Principal Component Networks: Parameter Reduction Early in Training

Roger Waleffe [1] **Theodoros Rekatsinas** [2]

Abstract

In this paper, we show that hidden layer activations in overparameterized neural networks for image classification exist primarily in subspaces smaller than the actual model width. We further show that these subspaces can be identified early in training. Based on these observations, we show how to efficiently find small networks that exhibit similar accuracy to their overparameterized counterparts after only a few training epochs. We term these network architectures Principal Component Networks (PCNs). We evaluate PCNs on CIFAR-10 and ImageNet for VGG and ResNet style architectures and find that PCNs consistently reduce parameter counts with little accuracy loss, thus providing the potential to reduce the computational costs of deep neural network training.

1. Introduction

Recent results suggest the importance of overparameterization in neural networks (Gunasekar et al., 2017; Li et al., 2020). The theoretical results of Gunasekar et al. (2017) demonstrate that training in an overparameterized regime leads to an implicit regularization that may improve generalization. At the same time, empirical results (Li et al., 2020) show that large models can lead to higher test accuracy. Yet, these generalization improvements come with increased time and resource utilization costs: models with more parameters generally require more FLOPS. The question then arises, *how can we retain the generalization benefits of overparameterized training but reduce its computational cost?*

This question has lead to several works which show that overparameterized networks contain *sparse subnetworks* that can be trained in isolation to comparable generalization performance (Frankle & Carbin, 2019; Ramanujan et al., 2019). These results highlight the potential to reduce training costs of high-accuracy models. Subnetwork identification, however, can be an expensive exercise—e.g., requiring iterative train-prune-reset procedures (Frankle & Carbin, 2019)—thus limiting end-to-end cost reductions. This motivates our study to find an efficient procedure for discovering small networks that exhibit the same accuracy as their overparameterized counterparts.

Principal Component Networks Our work builds upon the next compelling observation: We consider overparameterization due to increased network width, a key quantity associated with high-accuracy models (Park et al., 2019). We find that hidden layer activations in wide, image-classification networks exist in low-dimensional subspaces an order of magnitude smaller than the actual model width. We also find that these subspaces, which contribute most to the network accuracy, can be identified early in training.

Based on the above observations, we introduce a new family of deep learning models, which we term *Principal Component Networks* (PCNs). A PCN transforms the wide layers in an original overparameterized network into *smaller layers* that live in a lower dimensional space. To identify the basis of this space for each layer, we use Principal Component Analysis (PCA) to find the high-variance directions that describe the layer's input and output activations. The transformations introduced by PCNs eliminate all weights from the overparameterized model not relevant to these bases.

More specifically, PCNs introduce the following procedure to reduce the computational cost of training while achieving high end-model accuracy: (1) Randomly initialize a wide, overparameterized neural network. (2) Train the network for a few epochs (often 10-20% of the total number of epochs). (3) Use PCA to find the low-dimensional spaces of network activations and project the weights of the network onto these subspaces to obtain an equivalent PCN. (4) Continue training the smaller PCN until convergence. We describe the PCN transformation for a variety of neural networks, including dense neural networks (DNNs), convolutional neural networks (CNNs), and residual neural networks (ResNets).

We empirically validate training PCNs on CIFAR-10 (Krizhevsky et al., 2009) and ImageNet (Russakovsky et al., 2015) and compare against training the corresponding overparameterized model. For both ResNet (He et al.,

[1]Department of Computer Science, University of Wisconsin-Madison, Madison, Wisconsin, USA [2]Department of Computer Science, ETH Zurich, Zurich, Switzerland. Correspondence to: Roger Waleffe <waleffe@wisc.edu>.

Published at the 39^{th} International Conference on Machine Learning Workshop on Hardware Aware Efficient Training, Baltimore, Maryland, USA, 2022.

2016) and VGG-style architectures (Simonyan & Zisserman, 2014), we show that PCNs train faster, use less energy, and reach comparable end-model accuracy. Interestingly, we show that PCNs derived from wide ResNet models have less parameters but achieve higher accuracy than deep ResNet architectures. Our wide ResNet-20 PCN outperforms a deep ResNet-110 by 0.58% while training 49% faster per epoch.

2. Principal Component Networks

We present the empirical observation that motivates our work and then introduce Principal Component Networks.

Motivating Observation We use PCA to study the hidden layer activations in neural networks. Given a set of m-dimensional vectors, PCA computes a new basis for the vector space according to the directions of highest variance among the data. We define the *effective dimensionality* m_e of the m-dimensional space as the number of PCA directions with variance greater than a threshold τ. We review PCA in more detail in Section A and describe PCA for m-dimensional images in Section D (images with m channels form the input to CNN layers).

We measure the effective dimensionality of hidden layer activations during training. For this experiment we consider a WideResNet-20 (Table 4) over CIFAR-10. After each training epoch, we perform PCA on the hidden activations between each layer (obtained from a sampled set of training examples) and compute the effective dimensionality. We report results for the hidden layer activations input to the last ResNet block in Figure 1 (similar results hold for all layers). We use a threshold τ of 0.5 (5% of the peak variance after training). We also report train and test accuracy.

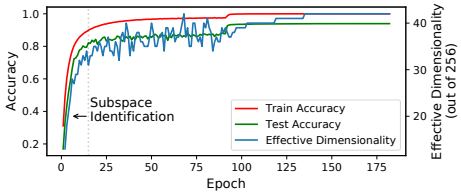


Figure 1: Effective dimensionality (# of high-variance directions) for the hidden layer activations input to the last ResNet block in a WideResNet-20 trained on CIFAR-10.

Figure 1 contains two interesting observations. First, in the early phase of training, up to epoch 15, the effective dimensionality and test accuracy grow quickly. In contrast, the latter phase of training (after epoch 15) is characterized by a slow increase in both accuracy and effective dimensionality. The second key observation is that the hidden layer activations exist in a subspace with dimension significantly smaller than the full space: Even though the layer inputs are 256-dimensional, they only exhibit variance above the trheshold τ in at most 42 directions—a space 6× smaller.

We present additional experiments studying the effective dimensionality of hidden layer activations in Section B. The key takeaway is that hidden layer activations in overparameterized networks do not occupy their full available dimensions. Rather, they exist in low-dimensional subspaces both during training and once test accuracy peaks. Further, the few high-variance directions which contribute most to generalization accuracy are identified early in training. We use these observations as motivation to transform activations into their high-variance PCA bases after only a few epochs.

Based on the above observations, we now introduce PCNs. We describe their application to dense (fully connected) layers and then discuss the end-to-end training procedure for PCNs. We extend to CNNs and ResNets in the appendix.

PCNs for Dense Layers Consider a network where the i^{th} hidden layer computes $\mathbf{h}_n^{i+1} = \sigma(\mathbf{h}_m^i W_{m\times n}^i + \mathbf{b}_n^i)$. Here, σ is the activation function, $\mathbf{h}_m^i \in \mathbb{R}^m$ is the input activation vector, and $\mathbf{h}_n^{i+1} \in \mathbb{R}^n$ is the output activation vector. $W_{m\times n}^i$ and $\mathbf{b}_n^i$ are the layer weights and bias vector respectively. We use superscripts to denote layer index and subscripts to denote dimension. Our goal is to transform W^i and $\mathbf{b}^i$, for each layer i, by considering the high-variance PCA basis of the input activation space.

To transform a layer based on the input activation space (called the input-based transformation), we compute PCA on a batch of input vectors $\mathbf{h}^i$ (Equation 3). Through this calculation, we obtain a mean vector $\boldsymbol{\mu}_m^i$, a vector $\mathbf{e}_m^i$ of variances, and a matrix $V_{m\times m}^i$ whose columns are the principal components (Section A). After finding the PCA basis, we can rewrite any input vector $\mathbf{h}^i$ using these coordinates. If we do so, however, we must also rewrite the layer weight matrix and bias vector. Transformations of $\mathbf{h}^i$, W^i, and $\mathbf{b}^i$ into the PCA basis are given in Equation 1. We denote variables represented using PCA coordinates with tilde.

$$\tilde{\mathbf{h}}_m^i = (\mathbf{h}_m^i - \boldsymbol{\mu}_m^i)V_{m\times m}^i \tag{1}$$
$$\tilde{W}^i_{m\times n} = (V^i_{m\times m})^T W^i_{m\times n} \qquad \tilde{\mathbf{b}}_n^i = \mathbf{b}_n^i + \boldsymbol{\mu}_m^i W_{m\times n}^i$$

By plugging in the definitions, we have that $\sigma(\tilde{\mathbf{h}}^i\tilde{W}^i + \tilde{\mathbf{b}}^i)$ is equivalent to the original $\sigma(\mathbf{h}^i W^i + \mathbf{b}^i)$. Changing basis does not change the output.

Instead of performing an identity transformation by using the full PCA basis, we can approximate the original hidden layer by using only the high-variance subspace. We define this subspace to contain m_e dimensions, which corresponds to the effective dimensionality of the PCA space under some threshold τ. Since the columns of V^i are sorted according to decreasing variance, the first m_e columns contain the PCA directions which describe the subspace. We denote

the matrix V^i truncated after m_e columns by the matrix $U^i_{m\times m_e}$. In Equation 2 we rewrite $\mathbf{h}^i$, W^i, and $\mathbf{b}^i$ in the high-variance subspace of the input activations.

$$\tilde{\mathbf{h}}^i_{m_e} = (\mathbf{h}^i_m - \boldsymbol{\mu}^i_m)U^i_{m\times m_e} \tag{2}$$
$$\tilde{W}^i_{m_e\times n} = (U^i_{m\times m_e})^T W^i_{m\times n} \qquad \tilde{\mathbf{b}}^i_n = \mathbf{b}^i_n + \boldsymbol{\mu}^i_m W^i_{m\times n}$$

The hidden layer $\sigma(\tilde{\mathbf{h}}^i\tilde{W}^i + \tilde{\mathbf{b}}^i)$ is no longer identical to the original $\sigma(\mathbf{h}^i W^i + \mathbf{b}^i)$ but it contains fewer weights; The size of the weight matrix reduces from $m \times n$ to $m_e \times n$. We empirically find that often $m_e \ll m$. Despite approximating the original hidden layer, we have not changed the dimension of the output $\mathbf{h}^{i+1}$. This means we do not need to modify layer $i+1$.

This approximation introduces limited error to the layer output. Errors arise due to dropping the dot product between the last $m-m_e$ elements of $\tilde{\mathbf{h}}^i_m$ and the last $m-m_e$ rows of $\tilde{W}^i_{m\times n}$. However, since $\tilde{\mathbf{h}}^i_m$ is represented using the PCA basis, the dropped coordinates have mean zero and variance less than the threshold τ. For small τ, the approximation ignores dot products between vectors with L_2 norm close to zero and the final $m - m_e$ rows of $\tilde{W}^i_{m\times n}$.

Training PCNs Given the above layer transformation, we present the training procedure for PCNs. We assume as input an overparameterized network architecture N with layers L, a set of layers $I \subseteq L$ to transform via the input-based transformation, a number of epochs K to train before applying the transformation, a number of epochs T to train after transformation, and a dictionary C containing the thresholds/number of dimensions to use when transforming each layer in I. Picking the optimal value for K is a challenging problem. Heuristically, we train N so long as the validation accuracy is increasing at a high rate.

We now describe the steps of the training procedure:

Step 1 Train N for K epochs. This step allows us to retain the benefits of the many random initializations present in overparameterized networks and thus achieve high accuracy.

Step 2 Use PCA to calculate $\mathbf{e}^i$, V^i, and $\boldsymbol{\mu}^i$ for each layer $i \in I$; truncate V^i into U^i using the variances $\mathbf{e}^i$ and $C[i]$, the compression configuration for layer i. Then, transform each layer $i \in I$ into its PCN version. This step has negligible runtime compared to a single training epoch (Section E).

Step 3 Train the generated PCN for T epochs. We do not update the U matrices. The trainable parameters are only the layer weight matrices and bias vectors. Training the PCN for the remaining epochs allows for more efficient training compared to the overparameterized model.

Table 1: Summary of results.

Dataset	Network	# Params	Acc.	PCN Params	PCN Acc.
CIFAR-10	Conv4	2.43M	75.16	101k	**77.25**
CIFAR-10	WideResNet-20	4.33M	93.60	1.32M	93.52
ImageNet	VGG-19	143M	70.99	27.2M	70.27
ImageNet	WideResNet-50	98.0M	77.52	62.3M	**77.64**

3. Experiments

We empirically validate the performance of PCNs against their overparameterized counterparts. We consider several architectures including VGG-style CNNs (Simonyan & Zisserman, 2014) and ResNets (He et al., 2016) over CIFAR-10 (Krizhevsky et al., 2009) and ImageNet (Russakovsky et al., 2015). To measure performance we consider standard test accuracy metrics. Finally, we evaluate the time and energy improvements that PCNs introduce. All experiment details and hyperparameters are discussed in Section F.

Key Takeaways We summarize end-to-end results in Table 1. Although PCNs perform compression early in training, they consistently reduce the number of weights in overparameterized networks yet sacrifice little, if any, accuracy. Our method extends from the comparatively simple dataset CIFAR-10 to the challenging ImageNet dataset. For the former, we achieve comparable compression and accuracy improvements to winning lottery tickets found in (Frankle & Carbin, 2019), but require no iterative train-prune procedure. For the latter, in one case we observe a reduction in parameters by $5.28\times$, while in the other case we observe improvements to end-model accuracy.

PCNs and Wide Networks We evaluate the hypothesis that wide networks improve generalization and that PCNs can help reduce the number of parameters early in training without sacrificing end-model accuracy. On both CIFAR-10 and ImageNet, we trained wide ResNet architectures—networks identical to the original ResNets (He et al., 2016), but with more filters at each layer. We compare these networks and their corresponding PCNs with conventional deep ResNet models. Results are shown in Tables 2 and 3. For CIFAR-10 we show three different PCNs with increasing amount of compression and include three-run max/min for WideResNet-20 and WideResNet-20-PCN0.

We find that wide networks improve accuracy more than deep networks. We also find that PCNs can reduce the number of parameters in wide models while retaining their test accuracy. On CIFAR-10, both PCN0 and PCN1 exceed the accuracy of a deep ResNet-110 with less parameters. On ImageNet, our PCN outperforms a ResNet-152 with a comparable number of trainable weights. Additionally, Tables 2 and 3 show that wide networks can be converted into PCNs early in training. The transformation occurs at

Table 2: Principal Component Networks trained on CIFAR-10. PCN transformations are performed after epoch 15.

Network Name	Total Parameters	Trainable Parameters	Test Acc. (Epoch 182)
ResNet-20	274,362	272,762	91.03
ResNet-110	1,739,322	1,731,002	92.94
WideResNet-20	4,338,378	4,331,978	**93.60** (+0.07, -0.08)
WideResNet-20-PCN0	1,443,018	1,323,850	**93.52** (+0.10, -0.07)
WideResNet-20-PCN1	1,223,114	1,094,154	93.23
WideResNet-20-PCN2	541,834	473,802	92.50

Table 3: Principal Component Networks trained on ImageNet. PCN transformations are performed after epoch 16.

Network Name	Total Parameters	Trainable Parameters	Center Crop Top-1 (Top-5) Val Acc. at Epoch 90
ResNet-50	25,610,152	25,557,032	75.60 (92.78)
ResNet-152[1]	60,344,232	60,192,808	77.00 (93.3)
WideResNet-50	98,110,312	98,004,072	**77.52** (93.92)
WideResNet-50-PCN	71,936,872	62,375,016	**77.64** (93.85)

epoch 15 out of 182 on CIFAR-10 and epoch 16 out of 90 on ImageNet. We provide additional experiments in Section G.

Performance: Training Time and Energy We highlight that by reducing parameter counts early in training, PCNs have the potential to significantly lower the time and energy required to learn high-accuracy models. Before discussing results we note the following: Our current implementation uses standard GPU kernels for existing neural network layers. For best performance a new GPU kernel is required which fuses the PCN basis transformation at each layer (multiplication with U) together with the layer itself.

Without an optimized GPU kernel, we already see performance improvements for training. For example, end-to-end training of WideResNet-20-PCN1 on CIFAR-10 uses 15% less total energy compared to the full WideResNet-20 and 25% less energy than the ResNet-110 while training 12% and 45% faster respectively. We expect fused multiplication with U will significantly improve the energy consumption and training time benefits of PCNs. The PCN transformation overhead is negligible and does not prevent the training time reductions which arise from using the smaller network for the bulk of the training epochs (Section E).

Discussion The observation that overparameterized networks contain subnetworks that can be trained in isolation to comparable accuracy has lead to a lot of work to identify such networks early in training. Initial approaches found sparse subnetworks (Frankle & Carbin, 2019; Frankle et al., 2020). While these approaches generally lead to the highest compression ratios, methods for finding sparse subnetworks are expensive and the resulting sparse architectures are not well suited for modern GPU hardware. Thus the practical potential of such methods is currently limited. More recent work identifies subnetworks using structured pruning techniques early in training (e.g., EB Train (You et al., 2019)). While these techniques efficiently produce dense architectures (thus leading to end-to-end training improvements), they generally focus on pruning using localized measurements (e.g., individual channels) leading to larger accuracy drops. In contrast, PCNs use holistic layer-wise measurements for pruning decisions. We find that for ResNet-50 on ImageNet, PCNs outperform EB Train: PCNs (transformed after 20% of training) with 30% trainable parameter reduction yield an accuracy drop of only -0.45 versus the drop of -1.73 for EB. For 50% reduction, we get -1.61 for PCNs and -2.24 for EB. These observations match those reported by Wang et al. (2021) where a holistic low-rank factorization of the network weights is performed early in training. Moreover, together PCNs and Wang et al. (2021) show that *both* hidden layer activations (PCNs) and weights (Wang et al., 2021) exhibit low-rank structures. Combining all existing methods and observations for compression early in training may be of interest for future work and lead to improved methods for reducing end-to-end training costs.

[1]Accuracy reported from https://github.com/kaiminghe/deep-residual-networks.

4. Related Work

A large body of work has focused on reducing the number of parameters and associated computational cost of overparameterized neural networks. To benefit model inference, pruning (LeCun et al., 1990; Hassibi & Stork, 1993; Han et al., 2015; Li et al., 2017; Hu et al., 2016; Srinivas & Babu, 2015a; Yang et al., 2017; He et al., 2017; Luo et al., 2017), distillation (Ba & Caruana, 2014; Hinton et al., 2015), quantization (Gong et al., 2014), and weight decomposition (Jaderberg et al., 2014; Lebedev et al., 2014; Kim et al., 2015; Denton et al., 2014; Zhang et al., 2015; Yaguchi et al., 2019) are performed after training. Specifically engineered networks (Iandola et al., 2016; Howard et al., 2017), binary weights (Zhuang et al., 2019), and low-rank (Denil et al., 2013) architectures help lessen model size from the start, while other works aim to sparsify networks during the learning process (Srinivas et al., 2017; Louizos et al., 2017; Srinivas & Babu, 2016; Molchanov et al., 2017; Neklyudov et al., 2017; Srinivas & Babu, 2015b; Alvarez & Salzmann, 2017; Xu et al., 2018). Additional works try to identify subnetworks contained in large models (Frankle & Carbin, 2019; Ramanujan et al., 2019). Garg et al. (2020) find that CNN hidden layer activations exist in low dimensional spaces after training while Chen et al. (2021) observe the same for NLP models. We find this observation holds *during* training and use this fact to reduce training costs.

5. Conclusions

In this paper, we show that early in the training process, overparameterized networks can be transformed into smaller versions which train to similar accuracy as the full model. We focus on hidden layer activations rather than model weights, and observe that they exist primarily in subspaces an order of magnitude smaller than the actual network width. This motivates us to introduce Principal Component Networks, architectures which represent layer weights using the high-variance subspaces of their input and output activations. We experimentally validate the ability of PCNs to reduce parameter counts early in training and to reach similar end-model accuracy as overparameterized models.

References

Alvarez, J. M. and Salzmann, M. Compression-aware training of deep networks. In *Advances in Neural Information Processing Systems*, pp. 856–867, 2017.

Ba, J. and Caruana, R. Do deep nets really need to be deep? In *Advances in neural information processing systems*, pp. 2654–2662, 2014.

Chen, P., Yu, H.-F., Dhillon, I., and Hsieh, C.-J. Drone: Data-aware low-rank compression for large nlp models. *Advances in neural information processing systems*, 34: 29321–29334, 2021.

Denil, M., Shakibi, B., Dinh, L., Ranzato, M. A., and de Freitas, N. Predicting parameters in deep learning. In *Advances in Neural Information Processing Systems 26*, pp. 2148–2156, 2013.

Denton, E. L., Zaremba, W., Bruna, J., LeCun, Y., and Fergus, R. Exploiting linear structure within convolutional networks for efficient evaluation. In *Advances in neural information processing systems*, pp. 1269–1277, 2014.

Frankle, J. and Carbin, M. The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *International Conference on Learning Representations*, 2019.

Frankle, J., Dziugaite, G. K., Roy, D., and Carbin, M. Linear mode connectivity and the lottery ticket hypothesis. In *International Conference on Machine Learning*, pp. 3259–3269. PMLR, 2020.

Garg, I., Panda, P., and Roy, K. A low effort approach to structured cnn design using pca. In *IEEE Access*, volume 8, pp. 1347–1360, 2020.

Glorot, X. and Bengio, Y. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pp. 249–256, 2010.

Gong, Y., Liu, L., Yang, M., and Bourdev, L. Compressing deep convolutional networks using vector quantization. *arXiv preprint arXiv:1412.6115*, 2014.

Gunasekar, S., Woodworth, B. E., Bhojanapalli, S., Neyshabur, B., and Srebro, N. Implicit regularization in matrix factorization. In *Advances in Neural Information Processing Systems*, pp. 6151–6159, 2017.

Han, S., Pool, J., Tran, J., and Dally, W. J. Learning both weights and connections for efficient neural networks. In *Advances in Neural Information Processing Systems 28*, pp. 1135–1143, 2015.

Hassibi, B. and Stork, D. G. Second order derivatives for network pruning: Optimal brain surgeon. In *Advances in Neural Information Processing Systems 5*, pp. 164–171, 1993.

He, K., Zhang, X., Ren, S., and Sun, J. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pp. 1026–1034, 2015.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

He, T., Zhang, Z., Zhang, H., Zhang, Z., Xie, J., and Li, M. Bag of tricks for image classification with convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 558–567, 2019.

He, Y., Zhang, X., and Sun, J. Channel pruning for accelerating very deep neural networks. In *Proceedings of the IEEE International Conference on Computer Vision*, pp. 1389–1397, 2017.

Hinton, G., Vinyals, O., and Dean, J. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., and Adam, H. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

Hu, H., Peng, R., Tai, Y.-W., and Tang, C.-K. Network trimming: A data-driven neuron pruning approach towards efficient deep architectures. *arXiv preprint arXiv:1607.03250*, 2016.

Iandola, F. N., Han, S., Moskewicz, M. W., Ashraf, K., Dally, W. J., and Keutzer, K. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016.

Ioffe, S. and Szegedy, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

Jaderberg, M., Vedaldi, A., and Zisserman, A. Speeding up convolutional neural networks with low rank expansions. *arXiv preprint arXiv:1405.3866*, 2014.

Kim, Y.-D., Park, E., Yoo, S., Choi, T., Yang, L., and Shin, D. Compression of deep convolutional neural networks for fast and low power mobile applications. *arXiv preprint arXiv:1511.06530*, 2015.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

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

Lebedev, V., Ganin, Y., Rakhuba, M., Oseledets, I., and Lempitsky, V. Speeding-up convolutional neural networks using fine-tuned cp-decomposition. *arXiv preprint arXiv:1412.6553*, 2014.

LeCun, Y., Denker, J. S., and Solla, S. A. Optimal brain damage. In *Advances in Neural Information Processing Systems 2*, pp. 598–605, 1990.

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.

Li, H., Kadav, A., Durdanovic, I., Samet, H., and Graf, H. P. Pruning filters for efficient convnets. In *International Conference on Learning Representations*, 2017.

Li, Z., Wallace, E., Shen, S., Lin, K., Keutzer, K., Klein, D., and Gonzalez, J. E. Train large, then compress: Rethinking model size for efficient training and inference of transformers. *arXiv preprint arXiv:2002.11794*, 2020.

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

Louizos, C., Welling, M., and Kingma, D. P. Learning sparse neural networks through l_0 regularization. *arXiv preprint arXiv:1712.01312*, 2017.

Luo, J.-H., Wu, J., and Lin, W. Thinet: A filter level pruning method for deep neural network compression. In *Proceedings of the IEEE international conference on computer vision*, pp. 5058–5066, 2017.

Molchanov, D., Ashukha, A., and Vetrov, D. Variational dropout sparsifies deep neural networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pp. 2498–2507. JMLR. org, 2017.

Neklyudov, K., Molchanov, D., Ashukha, A., and Vetrov, D. P. Structured bayesian pruning via log-normal multiplicative noise. In *Advances in Neural Information Processing Systems*, pp. 6775–6784, 2017.

Park, D. S., Sohl-Dickstein, J., Le, Q. V., and Smith, S. L. The effect of network width on stochastic gradient descent and generalization: an empirical study. *arXiv preprint arXiv:1905.03776*, 2019.

Ramanujan, V., Wortsman, M., Kembhavi, A., Farhadi, A., and Rastegari, M. What's hidden in a randomly weighted neural network? *arXiv preprint arXiv:1911.13299*, 2019.

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., et al. Imagenet large scale visual recognition challenge. *International journal of computer vision*, 115(3): 211–252, 2015.

Simonyan, K. and Zisserman, A. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

Srinivas, S. and Babu, R. V. Data-free parameter pruning for deep neural networks. *arXiv preprint arXiv:1507.06149*, 2015a.

Srinivas, S. and Babu, R. V. Learning neural network architectures using backpropagation. *arXiv preprint arXiv:1511.05497*, 2015b.

Srinivas, S. and Babu, R. V. Generalized dropout. *arXiv preprint arXiv:1611.06791*, 2016.

Srinivas, S., Subramanya, A., and Venkatesh Babu, R. Training sparse neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pp. 138–145, 2017.

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.

Wang, H., Agarwal, S., and Papailiopoulos, D. Pufferfish: Communication-efficient models at no extra cost. *Proceedings of Machine Learning and Systems*, 3:365–386, 2021.

Xu, Y., Li, Y., Zhang, S., Wen, W., Wang, B., Qi, Y., Chen, Y., Lin, W., and Xiong, H. Trained rank pruning for efficient deep neural networks. *arXiv preprint arXiv:1812.02402*, 2018.

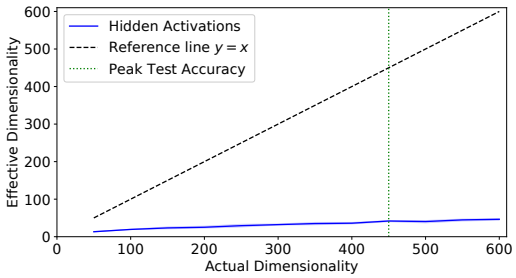
Yaguchi, A., Suzuki, T., Nitta, S., Sakata, Y., and Tanizawa, A. Scalable deep neural networks via low-rank matrix factorization. *arXiv preprint arXiv:1910.13141*, 2019.

Yang, T.-J., Chen, Y.-H., and Sze, V. Designing energy-efficient convolutional neural networks using energy-aware pruning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 5687–5695, 2017.

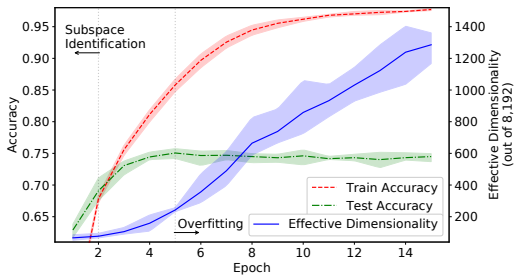
You, H., Li, C., Xu, P., Fu, Y., Wang, Y., Chen, X., Baraniuk, R. G., Wang, Z., and Lin, Y. Drawing early-bird tickets: Towards more efficient training of deep networks. *arXiv preprint arXiv:1909.11957*, 2019.

Zhang, X., Zou, J., Ming, X., He, K., and Sun, J. Efficient and accurate approximations of nonlinear convolutional networks. In *Proceedings of the IEEE Conference on Computer Vision and pattern Recognition*, pp. 1984–1992, 2015.

Zhuang, B., Shen, C., Tan, M., Liu, L., and Reid, I. Structured binary neural networks for accurate image classification and semantic segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 413–422, 2019.



(a) Effective dimensionality vs. width



(b) Effective dimensionality vs. training

Figure 3: Effective dimensionality (# of high-variance activations) for neural networks.

Appendix

A. Principal Component Analysis

Given a set of m-dimensional vectors, PCA computes a new basis for the vector space with the following property: The first basis vector (termed Principal Component) is the direction of highest variance among the data. The second basis vector has highest variance among directions perpendicular to the first, and so on. An example is shown in Figure 2. Two-dimensional vectors in the original $[x_1, x_2]$ basis can also be represented in the $[\tilde{x}_1, \tilde{x}_2]$ PCA basis. We define the *effective dimensionality* m_e of the m-dimensional space as the number of PCA directions with variance greater than a threshold τ. In Figure 2, the spread along $\tilde{x}_2$ does not help differentiate between the two sets of points, as the original two-dimensional vectors exist primarily in a one-dimensional space given by the coordinate $\tilde{x}_1$.

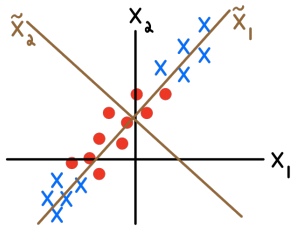


Figure 2: PCA illustration with principal components shown in brown.

We compute the principal components and associated variances using the spectral decomposition of the covariance matrix.We find this method to be significantly faster than using SVD in TensorFlow. Given N data points in $\mathbb{R}^m$ with empirical mean $\boldsymbol{\mu}_m \in \mathbb{R}^m$ we have:

$$\mathbf{e}_m, V_{m\times m} = eigh\left(\frac{1}{N-1}\left(X_{N\times m} - \boldsymbol{\mu}_m\right)^T \left(X_{N\times m} - \boldsymbol{\mu}_m\right)\right). \tag{3}$$

Function $eigh$ returns two quantities: the vector of eigenvalues (variances) $\mathbf{e}_m$, and a matrix $V_{m\times m}$ whose columns contain the eigenvectors (principal components). We assume that both outputs are sorted in descending eigenvalue order. To transform a vector $\mathbf{x}_m$ into the PCA basis, one computes: $\tilde{\mathbf{x}}_m = (\mathbf{x}_m - \boldsymbol{\mu}_m)V_{m\times m}$. Subtracting off the mean vector $\boldsymbol{\mu}_m$ centers the PCA coordinate system, ensuring each principal component has mean zero when average over the N data points.

B. Additional Motivating Experiments

As described in Section 2, we use PCA to study the effective dimensionality (effective width) of hidden layer activations in neural networks. We present two additional experiments that highlight our findings.

Experiment 1 We first analyze the activations of a DNN after learning is complete. We train a DNN with an input layer, a hidden layer with a variable number of nodes, and an output layer with 10 neurons on MNIST (LeCun et al., 1998). Both dense layers use sigmoid activation and we train until convergence of the validation accuracy. We computed PCA on the network activations *after* the hidden layer and *before* the output layer. The dimension of these activations is equal to the number of nodes in the hidden layer and is thus varied by changing the number of hidden neurons. In Figure 3a we plot the number of effective dimensions in the activation space versus the number of full dimensions using a PCA variance threshold of 0.1. We see that that the hidden layer activations exist in a subspace with smaller dimension than the full space: with 450 nodes in the hidden layer the network achieves peak test accuracy of 98%, but rather than occupy 450 dimensions, the hidden layer outputs occupy a space with just over 40 dimensions—a space more than 10× smaller.

Experiment 2 We also study the evolution of hidden layer subspaces during training. In this experiment we do not vary the network architecture, and thus, can train a more realistic network. We consider a CNN over CIFAR-10. We use the Conv4 network (Frankle & Carbin, 2019) which consists of four convolution layers followed by three dense layers. After each training epoch, we perform PCA on the hidden activations which form the input to the first fully connected layer. This activation space contains 8,192 dimensions. We calculate PCA for this layer because it contains more than 86% of the total weights, and thus, affects the end-model accuracy. We *do not* stop training at peak validation accuracy to observe the effective dimensionality of the activation space during overfitting.

The results are shown in Figure 3b. We again use a PCA variance threshold of 0.1. There are three regions of interest in this plot: In the first section, up to epoch two, notice that the effective dimensionality grows slowly, *but* the test accuracy grows to 70%—93% of its peak at 75%. The data variance in this ≈ 50-dimensional subspace is critical for generalization. Contrast this with region three, after the test accuracy peaks at epoch five. Now the network is heavily overfitting, increasing the train accuracy to no avail. Also observe that the network is rapidly creating directions with variance above the 0.1 threshold. All such directions are overfitting to meaningless noise. In between, during epochs 2-5, the effective dimensionality increases from ≈ 50 to ≈ 200. We conjecture that these directions are mainly noise, as the test accuracy grows slowly while the train accuracy rapidly surpasses it. The test accuracy can still increase due to *fine-tuning* weights related to the high-variance directions discovered in the first region.

Takeaway Experiments 1 and 2 show that hidden layer activations do not occupy their full available dimensions. Rather, they exist in low-dimensional subspaces both during training and once test accuracy peaks. Further, Experiment 2 shows that the few high-variance directions which contribute most to generalization accuracy are identified early in training. We use these observations as motivation to transform activations into their high-variance PCA bases after only a few epochs.

C. Transformation Based on Output Activations

In Section 2 we described how to transform a dense neural network layer based on the PCA basis of the input activations. We can also reduce the parameters in layer i using the PCA basis of the subsequent layer $i+1$ (which we refer to as the output-based transformation). We use the output-based transformation only as an optimization if the next layer's input is an order of magnitude compressed after the input-based transform. Note that the output transformation at layer i requires that layer $i+1$ perform the input transformation.

Assume that layer i is transformed per the input-based transformation (Equation 2). Layer i then computes an approximate $\hat{\mathbf{h}}^{i+1}$ output vector: $\hat{\mathbf{h}}^{i+1}_n = \sigma(\tilde{\mathbf{h}}^{i}_{m_e} \tilde{W}^{i}_{m_e \times n} + \tilde{\mathbf{b}}^{i}_n)$. We can further compress $\tilde{W}^{i}$ and $\tilde{\mathbf{b}}^{i}$ using information about the output activation space.

Since neural networks compose layers, the next layer $i+1$ will consider $\hat{\mathbf{h}}^{i+1}$ as its *input*. If we consider the input-based transformation for layer $i+1$, we have that $\tilde{\mathbf{h}}^{i+1}_{n_e} = (\hat{\mathbf{h}}^{i+1}_n - \boldsymbol{\mu}^{i+1}_n) U^{i+1}_{n \times n_e}$. We next rewrite $\tilde{\mathbf{h}}^{i+1}$ as a function of the approximate weight matrix $\tilde{W}^{i}$ of the previous layer. We have for the j^{th} component of $\tilde{\mathbf{h}}^{i+1}$:

$$\begin{aligned}
\tilde{\mathbf{h}}^{i+1}[j] &= \sum_{l=1}^{n} \left(\hat{\mathbf{h}}^{i+1}[l] - \boldsymbol{\mu}^{i+1}[l] \right) U^{i+1}[l, j] \\
&= \sum_{l=1}^{n} \left(\sigma \left(\sum_{k=1}^{m_e} \tilde{\mathbf{h}}^{i}[k] \tilde{W}^{i}[k, l] + \tilde{\mathbf{b}}^{i}[l] \right) - \boldsymbol{\mu}^{i+1}[l] \right) U^{i+1}[l, j]. \qquad (4)
\end{aligned}$$

The elements of $\tilde{\mathbf{h}}^{i+1}$ are linear combinations of the centered elements in $\hat{\mathbf{h}}^{i+1}$ weighted by the columns of U^{i+1}. For all elements j, the influence of the l^{th} output of layer i, denoted $\hat{\mathbf{h}}^{i+1}[l]$, on $\tilde{\mathbf{h}}^{i+1}[j]$ is determined by $U^{i+1}[l, j]$. If most entries in the the l^{th} row of U^{i+1} have large values, output $\hat{\mathbf{h}}^{i+1}[l]$ influences many entries in $\tilde{\mathbf{h}}^{i+1}$. On the other hand, if every entry in the l^{th} row of U^{i+1} is small, the l^{th} output of layer i does not influence *any* entry in $\tilde{\mathbf{h}}^{i+1}$. We use the L_1 norm of row l in U^{i+1} to determine how important output $\hat{\mathbf{h}}^{i+1}[l]$ is for calculating $\tilde{\mathbf{h}}^{i+1}$.

We use the above influence measurement to define the output-based transformation of layer i. Using the L_1 norm criterion described above, we find the subset $S \subseteq [1 \dots n]$ of the indices in $\hat{\mathbf{h}}^{i+1}$ with the highest row-wise L_1 norm in U^{i+1}. The

size of S is configurable and can be fixed by the user or determined using an L_1-norm threshold. Given S, Equation 4 becomes:

$$\tilde{\mathbf{h}}^{i+1}[j] = \sum_{l \in S} \left(\sigma \left(\sum_{k=1}^{m_e} \tilde{\mathbf{h}}^{i}[k] \tilde{W}^{i}[k,l] + \tilde{\mathbf{b}}^{i}[l] \right) - \boldsymbol{\mu}^{i+1}[l] \right) U^{i+1}[l,j]. \tag{5}$$

The columns of $\tilde{W}^{i}$, entries of $\tilde{\mathbf{b}}^{i}$, entries of $\boldsymbol{\mu}^{i+1}$, and rows of U^{i+1} with indices in S^C can be removed from the network. The rows in W^{i+1} with indices in S^C must also be removed so dimensions match when multiplying by $(U^{i+1})^T$ in Equation 2.

The output-based transformation and input-based transformation can be applied to layer i in either order.

C.1. Transformation Order

We discuss technical details regarding the order of the input- and output-based transformations.

1. We calculate the PCA basis (U) for each layer before performing any transformations. That is, we use the original network's hidden activations rather than first transforming layer i and then using its approximate outputs to compute PCA at layer $i+1$. This choice allows us to calculate PCA at each layer using a forward pass through the original network.

2. Given U^{i+1}, the order of the input and output-based transformation at layer i does not matter. Pruning columns of W^i and entries of $\mathbf{b}^i$ then transforming to $\tilde{W}^{i}$ and $\tilde{\mathbf{b}}^{i}$ is the same as first transforming to $\tilde{W}^{i}$ and $\tilde{\mathbf{b}}^{i}$ and then pruning these transformed variables.

3. The order of the output-based transformation at layer i and the input-based transformation at layer $i+1$ does matter. Suppose we already performed the input-based transformation at layer i, and we now plan to perform the output-based transformation at layer i. As described above, to do so we 1) find the subset $S \subseteq [1 \dots n]$ with highest row-wise L_1 norm in U^{i+1} and 2) prune columns of $\tilde{W}^{i}$, entries of $\tilde{\mathbf{b}}^{i}$, entries of $\boldsymbol{\mu}^{i+1}$, and rows of U^{i+1} with indices in S^C. The question is then: do we use the truncated $\boldsymbol{\mu}^{i+1}$ and U^{i+1} to calculate $\tilde{W}^{i+1}$ and $\tilde{b}^{i+1}$ in the input-based transformation at layer $i+1$? Or do we first calculate $\tilde{W}^{i+1}$ and $\tilde{b}^{i+1}$ and then truncate $\boldsymbol{\mu}^{i+1}$ and U^{i+1}? In the first case, we must also remove rows in W^{i+1} with indices in S^C so that dimensions match when performing the transformation.

Performing the input-based transformation at layer $i+1$ before truncating $\boldsymbol{\mu}^{i+1}$ and U^{i+1} according to the output-based transformation at layer i means that $\tilde{W}^{i+1}$ and $\tilde{b}^{i+1}$ are written in the unmodified high-variance PCA subspace. On the other hand, outputs of layer i which are no longer part of the network contribute to these directions. Currently, we truncate first, and then perform the input-based transformation at layer $i+1$. In the future, we plan to study the performance of each ordering.

Summary Given a compression configuration $C[i]$ for layer i, we perform the transformations as follows:

$$W^i, \mathbf{b}^i, \boldsymbol{\mu}^{i+1}, U^{i+1}, W^{i+1} \leftarrow \texttt{OutputTransformation}(W^i, \mathbf{b}^i, \boldsymbol{\mu}^{i+1}, U^{i+1}, W^{i+1}, C[i])$$
$$\tilde{W}^{i}, \tilde{\mathbf{b}}^{i} \leftarrow \texttt{InputTransformation}(W^i, \mathbf{b}^i, \boldsymbol{\mu}^i, U^i).$$

D. PCNs for Convolutional Layers and ResNets

We extend the PCN transformations to CNNs and ResNets. We provide a high-level description and then present the details.

Convolutional Layers To transform convolutional layers, the intuition is the same as for dense layers: Rather than representing hidden layer activations using their default filters, we would like to find a new basis of "principal filters" which exhibit sorted high to low variance.

Convolutional layers operate over matrices and tensors. We denote a convolutional layer $H^{i+1}_{h' \times w' \times n} = \sigma(H^{i}_{h \times w \times m} * W^{i}_{k_1 \times k_2 \times m \times n} + \mathbf{b}^{i}_{n})$, where h and w describe the size of the input data, m is the number of input filters, $k_1 \times k_2$ is the kernel size, and n is the number of output filters. PCA does not immediately extend to multi-dimensional inputs such as images. As a surrogate, for a batch of N $H_{h \times w \times m}$ matrices, we consider all $h \times w$ depth vectors of dimension m in each image to be examples of points in an m dimensional space. View each axis in this space to be a filter and the point cloud a distribution of how likely each filter is to exhibit a certain pixel value, regardless of $[h, w]$ location. We can then run standard

PCA on the flattened set of images $H_{N' \times m}$ with $N' = N \times h \times w$. The resulting $V_{m \times m}$ tells us how to combine *every* depth vector from an original image into a depth vector in the "principal image".

Once we calculate PCA for the input and output activation spaces, i.e., we calculate V^i and V^{i+1}, we can perform the input- and output-based transformation to convolutional layer i. Doing so requires extending Equation 2 and Equation 5 to handle tensors instead of vectors and matrices.

ResNets While ResNets consist of convolution layers, they require special care due to the inclusion of residual connections. Since the input transformation (for both dense and convolution layers) modifies only layer i and not subsequent layers, it can be applied immediately to ResNet architectures. The output transformation, however, needs to be modified for some layers. The output of the final layer in a residual block is added to the output of all other residual blocks in a residual stage. Thus, we introduce the added constraint that all layers whose outputs are added together need to perform the output transformation in the same way.

Details More specifically, consider a convolutional layer:

$$H^{i+1}_{h' \times w' \times n} = \sigma(H^i_{h \times w \times m} * W^i_{k_1 \times k_2 \times m \times n} + \mathbf{b}^i_n)$$

where h and w describe the size of the input data, m is the number of input filters, $k_1 \times k_2$ is the kernel size, and n is the number of output filters.

PCA for Images Given a batch of N input images to layer i, compute PCA as follows:

1. $H^i_{N' \times m} = \texttt{Flatten}(H^i_{N \times h \times w \times m})$
2. $\boldsymbol{\mu}^i_m, \mathbf{e}^i_m, V^i_{m \times m} = \texttt{PCA}(H^i_{N' \times m})$

Input-Based Transformation Transform an input image to the PCA version as so:

1. $H^i_{(h \times w) \times m} = \texttt{Flatten}(H^i_{h \times w \times m})$
2. $\tilde{H}^i_{(h \times w) \times m_e} = (H^i_{(h \times w) \times m} - \boldsymbol{\mu}^i_m) U^i_{m \times m_e}$ (subtraction applies to all rows)
3. $\tilde{H}^i_{h \times w \times m_e} = \texttt{Reshape}(\tilde{H}^i_{(h \times w) \times m_e})$

You can transform a batch of images all at once by flattening $H^i_{N \times h \times w \times m}$. Note that if the original layer performed zero padding, then the padding should be included in the transformation. Either $H^i_{h \times w \times m}$ should be zero padded and then transformed, in which case no additional padding is required when performing convolution, or channel j of $\tilde{H}^i_{h \times w \times m_e}$ should be padded with $(-\boldsymbol{\mu}^i_m U^i_{m \times m_e})[j]$ during convolution.

Transform the weights W^i by:

1. $W^i_{n \times m \times (k_1 \times k_2)} = \texttt{Reshape}(W^i_{k_1 \times k_2 \times m \times n})$
2. $\tilde{W}^i_{n \times m_e \times (k_1 \times k_2)} = \texttt{BatchMatrixMultiply}((U^i_{m \times m_e})^T, W^i_{n \times m \times (k_1 \times k_2)})$
3. $\tilde{W}^i_{k_1 \times k_2 \times m_e \times n} = \texttt{Reshape}(\tilde{W}^i_{n \times m_e \times (k_1 \times k_2)})$

Transform the bias $\mathbf{b}^i$ by:

1. $W^i_{n \times m \times (k_1 \times k_2)} = \texttt{Reshape}(W^i_{k_1 \times k_2 \times m \times n})$
2. $T_{n \times (k_1 \times k_2)} = \texttt{BatchMatrixMultiply}(\boldsymbol{\mu}^i_m, W^i_{n \times m \times (k_1 \times k_2)})$
3. $\tilde{\mathbf{b}}^i_n = \mathbf{b}^i_n + \texttt{ReduceSum}_{axis=1}(T_{n \times (k_1 \times k_2)})$

The resulting convolutional layer, transformed based on the input hidden activations, computes an approximate output image:

$$\hat{H}^{i+1}_{h' \times w' \times n} = \sigma(\tilde{H}^{i}_{h \times w \times m_e} * \tilde{W}^{i}_{k_1 \times k_2 \times m_e \times n} + \tilde{\mathbf{b}}^{i}_{n}).$$

Instead of containing m filters, the input images contain only m_e filters of high variance. Only the weights which act on these filters remain in $\tilde{W}^{i}$.

Output-Based Transformation Just as in Section C for dense layers, we use the L_1 norm of row l in U^{i+1} to determine the importance of the l^{th} output filter of layer i. Given the subset $S \subseteq [1 \dots n]$ of indices with highest row-wise L_1 norm in U^{i+1}, we can prune filters from layer i with indices in S^C. As before, we must also update $\boldsymbol{\mu}^{i+1}$, U^{i+1}, and W^{i+1}. We include the dimensions below for clarity on what is pruned in the higher dimensional tensors (assuming we already did the input transformation at layer i):

$$\tilde{W}^{i}_{k_1 \times k_2 \times m_e \times |S|}, \mathbf{b}^{i}_{|S|}, \boldsymbol{\mu}^{i+1}_{|S|}, U^{i+1}_{|S| \times n_e}, W^{i+1}_{k'_1 \times k'_2 \times |S| \times o}.$$

When performing the output-based transformation to a convolutional layer i which is followed by a dense layer in the original network, additional steps need to be taken. In this case, the output of layer i, $H^{i+1}_{h' \times w' \times n}$, is flattened to $\mathbf{h}^{i+1}_{(h' \times w' \times n)}$. Thus, U^{i+1} has $(h' \times w' \times n)$ rows instead of n rows. To determine the importance of the l^{th} output filter, we add together the L_1 norm of each of the corresponding $(h' \times w')$ rows in U^{i+1}. Pruning a single filter then requires removing all corresponding $(h' \times w')$ entries in $\boldsymbol{\mu}^{i+1}$, rows in U^{i+1}, and rows in W^{i+1} (which is just a matrix again).

Transformations for ResNets The input-based convolutional transformation can be directly applied to all layers in a given ResNet. Here, we discuss the additional constraint we impose to perform the output-based transformation.

Consider the partial ResNet diagram shown in Figure 4. We show a stage with three blocks followed by the first block of the next stage. Convolutional layers are depicted using the tuple (filters, kernel size/strides). Typically, $S = 2$ to downsample the image at the beginning of each stage. We omit batch normalization (Ioffe & Szegedy, 2015) and activation layers for brevity. All layers $[0, 9]$ can perform the input-based transformation without affecting any other layers. Note that Layers 0 and 1 share the same input activations and thus $U^0 = U^1$. The same is true for Layers 7 and 8.

We discuss the output-based transformation for layers in the first stage, i.e. layers $[0, 6]$. First note that Layers 1, 3, and 5 can perform the output-based transformation as if they were in a standard convolutional neural network. Since the output filters of Layers 0, 2, 4, and 6 are added together, we enforce the constraint that if these layers are to prune filters, they must prune the same filters.

To decide which filters to prune, we no longer have only one U^{i+1}. Instead, U^3, U^5, and $U^7 = U^8$ all provide information regarding the importance of specific filters. For the l^{th} filter of Layers 0, 2, 4, and 6, we use the average L_1 norm of the l^{th} rows in U^3, U^5, and U^7 as the influence measurement. Given this criteria, we can perform the output-based transformation for the even layers in the first stage. Just as $\boldsymbol{\mu}^{i+1}$, U^{i+1}, and W^{i+1} are updated in the standard output-based transformation, here we must update $\boldsymbol{\mu}$, U, and W for layers 3, 5, 7, and 8.

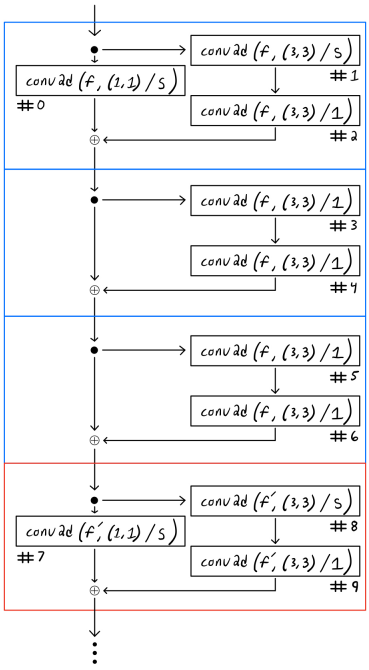


Figure 4: ResNet illustration.

E. Transformation Overhead

We discuss the overhead required to convert a network into its corresponding PCN version. In general, the transformation time is negligible compared to the overall training procedure. The primary bottleneck is computing hidden layer activations, which require partial forward passes through the original network. That said, however, only a small fraction of the training data is needed for sufficient PCA statistics. Thus, PCN conversion typically requires much less compute than one training epoch. On CIFAR-10, we transform WideResNet-20 architectures in $\approx$ 3s while a single training epoch takes $\approx$ 17s. On ImageNet, PCN conversion can be performed an order of magnitude faster than training one epoch. A single GPU transforms a WideResNet-50 in half an hour yet it takes the same time for eight GPUs to execute one epoch.

Table 4: Network architectures for CIFAR-10.

Name	Conv4 (Frankle & Carbin, 2019)	ResNet-20 (He et al., 2016)	ResNet-110 (He et al., 2016)	WideResNet-20
Conv Layers	64, 64, M 128, 128, M	16 3x[16, 16] 3x[32, 32] 3x[64, 64]	16 18x[16, 16] 18x[32, 32] 18x[64, 64]	64 3x[64, 64] 3x[128, 128] 3x[256, 256]
FC Layers	256, 256, 10	A, 10	A, 10	A, 10

F. Experiment Setup

All experiments were executed over TensorFlow. We implement no special optimizations and use only the high level TensorFlow Keras API. All models are trained from scratch. Network architectures and training details are included below. The CIFAR-10 dataset consists of 50k train and 10k test 32x32 images split into 10 classes while ImageNet contains 1.28M train and 50k validation images across 1k classes. We use standard hyperparameters for each model and dataset combination. For PCN hyperparameters (e.g., transformation epoch, compression configs) we did not perform parameters scans but instead used heuristics. We discuss in Section 2 the heuristic for picking the transformation epoch. For picking the layers to transform, we use the input-based transformation on the widest layers of each network; we use the output-based transformation only as an optimization if the next layer's input is an order of magnitude compressed after the input-based transformation. For the transformation size of each layer, we outline the used configurations below. For wide ResNets, we chose parameters so that the input width of compressed layers matched the original width of the corresponding layer in a non-wide ResNet.

Network Architectures and Training Details We present details on the architectures and training procedures used in the main body of the paper.

CIFAR-10 Network Architectures Parent networks for CIFAR-10, before transforming to PCN versions, are given in Table 4. Convolutional layers use 3×3 kernels unless otherwise stated. Numbers represent filters/neurons, M represents 2×2 max pooling, and A represents filter-wise average pooling. Brackets denote residual blocks, with the multiplier out front denoting the number of blocks in the residual stage. All layers use `ReLU` activation, except the last dense layers where we use `Softmax`, and all convolutional layers use zero-based "same" padding. For ResNet architectures, we use the original versions presented in (He et al., 2016). That is, we adopt batch normalization after convolution. Convolution layers in ResNets do not use a bias. While not explicitly necessary in the first stage, for symmetry we always use a projection shortcut (1×1 convolution) at the first residual block in each stage. Downsampling is performed by stride two convolutions in the first block of stages two and three.

ImageNet Network Architectures Parent networks for ImageNet are shown in Table 5. Notation is the same as for CIFAR-10 above. For ResNet architectures, initial convolution layers use 7×7 kernels with stride two and initial max pooling layers uses 3×3 pooling with stride two. We use bottleneck blocks (He et al., 2016) where the first and third convolution use 1×1 kernels and only the middle layer uses 3×3 kernels. As for ResNets on CIFAR-10, we use projection shortcuts at the beginning of every stage. We follow the widely adopted implementation which performs downsampling using stride two convolutions at the 3×3 convolutional layers in bottleneck blocks rather than the first 1×1 layer.

Training Conv4 on CIFAR-10 For all experiments, we train the parent Conv4 model and the Conv4-PCNs in the same manner. Specifically we use a batch size of 60, Adam (Kingma & Ba, 2014) optimizer with default TensorFlow learning rate 0.001, cross-entropy loss, and when applicable 5000 random training samples for computing PCA statistics. Since the network and dataset are small, we can compute PCA for all required layers using a single forward pass with batch size 5000. We train on a random 45k/5k train/val split (different each run) and train for a maximum of 20 epochs. When applicable, early stopping is defined as the epoch of maximum validation set accuracy. We evaluate on the test set and run each experiment five times. When shown, error bars correspond to five run max/min values, otherwise values correspond to five run mean. We do not use any dataset preprocessing. Experiments were run on a machine with one NVIDIA Tesla V100 GPU.

Table 5: Network architectures for ImageNet.

Name	VGG-19 (Simonyan & Zisserman, 2014)	ResNet-50 (He et al., 2016)	ResNet-152 (He et al., 2016)	WideResNet-50
Conv Layers	2x64, M 2x128, M 4x256, M 4x512, M 4x512, M	64, M 3x[64, 64, 256] 4x[128, 128, 512] 6x[256, 256, 1024] 3x[512, 512, 2048]	64, M 3x[64, 64, 256] 8x[128, 128, 512] 36x[256, 256, 1024] 3x[512, 512, 2048]	128, M 3x[128, 128, 512] 4x[256, 256, 1024] 6x[512, 512, 2048] 3x[1024, 1024, 4096]
FC Layers	2x4096, 1000	A, 1000	A, 1000	A, 1000

The Conv4-PCN reported in the main text summary of results (with 101,810 trainable parameters) was created using the following compression configuration: `conv1:(None, 40), conv2:(20, 50), conv3:(40, 100), conv4:(80, 60), fc1:(50, 90), fc2:(40, 180), output:(30, None)`. Tuples for each layer represent the effective dimensionality for the input-based transformation and the number of dimensions to keep for the output-based transformation respectively. For this PCN, we perform compression after epoch two.

Training ResNet on CIFAR-10 We use data augmentation as reported in (He et al., 2016). Namely, "4 pixels are padded on each side, and a 32×32 crop is randomly sampled from the padded image or its horizontal flip. For testing, we only evaluate the single view of the original 32×32 image". Additionally, we standardize each channel by subtracting off the mean and dividing by the standard deviation computed over the training data. As for the Conv4 network, we use a 45k/5k train/val split and 5000 samples for compression. Again, a single forward pass suffices for computing PCA statistics. We do not use data augmentation for compression samples. We follow the hyperparameters reported in (He et al., 2016): We train for 182 total epochs using a batch size of 128 and an initial learning rate of 0.1 which we drop by a factor of 10 after epochs 91 and 136. For ResNet-110 we use a 0.01 learning rate warm up for the first epoch. We use SGD with momentum 0.9, cross-entropy loss, and L_2 regularization coefficient of $5 * 10^{-5}$ for CNN and dense weights. Note that because TensorFlow incorporates L_2 regularization into the total loss, after taking the derivative the coefficient for weight decay becomes $1 * 10^{-4}$. This matches they weight decay used in different implementations. We do not use weight decay for batch normalization parameters. For CNN layers, we adopt `he-normal/kaiming-normal` initialization (He et al., 2015). We run each experiment three times and report the average test accuracy after epoch 182. Experiments were again run on a machine with one NVIDIA Tesla V100 GPU.

To create the WideResNet-20-PCN0, we use the input-based transformation for all convolutional layers in stages two and three of WideResNet-20 (except the first convolution and projection shortcut in stage two). We use a fixed effective dimensionality of either 32 or 64, corresponding to the input dimension of the respective layer in a ResNet-20. For WideResNet-20-PCN1, we use the input-based transformation for all CNN layers in all residual blocks. As for PCN0, we use an effective dimensionality of 16, 32, or 64, the equivalent input dimension of each layer in a ResNet-20. For WideResNet-20-PCN2, we further compress PCN1 by using the input-based transformation for the dense layer (effective dimensionality 64) and the output-based transformation for all CNN layers in stage three (retaining 64 out of 256 filters).

Training VGG on ImageNet We adopt the basic training procedure from the original paper (Simonyan & Zisserman, 2014). For train set data augmentation, we randomly sample a 224×224 crop from images isotropically rescaled to have smallest side length equal to 256. We include random horizontal flips, but omit random RGB color shift. We subtract the mean RGB values, computed on the train set, from each channel in an image. For testing, we use the center 224×224 crop from validation images rescaled to have smallest side 256. We train for 70 total epochs using a batch size of 256 and an initial learning rate of 0.01. We decrease the learning rate by a factor of 10 after epochs 50 and 60. We use $5 * 10^{-4}$ weight decay decoupled from training loss (Loshchilov & Hutter, 2017), but multiply by the learning rate to match other implementations. Optimization is done using SGD with momentum 0.9 and cross-entropy loss. We use dropout (Srivastava et al., 2014) with rate 50% for the first two dense layers. We use default TensorFlow weight initializations (`glorot-uniform` (Glorot & Bengio, 2010) for CNN layers) and train on AWS p3.16xlarge instances with eight NVIDIA Tesla V100 GPUs. We train networks only once for cost considerations.

We create the PCN presented in the main text summary of results by using the input-based transformation for the first two dense layers in the VGG-19 network. We use an effective dimensionality of 350 and 400 respectively. For PCA statistics, we use the hidden layer activations computed from the center crops of 50,176 (a multiple of the batch size) training images. We randomly sample a different set of images for each layer to increase the amount of training data used to create the PCN. We include dropout with rate 50% on the two compressed layers. For the network presented in the main text, we perform the transformation after epoch 15 out of 70.

Training ResNet on ImageNet For train set data augmentation, we use the widely adopted procedure described as the baseline in (He et al., 2019). Namely, we first randomly crop a region with aspect ratio sampled in [3/4, 4/3] and area randomly sampled in [8%, 100%]. This procedure is sometimes referred to as `RandomResizedCrop`. We then resize the crop to 224×224 and perform a random horizontal flip. The brightness, saturation, and hue are randomly adjusted with coefficients drawn from [0.6, 1.4]. We also use PCA color augmentation with coefficients sampled from $N(0, 0.1)$. Finally, we perform channel standardization by subtracting the mean RGB values and dividing by the standard deviations computed across the training data. For validation, we use the center crop of an image isotropically rescaled to have shorter side length 256.

We use the training procedure introduced in the original paper (He et al., 2016). We train for 90 total epochs, use a batch size of 256, and an initial learning rate of 0.1. We drop the learning rate by a factor of 10 after epochs 30 and 60. We use SGD with momentum 0.9, cross-entropy loss, and L_2 regularization with coefficient $5 * 10^{-5}$. When multiplied by two in the derivative of the loss function, this L_2 penalty equals the $1 * 10^{-4}$ weight decay used in other implementations. We do not use weight decay for batch normalization parameters. For convolution layers, we use `he-normal` initialization (He et al., 2015). Following common practice, we initialize γ in the final batch normalization of a residual block with zeros instead of ones and use label smoothing with coefficient 0.1. We train on AWS p3.16xlarge instances with eight NVIDIA Tesla V100 GPUs. We train networks only once for cost considerations.

For the WideResNet-50-PCN, we transform all convolutional layers in stage four of a WideResNet-50 using the input-based transformation. We use an effective dimensionality of 512 for transformed layers. To compute PCA statistics, we use the center crops of training images isotropically rescaled to shorter side length 256. We use 50,176 (a multiple of the batch size) random images for each layer. Since the hidden layer activations occupy significant GPU memory, once a batch of images reaches the hidden layer of interest, we downsample using spatial average pooling and use depth vectors from the smaller images for PCA.

G. Additional Experiments

Robustness of PCN Transformations We evaluate the sensitivity of PCNs with respect to 1) the transformation epoch and 2) the variance threshold for the input-based transformation. For this experiment, we use the Conv4 network (Frankle & Carbin, 2019), a VGG-style CNN adapted to CIFAR-10. It contains four convolution layers followed by three dense layers. In Figure 5 we plot PCN test accuracy at early stopping[2] versus transformation epoch using four different threshold configurations. Variance cutoffs are represented using tuples corresponding to (threshold for convolution layers, threshold for dense layers). The baseline accuracy, i.e. training the parent Conv4 network, is shown as a horizontal line.

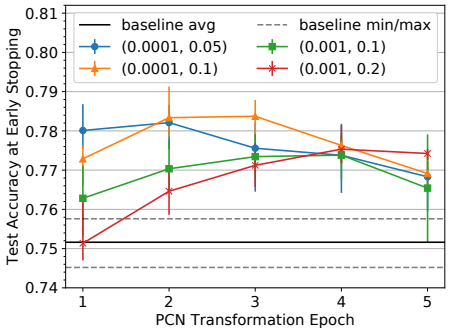


Figure 5: Robustness of PCNs.

We find that training PCNs is robust to both the transformation epoch and the variance thresholds; All configurations in Figure 5 exceed the baseline. In the best case, the Conv4-PCN improves upon the Conv4 accuracy by 3%. Furthermore, we see that the transformation can be performed early—in this case after one epoch. For reference, training requires 5-12 epochs.

Origins of Accuracy Improvement To understand why Conv4-PCNs train to a higher test accuracy than the Conv4 network (above), we ran a set of ablation experiments. When converting the Conv4 network into the Conv4-PCN, instead of transforming all layers using the input-based transformation, we transformed all layers except one.

[2]Early stopping is defined here as the epoch of peak validation accuracy.

The layer not transformed is shown on the x-axis of Figure 6. The test accuracy of the resulting PCNs is shown on the y-axis. For each PCN, we show four bars labeled by a tuple representing the variance threshold for convolution layers that were compressed, the variance threshold for dense layers that were compressed, and the epoch at which we performed the transformations. Omitting the transformation for the first fully connected layer (fc1) causes the accuracy to drop much closer to the baseline (75.16). Omitting the transformation for other layers does not change the PCN accuracy. From this ablation experiment, we conclude that the accuracy improvement of the Conv4-PCN comes from transforming the first dense layer. We conclude that by compressing the first dense layer—a layer which contains 86% of the original weights—the Conv4-PCN helps to regularize and prevent overfitting.

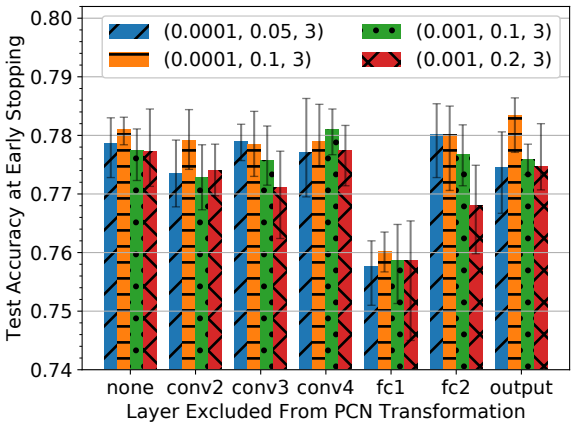


Figure 6: Origins of accuracy improvement.