# PCNS: PCA Network Shrinking

Research draft

P. Rolet

Sep 29, 2022

**Abstract**

This is a work in progress to explore an idea of algorithm that would significantly reduce both the size and the training time of deep learning models. Other shrinking methods such as distillation (e.g. Gou et. al., 2021) and weight pruning (e.g. Renda et al., 2020) require a full training to be done beforehand and do not offer proven adjustable performance guarantees. If this idea, called PCA network shrinking or *PCNS*, is successful, the expected benefits are:
a/ to minimize network size while maintaining good performance;
b/ to significantly reduce training time;
c/ to be faster and simpler to operate than other shrinking methods;
d/ to provide performance bounds wrt. the original network.

Outline of the PCNS algorithm on a fully-connected layer (oversimplified)
Given the output matrix of the layer on a large batch, i.e. its feature map (with features as columns), consider its rows (each row corresponds to the output of one neuron over the batch) and approximate them by performing PCA and retaining the largest eigenvectors; change the layer's weights so that applying the forward pass on the batch generates approximated eigenvectors instead of the transposed outputs (this is done via ordinary least squares and this is what actually reduces the size of the network); then change the next layer's weights so that for each former output the corresponding linear combination of the new outputs is computed during future forward passes.

## 1 Introduction

The cost of training deep models is often high, and many look for ways to reduce it. Reducing the model size is also a common objective–for faster

and cheaper inference, and ability to fit the model in limited-memory devices. However, to the best of the author's knowledge, the goals are usually considered separately: methods to reduce training time rarely focus on reducing the model size during training[1], and conversely methods to shrink model size usually rely on fully training the model first; reducing training time is not their stated goal.

This, combined with the growing availability of computing power tailored to deep learning, has led to model size reduction research being seen as application-specific, restricted to mobile or embedded computing. But if there was a way to shrink models during training, it may potentially make training much faster—e.g. being able to halve the size of the layers during training while maintaining approximately the same performance, would divide by 4 the remaining training time. Additionally, the ability to reduce the model size during training paves the way for deeper architectures with less parameters[2].

This work explores a new way to reduce model size called *PCA network shrinking* or *PCNS* that can be applied soundly during training, and as such can significantly reduce training time. As mentioned in the abstract, other deep learning model shrinking methods such as distillation (e.g. Gou, Jianping and Yu, Baosheng and Maybank, Stephen J and Tao, Dacheng, 2021) and weight pruning (e.g. Renda, Alex and Frankle, Jonathan and Carbin, Michael, 2020) require a full training to be done beforehand and do not offer provable adjustable performance guarantees. If the PCNS algorithm is successful, the expected benefits are:

- to minimize network size while maintaining good performance;
- to significantly reduce training time;
- to be faster and simpler to operate than other shrinking methods;
- to provide performance bounds wrt. the original network.

The idea behind this algorithm stems from *redundancy* that can be observed inside network layers, explained in the subsection below.

[1] An exception would be using sparse-inducing penalties in the training loss, e.g. proportional to the L0-norm (e.g. Louizos, Christos and Welling, Max and Kingma, Diederik P, 2017), but to the best of the author's knowledge, although this kind of approach has existed for a while, it is not used in practice when training deep networks. This is not to say that these techniques don't warrant further study—however, it is not the focus of this work.

[2] Another informal, intuitive justification for this work is the principle that learning something using a smaller description can mean learning it "better" (the use of "Minimum Description Length" approaches in machine learning likely stem from a similar intuition)

## 1.1 Redundancy

We can reduce the network during the training if we observe *redundancy* in a layer, i.e. groups of neurons performing similar computations. The simplest example would be 2 neurons having the exact same weights–in this case it would make sense to remove one. Another basic example would be a neuron that outputs 0 almost all the time: removing it would not change the network's output. A neuron outputting almost always a value close to 1, which may occur with saturating activation functions, can also be accounted for by the bias of the layer.

The PCNS algorithm is an attempt to generalize these observations and to remove redundancy during and after the training process, shrinking the network in the process.

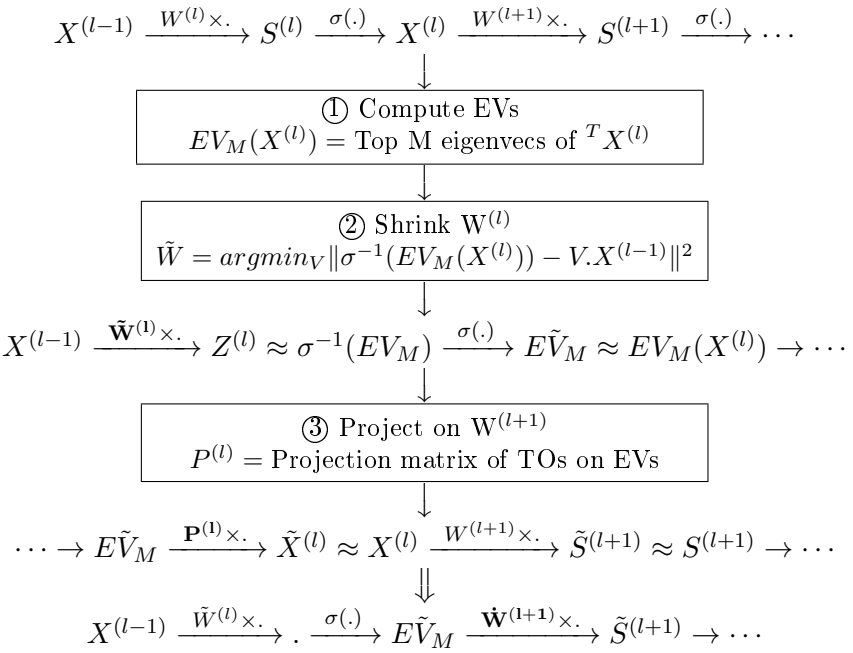


Figure 1: Steps of the PCA newtork shrinking algorithm
Notations used in the schema are listed in section 1.4

### 1.2 Intuition behind the algorithm

A direct approach to eliminate redundancy would be to look into directly into weights matrices looking for neurons with similar weights. However, limiting redundancy elimination to neurons with almost the same weights does not account for the input distribution in the approximation. Even if the neuron weights are not the same, we consider them redundant if they compute similar outputs on most possible inputs.

Therefore, we will spot neuron redundancies by comparing neuron outputs. An underlying assumption of the algorithm is that if two neurons generate almost the same ouput on a large input batch, they generate almost always the same output *in general*: one of them can be removed. The algorithm will therefore focus on *transposed ouputs*. A transposed output is the output of a single neuron on an entire input batch (whereas an output usually refers to the output of all neurons on a single input).

The second idea in PCNS is to generalize "removing neurons with similar outputs". Note that the existence of two neurons with the same outputs on a batch means the rank of the transposed outputs matrix is smaller by one (than if the neurons were not generating the same outputs). This leads to a generalization embodied by using PCA on the transposed outputs : approximating them by projecting them on a basis made of the largest eigenvectors. We then would like to use these eigenvectors in the network.

Since transposed outputs are approximated well by linear combinations of the eigenvectors, the third idea is to change the weight matrix of the layer so that instead of generating the transposed outputs, it generates those eigenvectors. Then, the fourth idea is to use those linear combinations in the next layer to make it so that the next layer's inputs are approximated versions of the former inputs (approximated via aforementioned linear combinations).

The main steps of the algorithm are schematized in fig. 1. Those steps are explained in more formal detail in section 2 (many important considerations, e.g. $\sigma^{-1}$ may not be defined, will also be adressed in sections 2 and 3).

The weight matrix of the layer in which we were looking to remove redundancies has been replaced with a new weight matrix whose number of rows (i.e. new number of neurons) is the number of top eigenvectors with which we approximate the transposed outputs—potentially much smaller than the number of transposed outputs themselves (i.e. the initial number of neurons in the layer).

### 1.3 Structure

This work is currently in progress. It is layed out as follows:

- section 2 explains the PCNS algorithm in more details;
- section 3 outlines key factors for soundness and efficiency of PCNS;
- sections 4 (not fully written yet) shows how PCNS runs on a toy network (on the MNIST dataset);
- section 5 (not fully written yet) expands on the algorithm's potential benefits and caveats;
- section 6 explains next steps to complete this work;

### 1.4 Notations

This work is interested in a fully-connected neural network of $L$ layers, such that:

$$\forall l \in [1, L] \begin{cases} S^{(l)} = W^{(l)}.X^{(l-1)} \\ X^{(l)} = \sigma(S^{(l)}) \text{ performed per coordinate} \end{cases}$$

using the following conventions:

- uppercase letters are used to denote matrices; uppercase $X$ is used to denote a batch of input vectors; $X^{(0)}...X^{(L-1)}$ are input batches of size $B$ for layers 1 to L, and $X^{(L)}$ is the output of layer L;
- subscript index is used to denote a column vector, therefore a matrix is a sequence of column vectors, e.g. $X^{(j)} = X_1^{(j)}...X_B^{(j)}$ (consequently, $X_i^{(j)} = (X_{i1}^{(j)}...X_{iN_j}^{(j)})$ denotes a single input vector of dimension $N_j$);
- vectors may also be designated with bold lowercacse letters (e.g. $\mathbf{y} = \mathbf{x}^{(l+1)} = X_1^{(l+1)}$), and scalar with non-bold lowercase letters (e.g. $\mathbf{y} = (y_1, \ldots, y_n)$) when appropriate;
- matrix $W^{(l)}$ represents the weights for layer $l$;
- $\sigma$ is the activation function (or rectifier), the $\odot$ operator may be used to remind it is applied per-coordinate to vectors and matrices;
- $S^{(l)}$ denote the pre-rectifier outputs, with capital letters since they are batch of vectors;

- at every layer, every input's last coefficient is 1, so that the last value of a neuron's weight is actually a bias–therefore there is no explicit bias addition operation.

Including the bias in the weight matrix in this manner is only aesthetic : it simplifies notations, but does not semantically alter computations presented below (nor forward/backward propagations).

# 2 Algorithm description

This section describes PCA network shrinking for deep networks of fully-connected layers[3].

Given a partially trained network, *PCNS* creates a shrinked network with smaller layers[4] whose output is a sound approximation of the original network's output. Therefore, its error rate is comparable—depending of course on the efficiency of the approximation.

Training can be resumed with the new shrinked network, and the shrinking operation can be repeated multiple times during training [5]. If the shrinking gains are large, training time will be significantly reduced. This is a major difference w.r.t. distillation / pruning techniques that require full training and sometimes re-training afterwards.

Each hidden layer is shrinked by performing the three steps schematized in fig. 1 (section 1.2). These steps are described in more detail below (layer superscripts may be dropped when obvious from context).

## 2.1 Step 1: Compute eigenvectors of transposed outputs

Compute the output matrix $X^{(l)}$ of hidden layer $l$ on a large[6] batch $\mathcal{B}$ and the *transposed outputs* (TOs) of the matrix—that is, the column vectors of ${}^{T}X^{(l)}$. Transposed output vector $TO_i^{(l)}$ of size $B$ can be understood as the

[3] Adaptations to convolutional kinds of layers (e.g. convolutional) seem clearly possible and will be studied in a next step (TODO rewrite this)

[4] how much smaller depends on the layer redundancy introduced in 1.1. This is discussed in more detail in section 3.5

[5] The computational complexity of the algorithm is equivalent to a few training epochs on a batch. When to apply it for best results and fast training is discussed in section 3.3

[6] 'Large' here refers to being at least a few times bigger than the number of neurons of the layer. This is discussed more precisely in 3.2

output of a single neuron (neuron $i$ in layer $l$) over the batch, as shown below.

$$\sigma \odot \begin{pmatrix} - & W_1 & - \\ & \vdots & \\ - & W_N & - \end{pmatrix} \begin{pmatrix} | & & | \\ X_1^{(l-1)} & \cdots & X_B^{(l-1)} \\ | & & | \end{pmatrix} \begin{pmatrix} X_{1,1}^{(l)} & \cdots & \\ \vdots & \ddots & \vdots \\ & \cdots & X_{N,B}^{(l)} \end{pmatrix} = \begin{pmatrix} - & TO_1 & - \\ & \vdots & \\ - & TO_n & - \end{pmatrix} \tag{1}$$

Then, approximate the set of the transposed output vectors (TOs) using PCA, that is using the eigenvectors (EVs) of the experiment matrix of the TOs. The first $M$ eigenvectors with the largest eigenvalues are selected such as the sum of the $M$ eigenvalues matches a desired variance threshold[7] (thus $M$ may be different for each layer):

$$\begin{aligned} TO_1 &\approx \hat{TO} + \sum_1^M \lambda_{1i} EV_i \\ &\vdots \\ TO_N &\approx \hat{TO} + \sum_1^M \lambda_{Ni} EV_i \end{aligned} \tag{2}$$

where $\hat{TO} = \sum_i TO_i$ is the mean vector of transposed outputs and $\lambda_{ji} =< TO_j | EV_i >$, dot product of $TO_j$ and $EV_i$, is the $i$ th coefficient of the projection of $TO_j$ on the eigenvectors.

### 2.2 Step 2: Shrink the weight matrix

In the forward pass operation $\sigma(W.X)$, replace $W$ by the *PCA-shrinked weight matrix* $\tilde{W}$ such that multiplying $\tilde{W}$ by $X$ and applying $\sigma$ yields approximated eigenvectors rather than transposed outputs. Compute $\tilde{W}$ by finding the least squares solution to the linear system below for each eigenvector $EV_i$ as well as for $\hat{TO}$.

$$\tilde{W}_i.X = \sigma^{-1}(EV_i) \tag{3}$$

where we solve for $\tilde{W}_i$

Note that $\sigma^{-1}$, the inverse of $\sigma$, may not be defined or directly applicable depending on the bijectivity and domain of $\sigma$. For some activation functions there is no issue, e.g. *LeakyReLU* can be used directly (bijective with domain spanning $\mathbb{R}$) and *tanh* and *hardtanh* only need a minor adaptation

[7] TODO

for being applied on $\hat{TO}$ (since the EVs have there values in (-1,1) and both functions are bijective on the (-1,1) domain). For most other common activation functions the issue can also be overcome, see section 3.4. In this work, it is assumed LeakyReLU is used unless otherwise specified.

$$\sigma\odot \begin{pmatrix} - & \tilde{W}_1 & - \\ & \vdots & \\ - & \tilde{W}_M & - \\ - & \tilde{W}_{\hat{TO}} & - \end{pmatrix} \begin{pmatrix} | & & | \\ X_1 & \cdots & X_B \\ | & & | \end{pmatrix} \begin{pmatrix} - & \tilde{EV}_1 & - \\ & \vdots & \\ - & \tilde{EV}_M & - \\ - & \tilde{\hat{TO}} & - \end{pmatrix} \tag{4}$$

Replacing $W$ by $\tilde{W}$ reduces the layer size (i.e. the number of rows) from $N$ to $M+1$. It also completely changes the layers' output on batch $\mathcal{B}$, since it yields approximated EVs (noted $\tilde{EV}_i$) rather than TOs, but each TO can be approximated by a linear combination of the EVs. A core assumption of PCNS is that the linear combination of EVs approximating $TO_i$ (i.e. the output of neuron $i$ on batch $B$) can be a good approximation of neuron $i$'s output *in general*—that is, on other batches than $B$.

### 2.3 Step 3: Project approximation on next layer

Replace the weight matrix $W^{(l+1)}$ of the following layer by $\dot{W}^{(l+1)} = W^{(l+1)}.P^{(l)}$ where $P^{(l)}$ is the projection matrix of the TOs on the EVs—therefore of dimension $(N_l, M+1)$. The resulting dimension of $\dot{W}^{(l+1)}$ is then $(N_{l+1}, M+1)$. By doing this, $\tilde{\mathbf{s}}^{(l+1)}$, the pre-$\sigma$ output of layer $l+1$, will be a sound approximation[8] of $\mathbf{s}^{(l+1)}$, the pre-$\sigma$ output before transforming layers with the PCNS algorithm, as schematised below.

[8] The overall efficiency of PCNS relies on the soundness of this approximation, discussed in section 3

$$\begin{pmatrix} | \\ \mathbf{x}^{(l-1)} \\ | \end{pmatrix}$$
$$\sigma \odot \begin{pmatrix} - & W_1^{(l)} & - \\ & \vdots & \\ - & W_{N_l}^{(l)} & - \end{pmatrix} \quad \begin{pmatrix} x_1^{(l)} = y_1 \\ \vdots \\ y_{N_l} \end{pmatrix} \tag{5}$$
$$\begin{pmatrix} - & W_1^{(l+1)} & - \\ & \vdots & \\ - & W_{N_{l+1}}^{(l+1)} & - \end{pmatrix} \quad \begin{pmatrix} | \\ \mathbf{s}^{(l+1)} \\ | \end{pmatrix}$$

Before: Given a new input $\mathbf{x}^{(l-1)}$ to layer $l$, the forward pass yields $\mathbf{y} = \mathbf{x}^{(l)}$, then the following layer's pass is applied via $W^{(l+1)}$ to yield $s^{(l+1)}$

$$\begin{pmatrix} | \\ \mathbf{x}^{(l-1)} \\ | \end{pmatrix}$$
$$\sigma \odot \begin{pmatrix} - & \tilde{W}_1^{(l)} & - \\ & \vdots & \\ - & \tilde{W}_{M+1}^{(l)} & - \end{pmatrix} \quad \begin{pmatrix} e_1 \\ \vdots \\ e_{M+1} \end{pmatrix}$$
$$\dot{W}^{(l+1)} \begin{cases} P^{(l)} = \begin{pmatrix} \lambda_{11} & \cdots & \\ \vdots & \ddots & \vdots \\ & \cdots & \lambda_{N_l M+1} \end{pmatrix} & \begin{pmatrix} \sum_i \lambda_{1i} e_i \approx y_1 \\ \vdots \\ \sum_i \lambda_{N_l i} e_i \approx y_{N_l} \end{pmatrix} \\ W^{(l+1)} = \begin{pmatrix} - & W_1^{(l+1)} & - \\ & \vdots & \\ - & W_{N_{l+1}}^{(l+1)} & - \end{pmatrix} & \begin{pmatrix} | \\ \tilde{\mathbf{s}}^{(l+1)} \approx \mathbf{s}^{(l+1)} \\ | \end{pmatrix} \end{cases} \tag{6}$$

After: $W^{(l)}$ is changed to $\tilde{W}^{(l)}$; the new output of layer $l$ is noted $\mathbf{e}$. Projecting using $P^{(l)}$ we fall back on approximated $y$ then using $W^{(l+1)}$ continues forward propagation: $\dot{W}^{(l+1)} = P^{(l)}.W^{(l+1)}$ can be soundly substituted to $W^{(l+1)}$

Repeating the operation on all the *hidden* layers shrinks the whole network. It makes no sense to apply PCNS to the last layer, as explained in section 3.6. Furthermore, the order in which layers are shrinked matters, as discussed in section 3.7.

# 3 Key factors for soundness and efficiency of PCNS

## 3.1 Soundness of approximation

Applying PCNS on layer $l$ works if the outputs of layer $l+1$ after changing $W^{(l)}$ and $W^{(l+1)}$ are close to what they were before. This requires the 3 approximations below to perform well:

- the linear combination of eigenvectors of step 2 must approximate the transposed outputs well;
- the new weights of layer $l$ as computed by step 3, must generate (via the forward pass) good approximations of the eigenvectors on the PCNS batch $\mathcal{B}$;
- the fact that the approximation is good on batch $\mathcal{B}$ must translate to it being good in general.

### 3.1.1 TODO The first one is ensured by the variance threshold discussed TODO, the second requires analysis, the third one is why we want a large batch see below section 3.2

## 3.2 Batch size

The PCNS batch size will determine the quality of the approximation (notably the 2nd & 3rd ones in section 3.1). A small batch size will imply a good approximation of the TO by step 2, but a poor generalization to other batches (similarly to overfitting). Therefore, a batch as large as possible ensures the best possible generalization–just as a training set as large as possible is often desirable. The limitation is the computational cost. The batch should be at least a few times bigger than the layer size, otherwise the risk of overfitting would be too great. Apart from that, the goal of PCNS being to reduce both the model size and training time, the chosen batch size would depend on the overall training methodology–e.g. if initial training is meant to run over tens thousands of epochs, choosing a PCNS batch size of 10 times the layer size and running PCNS every tens of thousands of epochs will be suitable since the pcns cost would be a small fraction of the training cost. Deciding precisely how the size of the batch impacts the precision of the algorithm is a question to be tackled.

### 3.3 Algorithm usage during training & complexity

The most costly step of the algorithm is the least square computations. The complexity of those are $O(N_{l+1}*N_l*B)$ and since B is supposed to be a few times bigger than $N_l$, this amounts to a cost of a few training epochs. Therefore, applying the algorithm every few epochs * 10 would ensure that its cost stays minimal as compared to the global cost of training.

### 3.4 Adapting PCNS to various activation functions

1. **TODO** Explain how bijective functions can be handled using a shift, and plain relu can be handled by decomposing into positive and negative components of eigenvectors.

### 3.5 In-layer redundancy and expected shrinking efficiency

The efficiency of PCNS relies on how much neurons are redundant, i.e. how much separate groups of neuron perform similar computations, as introduced in sec. 1.1. This paragraph will explain:

- Potential effectiveness of PCNS can be measured by checking the number of required eigenvectors to reach a given variance
- Looking at transposed outputs rather than weights ensures that even if the same thing is computed in a different fashion across multiple layers, the redundancy can still be removed

### 3.6 Last layer not concerned

TODO rewrite

The operation can be performed on all *hidden* layers, therefore shrinking the whole network except for the last layer. PCNS on layer $l$ operates on weights of both $l$ and $l+1$ and the intermediate activations (the ones in between the 2 layers) have no direct relationship–therefore PCNS cannot be straightforwardly applied to the last layer. This operation should a priori not include the last layer, since its output is the final output which usually has a precise semantic–e.g. logprobabilities for multiclass classification. PCNS would mess with this semantic. However, note that the weights of the last layer will be changed by performing PCNS on the second-to-last layer, since PCNS on a layer changes not only weights of the layer itself, but also those of the following layer.

### 3.7 Shrinking order matters

TODO rewrite

PCNS on the whole network should be performed in descending order starting with the second-to-last layer. Indeed, PCNS reduces the size of a neural layer by replacing the computation of the outputs by the computation of 'approximate eigenvectors' of the outputs. The approximation is made by computing eigenvectors of a batch of outputs (transposed see below), corresponding to a batch of inputs. These batches for the whole network have been computed via an initial forward pass. If we were to perform PCNS on layer l-1 before performing it on layer l, TBD

## 4 Example: running PCNS on a toy network

This section will illustrate PCNS on a toy network trained on MNIST

TODO Work in progress

- Train a 2-layers fully connected network on MNIST
- Apply PCNS on the first layer, illustrate and explain what happens
- Apply PCNS on the second layer and show the performance is still quite good
- Train for a few more epochs and show the performance improves, for a fraction of the training time since the shrinked network is faster to train

## 5 Benefits and caveats

This section will explain how the algorithm delivers the benefits mentioned in introduction

### 5.1 Benefits

#### 5.1.1 Network size minimization

#### 5.1.2 Reduction in training time

#### 5.1.3 Simplicity of operation

#### 5.1.4 Performance bounds

### 5.2 Caveats

#### 5.2.1 When does removing redundancy make sense

#### 5.2.2 Internal covariate shift

*

## 6 Next steps for this work

The next steps that are being taken to validate the PCNS algorithm are:

- Adapt the algorithm to convolutional networks (which would most likely translate in reducing the number of channels, not the size of the kernel)
- Check the algorithm is not redundant with other deep network optimization techniques (e.g. batch normalization, dropout, resnets) and can be seamlessly integrated with them;
- Measure redundancy (cf chap TODO) of various classical deep networks on standard problems (e.g. AlexNet on ImageNet) to see if there is one that would most benefit from being shrinked by PCNS
- Test PCNS on such a network and complete this work with the results.

  ** TODO say the following at the right place
- next steps for this work: determine precisely the batch size, etc.
- Choose variance threshold
- Estimate network reduction, ensure a proper bound (can we measure if it worked / how well it worked?)

# 7 Citations

Gou, Jianping and Yu, Baosheng and Maybank, Stephen J and Tao, Dacheng (2021). *Knowledge distillation: A survey*, Springer.
Louizos, Christos and Welling, Max and Kingma, Diederik P (2017). *Learning sparse neural networks through L_ 0 regularization*, arXiv preprint arXiv:1712.01312.
Renda, Alex and Frankle, Jonathan and Carbin, Michael (2020). *Comparing rewinding and fine-tuning in neural network pruning*, arXiv preprint arXiv:2003.02389.