Supplementary Material for 'Dense for the Price of Sparse: Improved Performance of Sparsely Initialized Networks via a Subspace Offset'

A. Additional Experiments

A.1. Tiny Imagenet

Figure A.1 compares the performance of DCTpS with SynFlow and FORCE[1] on Tiny Imagenet with ResNet18. DCTpS obtains higher validation accuracy than both FORCE and SynFlow for all densities less than or equal to 1%, and by 0.1% density DCTpS is outperforming them by approximately 10% accuracy. This confirms that the superior accuracy of DCTpS networks at low densities is maintained when the difficulty of the problem is scaled up.

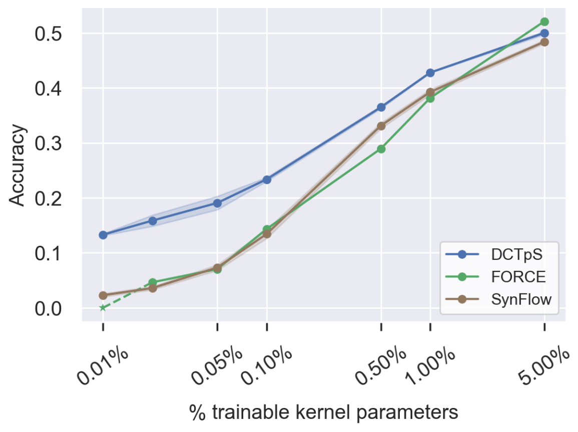

Figure A.1. Comparing DCTpS with FORCE and SynFlow on Tiny Imagenet with ResNet18.

A.2. Equal-per-filter (EPF) Support Distribution

All DCTpS experiments in Section 5 used the EPL heuristic to distribute trainable parameters between layers. Another naive heuristic which achieves the basic goal of maintaining some trainablility throughout the network is an 'Equal per Filter' (EPF) approach: given a specified sparsity, the total number of trainable parameters for the whole network is calculated, and divided equally between all convolutional filters (or rows in the case of linear layers). Within each filter, the locations of those trainable parameters is chosen uniformly at random.

[1] We note that FORCE was particularly unstable in these experiments, failing to prune the network to the required density in at least one of its three runs, at every density less than 0.5%. In these cases, accuracy is averaged over only those runs in which FORCE succeeded. At 0.01% density, FORCE failed on all three runs.

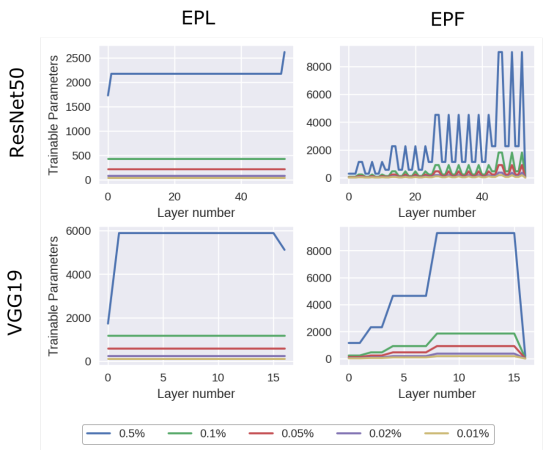

Figure A.2. The total number of trainable parameters per prunable layer determined by EPL and EPF heuristics, on ResNet50 and VGG19 with 10 output classes.

Figure A.2 highlights that the EPL and EPF heuristics result in substantially and qualitatively different layer-wise parameter allocations. Nevertheless, Figure A.3 shows that both methods achieve very similar accuracy across all tested densities, though EPF consistently performs marginally worse. This observation lends further support to the hypothesis that, provided a suitable offset $\boldsymbol{d}$ is used, there is a large class of subspace embeddings from which it suffices to draw U randomly to achieve high accuracy, and in particular that this class includes k-sparse disjoint U with a variety of support distributions.

A.3. Training with SGD as Opposed to Adam

The best test accuracy for large networks like ResNet and VGG is typically obtained by using SGD with momentum and a specified learning rate schedule, rather than adaptive methods like Adam. However, the results obtained with SGD are sensitive to hyperparameters like the initial learning rate and the learning rate schedule. Adam, though it tends to achieve lower final accuracy, is less sensitive to these hyperparameters. This makes Adam a sensible training algorithm for experiments in which the goal is to compare the relative drop in accuracy caused by one or other pruning method, as opposed to a goal of achieving maximum possible accuracy overall. Thus Adam is used, with default

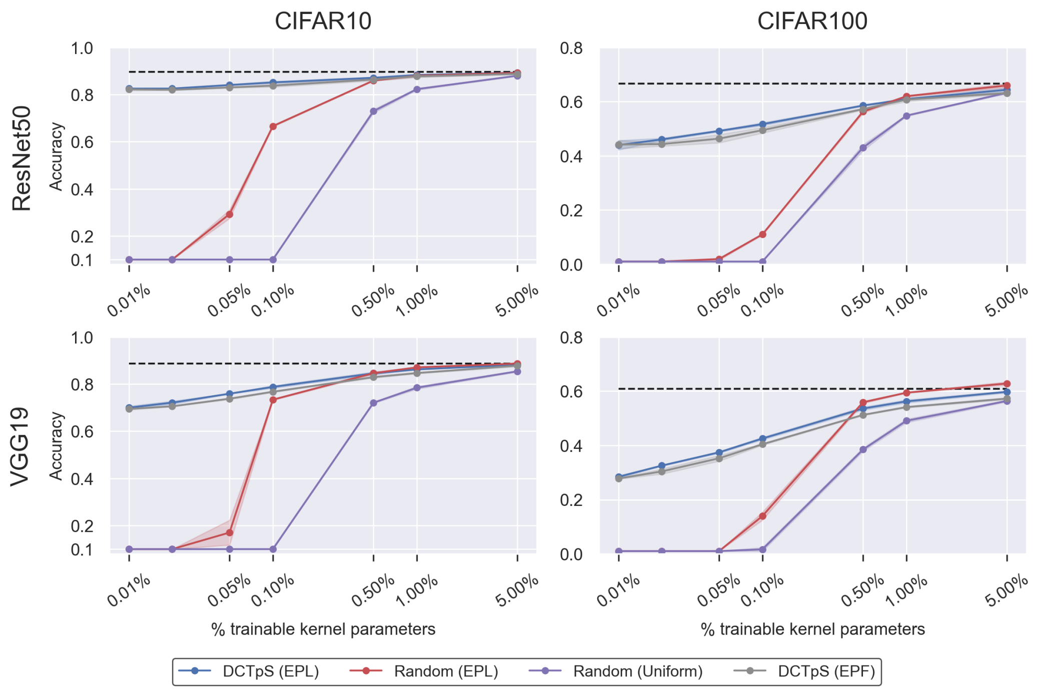

Figure A.3. Comparative accuracy of EPF and EPL heuristics for DCTpS networks.

settings, as the training algorithm for the experiments presented in Section 5. In order to preserve comparison with prior work, and to illustrate that our results are not unique to Adam, additional experiments using SGD with momentum on ResNet50, VGG19, MobileNetV2 and FixUpResNet110 are included here. A single, course sweep of base learning rates [0.1, 0.07, 0.05, 0.03, 0.01] was done with a DCTpS ResNet50 applied to CIFAR100, at 1% density, to select a base learning rate of 0.03, which was then used to train all DCTpS architectures, at all densities, without further fine-tuning. For PaI on standard architectures, a base learning rate of 0.1 was used as done in prior work (de Jorge et al., 2021).

Test accuracy is shown in Figure A.4 for ResNet50 and VGG19, and Figure A.5 for MobileNetV2 and FixupResNet110[2]. The results exhibit qualitatively similar trends to those observed in Section 5's Figures 2 and 3, though with slightly higher overall accuracy, in particular at higher densities, as expected.

A.4. Fixed α

In Figure A.6 we explore the impact of allowing α to be trainable, as compared with being fixed at $\alpha = 1$. The latter case corresponds exactly to subspace training, since the offset $\boldsymbol{d}$ and embedding U are then fixed during training. We can see that even with a fixed α, the same general trends are observed. However, we consistently achieve a 2%-3% increase in accuracy by allowing α to be trainable. We conjecture that with the appropriate initialisation of α, this gap would disappear, and α would not need to be trained.

A.5. Comparing Layer-wise Parameter Allocation Heuristics

In Figure A.7 we compare different heuristics for distributing trainable parameters between network layers – in particular, uniform density per layer (uniform), equal number of parameters per layer (EPL), equal number of parameters per filter (EPF) and the ERK distribution used in (Evci et al., 2020). Though no heuristic performs best in all cases at all densities, EPL performs best in the majority of network-density-dataset combinations - suggesting it could be a useful heuristic for future works to try in methods with fixed layer-wise sparsity distributions, and should certainly be included as an alternative baseline to uniform random pruning in future works.

B. DCTpS Implementation

In the code used to run the experiments in this paper, the DCT components of DCTpS layers have been implemented by setting the tensor W to be the DCT matrix (matrix of DCT basis vectors), as opposed to implementing them via

[2]SynFlow was not able to be included in these additional supplementary experiments having only recently been published (Tanaka et al., 2020).

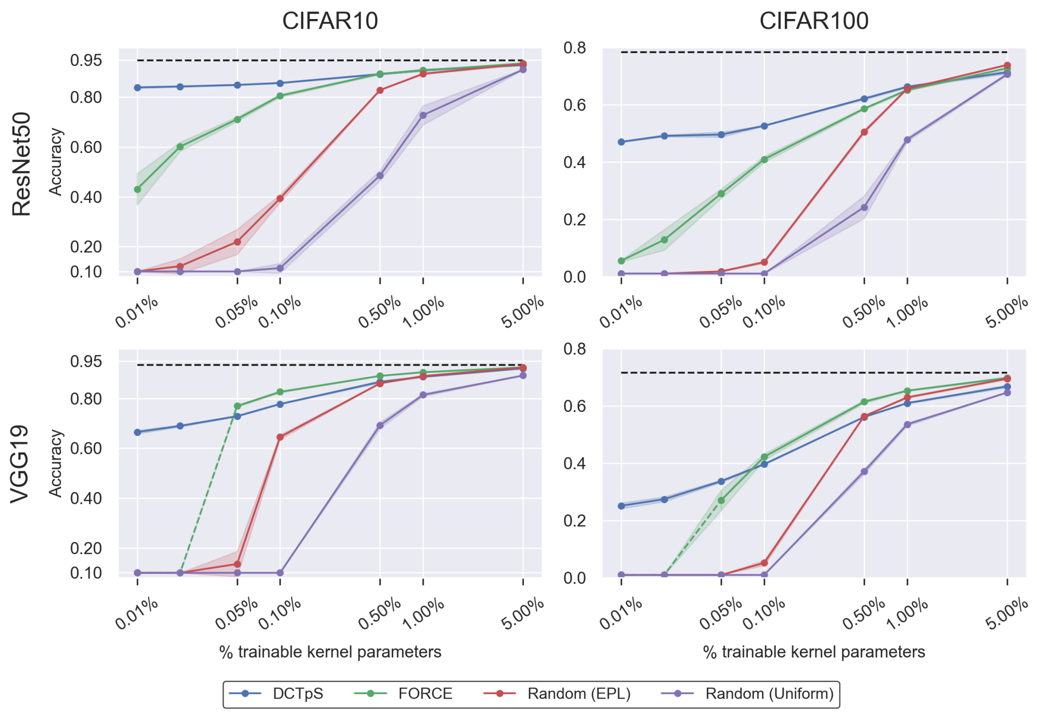

Figure A.4. Experiments training ResNet50 and VGG19 with SGD (with momentum), on CIFAR10 and CIFAR100.

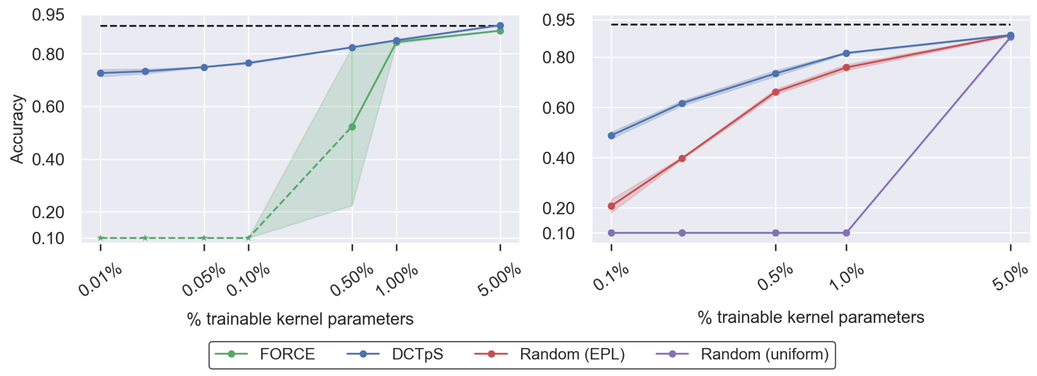

Figure A.5. Experiments training MobileNetV2 (left) and FixupResnet110 (right) with SGD (with momentum), on CIFAR10.

fast transforms. This is because deep learning libraries, as currently implemented, are optimised for the former rather than the latter.

Linear Layers: To be precise, in Linear layers with input $\in \mathbb{R}^n$ and output $\in \mathbb{R}^m$, with $q = \max(m, n)$, the DCT matrix $W \in \mathbb{R}^{q \times q}$ is formed and then truncated to size $m \times n$ by removing the surplus right-most columns (if $m > n$) or bottom-most rows (if $m < n$). Multiplication by this matrix is equivalent to a DCT, with a zero-padded input if $m > n$, or a truncated output if $m < n$.

Convolutional Layers:. In a convolutional layer, with m $(k \times k \times n)$ filters, $q = \max(k^2 n, m)$, the DCT matrix $W \in \mathbb{R}^{q \times q}$ is formed and truncated to size $k^2 n \times m$ and reshape appropriately. In the accompanying code, a simple test script is provided to confirm that our implementation indeed computes the DCT of each patch.

Figure B.8 provides a simple visualisation of how this is compatible with the efficiency of DCTpS layers, if implemented correctly. In the forward pass, each step of the convolution involves taking a patch of the image, and - for each filter - computing the sum of the elementwise product of the patch and the filter. Flattening (commonly known as 'lowering') the filters and the input patch, this is equivalent to a matrix-vector product (and indeed convolutional layers are commonly implemented with this 'lower $\rightarrow$ matrix

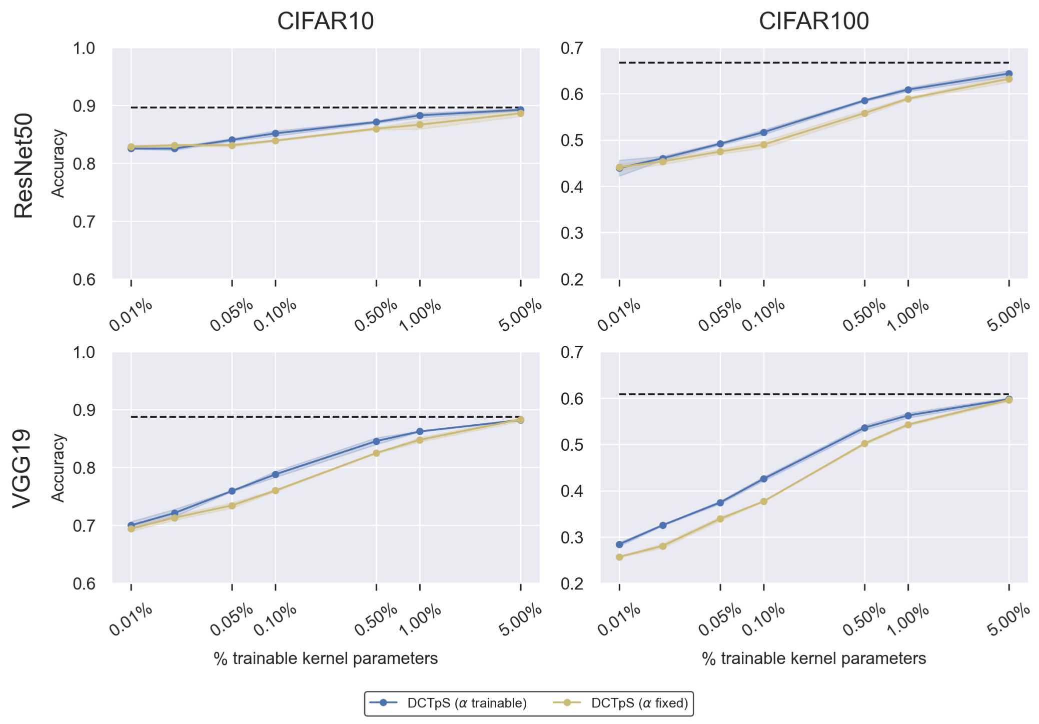

Figure A.6. Comparison of letting α be a trainable parameter with fixing $\alpha = 1$ throughout training in DCTpS networks.

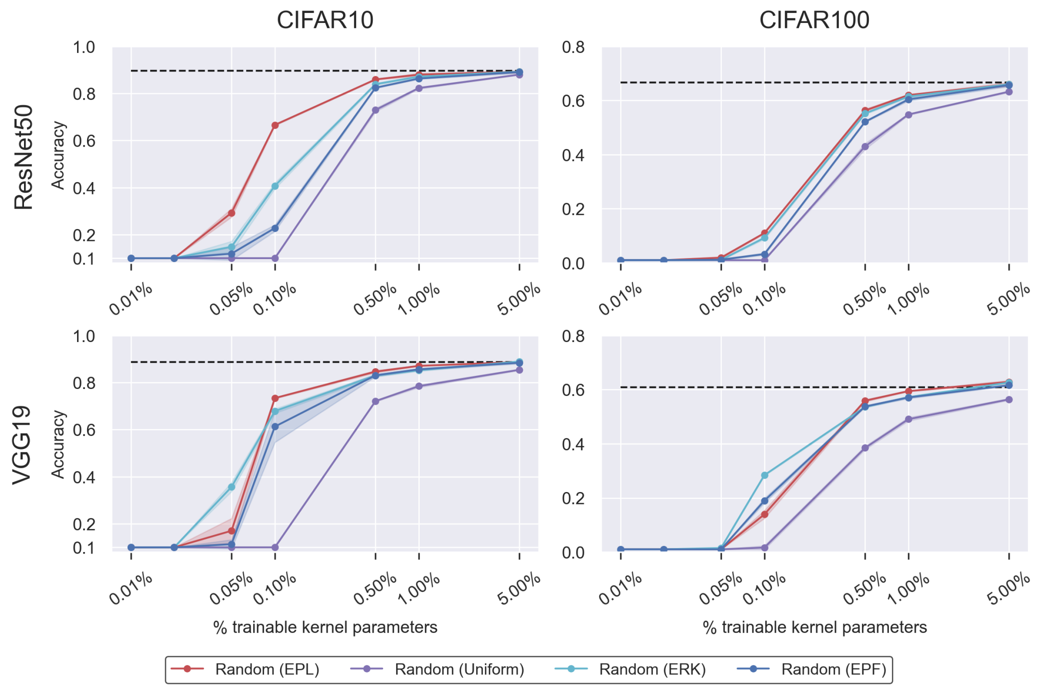

Figure A.7. Comparison of different heuristics for distributing trainable parameters between network layers.

multiply $\rightarrow$ lift' approach (Hadjis et al., 2015)). The DCT part of DCTpS convolutional layers set this flattened matrix of filters to be the DCT matrix, and is thus equivalent to computing the DCT of each patch.

This applies equally for the backward-pass, since backpropagation through convolutional layers involves convolutions as well. Let h be the layer input, y be the output of a layer with a single 2×2 filter F, and let L be the loss. Calculating $\frac{\partial L}{\partial h}$ involves calculating $\frac{\partial L}{\partial y} * \mathrm{Rot}_{180}(F)$. Again, each step in this convolution is equivalent to an inner product between the original filter, and a flattened, *permuted* patch of $\frac{\partial L}{\partial y}$, see Figure B.9. This generalises for larger filters (Boué, 2018).

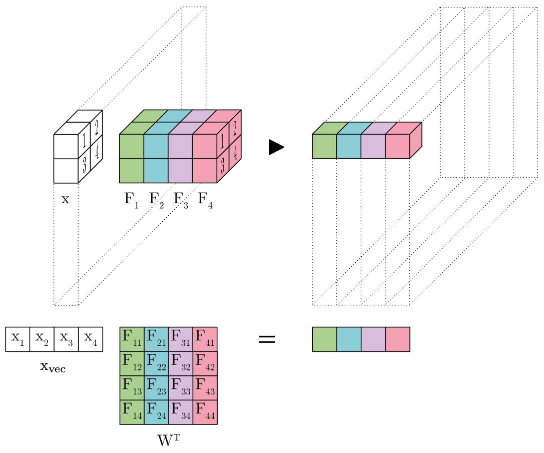

Figure B.8. Illustration of the matrix-vector product involved in each step of a 2D convolutional layer. Computing the DCT of each patch is equivalent to taking W to be a DCT matrix.

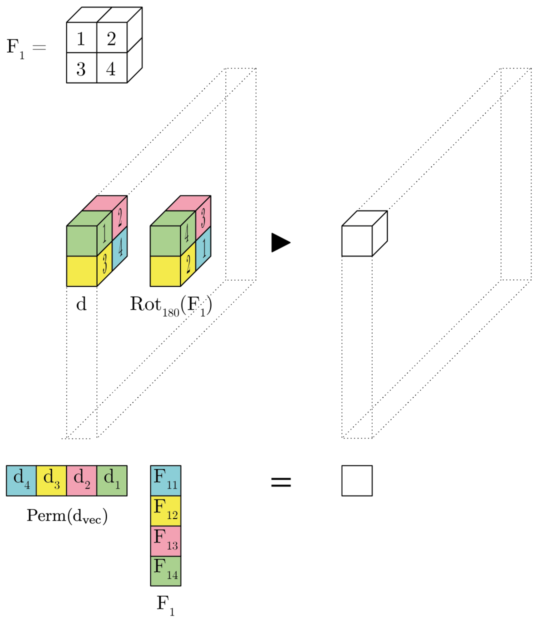

Figure B.9. Back-propagation involves inner products with a layer's filters. In this figure, d represents a patch of $\frac{\partial L}{\partial y}$, and F is one of the layers convolutional filters.

C. Parameter Allocation by SynFlow

It was noted in Section 6 that when applied to ResNet50 for CIFAR10 at modest sparsities, SynFlow pruned fully those residual connections which were prunable (those implemented as trainable, 1×1 convolutions), and in the remaining layers it appeared to approximate the EPL heuristic. Figure C.10 shows that this observation also applies to other architectures with different numbers of output classes. It is particularly striking the extent to which SynFlow applied to VGG19 leaves unpruned an approximately equal number of parameters per layer. On ResNet18, the pruning is observed to have the same structure per layer as on ResNet50 - a roughly equal number of trainable parameters per layer, except for the prunable shortcut connections, which are pruned completely. This behaviour is also present to some extent on MobileNetV2, though with larger oscillations around a central value, and a breakdown of this behaviour at lower densities, at which point multiple layers begin to be pruned in their entirety.

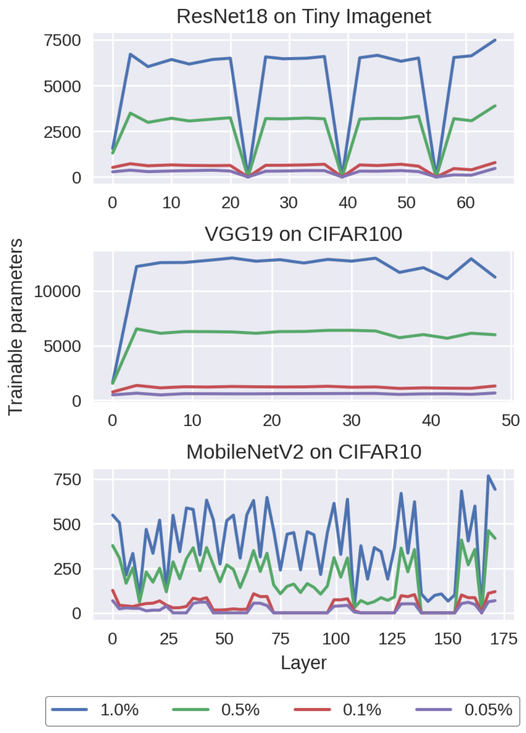

Figure C.10. Total number of trainable parameters per layer, after pruning at initialization with SynFlow.

D. Cases where SynFlow and FORCE cannot be applied

D.1. Extreme Sparsities (FORCE)

In some cases FORCE is unable to successfully prune past a certain sparsity. In particular, at some point in the pruning schedule, FORCE begins to assign all parameters a saliency score of 0, thus providing no basis for pruning, with the consequence being that the algorithm simply returns a dense network. This happens, for example, at the most extreme sparsities in VGG19, ResNet18, and MobileNetV2. In these cases, the test accuracy for FORCE is reported as equal to that of random guessing, since the algorithm cannot provide a trainable network at the given sparsity, but this is denoted with a dashed lined to indicate that no network of the specified sparsity was actually tested.

We conjecture that this phenomenon is a result of *throughput* collapse[3] - that is, once the algorithm fully prunes *all branches* of communication at some point in the network, though we did not investigate this further. We note that in investigating these collapses, we also tried doubling the number of pruning steps from 60 to 120, but this did not avoid the problem.

D.2. Fixup ResNet

Neither FORCE nor SynFlow can be applied without modification to FixupResNet110. As above, this failure is due to the fact that both algorithms assign a saliency score of 0 to all parameters and consequently have no basis on which to prune.

In particular, at initialization, the only non-zero gradients of both the training loss $\mathcal{L}$ and SynFlow's objective function $\mathcal{R}$, are in the network's final layer, where the weights themselves are initialized as 0. The saliency scores in both FORCE and SynFlow are obtained via the elementwise multiplication of the parameter matrices with their gradients, and thus are 0 in all cases.

E. Experimental Details

E.1. Code and Implementation

We implemented Force[4] and SynFlow [5] using the code published by the respective authors, adapted to include any additional architectures used in our experiments. For RigL, we adapted its PyTorch Implementation[6].

The code used to run experiments with DCTpS networks is available at github.com/IlanPrice/DCTpS.

[3]This is equivalent to layer collapse when there is only a single feedforward connection at each layer.

[4]https://github.com/naver/force

[5]https://github.com/ganguli-lab/Synaptic-Flow

[6]https://pypi.org/project/rigl-torch/

E.2. Model Architectures

Standard implementations of network architectures used here are taken from the following sources:

- ResNet50, ResNet18 and VGG19, as implemented in (de Jorge et al., 2021). See the FORCE Github Repo
- MobileNetV2 from the authors' published code here.
- FixupResNet110 from the authors' published code here.

E.3. Parameter Breakdown by Architecture

See Table E.1 for a breakdown of the prunable/non-prunable parameter totals in each of the architectures used in our experiments.

Table E.1. Division of total parameters between weights (pruned) and bias and/or batchnorm (BN) parameters (not pruned) in the architectures used in our experiments, with 10 output classes.

	Weights	Bias & BN	Total
ResNet50	23467712	53130	23520842
VGG19	20024000	11018	20035018
MobileNetV2	2261824	35098	2296922
FixupResNet110	1719856	282	1720138
ResNet18	11164352	9610	11173962

E.4. Pruning hyperparameters

See Table E.2 for the hyperparameters used when applying FORCE and SynFlow. RigL is tested with T_{end} set as 75% of total iterations, the RigL α parameter set to 0.3, with all convolutional and linear layers sparsified, according to the ERK distribution. When RigL is combined with DCTpS networks, EPL is used instead of ERK.

Table E.2. Pruning hyperparameters for FORCE and SynFlow. C10, C100, and TI stand for CIFAR10, CIFAR100, and Tiny Imagenet respectively.

	FORCE	SynFlow
Prune Steps	60	100
# Batches	1 (C10), 10 (C100), 20 (TI)	N/A
Schedule	exp	exp

E.5. Training Details

See Table E.3 for the training hyperparameters used in our experiments in Section 5 and Appendix A. On CIFAR10 and CIFAR100 (Krizhevsky et al., 2009), 10% of the training data is withheld as a validation set. The model with the maximum validation accuracy is selected as our final model, to be evaluated on the test set. In the case of Tiny Imagenet (Wu et al., 2017), where there are no labels for the test set, the maximum validation accuracy obtained during training is reported. All experiments were run using Adam, except for those in Appendix A.3 in which SGD with momentum was used.

Table E.3. Training hyperparameters used for experiments in Section 5 and Appendix A. Note that for training DCTpS networks with SGD, a base learning rate of 0.03 was used instead of 0.1. For experiments with Lenet-5 (only performed with Adam on CIFAR10) batch size was 64 and total epochs was 160.

	Adam	SGD
Epochs	200	200
Batch Size	128	128
Learning Rate (LR)	0.001	0.1
Momentum	N/A	0.9
LR Decay Epochs	N/A	120, 160
LR Drop factor	N/A	0.1
Weight Decay	5×10^{-4}	5×10^{-4}

References

Boué, L. Deep learning for pedestrians: backpropagation in cnns. *arXiv preprint arXiv:1811.11987*, 2018.

de Jorge, P., Sanyal, A., Behl, H., Torr, P., Rogez, G., and Dokania, P. K. Progressive skeletonization: Trimming more fat from a network at initialization. In *International Conference on Learning Representations*, 2021. URL `https://openreview.net/forum?id=9GsFOUyUPi`.

Evci, U., Gale, T., Menick, J., Castro, P. S., and Elsen, E. Rigging the lottery: Making all tickets winners. In *International Conference on Machine Learning*, pp. 2943–2952. PMLR, 2020.

Hadjis, S., Abuzaid, F., Zhang, C., and Ré, C. Caffe con troll: Shallow ideas to speed up deep learning. In *Proceedings of the Fourth Workshop on Data analytics in the Cloud*, pp. 1–4, 2015.

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

Tanaka, H., Kunin, D., Yamins, D. L., and Ganguli, S. Pruning neural networks without any data by iteratively conserving synaptic flow. *Advances in Neural Information Processing Systems*, 33, 2020.

Wu, J., Zhang, Q., and Xu, G. Tiny imagenet challenge, 2017.