CP and Tucker Decomposition for Network Compression

NLA 2022

Team Decomposers

Hai Le
Abdul Aziz Samra
Garsiya Evgeniy
Vladimir Kuzmin
Uyen Vo

22 December | 2022

Problem Statement & Importance (1)

Running deep convolutional neural networks is a challenging task on many devices, as the number of parameters (and FLOPS) is **increasing** dramatically in the recent years.

One of the most time and memory consuming task is the evaluation on convolution layers in the model. **In this project, we try to speed-up different CNN architectures by applying two Tensor Decomposition techniques to compress the kernels of convolution layers.**

Problem Statement & Importance (2)

- Fast neural network is well sought after in both research and industry.
- Modern tasks requires **huge** amount of parameters -> **big** floating point operations, increased memory usage and **lower** speed performance.
- Network compression can help better network performance which can be used in industry tasks as well as in research operations.

Roadmap

1) Training and Validating with the CNNs without decomposition

2) Decompose each Conv2D layer of our CNNs. Rebuild the compressed model

3) Evaluate the CNNs both with and without decomposition

4) Compare the computational speed of with and without decomposition

5) Have a beer and celebrate the end of NLA:}

Data & Architecture (1)

Data: **CIFAR10**

- 60000 32x32 colour images in 10 classes, with 6000 images per class

Network Architecture:

- Resnet18
- Densenet

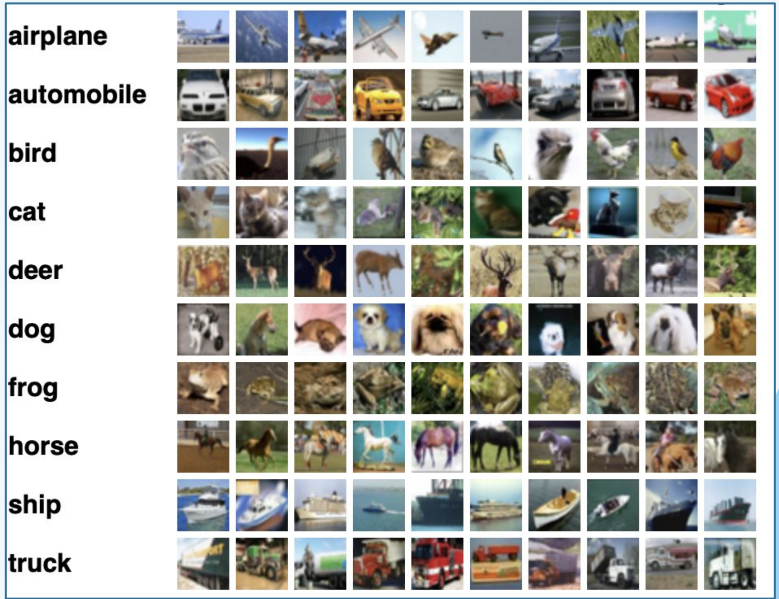
Augmentations:

- **Train transform:** normalize
- **Val transform:** RandomCrop, RandomHorizontalFlip, Normalize

Hyper-parameters:

- **Batch size:** 150
- **Optimizer:** SGD - LR=0.1, Momentum=0.9, weight_decay=5e-4
- **Scheduler:** CosineAnnealingLR - T_Max=150

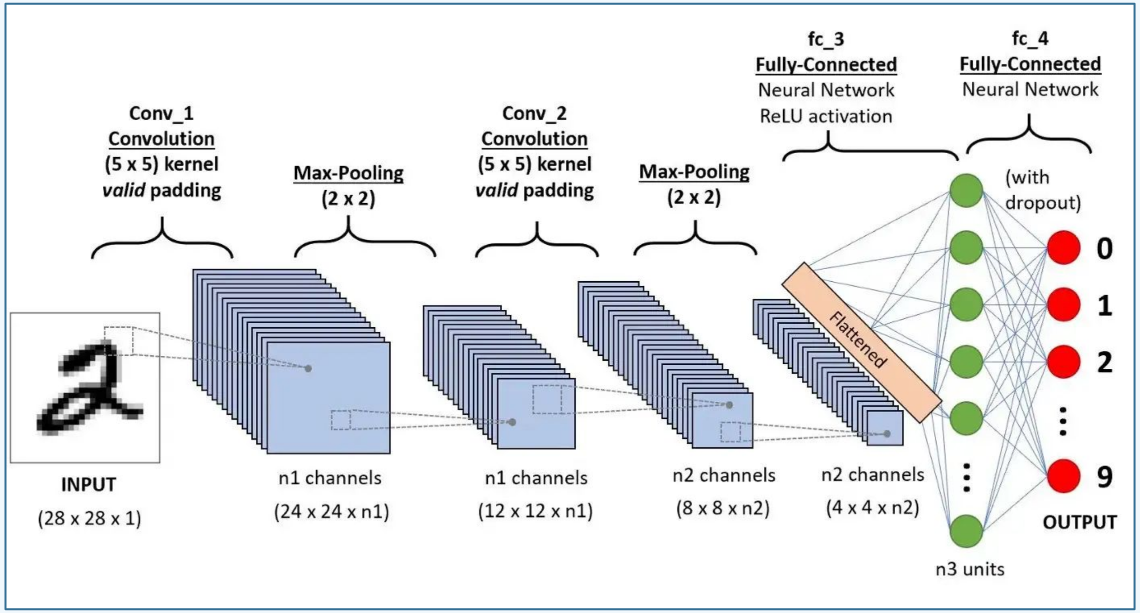
CIFAR10 data examples



Data & Architecture (2)

Convolutional neural networks (CNNS):

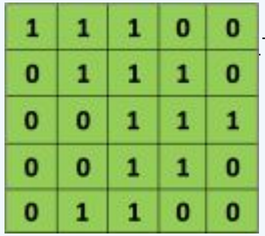
CNN is a basic network architecture constructed from a stack of **convolutional layers**. **Convolutional layers** provide a convolution operation for **tensor** of **size** (batch, channels, width, height) with some **kernel** which is also a tensor.



Data & Architecture (3)

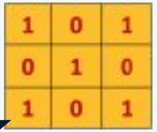
In our case we consider a 3 dimensional **filters** (a stuck of 2D **kernels)** and 3 channel **images**. After we trained a network, we obtain a stuck of trained **weights**, which are tensors and considering CNNs are the values of **kernels**. That is why we speak of **tensor decomposition** considering neural networks.

For example, if we have the **input 2D matrix** in green:



1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

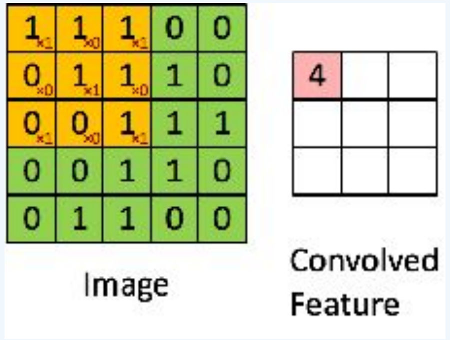
th the **convolution filter**



1	0	1
0	1	0
1	0	1

weights

Each matrix element in the convolution filter is the weight that is being trained. These weights will impact the extracted convolved features as

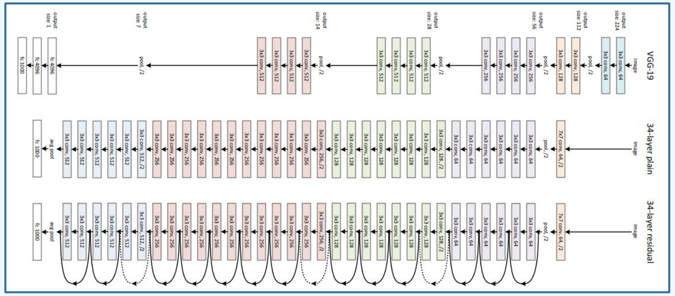


Data & Architecture (4)

Network Architectures:

- ResNet

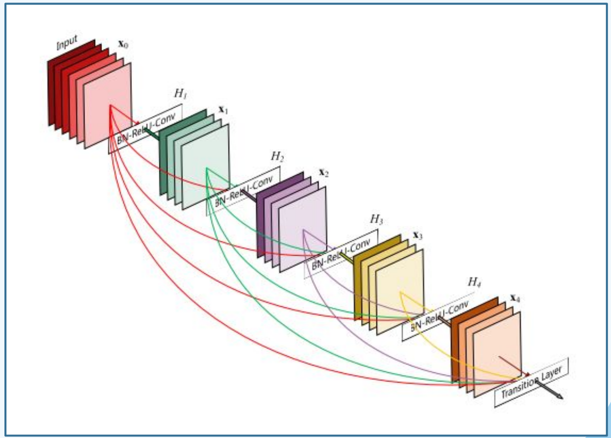
The main idea of this network is to use **Residual Blocks** which can help to solve the problem of the vanishing gradient. Such a blocks are formed with the help of ***skip connections***.



Resnet architecture

- DenseNet

DenseNet architecture

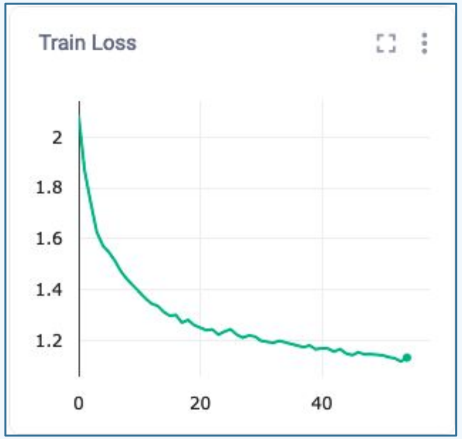


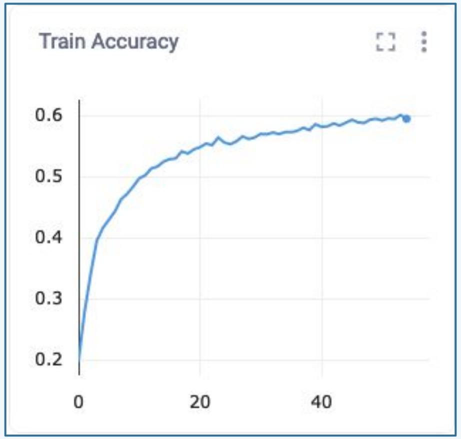
DenseNets require fewer parameters than an equivalent traditional CNN. DenseNets layers are very narrow, and they just add a small set of new feature-maps. Instead of drawing representational power from extremely deep or wide architectures, DenseNets exploit the potential of the network through feature reuse.

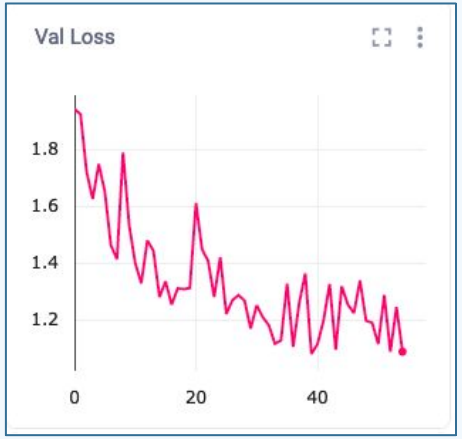
Initial Training and Validation (1)

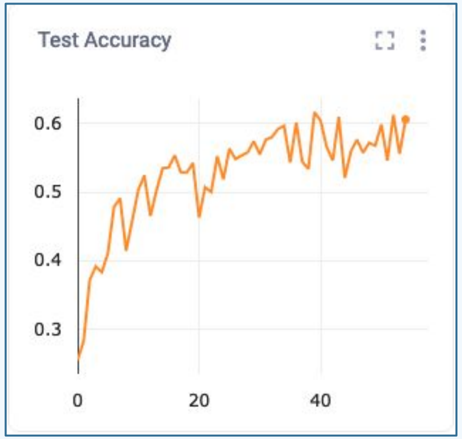
Densenet Training and Validation Plots

Comet-ML Plots: https://www.comet.com/highly0/cp-tucker-decomposition/view/new/panels







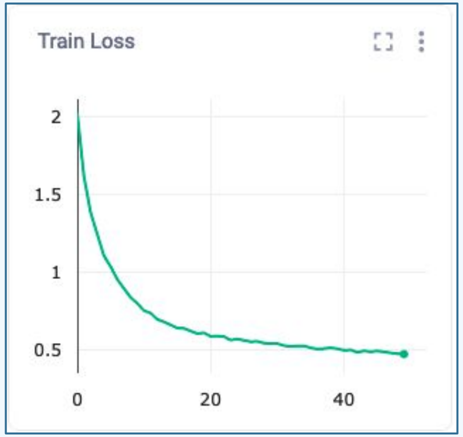


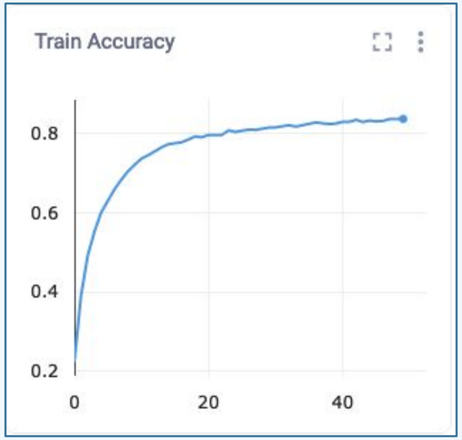
Final Train loss: 1.132 **Final Train Acc**: 0.5949 **Final Val Loss**: 1.088 **Final Val Acc**: 0.6058

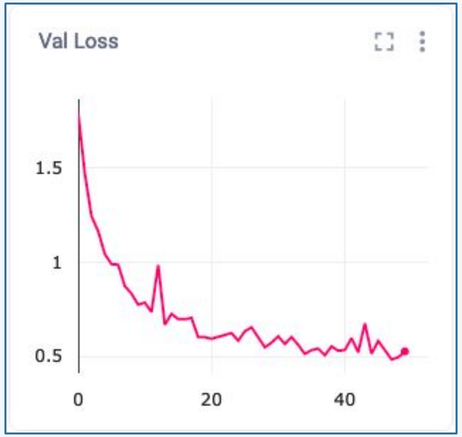
Initial Training and Validation (2)

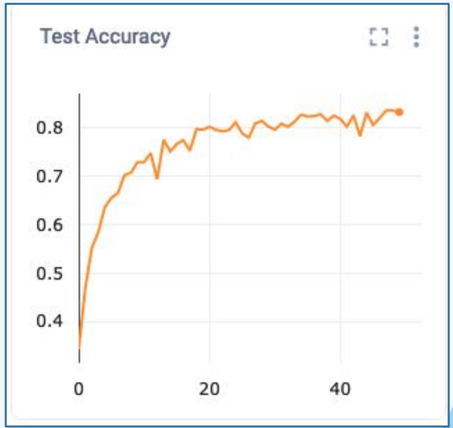
Resnet Training and Validation Plots

Comet-ML Plots: https://www.comet.com/highly0/cp-tucker-decomposition/view/new/panels









Final Train loss: 0.4755

Final Train Acc: 0.8364

Final Val Loss: 0.5257

Final Val Acc: 0.8214

Tensor Introduction - What is a tensor?

Tensor in general is a multidimensional array:

$$A = \left[A\left(i_1, \dots, i_d\right)\right], i_k \in 1, \dots, n_k$$

Any tensor, as well as any matrix, has some dimension:

- **dimensionality** (d) is a number of indices
- **size** (s) is a number of nodes along each axis

$$n_1 \times \dots \times n_d$$

When we speak about vector, we mean d = 1, for matrix d = 2, for **n-dimensional tensor** of n>2 (d = n)

Tensor decompositions Introduction (1)

- Tensor decompositions is a natural way to generalize low-rank approach to multidimensional case.
- Different tensor decompositions exists, e.g. tensor train, canonical polyadic, Tucker decompositions.
- In our research, we will investigate how CP and Tucker decompositions can help to solve stated problem of CNN acceleration and compression

Tensor decompositions Introduction (2) - Tensor rank decomposition

The **rank decomposition** for matrices:

$$A(i_1, i_2) = \sum_{\alpha=1}^{r} U(i_1, \alpha) V(i_2, \alpha)$$

This can be generalized to tensors.

Tensor rank decomposition (**canonical decomposition**):

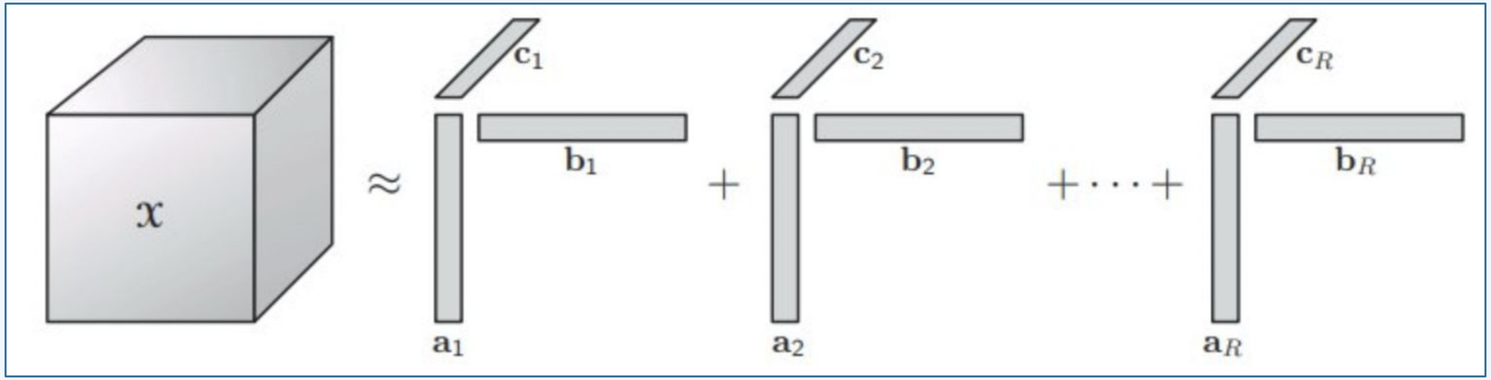
$$A(i_1, \ldots, i_d) = \sum_{\alpha=1}^{R} U_1(i_1, \alpha) \ldots U_d(i_d, \alpha)$$

The minimal possible **R** is called the **canonical rank** of the tensor A.

CP Decomposition

The **canonical polyadic decomposition (CP decomposition)** is the most straightforward way to decompose tensor.

CP decomposition represents tensor as a linear combination of rank **one tensors**:



For example, given a third-order tensor **X**:

$$\chi \approx [[\lambda; A, B, C]] = \sum_{r=1}^{R} \lambda_r a_r \circ b_r \circ c_r$$

Computing the CP Decomposition (1)

Assuming the number of components is fixed, **the alternating least squares (ALS) method** was proposed to compute a CP decomposition.

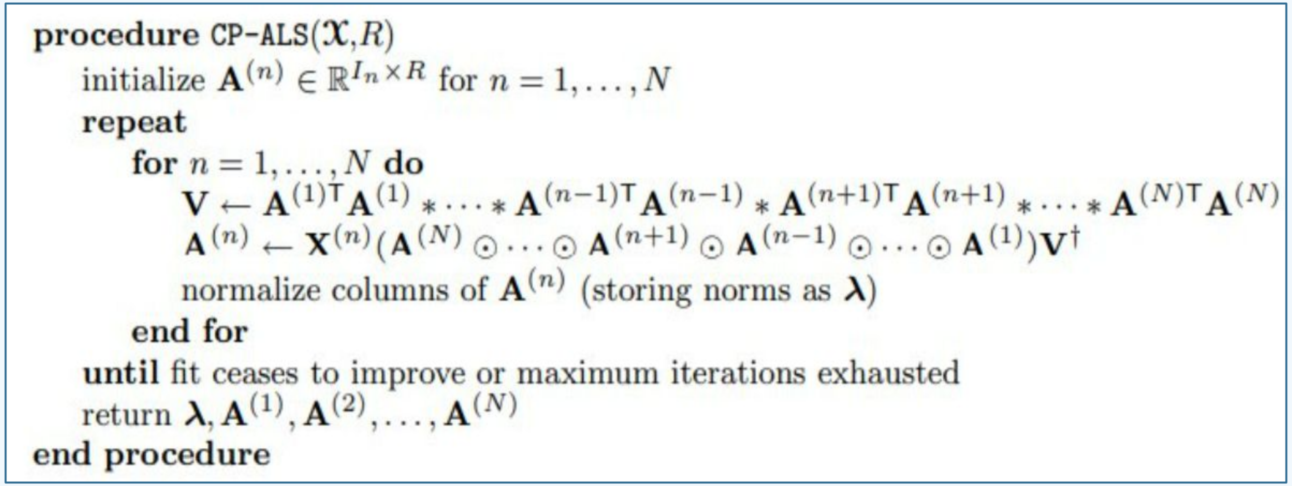
Let $\chi \in \mathbb{R}^{I \times J \times K}$ be a third-order tensor. The goal is to compute a CP decomposition with R components that best approximates χ, i.e. to find: $\min_{\hat{\chi}} \left\| \chi - \hat{\chi} \right\|$

$$with\ \hat{\chi} = \sum_{r=1}^{R} \lambda_r a_r \circ b_r \circ c_r = [[\lambda; A, B, C]]$$

The **ALS algorithm**:

1. Initialize random A, B, C
2. fix B, C, solve least squares for A
3. fix A, C, solve least squares for B
4. fix A, B, solve least squares for C
5. Go to 2 (continues to repeat the entire procedure until some convergence criterion is satisfied)

Computing the CP Decomposition (2)

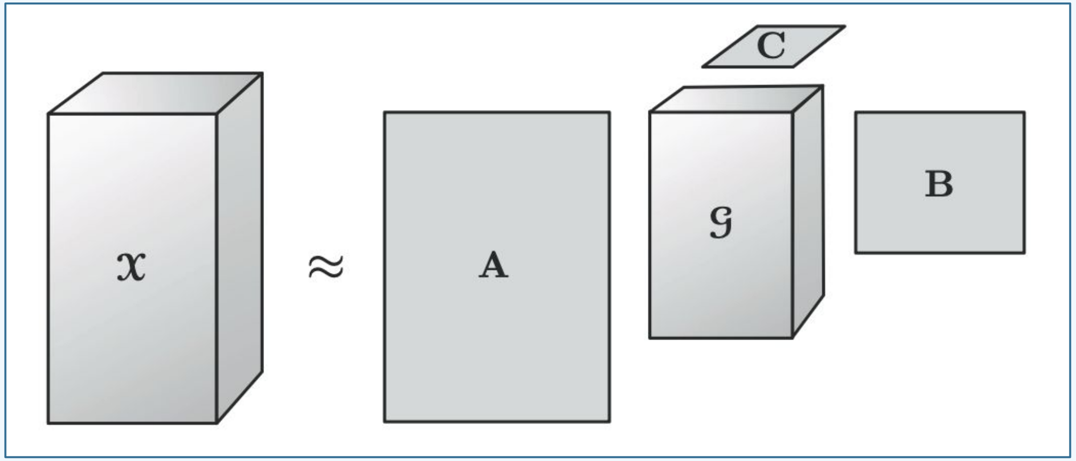


```
procedure CP-ALS($\mathcal{X}$,$R$)
    initialize $\mathbf{A}^{(n)} \in \mathbb{R}^{I_n \times R}$ for $n = 1, \ldots, N$
    repeat
        for $n = 1, \ldots, N$ do
            $\mathbf{V} \leftarrow \mathbf{A}^{(1)\mathsf{T}}\mathbf{A}^{(1)} * \cdots * \mathbf{A}^{(n-1)\mathsf{T}}\mathbf{A}^{(n-1)} * \mathbf{A}^{(n+1)\mathsf{T}}\mathbf{A}^{(n+1)} * \cdots * \mathbf{A}^{(N)\mathsf{T}}\mathbf{A}^{(N)}$
            $\mathbf{A}^{(n)} \leftarrow \mathbf{X}_{(n)}(\mathbf{A}^{(N)} \odot \cdots \odot \mathbf{A}^{(n+1)} \odot \mathbf{A}^{(n-1)} \odot \cdots \odot \mathbf{A}^{(1)})\mathbf{V}^{\dagger}$
            normalize columns of $\mathbf{A}^{(n)}$ (storing norms as $\boldsymbol{\lambda}$)
        end for
    until fit ceases to improve or maximum iterations exhausted
    return $\boldsymbol{\lambda}, \mathbf{A}^{(1)}, \mathbf{A}^{(2)}, \ldots, \mathbf{A}^{(N)}$
end procedure
```

ALS algorithm to compute a CP decomposition with R components for an Nth-order tensor X of size $I_1 \times I_2 \times \cdots \times I_N$.

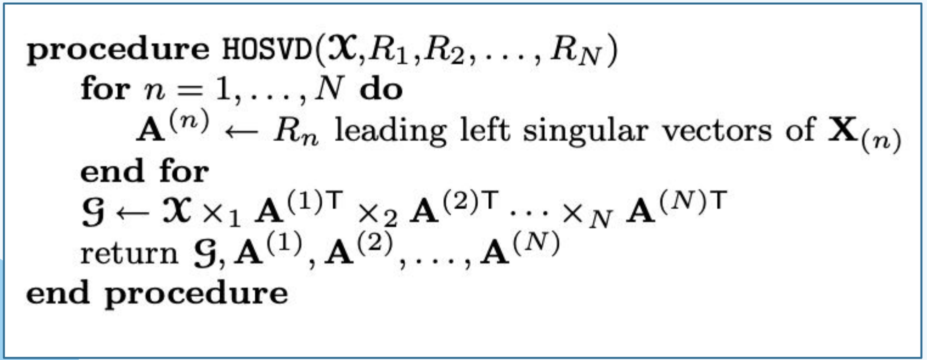
Tucker Decomposition

- The **Tucker** decomposition is a form of higher-order principal component analysis.
- It decomposes a tensor into **a core tensor multiplied (or transformed)** by a matrix along each mode.
- Thus, in the three-way case we have:

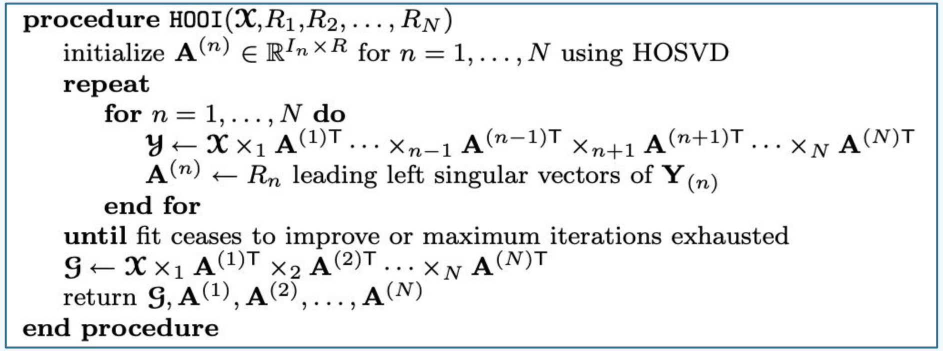


Tucker decomposition

- One of the algorithms for Tucker decomposition in a 3D space is so called the **higher-order SVD (HOSVD)**. The truncated **HOSVD** is not optimal in terms of giving the best fit as measured by the norm of the difference, but it is a good starting point for an iterative **ALS** algorithm.
- The **higher-order orthogonal iteration (HOOI)** is a more efficient technique for calculating the factor matrices for an N th-order tensor.



```
procedure HOSVD($\mathcal{X}$, $R_1$, $R_2$, ..., $R_N$)
    for $n = 1, \dots, N$ do
        $\mathbf{A}^{(n)} \leftarrow R_n$ leading left singular vectors of $\mathbf{X}_{(n)}$
    end for
    $\mathcal{G} \leftarrow \mathcal{X} \times_1 \mathbf{A}^{(1)\mathsf{T}} \times_2 \mathbf{A}^{(2)\mathsf{T}} \cdots \times_N \mathbf{A}^{(N)\mathsf{T}}$
    return $\mathcal{G}, \mathbf{A}^{(1)}, \mathbf{A}^{(2)}, \dots, \mathbf{A}^{(N)}$
end procedure
```



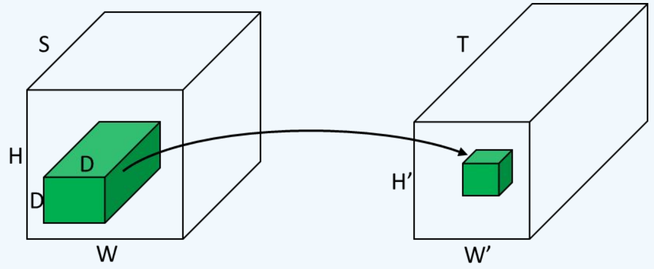
```
procedure HOOI($\mathcal{X}$, $R_1$, $R_2$, ..., $R_N$)
    initialize $\mathbf{A}^{(n)} \in \mathbb{R}^{I_n \times R}$ for $n = 1, \dots, N$ using HOSVD
    repeat
        for $n = 1, \dots, N$ do
            $\mathcal{Y} \leftarrow \mathcal{X} \times_1 \mathbf{A}^{(1)\mathsf{T}} \cdots \times_{n-1} \mathbf{A}^{(n-1)\mathsf{T}} \times_{n+1} \mathbf{A}^{(n+1)\mathsf{T}} \cdots \times_N \mathbf{A}^{(N)\mathsf{T}}$
            $\mathbf{A}^{(n)} \leftarrow R_n$ leading left singular vectors of $\mathbf{Y}_{(n)}$
        end for
    until fit ceases to improve or maximum iterations exhausted
    $\mathcal{G} \leftarrow \mathcal{X} \times_1 \mathbf{A}^{(1)\mathsf{T}} \times_2 \mathbf{A}^{(2)\mathsf{T}} \cdots \times_N \mathbf{A}^{(N)\mathsf{T}}$
    return $\mathcal{G}, \mathbf{A}^{(1)}, \mathbf{A}^{(2)}, \dots, \mathbf{A}^{(N)}$
end procedure
```

Tucker decomposition for a three-way arrays (HOSVD).

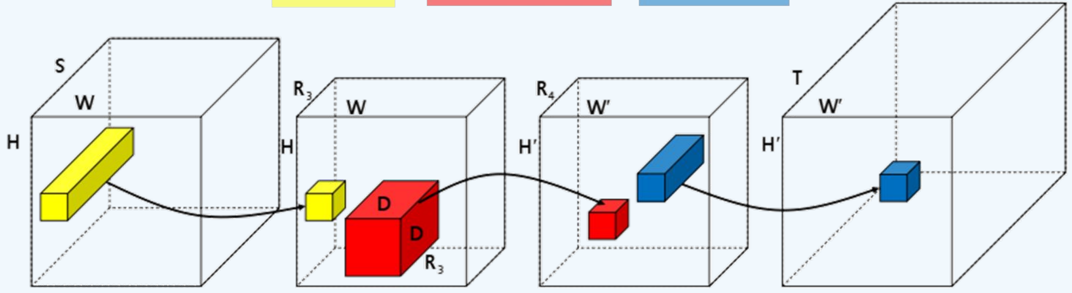
Tucker decomposition for an N th-order tensor (HOOI).

How compressed conv layer works?

Replace each convolutional layer with 3 sequential layers, it's similar to speed-up matvec by multiplication by the three component of SVD sequentially.
there is no need to compress mode-1 and mode-2 as they are already small.



$$E = \frac{D^2 S T H' W'}{S R_3 H W + D^2 R_3 R_4 H' W' + T R_4 H' W'}$$



E can be bounded by:

$$\frac{ST}{R_3 R_4}$$

But how to select these ranks without trail-and-error approach?

Variational Bayesian Matrix Factorization (VBMF)

Assume that we have:

$$V = U + E$$

where $V \in \mathbb{R}^{L \times M}$ - an observation matrix

$U \in \mathbb{R}^{L \times M}$ - the target matrix, is assumed to be low rank $U = BA^T$

$E \in \mathbb{R}^{L \times M}$ - a noise matrix

5. Evaluation results

Model	Applied decomposition	Total memory, MB	FLOPS, x10^6	Amount of multiply-adds, x10^6	Time of inference, s	CPU pred time, s	Accuracy
Resnet18	None	5.41	556.65	1110.00	5.786	0.056	79.27%
	Tucker	5.97	298.70	596.39	3.738	0.032	80.21%
	CP	6.04	275.20	549.37	3.565	0.030	79.21%
Densenet	None	9.88	128.98	257.19	8.403	0.075	58.48%
	Tucker	10.13	73.18	149.39	5.056	0.056	59.01%
	CP	10.02	73.31	145.67	5.532	0.058	58.58%

torchstat — a lightweight neural network analyzer was used for FLOPS and amount of multiply-add evaluation

Conclusions & Future Work

- Tensor decompositions can be useful to reduce amount of FLOPS, time of inference and CPU prediction time.
- Depending on network architecture acceleration ratio for different tensor deconvolution applied is within range **53% to 56%**
- Replacement of single convolution layer by sequence of 3 “decomposed” layers results in minor increment of total memory
- Future work:
  - Finetune the parameters to achieve the best results for compressed network
  - Experiment with more network architectures
  - Tucker works better with larger/more complex network?

Thank you for your attention:D

Yours truly <3

Team Decomposers

- Hai Le: Implementing the model, training/evaluating pipeline, experiments, tucker code
- Abdul Aziz Samra: Implementing the model, Tucker & VBMF decomposition research, tucker code
- Garsiya Evgeniy: Implementing the model, cp code, slide preparation
- Vladimir Kuzmin: Tucker & VBMF research, slide preparation, CP code
- Uyen Vo: CP decomposition research, VBMF research, slide preparation

Questions?

Github:
https://github.com/highly0/CP_Tucker_Decomposition