DNArch: Learning Convolutional Neural Architectures by Backpropagation

David W. Romero[*1] **Neil Zeghidour**[2]

Abstract

We present *Differentiable Neural Architectures* (DNArch), a method that learns the weights and the architecture of CNNs jointly by backpropagation. DNArch enables learning (*i*) the size of convolutional kernels, (*ii*) the width of all layers, (*iii*) the position and value of downsampling layers, and (*iv*) the depth of the network. DNArch treats neural architectures as continuous entities and uses learnable differentiable masks to control their size. Unlike existing methods, DNArch is not limited to a (small) predefined set of possible components, but instead it is able to discover CNN architectures across all feasible combinations of kernel sizes, widths, depths and downsampling. Empirically, DNArch finds effective architectures for classification and dense prediction tasks on sequential and image data. By adding a loss term that controls the network complexity, DNArch constrains its search to architectures that respect a predefined computational budget during training.

1. Introduction

Tailoring Convolutional Neural Networks (CNNs) (LeCun et al., 1998) to novel tasks and datasets requires substantial human intervention and cross-validation to find a good architecture, e.g. appropriate kernel sizes, width, depth, etc. This has motivated exploring the space of architectures in an automatic fashion, by developing architecture search algorithms. Although these methods can find good architectures, they must solve an expensive discrete optimization problem that involves training and evaluating candidate architectures in each iteration. Differentiable Architecture Search (DARTS) (Liu et al., 2018) addresses this issue by allowing the network to consider a set of *predefined* possible components in parallel, e.g., convolutions with kernels of size 3×3, 5×5, 7×7, and adjusting their contribution using learnable weights (Fig. 2). Although DARTS is able to *select* components via backpropagation, it requires (*i*) defining a (small) set of possible components beforehand, (*ii*) computing and keeping their responses in memory during training, and (*iii*) retraining the found architecture from scratch to remove the effect of other components in the output.

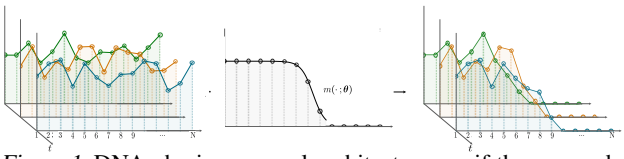

Figure 1. DNArch views neural architectures as if they were defined in a continuous multidimensional space, and uses learnable masks to learn their length by backpropagation. This example shows how DNArch learns the width of a layer by applying a differentiable mask $m(\cdot\,;\boldsymbol{\theta})$ to the channel dimension of the input. Different values of $\boldsymbol{\theta}$ lead to a different number of channels.

In this paper, we introduce *Differentiable Neural Architectures* (DNArch), a method that jointly learns the weights and the entire architecture of a CNN by backpropagation. Specifically, DNArch learns the weights as well as (*i*) the size of convolutional kernels at each layer, (*ii*) the number of channels at each layer, (*iii*) the position and resolution of downsampling layers, and (*iv*) the depth of the network. To this end, DNArch treats neural architectures as entities defined in a multidimensional continuous space with dimensions corresponding to network attributes, e.g., depth, width, etc., and uses learnable differentiable masks along each dimension to control their length (Fig. 1). Unlike DARTS methods (Liu et al., 2018; Shen et al., 2022), DNArch does *not* require a predefined set of components to choose from, but instead is able to explore among *all* feasible values, e.g., all kernel sizes between 1×1 and N×N for a N×N image. This is a result of the truly continuous nature of DNArch, which, unlike DARTS, does not require multiple instantiations of the same layer for different parameter values (Fig. 2). Instead, DNArch explores the parameter space by modifying the learnable parameters of the differentiable masks (Fig. 3), making it a much more scalable NAS method. Moreover, since both the architecture and the weights are optimized in a single run, no retraining is needed after training.

2. Differentiable masking

Consider a function $f : [\mathrm{a}, \mathrm{b}] \to \mathbb{R}$, which we want to be non-zero only in a subset $[\mathrm{c}, \mathrm{d}] \subseteq [\mathrm{a}, \mathrm{b}]$. To this end, we can multiply f with a mask m whose values are non-zero

[*]Work done during an internship at Google Research. [1]Vrije Universiteit Amsterdam. [2]Google Research, Paris, France. Correspondence to: David W. Romero <d.w.romeroguzman@vu.nl>.

Published at the Differentiable Almost Everything Workshop of the 40th International Conference on Machine Learning, Honolulu, Hawaii, USA. July 2023. Copyright 2023 by the author(s).

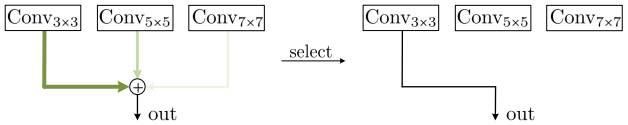

Figure 2. DARTS learns the size of convolutional kernels using backpropagation to select among predefined options.

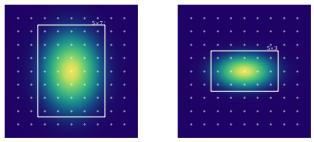

Figure 3. DNArch learns the size of convolutional kernels by modifying the parameters of the differentiable mask $m(\cdot\,;\boldsymbol{\theta})$. Different $\boldsymbol{\theta}$ values lead to different sizes.

only on $[\mathrm{c},\mathrm{d}]$, e.g., a rectangular mask $\Pi_{[\mathrm{c},\mathrm{d}]}(x)$=$\mathbf{1}_{[\mathrm{c},\mathrm{d}]}$. However, as the gradient of $\Pi_{[\mathrm{c},\mathrm{d}]}(x)$ is either zero or non-defined, it is not possible to learn the interval in which it is non-zero by backpropagation. To overcome this limitation, we can use a parametric differentiable mask $m(\cdot\,;\boldsymbol{\theta})$ whose interval of non-zero values is defined by its parameters $\boldsymbol{\theta}$. As the mask is differentiable w.r.t. its parameters $\boldsymbol{\theta}$, we can now use backprop to learn the interval on which it is non-zero.

Here, we consider two types of masks: a Gaussian mask m_{gauss} parameterized by its mean and variance $\boldsymbol{\theta}$=$\{\mu,\sigma^2\}$, and a Sigmoid mask m_{sigm} parameterized by its offset and its temperature $\boldsymbol{\theta}$=$\{\mu,\tau\}$ (Eqs. 3, 4). T_m is a predefined threshold below which the mask is zero (Fig. 4). Additional details can be found in Appx. A.

Materializing parameters only for non-zero mask values. Parts of the differentiable masks will map to zero based on the value of the parameters $\boldsymbol{\theta}$. Hence, it would be a waste of compute and memory to materialize the mask and the corresponding network parameters, e.g., channels $\mathrm{ch} \in [10,\mathrm{N}]$ in Fig. 1, to zero them out next. Fortunately, we can take advantage of the invertible form of the Gaussian and Sigmoid masks to materialize parameters *only for values for which the mask is non-zero* (Appx. A.3). Thus significantly reducing the compute and memory costs of DNArch.

3. Learning neural architectures by backprop

Most DNArch components, e.g., the learning of the network's width and depth, are not limited to convolutional architectures. However, here we aim to show *how DNArch can be used to learn as many components of a neural architecture as possible*. To that end, we take a general-purpose convolutional architecture: the CCNN (Knigge et al., 2023), and make all its architectural components learnable. CCNNs are an ideal base network for DNArch due to their ability to model global context on inputs of any resolution, length and dimensionality. This (*i*) prevents the formation of poor architectures due to insufficient receptive fields, and (*ii*) allows DNArch to be used on tasks on data of arbitrary length and dimensionality without changing the base network (needed in existing methods. Tu et al. (2022) shows many examples).

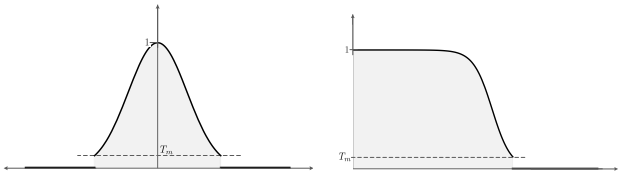

Figure 4. Gaussian and sigmoid masks.

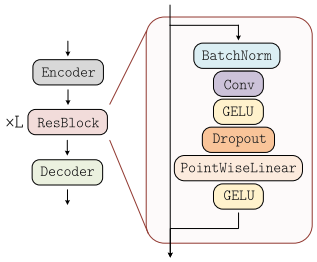

Figure 5. The Continuous CNN architecture (Romero et al., 2022).

3.1. Learning the size of convolutional kernels

Introduced in Romero et al. (2021a), differentiable masking can be combined with CKConvs (Romero et al., 2021b) to learn the size of convolutional kernels by backpropagation by parameterizing convolutional kernels ψ as the product of a small neural network $\texttt{MLP}_\psi$ and a learnable differentiable mask $m(\cdot\,;\boldsymbol{\theta})$, i.e., $\psi(\mathbf{c}_i)$=$\texttt{MLP}_\psi(\mathbf{c}_i)\cdot m(\mathbf{c}_i;\boldsymbol{\theta})$ (Fig. 6). Note that, we can construct the kernel only for non-zero values of the mask by following the method given in Appx. A.3.

3.2. Learning downsampling layers

We can use differentiable masking to learn downsampling by applying a differentiable mask on the Fourier domain. The Fourier transform $\mathcal{F}$ represents a function $f:\mathbb{R}^{\mathrm{D}}\rightarrow\mathbb{R}$ in terms of its *spectrum* $\tilde{f}:\mathbb{R}^{\mathrm{D}}\rightarrow\mathbb{C}$, which map frequencies ω to the amount of that frequency in the input $\tilde{f}(\omega)$. A useful identity is that cropping high frequencies in the Fourier domain equals downsampling in the spatial domain.

To learn downsampling, we use a learnable sigmoid mask m_{sigm} to perform a *learnable low-pass filtering* on the input by multiplying the spectrum of the input with the mask m_{sigm}. By doing so, all frequencies above the mask's cutoff frequency $\omega_{\max}$=T_m becomes zero (Fig. 7). An important consequence of low-pass filtering is that as the spectrum of the signal becomes zero above $\omega_{\max}$, the low-passed signal can be represented at a lower resolution determined by $\omega_{\max}$. With $\mathcal{F}$, $\mathcal{F}^{-1}$ be the Fourier and inverse Fourier transform, $\mathrm{crop}_{>\omega_{\max}}$ be an operator that crops values above $\omega_{\max}$, and f_{down} represent the downsampled signal f, we have that:

$$f_{\text{down}}=\mathcal{F}^{-1}\left[\mathrm{crop}_{>\omega_{\max}}\left(\mathcal{F}[f]\cdot m_{\text{sigm}}(\,\cdot;\boldsymbol{\theta})\right)\right]. \quad (1)$$

Unlike regular downsampling, e.g., max-pooling, *spectral downsampling* (Rippel et al., 2015) considers the spectral content of the input during downsampling, and thus prevents *aliasing* (Fig. 7, middle down), which has negative effects for learning (Zhang, 2019; Vasconcelos et al., 2021). More information w.r.t. the positioning of learnable downsampling and its use for dense predictions is given in Appx. B.

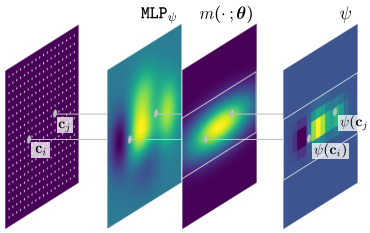

Figure 6. Learning kernel sizes with differentiable masking.

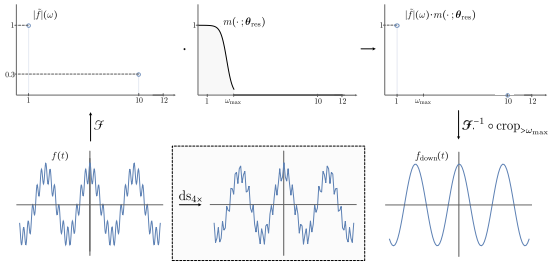

Figure 7. Learning downsampling with differentiable masking.

3.3. Learning the width of the network

To learn the width of a layer we apply a differentiable mask $m(\cdot\,;\boldsymbol{\theta})$ along the channel dimension of its feature representations (Fig. 1). To learn the width of all layers in the network, we apply differentiable masks with independent learnable parameters along the channel dimensions of all the network components that change the network's width, i.e., all `Conv` and `PWLinear` layers. That is, we learn an independent mask for the input ($\mathrm{N_{in}}$), the middle ($\mathrm{N_{mid}}$) and the output ($\mathrm{N_{out}}$) channels of each residual block in the network (Fig. 8). Layers that do not change the network's width, e.g., `GELU`, have their width determined by the previous mask.

3.4. Learning the depth of the network

To learn the network's depth we view the number of residual blocks as a continuous axis with values $[1, 2, .., \mathrm{D}]$ corresponding to the index of each block, and use a differentiable mask $m(\cdot, \boldsymbol{\theta})$ along this axis to dynamically mask out blocks based on the value of the mask parameters $\boldsymbol{\theta}$ (Fig. 8b). To ensure that information flows from the input to the output of the network regardless of the value of the mask parameters, we *only* apply the mask on the residual branch

3.5. Putting it all together

By using the techniques described in Sections 3.1-3.4, DNArch uses backpropagation to learn the weights, the size of convolutional kernels at each layer, the number of channels at each layer, the position and resolution of downsampling layers, and the depth of a convolutional network.

Learning architectures under computational constraints. As outlined in Appx. C, we can ensure that the architectures searched by DNArch respect a predefined computational complexity by including a regularization term $\mathcal{L}_{\text{comp}}$ that reflects the complexity of the current candidate architecture based on its mask parameters. To this end, we define the optimization loss $\mathcal{L}$ as the sum of the task objective loss

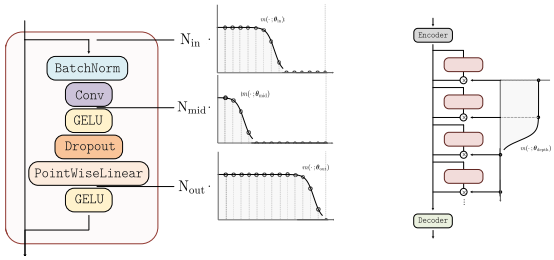

Figure 8. (a) Positioning of masks for the learning of the width. (b) Learning the network's depth with differential masking.

$\mathcal{L}_{\text{obj}}$ and the complexity loss $\mathcal{L}_{\text{comp}}$ weighted by a factor λ:

$$\mathcal{L} = \mathcal{L}_{\text{obj}} + \lambda\, \mathcal{L}_{\text{comp}}. \tag{2}$$

By minimizing this loss, DNArch is encouraged to find architectures that meet the desired computational budget while still achieving good performance on the end task.

4. Experiments

A detailed description of the experimental setup and the nomenclature used can be found in Appx. E.1. We evaluate DNArch on sequential and image datasets for classification and dense prediction tasks. On 1D, we use the Long Range Arena (LRA) benchmark (Tay et al., 2020). On 2D, we perform image classification on the CIFAR10/100 datasets (Krizhevsky et al., 2009) and report results on two dense prediction tasks from the NAS-Bench-360 benchmark (Tu et al., 2022): `DarcyFlow` and `Cosmic`. A detailed description of the datasets used can be found in Appx. D.

DNArch without computational constraints. First, we use DNArch to improve the expressiveness and computational efficiency of a $\text{CCNN}_{4,140}$. We start using DNArch to learn the size of all convolutional kernels, and then we learn both the kernel sizes and downsampling layers to simultaneously improve the expressiveness and the computational efficiency of the $\text{CCNN}_{4,140}$. Note that the learning is solely driven by the objective loss $\mathcal{L}_{\text{obj}}$, i.e., $\mathcal{L}_{\text{comp}}$ is not used.

Results. We observe that using DNArch to learn kernel sizes consistently improves the accuracy of the base architecture (DNArch_{K} models in Tabs. 1-3). Interestingly, found DNArch architectures perform on par, and even surpass, architectures specifically designed for each tasks, e.g., S4 (Gu et al., 2021) for sequences and NFOs (Li et al., 2020) for 2D PDEs, with a remarkably lower number of trainable parameters. In contrast to DARTS methods, e.g., DASH (Shen et al., 2022), DNArch can be applied across *all tasks* without the need to *manually change* the base architecture. When additionally learning downsampling, we observe that DNArch finds high-performant architectures with improved computational efficiency ($\text{DNArch}_{\text{K,R}}$ models). Interestingly, found models often exhibit slight accuracy improvements.

DNArch with computational constraints. Next, we utilize DNArch to learn entire neural architectures that respect a predefined computational budget. We start with base

Table 1. Performance on the LRA benchmark. × denotes random guessing. Highest per-section scores are in bold and the overall best scores are underlined. For DNArch, values in parenthesis indicate the computational cost of the architecture relative to the target complexity.

MODEL	LISTOPS	TEXT	RETRIEVAL	IMAGE	PATHFINDER	PATH-X	AVG.
Transformer (Vaswani et al., 2017)	**36.37**	64.27	57.46	42.44	71.40	×	53.66
Reformer (Kitaev et al., 2020)	37.27	56.10	53.40	38.07	68.50	×	50.56
Performer (Choromanski et al., 2020)	18.01	**65.40**	53.82	**42.77**	**77.05**	×	51.18
BigBird (Zaheer et al., 2020)	36.05	64.02	**59.29**	40.83	74.87	×	**54.17**
Mega ($\mathcal{O}(L^2)$) (Ma et al., 2022)	**<u>63.14</u>**	**<u>90.43</u>**	**91.25**	**<u>90.44</u>**	**<u>96.01</u>**	**97.98**	**<u>88.21</u>**
Mega-chunk ($\mathcal{O}(L)$) (Ma et al., 2022)	58.76	90.19	90.97	85.80	94.41	93.81	85.66
S4D (Gu et al., 2022)	60.47	86.18	89.46	88.19	93.06	91.95	84.89
S4 (Gu et al., 2021)	59.60	86.82	90.90	**88.65**	94.20	96.35	86.09
S5 (Smith et al., 2022)	**61.50**	**89.31**	**<u>91.40</u>**	88.00	**95.33**	**<u>98.58</u>**	**87.35**
FNet (Lee-Thorp et al., 2021)	35.33	65.11	59.61	38.67	77.80	×	54.42
Luna-256 (Ma et al., 2021)	37.25	64.57	79.29	47.38	77.72	×	59.37
$\text{CCNN}_{4,140}$ (Romero et al., 2022)	**44.85**	**83.59**	×	**87.62**	**91.36**	×	**76.86**
$\text{CCNN}_{4,140}$ (Global Kernels)	55.65	87.80	90.55	85.51	94.26	91.15	84.15
$\text{DNArch}_{\text{K}}(\text{CCNN}_{4,140})$	59.90	88.28	90.66	86.07	93.46	89.93	84.72
$\text{DNArch}_{\text{K,R}}(\text{CCNN}_{4,140})$	$60.15_{(0.80\times)}$	$88.50_{(0.75\times)}$	$91.08_{(0.78\times)}$	$86.55_{(0.82\times)}$	$94.05_{(0.89\times)}$	$91.15_{(0.82\times)}$	85.25
$\text{DNArch}_{\text{K,R,W,D}}(\text{CCNN}_{4,140})$	$\mathbf{60.55}_{(1.01\times)}$	$\mathbf{89.03}_{(1.00\times)}$	$\mathbf{91.22}_{(1.02\times)}$	$\mathbf{87.20}_{(1.02\times)}$	$\mathbf{94.95}_{(1.00\times)}$	$\mathbf{91.71}_{(1.01\times)}$	**85.78**

Table 2. Performance on Dense Tasks of NAS-Bench-360.

MODEL	DARCYFLOW rel. ℓ^2 loss	COSMIC 1 - AUROC
Expert*	**0.008**	**0.13**
WRN (Zagoruyko & Komodakis, 2016)	0.073	0.24
DenseNAS (Fang et al., 2020)	0.100	0.38
DARTS (Liu et al., 2018)	**0.026**	0.229
Auto-DL (Liu et al., 2019)	0.049	0.495
DASH (Shen et al., 2022)	0.060	**0.190**
$\text{CCNN}_{4,140}$ (Global Kernels)	0.002989	0.059
$\text{DNArch}_{\text{K}}(\text{CCNN}_{4,140})$	0.002970	0.058
$\text{DNArch}_{\text{K,R}}(\text{CCNN}_{4,140})$	$0.002929_{(0.79\times)}$	$0.056_{(0.82\times)}$
$\text{DNArch}_{\text{K,R,W,D}}(\text{CCNN}_{4,140})$	$\mathbf{0.002285}_{(1.01\times)}$	$\mathbf{0.055}_{(1.01\times)}$
$\text{CCNN}_{6,380}$ (Global Kernels)	0.004521	0.059
$\text{DNArch}_{\text{K,R,W,D}}(\text{CCNN}_{6,380})$	$\mathbf{\underline{0.001763}}_{(1.00\times)}$	$\mathbf{\underline{0.048}}_{(1.00\times)}$

* FNO (Li et al., 2020) and deepCR (Zhang & Bloom, 2020).

Table 3. Performance on Image Classification Datasets.

MODEL	CIFAR10	CIFAR100
WRN (Zagoruyko & Komodakis, 2016)	-	**<u>76.65</u>**
DenseNAS (Fang et al., 2020)	-	74.51
DARTS (Liu et al., 2018)	-	75.98
DASH (Shen et al., 2022)	-	75.63
$\text{CCNN}_{4,140}$ (Global Kernels)	90.52	64.72
$\text{DNArch}_{\text{K}}(\text{CCNN}_{4,140})$	92.51	69.01
$\text{DNArch}_{\text{K,R}}(\text{CCNN}_{4,140})$	$92.77_{(0.82\times)}$	$68.96_{(0.85\times)}$
$\text{DNArch}_{\text{K,R,W,D}}(\text{CCNN}_{4,140})$	$\mathbf{93.47}_{(1.01\times)}$	$\mathbf{72.98}_{(1.05\times)}$
$\text{CCNN}_{6,380}$ (Global Kernels)	94.18	72.29
$\text{DNArch}_{\text{K,R,W,D}}(\text{CCNN}_{6,380})$	$\mathbf{\underline{95.03}}_{(1.00\times)}$	$\mathbf{76.37}_{(1.02\times)}$

$\text{CCNN}_{4,140}$ and $\text{CCNN}_{6,380}$ networks, and allow DNArch to learn their width, depth, kernel sizes and downsampling. We define the target complexity $\mathcal{L}_{\text{comp}}$ as the complexity of the base CCNN networks. In other words, we use DNArch to find better convolutional architectures of computational complexity roughly equal to that of the base networks.

Results. Our results ($\text{DNArch}_{\text{K,R,W,D}}$ models) show that DNArch finds neural architectures with higher accuracy than the base CCNN networks but with the same complexity. In addition, we observe that learning more architectural components consistently leads to better results, thus supporting the claim that gradient-steered architectures can be more beneficial than handcrafted ones. Furthermore, using base architectures with larger complexity and capacity ($\text{CCNN}_{6,380}$ vs. $\text{CCNN}_{4,140}$) consistently leads to better results. This result is encouraging for the use of DNArch to large architectures, e.g., LLMs (Brown et al., 2020).

Computational complexity of DNArch. To assess the applicability of DNArch, we also analyze its computational overhead. To this end, we plot the evolution of the relative complexity ($\mathcal{C}_{\text{curr}}/\mathcal{C}_{\text{target}}$) during training (Fig. 9). Interestingly, we observe that *the theoretical complexity of candidate architectures $\mathcal{C}_{\text{curr}}$ stays close to the target complexity $\mathcal{C}_{\text{target}}$ during the whole training*. This indicates that: (*i*) DNArch only searches among architectures close to the target computational complexity, and that (*ii*) the computational overhead of DNArch is *negligible*. As a result, the cost of using DNArch on top of a CCNN is comparable to the cost of training the base CCNN network.

Architectures found by DNArch. We observe that found architectures are very diverse, even within each architecture (Tabs. 6-8). For instance, some residual blocks have a bottleneck structure, some an expanded structure, and others have monotonically decreasing or increasing widths. Interestingly, the resolution of found architectures for classification, e.g., `Text`, often follow the style of U-Nets, and not the monotonically decreasing pattern commonly used in handcrafted networks. On dense tasks, found architectures resemble U-Nets, and even concatenated U-Nets, e.g., the 1.5× U-Net architectures found for the `Cosmic` task.

5. Conclusion

We presented *Differentiable Neural Architectures* (DNArch) a method that jointly learns the weights and the architecture of CNNs by backpropagation. DNArch finds effective architectures across a broad set of tasks, and can include additional loss terms to encourage the discovery of neural architectures with desirable properties. Appx. F, G discuss the limitations and promising future directions for DNArch.

References

Bengio, Y., Léonard, N., and Courville, A. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432*, 2013.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33: 1877–1901, 2020.

Choromanski, K., Likhosherstov, V., Dohan, D., Song, X., Gane, A., Sarlos, T., Hawkins, P., Davis, J., Mohiuddin, A., Kaiser, L., et al. Rethinking attention with performers. *arXiv preprint arXiv:2009.14794*, 2020.

Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., Barham, P., Chung, H. W., Sutton, C., Gehrmann, S., et al. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*, 2022.

Fang, J., Sun, Y., Zhang, Q., Li, Y., Liu, W., and Wang, X. Densely connected search space for more flexible neural architecture search. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 10628–10637, 2020.

Ghodrati, A., Bejnordi, B. E., and Habibian, A. Frameexit: Conditional early exiting for efficient video recognition. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 15608–15618, 2021.

Gu, A., Goel, K., and Ré, C. Efficiently modeling long sequences with structured state spaces. *arXiv preprint arXiv:2111.00396*, 2021.

Gu, A., Gupta, A., Goel, K., and Ré, C. On the parameterization and initialization of diagonal state space models. *arXiv preprint arXiv:2206.11893*, 2022.

Hendrycks, D. and Gimpel, K. Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415*, 2016.

Ioffe, S. and Szegedy, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pp. 448–456. pmlr, 2015.

Kitaev, N., Kaiser, Ł., and Levskaya, A. Reformer: The efficient transformer. *arXiv preprint arXiv:2001.04451*, 2020.

Knigge, D. M., Romero, D. W., Gu, A., Gavves, E., Bekkers, E. J., Tomczak, J. M., Hoogendoorn, M., and jakob Sonke, J. Modelling long range dependencies in nd: From task-specific to a general purpose CNN. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=ZW5aK4yCRqU.

Krizhevsky, A., Hinton, G., et al. Learning multiple layers of features from tiny images. 2009.

LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

Lee-Thorp, J., Ainslie, J., Eckstein, I., and Ontanon, S. Fnet: Mixing tokens with fourier transforms. *arXiv preprint arXiv:2105.03824*, 2021.

Li, Z., Kovachki, N., Azizzadenesheli, K., Liu, B., Bhattacharya, K., Stuart, A., and Anandkumar, A. Fourier neural operator for parametric partial differential equations. *arXiv preprint arXiv:2010.08895*, 2020.

Linsley, D., Kim, J., Veerabadran, V., Windolf, C., and Serre, T. Learning long-range spatial dependencies with horizontal gated recurrent units. *Advances in neural information processing systems*, 31, 2018.

Liu, C., Chen, L.-C., Schroff, F., Adam, H., Hua, W., Yuille, A. L., and Fei-Fei, L. Auto-deeplab: Hierarchical neural architecture search for semantic image segmentation. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 82–92, 2019.

Liu, H., Simonyan, K., and Yang, Y. Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055*, 2018.

Loshchilov, I. and Hutter, F. Sgdr: Stochastic gradient descent with warm restarts. *arXiv preprint arXiv:1608.03983*, 2016.

Loshchilov, I. and Hutter, F. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

Ma, X., Kong, X., Wang, S., Zhou, C., May, J., Ma, H., and Zettlemoyer, L. Luna: Linear unified nested attention. *Advances in Neural Information Processing Systems*, 34: 2441–2453, 2021.

Ma, X., Zhou, C., Kong, X., He, J., Gui, L., Neubig, G., May, J., and Zettlemoyer, L. Mega: moving average equipped gated attention. *arXiv preprint arXiv:2209.10655*, 2022.

Maas, A., Daly, R. E., Pham, P. T., Huang, D., Ng, A. Y., and Potts, C. Learning word vectors for sentiment analysis. In *Proceedings of the 49th annual meeting of the association for computational linguistics: Human language technologies*, pp. 142–150, 2011.

Nangia, N. and Bowman, S. R. Listops: A diagnostic dataset for latent tree learning. *arXiv preprint arXiv:1804.06028*, 2018.

Radev, D. R., Muthukrishnan, P., Qazvinian, V., and Abu-Jbara, A. The acl anthology network corpus. *Language Resources and Evaluation*, 47(4):919–944, 2013.

Riad, R., Teboul, O., Grangier, D., and Zeghidour, N. Learning strides in convolutional neural networks. *arXiv preprint arXiv:2202.01653*, 2022.

Rippel, O., Snoek, J., and Adams, R. P. Spectral representations for convolutional neural networks. *Advances in neural information processing systems*, 28, 2015.

Romero, D. W., Bruintjes, R.-J., Tomczak, J. M., Bekkers, E. J., Hoogendoorn, M., and van Gemert, J. C. Flexconv: Continuous kernel convolutions with differentiable kernel sizes. *arXiv preprint arXiv:2110.08059*, 2021a.

Romero, D. W., Kuzina, A., Bekkers, E. J., Tomczak, J. M., and Hoogendoorn, M. Ckconv: Continuous kernel convolution for sequential data. *arXiv preprint arXiv:2102.02611*, 2021b.

Romero, D. W., Knigge, D. M., Gu, A., Bekkers, E. J., Gavves, E., Tomczak, J. M., and Hoogendoorn, M. Towards a general purpose cnn for long range dependencies in nd. *arXiv preprint arXiv:2206.03398*, 2022.

Ronneberger, O., Fischer, P., and Brox, T. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, pp. 234–241. Springer, 2015.

Schuster, T., Fisch, A., Gupta, J., Dehghani, M., Bahri, D., Tran, V. Q., Tay, Y., and Metzler, D. Confident adaptive language modeling. *arXiv preprint arXiv:2207.07061*, 2022.

Shen, J., Khodak, M., and Talwalkar, A. Efficient architecture search for diverse tasks. *arXiv preprint arXiv:2204.07554*, 2022.

Sitzmann, V., Martel, J., Bergman, A., Lindell, D., and Wetzstein, G. Implicit neural representations with periodic activation functions. *Advances in Neural Information Processing Systems*, 33:7462–7473, 2020.

Smith, J. T., Warrington, A., and Linderman, S. W. Simplified state space layers for sequence modeling. *arXiv preprint arXiv:2208.04933*, 2022.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

Tancik, M., Srinivasan, P., Mildenhall, B., Fridovich-Keil, S., Raghavan, N., Singhal, U., Ramamoorthi, R., Barron, J., and Ng, R. Fourier features let networks learn high frequency functions in low dimensional domains. *Advances in Neural Information Processing Systems*, 33: 7537–7547, 2020.

Tay, Y., Dehghani, M., Abnar, S., Shen, Y., Bahri, D., Pham, P., Rao, J., Yang, L., Ruder, S., and Metzler, D. Long range arena: A benchmark for efficient transformers. *arXiv preprint arXiv:2011.04006*, 2020.

Teerapittayanon, S., McDanel, B., and Kung, H.-T. Branchynet: Fast inference via early exiting from deep neural networks. In *2016 23rd International Conference on Pattern Recognition (ICPR)*, pp. 2464–2469. IEEE, 2016.

Tu, R., Roberts, N., Khodak, M., Shen, J., Sala, F., and Talwalkar, A. Nas-bench-360: Benchmarking neural architecture search on diverse tasks. In *Thirty-sixth Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2022.

Vasconcelos, C., Larochelle, H., Dumoulin, V., Romijnders, R., Le Roux, N., and Goroshin, R. Impact of aliasing on generalization in deep convolutional networks. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 10529–10538, 2021.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Zagoruyko, S. and Komodakis, N. Wide residual networks. *arXiv preprint arXiv:1605.07146*, 2016.

Zaheer, M., Guruganesh, G., Dubey, K. A., Ainslie, J., Alberti, C., Ontanon, S., Pham, P., Ravula, A., Wang, Q., Yang, L., et al. Big bird: Transformers for longer sequences. *Advances in Neural Information Processing Systems*, 33:17283–17297, 2020.

Zhang, K. and Bloom, J. S. deepcr: cosmic ray rejection with deep learning. *The Astrophysical Journal*, 889(1): 24, 2020.

Zhang, R. Making convolutional networks shift-invariant again. In *International conference on machine learning*, pp. 7324–7334. PMLR, 2019.

Appendix

A. Differentiable Masking

A.1. Definition

Gaussian mask:

$$m_{\text{gauss}}(x\,;\mu,\sigma^2) = \begin{cases} \bar{m} = \exp\left(-\frac{1}{2}\frac{(x-\mu)^2}{\sigma^2}\right) & \bar{m} \geq T_m \\ 0 & \text{else} \end{cases} \quad (3)$$

Sigmoid mask:

$$m_{\text{sigm}}(x\,;\mu,\tau) = \begin{cases} \bar{m} = 1-\frac{1}{1+\exp(-\tau(x-\mu))} & \bar{m} \geq T_m \\ 0 & \text{else} \end{cases} \quad (4)$$

A.2. Multidimensional masks

N-dimensional masks can be constructed by combining N 1D masks, each with their own parameters. For example, the Gaussian mask used to learn the size of convolutional kernels in Fig. 3 is constructed as:

$$m_{\text{gauss}}(x,y;\{\{\mu_X,\mu_Y\},\{\sigma_X^2,\sigma_Y^2\}\}) = m_{\text{gauss}}\left(x\,;\{\mu_X,\sigma_X^2\}\right)\cdot m_{\text{gauss}}\left(y\,;\{\mu_Y,\sigma_Y^2\}\right)$$

A.3. Materializing parameters only for non-zero mask values

Parts of differentiable masks will map to zero based on the value of the parameters $\boldsymbol{\theta}$. Therefore, it would be a waste of compute and memory to materialize the mask – and the corresponding network parameters, e.g., channels $\text{ch} \in [10, \text{N}]$ in Fig. 1– to zero them out next. Luckily, we can take advantage of the invertible form of the Gaussian and Sigmoid masks to materialize parameters only for values for which the mask is non-zero. To this end, we find the value x_{T_m} for which the mask is equal to the threshold T_m, i.e., x_{T_m}=x such that $m(x;\boldsymbol{\theta})$=T_m, and only materialize the mask and the corresponding network parameters for values of x for which the value of the mask is greater than T_m. By inverting the mask equations (Eqs. 3, 4), we obtain x_{T_m} as:

$$\pm x_{T_m} = \mu \pm \sqrt{-2\sigma^2\log(T_m)}, \quad \text{and} \quad (5)$$

$$x_{T_m} = \mu - \frac{1}{\tau}\log\left(\frac{1}{1-T_m}-1\right), \quad (6)$$

for Gaussian and Sigmoid masks, respectively. Consequently, we can make sure that all rendered values will be used by only materializing the mask and related network parameters for values of x within the range $[-x_{T_m}, x_{T_m}]$ for Gaussian masks and $[x_{\min}, x_{T_m}]$ for Sigmoid masks, where $x_{\min}$ depicts the lowest coordinate indexing the mask.

B. Learning Downsampling Layers

B.1. Combining learnable downsampling and convolution

The previous method requires mapping inputs to the Fourier domain and back to learn downsampling. Fortunately, CCNNs as well as most methods that rely on global convolutions, e.g., CKConv (Romero et al., 2021b), S4 (Gu et al., 2021), rely on the *Fourier convolution theorem*: $(f*\psi)$=$\mathcal{F}^{-1}\left[\mathcal{F}[f]\cdot\mathcal{F}[\psi]\right]$ to compute convolutions with large kernels efficiently. This means that CCNNs already use a Fourier and inverse Fourier transforms in each residual block to compute convolutions. Hence, we can avoid recomputing these steps by placing the learnable downsampling operation *within the Fourier convolution*. Specifically, we can simultaneously compute downsampling and convolution by applying the differentiable mask m_{sigm} and the cropping operations $\text{crop}_{>\omega_{\max}}$ before returning from the Fourier domain back to the spatial domain. That is:[1]

$$(f*\psi)_{\text{down}} = \mathcal{F}^{-1}\left[\text{crop}_{>\omega_{\max}}(m_{\text{sigm}}(\cdot\,;\boldsymbol{\theta})\cdot\mathcal{F}[f]\cdot\mathcal{F}[\psi])\right]. \quad (7)$$

B.2. Materializing functions only on the output resolution

Note that Eq. 7 computes the convolution on the resolution of the input and downsamples next. This incurs in an unnecessary overhead as the output of the convolution will be downsampled directly after. A more efficient approach comes from inverting the order of these operations to compute the convolution at the downsampled resolution. Luckily, this can be achieved by using the method outlined in Sec. A.3. Since the cutoff frequency of the mask corresponds to the coordinate at which the mask equals the threshold, i.e., $\omega_{\max}$=x_{T_m}, it can be calculated using Eq. 6. Next, since the cutoff frequency defines the minimum resolution required to faithfully represent the input, we can simply downsample the input and convolutional kernel to that resolution before the convolution to compute it on the output resolution.

B.3. Learning subsampling for dense tasks

Riad et al. (2022) apply learned downsampling on both the identity and the residual branches of a residual block to limit resolution of all representations after a specific residual block. In the context of DNArch, this is undesirable for two reasons: First, as we learn the whole network architecture during training, it is not known a priori what resolution

[1] We note that the Fourier transform is not strictly necessary to learn downsampling, e.g., for CNNs with local kernels. Leveraging the Fourier convolution theorem, equivalent downsampling can be achieved by convolving the input with the inverse Fourier transform of the mask in the spatial domain (see Appx. B.4 for details).

mappings will require at each layer. However, forcing the identity branch to have the same resolution as the corresponding residual branch restricts all subsequent mappings to be of maximum that resolution. Secondly, dense prediction tasks, e.g., segmentation, require the learned architecture to produce outputs that share the same resolution as the input. However, if the identity branch is also downsampled, the output of the network would be of lower resolution even for a single level of downsampling in the network. This in turn, would result in over-smoothed predictions.

Based on these observations, we use downsampling *only* on the residual branch and upsample features at the end of each residual block back to the resolution of the input. This allows us to (*i*) have features at same resolution as the input in the last layer, and (*ii*) learn U-Net (Ronneberger et al., 2015) like architectures.

B.4. Learning downsampling in the spatial domain

Differentiable Masking learns downsampling by multiplying the spectrum $\tilde{f}$=$\mathcal{F}[f]$ of a signal f with a differentiable mask $m(\cdot\,;\boldsymbol{\theta})$, and cropping the output above the cutoff frequency of the mask $\omega_{\max}$ next. However, it is not necessary to perform this operation in the Fourier domain. Downsampling can also be learned directly in the spatial domain.

The *Fourier convolution theorem* states that the spatial convolution is equivalent to a pointwise multiplication in the Fourier domain. However, this equivalence works in both directions. That is, we can equivalently say that the pointwise multiplication on the Fourier domain is equal to a convolution on the spatial domain. Consequently, we can represent the pointwise multiplication of the spectrum of the input $\mathcal{F}[f]$ and the differentiable mask $m(\cdot\,;\boldsymbol{\theta})$ as the convolution of their inverse Fourier transforms. Formally:

$$\begin{aligned}\tilde{f}\cdot m(\cdot\,;\boldsymbol{\theta}) &= \mathcal{F}\left[\mathcal{F}^{-1}\left[\tilde{f}\right] * \mathcal{F}^{-1}[m(\cdot\,;\boldsymbol{\theta})]\right]\\ &= \mathcal{F}\left[\mathcal{F}^{-1}\left[\mathcal{F}[f]\right] * \mathcal{F}^{-1}[m(\cdot\,;\boldsymbol{\theta})]\right]\\ &= \mathcal{F}\left[f * \mathcal{F}^{-1}[m(\cdot\,;\boldsymbol{\theta})]\right] \qquad (8)\end{aligned}$$

In other words, we can perform the same operation in the spatial domain by convolution the original input signal f with the inverse Fourier transform of the mask $m(\cdot\,;\boldsymbol{\theta})$.

Defining the output resolution. Eq. 8 defines how low-pass filtering can be performed on the spatial domain, but it does not provide information regarding the final resolution of the operation. To derive the resolution of the output, we can simply use Eqs. 10, 11 to analytically derive the size of the mask. Once the size of the mask is derived, we can simply take the downsampled signal –after using Eq. 8–, and downsample it to match the size of the mask.

C. Learning convolutional architectures under computational constraints

We can ensure that the architectures searched by DNArch respect a predefined computational complexity by including an additional regularization term $\mathcal{L}_{\text{comp}}$ that reflects the complexity of the current candidate architecture based on its mask parameters. To this end, we define the optimization loss $\mathcal{L}$ as the sum of the task objective loss $\mathcal{L}_{\text{obj}}$ and the complexity loss $\mathcal{L}_{\text{comp}}$ weighted by a factor λ:

$$\mathcal{L} = \mathcal{L}_{\text{obj}} + \lambda\,\mathcal{L}_{\text{comp}}. \qquad (9)$$

By minimizing this loss, DNArch is encouraged to find architectures that meet the desired computational budget while still achieving good performance on the end task.

C.1. Defining the complexity loss $\mathcal{L}_{\text{comp}}$

The purpose of $\mathcal{L}_{\text{comp}}$ is to use the size of the learned masks to estimate the total computation needed for a forward pass of the network. Its construction is outlined below.

Layer-wise complexities. Let $\mathcal{C}_{\text{layer}}(\text{L}, \text{N}_{\text{in}}, \text{N}_{\text{out}})$ be the number of operations required in a given layer with an input of length L and N_{in} and N_{out} input and output channels. To estimate the number of computations required based on the current size of the masks, we can substitute the lenght of each dimension with the size of the corresponding masks: $\mathcal{C}_{\text{layer}}\left(\text{size}(m_{\text{res}}), \text{size}(m_{\text{N}_{\text{in}}}), \text{size}(m_{\text{N}_{\text{out}}})\right)$. As an example, consider a pointwise linear layer. A pointwise linear layer $\text{lin} : \mathbb{R}^{\text{N}_{\text{in}}} \rightarrow \mathbb{R}^{\text{N}_{\text{out}}}$ takes an input f of length L and N_{in} channels and multiplies each element along the spatial dimensions of the input with a matrix of dimensions $[\text{N}_{\text{in}}, \text{N}_{\text{out}}]$ to produce an output of the same length, but with N_{out} number of channels. The total operations required in this layer is given by $\mathcal{C}_{\text{lin}}(f)$=$\text{L}\cdot\text{N}_{\text{in}}\cdot\text{N}_{\text{out}}$.

Now, let us use three differentiable masks m_{res}, $m_{\text{N}_{\text{in}}}$ and $m_{\text{N}_{\text{out}}}$ to mask the resolution, input and output channels of the linear layer. The total number of computations is now given by:

$$\mathcal{C}_{\text{lin,masked}} = \text{size}(m_{\text{res}})\cdot\text{size}(m_{\text{N}_{\text{in}}})\cdot\text{size}(m_{\text{N}_{\text{out}}}).$$

Since the size of the masks is now involved in the computation of the operations required, we can utilize it as an additional source of feedback to update the masks by making the function size differentiable with regard to the mask parameters. The same concept is used to calculate the cost of other layers based on the size of the masks. A summary of these costs can be found in Appx. C.2.

Effect of the depth mask. To take into account the effect of the depth mask, we use it to determine the number of residual blocks in the network. If the number of operations of a network with D residual blocks is denoted as $\mathcal{C}_{\text{net,D}}$, the complexity of a network with masked depth is given by $\mathcal{C}_{\text{net,size}(\text{m}_{\text{depth}})}$ with $\text{size}(m_{\text{depth}})$ the size of the depth mask.

Computing the size **of the masks.** The size of a mask can be calculated in a differentiable manner by determining the length of the mask in continuous space and using that length to estimate the change in size of the corresponding network dimension. Specifically, the length at a time t is $2x^t_{T_m}$ and $x^t_{T_m} - x_{\min}$, for Gaussian and Sigmoid masks, respectively (see Fig. 4). For some initial $x^0_{T_m}$ and corresponding initial length N, the size of a Gaussian and a Sigmoid mask at time t is respectively:

$$\text{size}(m_{\text{gauss}}) = \frac{2x^t_{T_m}}{2x^0_{T_m}}\text{N}, \quad \text{and} \tag{10}$$

$$\text{size}(m_{\text{sigm}}) = \frac{x^t_{T_m} - x_{\min}}{x^0_{T_m} - x_{\min}}\text{N}. \tag{11}$$

Computational constraints as an additional loss. Let $\mathcal{C}_{\text{curr}}$ be the current complexity of the network and $\mathcal{C}_{\text{target}}$ be the desired target complexity. We define the computational loss $\mathcal{L}_{\text{comp}}$ as the relative ℓ^2 difference between the relative complexity of the current network and the target:

$$\ell^2\left(\frac{\mathcal{C}_{\text{curr}}}{\mathcal{C}_{\text{target}}}, 1\right) = \left\|\frac{\mathcal{C}_{\text{curr}}}{\mathcal{C}_{\text{target}}} - 1.0\right\|_2^2. \tag{12}$$

This form has two advantages over the alternative form $\ell^2(\mathcal{C}_{\text{curr}}, \mathcal{C}_{\text{target}})$. It (*i*) prevents overflow that might occur when comparing large values –$\mathcal{C}_{\text{curr}}$ and $\mathcal{C}_{\text{target}}$ may easily be of order 1e^{10}–, and (*ii*) allows for consistent tuning of λ for different tasks and complexities. In the alternative form $\ell^2(\mathcal{C}_{\text{curr}}, \mathcal{C}_{\text{target}})$, λ might need to be tuned independently for different complexity regimes.

C.2. Computational complexity of masked network components

In this section, we derive the computational complexity of all layers used in the CCNN architecture with and without the use of masks. The calculation of these complexities follows the same reasoning as the pointwise linear layer provided as example in the main text.

With L, N_{in} and N_{out} the length, number of input channels and number of output channels of a given layer, and $\text{size}(m_{\text{res}})$, $\text{size}(m_{\text{N}_{\text{in}}})$, $\text{size}(m_{\text{N}_{\text{out}}})$ the size of the masks along the corresponding dimensions, the complexity of the layers used in the CCNN architectures are given by:

Pointwise linear layer:

$$\mathcal{C}_{\text{lin}}(f) = \text{L} \cdot \text{N}_{\text{in}} \cdot \text{N}_{\text{out}}$$
$$\mathcal{C}_{\text{lin,masked}} = \text{size}(m_{\text{res}}) \cdot \text{size}(m_{\text{N}_{\text{in}}}) \cdot \text{size}(m_{\text{N}_{\text{out}}})$$

Fourier convolution:

$$\mathcal{C}_{\mathcal{F}\text{conv}} = \text{L}\log(\text{L})$$
$$\mathcal{C}_{\mathcal{F}\text{conv,masked}} = \text{size}(m_{\text{res}})\log(\text{size}(m_{\text{res}}))$$

Pointwise operations –GELU, DropOut, etc.–:

$$\mathcal{C}_{\text{pointwise}} = \text{L} \cdot \text{N}_{\text{in}}$$
$$\mathcal{C}_{\text{pointwise,masked}} = \text{size}(m_{\text{res}}) \cdot \text{size}(m_{\text{N}_{\text{in}}})$$

D. Dataset descriptions

D.1. The Long Range Arena benchmark

The Long Range Arena (Tay et al., 2020) consists of six sequence modelling tasks with sequence lenghts ranging from 1024 to over 16000. It encompasses modalities and objectives that require similarity, structural, and visuospatial reasoning. We follow the data preprocessing steps of Gu et al. (2021), which we also include here for completeness.

ListOps. An extended version of the dataset presented by Nangia & Bowman (2018). The task involves computing the integer result in the range zero to nine of a mathematical expression represented in prefix notation with brackets, e.g., [MAX29[MIN47]0] → 9. Characters are encoded as one-hot vectors, with 17 unique values possible (opening brackets and operators are grouped into a single token). The sequences are of unequal length. Hence, the end of shorter sequences is padded with a fixed indicator value to a maximum length of 2048. The task has 10 different classes representing the possible integer results of the expression. It consists of 96K training sequences, 2K validation sequences, and 2K test sequences. No data normalization is applied.

Text. Based on the IMDB sentiment analysis dataset presented by Maas et al. (2011), the task is to classify movie reviews as having a positive or negative sentiment. The reviews are presented as a sequence of 129 unique integer tokens padded to a maximum length of 4096. The dataset contains 25K training sequences and 25K test sequences. No validation set is provided. No data normalization is applied.

Retrieval. Based on the ACL Anthology network corpus presented by Radev et al. (2013), the task is to classify whether two given textual citations are equivalent. To accomplish this, each citation is separately passed through an encoder, and passed to a final classifier layer. Denoting X_1 as the encoding for the first document and X_2 as the encoding for the second document, four features are created and concatenated together as:

$$X = [X_1, X_2, X_1 \times X_2, X_1 - X_2],$$

which are subsequently passed to a two layered MLP. The goal of the task is to evaluate how well the network can represent the text by evaluating if the two citations are equivalent or not. Characters are encoded into a one-hot vector with 97 unique values and sequences are padded to a maximum length of 4000. The dataset includes 147.086 training pairs, 18.090 validation pairs, and 17.437 test pairs. No normalization is applied.

Image. The Image task uses 32×32 images of the CIFAR10 dataset (Krizhevsky et al., 2009). It views the images as sequences of length 1024 that correspond to a one-dimensional raster scan of the image. There are a total of 10 classes, 45K training examples, 5K validation examples and 10K test examples. The RGB pixel values are converted to grayscale intensities and then normalized to have zero mean and unit variance across the entire dataset.

PathFinder. Based on the PathFinder challenge introduced by Linsley et al. (2018), the task presents a 32×32 grayscale image with an start and an end point depicted as small circles. The task is to classify whether there is a dashed line (or path) joining the start and end points while presenting the input as a one-dimension raster scan of the image, alike the `Image` task. The dataset includes 160K training examples, 20Kvalidation examples and 20K test examples. The input data is normalized to be in the range [-1, 1].

Path-X. Path-X is an "extreme" version of the `PathFinder` dataset, in which the input images are of size 128×128. As a result, the input sequences are sixteen times longer with a total length of 16384. Aside from this difference, the task is identical to the `PathFinder` dataset.

D.2. Image classification datasets

CIFAR10 and CIFAR100. The CIFAR10 dataset (Krizhevsky et al., 2009) consists of 60K real-world 32×32 RGB images uniformly drawn from 10 classes divided into training and test sets of 50K and 10K samples, respectively. The CIFAR100 dataset (Krizhevsky et al., 2009) is similar to the CIFAR10 dataset, with the difference that the images are uniformly drawn from 100 different classes. For validation purposes, we divide the training dataset of both CIFAR10 and CIFAR100 into training and validation sets of 45K and 5K samples, respectively. Both datsets are normalized to have zero mean and unit variance across the entire dataset.

D.3. NAS-Bench-360

NAS-Bench-360 (Tu et al., 2022) is a benchmark suite to evaluate Neural Architecture Search methods beyond image classification. The benchmark is composed of ten tasks spanning a diverse array of application domains, datset sizes, problem dimensionalities, and learning objectives. In this work, we consider two tasks from the NAS-Bench-360 suite which require dense predictions: The `DarcyFlow` (Li et al., 2020) and `Cosmic` (Zhang & Bloom, 2020) datasets.

DarcyFlow: Solving Partial Differential Equations. DarcyFlow aims to solve Partial Differential Equation (PDE) by using neural networks as a replacement for traditional solvers. The input for this task is a 85×85 grid specifying the initial conditions and coordinates of a fluid, and the output is a 2D grid of the same dimensions representing the fluid state at a later time. The ground truth for this task is the result computed by a traditional solver, and the objective is to minimize the Mean Squared Error (MSE) between the predicted fluid state and the ground truth.

Cosmic: Identifying Cosmic Ray Contamination. The Cosmic task involves identifying and masking corruption caused by charged particles collectively referred to as "cosmic rays" on images taken from space-based facilities. It uses imaging data of local resolved galaxies collected from the Gubble Space Telescope. The input is an 128×128 image corresponding to the artifact of cosmic rays, and the output is a 2D grid of the same dimensions predicting whether each pixel in the input is an artifact of cosmic rays. We report the false-negative rate of identification results.

E. Experimental details

E.1. General remarks

Experimental setup. We use two CCNNs of different capacity as base networks: a $\text{CCNN}_{4,140}$ –4 blocks, 140 channels, 200K parameters–, and a $\text{CCNN}_{6,380}$ –6 blocks, 380 channels, 2M parameters–, and use DNArch to learn their architectures. To understand the impact of learning each network component, we also report results learning some and none of the neural architecture components.

Mask configurations. We initialize all the masks to match the architecture of the baseline CCNNs at the beginning of training. We use a Gaussian mask to learn kernel sizes as in FlexConv (Romero et al., 2021a), and Sigmoid masks to learn width, depth and downsampling. All masks use a threshold of T_m=0.1. All kernel masks are centered, i.e., μ=0, and initialized to either be small or global, i.e., $\sigma \in [0.0325, 0.5]$. Resolution masks are initialized to weight the highest input frequency by 0.85, and the width and depth masks are initialized to match the size of the base network's architecture. More information on hyperparameters, training regimes, and experimental settings can be found in Appx. E.

Notations. We use DNArch as an operator acting on a base network and specify the learned components with indices $\text{K}, \text{R}, \text{W}, \text{D}$ representing kernel sizes, downsampling, width and depth. $\text{DNArch}_{\text{K}}(\text{CCNN}_{4,140})$ indicates using DNArch to learn only the kernel sizes of a $\text{CCNN}_{4,140}$.

Code. Our code is written in `JAX` and our experiments are conducted on `TPUs` and `GPUs`. As outlined in the Limitations (Sec. F), `JAX` and `TPU` training prevent DNArch from performing operations that change the dimensions of arrays during training. In addition to our `JAX` implementation, we will also release a `PyTorch` implementation of DNArch that supports these operations. We hope that this implementation will make DNArch more flexible and accessible, especially in scenarios where it is crucial to keep candidate networks close to the target complexity during the course of training.

The Continuous CNN and the CCNN residual block. The CCNN architecture is shown in Fig. 5. It is composed by

an `Encoder`, a `Decoder`, and a number of CCNN residual blocks `ResBlock` (**?**). The `Encoder` is defined as a sequence of [`PWLinear`, `BatchNorm` (Ioffe & Szegedy, 2015), `GELU` (Hendrycks & Gimpel, 2016)] layers. For tasks dealing with text, we additionally utilize an Embedding layer mapping each token in the vocabulary to a vector representation of length equal to that used by Gu et al. (2021) (see Appx. D.1). For dense prediction tasks, the `Decoder` is a `PWLinear` layer, which is preceded by `GlobalAvgPooling` for global prediction tasks.

Batch Normalization in DNArch. As the architecture is constantly changing during the search process, we use batch-specific statistics for batch normalization instead of the global moving average. This approach was adopted after early experiments showed that using the global moving average leads to a significant discrepancy in the behavior of the validation and training curves. Specifically, we observed that while the training curves were converging to a good solution, the validation curves resembled random predictions. This issue was resolved by deactivating the global moving average in Batch Normalization layers.

Continuous convolutional kernels $\texttt{MLP}_\psi$. We parameterize our convolutional kernels as a 4-layer MLP with 128 hidden units and a Fourier Encoding (Tancik et al., 2020) of the form $\gamma(\mathbf{x})$=$[\cos(2\pi\omega_0\mathbf{W}\mathbf{x}]), \sin(2\pi\omega_0\mathbf{W}\mathbf{x})]$, with $\mathbf{W} \in \mathbb{R}^{\mathrm{D}\times 128}$ and ω_0 a hyperparameter that acts as a prior on the frequency content of the kernels (Romero et al., 2021b; Sitzmann et al., 2020). In contrast to Romero et al. (2021b), we utilize a single larger $\texttt{MLP}_\psi$ to generate the kernels of the entire network. This allows the network $\texttt{MLP}_\psi$ to administrate its capacity across all layers. Using different $\texttt{MLP}_\psi$'s for each layer as Romero et al. (2021b) is inadequate in the learnable architectures setting as some layers can be entirely erased. With our proposed solution, the capacity of the otherwise zeroed-out $\texttt{MLP}_\psi$ is used to generate the kernels of the remaining layers.

Normalized relative positions. Following Romero et al. (2021b;a), we normalize the coordinates going into $\texttt{MLP}_\psi$ to lie in the space $[-1, 1]^{\mathrm{D}}$ for D-dimensional kernels.

Parameters and hyperparameters of the differentiable masks. We learn some of the parameters of the masks, and leave the others constant or treat them as a hyperparameter. Specifically, for Gaussian masks, we only learn their width, i.e., σ, and fix its mean to zero. For Sigmoid masks, we learn their offset μ and treat their temperature τ as a hyperparameter. For more information regarding the values of τ used in our parameter tuning step, please refer to Appx. E.2.

Maximum and minimum allowable sizes for the learnable differentiable masks. We define some minimum and maximum allowable sizes for the mask parameters, and reset them to these values after each training iteration if the updated parameter values lie outside that range. For the Gaussian mask, we constraint the minimum admissible value of σ such that the length of the corresponding dimension never collapses to a value of 1. This is to prevent the corresponding dimension to collapse such that it can grow afterwards if required. The minimum value depends on the resolution of the corresponding dimension, e.g., the maximum size of the convolutional kernel, and can be easily calculated with Eq. 5.

Note that the offset value of the Sigmoid mask μ could in principle assume any value in $\mathbb{R}$. However, if not controlled, μ could become too small and mask all values along a particular dimension to zero. Similarly, if μ is too large, the gradient of the mask at all positions would become very small and it would difficult to update the mask. To avoid these situations, we define minimum and maximum values of μ such that for the lowest value, the mask at the lowest position is equal to 0.95, and for the largest value, the mask at the highest position is equal 0.85. These values are dependent on the value of the mask temperature τ, and can be easily calculated with Eq. 6.

Limiting the size of the mask to the maximum allowable ranges. As outlined in the Limitations (Sec. F), we must set a maximum allowable size for the width and depth of the network on `JAX`. However, the maximum allowed value for the parameters of the masks (see previous paragraph) allows both masks to grow beyond the point on which the theoretical size of the masks is equal to the maximum allowable network size. For instance, for the maximum allowed parameter values of a Sigmoid mask, the last channel, i.e., the 280-th channel, would be weighted by a factor of 0.85. Consequently, the theoretical size of the mask as calculated by Eq. 11 will be well beyond 280. This value would lead to an unrealistic theoretical computational complexity that surpasses the real computational complexity the CCNN used.

To overcome this issue, we limit the maximum size of the mask calculated by Eq. 11 to be less or equal than the maximum allowable size, e.g., $\mathrm{size}(m)$= $\min(\mathrm{size}(m), 280)$. It is important to note, however, that clipping the value of size directly would stop the gradient flow for parameter values leading to sizes larger 280. As a result, once the maximum size is reached, the mask would not be able to contract anymore. We avoid gradient flow stop by using clipping in combination with a straight-through estimator (Bengio et al., 2013). As a result, we are able to propagate the gradient across the clipping operation, and the resulting mask can still be modified even in the cropping operation is used.

E.2. Hyperparameters and training configurations

In this section, we include more information about the found hyperparameters, the values that were considered during

Table 4. Hyperparameters used for the experiments with the target complexity of a $\text{CCNN}_{4,140}$.

Dataset	Epochs	Batch Size	Learning Rate	DropOut	Weight Decay	ω_0	λ	$\tau_{\text{resolution}}$	τ_{channel}	τ_{depth}
ListOps	50	50	0.005	0.0	0.01	27.5K	5.0	50	25	8
Text	100	50	0.02	0.2	0.01	19.5K	0.1	50	25	8
Retrieval	50	50	0.001	0.1	0.01	21.5K	0.1	50	25	8
Image	210	50	0.02	0.1	0.001	12.5K	0.1	25	25	8
PathFinder	210	50	0.005	0.0	0.001	21.5K	0.1	50	25	8
Path-X	80	32	0.001	0.0	0.0	30K	0.1	100	25	8
CIFAR10	210	50	0.01	0.1	0.01	21.5K	0.1	25	25	8
CIFAR100	210	50	0.01	0.0	0.01	6.5K	5.0	25	25	8
DarcyFlow	310	8	0.02	0.0	0.0001	24.5K	0.1	50	25	8
Cosmic	310	8	0.02	0.3	0.0001	5.5K	0.1	100	25	8

Table 5. Hyperparameters used for the experiments with the target complexity of a $\text{CCNN}_{6,380}$.

Dataset	Epochs	Batch Size	Learning Rate	DropOut	Weight Decay	ω_0	λ	$\tau_{\text{resolution}}$	τ_{channel}	τ_{depth}
CIFAR10	210	50	0.005	0.0	0.01	21.5K	0.1	50	50	16
CIFAR100	210	50	0.01	0.0	0.01	6.5K	0.1	25	25	8
DarcyFlow	310	12	0.01	0.2	0.0	7.5K	0.1	50	25	8
Cosmic	310	4	0.01	0.2	0.01	5.5K	0.1	50	50	8

hyperparameter tuning, and other training settings. The final hyperparameters used are listed in Table 5.

Optimizer, learning rates and learning rate schedule. All our models are optimized with AdamW (Loshchilov & Hutter, 2017) in combination with a cosine annealing learning rate scheduler (Loshchilov & Hutter, 2016), and a linear learning rate warm-up stage of 10 epochs, except for `ListOps`, `Retrieval` and `Path-X` for which we have a warm-up stage of 5 epochs.

Regularization. We utilize dropout (Srivastava et al., 2014) –as shown in Fig. 5– as well as weight decay during training.

E.2.1. Hyperparameter tuning

Frequency prior ω_0. The possible ω_0 values explored in this work are $[1, 500, 1500, 2500, ...28500, 29500, 30000]$.

Tuning the value of λ. λ plays the role of controlling the weight of the computational loss $\mathcal{L}_{\text{comp}}$ relative to the task objective loss $\mathcal{L}_{\text{obj}}$. In this work, we find two settings which require different values of λ. One, given by the tasks that converge to a low prediction values relative to perfection, i.e., `ListOps` and `CIFAR100`, and for which the loss $\mathcal{L}_{\text{obj}}$ remains relatively high at the end of training. The other group is given by all the other tasks, which converge to high prediction values –many even obtaining a perfect accuracy on the train set–, and for which $\mathcal{L}_{\text{obj}}$ converges to values close to zero. For the first group, we require a higher value of λ such that the computational complexity loss $\mathcal{L}_{\text{comp}}$ remains relevant to the optimization objective. The final values of λ used are 5.0 and 0.1, respectively.

Tuning the temperature of the Sigmoid masks τ. For the resolution mask, we consider three values of τ, $[25, 50, 100]$ which correspond to a minimum size of 10%, 5% and 2.5% of the corresponding dimension. For the channel mask, we consider two values $\tau \in [25, 50]$ which correspond to a minimum size of 10% and 5% of the corresponding dimension, but observe early during tuning that models prefer τ=25. For the depth dimension, which is much more sparse than the channel and resolution dimensions, we consider two values $\tau \in [8, 16]$, which result on a minimum depth of 2 and 1 layers, respectively. We observe early during tuning that models prefer τ=8.

Learning rate. The possible learning rate values explored in this work are $[0.0001, 0.0005, 0.001, 0.005, 0.01, 0.02]$.

Dropout. The possible dropout values explored in this work are $[0.0, 0.1, 0.2, 0.3]$.

Weight decay. The possible weight decay values considered in this work are $[0.0, 0.0001, 0.001, 0.01, 0.05]$.

E.3. Computational cost of DNArch

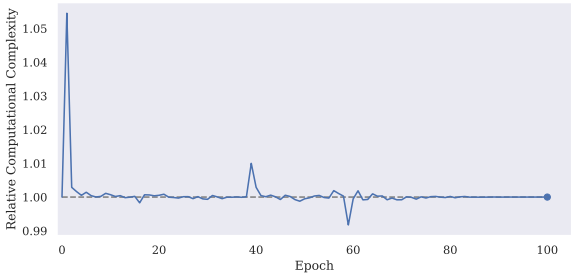

Figure 9. Relative complexity during the course of training on the `Text` task. This behavior is consistent across all tasks.

F. Limitations

Training on `TPU` requires static shapes. We train our models on `TPUs`, a type of accelerator that requires a static computational graph derived for specific input and network shapes via the `XLA` (Accelerated Linear Algebra) compiler.

As a result, TPUs do not support operations that change the shapes of arrays during training. This means that on TPUs, DNArch can only perform masking modifications to the network during training, i.e., setting certain channels to zero but still computing their outputs. At inference, however, the masks are fixed. Consequently, we can effectively trim unused values to remove useless computations in a way that is compatible with XLA. It is important to note that this limitation is solely an implementation issue caused by nature of TPUs' hardware and can be avoided by using libraries and hardware that support dynamic computational graphs, e.g., PyTorch and GPUs. While our results were obtained using TPUs, we also provide a PyTorch implementation that avoids this issue, making it more flexible and accessible, especially in scenarios where one needs to keep candidate networks close to the target complexity C_{target} during training.

DNArch requires instantiating the largest possible architecture. While masking weights through a gradient update is straightforward, increasing the number of active weights requires those weights to be instantiated in memory. This means that even with dynamic computational graphs, it is necessary to instantiate the largest possible architecture learnable by DNArch in memory. To overcome this limitation, we set the maximum kernel size to the size of the input, and limit the maximum network size along the depth and width dimensions to double the number of blocks and channels of the base network. While this trick allows DNArch to easily shrink and grow representations within that range, this restricts the potential sizes of optimal architectures and can restrict the applicability of DNArch to very large models, e.g., LLMs (Brown et al., 2020; Chowdhery et al., 2022), which can have billions of weights.

G. Outlook and future work

Input-dependent neural architectures. In this work, the mask parameters are constant for all inputs within a task. An alternative approach could use an additional neural network $\texttt{MLP}_{\text{mask}}$ to predict the mask parameters based on context, e.g., the current input, current task, etc. This would enable the creation of context-dependent neural architectures such as early-exit systems (Teerapittayanon et al., 2016; Ghodrati et al., 2021; Schuster et al., 2022), but where the whole network architecture is context-dependent. Consequently, resulting architectures would providing finer control of per-sample / per-modality complexity than existing methods.

Dynamic weighting of $\mathcal{L}_{\text{comp}}$ during training. DNArch explores architectures with complexity similar to target complexity throughout training. This results from using a constant λ in Eq. 9. Alternatively, one could use a dynamic value of λ during training to induce a different training behavior. For example, gradually increasing λ would allow DNArch to explore architectures with larger complexity at first, and progressively encourage it to converge to networks with the desired target complexity. Such a weighting scheduling of λ could lead DNArch to find better architectures.

Training DNArch with additional / multiple constraints. Here, we only consider computational complexity as a constraint when training with DNArch. However, other properties such as memory efficiency, hardware-awareness and robustness are equally important. Designing regularization terms that encourage other properties in DNArch as well as exploring how different properties can be optimized in unison are important directions for further research.

H. Architectures Found by DNArch

Table 6. Architectures found by DNArch on LRA with the target complexity of a $\text{CCNN}_4, 140$.

Task	Depth	Kernel Size	Resolution	Width $[N_{\text{in}}, N_{\text{mid}}, N_{\text{out}}]$
ListOps	8	266	2048	[150 189 145]
		569	632	[150 168 168]
		1401	1416	[176 186 162]
		310	310	[166 175 153]
		213	213	[154 159 163]
		12	301	[168 128 162]
		5	250	[170 158 153]
		24	502	[153 171 165]
Text	8	445	2284	[180 217 205]
		691	2939	[208 176 153]
		1420	1420	[152 152 120]
		415	1313	[120 120 147]
		1467	1467	[147 118 135]
		52	594	[134 173 153]
		101	932	[150 156 183]
		149	1036	[180 92 192]
Retrieval	8	2	1913	[29 33 172]
		136	2058	[184 174 183]
		1013	2363	[205 171 161]
		1446	2724	[188 164 115]
		7	2604	[29 29 163]
		1	2756	[29 35 154]
		6	3545	[71 110 147]
		1	3899	[71 88 137]
Image	8	203	1024	[118 155 147]
		279	1024	[146 172 164]
		219	486	[173 166 196]
		308	308	[199 197 196]
		144	144	[207 197 92]
		8	125	[106 29 75]
		30	96	[78 28 110]
		40	126	[104 51 104]
PathFinder	8	195	1024	[109 140 171]
		493	770	[171 168 158]
		418	507	[144 183 170]
		318	318	[173 187 178]
		236	236	[182 162 160]
		231	231	[161 121 103]
		8	251	[105 47 210]
		4	253	[116 29 188]
Path-X	5	2484	15331	[280 174 157]
		7204	7204	[177 280 159]
		3669	3772	[167 280 98]
		2323	5496	[123 164 164]
		513	4768	[136 128 195]

Table 7. Architectures found by DNArch on 2D datasets with the target complexity of a $\text{CCNN}_{4,140}$.

TASK	DEPTH	KERNEL SIZE [Y X]	RESOLUTION [Y X]	WIDTH [N_{in}, N_{mid}, N_{out}]
IMAGE CLASSIFICATION TASKS				
CIFAR10	8	[9 7]	[32 32]	[142 139 145]
		[12 8]	[32 32]	[145 160 157]
		[25 7]	[32 20]	[158 186 182]
		[9 10]	[9 15]	[186 208 168]
		[1 13]	[5 15]	[169 177 150]
		[1 10]	[6 11]	[151 139 156]
		[5 1]	[15 4]	[154 115 110]
		[6 5]	[11 7]	[108 41 166]
CIFAR100	8	[13 7]	[32 32]	[104 107 116]
		[6 10]	[32 32]	[114 134 134]
		[11 8]	[22 22]	[139 192 166]
		[13 7]	[16 18]	[173 201 197]
		[8 12]	[10 12]	[205 251 51]
		[1 1]	[8 9]	[62 56 157]
		[5 9]	[8 10]	[162 175 254]
		[8 7]	[9 9]	[280 280 280]
DENSE TASKS				
DARCY FLOW	3	[43 38]	[80 72]	[156 280 107]
		[22 22]	[22 22]	[180 204 78]
		[76 76]	[85 85]	[280 280 50]
COSMIC	6	[94 111]	[128 128]	[18 110 23]
		[2 13]	[20 45]	[186 207 139]
		[129 129]	[129 129]	[126 265 100]
		[129 121]	[129 129]	[78 105 59]
		[90 89]	[129 129]	[57 201 197]
		[76 74]	[76 74]	202 145 216]

Table 8. Architectures found by DNArch on 2D datasets with the target complexity of a $\text{CCNN}_{6,380}$.

TASK	DEPTH	KERNEL SIZE [Y X]	RESOLUTION [Y X]	WIDTH [N_{in}, N_{mid}, N_{out}]
IMAGE CLASSIFICATION TASKS				
CIFAR10	12	[4 7]	[32 32]	[380 328 384]
		[9 10]	[32 32]	[384 371 393]
		[12 6]	[32 32]	[392 361 391]
		[20 6]	[32 32]	[388 370 421]
		[10 11]	[23 26]	[421 417 486]
		[11 11]	[12 22]	[496 444 479]
		[1 11]	[6 11]	[493 482 304]
		[1 6]	[5 21]	[211 78 384]
		[29 4]	[32 4]	[363 459 280]
		[18 15]	[18 15]	[277 394 67]
		[1 1]	[4 4]	[111 109 361]
		[4 3]	[21 15]	[121 374 449]
CIFAR100	12	[8 9]	[32 32]	[343 275 354]
		[12 10]	[32 32]	[351 316 397]
		[11 10]	[32 32]	[495 355 420]
		[18 12]	[29 21]	[421 498 419]
		[11 15]	[27 24]	[432 449 407]
		[19 8]	[25 20]	[412 419 413]
		[11 10]	[12 23]	[423 454 600]
		[8 8]	[8 9]	[709 685 416]
		[5 7]	[5 8]	[419 311 446]
		[8 4]	[8 4]	[446 433 389]
		[6 4]	[6 4]	[386 501 570]
		[8 9]	[8 9]	[568 453 655]
DENSE TASKS				
DARCY FLOW	7	[54 49]	[54 49]	[435 428 289]
		[43 47]	[70 72]	[499 393 284]
		[65 69]	[85 85]	[496 434 281]
		[67 66]	[85 85]	[323 412 275]
		[85 85]	[85 85]	[319 369 271]
		[85 85]	[85 85]	[306 379 258]
		[68 68]	[85 85]	[521 435 271]
COSMIC	12	[35 32]	[35 33]	[146 236 272]
		[11 21]	[95 72]	[170 284 319]
		[44 24]	[128 128]	[141 339 388]
		[23 41]	[128 128]	[385 407 361]
		[28 27]	[128 128]	[351 279 356]
		[21 19]	[128 128]	[354 362 310]
		[29 24]	[128 128]	[310 351 466]]
		[18 25]	[128 128]	[396 292 183]
		[57 16]	[128 128]	[179 210 580]
		[50 11]	[127 77]	[273 250 63]
		[18 12]	[89 67]	[347 400 77]
		[22 23]	[97 79]	[171 241 79]