Principal Component Networks: Utilizing Low-Rank Activation Structure to Reduce Parameters Early in Training

ROGER WALEFFE*, University of Wisconsin-Madison, USA
THEODOROS REKATSINAS†, ETH Zurich, Switzerland

Recent works show that overparameterized neural networks contain small subnetworks that exhibit comparable accuracy to the full model when trained in isolation. These results highlight the potential to reduce the computational costs of deep neural network training without sacrificing generalization performance. Initial approaches for finding these small networks relied on expensive multi-round train-and-prune procedures, limiting their practical potential, but more recent work identifies subnetworks using structured pruning techniques early in training. In this paper, we study network activations, rather than network weights, and find that hidden layer activations in overparameterized networks exist primarily in subspaces smaller than the actual model width. We further notice 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). PCNs compress individual layers by retaining only the high variance linear combinations of channels—defined by the principal components of the layer inputs—a key difference from structured pruning techniques which focus on individual channel pruning using localized channel measurements. We evaluate PCNs on CIFAR-10 and ImageNet for VGG and ResNet style architectures and compare against existing methods for subnetwork identification during early training. We 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. Beyond model compression, we also connect our observation regarding hidden layer activations to the feature representations learned by neural networks and discuss areas for future work.

CCS Concepts: • **Computing methodologies** → **Neural networks**.

Additional Key Words and Phrases: model pruning, compression, activations, sparsity, resource-efficient training

ACM Reference Format:
Roger Waleffe and Theodoros Rekatsinas. 2023. Principal Component Networks: Utilizing Low-Rank Activation Structure to Reduce Parameters Early in Training. *ACM/IMS J. Data Sci.* 1, 1, Article 1 (January 2023), 27 pages.
https://doi.org/10.1145/3617778

1 INTRODUCTION

Recent results suggest the importance of overparameterization in neural networks [17, 39]. The theoretical results of Gunasekar et al. [17] demonstrate that training in an overparameterized regime leads to an implicit regularization that may improve generalization. At the same time,

*Corresponding author
†Currently at Apple

Authors' addresses: Roger Waleffe, waleffe@wisc.edu, University of Wisconsin-Madison, Madison, Wisconsin, USA; Theodoros Rekatsinas, theo.rekatsinas@inf.ethz.ch, ETH Zurich, Zurich, Switzerland.

2831-3194/2023/1-ART1 $15.00
https://doi.org/10.1145/3617778

empirical results [39] 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 *subnetworks* with comparable generalization performance [3, 11, 47, 59]. Many of these works target efficient inference by pruning the weights of a fully trained network [3]. More recent works, however, show that overparameterized networks contain subnetworks that can be trained in isolation to the same (or nearly the same) generalization performance as compared to training the full model [11, 47, 59]. These results highlight the potential to reduce the training costs of high-accuracy networks. Early approaches for finding these subnetworks relied on multi-round train-and-prune procedures to periodically remove low-magnitude weights [11, 12]. While these methods result in subnetworks capable of training to accuracy matching the original overparameterized network (the lottery ticket hypothesis), they face two main challenges limiting their practical potential for end-to-end training cost reductions: 1) iterative train-prune-reset procedures are computationally expensive (they require paying the full training cost of the original model *each* iteration) and 2) individual weight pruning leads to sparse architectures that are hard to execute efficiently on modern hardware; in fact, individual weight pruning is generally implemented by multiplying weight matrices with an additional binary mask matrix, thus actually adding computation rather than reducing it. As such, follow up work identifies subnetworks using one-shot structured pruning techniques early in training [59]. These methods produce dense architectures that are hardware friendly, but rely on channel pruning using individual channel measurements. Such localized heuristics assume strong independence between channels in each network layer and, together with one-shot pruning early in training, lead to compressed models which train to lower accuracy than the full network (e.g., EB in Table 5). This motivates our study to find an efficient procedure for discovering small, dense networks early in training that can train to the same accuracy as their overparameterized counterparts.

Principal Component Networks We show that after only a few training epochs, overparameterized networks can be transformed into small networks that exhibit comparable generalization performance. Our work builds upon the next compelling observation: We consider overparameterization due to increased network width, a key quantity associated with high-accuracy models [4, 46]. *We find that hidden layer activations in wide 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 generalization accuracy of the network, 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. Only the linear combinations of channels defined by the high-variance principal components are used to construct the transformed PCN layers. As such, PCNs have the following desirable properties: 1) overparameterized layers are compressed using holistic layer-wise information rather than individual channel measurements, 2) based on the above observations, the transformation for each layer can be done once early in training, and 3) transformed PCN layers produce dense architectures well suited for GPU acceleration.

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 transform the weights of the network using these subspaces to obtain the corresponding PCN.
(4) Continue training the smaller PCN for the remaining number of epochs that the overparameterized model would have required for convergence.

We describe the PCN transformation (step 3) 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 [34] and ImageNet [48] and compare against training the corresponding overparameterized model. For both ResNet [22] and VGG-style architectures [49], we show that PCNs can 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 WideResNet-20-PCN1 (Table 3) outperforms a deep ResNet-110 in test accuracy by 0.29% while training 50% faster and with 27% less energy. This observation indicates that partially training and then compressing wide models may lead to a more resource-efficient method for obtaining high-accuracy versus standard training of deep neural networks. Furthermore, we compare PCNs against existing structure pruning methods for subnetwork identification during early training and show that for the same level of trainable parameter reduction PCNs achieve higher end-model accuracy.

We conclude this work by discussing the implications of our observations regarding hidden layer activations beyond network compression. In particular, we highlight the connection of this observation to the features learned by neural networks and discuss questions for future work in areas such as model robustness and improved model accuracy.

2 BACKGROUND AND MOTIVATING OBSERVATIONS

We first review PCA and then present the empirical observations about hidden layer activations that motivate our work.

2.1 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 1. 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 1, 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 $\tilde{x}_1$.

We compute the principal components and associated variances using the spectral decomposition of the covariance matrix. We find this method to be significantly faster computationally compared to using the Singular Value Decomposition. Given N data points in $\mathbb{R}^m$ (organized in the matrix $X_{N\times m}$) with empirical mean $\boldsymbol{\mu}_m \in \mathbb{R}^m$ we have:

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

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

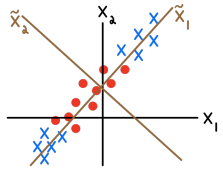


Fig. 1. Two dimensional data in the original $[x_1, x_2]$ basis can also be represented in the $[\tilde{x}_1, \tilde{x}_2]$ PCA basis. The data only have high variance along the $\tilde{x}_1$ direction. The effective dimensionality $m_e = 1$.

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.

PCA does not immediately extend to multi-dimensional inputs such as images. As a surrogate, for a batch of N matrices each with dimension $h \times w \times m$ (organized in the matrix $X_{N\times h\times w\times m}$), 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 an image channel and the point cloud a distribution of how likely each channel is to exhibit a certain pixel value, regardless of $[h, w]$ location. We can then run standard PCA on the flattened set of images $X_{N'\times m}$ 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". More specifically, given a batch of N input images, compute PCA as follows:

(1) $X_{N'\times m} = \texttt{Flatten}(X_{N\times h\times w\times m})$
(2) $\boldsymbol{\mu}_m, \mathbf{e}_m, V_{m\times m} = \texttt{PCA}(X_{N'\times m})$.

To transform an individual image $I_{h\times w\times m}$ into the PCA basis, one computes:

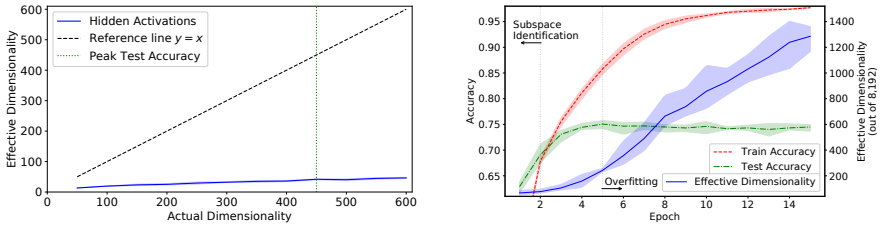
(1) $I_{(h\times w)\times m} = \texttt{Flatten}(I_{h\times w\times m})$
(2) $\tilde{I}_{(h\times w)\times m} = (I_{(h\times w)\times m} - \boldsymbol{\mu}_m)V_{m\times m}$ (subtraction applies to all rows)
(3) $\tilde{I}^i_{h\times w\times m} = \texttt{Reshape}(\tilde{I}_{(h\times w)\times m})$.

Channels in the transformed image are sorted according to decreasing variance of their pixel values. E.g., the first channel (termed principal filter) is has the highest variance of pixel values (regardless of $[h, w]$ location) among the data.

2.2 Motivation Observations: Effective Dimensionality of Hidden Layer Activations

We use PCA to study the effective dimensionality (effective width) of hidden layer activations in neural networks. We present three experiments that highlight our findings and then summarize key takeaways.

Experiment 1 We first analyze the activations of a simple neural network after learning is complete. We train a network with an input layer, a hidden layer with a variable number of nodes, and an output layer with 10 neurons on MNIST [36]. Both dense layers use sigmoid activation and we train until convergence of the validation accuracy. We then compute 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 2a 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 dimension substantially smaller the full space: with 450 nodes in the hidden layer the network achieves peak test accuracy of 98%, but rather than



(a) Effective dimensionality vs. width (b) Effective dimensionality vs. training

Fig. 2. Effective dimensionality (# of high-variance directions) for hidden layer activations in simple neural networks. (a) Hidden layer activations in trained neural networks exist primarily in subspaces more than 10× smaller than the actual model width. (b) Moreover, the subspaces which contribute most to the generalization accuracy of the network are identified early in training.

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 in simple neural networks during training. In this experiment we do not vary the network architecture, but instead consider a fixed CNN over CIFAR-10. We use the Conv4 network [11] 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 2b. 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 $\sim$ 50-dimensional subspace (recall the full space has 8,192 dimensions) 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 $\sim$ 50 to $\sim$ 200. We conjecture that these directions are mainly overfitting to the training data, as the test accuracy grows slowly while the train accuracy rapidly surpasses it. The test accuracy can still increase in this regime due to *fine-tuning* of weights related to the high-variance directions discovered in the first region. We conclude that the high-variance subspace which contributes most to the generalization accuracy of the network is identified early in training.

Experiment 3 Lastly, we measure the effective dimensionality of hidden layer activations during training in models representative of those used by practitioners. For this experiment we use a WideResNet-20 (Table 6) over CIFAR-10. After each training epoch, we perform PCA on the hidden activations between each layer and compute the effective dimensionality. We report results for the hidden layer activations input to the last ResNet block in Figure 3 (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 3 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.

Second, notice that (as was the case for the simple models in Experiments 1 and 2) 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 threshold τ in at most 42 directions—a space 6× smaller. Finally, observe that while the training dynamics of Experiment 2 and Experiment 3 qualitatively differ—in particular the more complex ResNet model suffers less from overfitting—both experiments show that the generalization accuracy of the network is primarily determined by the high-variance subspaces of network activations which are identified early in training.

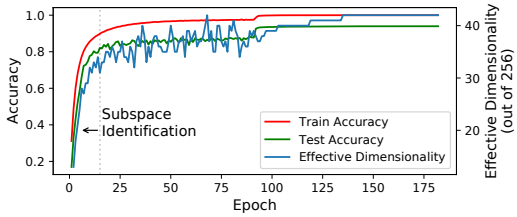


Fig. 3. Effective dimensionality (# of high-variance directions) for the hidden layer activations input to the last ResNet block in a WideResNet-20 trained on CIFAR-10. The high-variance subspace of activations which contributes most to the generalization accuracy of the network is 1) 7× smaller than the full space (35 vs. 256) and 2) identified early in training (before epoch 15 out of 182).

Takeaway Experiments 1, 2, and 3 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, Experiments 2 and 3 show that the few high-variance directions which contribute most to generalization accuracy of a given network are identified early in training. *We find these observations hold across a diverse array of models and datasets*, beyond just the experiments presented above. We use these observations as motivation to transform activations into their high-variance PCA bases after only a few epochs.

3 PRINCIPAL COMPONENT NETWORKS

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

3.1 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 when defining a new variable, or for clarity. Our goal is to transform W^i and $\mathbf{b}^i$, for each layer i, by considering the high-variance PCA basis of both the input and output activation spaces.

Transformation Based on Input Activations To transform a layer based on the input activation space (called the input-based transformation), we compute PCA on a batch of input vectors $\mathbf{h}^i$ (using Equation 1). 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 2.1). After finding the PCA basis, we can rewrite any input vector $\mathbf{h}^i$ using these coordinates. If we do so, however, we

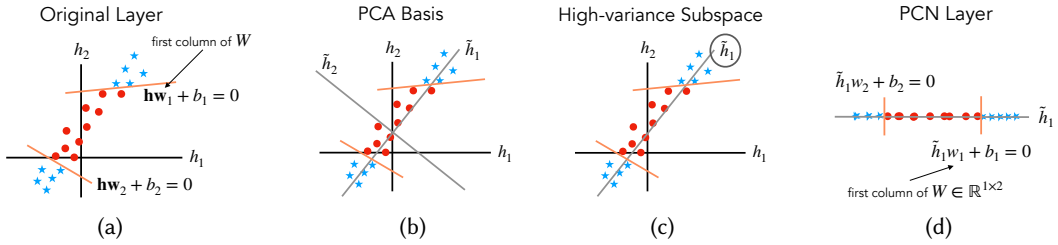


Fig. 4. End-to-end illustration of the PCN input-based transformation. (a) A batch of two-dimensional $[h_1, h_2]$ hidden layer activations input to the layer (shown as points with the different colors illustration different classes) and the original weights which act on these activations. (b) The input hidden activations have an effective dimensionality of one. They only have high-variance along the first principal component $\tilde{h}_1$. (c) The input-based transformation will approximate the original layer by selecting only the high-variance subspace. (d) The original input activations and weights of the layer are transformed into the high-variance subspace. The columns of the weight matrix are now one-dimensional instead of two-dimensional.

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 2. We denote variables represented using PCA coordinates with tilde.

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

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 (see Section 2.1). 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 3 we rewrite $\mathbf{h}^i$, W^i, and $\mathbf{b}^i$ in the high-variance subspace of the input activations.

$$\tilde{\mathbf{h}}^i_{m_e} = (\mathbf{h}^i_m - \boldsymbol{\mu}^i_m)U^i_{m\times m_e} \qquad (3)$$
$$\tilde{W}^i_{m_e\times n} = (U^i_{m\times m_e})^T W^i_{m\times n} \qquad \tilde{\mathbf{b}}^i_n = \mathbf{b}^i_n + \boldsymbol{\mu}^i_m W^i_{m\times n}$$

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 trainable weights; The size of the weight matrix W reduces from $m \times n$ to $m_e \times n$. We empirically find that often $m_e \ll m$ (see Section 2.2). 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 $i+1$. In Figure 4 we conceptually illustrate the end-to-end steps in the input-based transformation for dense layers.

This approximation introduces limited error to the output of the transformed layer. 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 $\tilde{W}^i_{m\times n}$.

Transformation Based on Output 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). The basic idea is to use the fact that if we know layer $i+1$ is going to perform the input-based transformation, then we can analyze this transformation to see which outputs of layer i are required to execute the transformation with low error. As such, the output-based transformation at layer i requires that layer $i+1$ perform the input-based transformation.

More specifically, assume that layer i is transformed per the input-based transformation (Equation 3). Layer i then computes an approximate $\hat{\mathbf{h}}^{i+1}$ output vector: $\hat{\mathbf{h}}_n^{i+1} = \sigma(\tilde{\mathbf{h}}_{m_e}^i \tilde{W}_{m_e \times n}^i + \tilde{\mathbf{b}}_n^i)$. 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 $\tilde{\mathbf{h}}_{n_e}^{i+1} = (\hat{\mathbf{h}}_n^{i+1} - \boldsymbol{\mu}_n^{i+1}) U_{n \times n_e}^{i+1}$. 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 $\tilde{\mathbf{h}}^{i+1}$:

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

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 3.

The output-based transformation and input-based transformation can be applied to layer i in either order. In the Section A we discuss technical details regarding the transformation order.

3.2 PCNs for Convolutional Layers and ResNets

We extend the PCN transformations to CNNs and ResNets. We provide a high-level description but leave additional details to the Appendix.

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 height and width 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. Recall from Section 2.1 that to compute PCA 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. 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 3 and Equation 5 to handle tensors instead of vectors and matrices. We leave the details to Section B.

ResNets While ResNets consist of convolution layers, they require special care due to the inclusion of residual connections. Since the input-based 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-based 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-based transformation in the same way. More details can be found in Section C.

3.3 Training PCNs

Given the above layer transformations, we present the training procedure for PCNs.

Input 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 set of layers $O \subseteq L$ to transform via the output-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 or O. Since the output-based transformation at layer i requires that layer $i+1$ perform the input-based transformation, there exists a constraint between the sets I and O: if i is in O then $i+1$ must be in I.

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 layers from N into their PCN version (Steps 3-4). These steps have negligible runtime compared to a single training epoch (Section D).

Step 3 For each layer $i \in O$ we perform the output-based transformation.

Step 4 Transform all layers $i \in I$ using the input-based transformation.

Step 5 Train the generated PCN for T epochs. We do not update the PCA-basis transformation matrices (U) at each PCN layer. The trainable parameters are only the layer weight matrices (W) and bias vectors ($\mathbf{b}$). Training the PCN for the remaining epochs allows for more efficient training compared to the overparameterized model.

4 EXPERIMENTS

We empirically validate the performance of PCNs against their overparameterized counterparts and compare against structured pruning methods for model compression early in training.

4.1 Experimental Setup

We first discuss the setup used throughout the experiments.

General Setup: Datasets, Models, Metrics, and Baselines We consider several architectures including VGG-style CNNs [49] and ResNets [22] over CIFAR-10 [34] and ImageNet [48]. 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 train all models from scratch. To measure performance, we consider standard test accuracy metrics. We also evaluate the time and energy improvements that PCNs introduce. Experiments on CIFAR-10 are averaged over at least three runs. For ImageNet, we run each experiment only once for cost considerations. As a baseline for structured pruning early in training, we use the method for identifying Early-Bird Tickets from You et al. [59]. All experiments were executed using TensorFlow. We implement no special optimizations and use only the high level TensorFlow Keras API. We run experiments using AWS P3 instances with V100 GPUs.

Hyperparameters We use standard hyperparameters for each model and dataset combination. I.e., we use existing learning rate schedules, batch sizes, data augmentation, etc. that are reported in the literature [22, 23, 33, 49]. The exact architecture details and hyperparameters for each model and dataset can be found Section E. For PCN hyperparameters (e.g., the transformation epoch and the compression configurations) we use simple heuristics rather than performing hyperparameter scans. Picking the optimal value for K (the transformation epoch, Section 3.3) is a challenging problem. Heuristically, we train the original overparameterized network so long as the validation accuracy is increasing at a high rate (e.g., more than 5% per epoch). To decide which layers to transform, we use the input-based transformation on the widest layers of each network until we reach the desired compression ratio. For the transformation size of each layer, we use either a small PCA variance threshold τ or set an explicit target effective dimensionality (Section 2.1). The latter is useful for compressing layers in wide ResNet models, as we can explicitly chose parameters so that the input width of compressed layers is equal to the original width of the corresponding layer in a non-wide ResNet of the same depth. 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. PCN hyperparameter details can be found in Section E.

4.2 End-to-End Experiments

We now discuss end-to-end training results for PCNs and baseline structured pruning methods. Results are reported in Tables 1-5. Where applicable, for each experiment we train all models for the same total number of epochs and using the same hyperparameters. Next, we highlight key takeaways among all end-to-end results before focusing on each setting (Table) in more detail.

Key Takeaways We summarize end-to-end results in Table 1. Although PCNs perform compression early in training (between 8% and 21% of the total number of epochs), they consistently reduce the number of trainable weights in overparameterized networks yet sacrifice little, if any, accuracy. Our method extends from the comparatively simple CIFAR-10 dataset to the challenging ImageNet dataset. For the former, we achieve comparable compression and accuracy improvements to the original winning lottery tickets found in [11] (on Conv4 at 4% weights remaining both PCNs and lottery tickets achieve 77% accuracy), but require no iterative train-prune-reset procedure. For the latter, in one case we observe a reduction in parameters by 5.28×, while in the other case we observe no loss in end-model accuracy. These small, if any accuracy drops are a common characteristic of PCN models. The holistic layer-wise compression criteria used by PCNs allows for higher end-model accuracy when compared to structured pruning methods based on localized channel measurements (Table 5). We next discuss individual experiments in more detail.

Table 1. Summary of results. Across both VGG-style and ResNet architectures, PCNs can reduce parameter counts early in training while achieving high end-model accuracy. (Trans. Epoch = Transformation Epoch)

Dataset	Network	Trainable Parameters	Accuracy	PCN Parameters	PCN Accuracy	Trans. Epoch/Total Epochs
CIFAR-10	Conv4	2.43M	75.16	101k	**77.25**	2/20
CIFAR-10	WideResNet-20	4.33M	93.60	1.32M	93.52	15/182
ImageNet	VGG-19	143M	70.99	27.2M	70.27	15/70
ImageNet	WideResNet-50	98.0M	77.52	62.3M	**77.64**	16/90

Table 2. ResNet-50-PCNs trained on ImageNet compared to the original ResNet-50 model. PCN transformations are performed after epoch 18 out of 90.

Network Name	Trainable Parameters	Center Crop Val. Acc. at Epoch 90	
		Top-1	Top-5
ResNet-50	25,557,032	75.60	92.78
ResNet-50-PCN0	22,919,720	75.53	92.79
ResNet-50-PCN1	20,421,160	75.32	92.71
ResNet-50-PCN2	17,835,048	75.15	92.63
ResNet-50-PCN3	12,897,320	73.96	91.87

We start by empirically validating the performance of PCNs against their overparameterized counterparts. For these experiments, we train an overparameterized model to completion and report the model size and end-model accuracy. We then compare against the end-model accuracy and resulting model size of a corresponding PCN. We focus on ResNet models rather than VGG-style networks, as they better approximate the state-of-the-art models used by practitioners.

ResNet-50 on ImageNet We report the results for ResNet-50 architectures and ResNet-50-PCNs trained on ImageNet in Table 2. We use four different PCNs (PCN0-PCN3) representing 10, 20, 30, and 50 percent trainable parameter reduction respectively. PCNs achieve near-matching top-1 and top-5 accuracy as compression ratios increase from 10 to 30 percent. PCN accuracy, however, begins to fall off more rapidly between 30 and 50 percent parameter reduction (PCN2 to PCN3). This occurs for the following reason: In order to reach these higher compression ratios, we need to use the input-based transformation on increasingly narrow layers in the ResNet-50 architecture (e.g., with width less than or equal to 256 channels). The narrower layers get, the less suited they are for PCN transformations as the effective dimensionality approaches the true layer width (Figure 2a). If we are unable to reach a desired compression ratio, we are forced to increase PCA variance thresholds (τ) to decrease effective dimensionalities (and thus increase compression). This is particularly important for end-model accuracy, as the error of the PCN transformation with respect to the orignal layer is proportional to the variance threshold. In fact, the observation that ResNet-50 architectures can not be compressed by more than $\sim 30\%$ without increasing PCA variance thresholds actually indicates a profound strength of ResNet models: standard ResNet architectures generally use their available dimensions in each layer quite well, i.e., these architectures are not heavily overparameterized with respect to model width. As PCNs are better suited for wide layers, ResNet models provide a challenging setting for our method. Yet, we show in Table 5 that PCNs considerably outperform structured pruning methods with respect to end-model accuracy on ResNet models trained over ImageNet.

Effect of Network Width As mentioned in the introduction, network width is a key quantity associated with high-accuracy models [4, 46]. We now evaluate the hypothesis that wide networks improve generalization and that PCNs are well suited to reduce the number of parameters in wide, overparameterized models early in training while achieving near-matching accuracy. On both

Table 3. WideResNet-20 PCNs trained on CIFAR-10 compared to the original WideResNet-20 model. We include standard ResNet-20 and ResNet-110 models for additional comparison points. WideResNet-20 PCNs can achieve ≥ 75% trainable parameter reductions while achieving high end-model accuracy (≥ 93.00). 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 4. Comparison of a WideResNet-50-PCN trained on ImageNet to the original WideResNet-50 model. We include standard ResNet-50 and ResNet-152 models for additional comparison points. The WideResNet-50-PCN achieves a 37% parameter reduction while achieving matching model accuracy (77.52%+). This model outperforms a standard ResNet-152 with the same number of parameters. PCN transformations are performed after epoch 16.

Network Name	Total Parameters	Trainable Parameters	Center Crop Val. Acc. at Epoch 90	
			Top-1	Top-5
ResNet-50	25,610,152	25,557,032	75.60	92.78
ResNet-152[1]	60,344,232	60,192,808	77.00	93.30
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

CIFAR-10 and ImageNet, we train wide ResNet architectures—networks identical to the original ResNets [22], 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 3 and 4. 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 as the accuracy difference between these models falls within the error bars.

We find that wide networks improve accuracy compared to standard ResNet models and that they can outperform deep ResNet models. On CIFAR-10, the WideResNet-20 model achieves 93.6% accuracy, outperforming a standard ResNet-20 (91.03%) and a deep ResNet-110 (92.94%). Similar results hold for the WideResNet-50 model on ImageNet. We also find that PCNs can reduce the number of parameters in these 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. PCN0 achieves near-matching accuracy to the WideResNet-20 with only 30% of the trainable parameters. On ImageNet, our PCN achieves matching accuracy with 63% of the original weights and outperforms a ResNet-152 containing a comparable number of trainable parameters. Moreover, Tables 3 and 4 show that wide networks can be converted into PCNs early in training. The transformation occurs at epoch 15 out of 182 on CIFAR-10 and epoch 16 out of 90 on ImageNet. We evaluate the potential for these parameter reductions early in training to reduce the computational costs of training in the following paragraphs. In summary, we have shown that PCN transformations of wide, overparameterized models are well suited to achieve high end-model accuracy with reduced parameter overhead.

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

Performance: Training Time and Energy We next 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. For these results, we calculate energy usage by integrating `nvidia-smi` power measurements. Before discussing results, however, we note the following: Our current implementation uses standard GPU kernels for existing neural network layers. For best performance, new GPU kernels and training procedures are required which optimize the PCN basis transformation at each layer (multiplication with fixed parameters U) together with the layer itself (multiplication/convolution with learnable weights W). For example, note that performant kernels for convolution fuse zero-based input padding together with compute, reading a batch of image data from memory only once. In contrast, using existing kernels, our implementation must manually pad[2], then multiply by U, then perform convolution, a process which requires three memory I/O operations (memory access is known to bottleneck training [6]). Moreover, optimal training in the presence of fixed and trainable parameters is not well studied. In particular, how can existing neural network frameworks (e.g., TensorFlow) improve energy efficiency and runtime for parameters which do not require gradients (e.g., quantization, caching, etc.)? We leave the implementation of custom kernels and optimized frameworks for combining fixed and learnable parameters as well as a detailed analysis of the PCN training time and energy cost reductions for future work. In the following paragraph, however, we highlight existing results that offer promising evidence for the potential of PCNs to reduce the computational costs of training high-accuracy models.

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 end-to-end training of the full WideResNet-20 and 27% less energy than end-to-end training of the deep ResNet-110 network. At the same time, the end-to-end training time of the PCN (44.73 minutes) is 10% and 50% faster than these models respectively (49.47 and 90.12 minutes). We believe the fact that wide ResNet PCNs can train faster and achieve higher accuracy than deep ResNet models is an important takeaway from this work. Furthermore, performance improvements extend to inference time: e.g., on ImageNet, our unoptimzed PCN reduces the time to compute the validation set predictions (36s) by 47% compared to the deep ResNet-152 (66s). Yet the WideResNet-50-PCN achieves the highest accuracy of these two models. As described above, we expect a fused PCN kernel and framework optimizations could significantly improve the energy consumption and training time benefits of PCNs. *Finally, we also note here that 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 D).*

Comparison To Structured Pruning While we have primarily focused so far on comparing PCNs to their overparameterized counterparts, we conclude this section by comparing them directly to other methods for compressing models early in training. We focus on methods which use structured pruning techniques as they, like PCNs, produce compressed models with dense architectures that can be easily accelerated using GPUs. As our structured pruning baseline we use the method for drawing Early Bird Tickets from You et al. [59]. We do not compare directly to iterative train-prune-reset procedures for finding sparse lottery tickets. While these approaches generally lead to the highest compression ratios (e.g., [12]), methods for finding sparse subnetworks are expensive (they require paying the full training cost of the original model *each* iteration) and the resulting sparse architectures are not well suited for modern GPU hardware (they actually add computation to the original model rather than reduce it, as individual weight pruning is generally

[2]When transforming a convolution layer, the padding also needs to be transformed (multiplied by U), or the convolution kernel needs to support non-zero-based padding. See Section B for more details.

Table 5. Comparison of the accuracy relative to the overparameterized model for PCNs and Early Bird Tickets at different reductions in trainable parameters. We use a ResNet-50 on ImageNet as the base model. PCNs achieve accuracy closer to the original model compared to the structured pruning method used to find Early Bird Tickets.

Method	Parameter Reduction	Center Crop Val. Acc. at Epoch 90	
		Top-1	Top-5
-	0%	75.60	92.78
PCN	30%	**75.15**	**92.63**
EB	30%	73.87	91.32
PCN	50%	**73.96**	**91.87**
EB	50%	73.36	91.16

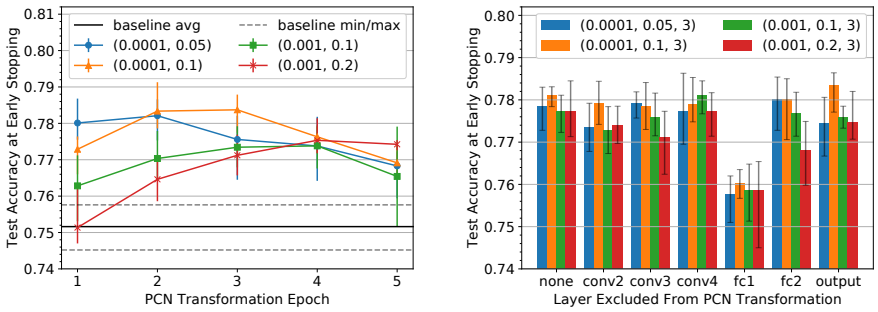
implemented by multiplying the original weight matrices by added binary mask matrices). Thus the practical potential of such methods is currently limited.

We compare the accuracy of overparameterized models, PCNs, and Early Bird Tickets using ResNet-50 architectures on ImageNet. Results are shown in Table 5. Recall that compressing ResNet-50 models is a challenge for the PCN transformations. Yet, our PCN models consistently achieve end-model accuracy closer to the original ResNet-50 when compared to Early Bird Tickets with the same parameter reduction. The reason for this is that structure pruning techniques (like those used to find Early Bird Tickets) make pruning decisions using only localized measurements (e.g., data from individual channels). In contrast, PCNs use holistic layer-wise measurements for pruning decisions (keeping only the high-variance linear combinations of channels) allowing them to better capture the information relevant to the generalization accuracy. The advantage of localized pruning decisions, however, is that the pruned channels can be easily removed from the network without any additional overhead. On the other hand, PCNs introduce the need to compute the linear combinations of channels (multiplication by U) in order to perform pruning. As such, Early Bird Tickets currently lead to an improved reduction in model compute. For example, an Early Bird Ticket with 50% trainable parameter reduction reduces the total training FLOPs by 39% compared to a 14% drop in total training flops for the corresponding PCN. As discussed above, optimizing the PCN basis transformation (multiplication by U) is an open area of future work. We believe with new GPU kernels, the PCN overhead (dominated by memory access to image data rather than compute) can be reduced, allowing the end-to-end computational improvements of PCNs to approach those of structured pruning methods. In the discussion section below, we also discuss potential areas of future work on combining the strengths of both methods.

4.3 Microbenchmarks

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 simple Conv4 network [11] (to allow us to vary many parameters), a VGG-style CNN adapted to CIFAR-10. It contains four convolution layers followed by three dense layers. In Figure 5a we plot PCN test accuracy at early stopping—early stopping is defined here as the epoch of peak validation accuracy—versus the transformation epoch using four different variance threshold configurations. Variance cutoffs are represented using tuples corresponding to (variance threshold for convolution layers, variance threshold for dense layers). The baseline accuracy, i.e. training the original Conv4 network, is shown as a horizontal line.

We find that training PCNs is robust to both the transformation epoch and the variance thresholds (so long as the variance thresholds are low); All configurations in Figure 5a reach high end-model



(a) Robustness of PCN transformations. (b) Origins of accuracy improvement.

Fig. 5. Evaluation of the sensitivity of PCNs with respect to 1) the transformation epoch and 2) the variance threshold for the input-based transformation using the simple Conv4 network. We list variance thresholds using tuples corresponding to (variance threshold for convolution layers, variance threshold for dense layers). (a) PCNs achieve high end-model accuracy across varying transformation epochs and variance thresholds. (b) We investigate the origins of the accuracy improvement for Conv4-PCNs compared to the baseline Conv4 model and find that PCNs help to regularize and prevent overfitting in layers with many weights (in this case the first fully connected layer (fc1)).

accuracy. 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 up to 20 epochs depending on early stopping.

To understand why Conv4-PCNs train to a higher test accuracy than the Conv4 network, 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. The layer *not* transformed is shown on the x-axis of Figure 5b. 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.

While the significant accuracy improvements of the PCN models in the above microbenchmarks does not extend to more complex models like ResNets (as these models do not suffer heavily from overfitting like the Conv4 network does), we find that the above takeaway—that PCNs can help to regularize overparameterized models and improve generalization—does extend beyond the Conv4 model. In particular, we notice that the generalization gap, i.e., the difference between the train and test accuracy, consistently decreases for PCN models compared to the original networks. For example, the difference between the train and test accuracy for a WideResNet-50 on ImageNet is 6%, while the generalization gap for our WideResNet-50-PCN is only 2.5%.

5 DISCUSSION AND FUTURE WORK

While we have primarily focused on how the observations in Section 2.2 can be used to reduce model size early in training, in this section we briefly discuss the implications of our observation

more generally and highlight areas for future work. We start by discussing future work in model compression and resource-efficient training, then focus on how our observation—that hidden layer activations exist in low-dimensional subspaces—connects to feature representations learned by neural networks as well as model robustness and accuracy.

Given the large number of recent works that study methods to reduce the computational cost of training (many of which utilize model compression), we believe an important area of future work is to focus on the following question: *What is the best way to train neural networks under a certain objective, e.g., lowest energy consumption to achieve a certain accuracy?* In a related manner, how can we synthesize the suite of existing compression methods, many of which have different advantages? Namely, while we have shown that PCNs have several performance benefits (e.g., the transformed PCN models are dense and can train to high end-model accuracy) other pruning techniques (sparse pruning, structured pruning, etc.) also provide advantages (see the Comparison to Structured Pruning in Section 4.2). For example, sparse pruning based methods [11] can often achieve the highest compression ratios for a given accuracy. This is likely to be particularly useful for certain objectives, e.g., those which care about the parameter overhead, but potentially harmful for others, e.g., those which care about the end-to-end computational cost of training. Moreover, we believe it is also of interest to study *why* each of these methods achieves their respective performance, primarily accuracy, in an effort to better understand the role of sparsity in deep learning more generally [13, 26]. Combining the observations of existing methods for compression early in training is likely to lead to both improved methods for reducing end-to-end training costs for a given accuracy and to a better understanding of training dynamics.

As described in the preceding paragraph, in addition to reducing the computational cost associated with training overparameterized models, we are also interested in the potential insights of PCNs into the learning dynamics of deep neural networks. As such, for the remainder of this section we discuss potential implications and areas for future work related to our underlying observation: hidden layer activations are grouped into low-dimensional subspaces early in training, and remain as such in fully trained networks. Recall that these hidden layer subspaces do not correspond to subsets of the activations at each layer, but rather to linear combinations, i.e., *superpositions*, of the activations. The fact that these superpositions still lead to high end-model accuracy means that they must capture important features related to the data and the learning task. Recent work by Elhage et al. [9] also finds that models often store features in superposition given "relatively natural setups, suggesting this may also occur in practice". We believe their results along without our observations may be of broad interest for future work.

If linear combinations of hidden layer activations are connected to the features learned by neural networks, our observations may lead to interesting opportunities for future work in areas such as adversarial examples, model robustness, and improved model accuracy. Specifically, how do we utilize this information to design more robust models? Can these observations lead to methods to prevent adversarial attacks? Are these superposition features transferable? Why does the number of superposition features stop increasing after the early phase of training? Can we continue to add new features in the latter phase of training to improve model performance? Can we use information about learned features to do better routing for mixture-of-expert models [10], thus achieving higher accuracy? We encourage further research and understanding of these questions.

Finally, we conclude by highlighting the strong connection between our observation and the observation by Gur-Ari et al. [18] that: the neural network's gradient "dynamically converges to a very small subspace after a short period of training. The subspace is spanned by a few top eigenvectors of the Hessian ... and is mostly preserved over long periods of training". Our observation closely mirrors this statement, but focuses on hidden layer activations rather than gradients. We believe that these two observations *are likely highlighting the same training behavior*,

with one observing these dynamics during the forward pass and one during the backward pass. This connection adds additional intrigue to the above questions about network features etc., but also introduces additional questions such as those surrounding the optimization algorithms used for learning. As such, while we have focused in this work on the implication of low-dimensional hidden layer activations on compression during early training, we believe this observation is of independent interest to the machine learning community and are particularly excited about the results of this observation in future research.

6 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 [19, 20, 24, 28, 37, 38, 50, 58], distillation [2, 25], quantization [16], and weight decomposition [8, 31, 32, 35, 57, 60] are performed after training. Specifically engineered networks [27, 29], binary weights [61], and low-rank [7] architectures help lessen model size from the start, while other works aim to sparsify networks during the learning process [1, 42, 44, 45, 51–53, 55, 56]. Additional works try to identify subnetworks contained in large models that can train on their own to high accuracy [11, 47, 59]. We discuss works related to our paper in more detail.

After Training The compression works of [14, 43] relate the most to the transformations we use in Section 3. Luo et al. [43] prune a subset of filters at layer i based on how accurately layer $i+1$ can produce its output. In contrast, our output-based transformation prunes a subset of filters based on how accurately the next layer can write its input in the high-variance PCA subspace. PCNs additionally transform overparameterized layers using our input-based transformation (Section 3.1) and can perform these transformations early in training (rather than only after training finishes). Garg et al. [14] find that CNN hidden layer activations exist in low dimensional spaces after training completes. Based on this, they use PCA on the hidden layer activations to determine the optimal width of each layer and the optimal depth of a pre-trained network. They then retrain a new network with the optimal dimensions from scratch. Chen et al. [5] observe similar properties for trained NLP models. Contemporaneous to the preparation of this work, Li et al. [40] highlight the activation sparsity in trained transformers. In this work, we find these observation hold *during* training and show how to use PCA during the learning process to dynamically reduce network size and thus provide the potential to reduce end-to-end training costs.

Before Training Denil et al. [7] represent weight matrices using products of low-rank factors. They fix one factor to contain the layer basis and train the other. To construct the basis vectors, they propose to use prior knowledge or to pre-train layers as autoencoders and use the empirical covariance for kernel ridge regression. While PCNs also represent layer weight matrices using a fixed basis factor, together with a trainable matrix, the key difference is that we do not impose this structure at the start. Instead we allow the first training epochs of an overparameterized model to identify an appropriate high-variance basis of hidden activations and use PCA to find it. We then directly use this basis and rewrite the weight matrix in the corresponding subspace.

Subnetwork Identification Frankle and Carbin [11] show the existence of small, sparse subnetworks embedded in overparameterized models that, when trained in isolation, meet or exceed the accuracy of the full model. They introduce an iterative train-prune-reset procedure to identify such subnetworks. In place of sparse subnetworks, we show how to find smaller, dense versions of base networks, and we show how to do this early in a single training procedure. Eliminating the iterative train-prune-reset loop allows us to scale to large datasets such as ImageNet and realize end-to-end training cost reductions. You et al. [59] identify subnetworks early in training using one-shot structured pruning techniques. Such methods have the advantage of producing dense architectures (like PCNs) which are hardware friendly, but use localized channel measurements for

pruning. We show in Section 4 that the holistic layer-wise pruning criteria used by PCNs allows our compressed models to achieve higher end-model accuracy. Wang et al. [55] also use holistic layer-wise pruning early in training but focus on finding low-rank decompositions of each weight matrix, rather than on activations as we do. Together PCNs and Wang et al. [55] show that *both* hidden layer activations (PCNs) and weights [55] exhibit low-rank structures. Finally, Ramanujan et al. [47] find a subset of random weights in an overparameterized model which achieve good generalization performance without any modification (i.e., training), but these randomly weighted subnetworks do not match the accuracy obtained by training the base model. In this work, we focus on methods for resource-efficient training of high-accuracy neural networks.

7 CONCLUSIONS

In this paper, we show that early in the training process, overparameterized networks can be transformed into smaller versions that can train to the same, or similar accuracy, as the original 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. We further observe that these subspaces, which contribute most to the generalization accuracy of the network, can be identified early in training. This motivates us to introduce Principal Component Networks, resource-efficient deep learning architectures which represent layer weights using the high-variance subspaces of their input and output activations. We demonstrate that PCNs are applicable to a diverse set of neural networks, including traditional dense neural networks, convolutional neural networks, and residual neural networks. We experimentally validate the ability of PCNs to reduce parameter counts early in training and to reach similar end-model accuracy as their overparameterized counterparts, thus providing the potential to reduce the end-to-end computational costs of training high-accuracy models. We further show that the the holistic layer-wise pruning criteria used by PCNs allows our transformed networks to reach higher end-model accuracy than those found by structured pruning techniques which use localized channel measurement based pruning criteria. Finally, we discussed the connections between our observations regarding hidden layer activations and the features learned by deep neural networks and posed questions for future work on model robustness and techniques for learning high-accuracy models.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their constructive comments on our paper. This work was supported by DARPA under grants ASKE HR00111990013 and ASKEM HR001122S0005. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views, policies, or endorsements, either expressed or implied, of DARPA or the U.S. Government. This work is also supported with funding from the Wisconsin Alumni Research Foundation.

REFERENCES

[1] Jose M Alvarez and Mathieu Salzmann. 2017. Compression-aware training of deep networks. In *Advances in Neural Information Processing Systems*. 856–867.

[2] Jimmy Ba and Rich Caruana. 2014. Do deep nets really need to be deep?. In *Advances in neural information processing systems*. 2654–2662.

[3] Davis Blalock, Jose Javier Gonzalez Ortiz, Jonathan Frankle, and John Guttag. 2020. What is the State of Neural Network Pruning?. In *Proceedings of Machine Learning and Systems*.

[4] Lingjiao Chen, Hongyi Wang, Jinman Zhao, Dimitris Papailiopoulos, and Paraschos Koutris. 2018. The effect of network width on the performance of large-batch training. In *Advances in Neural Information Processing Systems*. 9302–9309.

[5] Patrick Chen, Hsiang-Fu Yu, Inderjit Dhillon, and Cho-Jui Hsieh. 2021. Drone: Data-aware low-rank compression for large nlp models. *Advances in neural information processing systems* 34 (2021), 29321–29334.

[6] Tri Dao, Daniel Y Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. *arXiv preprint arXiv:2205.14135* (2022).

[7] Misha Denil, Babak Shakibi, Laurent Dinh, Marc' Aurelio Ranzato, and Nando de Freitas. 2013. Predicting Parameters in Deep Learning. In *Advances in Neural Information Processing Systems 26*. 2148–2156.

[8] Emily L Denton, Wojciech Zaremba, Joan Bruna, Yann LeCun, and Rob Fergus. 2014. Exploiting linear structure within convolutional networks for efficient evaluation. In *Advances in neural information processing systems*. 1269–1277.

[9] Nelson Elhage, Tristan Hume, Catherine Olsson, Nicholas Schiefer, Tom Henighan, Shauna Kravec, Zac Hatfield-Dodds, Robert Lasenby, Dawn Drain, Carol Chen, et al. 2022. Toy Models of Superposition. *arXiv preprint arXiv:2209.10652* (2022).

[10] William Fedus, Barret Zoph, and Noam Shazeer. 2021. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity.

[11] Jonathan Frankle and Michael Carbin. 2019. The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks. In *International Conference on Learning Representations*.

[12] Jonathan Frankle, Gintare Karolina Dziugaite, Daniel Roy, and Michael Carbin. 2020. Linear mode connectivity and the lottery ticket hypothesis. In *International Conference on Machine Learning*. PMLR, 3259–3269.

[13] Trevor Gale, Erich Elsen, and Sara Hooker. 2019. The state of sparsity in deep neural networks. *arXiv preprint arXiv:1902.09574* (2019).

[14] Isha Garg, Priyadarshini Panda, and Kaushik Roy. 2020. A Low Effort Approach to Structured CNN Design Using PCA. In *IEEE Access*, Vol. 8. 1347–1360.

[15] Xavier Glorot and Yoshua Bengio. 2010. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. 249–256.

[16] Yunchao Gong, Liu Liu, Ming Yang, and Lubomir Bourdev. 2014. Compressing deep convolutional networks using vector quantization. *arXiv preprint arXiv:1412.6115* (2014).

[17] Suriya Gunasekar, Blake E Woodworth, Srinadh Bhojanapalli, Behnam Neyshabur, and Nati Srebro. 2017. Implicit regularization in matrix factorization. In *Advances in Neural Information Processing Systems*. 6151–6159.

[18] Guy Gur-Ari, Daniel A Roberts, and Ethan Dyer. 2018. Gradient descent happens in a tiny subspace. *arXiv preprint arXiv:1812.04754* (2018).

[19] Song Han, Jeff Pool, John Tran, and William J. Dally. 2015. Learning Both Weights and Connections for Efficient Neural Networks. In *Advances in Neural Information Processing Systems 28*. 1135–1143.

[20] Babak Hassibi and David G. Stork. 1993. Second order derivatives for network pruning: Optimal Brain Surgeon. In *Advances in Neural Information Processing Systems 5*. 164–171.

[21] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*. 1026–1034.

[22] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[23] Tong He, Zhi Zhang, Hang Zhang, Zhongyue Zhang, Junyuan Xie, and Mu Li. 2019. Bag of tricks for image classification with convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 558–567.

[24] Yihui He, Xiangyu Zhang, and Jian Sun. 2017. Channel pruning for accelerating very deep neural networks. In *Proceedings of the IEEE International Conference on Computer Vision*. 1389–1397.

[25] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. 2015. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531* (2015).

[26] Torsten Hoefler, Dan Alistarh, Tal Ben-Nun, Nikoli Dryden, and Alexandra Peste. 2021. Sparsity in Deep Learning: Pruning and growth for efficient inference and training in neural networks. *J. Mach. Learn. Res.* 22, 241 (2021), 1–124.

[27] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861* (2017).

[28] Hengyuan Hu, Rui Peng, Yu-Wing Tai, and Chi-Keung Tang. 2016. Network Trimming: A Data-Driven Neuron Pruning Approach towards Efficient Deep Architectures. *arXiv preprint arXiv:1607.03250* (2016).

[29] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and< 0.5 MB model size. *arXiv preprint arXiv:1602.07360* (2016).

[30] Sergey Ioffe and Christian Szegedy. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167* (2015).

[31] Max Jaderberg, Andrea Vedaldi, and Andrew Zisserman. 2014. Speeding up convolutional neural networks with low rank expansions. *arXiv preprint arXiv:1405.3866* (2014).
[32] Yong-Deok Kim, Eunhyeok Park, Sungjoo Yoo, Taelim Choi, Lu Yang, and Dongjun Shin. 2015. Compression of deep convolutional neural networks for fast and low power mobile applications. *arXiv preprint arXiv:1511.06530* (2015).
[33] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).
[34] Alex Krizhevsky, Geoffrey Hinton, et al. 2009. Learning multiple layers of features from tiny images. *University of Toronto* (2009).
[35] Vadim Lebedev, Yaroslav Ganin, Maksim Rakhuba, Ivan Oseledets, and Victor Lempitsky. 2014. Speeding-up convolutional neural networks using fine-tuned cp-decomposition. *arXiv preprint arXiv:1412.6553* (2014).
[36] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.
[37] Yann LeCun, John S. Denker, and Sara A. Solla. 1990. Optimal Brain Damage. In *Advances in Neural Information Processing Systems 2*. 598–605.
[38] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. 2017. Pruning Filters for Efficient ConvNets. In *International Conference on Learning Representations*.
[39] Zhuohan Li, Eric Wallace, Sheng Shen, Kevin Lin, Kurt Keutzer, Dan Klein, and Joseph E Gonzalez. 2020. Train large, then compress: Rethinking model size for efficient training and inference of transformers. *arXiv preprint arXiv:2002.11794* (2020).
[40] Zonglin Li, Chong You, Srinadh Bhojanapalli, Daliang Li, Ankit Singh Rawat, Sashank J Reddi, Ke Ye, Felix Chern, Felix Yu, Ruiqi Guo, et al. 2022. Large Models are Parsimonious Learners: Activation Sparsity in Trained Transformers. *arXiv preprint arXiv:2210.06313* (2022).
[41] Ilya Loshchilov and Frank Hutter. 2017. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101* (2017).
[42] Christos Louizos, Max Welling, and Diederik P Kingma. 2017. Learning Sparse Neural Networks through L_0 Regularization. *arXiv preprint arXiv:1712.01312* (2017).
[43] Jian-Hao Luo, Jianxin Wu, and Weiyao Lin. 2017. Thinet: A filter level pruning method for deep neural network compression. In *Proceedings of the IEEE international conference on computer vision*. 5058–5066.
[44] Dmitry Molchanov, Arsenii Ashukha, and Dmitry Vetrov. 2017. Variational dropout sparsifies deep neural networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org, 2498–2507.
[45] Kirill Neklyudov, Dmitry Molchanov, Arsenii Ashukha, and Dmitry P Vetrov. 2017. Structured bayesian pruning via log-normal multiplicative noise. In *Advances in Neural Information Processing Systems*. 6775–6784.
[46] Daniel S Park, Jascha Sohl-Dickstein, Quoc V Le, and Samuel L Smith. 2019. The effect of network width on stochastic gradient descent and generalization: an empirical study. *arXiv preprint arXiv:1905.03776* (2019).
[47] Vivek Ramanujan, Mitchell Wortsman, Aniruddha Kembhavi, Ali Farhadi, and Mohammad Rastegari. 2019. What's Hidden in a Randomly Weighted Neural Network? *arXiv preprint arXiv:1911.13299* (2019).
[48] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. 2015. Imagenet large scale visual recognition challenge. *International journal of computer vision* 115, 3 (2015), 211–252.
[49] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).
[50] Suraj Srinivas and R Venkatesh Babu. 2015. Data-free parameter pruning for deep neural networks. *arXiv preprint arXiv:1507.06149* (2015).
[51] Suraj Srinivas and R Venkatesh Babu. 2015. Learning neural network architectures using backpropagation. *arXiv preprint arXiv:1511.05497* (2015).
[52] Suraj Srinivas and R Venkatesh Babu. 2016. Generalized dropout. *arXiv preprint arXiv:1611.06791* (2016).
[53] Suraj Srinivas, Akshayvarun Subramanya, and R Venkatesh Babu. 2017. Training sparse neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*. 138–145.
[54] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research* 15, 1 (2014), 1929–1958.
[55] Hongyi Wang, Saurabh Agarwal, and Dimitris Papailiopoulos. 2021. Pufferfish: Communication-efficient models at no extra cost. *Proceedings of Machine Learning and Systems* 3 (2021), 365–386.
[56] Yuhui Xu, Yuxi Li, Shuai Zhang, Wei Wen, Botao Wang, Yingyong Qi, Yiran Chen, Weiyao Lin, and Hongkai Xiong. 2018. Trained rank pruning for efficient deep neural networks. *arXiv preprint arXiv:1812.02402* (2018).
[57] Atsushi Yaguchi, Taiji Suzuki, Shuhei Nitta, Yukinobu Sakata, and Akiyuki Tanizawa. 2019. Scalable Deep Neural Networks via Low-Rank Matrix Factorization. *arXiv preprint arXiv:1910.13141* (2019).
[58] Tien-Ju Yang, Yu-Hsin Chen, and Vivienne Sze. 2017. Designing energy-efficient convolutional neural networks using energy-aware pruning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 5687–5695.

[59] Haoran You, Chaojian Li, Pengfei Xu, Yonggan Fu, Yue Wang, Xiaohan Chen, Richard G Baraniuk, Zhangyang Wang, and Yingyan Lin. 2019. Drawing early-bird tickets: Towards more efficient training of deep networks. *arXiv preprint arXiv:1909.11957* (2019).

[60] Xiangyu Zhang, Jianhua Zou, Xiang Ming, Kaiming He, and Jian Sun. 2015. Efficient and accurate approximations of nonlinear convolutional networks. In *Proceedings of the IEEE Conference on Computer Vision and pattern Recognition*. 1984–1992.

[61] Bohan Zhuang, Chunhua Shen, Mingkui Tan, Lingqiao Liu, and Ian Reid. 2019. Structured binary neural networks for accurate image classification and semantic segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 413–422.

APPENDIX

A INPUT- AND OUTPUT- BASED 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 in the main body of the paper, 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:

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

B TRANSFORMATIONS FOR CONVOLUTIONAL LAYERS

We provide technical details to extend the discussion on transforming convolutional layers presented in the main body of the paper.

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 3.1 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).

C 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 6. 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 [30] 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.

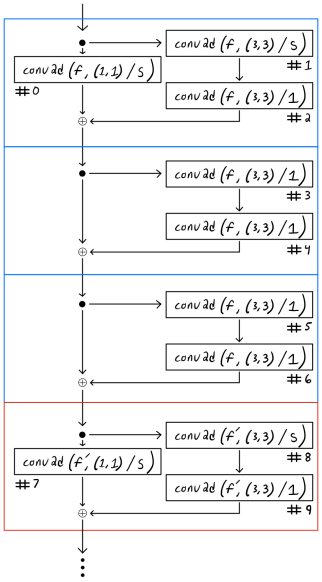


Fig. 6. ResNet Illustration.

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.

D 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 for the transformation 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 6. Network architectures for CIFAR-10.

Name	Conv4 [11]	ResNet-20 [22]	ResNet-110 [22]	WideResNet-20
Conv Layers	64, 64, M	16	16	64
	128, 128, M	3x[16, 16]	18x[16, 16]	3x[64, 64]
		3x[32, 32]	18x[32, 32]	3x[128, 128]
		3x[64, 64]	18x[64, 64]	3x[256, 256]
FC Layers	256, 256, 10	A, 10	A, 10	A, 10

Table 7. Network architectures for ImageNet.

Name	VGG-19 [49]	ResNet-50 [22]	ResNet-152 [22]	WideResNet-50
Conv	2x64, M	64, M	64, M	128, M
Layers	2x128, M	3x[64, 64, 256]	3x[64, 64, 256]	3x[128, 128, 512]
	4x256, M	4x[128, 128, 512]	8x[128, 128, 512]	4x[256, 256, 1024]
	4x512, M	6x[256, 256, 1024]	36x[256, 256, 1024]	6x[512, 512, 2048]
	4x512, M	3x[512, 512, 2048]	3x[512, 512, 2048]	3x[1024, 1024, 4096]
FC Layers	2x4096, 1000	A, 1000	A, 1000	A, 1000

E NETWORK ARCHITECTURES AND TRAINING DETAILS

In this section, 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 6. 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 [22]. 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 7. 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 [22] 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 [33] 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.

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 [22]. 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 [22]: 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 [21]. 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 [49]. 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 [41], 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 [54] with rate 50% for the first two dense layers. We use default TensorFlow weight

initializations (`glorot-uniform` [15] 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 [23]. 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 [22]. 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 [21]. 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. For ResNet-50-PCN0 we use the input-based transformation for all convolutional layers in stage four with the following effective dimensionalities: We do not compress the first two convolutions which take as input activations from the previous stage; We use an effective dimensionality of 1024 for the first convolution in blocks two and three; All others use an effective dimensionality of 432. For Resnet-50-PCN1 the effective dimensionalities are similar: We use 768 and 320 respectively. For ResNet-50-PCN2 we do compress the first two convolutions using an effective dimensionality of 512. Other layers are compressed as above using 768 and 256 as the effective dimensionalities respectively. Finally, for ResNet-50-PCN3 we compress stage four as in PCN2 using 256 for the first two convolutions, 512 for the first convolution in blocks two and three, and 256 elsewhere. We also compress the convolutions in stage three as follows: The first convolution of blocks two through six uses an effective dimensionality of 256. All other convolutions use an effective dimensionality of 128. 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.