`einspace`: Searching for Neural Architectures from Fundamental Operations

Linus Ericsson[1] Miguel Espinosa Minano[1] Chenhongyi Yang[1]
Antreas Antoniou[1] Amos Storkey[1] Shay B. Cohen[1] Steven McDonagh[1] Elliot J. Crowley[1]

[1]The University of Edinburgh

Abstract Neural architecture search (NAS) finds high performing networks for a given task. Yet the results of NAS are fairly prosaic; they did not e.g. create a shift from convolutional structures to transformers. This is not least because the search spaces in NAS often aren't diverse enough to include such transformations *a priori*. Instead, for NAS to provide greater potential for fundamental design shifts, we need a novel expressive search space design which is built from more fundamental operations. To this end, we introduce `einspace`, a search space based on a parameterised probabilistic context-free grammar (CFG). Our space is versatile, supporting architectures of various sizes and complexities, while also containing diverse network operations which allow it to model convolutions, attention components and more. It contains many existing competitive architectures, and provides flexibility for discovering new ones. Using this search space, we perform experiments to find novel architectures as well as improvements on existing ones on the diverse Unseen NAS datasets. We show that competitive architectures can be obtained by searching from scratch, and we consistently find large improvements when initialising the search with strong baselines. We believe that this work is an important advancement towards a transformative NAS paradigm where search space expressivity and strategic search initialisation play key roles.

1 Introduction

The goal of neural architecture search (NAS) is to automatically choose a network architecture for a given task, removing the need for expensive human expertise. A NAS method defines a search space of all possible architectures that can be chosen, and a search algorithm to navigate through the space, selecting the most suitable architecture with respect to search objectives. Despite significant research investment in NAS, with over 1000 papers released since 2020 [49], manually designed architectures such as transformers [47], MLP-Mixers [45], and ResNets [17] still dominate the landscape, posing the question of why NAS isn't more widely used.

Part of the problem with NAS is that most search spaces are not expressive enough, relying heavily on high-level operations and rigid structures, making it impossible to discover anything beyond ConvNet characteristics [12, 51]. To address this, we propose `einspace`: a neural architecture search space based on a parameterised probabilistic context-free grammar (CFG), which is highly expressive and able to represent various network configurations including diverse state-of-the-art architectures like ResNets, transformers, and the MLP-Mixer. This configuration allows for the integration of tried-and-tested architectures as powerful priors, making the most out of architectural research to date.

To demonstrate the effectiveness of `einspace`, we perform experiments on the Unseen NAS [14] datasets—eight diverse classification tasks including vision, language, audio, and chess problems—using simple random and evolutionary search strategies. We find that in such an expressive search space, the choice of search strategy is important and random search underperforms. When using the powerful priors of human-designed architectures to initialise an evolutionary search, we consistently find both large performance gains and significant architectural changes. Code to reproduce our experiments is available in the supplementary material.

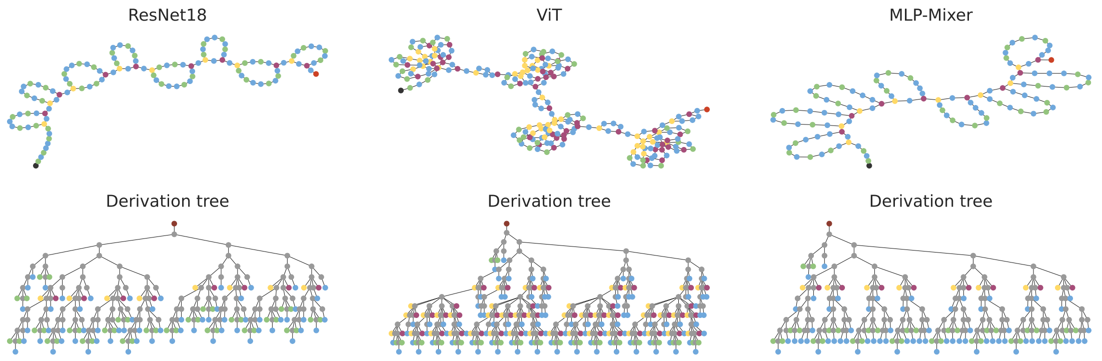

Figure 1: Three state-of-the-art architectures and their associated derivation trees within `einspace`. Top row shows the architectures where the black node is the input tensor and the red is the output. Bottom row shows derivation trees where the top node represents the starting symbol, the grey internal nodes the non-terminals and the leaf nodes the terminal operations. See Section 2.1 for details on other node colouring. Best viewed with digital zoom.

2 `einspace`: A Search Space of Fundamental Operations

Our neural architecture search space, `einspace`[1] is introduced here. Based on a parameterised probabilistic CFG, it provides an expressive space containing several state-of-the-art neural architectures. We first describe the groups of operations we include in the space, then how macro structures are represented. We then present the CFG that defines the search space and finally, in the appendix, its parameterised and probabilistic extensions.

As a running example we will be constructing a simple convolutional block with a skip connection within `einspace`, explaining at each stage how it relates to the architecture. The block will consist of a convolution, a normalisation and an activation, wrapped inside a skip connection.

2.1 Fundamental Operations

Each fundamental operation in `einspace` takes as input a tensor, either from the network input or a preceding operation, and processes it further. The operations can be categorised into four groups based on their role in the network architecture. The classifications one-to-one, one-to-many, and many-to-one describe the number of input and output tensors for the functions within each group.
Branching. *One-to-many* functions that direct the flow of information through the network by cloning or splitting tensors. Examples include the branching within self-attention modules into queries, keys and values. In our visualisations, these are coloured yellow.
Aggregation. *Many-to-one* functions that merge multiple tensors into one. Examples include matrix multiplication, summation and concatenation. In our visualisations, these are coloured **purple**.
Routing. *One-to-one* functions that change the shape or the order of the content in a tensor without altering its information. Examples include axis permutations as well as the `im2col` and `col2im` operations. In our visualisations, these are coloured **green**.
Computation. *One-to-one* functions that alter the information of the tensor, either by parameterised operations, normalisation or non-linearities. Examples include linear layers, batch norm and activations like ReLU and softmax. In visualisations, these are coloured **blue**.

In our example, the skip connection will be handled by a combination of branching and aggregation functions, the convolution is decomposed into the routing functions `im2col` and `col2im`, with a `linear` layer from the computation group between them. The normalisation and activation come from the computation group. In the next subsection, we discuss the larger structures of the architecture.

[1]The name is inspired by the generality of the *Einstein summation* and related Python library `einops` [38] as many of our operations can be implemented in it.

2.2 Macro Structure

The groups of functions above describe the fundamental operations that make up an architecture. We now describe how these functions are composed in different ways to form larger components.

A *module* is defined as a composition of functions from above that takes one input tensor and produces one output tensor, with potential branching inside. A module may contain multiple *computation* and *routing* operations, but each *branching* must be paired with a subsequent *aggregation* operation. Thus, the whole network can be seen as a module that takes a single tensor as input and outputs a single prediction. A network module may itself contain multiple modules, directly pertaining to the hierarchical phrase nature of CFG structures. We divide modules into four types, visualised in Fig. 2.

Sequential module. A pair of modules and/or functions that are applied to the input tensor sequentially. Using our grammar, defined in Sec.2.3, this can be produced using the rule (M ⟶ M M), or equivalently from the starting symbol S. This also applies to the rules below.

Branching module. A branching function first splits the input into multiple branches. Each branch is processed by some inner set of modules and/or functions. The outputs of all branches are subsequently merged in an aggregation function. In the grammar below this can be produced by the rule (M ⟶ B M A).

Routing module. A routing function is applied, followed by a module and/or function. A final routing function then processes the tensor. In the grammar below this is produced by the rule (M ⟶ R1 M R2).

Computation module. This module only contains a single function, selected from the one-to-one computation functions described above. While this module is trivial, we will see later how its inclusion is helpful when designing our CFG and its probabilistic extension. In the grammar below this is produced by the rule (M ⟶ C).

To construct our example, we will use all four modules. The branching module combines the `clone` and `add` functions from before to create a 2-branch structure. One branch is a simple skip connection by using the `identity` function inside a computation module. The other branch is the more complex sequence. The convolutional layer is created by combining `im2col`, `linear` and `col2im` in a routing module. The norm and activation are each wrapped in a computation module and these are all composed in sequential modules. Fig. 2 shows similar module instantiations in action.

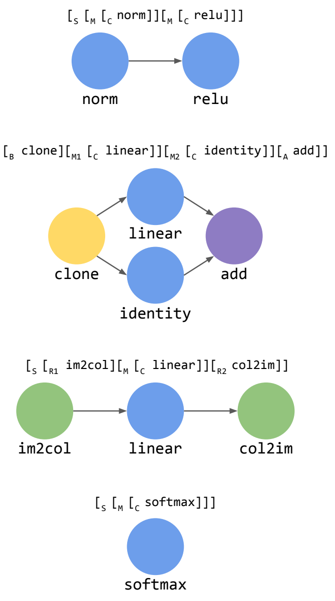
[$_S$ [$_M$ [$_C$ norm]][$_M$ [$_C$ relu]]]
norm
relu
[$_S$ [$_B$ clone][$_{M1}$ [$_C$ linear]][$_{M2}$ [$_C$ identity]][$_A$ add]]
linear
clone
add
identity
[$_S$ [$_{R1}$ im2col][$_M$ [$_C$ linear]][$_{R2}$ col2im]]
im2col
linear
col2im
[$_S$ [$_M$ [$_C$ softmax]]]
softmax

Figure 2: Visualisation of example modules with their CFG derivations in bracket notation. From top to bottom; sequential, branching, routing and computation modules.

2.3 Search Space as a Context-Free Grammar

```
 S  →  M M | B M A | R1 M R2,
 M  →  M M | B M A | R1 M R2 | C,
 B  →  clone | group-dim,
 A  →  matmul | add | concat,
R1  →  identity | im2col | permute,
R2  →  identity | col2im | permute,
 C  →  identity | linear | norm | relu | softmax | pos-enc.
```

The CFG above defines our `einspace`, where uppercase symbols represent non-terminals and lowercase represent terminals. The colours refer to the function groups.

Table 1: NAS performance in `einspace` with the simple search strategies of random sampling, random search, and regularised evolution (RE). See text for further detail. We evaluate performance across multiple datasets and modalities from Unseen NAS [14]. Search space results transcribed from [14] are denoted *, where DARTS [25] and Bonsai [13] search spaces are employed. The expressiveness of `einspace` enables performance that remains competitive with significantly more elaborate search strategies. We highlight **best** and second best performance per dataset.

	Baselines				Regularised evolution (RE) `einspace`			Rand. Search			Rand. Sampl.
Dataset	RN18	PC-DARTS*	Dr NAS*	Bonsai-Net*	RE (RN18)	RE (Mix)	RE (Scratch)	DARTS*	Bonsai*	`einspace`	`einspace`
AddNIST	93.36	96.60	97.06	**97.91**	97.54	97.72	83.87	97.07	34.17	67.00	10.13
Language	92.16	90.12	88.55	87.65	96.84	**97.92**	88.12	90.12	76.83	87.01	35.26
MultNIST	91.36	96.68	**98.10**	97.17	96.37	92.25	93.72	96.55	39.76	66.09	18.87
CIFARTile	47.13	**92.28**	81.08	91.47	60.65	62.76	30.89	90.74	24.76	30.90	25.25
Gutenberg	43.32	49.12	46.62	48.57	**54.02**	50.16	36.70	47.72	29.00	39.58	19.69
Isabella	63.65	65.77	64.53	64.08	64.30	62.72	56.33	**66.35**	58.53	56.90	32.24
GeoClassing	90.08	94.61	**96.03**	95.66	95.31	95.13	60.43	95.54	63.56	69.13	24.35
Chesseract	59.35	57.20	58.24	60.76	60.31	61.86	59.50	59.16	**68.83**	61.46	44.83
Average acc. ↑	72.55	80.30	78.78	**80.41**	78.17	77.56	63.70	**80.41**	49.43	59.76	26.32
Average rank ↓	6.50	4.44	4.38	**3.50**	3.75	3.88	8.12	4.06	8.62	7.88	10.88

Our networks are all constructed according to the high-level blueprint: `backbone` $\longrightarrow$ `head` where `head` is a predefined module that takes an output feature from the backbone and processes it into a prediction (see Appendix C for more details). The `backbone` is thus the section of the network that is generated by the above CFG. When searching for architectures we search different backbones.

For a thorough example of how sampling and mutation is performed in `einspace`, we recommend looking at Appendix B, where we also present a full derivation tree of our running example.

3 Experiments

In this section we evaluate the effectiveness of `einspace` as a NAS search space. We use simple random search and evolutionary search strategies on the diverse Unseen NAS datasets (Table 1) and on NAS-Bench-360 (Table 5 in the appendix).
For implementation details, see appendix Section C.

3.1 Random Sampling and Search

In previous NAS search spaces e.g. [12, 25, 51], complex search methods often perform very similarly to random search [22, 52]. Indeed, we can see this in Tab. 1 comparing the PC-DARTS strategy to DARTS random search.

However for `einspace`, this is not the case for most datasets. Random sampling improves on pure random guessing (not shown), but is far from the baseline performance of a ResNet18. The random search baseline is also far behind, but intriguingly outperforms baseline NAS approaches on Chesseract.

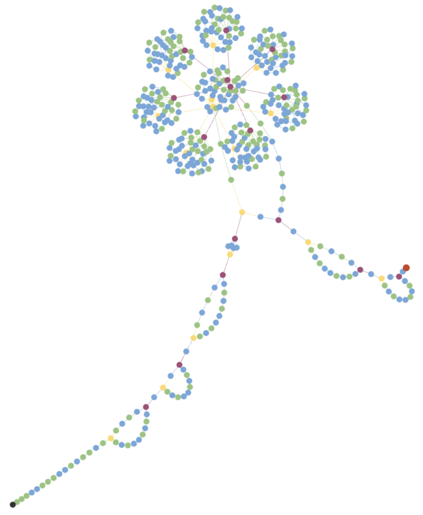
Figure 3: The top RE(Mix) architecture on AddNIST, found in `einspace`.

3.2 Evolutionary Search from Scratch

We now turn to a more sophisticated search strategy. We perform regularised evolution in `einspace` for 1000 iterations across all datasets, initialising the population with 100 random samples. In Tab. 1 the results are shown in the column named RE(Scratch). The performance of this strategy is significantly higher than random search on several datasets, indicating that the search strategy is more important in an expressive search space like `einspace` compared to DARTS. Compared to the top performing NAS methods, however, it is significantly behind on some datasets.

3.3 Evolutionary Search from Existing SOTA Architectures

To fully use the powerful priors of existing human-designed structures, we now search with the initial population of our evolutionary search seeded with a set of existing state-of-the-art architectures. We first seed the entire population with the ResNet18 architecture. The search applies mutations to these networks for 500 iterations. In Tab. 1, these results can be found in the RE(RN18) column.

To further highlight the expressivity of `einspace`, we perform experiments with the initial population as a mix of ResNet18, WRN16-4, ViT and MLP-Mixer architectures. To our knowledge, no other NAS space is able to represent such a diverse set of architectures in a single space. These results are shown in the RE(Mix) column.

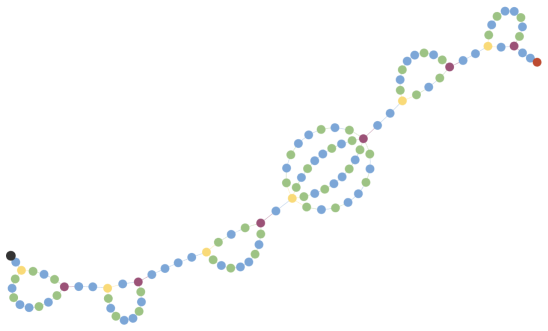
Figure 4: The best model on the Language dataset, found by RE(Mix) in `einspace`.

On **every single task**, we find an improved version of the initial architecture using RE(RN18) and on all but one using RE(Mix). Moreover, in some cases we beat existing state-of-the-art, especially on tasks further from the traditional computer vision setting. In particular, where previous NAS methods fail—i.e. the Language dataset—the architecture in Fig. 4 has a direct improvement over the ResNet18 by 5.76%. See also the architecture in Fig. 3 and the collection in Fig. 8 in the Appendix for the breadth of structures that are found in `einspace`.

4 Discussion

Limitations Our search space, designed for diversity, is extremely large and uniquely unbounded in terms of depth and width. This complexity makes formulating one-shot models like ENAS [31] or DARTS [25] challenging. Instead, developing an algorithm to learn the probabilities of the PCFG might be more viable. This approach must address the fact that sampling probabilities do not consider network depth or previous decisions, although this could be mitigated by using the parameters outlined in Section B.3. Furthermore, while `einspace` is diverse, it lacks options for recurrent computation as found in RNNs and the new wave of state-space models like Mamba [15]. We also chose to keep the options for activations and normalisation layers narrow since in practice the benefit from changing these tends to be small.

Broader Impact Statement After careful reflection, the authors have determined that this work presents no notable negative impacts to society or the environment.

Conclusion We have introduced `einspace`: an expressive NAS search space based on a parameterised probabilistic CFG. We show that our work enables the construction of a comprehensive and diverse range of existing state-of-the-art architectures and can further facilitate discovery of novel architectures directly from fundamental operations. With only simple search strategies, we report competitive resulting architectures across a diverse set of tasks, highlighting the potential value of defining highly expressive search spaces. We further demonstrate the utility of initialising search with existing architectures as priors. We believe that future work on developing intelligent search strategies within `einspace` can lead to exciting advancements in neural architectures.

Acknowledgements. The authors are grateful to Joseph Mellor, Henry Gouk, and Thomas L. Lee for helpful suggestions. Funding for this research is provided by an EPSRC New Investigator Award (EP/X020703/1).

References

[1] Bender, G., Kindermans, P.-J., Zoph, B., Vasudevan, V., and Le, Q. V. (2018). Understanding and simplifying one-shot architecture search. In *International Conference on Machine Learning*.

[2] Bender, G., Liu, H., Chen, B., Chu, G., Cheng, S., Kindermans, P.-J., and Le, Q. V. (2020). Can weight sharing outperform random architecture search? An investigation with tunas. In *Proceedings of the IEEE/CVF Conference on computer vision and pattern recognition*.

[3] Cai, H., Gan, C., Wang, T., Zhang, Z., and Han, S. (2020). Once-for-all: Train one network and specialize it for efficient deployment. In *International Conference on Learning Representations*.

[4] Cai, H., Zhu, L., and Han, S. (2019). Proxylessnas: Direct neural architecture search on target task and hardware. In *International Conference on Learning Representations*.

[5] Chen, A., Dohan, D., and So, D. (2023). Evoprompting: Language models for code-level neural architecture search. *Advances in Neural Information Processing Systems*.

[6] Chen, M., Peng, H., Fu, J., and Ling, H. (2021). Autoformer: Searching transformers for visual recognition. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*.

[7] Chen, X., Wang, R., Cheng, M., Tang, X., and Hsieh, C.-J. (2020). DrNAS: Dirichlet neural architecture search. In *International Conference on Learning Representations*.

[8] Chi, Z. (1999). Statistical properties of probabilistic context-free grammars. *Computational Linguistics*, 25(1):131–160.

[9] Chu, X., Zhang, B., and Xu, R. (2021). FairNAS: Rethinking evaluation fairness of weight sharing neural architecture search. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*.

[10] Ci, Y., Lin, C., Sun, M., Chen, B., Zhang, H., and Ouyang, W. (2021). Evolving search space for neural architecture search. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*.

[11] Cohen, S. (2017). Latent-Variable PCFGs: Background and applications. In *Proceedings of the 15th Meeting on the Mathematics of Language*.

[12] Dong, X. and Yang, Y. (2019). Nas-bench-201: Extending the scope of reproducible neural architecture search. In *International Conference on Learning Representations*.

[13] Geada, R., Prangle, D., and McGough, A. S. (2020). Bonsai-net: One-shot neural architecture search via differentiable pruners. *arXiv preprint arXiv:2006.09264*.

[14] Geada, R., Towers, D., Forshaw, M., Atapour-Abarghouei, A., and McGough, A. S. (2024). Insights from the use of previously unseen neural architecture search datasets. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*.

[15] Gu, A. and Dao, T. (2023). Mamba: Linear-time sequence modeling with selective state spaces. *arXiv preprint arXiv:2312.00752*.

[16] Guo, Z., Zhang, X., Mu, H., Heng, W., Liu, Z., Wei, Y., and Sun, J. (2020). Single path one-shot neural architecture search with uniform sampling. In *Proceedings of the European Conference on Computer Vision*.

[17] He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*.

[18] Hopcroft, J. E., Motwani, R., and Ullman, J. D. (2001). Introduction to automata theory, languages, and computation. *ACM SigAct News*, 32(1):60–65.

[19] Hu, J., Shen, L., and Sun, G. (2018). Squeeze-and-excitation networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*.

[20] Kandasamy, K., Neiswanger, W., Schneider, J., Poczos, B., and Xing, E. P. (2018). Neural architecture search with bayesian optimisation and optimal transport. *Advances in Neural Information Processing Systems*.

[21] Li, C., Tang, T., Wang, G., Peng, J., Wang, B., Liang, X., and Chang, X. (2021). BossNAS: Exploring hybrid CNN-transformers with block-wisely self-supervised neural architecture search. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*.

[22] Li, L. and Talwalkar, A. (2019). Random search and reproducibility for neural architecture search. In *Conference on Uncertainty in Artificial Intelligence*.

[23] Liu, C., Chen, L.-C., Schroff, F., Adam, H., Hua, W., Yuille, A. L., and Fei-Fei, L. (2019a). Auto-deeplab: Hierarchical neural architecture search for semantic image segmentation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*.

[24] Liu, H., Simonyan, K., Vinyals, O., Fernando, C., and Kavukcuoglu, K. (2018). Hierarchical representations for efficient architecture search. In *International Conference on Learning Representations*.

[25] Liu, H., Simonyan, K., and Yang, Y. (2019b). DARTS: Differentiable architecture search. In *International Conference on Learning Representations*.

[26] Lu, Z., Whalen, I., Boddeti, V., Dhebar, Y., Deb, K., Goodman, E., and Banzhaf, W. (2019). NSGA-Net: neural architecture search using multi-objective genetic algorithm. In *Proceedings of the Genetic and Evolutionary Computation Conference*.

[27] Ma, N., Zhang, X., Zheng, H.-T., and Sun, J. (2018). Shufflenet v2: Practical guidelines for efficient cnn architecture design. In *Proceedings of the European Conference on Computer Vision*.

[28] Manning, C. and Schutze, H. (1999). *Foundations of statistical natural language processing*. MIT press.

[29] Mendoza, H., Klein, A., Feurer, M., Springenberg, J. T., and Hutter, F. (2016). Towards automatically-tuned neural networks. In *Proceedings of the Workshop on Automatic Machine Learning*. PMLR.

[30] Miikkulainen, R., Liang, J., Meyerson, E., Rawal, A., Fink, D., Francon, O., Raju, B., Shahrzad, H., Navruzyan, A., Duffy, N., et al. (2024). Evolving deep neural networks. In *Artificial Intelligence in the Age of Neural Networks and Brain Computing*, pages 269–287. Elsevier.

[31] Pham, H., Guan, M., Zoph, B., Le, Q., and Dean, J. (2018). Efficient neural architecture search via parameters sharing. In *International Conference on Machine Learning*.

[32] Radosavovic, I., Johnson, J., Xie, S., Lo, W.-Y., and Dollár, P. (2019). On network design spaces for visual recognition. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*.

[33] Ramachandran, P., Zoph, B., and Le, Q. V. (2017). Searching for activation functions. *arXiv preprint arXiv:1710.05941*.

[34] Real, E., Aggarwal, A., Huang, Y., and Le, Q. V. (2019). Regularized evolution for image classifier architecture search. In *Proceedings of the AAAI Conference on Artificial Intelligence*.

[35] Real, E., Liang, C., So, D., and Le, Q. V. (2020). AutoML-zero: Evolving machine learning algorithms from scratch. In *International Conference on Machine Learning*.

[36] Real, E., Moore, S., Selle, A., Saxena, S., Suematsu, Y. L., Tan, J., Le, Q. V., and Kurakin, A. (2017). Large-scale evolution of image classifiers. In *International Conference on Machine Learning*.

[37] Roberts, N., Khodak, M., Dao, T., Li, L., Ré, C., and Talwalkar, A. (2021). Rethinking neural operations for diverse tasks. *Advances in Neural Information Processing Systems*.

[38] Rogozhnikov, A. (2022). Einops: Clear and reliable tensor manipulations with einstein-like notation. In *International Conference on Learning Representations*.

[39] Ru, R., Esperanca, P., and Carlucci, F. M. (2020). Neural architecture generator optimization. *Advances in Neural Information Processing Systems*.

[40] Schrodi, S., Stoll, D., Ru, B., Sukthanker, R., Brox, T., and Hutter, F. (2024). Construction of hierarchical neural architecture search spaces based on context-free grammars. *Advances in Neural Information Processing Systems*.

[41] So, D., Mańke, W., Liu, H., Dai, Z., Shazeer, N., and Le, Q. V. (2021). Primer: Searching for efficient transformers for language modeling. In *Advances in Neural Information Processing Systems*.

[42] Springenberg, J. T., Dosovitskiy, A., Brox, T., and Riedmiller, M. (2015). Striving for Simplicity: The All Convolutional Net. In *ICLR Workshops*.

[43] Tan, M., Chen, B., Pang, R., Vasudevan, V., Sandler, M., Howard, A., and Le, Q. V. (2019). MnasNet: Platform-aware neural architecture search for mobile. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*.

[44] Tan, M. and Le, Q. V. (2019). Mixconv: Mixed depthwise convolutional kernels. In *Proceedings of the British Machine Vision Conference*.

[45] Tolstikhin, I., Houlsby, N., Kolesnikov, A., Beyer, L., Zhai, X., Unterthiner, T., Yung, J., Steiner, A., Keysers, D., Uszkoreit, J., Lucic, M., and Dosovitskiy, A. (2021). MLP-Mixer: An all-MLP Architecture for Vision. In *Advances in Neural Information Processing Systems*.

[46] Tu, R., Roberts, N., Khodak, M., Shen, J., Sala, F., and Talwalkar, A. (2022). NAS-Bench-360: Benchmarking Neural Architecture Search on Diverse Tasks. In *Neural Information Processing Systems Datasets and Benchmarks Track*.

[47] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017). Attention is all you need. In *Advances in Neural Information Processing Systems*.

[48] White, C., Neiswanger, W., and Savani, Y. (2021). BANANAS: Bayesian optimization with neural architectures for neural architecture search. In *Proceedings of the AAAI Conference on Artificial Intelligence*.

[49] White, C., Safari, M., Sukthanker, R., Ru, B., Elsken, T., Zela, A., Dey, D., and Hutter, F. (2023). Neural Architecture Search: Insights from 1000 papers. *arXiv preprint arXiv:2301.08727*.

[50] Xu, Y., Xie, L., Zhang, X., Chen, X., Qi, G.-J., Tian, Q., and Xiong, H. (2020). Pc-darts: Partial channel connections for memory-efficient architecture search. In *International Conference on Learning Representations*.

[51] Ying, C., Klein, A., Christiansen, E., Real, E., Murphy, K., and Hutter, F. (2019). Nas-bench-101: Towards reproducible neural architecture search. In *International Conference on Machine Learning*.

[52] Yu, K., Sciuto, C., Jaggi, M., Musat, C., and Salzmann, M. (2020). Evaluating the search phase of neural architecture search. In *International Conference on Learning Representations*.

[53] Zagoruyko, S. and Komodakis, N. (2016). Wide Residual Networks. In *Proceedings of the British Machine Vision Conference*.

[54] Zhang, X., Huang, Z., Wang, N., Xiang, S., and Pan, C. (2020). You only search once: Single shot neural architecture search via direct sparse optimization. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 43(9):2891–2904.

[55] Zhong, Z., Yan, J., Wu, W., Shao, J., and Liu, C.-L. (2018). Practical block-wise neural network architecture generation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*.

[56] Zoph, B. and Le, Q. V. (2017). Neural architecture search with reinforcement learning. In *International Conference on Learning Representations*.

[57] Zoph, B., Vasudevan, V., Shlens, J., and Le, Q. V. (2018). Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*.

Submission Checklist

1. For all authors. . .

   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes] All claims accurately reflect the paper's contributions and scope.

   (b) Did you describe the limitations of your work? [Yes] We discuss limitations in section 4.

   (c) Did you discuss any potential negative societal impacts of your work? [Yes] We discuss potential negative societal impacts in section 4.

   (d) Did you read the ethics review guidelines and ensure that your paper conforms to them? `https://2022.automl.cc/ethics-accessibility/` [Yes] We have read the guidelines and ensured we conform to them.

2. If you ran experiments. . .

   (a) Did you use the same evaluation protocol for all methods being compared (e.g., same benchmarks, data (sub)sets, available resources)? [Yes] For all experiments we ran, we ensured the same evaluation protocol. This is detailed in section C of the appendix.

   (b) Did you specify all the necessary details of your evaluation (e.g., data splits, pre-processing, search spaces, hyperparameter tuning)? [Yes] The experimental details are available in section C of the appendix.

   (c) Did you repeat your experiments (e.g., across multiple random seeds or splits) to account for the impact of randomness in your methods or data? [No] Due to the cost of these types of experiments, we could only afford one run per dataset, and we decided to prioritise breadth of tasks instead of multiple runs for fewer tasks.

   (d) Did you report the uncertainty of your results (e.g., the variance across random seeds or splits)? [No] As we could not run over multiple seeds, this was not possible. We do however report the average accuracy and ranks across the methods compared in Table 1.

   (e) Did you report the statistical significance of your results? [No] We have not included results on the statistical significance of the results across datasets, though we could add this if requested.

   (f) Did you use tabular or surrogate benchmarks for in-depth evaluations? [N/A] This paper focuses on a new search space, and hence could not make use of tabular or surrogate benchmarks with fixed search spaces.

   (g) Did you compare performance over time and describe how you selected the maximum duration? [No] Comparing performance over time for one-shot methods vs. our multi-trial evolutionary search is non-trivial due to their fundamental differences. According to the authors of Unseen NAS [14], they ran their methods for around 24 hours each. We ran ours for 48-96 hours and selected the duration based on doing 500 or 1000 iterations of search.

   (h) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes] This can be found in section C.4 of the appendix.

   (i) Did you run ablation studies to assess the impact of different components of your approach? [No] Running many searches over multiple versions of the search space is unfortunately too costly. We do however include an ablation of sorts on the architecture complexity of `einspace` in Table 6.

3. With respect to the code used to obtain your results…

   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results, including all requirements (e.g., `requirements.txt` with explicit versions), random seeds, an instructive `README` with installation, and execution commands (either in the supplemental material or as a URL)? [Yes] This has been included in the supplementary materials.

   (b) Did you include a minimal example to replicate results on a small subset of the experiments or on toy data? [Yes] This is described in the Readme of the codebase.

   (c) Did you ensure sufficient code quality and documentation so that someone else can execute and understand your code? [Yes] To the best of our ability, we have ensure good code quality and documentation in our large and complex codebase.

   (d) Did you include the raw results of running your experiments with the given code, data, and instructions? [No] The results and log files were not included in the supplementary materials.

   (e) Did you include the code, additional data, and instructions needed to generate the figures and tables in your paper based on the raw results? [N/A] The necessary raw results files were not included.

4. If you used existing assets (e.g., code, data, models)…

   (a) Did you cite the creators of used assets? [Yes] We describe the datasets we use, along with citations to the authors, in C.3.

   (b) Did you discuss whether and how consent was obtained from people whose data you're using/curating if the license requires it? [N/A] The Unseen NAS datasets are released under the CC By 4.0 license and NAS-Bench-360 under the MIT license, requiring no additional consent.

   (c) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A] The data does not contain personally identifiable information or offensive content.

5. If you created/released new assets (e.g., code, data, models)…

   (a) Did you mention the license of the new assets (e.g., as part of your code submission)? [Yes] Our code is released under the MIT license.

   (b) Did you include the new assets either in the supplemental material or as a URL (to, e.g., GitHub or Hugging Face)? [Yes] Our code is included in the supplementary materials.

6. If you used crowdsourcing or conducted research with human subjects…

   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A] No crowdsourcing or human subjects were used.

   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A] No crowdsourcing or human subjects were used.

   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A] No crowdsourcing or human subjects were used.

7. If you included theoretical results…

   (a) Did you state the full set of assumptions of all theoretical results? [N/A] No theoretical results are claimed.

(b) Did you include complete proofs of all theoretical results? [N/A] No theoretical results are claimed.

A Background

Neural architecture search

The search space used in NAS has a significant impact on results [52, 54]. This has facilitated the need to investigate search space design alongside the actual search algorithms [32]. Early macro design spaces [20, 56] made use of naive building blocks while accounting for skip connections and branching layers. Further design strategies have looked at chain-structured [3, 4, 6, 37], cell-based [13, 25, 55, 57] and hierarchical approaches. Hierarchical search spaces have been shown to be expressive and effective in reducing search complexity and methods include factorised approaches [43], n-level hierarchical assembly [23, 24, 41], parameterisation of hierarchical random graph generators [39] and topological evolutionary strategies [30]. Additional work on search spaces have proposed new candidate operations and module designs such as hand-crafted multi-branch cells [44], tree-structures [4], shuffle operations [27], dynamic modules [19], activation functions [33] and evolutionary operators [10]. In AutoML-Zero [35], the authors try to remove human bias from search space construction by defining a space of basic mathematical operations as building blocks.

The pioneering work of [40] constructs search spaces using CFGs. We take this direction further and construct `einspace` as a probabilistic CFG allowing for unbounded derivations, balanced by careful tuning of the branching rate. We aim to strike a balance of the level of complexity in the search space while incorporating components from diverse state-of-the-art architectures. Crucially, our space enables flexibility in both macro structure and at the individual operation level. While previous search spaces can be instantiated for specific architecture classes, our single space incorporates multiple classes in one, ConvNets, transformers and MLP-only architectures. Such hybrid spaces have been explored before [21], but they have been limited in their flexibility, offering only direct choices between convolution and attention operations and not allowing the construction of novel components.

Prominent search strategies include Bayesian optimisation [29, 48], reinforcement learning [55–57] and genetic algorithms [5, 34, 36]. A popular thread of work, towards improving computational efficiency via amortising training cost, involves the sharing of weights between different architectures via a supernet [1, 4, 9, 16, 25, 26]. Efficiency has been further improved by sampling only a subset of supernet channels [50], thus reducing both space exploration redundancies and memory consumption. Alternative routes to mitigating space requirements have considered both architecture and operation-choice pruning [7, 13]. We however highlight that random search often proves to be a very strong baseline [22, 52]; a consequence of searching within narrow spaces. This is commonly the case for highly engineered search spaces that contain a high fraction of strong architectures [49]. Contrasting this, in our `einspace` we see that random search across many tasks performs poorly, underpinning the value of a good search strategy for large, diverse search spaces [2, 35].

Context-free grammars

A context-free grammar (CFG; [18]) is a tuple (N,Σ,R,S), where N is a finite set of *non-terminal* symbols, Σ is a finite set of *terminal* symbols, R is the set of *production rules*—where each rule maps a non-terminal $A \in N$ to a string of non-terminal or terminals $A \to (N \cup \Sigma)^+$—and S is the *starting symbol.* A CFG describes a context-free language, containing all the strings that the CFG can generate. By recursively selecting a production, starting with the rules containing the starting symbol, we can generate strings within the grammar. CFGs can be *parameterised*: each non-terminal, in each rule in R, is annotated with parameters $p_1,\dots,p_n$ that influence the production (right hand side parameters have to appear in the left hand side). These parameters can condition production, based on an external state or contextual information, thus extending the power of the grammar.

A *probabilistic* context-free grammar (PCFG) associates each production rule with a probability [28]. These define the likelihood of selecting a particular rule given a parent non-terminal. These probabilities allow for stochastic string generation.

B Search Space Details

B.1 Prior Choices

When designing a search space, we must balance the need for flexibility—which allows more valid architectures to be included—and constraints – which reduce the size of the search space. We can view constraints as imposing priors on which architectures we believe are worth including. As discussed, many previous frameworks are too restrictive; therefore, we aim to impose minimal priors, listed below.

Convolutional prior. We design our routing module to enable convolutions to be easily constructed, while also allowing components like patch embeddings and transpose operations. We thus enforce that a routing function is followed by another routing function later in the module. Moreover, `im2col` only appears in the production rule of the first routing function (`R1`) and `col2im` in the last (`R2`). As shown in Fig. 2, to construct a convolution, we start from the rule (`M` $\longrightarrow$ `R1 M R2`) and derive the following (`R1` $\longrightarrow$ `im2col`), (`M` $\longrightarrow$ `C` $\longrightarrow$ `linear`) and (`R2` $\longrightarrow$ `col2im`).

Branching prior. We also impose a prior on the types of branching that can occur in a network. The branching functions `clone` and `group-dim` can each have a branching factor of 2, 4 or 8. For a factor of 2, we allow each inner function to be unique, processing the two branches in potentially different ways. For branching factors of 4 or 8, the inner function `M` is repeated as is, processing all branches identically (though all inner functions are always initialised with separate parameters). Symbolically, given a branching factor of 2 we have ($\mathtt{BM_1M_2A}$) but with a branching factor of 4 we have ($\mathtt{BM_1M_1M_1M_1A}$). Examples of components instantiated by a branching factor of 2 include skip connections, and for 4, or 8, multi-head attention.

B.2 Feature Mode

Different neural architectures operate on different feature shapes. ConvNets maintain 3D features throughout most of the network while transformers have 2D features. To enable such different types of computations in the same network, we introduce the concept of a *mode* [2] that affects the shape of our features and which operations are available at that point in the network. Before and after each module, we fix the feature tensor to be of one of two specific shapes, depending on which mode we are in.

Im mode. Maintains a 3D tensor of shape `(C, H, W)`, where `C` is the number of channels, `H` is the height and `W` is the width. Most convolutional architectures operate in this mode.

Col mode. Maintains a 2D tensor of shape `(S, D)`, where `S` is the sequence length and `D` is the token dimensionality. This is the mode in which most transformer architectures operate.

The mode is changed by the routing functions `im2col` and `col2im`. Most image datasets will provide inputs in the `Im` mode, while most tasks that use a language modality will provide it in `Col` mode.

Our example architecture maintains the `Im` mode at almost all stages, apart from inside the routing modules where the `im2col` function briefly puts us in the `Col` mode before `col2im` brings us back.

[2] Note that this is similar but not the same as the *mode* of a general tensor, which determines the number of dimensions of that tensor. We use the term mode to refer to the state that a particular part of the architecture is in.

B.3 Parameterising the Grammar

Due to the relatively weak priors we have put on the search space, sampling a new architecture naïvely will often lead to invalid networks. For example, the shape of the output tensor of one operation may not match the expected input shape of the next. Alternatively, the branching factor of a branching function may not match the branching factor of its corresponding aggregation function.

We therefore extend the grammar with parameters. Each rule r now has an associated set of parameters (s, m, b) which defines in what situations this rule can occur. When we sample an architecture from the grammar, we start by assigning parameter values based on the expected input to the architecture. For example, they might be the input tensor shape, feature mode and branching factor:

$$(s=[3,224,224], m=\texttt{Im}, b=1). \tag{1}$$

Given this, we can continuously infer the current parameters during each stage of sampling by knowing how each operation changes them. When we expand a production rule, we must choose a rule which has matching parameters. If at some point, the sampling algorithm has no available valid options, it will backtrack and change the latest decision until a full valid architecture is found. Hence, we can ensure that we can sample architectures without risk of obtaining invalid ones.

As an example of this, the CFG rule for `R1` was previously

$$\texttt{R1} \quad \longrightarrow \quad \texttt{identity} \quad | \quad \texttt{im2col} \quad | \quad \texttt{permute}. \tag{2}$$

Enhanced with parameters, this now becomes two rules

$$\texttt{R1}(m=\texttt{Im}) \quad \longrightarrow \quad \texttt{identity} \quad | \quad \texttt{im2col} \quad | \quad \texttt{permute}, \tag{3}$$

$$\texttt{R1}(m=\texttt{Col}) \quad \longrightarrow \quad \texttt{identity} \quad | \quad \texttt{permute}. \tag{4}$$

This signifies that an `im2col` operation is not available in the `Col` mode. Similarly, the available aggregation options depend on the branching factor of the current branching module

$$\texttt{A}(b=2) \quad \longrightarrow \quad \texttt{matmul} \quad | \quad \texttt{add} \quad | \quad \texttt{concat}, \tag{5}$$

$$\texttt{A}(b=4) \quad \longrightarrow \quad \texttt{add} \quad | \quad \texttt{concat}, \tag{6}$$

$$\texttt{A}(b=8) \quad \longrightarrow \quad \texttt{add} \quad | \quad \texttt{concat}. \tag{7}$$

B.4 Balancing Architecture Complexity

When sampling an architecture, we construct a decision tree where non-leaf nodes represent decision points and leaf nodes represent architecture operations. In each iteration, we either select a non-terminal module to expand the architecture and continue sampling, or choose a terminal function to conclude the search at that depth. Continuously selecting modules results in a deeper, more complex network, whereas selecting functions leads to a shallower, simpler network. We can balance this complexity by assigning probabilities to our production rules, thereby making a PCFG. Recall our CFG rule

$$(\texttt{M} \quad \longrightarrow \quad \texttt{M} \;\; \texttt{M} \quad | \quad \texttt{B} \;\; \texttt{M} \;\; \texttt{A} \quad | \quad \texttt{R1} \;\; \texttt{M} \;\; \texttt{R2} \quad | \quad \texttt{C}). \tag{8}$$

If we choose one of the first three options we are delving deeper in the search tree since there is yet another `M` to be expanded, but if we choose ($\texttt{M} \rightarrow \texttt{C}$), the *computation-module*, then we will reach a terminal function. Thus, to balance the depth of our traversal and therefore expected architecture complexities, we can set probabilities for each of these rules:

$$p(\texttt{M} \rightarrow \texttt{M M} | \texttt{M}), \quad p(\texttt{M} \rightarrow \texttt{B M A} | \texttt{M}), \quad p(\texttt{M} \rightarrow \texttt{R1 M R2} | \texttt{M}), \quad p(\texttt{M} \rightarrow \texttt{C} | \texttt{M}). \tag{9}$$

The value of $p(\mathtt{M} \longrightarrow \mathtt{C} \mid \mathtt{M})$ is especially important as it can be interpreted as the probability that we will stop extending the search tree at the current location.

We could set these probabilities to match what we wish the expected depth of architectures to be, and for empirical results on the architecture complexity, see Tab. 6 of the Appendix. However, we can actually ensure that the CFG avoids generating infinitely long architecture strings by setting the probabilities such that the branching rate of the CFG is less than one [8]. For details of how, see the next section. So, as shown in Fig. 5, we set the computation module probability to $p(\mathtt{M} \longrightarrow \mathtt{C} \mid \mathtt{M}) = 0.32$ and the probabilities of the other modules to $\frac{1-0.32}{3}$. For simplicity, all other rule probabilities are uniform.

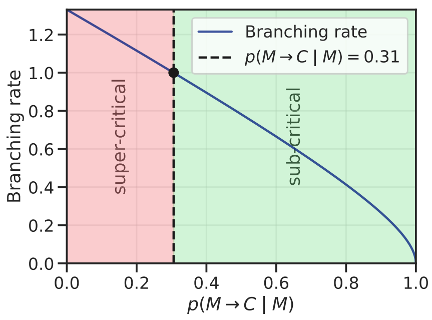

Figure 5: To ensure our CFG is consistent and does not generate infinite architectures, we keep the branching rate in the sub-critical region by setting $p(\mathtt{M} \longrightarrow \mathtt{C} | \mathtt{M}) > 0.31$.

B.5 Branching Rate of the CFG

If a PCFG is consistent, the probabilities of all finite derivations sum to one, or equivalently, it has a zero probability of generating infinite strings or derivations. For us, that means a sampled architecture can not be infinitely large, and that the sampling algorithm will halt with probability one. In order to check if a CFG is consistent, we can inspect the spectral radius ρ of its non-terminal expectation matrix [8]. If $\rho < 1$, then the PCFG is consistent. This expectation matrix is indexed by the non-terminals in the grammar (both the columns and the rows), and at each cell it provides the expected number of instances the column non-terminal being generated from the row non-terminal by summing the probabilities of the row non-terminal multiplied by the count of the column non-terminal in each rule.

We can also solve a (simple, in our case) set of linear equations in order to compute the expected length of an architecture string, ℓ, as a function of the rule probabilities. More specifically, denote by $\mathbb{E}[A]$ the expected length of string generated by a nonterminal in the grammar A. Then $\ell = \mathbb{E}[\mathtt{S}]$, where:

$$\mathbb{E}[\mathtt{S}] = \sum_{\mathtt{S} \longrightarrow \alpha} p(\mathtt{S} \longrightarrow \alpha \,|\, \mathtt{S}) \sum_i \mathbb{E}[\alpha_i] \tag{10}$$

$$\mathbb{E}[\mathtt{M}] = \sum_{\mathtt{M} \longrightarrow \alpha} p(\mathtt{M} \longrightarrow \alpha \,|\, \mathtt{M}) \sum_i \mathbb{E}[\alpha_i] \tag{11}$$

$$\mathbb{E}[\mathtt{B}] = 1 \tag{12}$$

$$\mathbb{E}[\mathtt{A}] = 1 \tag{13}$$

$$\mathbb{E}[\mathtt{R1}] = 1 \tag{14}$$

$$\mathbb{E}[\mathtt{R2}] = 1 \tag{15}$$

$$\mathbb{E}[\mathtt{C}] = 1 \tag{16}$$

In the above, α is the right hand side of a rule and α_i varies over the nonterminals in that right hand side.

B.6 Sampling

In this section, we explain the process of sampling an architecture from our parameterised PCFG through an example. We specifically focus on how the running example from the main paper, a simple convolutional block with a skip connection, could be generated.

We begin the process with the starting symbol `S`, which could produce several forms based on the production rules, including (`M M`), (`B M A`), or (`R1 M R2`). Since our block includes a skip connection, the macro structure of our architecture is best represented by a branching module (`B M A`).

Within this module, we expand the string from left to right, thereby starting with (`B`). The specific branching operation that fits our goal is ($\texttt{B} \rightarrow \texttt{clone}$) as we wish to later combine a transformed version of the tensor with itself. Since we have two branches, they are produced separately (see branching prior) and our module becomes ($\texttt{B}\ \texttt{M}_1\texttt{M}_2\ \texttt{A}$).

For the first branch, ($\texttt{M}_1$), we need a set of components that constitute a convolution followed by batch normalisation and an activation. Since this involves several composed operations we first expand into a sequential module ($\texttt{M}_1 \rightarrow \texttt{M}_3\texttt{M}_4$). The first of these operations represents the convolution. Within `einspace`, a convolution, $\texttt{conv}(x)$, is decomposed into the three operations, $\texttt{col2im}(\texttt{linear}(\texttt{im2col}(x)))$. Thus, in our grammar we unfold it as a routing module, ($\texttt{M}_3 \rightarrow \texttt{R1}\ \texttt{M}_5\ \texttt{R2}$) which further produces ($\texttt{R1} \rightarrow \texttt{im2col}$), ($\texttt{M}_5 \rightarrow \texttt{C} \rightarrow \texttt{linear}$) and ($\texttt{R2} \rightarrow \texttt{col2im}$). The normalisation and activation are then generated under ($\texttt{M}_4$), defined as another sequential module ($\texttt{M}_4 \rightarrow \texttt{M}_6\texttt{M}_7$) with ($\texttt{M}_6 \rightarrow \texttt{C} \rightarrow \texttt{norm}$) and ($\texttt{M}_7 \rightarrow \texttt{C} \rightarrow \texttt{relu}$). The second branch, $\texttt{M}_2$, acts as a skip connection and is thus derived as ($\texttt{M}_2 \rightarrow \texttt{C} \rightarrow \texttt{identity}$).

To finalise the architecture, the aggregation symbol (`A`) merges the tensors back into one unit. To complete the residual connection, we use the rule ($\texttt{A} \rightarrow \texttt{add}$).

Completing our running example, we present the full derivation of the architecture in the CFG in Fig. 6.

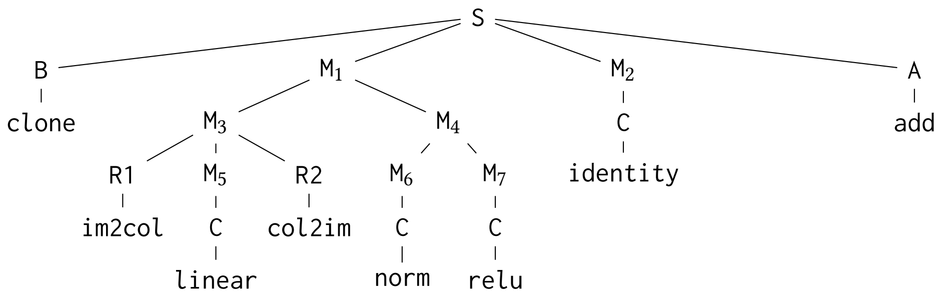

Figure 6: Example derivation tree of a traditional convolutional block with a skip connection.

B.7 Mutation

In the experiments that follow, we investigate how well a simple evolutionary search strategy performs in `einspace`. We therefore define what a *mutation* looks like in three straightforward steps:

1. *Sample a Node*: Choose a node randomly from the architecture's derivation tree. Each node has an equal chance of being selected, ensuring every part of the architecture can be modified. If a non-leaf node is chosen, a whole subtree will be mutated.

2. *Resample the Subtree*: After selecting a node, replace the subtree rooted at this node by regenerating it based on the grammar rules. This step allows the exploration of new configurations.

3. *Validate Architecture*: Check if the new architecture can produce a valid output in the forward pass, given an input of the expected shape, and that it fits within the resource constraints, e.g. GPU memory. If it does, accept it; otherwise, discard and retry the mutation.

As an example, suppose we have the derivation tree of Fig. 6. If we randomly pick the node ($\texttt{M}_4$) and resample it, we might produce another convolution+norm+relu component, making this whole architecture akin to a basic ResNet block. We then check if this new architecture is valid. If it is, this new architecture becomes a new individual in the population of the evolutionary search.

B.8 Fundamental Operations

Our grammar in the main paper is somewhat simplified. There are some fundamental operations that have hyperparameters that allow multiple versions to be chosen from. They are detailed here.

Branching functions. For the production rule (`B` → `clone` | `group-dim`), the full set of options is:

$$\begin{aligned}
\texttt{B} \quad \longrightarrow \quad & \texttt{clone(b=2)} \,|\, \texttt{clone(b=4)} \,|\, \texttt{clone(b=8)} \,| && (17)\\
& \texttt{group-dim(dim=1,b=2)}|\texttt{group-dim(dim=1,b=4)}|\texttt{group-dim(dim=1,b=8)}| && (18)\\
& \texttt{group-dim(dim=2,b=2)}|\texttt{group-dim(dim=2,b=4)}|\texttt{group-dim(dim=2,b=8)}| && (19)\\
& \texttt{group-dim(dim=3,b=2)}|\texttt{group-dim(dim=3,b=4)}|\texttt{group-dim(dim=3,b=8)}, && (20)
\end{aligned}$$

where `b` refers to the branching factor and `dim` is the dimension we group over.

Aggregation functions. Similarly, for the production rule (`A` → `matmul` | `add` | `concat`), the full set of options is:

$$\begin{aligned}
\texttt{A} \quad \longrightarrow \quad & \texttt{matmul(scaled=False)} \,|\, \texttt{matmul(scaled=True)} \,|\, \texttt{add} \,| && (21)\\
& \texttt{concat(dim=1,b=2)}|\texttt{concat(dim=1,b=4)}|\texttt{concat(dim=1,b=8)}| && (22)\\
& \texttt{concat(dim=2,b=2)}|\texttt{concat(dim=2,b=4)}|\texttt{concat(dim=2,b=8)}| && (23)\\
& \texttt{concat(dim=3,b=2)}|\texttt{concat(dim=3,b=4)}|\texttt{concat(dim=3,b=8)}, && (24)
\end{aligned}$$

where `scaled=True` makes the `matmul` operation equivalent to the scaled dot product used in many transformer architectures, `dim` is the dimension we concatenate over and `b` is the branching factor.

Routing functions. The `im2col` and `col2im` functions are implemented to offer the standard functionality that enables convolutional operations, including variables that set the kernel sizes, stride, dilation and padding. Below are the full set of options for `im2col`. The `col2im` only takes the predicted output shape as a parameter so we can include only a single version of this operation.

$$\begin{aligned}
& \texttt{im2col(k=1,s=1,p=0)}, \quad \texttt{im2col(k=1,s=2,p=0)}, && (25)\\
& \texttt{im2col(k=3,s=1,p=1)}, \quad \texttt{im2col(k=3,s=2,p=1)}, && (26)\\
& \texttt{im2col(k=4,s=4,p=0)}, \quad \texttt{im2col(k=8,s=8,p=0)}, \quad \texttt{im2col(k=16,s=16,p=0)}, && (27)
\end{aligned}$$

where `k` is the kernel size, `s` is the stride and `p` the padding.

For the permute operation, there are 6 versions of the order parameter o. There is one for the `Col` mode and 5 for the `Im` mode:

$$\begin{aligned}
& \texttt{permute(o=(2,1))}, && (28)\\
& \texttt{permute(o=(1,3,2))}, \quad \texttt{permute(o=(2,1,3))}, \quad \texttt{permute(o=(2,3,1))}, && (29)\\
& \texttt{permute(o=(3,1,2))}, \quad \texttt{permute(o=(3,2,1))}. && (30)
\end{aligned}$$

For completeness, the `identity` operation can also be included, making 2 in the `Col` mode (with `identity=permute(o=(1,2))`) and 6 in the `Im` mode (with `identity=permute(o=(1,2,3))`).

Computation functions. For linear layers, we vary the output dimension d across powers of two:

$$\begin{aligned}
& \texttt{linear(d=16)}, \quad \texttt{linear(d=32)}, \quad \texttt{linear(d=64)}, && (31)\\
& \texttt{linear(d=128)}, \quad \texttt{linear(d=256)}, \quad \texttt{linear(d=512)}, && (32)\\
& \texttt{linear(d=1024)}, \quad \texttt{linear(d=2048)}. && (33)
\end{aligned}$$

The `norm` operation takes on the `batch-norm` functionality in the `Im` mode and `layer-norm` in `Col` mode. The `softmax` is just a softmax operation applied to the final dimension, the `relu` activation is implemented as the single option `leaky-relu` and `pos-enc` is a learnable positional encoding.

B.9 Patch Embeddings and Convolutions

In this section we provide some more information about how the routing module can represent common components.

The routing module, (`M → R1 M R2`), puts a prior on certain types of operations inside our architectures. A patch embedding, such as those found in many transformers, is achieved by the following derivation: (`M → im2col linear identity`), while a convolution can be obtained by (`M → im2col linear col2im`). In terms of the process required to sample such combinations, the first is easy since there are no dependencies between the operations. The second, however, is more complicated and requires some discussion.

Let x be a tensor in $\mathbb{R}^{3\times32\times32}$ and let's consider a `routing module` containing the functions: `im2col`, `linear`, `col2im`. The functions will be applied in order to the input tensor, giving us

$$x' = \texttt{im2col}(x), \tag{34}$$

$$x'' = \texttt{linear}(x'), \tag{35}$$

$$y = \texttt{col2im}(x''). \tag{36}$$

The shapes of the intermediate and final tensors $\{x', x'', y\}$ depend on several function hyperparameters. These are listed below.

Table 2: Hyperparameters for the three functions involved in a convolution component.

`im2col`	`col2im`	`linear`
$k_{in} = 7$ (kernel size)	$k_{out} = 7$ (kernel size)	$c_{out} = 64$ (output channels)
$s_{in} = 2$ (stride)	$s_{out} = 2$ (stride)	
$p_{in} = 3$ (padding)	$p_{out} = 0$ (padding)	

The input tensor has height dimension h_{in} and width w_{in}. The `im2col` operation will extract column vectors from this space a number of times depending on the kernel size k_{in}, stride s_{in} and padding p_{in} values in the table above. The number of column vectors equals

$$l = \left\lfloor \frac{h_{in} + 2\times p_{in} - (k_{in} - 1) - 1}{s_{in}} + 1 \right\rfloor \times \left\lfloor \frac{w_{in} + 2\times p_{in} - (k_{in} - 1) - 1}{s_{in}} + 1 \right\rfloor, \tag{37}$$

which in our case gives $l = 256$. The shapes of all intermediate tensors can therefore be written as in Table 3.

Table 3: Tensor shapes in the forward pass of our convolution component. $c_{in} = 3, h_{in} = 32, w_{in} = 32$ in this example.

Tensor	Shape
x	$[c_{in}, h_{in}, w_{in}]$
x'	$[l,\, c_{in} \times k_{in} \times k_{in}]$
x''	$[l,\, c_{out}]$
y	$[c_{out},\, h_{out},\, w_{out}]$

Therefore, to successfully apply the `col2im` function, the constraint $l = h_{out} \times w_{out}$ must be satisfied. From Equation 37 we can see that the output height and width can be predicted by the `im2col` parameters.

$$h_{out} = \left\lfloor \frac{h_{in} + 2 \times p_{in} - (k_{in} - 1) - 1}{s_{in}} + 1 \right\rfloor \tag{38}$$

$$w_{out} = \left\lfloor \frac{w_{in} + 2 \times p_{in} - (k_{in} - 1) - 1}{s_{in}} + 1 \right\rfloor \tag{39}$$

So, in practice, the `im2col` operation fully defines the behaviour of the convolution—apart from the number of output channels defined by the linear layer—and the col2im only rearranges the tensor back into its correct 3D form. This is trivial in the case where $h_{in} = w_{in}$ since $h_{out} = w_{out} = \sqrt{l}$. However, if $h_{in} \neq w_{in}$, then the predicted output shapes must be remembered until the `col2im` operation is applied.

Thus, in our sampling and mutation algorithm, whenever an `im2ol` operation is sampled, we must store the predicted output shape until a corresponding `col2im` is applied. Additionally, the dimensionality of l must not change in the connecting branch as it would break the constraint $h_{out} = w_{out} = \sqrt{l}$.

C Implementation and Experimental Details

C.1 Search strategies

We explore three search strategies within `einspace`: random sampling, random search, and regularised evolution (RE). *Random sampling* estimates the average expected test performance from K random architecture samples. *Random search* samples K architectures and selects the best performer on a validation set. In *regularised evolution*, we start by constructing an initial population of 100 individuals, which are either randomly sampled from the search space or seeded with existing architectures. For $(K - 100)$ iterations, the algorithm then randomly samples 10 individuals and selects the one with the highest fitness as the parent. This parent is mutated to produce a new child. This child is evaluated and enters the population while the oldest individual in the population is removed, following a first-in-first-out queue structure. An architecture is mutated in three straightforward steps shown in Section B.7.

Note that these are very simple search strategies, and that there is huge potential to design more intelligent approaches, e.g. including crossover operations in the evolutionary search, using hierarchical Bayesian optimisation [40] or directly learning the probabilities of the CFG [11]. In this work, we focus on the properties of our search space and investigate whether simple search strategies are able to find good architectures, and leave investigations on more complex search strategies for future work.

C.2 Networks

Baseline networks

We use the convolutional baselines of ResNet18 [17] and WideResNet-16-4 [53]. Both stems use a 3× convolution instead of the standard 7×7 as most input shapes in the datasets we use are small. The former contains a max-pooling layer in the stem, which for simplicity we decide to not represent in our search space. Instead we modify the pooling operation and replace it with a 3x3 convolution with stride 2. This has been shown to be equally powerful [42] and in our experiments we find that it performs similarly. Our ViT model is a small 4-layer network with a patch size of 4, model width of 512, and 4 heads. The MLP-Mixer shares the same patch embedding stem with a patch size of 4. It has 8 layers and, similarly, a model width of 512. The channel mixer expands the dimension by 4 and the token mixer cuts it in half. The models have the following number of parameters (given an input image of shape [3, 64, 64]): Resnet18: 11.2M, WRN16-4: 2.8M, ViT: 4.4M and MLP-Mixer: 6.5M.

We compare our experimental results to PC-DARTS [50], DrNAS [7] and Bonsai-Net [13] with results transcribed from [14]. We also compare to the performance of a trained ResNet18 (RN18).

Network head
Every network that is instantiated contains a few common operations. For classification tasks, the network head looks like this: the output features of the sampled backbone are reduced to their channel dimension via `reduce('B C H W -> B C', 'mean')` or `reduce('B C H -> B C', 'mean')`, depending on if the features are in `Im` or `Col` mode. Second, a final linear layer that maps the channel dimension to the target dimension (i.e., the number of classes) is appended. For dense prediction tasks: the head contains an adaptive average pooling layer that upsamples the two final dimensions of the backbone features to the target image size. If the features are in the `Col` mode, we the insert a new dimension after the batch size. Then regardless of mode, a linear layer adjusts the number of channels to the target channel number.
Network training
Each network is trained and evaluated separately with no weight sharing. During the search phase we minimise the loss on a train split and compute the validation metric on a validation set. To evaluate the final chosen module, we retrain on train+val splits and evaluate on test. To speed up our experiments, the inner loop optimisation of architectures uses fewer epochs compared to the evaluation phase. All networks are trained using the SGD optimizer with momentum of 0.9. The values for learning rate, weight decay, batch size and more can be found in Tab. 4.

Table 4: Hyperparameters for each dataset.

Dataset name	Metric type	Baseline model	Epochs (search)	Epochs (eval)	Batch size	Learning rate	Weight decay	Mom-entum
AddNIST	Accuracy	ResNet18	8	64	256	0.04	3×10^{-4}	0.9
Language	Accuracy	ResNet18	8	64	256	0.04	3×10^{-4}	0.9
MultNIST	Accuracy	ResNet18	8	64	256	0.04	3×10^{-4}	0.9
CIFARTile	Accuracy	ResNet18	8	64	256	0.04	3×10^{-4}	0.9
Gutenberg	Accuracy	ResNet18	8	64	256	0.04	3×10^{-4}	0.9
Isabella	Accuracy	ResNet18	8	64	256	0.04	3×10^{-4}	0.9
GeoClassing	Accuracy	ResNet18	8	64	256	0.04	3×10^{-4}	0.9
Chesseract	Accuracy	ResNet18	8	64	256	0.04	3×10^{-4}	0.9
CIFAR100	0-1 error	WRN16-4	25	200	128	0.1	5×10^{-4}	0.9
Spherical	0-1 error	WRN16-4	25	200	128	0.1	5×10^{-4}	0.9
NinaPro	0-1 error	WRN16-4	25	200	128	0.1	5×10^{-4}	0.9
Darcy Flow	relative ℓ_2	WRN16-4	25	200	4	0.001	5×10^{-4}	0.9
Cosmic	1 - AUROC	WRN16-4	25	200	8	0.001	5×10^{-4}	0.9

C.3 Datasets

We followed the official instructions of Unseen NAS [14] to setup the datasets. The information of each dataset is listed in the following:

AddNIST
This dataset is derived from the MNIST dataset. Specifically, each RGB image is computed by stacking three MNIST images in the channel dimension. It has a total of 20 categories; the class label is computed by summing the MNIST labels in all three channels. Among the 70,000 images, 45,000 are used for training, 15,000 are used for validation, and 10,000 images are used for testing.

Language
Language consists of six-letter words extracted from dictionaries of ten Latin alphabet languages: English, Dutch, German, Spanish, French, Portuguese, Swedish, Zulu, Swahili, and Finnish. Words

containing diacritics or the letters 'y' and 'z' are excluded. The dataset is split into 50,000 training samples, 10,000 validation samples, and 10,000 test samples.

MultNIST
MultNIST is a dataset designed similarly to AddNIST, originating from the same research objective. Each channel of the 3 channel images contains an image from the MNIST dataset. The dataset is divided into 50,000 training images, 10,000 validation images, and 10,000 test images. Unlike AddNIST, MultNIST comprises ten classes (0-9), the label for each MultNIST image is computed using the formula $l = (r \times g \times b) \text{mod} 10$, where r, g and b are the MNIST labels of the red, green, and blue channels, respectively.

CIFARTile
CIFARTile is a dataset where each image is a composite of four CIFAR-10 images arranged in a 2x2 grid. The dataset is divided into 45,000 training images, 15,000 validation images, and 10,000 test images. CIFARTile has four classes (0-3), determined by the number of distinct CIFAR-10 classes in each grid, minus one.

Gutenberg
Gutenberg is a dataset sourced from Project Gutenberg. It includes texts from six authors, with basic text preprocessing applied: punctuation removal, diacritic conversion, and elimination of structural words. The dataset contains consecutive sequences of three words (3-6 letters each), padded to 6 characters and concatenated into 18-character strings. These strings are converted into images with size $1 \times 27 \times 18$, with the x-axis representing character positions and the y-axis representing alphabetical letters or spaces. The task is to predict the author of each sequence. The dataset is split into 45,000 training, 15,000 validation, and 6,000 test images.

Isabella
Isabella is a dataset derived from musical recordings of the Isabella Stewart Gardner Museum, Boston. It includes four classes based on the era of composition: Baroque, Classical, Romantic, and 20th Century. The recordings are split into five-second snippets and converted into 64-band spectrograms, resulting in images with dimensions $1 \times 64 \times 128$. The dataset is divided into 50,000 training images, 10,000 validation images, and 10,000 test images, ensuring no overlap of recordings across splits. The task is to predict the era of composition from the spectrogram.

GeoClassing
GeoClassing is based on the BigEarthNet dataset. It uses satellite images initially labeled for ground-cover classification but reclassified by the European country they depict. The dataset includes ten classes: Austria, Belgium, Finland, Ireland, Kosovo, Lithuania, Luxembourg, Portugal, Serbia, and Switzerland. Each image is of size $3 \times 60 \times 60$. The dataset is split into 43,821 training images, 8,758 validation images, and 8,751 test images. The task is to predict the country depicted in each image based on topology and ground coverage.

Chesseract
Chesseract is a dataset derived from public chess games of eight grandmasters. The dataset consists of the final 15% of board states from these games. Each position is one-hot encoded by piece type and color, resulting in $12 \times 8 \times 8$ images. The dataset is divided into 49,998 training images, 9,999 validation images, and 9,999 test images, ensuring no positions from the same game appear across splits. Each image is classified into one of three classes: White wins, Draw, or Black wins. The task is to predict the game's result based on the given board position. We pad the input with 5 zero-valued pixels to make a $12 \times 18 \times 18$ tensor.

Table 5: Lower is better. Our results from performing random sampling, random search and regularised evolution in `einspace` on NASBench360 [46]. We compare to the results from [46], where the DARTS search space is used. RE(WRN) refers to initialising the regularised evolution search algorithm with WRN architecture. **Best** and <u>second</u> best performance per dataset.

	WRN	DARTS (GAEA)	Expert	RE(WRN) `einspace`
CIFAR100	25.61	24.02	**19.39**	<u>21.47</u>
Spherical	76.32	**48.23**	67.41	<u>66.37</u>
NinaPro	10.32	17.67	<u>8.73</u>	**6.37**
Darcy Flow	0.032	0.026	**0.008**	<u>0.014</u>
Cosmic	0.245	<u>0.229</u>	**0.127**	0.73
Average rank ↓	3.60	2.60	**1.60**	<u>2.20</u>

We follow the official instructions of NASBench360 [46] to setup the datasets. The information of each dataset is listed in the following (noting that we are using a subset of all the datasets):

CIFAR100
CIFAR100 is a widely known image classification dataset, comprised of 100 fine-grained classes. Each image is of size 3×32×32. The dataset is split into 40,000 training images, 10,000 validation images and 10,000 testing images. We preprocess the images by applying random crops, horizontal flips, and normalisation.

Spherical
Spherical dataset consists on classifying spherically projected CIFAR100 images. Specifically, CIFAR images are projected to the northern hemisphere with a random rotation. Each image is of size 3×60×60. Spherical dataset uses the same split ratios as CIFAR100. In this case, there is no data augmentation nor pre-processing steps.

NinaPro
NinaPro is a dataset for classifying hand gestures given their electromyography signals. EMG data signals are collected with two Myo armbands as wave signals. Wave signals, along with their wavelength and frequency, are processed 2D signals of shape 1×12×52. Classes are imbalanced, with the neutral position amounting for 65% of all gestures. The dataset is split into 2,638 training samples, 659 validation samples, and 659 testing samples. No further data augmentation is applied.

Darcy Flow
Darcy Flow is a regression task for predicting the solution of a 2D PDE at a predefined later stage given some 2D initial conditions. The input is a 1×85×85 image describing the initial state of the fluid. The output should match the same dimensions. The dataset is split into 900 images for training, 100 for validation, and 100 for test. Input data is normalised.

Cosmic
Cosmic dataset contains images from the F435W filter collected from the Hubble Space Telescope. It aims to identify cosmic rays (corrupted pixels) in the images. Inputs are images of 1×256×256, and outputs are pixel binary predictions (artifact vs. no-artifact). The dataset is split into 4,347 images for training, 483 for validation, and 420 for test.

C.4 Compute Resources

All our experiments ran on our two internal clusters with the following infrastructure:

- AMD EPYC 7552 48-Core Processor with 1000GB RAM and 8× NVIDIA RTX A5500 with 24GB of memory (dedicated use)
- AMD EPYC 7452 32-Core Processor with 400GB RAM and 7× NVIDIA A100 with 40GB of memory (shared use)

Each experiment used a single GPU to train each architecture. Running 1000 iterations of RE(Scratch) on the quickest datasets (Language and Chesseract) took around 2 days, while the slowest (GeoClassing) took 4 days. We had very similar training times for RE(RN18) and RE(Mix) which ran for 500 iterations.

D Additional Results

D.1 NAS-Bench-360

We test `einspace` on a selection of datasets from NAS-Bench-360 [46] to further showcase its potential. These results can be found in Tab. 5. In this setting the baseline network is a WideResNet-16-4 (WRN). We see that our regularised evolution with the baseline as the initial seed, RE(WRN), again consistently finds architectural improvements. It also outperforms the GAEA search strategy on the DARTS search space on all but the Cosmic task. On NinaPro, it even beats the Expert architecture, specifically designed for this task. Note that the WRN baseline network on its own does not achieve as low an error as reported in [46]. However, our `einspace` architecture beats even this number and to the best of our knowledge sets a new state-of-the-art.

D.2 Empirical Architecture Complexity

For our experiments we set the computation module probability to $p(\mathtt{M} \rightarrow \mathtt{C} \mid \mathtt{M}) = 0.32$ using the branching rate method described in B.5. We now report some empirical results for the architecture complexities as we vary this value. In Tab. 6 we can see that the complexity, as measured by the count of terminals and non-terminals in the derivation trees, grows as the probability decreases.

When searching for an architecture on a new unknown task, the flexibility of the search space is key. During our random searching on `einspace` we found that we sampled networks with parameter counts ranging from zero up to 1 billion, and as few as two operations to as many as 3k. The frequency of all functions in `einspace` that appear in these networks can be found in Table 7.

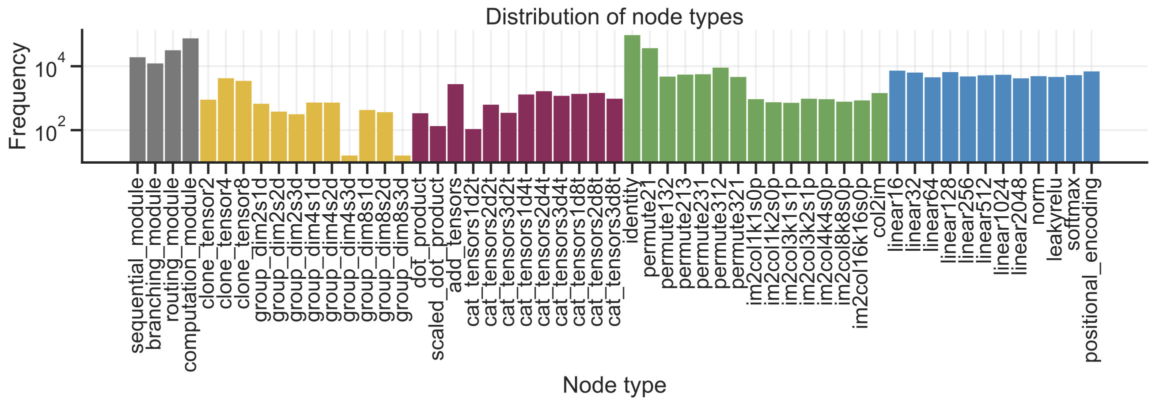

Figure 7: Frequency of each module/function in 8k sampled architectures with $p(\mathtt{M} \rightarrow \mathtt{C} \mid \mathtt{M}) = 0.32$.

Table 6: The distribution of terminal and nonterminal symbols as well as average branching factors in 2000 sampled architectures with varying values for the computation module probability $p(\texttt{M} \rightarrow \texttt{C|M})$. For probability values 0.2 and 0.1 there is no data, as the sampling process is too time-consuming.

Type	$p(\texttt{M} \rightarrow \texttt{C|M})$	Min	Mean	Median	Std	Max
terminals	0.9	2	3.87	3.00	2.84	28
	0.8	2	5.34	4.00	6.32	54
	0.7	2	6.72	4.00	11.22	118
	0.6	2	8.65	4.00	16.73	202
	0.5	2	40.22	5.00	303.65	4779
	0.4	2	100.15	8.00	749.61	12026
	0.3	2	6070.77	9.00	64863.45	878122
non-terminals	0.9	2	3.63	3.00	3.45	42
	0.8	2	5.04	3.00	6.91	64
	0.7	2	6.00	3.00	9.21	81
	0.6	2	7.74	4.00	13.61	165
	0.5	2	39.03	5.00	325.18	5219
	0.4	2	88.58	8.00	649.44	10417
	0.3	2	4588.13	8.00	50203.87	734497
branching factor	0.9	1	1.75	1.00	1.63	7.61
	0.8	1	2.05	1.00	1.95	7.73
	0.7	1	2.05	1.00	1.91	7.83
	0.6	1	2.17	1.00	1.96	7.73
	0.5	1	2.34	1.00	2.00	7.88
	0.4	1	2.81	1.75	2.14	7.87
	0.3	1	2.80	1.75	2.13	7.96

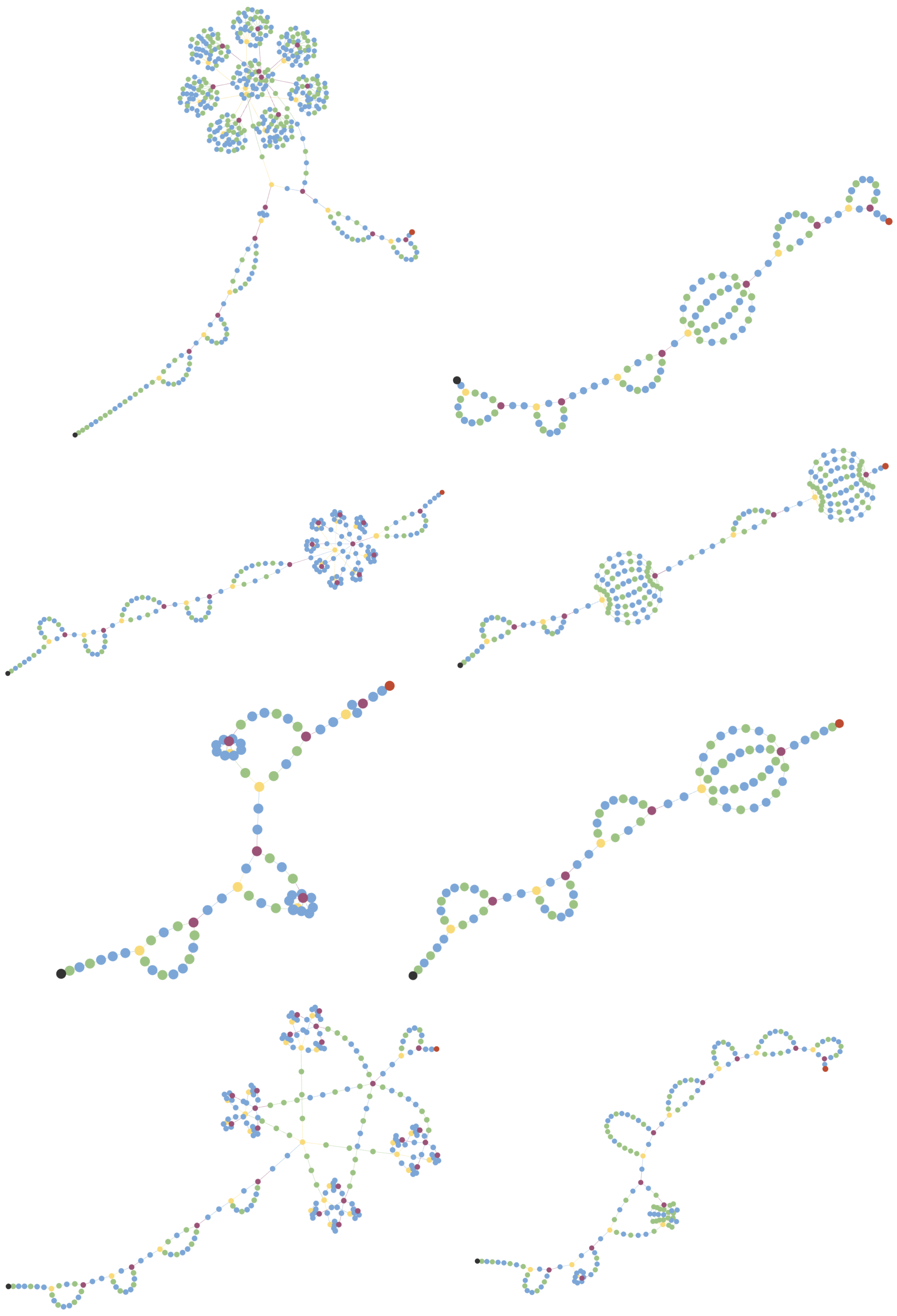

Figure 8: The top architectures found by RE(Mix) in `einspace` on Unseen NAS. From left to right, row by row: AddNIST, Language, MultNIST, CIFARTile, Gutenberg, Isabella, GeoClassing, Chesseract.