.3 Data Augmentation

Table 17: Data Augmentation Applied During Architecture Training

Operation	CIFAR-10/100	ImageNet-16	SVHN
Random Crop + Pad	$32 \times 32 + 4$	$16 \times 16 + 2$	None
Horizontal Flip	$p = 0.5$	$p = 0.5$	None
Normalize	Per-channel	Mean subtraction*	$[-1, 1]$ range

Pipeline:

- **CIFAR-10/100**: RandomHorizontalFlip($p = 0.5$) $\rightarrow$ RandomCrop(padding) $\rightarrow$ ToTensor $\rightarrow$ Normalize(μ, σ)
- **ImageNet-16-120**: RandomHorizontalFlip($p = 0.5$) $\rightarrow$ RandomCrop(padding) $\rightarrow$ ToTensor (mean subtraction applied during data loading)
- **SVHN**: ToTensor $\rightarrow$ Normalize to $[-1, 1]$ range (no augmentation)

*ImageNet-16-120 applies mean image subtraction during data loading (before transforms), not per-channel normalization after ToTensor. See §**??** for details.

D.4 Data Splits

D.4.1 Architecture Training

CIFAR-10 and CIFAR-100: Both datasets consist of 50,000 training samples and 10,000 test samples. Following the NAS-Bench-201 protocol, the full training set (50,000 samples) is used for training. The test set (10,000 samples) is split into a fixed validation subset and remaining test samples. Fixed validation indices are used to ensure reproducibility across all architecture training runs. This approach maintains consistency with the NAS-Bench-201 benchmark while providing a held-out validation set for architecture evaluation during training and a final test set for evaluation.

ImageNet-16-120: The dataset provides 151,700 training samples and approximately 6,000 validation samples as separate pre-processed sets. The validation set is used directly without further splitting.

SVHN: The dataset consists of 73,257 training samples and 26,032 test samples. The training set is split into 51,280 training samples and 21,977 validation samples using fixed indices to ensure reproducibility. The test set (26,032 samples) remains completely separate for final evaluation.

D.4.2 Predictor Training

Architectures (not data samples) are split for CAE/LSTM training:

Table 18: Architecture Splits for Predictor Training

Train	**Validation**	**Test**	**Total**
50–700	100	200	350–1000

Architectures sampled with stratified sampling across performance bins (50-55%, 55-60%, ..., 90-95% accuracy). Splits saved to `train_files.json`, `val_files.json`, `test_files.json`.

D.5 Snapshot Protocol

Weights and gradients logged every $\Delta t = 100$ mini-batches during architecture training, yielding $T = 23$ snapshots over 16 epochs (CIFAR/ImageNet) or 40 epochs (SVHN). Snapshots compressed and saved as `weights_stats_<id>.bz2` and `gradients_stats_<id>.bz2`.

Training Configuration: SGD + Nesterov (momentum=0.9, weight decay=5×10^{-4}), lr=0.1 with cosine decay, batch size=256. See Appendix C for full training protocol.

E Reproducibility & Computational Environment

This appendix documents the experimental design, randomness control, hardware specifications, and software environment to facilitate reproducibility.

E.1 Experimental Design

We evaluate predictor performance across varying training set sizes to assess data efficiency. Each configuration is repeated over multiple independent trials to ensure statistical robustness.

Table 19: Experimental Design: Training Set Sizes and Trials

Train Size	**Trials**	**Total Runs**
50	10	10
100	10	10
200	10	10
300	10	10
400	10	10
500	10	10
600	10	10
700	10	10
Total Configurations	**8**	**80 runs**

Split Configuration: All experiments use fixed validation (100 architectures) and test (200 architectures) sets. Architectures are sampled with stratified sampling across performance bins to ensure balanced coverage of the performance distribution.

Reporting: All results presented as mean ± standard deviation over 10 independent trials.

E.2 Randomness Control

We employ multi-level seeding to balance reproducibility with statistical robustness.

Table 20: Seed Configuration and Determinism Levels

Component	**Seed**	**Deterministic**	**Implementation**
Phase 1: Architecture Training (Log Generation)			
Random/NumPy/PyTorch	4	Full	`set_seed()` with all flags
cuDNN operations	–	Full	`cudnn.deterministic=True`
cuDNN benchmark	–	Off	`cudnn.benchmark=False`
TensorFloat-32	–	Off	`allow_tf32=False`
CUBLAS workspace	–	Set	`CUBLAS_WORKSPACE_CONFIG=':16:8'`
Phase 2: Data Splitting			
Train/Val/Test split	42	Full	`train_test_split(random_state=42)`
Split persistence	–	Yes	Saved to JSON, reused across trials
Phase 3: Predictor Training (CAE + LSTM)			
Weight initialization	None	Stochastic	PyTorch default initialization
Optimizer state	None	Stochastic	Adam with random initial state
Data shuffling	None	Stochastic	DataLoader shuffling
Statistical Robustness Strategy			
Independent trials	–	–	10 trials per configuration
Aggregation	–	–	Report mean ± SD

E.2.1 Determinism Flags (Architecture Training)

Full determinism is enforced during architecture training via `set_seed()` function:

```
random.seed(4)
np.random.seed(4)
torch.manual_seed(4)
torch.cuda.manual_seed_all(4)
os.environ['CUBLAS_WORKSPACE_CONFIG'] = ':16:8'
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False
torch.backends.cudnn.allow_tf32 = False
```

Rationale: Architecture training must be deterministic to ensure identical weights/gradients logs across predictor training trials. Predictor training (CAE/LSTM) uses stochastic initialization with multiple trials for statistical robustness.

E.3 Hardware Specifications

Table 21: Computational Hardware

Component	Specification	Details
GPU	NVIDIA GeForce GTX 1080 Ti	–
- Memory	11 GB GDDR5X	–
- Architecture	Pascal (GP102)	Compute Cap. 6.1
- TDP	250W	–
CPU	Intel Xeon E5-1650 v4	–
- Base Frequency	3.60 GHz	–
- Cores / Threads	6 / 12	–
Memory	125 GB DDR4	–
Cluster	Slurm-managed HPC	University cluster
- Partitions	main, gtx1080, rtx2080, etc.	–

GPU Selection: The GTX 1080 Ti was chosen to match prior NAS predictor work (White et al., 2021), enabling direct timing comparisons. Query times reported in the main paper are measured on this hardware.

E.4 Software Environment

E.5 Training Time & Computational Cost

Note: Times are approximate and vary with training set size. Reported query times in main paper (e.g., 54s, 108s, 207s) refer to architecture training only and are measured on GTX 1080 Ti.

E.6 Code Availability

All code, trained models, and data splits are available at: [repository URL to be added]. The repository includes:

- Architecture training scripts with snapshot logging
- Statistics extraction and feature map creation
- CAE and LSTM training code with all hyperparameters
- Pre-computed data splits (JSON files) for exact reproducibility

Table 22: Software Versions

Software	**Version**
Operating System	Rocky Linux 9.5 (Blue Onyx)
Kernel	5.14.0-503.40.1.el9_5.x86_64
Python	3.9.21
PyTorch	$\geq$ 1.13
CUDA (Driver)	12.5 (Driver 555.42.06)
cuDNN	$\geq$ 8.0
Core Libraries	
NumPy	2.0.2
SciPy	$\geq$ 1.10
Pandas	$\geq$ 1.5
Scikit-learn	$\geq$ 1.0
Infrastructure	
Cluster Manager	Slurm
Conda Environment	Custom environment per component

Table 23: Approximate Training Times (per trial, GTX 1080 Ti)

Component	**Time per Architecture**	**Time per Trial**
Architecture Training (16 epochs)	3-4 minutes	–
Snapshot T=23	Included above	–
Statistics Extraction	30 seconds	–
Feature Map Creation	10 seconds	–
CAE Training (150 epochs)	–	2-4 hours
LSTM Training (150 epochs)	–	1-2 hours
Total per Trial	–	5-8 hours (train_size=700)
Total for 80 Experiments	–	400-640 GPU hours

- Evaluation scripts and analysis notebooks
- Slurm job scripts for cluster deployment

F Top-K Identification Analysis

This appendix provides a comprehensive analysis of NAP2's ability to identify top-performing architectures through Top-K recall and Top-N Coverage metrics. While Kendall's Tau measures overall ranking quality, these metrics directly address practical NAS questions: *Can NAP2 reliably identify the best architectures early in training, and how many promising candidates can it identify simultaneously?*

F.1 Metric Definitions

Top-K Recall: For each trial, we compute the probability that the true best architecture (the single architecture with highest final accuracy, i.e., true top-1) appears within the predicted top-K architectures (ranked by predicted score). Formally:

$$\text{Top-}K \text{ Recall} = \frac{1}{T}\sum_{t=1}^{T} \mathbb{1}[\text{true top-1} \in \text{predicted top-}K] \tag{32}$$

where $T = 10$ is the number of trials (200 architectures per trial). Values range from 0 (never found) to 1 (always found).

Top-N Coverage: Measures the average proportion of true top-N architectures that appear in predicted top-K rankings. For each trial, we count how many of the true top-N architectures appear in predicted top-K, then divide by N to get the proportion. Values are averages of these proportions across trials, ranging from 0.0 (none found) to 1.0 (all found).

The distinction is crucial for NAS: Top-K Recall answers whether the single best architecture is found (binary), while Top-N Coverage reveals NAP2's ability to identify multiple promising candidates simultaneously (continuous)—enabling parallel exploration and robust ensemble evaluation in NAS workflows.

F.2 In-Domain Results

Table 24 reports Top-K recall for in-domain setups. NAP2 achieves strong Top-K recall even at early query times, with Top-20 recall exceeding 0.7 by step 14-16 (1,400-1,600 mini-batches) for CIFAR-10/100, and Top-10 recall exceeding 0.6 by step 12 across all datasets.

Tables 25–27 report Top-N Coverage metrics, measuring the average proportion of true top-N architectures found in predicted top-K rankings. Column labels follow the pattern "Top-N in Top-K": e.g., "Top-3 in Top-20" means the proportion of true top-3 architectures found in predicted top-20.

F.3 Cross-Dataset Results

Table 28 reports Top-K recall for cross-dataset setups where models are trained on CIFAR-10 and tested on CIFAR-100 and ImageNet-16-120. Results show that NAP2 achieves strong Top-K recall even when trained on a smaller dataset and tested on larger ones, demonstrating effective transferability. Top-20 recall reaches 0.6–0.9 by step 3–4 (300-400 mini-batches) for CIFAR-100 and 0.5–0.7 by step 3–4 for ImageNet-16-120.

F.4 Key Insights & Practical Implications

Main Findings:

1. **Early Identification**: Top-20 recall exceeds 0.6 within 200-500 mini-batches, enabling early search space narrowing in NAS frameworks
2. **Cross-Dataset Transfer**: Tables 28–30 show strong Top-K recall when trained on different datasets, demonstrating NAP2's transferability without retraining. Notably, CIFAR-100 → ImageNet achieves 1.0 Top-20 recall by step 10, and ImageNet → CIFAR-100 reaches 0.8–0.9 Top-20 recall by step 3–4

Table 24: Top-K recall for in-domain datasets (train_size=700). Values represent the probability that the true best architecture appears in NAP2's predicted top-K rankings, averaged across 10 trials.

mini-batch step ($\times 10^2$)	CIFAR-10				CIFAR-100				ImageNet-16-120			
	Top-3	Top-5	Top-10	Top-20	Top-3	Top-5	Top-10	Top-20	Top-3	Top-5	Top-10	Top-20
1	0.0	0.0	0.0	0.1	0.2	0.3	0.4	0.4	0.1	0.1	0.1	0.2
2	0.1	0.1	0.1	0.3	0.1	0.1	0.3	0.7	0.1	0.1	0.2	0.4
3	0.1	0.1	0.2	0.4	0.1	0.2	0.4	0.7	0.1	0.2	0.3	0.4
4	0.1	0.1	0.2	0.3	0.0	0.1	0.4	0.6	0.2	0.2	0.4	0.5
5	0.1	0.1	0.2	0.4	0.1	0.1	0.5	0.8	0.2	0.3	0.4	0.5
6	0.1	0.1	0.2	0.3	0.0	0.1	0.5	0.8	0.2	0.3	0.4	0.5
7	0.1	0.1	0.2	0.4	0.1	0.2	0.5	0.8	0.3	0.4	0.5	0.6
8	0.1	0.1	0.2	0.3	0.1	0.3	0.5	1.0	0.4	0.5	0.5	0.6
9	0.1	0.1	0.2	0.4	0.1	0.1	0.7	1.0	0.4	0.5	0.5	0.7
10	0.1	0.1	0.3	0.3	0.0	0.1	0.7	1.0	0.4	0.4	0.5	0.7
11	0.1	0.1	0.3	0.4	0.1	0.1	0.8	1.0	0.4	0.4	0.6	0.7
12	0.1	0.1	0.3	0.6	0.0	0.2	0.8	1.0	0.4	0.4	0.6	0.7
13	0.1	0.1	0.3	0.6	0.1	0.2	0.8	1.0	0.4	0.4	0.7	0.7
14	0.1	0.2	0.4	0.8	0.1	0.4	0.8	1.0	0.4	0.4	0.6	0.7
15	0.0	0.2	0.5	0.8	0.1	0.2	0.8	1.0	0.4	0.4	0.6	0.7
16	0.0	0.2	0.5	0.9	0.2	0.3	0.8	1.0	0.4	0.5	0.6	0.7
17	0.0	0.2	0.5	0.9	0.2	0.3	0.7	0.9	0.4	0.5	0.6	0.7
18	0.0	0.2	0.5	0.8	0.2	0.3	0.7	0.9	0.4	0.5	0.6	0.7
19	0.0	0.2	0.7	0.9	0.2	0.3	0.7	0.9	0.4	0.5	0.6	0.7
20	0.0	0.1	0.7	0.9	0.2	0.3	0.7	0.9	0.4	0.5	0.6	0.7
21	0.0	0.1	0.7	0.9	0.2	0.3	0.7	0.8	0.4	0.5	0.6	0.7
22	0.0	0.1	0.7	0.9	0.2	0.2	0.7	0.8	0.4	0.5	0.6	0.7
23	0.0	0.1	0.7	0.9	0.2	0.3	0.6	0.8	0.4	0.5	0.6	0.7

Table 25: Top-N Coverage in Top-K for in-domain CIFAR-10 (train_size=700). Values are averaged across 10 trials.

mini-batch step ($\times 10^2$)	Top-3 in Top-3	Top-3 in Top-5	Top-3 in Top-10	Top-3 in Top-20	Top-5 in Top-5	Top-5 in Top-10	Top-5 in Top-20	Top-10 in Top-10	Top-10 in Top-20
1	0.033	0.067	0.100	0.167	0.100	0.140	0.240	0.150	0.270
2	0.067	0.100	0.233	0.400	0.120	0.240	0.420	0.280	0.520
3	0.067	0.100	0.200	0.433	0.140	0.240	0.460	0.300	0.580
4	0.100	0.167	0.300	0.500	0.180	0.320	0.500	0.390	0.590
5	0.033	0.133	0.333	0.533	0.180	0.380	0.580	0.400	0.640
6	0.067	0.167	0.367	0.600	0.240	0.400	0.640	0.450	0.680
7	0.100	0.233	0.367	0.633	0.280	0.420	0.720	0.440	0.720
8	0.133	0.233	0.367	0.600	0.300	0.440	0.740	0.470	0.730
9	0.167	0.267	0.433	0.667	0.340	0.500	0.760	0.490	0.750
10	0.167	0.300	0.500	0.633	0.360	0.580	0.740	0.550	0.760
11	0.200	0.333	0.500	0.667	0.360	0.560	0.760	0.540	0.790
12	0.233	0.333	0.533	0.767	0.360	0.600	0.820	0.570	0.820
13	0.300	0.333	0.533	0.733	0.380	0.580	0.800	0.560	0.820
14	0.333	0.367	0.533	0.800	0.380	0.600	0.860	0.570	0.850
15	0.267	0.367	0.567	0.767	0.400	0.620	0.840	0.570	0.840
16	0.267	0.367	0.567	0.800	0.400	0.620	0.860	0.560	0.850
17	0.300	0.367	0.600	0.833	0.400	0.620	0.880	0.570	0.850
18	0.300	0.400	0.600	0.800	0.420	0.600	0.880	0.560	0.860
19	0.300	0.400	0.667	0.833	0.420	0.640	0.880	0.580	0.840
20	0.300	0.367	0.667	0.833	0.400	0.640	0.880	0.590	0.830
21	0.267	0.367	0.667	0.833	0.380	0.640	0.860	0.570	0.840
22	0.267	0.367	0.667	0.867	0.360	0.640	0.880	0.570	0.850
23	0.200	0.400	0.667	0.867	0.400	0.640	0.880	0.570	0.840

3. **Multi-Architecture Identification**: Top-N Coverage analysis reveals NAP2 successfully identifies multiple high-performing architectures simultaneously (e.g., Top-3 Coverage in Top-20 reaches 0.80-0.97 by step 16), enabling parallel exploration of promising candidates

4. **Monotonic Improvement**: Both metrics improve with query time, with Top-K recall generally plateauing after 1,000-1,500 mini-batches

Practical Benefits for NAS:

Table 26: Top-N Coverage in Top-K for in-domain CIFAR-100 (train_size=700). Values are averaged across 10 trials.

mini-batch step ($\times 10^2$)	Top-3 in Top-3	Top-3 in Top-5	Top-3 in Top-10	Top-3 in Top-20	Top-5 in Top-5	Top-5 in Top-10	Top-5 in Top-20	Top-10 in Top-10	Top-10 in Top-20
1	0.133	0.167	0.267	0.367	0.120	0.200	0.340	0.180	0.370
2	0.167	0.233	0.433	0.733	0.200	0.340	0.600	0.310	0.550
3	0.300	0.333	0.500	0.767	0.300	0.420	0.660	0.390	0.630
4	0.200	0.333	0.533	0.767	0.320	0.500	0.720	0.480	0.670
5	0.233	0.367	0.567	0.800	0.380	0.500	0.760	0.470	0.670
6	0.233	0.333	0.600	0.833	0.340	0.560	0.800	0.490	0.720
7	0.333	0.400	0.633	0.867	0.360	0.600	0.820	0.520	0.710
8	0.300	0.467	0.633	0.933	0.420	0.640	0.840	0.540	0.720
9	0.300	0.400	0.733	0.933	0.380	0.700	0.840	0.560	0.730
10	0.300	0.433	0.733	0.933	0.440	0.700	0.880	0.560	0.740
11	0.367	0.433	0.733	0.933	0.440	0.700	0.880	0.550	0.730
12	0.333	0.467	0.733	0.933	0.440	0.700	0.900	0.550	0.740
13	0.300	0.433	0.700	0.933	0.420	0.680	0.900	0.550	0.730
14	0.367	0.533	0.700	0.933	0.460	0.660	0.880	0.540	0.730
15	0.367	0.433	0.733	0.933	0.420	0.700	0.880	0.560	0.740
16	0.367	0.467	0.767	0.967	0.440	0.720	0.900	0.570	0.740
17	0.400	0.533	0.733	0.933	0.480	0.680	0.880	0.550	0.740
18	0.400	0.533	0.767	0.900	0.500	0.700	0.860	0.570	0.740
19	0.433	0.533	0.767	0.900	0.480	0.680	0.860	0.570	0.740
20	0.433	0.533	0.767	0.900	0.500	0.700	0.860	0.570	0.740
21	0.433	0.533	0.800	0.867	0.520	0.700	0.840	0.580	0.730
22	0.433	0.500	0.767	0.867	0.460	0.700	0.840	0.560	0.730
23	0.433	0.533	0.767	0.867	0.460	0.700	0.860	0.560	0.750

Table 27: Top-N Coverage in Top-K for in-domain ImageNet-16-120 (train_size=700). Values are averaged across 10 trials.

mini-batch step ($\times 10^2$)	Top-3 in Top-3	Top-3 in Top-5	Top-3 in Top-10	Top-3 in Top-20	Top-5 in Top-5	Top-5 in Top-10	Top-5 in Top-20	Top-10 in Top-10	Top-10 in Top-20
1	0.067	0.067	0.133	0.333	0.100	0.180	0.400	0.160	0.360
2	0.133	0.133	0.233	0.467	0.220	0.300	0.520	0.250	0.420
3	0.167	0.267	0.400	0.500	0.300	0.420	0.560	0.320	0.470
4	0.233	0.267	0.400	0.567	0.300	0.420	0.640	0.330	0.500
5	0.233	0.333	0.433	0.567	0.320	0.420	0.640	0.310	0.520
6	0.167	0.233	0.433	0.567	0.280	0.420	0.640	0.330	0.520
7	0.233	0.333	0.467	0.567	0.320	0.440	0.660	0.300	0.550
8	0.267	0.367	0.467	0.600	0.360	0.460	0.680	0.340	0.570
9	0.367	0.467	0.467	0.633	0.380	0.460	0.700	0.340	0.590
10	0.267	0.367	0.467	0.667	0.360	0.480	0.720	0.370	0.610
11	0.333	0.400	0.533	0.667	0.360	0.560	0.720	0.410	0.640
12	0.333	0.400	0.533	0.733	0.400	0.600	0.760	0.410	0.660
13	0.400	0.400	0.533	0.733	0.380	0.600	0.760	0.430	0.680
14	0.400	0.400	0.500	0.733	0.400	0.600	0.760	0.470	0.680
15	0.400	0.400	0.500	0.733	0.360	0.580	0.760	0.450	0.700
16	0.400	0.433	0.533	0.733	0.380	0.600	0.760	0.490	0.720
17	0.400	0.433	0.500	0.733	0.400	0.560	0.760	0.470	0.720
18	0.400	0.433	0.533	0.700	0.400	0.580	0.740	0.500	0.710
19	0.333	0.467	0.533	0.700	0.420	0.580	0.740	0.500	0.710
20	0.367	0.467	0.533	0.733	0.440	0.580	0.760	0.480	0.740
21	0.367	0.433	0.567	0.700	0.380	0.600	0.740	0.510	0.740
22	0.400	0.433	0.533	0.700	0.400	0.580	0.740	0.500	0.750
23	0.400	0.433	0.567	0.700	0.400	0.580	0.740	0.510	0.730

- **Search Space Reduction**: Early identification of top-K candidates allows NAS frameworks to reduce the search space from thousands to tens of architectures, dramatically reducing computational cost
- **Resource Allocation**: Focusing computational budget on promising architectures identified early (Top-20 recall > 0.7 by 1,600 mini-batches) maximizes efficiency
- **Early Stopping**: Top-K metrics provide concrete signals for early stopping—once Top-20 recall plateaus, additional training provides limited value
- **Parallel Exploration**: High Top-N Coverage (e.g., Top-5 Coverage in Top-20 > 0.85) enables simultaneous evaluation of multiple promising candidates, accelerating the search process

Table 28: Top-K recall for cross-dataset setups where models are trained on CIFAR-10 and tested on CIFAR-100 and ImageNet-16-120. Values represent the probability that the true best architecture appears in NAP2's predicted top-K rankings, averaged across 10 trials.

mini-batch step ($\times 10^2$)	CIFAR-100				ImageNet-16-120			
	Top-3	Top-5	Top-10	Top-20	Top-3	Top-5	Top-10	Top-20
1	0.2	0.2	0.2	0.5	0.0	0.1	0.1	0.3
2	0.4	0.4	0.6	0.7	0.1	0.2	0.2	0.5
3	0.3	0.3	0.4	0.9	0.0	0.1	0.3	0.5
4	0.2	0.3	0.6	0.8	0.2	0.3	0.4	0.6
5	0.2	0.3	0.5	0.8	0.2	0.3	0.5	0.6
6	0.2	0.3	0.6	0.8	0.2	0.3	0.5	0.6
7	0.2	0.4	0.4	0.8	0.3	0.3	0.5	0.7
8	0.2	0.3	0.6	0.8	0.3	0.4	0.5	0.7
9	0.2	0.4	0.5	0.7	0.3	0.4	0.6	0.7
10	0.1	0.4	0.6	0.7	0.1	0.4	0.6	0.7
11	0.1	0.3	0.5	0.7	0.2	0.4	0.7	0.7
12	0.1	0.3	0.6	0.7	0.1	0.4	0.7	0.7
13	0.1	0.3	0.5	0.6	0.3	0.5	0.7	0.7
14	0.1	0.3	0.6	0.6	0.4	0.5	0.7	0.7
15	0.1	0.3	0.5	0.6	0.4	0.5	0.6	0.7
16	0.2	0.3	0.6	0.6	0.3	0.5	0.6	0.7
17	0.2	0.3	0.5	0.6	0.4	0.5	0.6	0.7
18	0.2	0.3	0.5	0.6	0.3	0.4	0.6	0.7
19	0.2	0.3	0.5	0.6	0.3	0.5	0.6	0.7
20	0.2	0.4	0.5	0.6	0.3	0.5	0.7	0.7
21	0.2	0.4	0.5	0.6	0.4	0.5	0.6	0.7
22	0.1	0.4	0.5	0.6	0.4	0.5	0.6	0.7
23	0.1	0.4	0.4	0.6	0.5	0.5	0.7	0.7

Table 29: Top-K recall for cross-dataset setups where models are trained on CIFAR-100 and tested on CIFAR-10 and ImageNet-16-120. Values represent the probability that the true best architecture appears in NAP2's predicted top-K rankings, averaged across 10 trials.

mini-batch step ($\times 10^2$)	CIFAR-10				ImageNet-16-120			
	Top-3	Top-5	Top-10	Top-20	Top-3	Top-5	Top-10	Top-20
1	0.0	0.2	0.2	0.4	0.1	0.3	0.3	0.3
2	0.2	0.2	0.4	0.4	0.3	0.3	0.5	0.8
3	0.2	0.3	0.3	0.5	0.2	0.6	0.8	0.9
4	0.1	0.2	0.4	0.5	0.2	0.3	0.5	0.9
5	0.1	0.2	0.4	0.4	0.3	0.4	0.8	0.9
6	0.2	0.3	0.4	0.4	0.2	0.3	0.7	0.9
7	0.2	0.3	0.4	0.5	0.2	0.3	0.8	0.9
8	0.2	0.3	0.4	0.4	0.3	0.4	0.9	0.9
9	0.1	0.3	0.4	0.4	0.1	0.3	0.8	0.9
10	0.2	0.3	0.4	0.5	0.2	0.3	0.8	1.0
11	0.1	0.3	0.4	0.4	0.2	0.3	0.9	1.0
12	0.2	0.3	0.4	0.5	0.2	0.3	0.9	1.0
13	0.1	0.3	0.4	0.5	0.2	0.3	0.9	1.0
14	0.2	0.3	0.4	0.6	0.2	0.4	0.9	1.0
15	0.1	0.3	0.3	0.6	0.2	0.3	1.0	1.0
16	0.2	0.3	0.4	0.5	0.2	0.4	1.0	1.0
17	0.2	0.3	0.3	0.6	0.2	0.3	0.8	1.0
18	0.1	0.3	0.3	0.5	0.2	0.6	0.9	1.0
19	0.1	0.3	0.3	0.5	0.2	0.5	0.9	1.0
20	0.2	0.3	0.3	0.6	0.2	0.4	0.9	1.0
21	0.3	0.3	0.3	0.5	0.1	0.6	0.9	1.0
22	0.3	0.3	0.3	0.6	0.1	0.6	0.9	1.0
23	0.2	0.3	0.3	0.6	0.2	0.6	1.0	1.0

- **Transfer Learning**: Strong cross-dataset Top-K recall enables applying models trained on fast datasets (CIFAR-10) to larger tasks (ImageNet) without retraining

Table 30: Top-K recall for cross-dataset setups where models are trained on ImageNet-16-120 and tested on CIFAR-10 and CIFAR-100. Values represent the probability that the true best architecture appears in NAP2's predicted top-K rankings, averaged across 10 trials.

mini-batch step ($\times 10^2$)	CIFAR-10				CIFAR-100			
	Top-3	Top-5	Top-10	Top-20	Top-3	Top-5	Top-10	Top-20
1	0.2	0.2	0.3	0.5	0.2	0.2	0.4	0.5
2	0.1	0.2	0.3	0.6	0.2	0.2	0.5	0.6
3	0.1	0.2	0.5	0.6	0.1	0.4	0.7	0.8
4	0.2	0.3	0.5	0.7	0.3	0.5	0.8	0.8
5	0.2	0.3	0.5	0.7	0.4	0.5	0.8	0.8
6	0.1	0.3	0.5	0.7	0.2	0.5	0.8	0.8
7	0.1	0.3	0.5	0.7	0.2	0.5	0.8	0.8
8	0.2	0.3	0.5	0.7	0.3	0.6	0.8	0.9
9	0.1	0.3	0.4	0.7	0.2	0.5	0.7	0.8
10	0.1	0.3	0.5	0.8	0.2	0.5	0.7	0.9
11	0.1	0.3	0.5	0.7	0.2	0.5	0.7	0.8
12	0.1	0.3	0.5	0.7	0.2	0.6	0.7	0.8
13	0.1	0.2	0.5	0.7	0.2	0.4	0.7	0.8
14	0.1	0.3	0.5	0.7	0.1	0.5	0.7	0.8
15	0.1	0.2	0.5	0.8	0.2	0.4	0.7	0.9
16	0.1	0.2	0.5	0.7	0.2	0.4	0.7	0.8
17	0.1	0.2	0.5	0.8	0.2	0.4	0.7	0.9
18	0.1	0.2	0.5	0.7	0.2	0.4	0.7	0.8
19	0.1	0.3	0.6	0.8	0.1	0.5	0.7	0.9
20	0.1	0.2	0.5	0.8	0.1	0.4	0.7	0.9
21	0.1	0.2	0.6	0.8	0.1	0.4	0.7	0.9
22	0.1	0.2	0.6	0.7	0.1	0.4	0.7	0.8
23	0.1	0.3	0.6	0.8	0.1	0.5	0.7	0.8

G Theoretical Foundations

This appendix provides a detailed theoretical treatment of the principles underlying NAP2's design. We formalize the connections between our meta-features and established generalization theory, justify the temporal modeling approach, and analyze the conditions under which cross-dataset transfer is expected to succeed.

G.1 Norm-Based Features and PAC-Bayes Generalization Bounds

We begin by formalizing the relationship between NAP2's norm-based features and known generalization bounds. Consider a feedforward network f_w with L layers, where W_l denotes the weight matrix of layer l. The PAC-Bayes framework McAllester (1999) provides generalization bounds of the form:

$$L_{\text{test}}(w) \leq L_{\text{train}}(w) + \mathcal{O}\left(\sqrt{\frac{\text{KL}(q \,\|\, p)}{n}}\right) \tag{33}$$

where q is a posterior distribution centered on the learned weights w, p is a data-independent prior, n is the number of training examples, and KL denotes the Kullback–Leibler divergence. When both q and p are Gaussian, the KL term reduces to a function of $\|w\|^2$ (plus terms depending on the variances), establishing a direct link between weight norms and generalization.

More specifically, Neyshabur et al. (2017) showed that for a network with ReLU activations, the generalization bound scales as:

$$\mathcal{O}\left(\frac{B^2}{\gamma^2} \cdot \prod_{l=1}^{L} \|W_l\|_\sigma^2 \cdot \sum_{l=1}^{L} \frac{\|W_l\|_F^2}{\|W_l\|_\sigma^2} \cdot \frac{\ln L}{n}\right) \tag{34}$$

where $\|W_l\|_\sigma$ is the spectral norm of layer l, $\|W_l\|_F$ is the Frobenius norm, B is the input norm bound, and γ is the classification margin. The terms $\|W_l\|_F$ and $\|W_l\|_\sigma$ are both captured (or closely approximated) by NAP2's per-layer L2 norm features: the L2 norm computed over all weights in a layer corresponds to the Frobenius norm, while the maximum singular value is bounded by the Frobenius norm and reflected in the max and variance statistics across units.

Similarly, Bartlett et al. (2017) derived margin-based bounds that depend on the product of spectral norms $\prod_l \|W_l\|_\sigma$ weighted by per-layer $(2,1)$-matrix norms. Our L1 norm features (the sum of absolute values per layer) relate to these $(2,1)$-norms, providing the predictor with a signal that tracks this specific complexity measure.

The key insight is that NAP2 does not need to compute these bounds explicitly—it only needs features that correlate with the quantities appearing in them. By providing per-layer L1 and L2 norms, along with distributional statistics (variance, standard deviation) that characterize the spread of unit-level norms within each layer, NAP2 gives the LSTM sufficient information to implicitly learn a function that approximates these complexity–generalization relationships.

G.2 Distributional Shape Features and Spectral Properties

The kurtosis and skewness features in NAP2 are motivated by connections between weight matrix distributional properties and generalization quality. Martin & Mahoney (2021) applied random matrix theory to study the empirical spectral density (ESD) of weight matrices in trained neural networks. Their central finding is that well-generalizing networks tend to develop weight matrices whose ESDs exhibit heavy-tailed behavior, well-approximated by a power-law distribution. The fitted power-law exponent $\hat{\alpha}$ was shown to be strongly predictive of test accuracy across a range of architectures and datasets, without access to test data.

Computing full ESDs requires singular value decomposition at each layer, which is expensive. However, the tail behavior of a distribution is intimately connected to its higher-order moments:

- **Kurtosis** measures the heaviness of the tails relative to a Gaussian distribution. A distribution with power-law tails (the hallmark of well-generalizing networks per Martin & Mahoney) exhibits high kurtosis. Intuitively, per-layer kurtosis of the weight distribution can thus serve as a computationally cheap proxy for the ESD tail exponent.
- **Skewness** captures asymmetry in the weight distribution. During training, the development of asymmetric weight distributions can reflect the emergence of specialized features. Symmetric distributions (zero skewness) are characteristic of initialization, while increasing skewness during training reflects the breaking of this symmetry as the network learns.

Together, kurtosis and skewness provide a compact characterization of the distributional shape that captures generalization-relevant spectral properties at a fraction of the computational cost of full spectral analysis. The covariance feature further captures inter-unit relationships within a layer, reflecting the degree of functional specialization: low covariance indicates redundant units, while structured covariance suggests efficient use of the layer's representational capacity.

G.3 Gradient Features and Loss Landscape Curvature

Gradient-based features provide complementary information about the local geometry of the loss surface. The *empirical* Fisher information matrix at parameters w is defined as:

$$\hat{F}(w) = \frac{1}{n} \sum_{i=1}^{n} \nabla_w \ell(w, x_i) \cdot \nabla_w \ell(w, x_i)^\top \tag{35}$$

where $\ell(w, x_i)$ is the per-sample loss. The diagonal entries of $\hat{F}$ correspond to the squared gradient magnitudes per parameter, and its eigenspectrum characterizes the curvature of the loss surface in different directions. While computing the full Fisher is prohibitively expensive, our per-layer gradient statistics capture summary information about these quantities:

- **Gradient L2 norms** approximate the trace of the per-layer Fisher block, which equals the sum of its eigenvalues. This provides a scalar summary of the total curvature magnitude at each layer.

- **Gradient variance and standard deviation** across units within a layer reflect the anisotropy of the local curvature. High variance indicates that some directions in parameter space have much steeper loss curvature than others—a condition associated with sharp minima and potentially poor generalization Keskar et al. (2016).
- **Gradient kurtosis** captures whether the curvature is concentrated in a few extreme directions (high kurtosis) or spread more uniformly (low kurtosis), reflecting the effective dimensionality of the loss landscape curvature.

The combination of weight and gradient features at each snapshot provides a joint characterization of the network's position in parameter space and the local geometry of the loss surface at that position. This is analogous to characterizing a physical system by both its state (position) and its dynamics (forces/accelerations), which is strictly more informative than either alone.

G.4 Formal Justification for Temporal Modeling

Let $\phi(w_t) \in \mathbb{R}^d$ denote the feature vector extracted from the network weights at training step t (encompassing both weight and gradient statistics). A static predictor observes a single $\phi(w_{t_0})$ and must predict final performance y from this point estimate alone. NAP2, by contrast, observes a sequence $\{\phi(w_{t_1}), \phi(w_{t_2}), \ldots, \phi(w_{t_k})\}$ and can therefore access temporal information including:

1. **First-order dynamics (velocity):** $\Delta\phi_t = \phi(w_{t+1}) - \phi(w_t)$, which captures the rate of change of each feature. Rapidly decreasing gradient norms indicate fast convergence, while oscillating norms suggest instability.
2. **Second-order dynamics (acceleration):** $\Delta^2\phi_t = \Delta\phi_{t+1} - \Delta\phi_t$, which captures whether the optimization is accelerating, decelerating, or oscillating. This is relevant to phenomena such as the edge of stability Cohen et al. (2021), where the sharpness of the loss surface is constrained by the step size.
3. **Trajectory shape:** The overall pattern of the sequence—monotonic convergence, oscillation with decreasing amplitude, sudden transitions—provides information about the loss landscape topology that cannot be inferred from any finite number of derivatives at a single point.

Formally, consider two architectures A_1 and A_2 such that $\phi(w_{t_0}^{A_1}) = \phi(w_{t_0}^{A_2})$ at some snapshot t_0—that is, they are indistinguishable from a single observation. If their trajectories diverge (e.g., A_1's gradient norms decrease monotonically while A_2's oscillate), the sequence-based predictor can distinguish them while the static predictor cannot. The LSTM architecture is a natural choice for modeling such sequential dependencies, as it can learn to attend to both short-term fluctuations and long-term trends through its gating mechanism.

The empirical observation that NAP2's performance improves rapidly over the first approximately 10 snapshots and then plateaus (Section **??**) is consistent with this framework: the early snapshots provide the most informative trajectory features (capturing the initial transient dynamics), while later snapshots contain diminishing marginal information as the optimization settles into a more predictable regime.

G.5 Conditions for Cross-Dataset Transfer

We provide a more formal analysis of why NAP2's features should support cross-dataset transfer. Let $\mathcal{D}_s$ and $\mathcal{D}_t$ denote the source and target datasets, and let f_A denote a fixed architecture A. We argue that NAP2's transfer relies on the following conditions, each with theoretical and empirical support:

Condition 1: Architecture-dependent loss landscape structure. For a fixed architecture A, the topology of the loss landscape $\mathcal{L}_{A,\mathcal{D}}(w) = \mathbb{E}_{(x,y)\sim\mathcal{D}}[\ell(f_A(x;w), y)]$—including the distribution and connectivity of local minima—is primarily determined by A. Choromanska et al. (2015) showed that under

simplifying assumptions, the loss function of a deep network with random i.i.d. inputs has a structure where local minima concentrate near a global band, with the number and depth of these minima determined by the network size (a property of A, not $\mathcal{D}$). Li et al. (2018) provided complementary empirical evidence that network architecture—particularly depth, width, and the presence of skip connections—has a dramatic effect on loss landscape geometry, with these structural properties remaining qualitatively consistent across training runs.

Condition 2: Qualitative preservation of optimization dynamics. The early training dynamics—the specific sequence of weight updates, gradient magnitudes, and their evolution—are governed by the interaction between the architecture's parameterization and the optimizer. While the specific numerical values change with the dataset, the qualitative patterns (which layers converge fastest, how gradient magnitudes relate across layers, whether optimization is stable or oscillatory) are largely determined by architectural properties such as depth, skip connections, and activation functions. The implicit bias literature Gunasekar et al. (2018) provides theoretical support: gradient descent converges to solutions with specific structural properties determined by the architecture's parameterization.

Condition 3: Scale normalization removes dataset-specific calibration. By min-max normalizing the target accuracies to $[0, 1]$ using training-set statistics, NAP2 converts the prediction problem from estimating absolute accuracy (which is highly dataset-dependent) to estimating relative ranking (which is more architecture-dependent). If architectures A_1 and A_2 satisfy $\text{acc}(A_1, \mathcal{D}_s) > \text{acc}(A_2, \mathcal{D}_s)$ because A_1 has better optimization dynamics on $\mathcal{D}_s$, and if Conditions 1 and 2 hold, then the same relative ordering is likely preserved on $\mathcal{D}_t$. The normalization ensures that the predictor need not recalibrate to the different absolute accuracy range of $\mathcal{D}_t$.

When transfer may fail. These conditions are not absolute. If the target dataset has qualitatively different properties—for example, extremely different input dimensionality, label structure, or noise characteristics—the optimization dynamics may diverge sufficiently to break the architecture-dependent patterns. The SVHN experiment (Section **??**) provides an encouraging result: even under significant distribution shift (natural images vs. digit crops with very different accuracy distributions), transfer remains effective, suggesting the architecture-dependent signal in early training dynamics is robust. However, we expect that transfer to fundamentally different tasks (e.g., from image classification to sequence modeling) would require at minimum a retraining of the LSTM predictor, as the feature distributions would shift substantially.

G.6 Summary of Design Choice Justifications

We summarize the theoretical justification for each major design choice in NAP2 in Table 31.

Table 31: Summary of theoretical justifications for NAP2's design choices.

Design Choice	**Theoretical Justification**
L1 and L2 norm features	Track quantities appearing in PAC-Bayes and spectral-norm generalization bounds Neyshabur et al. (2017); Bartlett et al. (2017)
Kurtosis and skewness	Computationally efficient proxies for heavy-tail spectral properties linked to generalization Martin & Mahoney (2021)
Gradient statistics	Capture local loss landscape curvature via relationship to the Fisher information matrix eigenspectrum
Combined weight + gradient features	Joint state-and-dynamics characterization; strictly more informative than either alone
Temporal sequence of snapshots	Captures trajectory velocity and acceleration, enabling discrimination between architectures indistinguishable from single-point observations
LSTM predictor	Natural architecture for variable-length sequences with both short-term and long-term temporal dependencies
100 mini-batch snapshot interval	Balances temporal resolution with storage overhead; sufficient to capture early transient dynamics where most predictive signal concentrates
Min-max normalization of targets	Removes dataset-specific scale, converting prediction to relative ranking that is more architecture-dependent and thus transferable
Cross-dataset transfer	Supported by architecture-dependent loss landscape structure Choromanska et al. (2015); Li et al. (2018) and qualitative preservation of optimization dynamics Gunasekar et al. (2018)