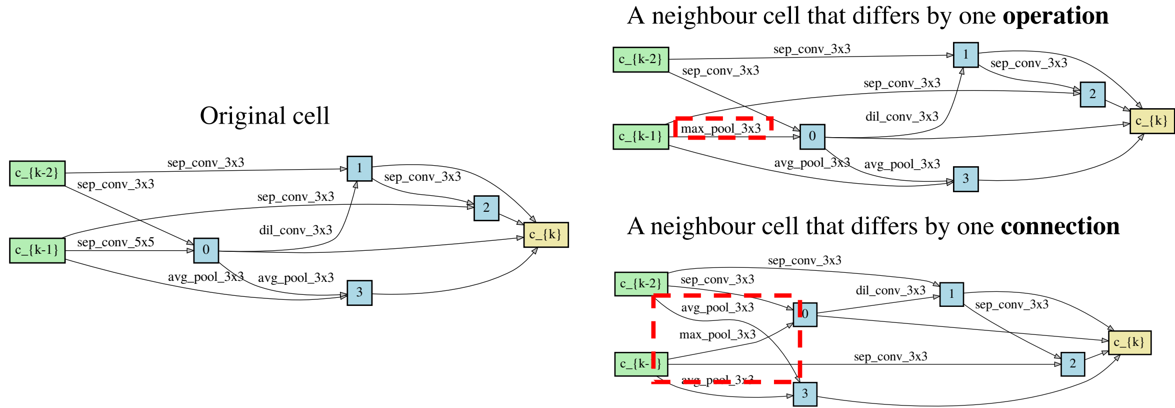

Figure 4: Examples of neighbouring cells in the architecture space. The first neighbour differs by one operation (circled in red) and the second neighbour differs by one connection (note the average pooling block is connected to input `c_{k-2}` instead of node 0 in the original cell.

Appendices

A. Experimental Setup

In this section we outline the detailed implementation setup for both our method and the baselines we implemented, on both the benchmark and open-domain tasks considered in this paper.

A.1. Benchmark Tasks

Data We experiment on NB201 and NB301. NB201 has a search space in the form of acyclic directed graph with 4 nodes and 6 edges, with each edge corresponding to a possible operation from `conv1×1`, `conv×3`, `skip-connect`, `maxpool` and `zeroize` (`zeroize` suggests the ending node will regard any input information as zero). The data-set contains 15,625 neural architectures, evaluated exhaustively on 3 image datasets, namely CIFAR-10, CIFAR-100 and ImageNet16 (the downsampled ImageNet to 16×16). In addition to standard metrics such as validation and test accuracy/losses, we may also access additional information such as the complete training curve, time spent in training and FLOPs. The dataset and API can be downloaded from `https://github.com/D-X-Y/NAS-Bench-201`.

NB301 [38] differs from the previous tabular benchmark in that it contains a much larger search space (the search space is identical to the DARTS search space, which contains more than 10^{18} unique architectures). Each architecture contains 2 *cells*, with each cell again being a acyclic directed graph similar to NB201, but with 8 nodes and 14 possible edges. Each edge corresponds to one of a wider range of possible operations, namely separable convolution 3×3, 5×5; dilated convolution 3×3, 5×5; max pooling 3×3, average pooling 3×3, skip connection. Due to the prohibitively large search space it is not possible to exhaustively train each unique neural architecture contained in it. Instead, a large number of query-based NAS methods, ranging from the simplest random search to more complicated methods such as BANANAS [44], are run on this search space to give a good overall coverage by training a large number of architectures. With this large corpus of trained architectures and their corresponding accuracy, an ensemble of *surrogate* models, including an ensemble of models including LGBoost[20], XGBoost [7] and graph isomorphic networks (GIN) [47], are fit so that given an unknown architecture G, we might use the *predicted accuracy* from this surrogate model as a much cheaper proxy to the true, expensive accuracy that might only be obtained from training the architecture from scratch. Similar to NB201, NB301 similarly provides the additional diagnostic information as listed above. The dataset and the API may be downloaded from `https://github.com/automl/nasbench301`.

Algorithm Setup Unless otherwise stated, each method is repeated with 10 random starts and the mean/standard error are reported and where applicable, each method is also initialised with 10 random samples/evaluations at the beginning of the routines.

- **Random Search (RS)** Both the original RS and the unbiased ANASOD-RS are hyperparameter free, but due to the presence of bias away from the uniform distribution in ANASOD-biasedRS, there exists some additional hyperparameters. In our experiments, we set the maximum Dirichlet scale $\beta_{\max}$ (explained in Section 3.2) to be 2 (i.e. the β at the last random search iteration $t = T$), and we use a linear annealing schedule such that the β at iteration t is simply $t\frac{\beta_{\max}}{T}$.
- **Local Search (LS)** For LS, we mainly adapt the implementation available in the repository associated

with [43] (available at `https://github.com/realityengines/local_search` and `https://github.com/naszilla/nas-encodings`). For the baseline LS, at each iteration the best performing architecture up to the current iteration t (α_t^*) is identified; From this, the algorithm exhaustively enumerates and queries its *neighbours*, defined in this context as the other architectures α such that each of them only differs from α_t^* by edit distance of 1 (i.e. only different by 1 operation block or wiring. See Figure 4 for examples of neighbour architectures). The algorithm terminates (and restarts again if there is still unused budget) when it cannot find neighbour(s) to improve the best performance, suggesting that a (local) optimum has been reached.

In ANASOD-LS, we use the same procedure but conduct the search in the ANASOD encoding space instead of the architecture space at the beginning. Specifically, instead of identifying the best performing *architecture*, we identify the best performing *encoding* $\tilde{\mathbf{p}}_t^*$ up to the current iteration t and search its neighbours as described above. In this case, since each valid ANASOD encoding is a vertex on the regular grid in the k-simplex, the neighbours are simply the adjacent vertices on the grid (for example, for a cell with $N = 5$ and $k = 3$, the neighbours of encoding $[1, 1, 3]$ are $[0, 1, 4], [1, 0, 4], [0, 2, 3], [2, 0, 3], [1, 2, 2]$ and $[2, 1, 2]$).

- **ANASOD-BO** We use a standard Gaussian Process (GP) as the surrogate for ANASOD-BO with Mátern 5/2 kernel. To improve the goodness of fit of GP, we always normalise the input (in our case, the ANASOD encodings) into standard hyperrectangular boxes $[0, 1]^k$. For the targets (in our case, the validation error), we find that using a log-transform often leads to better estimation of the predictive uncertainty, similar to the finding in [35]. We then use a normalisation wrapping around the log-predictive uncertainty by de-meaning the data and dividing by the data standard deviation such that the GP targets are approximately within the range of $\mathcal{N}(0, 1)$.

  For the GP hyperparameters, we constrain the lengthscales to be within $[0.01, 0.5]$, the outputscale to be in $[0.5, 5]$ and the noise variance to be within $[10^{-6}.10^{-1}]$ and by default we do not enable the automatic relevance determination (ARD). The exact hyperparameters of the GP are obtained by optimising the GP log-marginal likelihood and the GP model is implemented using the gpytorch [14] package which feature automatic differentiation.

  For the implementation of the BO routine, we use the expected improvement criterion as the acquisition function that needs to be optimised at each iteration. To optimise the acquisition function to obtain the recommendation from the BO $\arg\max a(\tilde{\mathbf{p}})$, in this paper we use the sample-based optimisation by first randomly generating $100k$ samples via randomly sampling the *candidate generating distribution* (described in Algorithm 2 in the main text). However, we also explore a gradient-based optimisation approach in to the acquisition function, whose update may be written as:

$$\tilde{\mathbf{p}}' \leftarrow \tilde{\mathbf{p}} - \eta \frac{\partial a(\tilde{\mathbf{p}})}{\partial \tilde{\mathbf{p}}}$$

$$\tilde{\mathbf{p}} \leftarrow \frac{\tilde{\mathbf{p}}'}{\sum_i^k \tilde{p}_i} \tag{4}$$

  where we conduct mirror descent by combining the gradient-based update (we use the simple gradient descent for simplicity but the optimisation is compatible with gradient-based optimisers such as Adam; here η is the generic learning rate for gradient-based methods) with the normalisation to enforce the simplex constraint. We find that both acquisition optimisation methods to be broadly comparable, but it is trivial in concept and implementation to extend the sample-based optimisation technique into batch setting where at each BO iteration, multiple recommendations are produced.

- **Other baselines** For reinforcement learning (RL), regularised evolution (RE), Sequential Model-based Algorithm Configuration (SMAC), Tree-structured Parzen Estimator (TPE), we use the code available at `https://github.com/automl/nas_benchmarks/` for the NB201 search space and adapt the codes for the NB301 repository. For BANANAS [44], we use the official implementation available at `https://github.com/naszilla/bananas`; for GCN-based BO, we use the implementation available at NASLib [36] `https://github.com/automl/NASLib`; for NAS-BOWL [35], we use the official implementation at `https://github.com/xingchenwan/nasbowl`. Unless otherwise specified, we use the default settings and hyperparameters in these methods. Since NB301 is to be used as a cheaper proxy to the DARTS search space, for the baseline methods, we use the default settings for DARTS search as the settings for the NB301 experiments.

A.2. Open-domain Tasks

For the open-domain tasks we use the popular NASNET-style [56] search space. Since NB301 benchmark already includes the typical DARTS search space (i.e. NASNET-style search space with 8 nodes and 14 possible edges), here we follow previous works [40, 41] by opting for an enlarged

search space with 10 nodes and 20 possible edges, containing order-of-magnitudes more unique architectures even than the standard DARTS space – this makes the search of a good performing cell within a small number of queries even more challenging. For the implementation details, we largely follow [40, 41, 28] but with two major exceptions: first, as a demonstration of our sample efficiency, we both train much fewer *architectures* during search time, and train fewer *epochs per architecture* compared to previous works such as [35, 44, 37]: during search time we use the ANASOD-BO procedure described above. Second, while [41] restricts the candidate operations to 4 perceived to be high-performing (max pool, separable convolution 3×3, skip connection and dilated convolution 3×3), we retain the full range of 8 operation candidates for fair comparison with other previous works. For an encoding $\tilde{\mathbf{p}}$, we sample 24 valid cells randomly to build the architecture (this is possible due to the fact that we may sample architectures from the ANASOD encoding; previous work typically repeats the identical cell multiple times to build a network), and we train each sampled architecture for 30 epochs using SGD with momentum optimiser with cosine annealed learning rate schedule (learning rate at start $\eta_0 = 0.025$ which anneals to 0 at the 30th epoch; momentum $= 0.9$; weight decay = 3×10^{-4}) with a batch size of 96, depth of 24 and initial channel number of 36. We terminate the search after training 50 architectures as described, and select the encoding that leads to the best validation error for *evaluation*. To utilise the parallel computing facilities, we run ANASOD-BO with batch size of 4 (at each BO iteration, 4 encodings are generated and recommended for training); it is worth noting that further parallelisation is possible to reduce the wall clock time. For example, it is possible to trivially reduce the wall-clock time to 0.3 days if 8 GPUs are used.

We use two slightly different evaluation procedures and thus report two results, namely ANASOD-BO and ANASOD-BO+ in Tables 4 and 5 where we largely follow [41]. For the standard, ANASOD-BO result, we train the architecture induced by the encoding for 600 epochs, again with the SGD with momentum optimiser with initial learning rate set at $\eta_0 = 0.025$ (but the learning rate schedule is now annealed across the entire 600 epochs instead of the 30 epochs described above) and every other hyperparameter is kept consistent with the description above for search. During evaluation, we also include additional augmentation techniques such as CutMix [52], Auxiliary Tower with probability 0.4 and drop path with probability 0.2. For ANASOD-BO+, we follow previous works such as [27] to increase the training to 1,500 epochs with identical optimiser setting (with the learning rate annealed over 1,500 epochs), and we additionally use AutoAugment. We use the default train/validation partition sets of CIFAR-10 for both search and evaluation, and for the CIFAR-100 transfer learning experiment, we directly use the cell searched from CIFAR-10 for evaluation. The evaluation protocol on CIFAR-100 is otherwise identical to that on CIFAR-10 described above.

B. Results on Local Search

Local search has recently been shown to be an extremely powerful optimiser, achieving state-of-the-art performance especially in small search spaces [45]. However, a well-known theoretical and empirical drawback of LS is its susceptibility of getting trapped locally [17, 1], especially when the search space is large and/or is rife with local optima (in the NAS context, LS is found to perform much worse in larger search spaces (e.g. DARTS space) both in [45] and by us in Sec 4.1). To leverage the ANASOD encoding, we propose a simple modification (which we term ANASOD-LS in Algorithm 3) which performs LS on the ANASOD encoding space first and switches to the architecture space when the algorithm fails to improve (i.e. it reaches a local optimum). As discussed, ANASOD encoding effectively abstracts the huge, original architecture space by compressing multiple unique architectures into a single encoding with fewer neighbours (a neighbour in encoding space is an adjacent vertex on the regular grid on the k-simplex), both of which allow LS on the encoding space to make rapid progress on a smoothed objective function landscape. It is worth noting that for this acceleration to be possible, we have to both 1) be able to easily sample architectures *from* the encoding and 2) have an encoding scheme that compresses and smooths the original architecture search space so that LS is less likely to be stuck. To our knowledge, ANASOD is the only one that meets both desiderata.

We conduct experiments of ANASOD-LS on NB201 and NB301 search spaces, and the results are shown in Table 6 and Fig 6. In line with the observation of [45], we observe that the baseline LS performs extremely competitively in the NB201 tasks but struggles in the larger NB301 search space and under-performs even random search—presumably due to the more prevalent local optima that prevents LS from progressing further. However, ANASOD-LS restores the competitiveness in NB301, and even for the NB201 tasks where the original LS already performs strongly, the use of ANASOD nonetheless improves its convergence speed in the initial stage of the optimisation. Generally, our experiments show that the margin of improvement increases as the task difficulty increases. Indeed, in the ImageNet task ANASOD improves both convergence speed and final performance; and in the CIFAR tasks ANASOD does not improve the final performance further as the baseline is already very close to the ground-truth optimum.

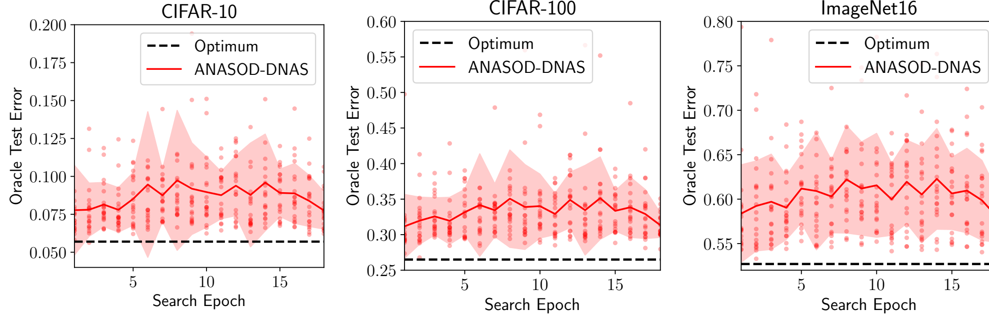

Figure 5: Learning curves of ANASOD-DNAS on NB201 datasets (oracle test error vs the number of learning epochs). We search on the CIFAR-10 training set and transfer the searched cell directly to the other two datasets. Lines and shades denote mean $\pm$ 1 standard deviation, and dots denote the data points every each trial. Note that unlike previous methods such as DARTS that quickly collapse to all-skip-connection cells that perform poorly, ANASOD-DNAS converges quickly to good solutions and maintains high performance throughout the optimisation process.

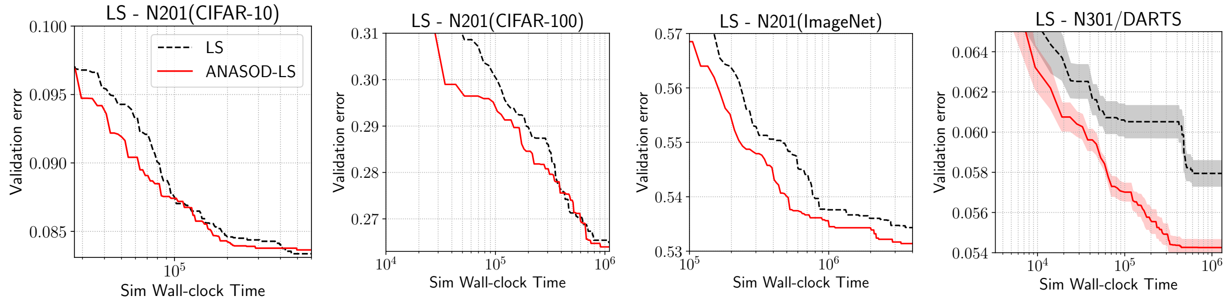

Figure 6: Performance of LS vs ANASOD-LS on NB201 and NB301. Lines and shades denote mean $\pm$ 1 standard error, across 10 different random trials.

Table 6: Comparison of original LS with ANASOD-LS. We report mean $\pm$ 1 standard error of the validation error across 10 random trials. Numbers shown denote the best validation error seen after **300** architecture queries.

Benchmark Dataset	 CIFAR-10	NB201 CIFAR-100	 ImageNet16	NB301 CIFAR-10
LS [45]	$\mathbf{8.34}_{\pm 0.03}$	$26.41_{\pm 0.04}$	$53.43_{\pm 0.08}$	$5.79_{\pm 0.06}$
ANASOD-LS	$8.36_{\pm 0.02}$	$\mathbf{26.38}_{\pm 0.07}$	$\mathbf{53.07}_{\pm 0.09}$	$\mathbf{5.43}_{\pm 0.04}$

C. Further results on ANASOD-DNAS

We show the learning curves of ANASOD-DNAS in Figure 5, where at each training epoch we sample and query the oracle test accuracy of an architecture from the encoding learnt at that point: ANASOD-DNAS almost *instantly* finds and converges to a location of good solutions, allowing us to shorten the search time while still recovering good solutions – this is possible because ANASOD encoding massively compresses and smooths the search space into a comparatively low-dimensional vector, allowing gradient-based optimisers to converge efficiently. Furthermore, unlike methods like DARTS that are shown to lack sufficient regularisation and quickly overfits to find cells dominated by skip-connections that hence perform poorly [11], the performance of ANASOD-DNAS does not drop as optimisation proceeds.

Algorithm 3 Local search. Steps added or modified by ANASOD-LS are marked blue.

1: **Input:** Search space A, objective function (default: validation error) y
2: Sample an ANASOD encoding at uniform $\tilde{\mathbf{p}}_1 \sim \mathrm{Dir}(1, ..., 1)$
3: Evaluate a single architecture α_1 from the encoding $\tilde{\mathbf{p}}_1$ to approximate the performance of the all architectures parameterised by $\tilde{\mathbf{p}}_1$: $y(\tilde{\mathbf{p}}_1) = \mathbb{E}_{\alpha \sim p(\alpha|\tilde{\mathbf{p}})}\left[y(\alpha)\right] \approx y(\alpha_1), \alpha_1 \sim p(\alpha|\tilde{\mathbf{p}})$; denote dummy variable $y(\tilde{\mathbf{p}}_0) = \infty$.
4: **while** $y(\tilde{\mathbf{p}}_i) \leq y(\tilde{\mathbf{p}}_{i-1})$ **do**
5: Search and evaluate the neighbourhood on the **encoding space** $\tilde{\mathbf{p}}_{i+1} \leftarrow \mathrm{SearchNeighbour}(\tilde{\mathbf{p}}_i)$.
6: **end while**
7: */* Switch to arch space when* LS *on encoding reaches an optimum */*
8: **if** Using ANASOD-LS **then**
9: Sample an arch *from the best encoding found* : $\alpha_i \sim p(\alpha|\tilde{\mathbf{p}}^*)$
10: **else**
11: Sample an arch uniformly from the search space A.
12: **end if**
13: **while** $y(\tilde{\alpha}_i) \leq y(\tilde{\alpha}_{i-1})$ **do**
14: Search and evaluate the neighbourhood on the **architecture space** $\tilde{\alpha}_{i+1} \leftarrow \mathrm{SearchNeighbour}(\tilde{\alpha}_i)$.
15: **end while**

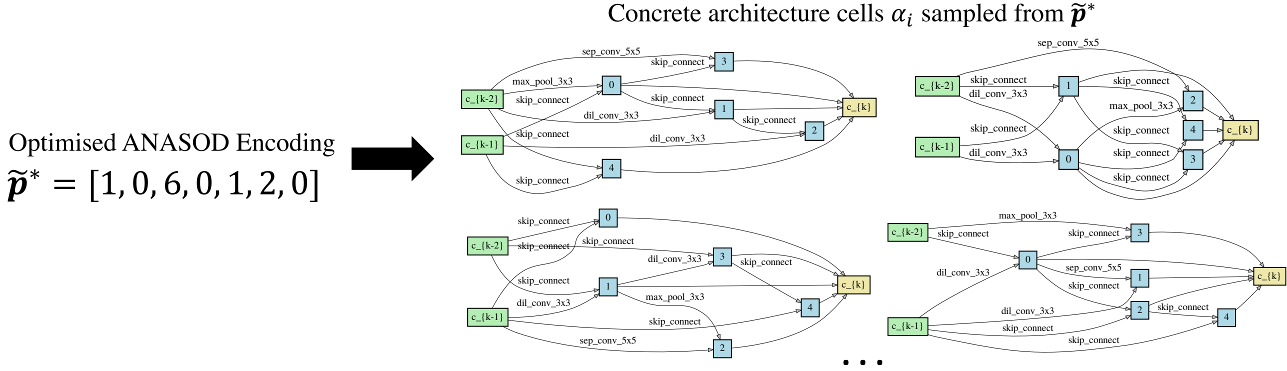

Figure 7: Best encoding found by ANASOD-BO in the open-domain task (i.e. 1 max pooling, 6 skip connections, 1 separable convolution 5×5 and 2 dilated convolution 3×3) and examples of concrete cells sampled from the optimised ANASOD encoding. Note that, similar to [44, 41, 35] who use the same cell, we sample the normal and reduction cells from the same encoding.

Implementation details We mainly use the repository open-sourced by the authors of [24] available at `https://github.com/liamcli/gaea_release`. Implementing ANASOD-DNAS on the basis of [24] is extremely simple: instead of searching for the optimal categorical distribution parameters on each edge, we force all edges to share one single set of parameters (i.e. the ANASOD encoding). We use the default hyperparameters settings as in [24].

D. Optimal Encoding Searched on the Open-domain Task

In this section we present the best encoding $\tilde{\mathbf{p}}^*$ found by ANASOD-BO in the open-domain search space described in Section A.2 (Figure 7). Unlike existing NAS method which searches for an exact *cell*, the optimal encoding maps to a distribution of architectures from which we sample concrete cells to build the neural network for evaluation, and some examples of which are also demonstrated in Figure 7. It is very interesting that ANASOD-BO produces light-weight cells with a large number of skip connections but nevertheless perform promisingly (i.e. the skip connections might indeed help in the performance, as opposed to being an artefact of overfitting). Furthermore, the cells obtained are very different from the optimised cells from most methods that are dominated by separable convolution 3×3; this suggests that the cells dominated by separable convolution 3×3 might be only one of the local minima in the loss landscape w.r.t the *architecture* parameters; other possible optima on the possibly multi-modal landscape, while not often discovered by the existing methods, might perform equally competitively. We believe the characterisation and analysis of the loss landscape w.r.t architecture parameters , a heretofore under-explored area, could be an exciting future direction for further theoretical understanding of NAS.