PCNS: Principal Component 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 using them can be costly in terms of human and/or computer time. If this idea, called Principal Component 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;

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 according to the linear combinations of eigenvectors approximating the transposed outputs.

1 Introduction

The cost of training deep models is often high, therefore reducing this cost is a sought-after goal for deep learning practitioners and researchers. 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 *Principal Component Network Shrinking* or *PCNS* that can be applied soundly during training, and as such can significantly reduce training time. 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; they also usually require further training and empirical parameter adjustments. 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;

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.

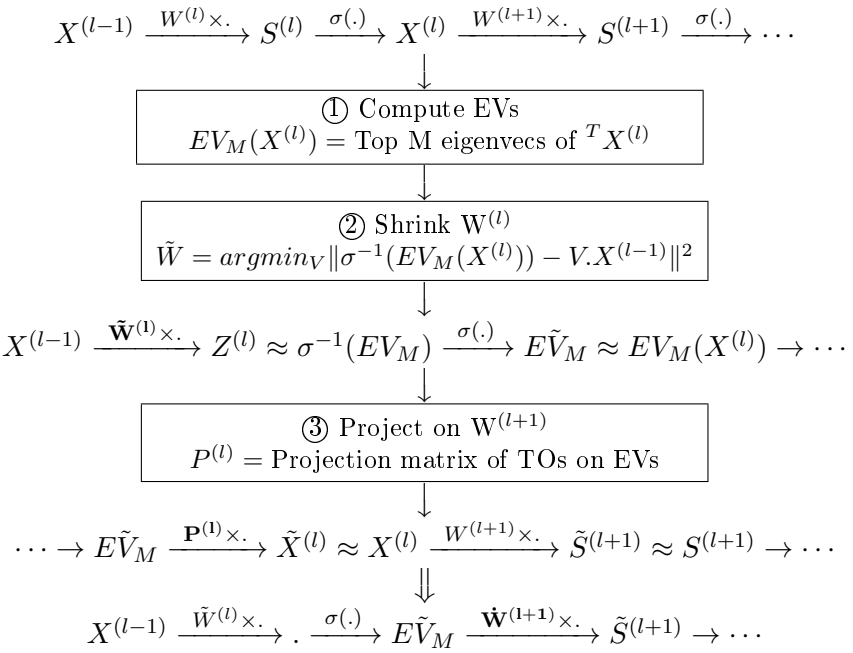

Figure 1: Steps of the Principal Component Newtork Shrinking algorithm
Notations used in the schema are listed in section 1.4

1.2 Intuition behind the algorithm

A direct approach to eliminate redundancy would be to look 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 its 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. the fact that σ^{-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) illustrates the PCNS algorithm on a toy network trained on the MNIST dataset;
- section 5 (not fully written yet) explains the benefits mentioned in the abstract and introduction, and detail important considerations 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; ${}^TS$ denotes the transpose of matrix S; 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 lowercase 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;
- σ 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 a capital S 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 Principal Component 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 layers are mentioned in section 6

[4] how much smaller depends on the layer redundancy introduced in 1.1. This is discussed in more detail in section 3.1

[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 5.1.2

[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 so that 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 $\hat{TO} = \frac{1}{N}\sum_i TO_i$ 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 σ 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 σ^{-1}, the inverse of σ, may not be defined or directly applicable depending on the bijectivity and domain of σ. 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 for being

[7] The variance threshold is discussed in section 3.3

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 5.2.3. 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, given a new input vector $\mathbf{x}^{(l-1)}$ going through shrinked layers l and $l+1$, the pre-σ output of layer $l+1$ noted $\tilde{\mathbf{s}}^{(l+1)}$ will be a sound approximation[8] of $\mathbf{s}^{(l+1)}$, the pre-σ output before transforming layers with the PCNS algorithm, as schematised below.

As mentioned in the notations, to simplify equations the bias is included in inputs as a 1 at the end. This is accounted here by adding a column of zeros and a line of zeros at the end of $P^{(l)}$, with a single one in the bottom right corner of the matrix. Thus, the approximated $\mathbf{y}$ will have a 1 at the end, coming from the 1 at the end of $\mathbf{e}$ (this is not mentioned in schemas, equations and the rest of this work for readability—it does not impact in any way the reasoning around the algorithm).

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

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

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 $\mathbf{s}^{(l+1)}$

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

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 5.2.4. Furthermore, the order in which layers are shrinked matters, as discussed in section 5.2.5.

3 Key factors for soundness and efficiency of PCNS

PCNS relies on two main assumptions to perform well:

1. [Redundancy] There is redundancy in the network computations[9];
2. [Sound approximation] The various approximations that PCNS performs are good enough so that the shrinked network approximates well the original network's output.

Assumption 2 can be subdivided in the 3 approximations below—considering an application of PCNS on layer l works well if the outputs of layer $l+1$ after changing $W^{(l)}$ and $W^{(l+1)}$ are close to what they were before:

a. the fact that the approximation is good on batch $\mathcal{B}$ must translate to it being good in general;

b. the linear combination of eigenvectors of step 1 of PCNS must approximate the transposed outputs well;

c. the new weights of layer l as computed by step 2, must generate (via the forward pass) good approximations of the eigenvectors on the PCNS batch $\mathcal{B}$;

This section explains how those assumptions impact the way the algorithm works. Admittedly, the PCNS algorithm is at this stage theoretical; its potential benefits need experiments to be confirmed in practice and to justify further research.

3.1 Redundancy and expected shrinking efficiency

Assumption 1 states that 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.

In PCNS, this redundancy can be measured at each layer by checking the number of required eigenvectors to reach a given variance—e.g. on a 1000 neurons layer, if using the 100 first eigenvectors accounts for 99.99% of the variance[10], i.e. an error under 10^{-4}, then the shrinking algorithm will perform well, whereas if say more than 800 of them are needed to reach 90% variance then PCNS probably won't be effective.

[9] intuitively, this assumption is probably not specific to PCNS. In its most general form, assuming the network has no redundancy in any of its computations is likely (although not certain) to impact negatively any model size reduction algorithm

[10] Early experiments on toy problems suggest that these kind of values (a tenth of the layer size for 10^{-4} variance) may be common once the network has been partly trained

It is an interesting property of PCNS to be able to know in advance if it is worth running it by looking at the distribution of the eigenvalues. As training goes on, redundancy is likely to appear: the practitioner can wait until it becomes worth it to actually run the algorithm in full and change the network.

Note that looking at transposed outputs rather than weights ensures that even if the same values are computed in a different fashion across multiple layers, the redundancy can still be removed starting from the last layers (since ultimately similar outputs must appear at some point in later layers).

3.2 Sound approximation and batch size

The batch size used to run PCNS will impact the quality of the approximations.

A small batch size will allow for a good approximation of the TO by step 1 of PCNS, and of the EVs 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. If it results in poor approximation quality in steps 1 and 2, it will be observable by the fact that a high number of eigenvectors are necessary to account for enough variance—showing that the network is not that redundant and that model size reduction via PCNS will not be effective (see section 3.1 above).

The limiting factor for the batch size is of course the computational cost. The batch should be at least a few times (e.g. 5-10 times) bigger than the layer size, otherwise the risk of overfitting would be too great. Further work will be done on the calculations of the theoretical approximation error to determine more precisely the minimal size of the batch—that is, a theoretical estimation of $P(\epsilon, B|\epsilon_0)$, the probability that the shrinked network errs less than ϵ on a new batch, given that it errs less than ϵ_0 on batch $\mathcal{B}$ of size B.

The goal of PCNS being to reduce both the model size and training time, the chosen batch size would also 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 thousands of epochs will be suitable since it will limit the PCNS cost to a small fraction of the training cost.

3.3 Sound approximation and variance threshold

The chosen variance threshold also impacts the quality of approximation (mostly, assumption 2b): a low variance threshold will provide a better approximation by the eigenvectors, but the model size reduction will not be as effective.

Since many applications and systems already have multiple sources of randomness (e.g. small noise in input), setting the variance here to e.g. half the value of a guess of the system's inherent variance should be good enough.

Basic experiments on toy problems lead to believe that the number of top EVs for limiting the variance to e.g. 10^{-4} can sometimes be enough to divide a layer size by 10. It is expected that observing the cumulative distribution of eigenvalues and estimating the system's intrinsic variance should allow to make an informed choice.

Further work will calculate formal estimates of the impact of the variance threshold in the shrinked network performance.

3.4 Sound approximation and least squares estimation for $\tilde{W}$

When computing $\tilde{W}$ that generates eigenvector approximations in step 2, the least squares efficiency in approximation is also a determining factor (assumption 2c).

When running the algorithm, it is simple to check whether the least squares step performed well, by looking at the L2 norms between the "approximated eigenvectors" generated by $\tilde{W}$, and the initial eigenvectors.

Further work will look at theoretical criteria and methods to guarantee a priori that the least squares solutions will be good enough.

4 Example: running PCNS on a toy network

This section intends to illustrate PCNS on a toy network trained on MNIST

TODO

- 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, and will detail a few important points to consider in order to use the algorithm properly.

5.1 Benefits

Notable benefits wrt other shrinking methods would be 1/ to be able to be used during training and thus reduce training time, and 2/ even used only as a shrinking method, to be less costly and more straightforward to operate.

5.1.1 Lower cost and simplicity of operation

For a direct comparison with other methods, let us consider the case in which PCNS is not performed during training (in that case it does not affect training time)

- Distillation methods require trainings of a number of small models by using the initally trained large model and selecting the one providing the best size/accuracy tradeoff (costing human time and computer time);
- Weight pruning require retraining the model grid-searching on various parameters (pruning %, training rewinding procedure) (costing human time and computer time);
- PCNS can be applied one-shot at the end of training for a fraction of the overall training time cost.

PCNS parameters do not need to be grid-searched—e.g. once the eigenvalue distribution is known, the choice for setting the variance threshold can be made, without need to run the algorithm multiple times. Also, PCNS allows to know the expected size of the shrinking (number of eigenvectors) or whether it will work well (error rate of EVs generated by $\tilde{W}$).

5.1.2 Computational complexity & usage during training

The most costly steps of the algorithm are the eigenvectors and least square solutions 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 ten times "a few epochs" would ensure that its cost stays minimal as compared to the global cost of training.

TODO

- Explain in more detail the computational cost
- Explain when (i.e. after how much training) PCNS should be expected to shrink enough in a way that makes sense
- However, a point is that the cost of checking the number of top eigenvectors required is small wrt the total training time, so an option is just to make this check every once in a while and actually apply PCNS when the reduction is good enough

5.2 Important consideration and caveats

5.2.1 When does removing redundancy make sense

TODO

- Removing redundancy clearly makes sense after training (e.g. removing a neuron always outputing 0 or the same as another neuron)
- Before training, it may not be the case for linear combination approximations (because backprop may decombinate them)
- during training, it probably often makes sense, but find out when and why

5.2.2 Covariate shift (incl. internal covariate shift)

TODO

- Looking at the hidden layer's outputs, their distribution will change during training, so this may somehow impact PCNS
- Would batch normalization solve the issue here as it usually does with internal covariate shift?

- This is not an issue at later stages of training, when the input distribution stabilizes.

5.2.3 Adapting PCNS to various activation functions

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.

5.2.4 Last layer not concerned

The operation can be performed on all *hidden* layers, therefore shrinking the whole network except for the last layer. Shrinking the last layer does not make sense notably if it outputs e.g. class probabilities. Furthermore, PCNS on layer l operates on weights of both l and $l+1$. The last layer has no following layer, it would make no sense that it outputs approximated eigenvectors of the class probabilities.

5.2.5 Layer shrinking order

TODO explain that when applying PCNS

- performing a full forward pass on the batch and then shrinking layer in increasing order cannot work given how the outputs are changed
- it would however work using a descending order
- another option would be to perform single-layer passes
- compare the 2 options

6 Next steps for this work

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

- Provide sound formal estimations for variance threshold & batch size impact on shrinked network performance;
- Provide theoretical criteria and methods to guarantee a priori that the least squares solutions of step 2 will be good;
- Provide performance bounds when applying the algorithm at the end of training

- Adapt the algorithm to convolutional networks (which would most likely result in reducing the number of channels, not the size of the kernel)
  - show how each PCNS step for fully-connected layers translates to convolutional layers
- 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 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 networks and complete this work with the results.

7 Bibliography

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.